Festschrift

LNAI 6565

Marcello Balduccini
Tran Cao Son (Eds.)

# Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning

Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday

Springer

Marcello Balduccini   Tran Cao Son (Eds.)

# Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning

Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday

Springer

Series Editors

Randy Goebel, University of Alberta, Edmonton, Canada
Jörg Siekmann, University of Saarland, Saarbrücken, Germany
Wolfgang Wahlster, DFKI and University of Saarland, Saarbrücken, Germany

Volume Editors

Marcello Balduccini
Eastman Kodak Company
Rochester, NY 14650, USA
E-mail: marcello.balduccini@gmail.com

Tran Cao Son
New Mexico State University
Department of Computer Science
Las Cruces, NM 88003, USA
E-mail: tson@cs.nmsu.edu

The illustration appearing on the cover of this book is the work of Daniel Rozenberg (DADARA).

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-20831-7 e-ISBN 978-3-642-20832-4
DOI 10.1007/978-3-642-20832-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011926416

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

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)

Sketch of Michael Gelfond by Veena Mellarkod

# Preface

Michael Gelfond has been an extraordinary mentor, teacher, and colleague for many people in the knowledge representation and reasoning (KR&R), logic programming (LP), and answer set programming (ASP) communities.

Michael's current and former students often like to tell stories about their experience with Michael as a supervisor. These stories invariably begin with meetings in which Michael would review the student's work. The review would usually not go past the first few paragraphs of the work: Michael would simply find too many mistakes, inaccuracies, parts that need clarification! But these stories also invariably end with the students explaining how Michael's mentoring turned them into better researchers and better persons, how he taught them to appreciate the importance of accuracy, of slow, deliberate and careful thinking, of academic and personal integrity.

Michael is not only an amazing researcher, who has changed our areas of research in many ways. He is also an amazing person. Even among people who do not share his views, he is deeply respected for his integrity and straightforwardness. He has a keen interest in people, which makes him very caring and understanding – first of all toward his students. Thanks to Michael's ability of "slow thinking," which he says he learned from his advisor Nikolai Aleksandrovich Shanin – but which Michael has undoubtedly refined on his own – he can think with astonishing lucidity about any topic, be it scientific or non-scientific.

It is because of all these reasons, and so many more that we could never discuss in this brief preface, that we have decided to honor Michael on the occasion of his $65^{th}$ birthday with a collection of papers written by his closest friends and colleagues. Several of these papers were presented during the Symposium on Constructive Mathematics in Computer Science, held in Lexington, KY, during October 25–26, 2010.

We would like to take this opportunity to thank all the authors, who worked so hard to turn this collection into a reality, and to the colleagues who acted as peer-reviewers, whose names are listed at the end of the preface. We thank Veena Mellarkod for drawing the beautiful sketch of Michael that opens this volume. Special thanks also go to Victor Marek and Mirek Truszczynski for taking care of the local organization, and to Lara and Greg Gelfond for convincing Michael to spend a few extra days in Lexington after the end of NonMon@30, thus giving us the opportunity to surprise him with the symposium.

Dear Michael, this volume is a testimony to how you have touched the lives of many people around the whole world. We wish to be able to celebrate many such anniversaries in the years to come!

October 2010

Marcello Balduccini
Tran Cao Son

# Organization

## Reviewers

The contributions that appear in this volume have been peer-reviewed by:

| | |
|---|---|
| Marcello Balduccini | Eastman Kodak Company, USA |
| Chitta Baral | Arizona State University, USA |
| Gerhard Brewka | University of Leipzig, Germany |
| Pedro Cabalar | Corunna University, Spain |
| Marina De Vos | University of Bath, UK |
| Agostino Dovier | Università degli Studi di Udine, Italy |
| Esra Erdem | Sabanci University, Turkey |
| Michael Fink | Technische Universität Wien, Austria |
| Alfredo Gabaldon | New University of Lisbon, Portugal |
| Martin Gebser | Universität Potsdam, Germany |
| Roland Kaminski | Universität Potsdam, Germany |
| Vladimir Lifschitz | University of Texas at Austin, USA |
| Fangzhen Lin | Hong Kong University of Science and Technology, China |
| Jorge Lobo | IBM J. Watson Research Center, Hawthorne, NY USA |
| Luís Moniz Pereira | Universidade Nova de Lisboa, Portugal |
| Alexandre Miguel Pinto | Universidade Nova de Lisboa, Portugal |
| Enrico Pontelli | New Mexico State University |
| Francesco Ricca | Università della Calabria, Italy |
| Nelson Rushton | Texas Tech University, USA |
| Chiaki Sakama | Wakayama University, Japan |
| Tran Cao Son | New Mexico State University |
| Francesca Toni | Imperial College London, UK |

Michael Gelfond and Colleagues
Picture taken at "Symposium on Constructive Mathematics in Computer Science" held in honor of Michael Gelfond's 65th birthday (Oct 25-26 2010)

Michael Gelfond and Colleagues
Picture taken at "Theories of Logic Programming, Non-Monotonic Reasoning, and their Application to Reasoning about Actions: a Symposium in Honor of Michael Gelfond's 50th Birthday" (Nov 5 1995)

# Table of Contents

# Homage to Michael Gelfond on His 65th Birthday

Jack Minker

Department of Computer Science
and
University of Maryland Institute for Advanced Computer Studies
College Park, MD 20742
minker@cs.umd.edu
http://www.cs.umd.edu/~minker

**Abstract.** Michael Gelfond is one of the world leading scientists in the field of logic programming and nonmonotonic reasoning. This essay covers several aspects of Michael's personal life, starting from his birth in the USSR, through his experiences in the USSR up to the time he emigrated to the United States (U.S.). This is followed by his first experiences in the U.S.: how he became involved in logic programming and nonmonotonic reasoning and some of his major scientific achievements. Michael is a warm, generous person, and I discuss his impact on some colleagues and students. In the concluding section, I observe that starting his career with impediments in the FSU, he overcame them to become one of the top computer scientists in logic programming and nonmonotonic reasoning.

## 1 Introduction

The editors to this Festschrift invited me to contribute an essay on Michael Gelfond and his personal life. They had heard that I had helped several scientists to emigrate from the Former Soviet Union (FSU) and asked whether or not this was true. Although I did help many Soviet scientists to emigrate, I never helped Michael. I wrote several articles in the Communications of the Association for Computing Machinery, starting in 1981, listing all computer scientists in the world, whose human rights were violated. Most of the computer scientists were from the Soviet Union. Michael never appeared on my lists since he emigrated in 1977, before I wrote my reports. Although, in general, I know a great deal about the reasons Soviet scientists wished to emigrate from the FSU, I did not know Michael's particular situation. Thanks to those cited in the Acknowledgment section, I was able to obtain the information needed for this essay.

## 2 Michael Gelfond in the USSR (1945-1977)

Michael (Misha) Gelfond was born a few months after the end of World War II on November 7, 1945 in the city of Leningrad (now St. Petersburg) in the Former Union of Soviet Socialist Republics (USSR) to Jewish parents, Ruth and Gregory Gelfond. Michael was brought up in Leningrad, and went to Leningrad State University, where he received an M.S. in Mathematics in 1968. He met his friend and colleague Vladimir

Lifschitz in 1963, when they both enrolled at Leningrad State University and took many courses together. At the time they enrolled, it was difficult for Jews to get into the best universities, such as Leningrad, because of anti-Semitism. However, because of their excellent scholarship they were accepted and completed their degrees.

Michael attended the famous Steklov Mathematical Institute, where he studied for a Ph.D. in Mathematics. As told to me by Vladimir Lifschitz, both he and Michael obtained their Kandidat degrees (Ph.D. equivalent) from the Steklov Institute without being enrolled. In the USSR, one could take several exams and write and defend a dissertation to obtain the degree and that is what they did. Although not an official student at the Steklov Institute, Michael was able to study with two outstanding researchers in logic, Sergey Maslov and Nikolai Alexandrovich Shanin who were at the Institute. He completed his Ph.D. thesis, "Classes of formulae of classical analysis compatible with constructive interpretation," under the direction of Shanin and received his degree in 1974.

Michael met Larisa (Lara) Kozinetz, his wife-to-be, in a desert near the sea, as might have happened in biblical days. The story, as told to me by Lara, is as follows.

> It was standard practice in the USSR to send non-workers (non manual labor) to work on collective farms (seasonal) or other projects, such as construction work. Students were the first to go (they provided free labor), whether or not they liked the idea. In her 2nd year at the University, Lara had a choice: to work on a collective farm in the fall or on construction work in the summer. Refusal to do either would mean that she would not be able to have a room in a dormitory. Renting in a private home was close to impossible.
>
> Her choice was "railroad construction", which was, at least, something new and far away. The project was in Kazakhstan, by the Caspian Sea. It is a desert, with very rough conditions and absolutely nothing around for miles. Michael made the same choice that year. They met when Michael and a friend were visiting the camp. It was love at first sight.

Several months after he received his M.S. degree, Michael married Larisa on February 12, 1969. Lara was also a mathematician who received her M.S. degree from Leningrad State University, and then worked as an engineer/chief engineer at a laboratory that studied turbine blades, and at a geographical image processing laboratory where she made topological maps from satellite images. Their first child, Yulia (now Yulia Gelfond Kahl), was born in Leningrad on January 8, 1970. A second child, Gregory Gelfond (named after Michael's father), was born in San Jose, California on October 15, 1979.

Both Lara and Michael's families had lived in the Greater Russian Empire for many generations. Michael's family lived in today's Latvia. Lara's family was from Russia and Ukraine. Their parents served valiantly in the Russian military during World War II. Michael's father served as a naval commander in the battles of Sevastopol and Odessa (two major battles on the Russian front on the Black Sea). He also served in the Japanese campaign as well as in naval military intelligence. His mother served as a naval officer in naval intelligence as an analyst and a translator. Lara's father served as a tank battalion commander on the western front, and her mother served as a medical doctor operating in a Mobile Army Surgical Hospital unit on the western front. Her parents helped to liberate the concentration camp at Majdanek on the outskirts of Lublin, Poland. A large number of Michael's relatives were murdered during the Nazi invasion of the USSR.

To understand the situation faced by Jews in the USSR, I quote a part of an unpublished paper by Ambassador Richard Schifter written in honor of my $65^{th}$ birthday in 1992.

> Russia has had a long tradition of anti-Semitism. In the early part of this Century, when most other European countries had left the most overt aspects of anti-Jewish discrimination behind them, the Empire of the Czar still held on to them. Equal rights for Jews came finally with the democratic revolution in February 1917. When the Bolsheviks take over in October, they continued this non-discrimination policy.
>
> But this period of equal rights lasted for not much more than about fifteen years. As Stalin consolidated his hold on the country, his own-anti-Semitic feelings became increasingly more evident. A large number of Jews could be found among the victims of the purges of the Thirties. Thereafter, gradually, Jews were barred from entering on a career in certain sensitive job classifications, such as the secret police and the foreign service. Glass ceilings beyond which Jews who were already in the system could not rise were established in these job categories and in a variety of other fields, such as the Communist Party and the military forces. After World War II, Stalin set about wiping out traditional Jewish culture, killing a number of writers and actors. The so-called doctors' plot of 1952 indicated that he had additional anti-Jewish measures in mind.
>
> Stalin's successors were not men who equaled him in brutality. But they, too, were committed to moving the Soviet Union's Jewish population to the sidelines. By the 1960's they had a policy in place under which the glass ceiling against the promotion of Jews beyond a certain level was extended throughout the entire system of governmental institutions. And then, discriminatory policies were initiated in education.

During his university days, Michael spoke freely and openly and his comments did not always follow the USSR party line. He also attended a seminar called "Theory of Systems," organized by Sergey Maslov. A variety of different intellectuals such as scientists, priests and writers, lectured on different subjects. According to Larisa, the "Maslov seminar was a very important part of our lives. We had a chance to meet a lot of absolutely remarkable people and to hear exceptional presentations on different subjects - science, literature, history, religion, etc. It helped us to be "alive" and mind growing, and be informed on today's events (there were no information/news, just plain propaganda every minute of the day - radio, TV, newspapers - all the same and absolutely unreadable for anyone who can think)." In addition there were also discussions of some underground literature, and unofficial news. As was common in the USSR, there were always individuals who reported to government authorities and informed the authorities of the discussions. As a consequence, attendance at the seminar was considered a form of unconventional behavior not approved by the KGB (along with attending church or synagogue services, studying non-Marxist philosophy, telling jokes about Brezhnev, socializing with foreigners, and many others). Individuals not approved by the KGB were "black listed." When they applied for a job, their names were checked against this list. "Dissidents" in the USSR referred to individuals who generally wanted to change the country or openly protested government actions. One such dissident was the world renowned nuclear physicist Andre Sakharov. Unlike Sakharov, Michael and Lara did not consider themselves to be dissidents–they were intellectuals who wanted to discuss the pros and cons of current issues freely and openly. However, Lara is convinced that many of the attendees of the Maslov seminar were viewed as dissidents and some may really have been dissidents. However, merely because one sometimes may have openly disagreed with the government, was sufficient, either to place them on a "black list" or to add this information to the investigative files kept by the KGB.

Following his Ph.D., Michael found it difficult to obtain a job commensurate with his talents either because of anti-Semitism or because he was on a black list. However, for a while, he was able to obtain a job doing programming; then found a job teaching in a high school for gifted students; at a regular high school; and for a short while at a financial/economics institute. As related to me by Vladimir, Michael's attitude towards teaching high-school students was to be open to students about his values and views, and he spent a lot of time discussing these "dangerous" topics with them. This was rare among high school teachers: their job was to prepare their students to the life of loyal Soviet citizens, enthusiastic about Communist ideology! Michael's unorthodox attitude towards his teenage students was probably not looked upon favorably in the eyes of the KGB. He was unable to retain these jobs and became unemployed. It was fortunate that anonymous individuals or organizations sent the family packages. They were able to sell some of the packages to survive.

The USSR had a policy that if an individual was not working, he was deemed a "parasite" (a drain on society) and was sent to a labor camp. To avoid being a "parasite", Michael had an opportunity to register as a baby sitter for some of his friends and also as a tutor. Although being a tutor was not sufficient to avoid being labeled a parasite, being registered with the state as a baby sitter kept him from being considered a parasite.

During the period he was not working, he began to consider his future in the USSR and the future of his daughter. He could not work up to his level of competence because of anti-Semitism and because he was black listed and was unable to find a job; his daughter, was subjected to anti-Semitic statements while in kindergarten and when she grew up she would not have the opportunity to live up to her potential. However, according to Vladimir Lifschitz the main reason that he and Michael had for emigrating was, "Unwillingness to live under totalitarian control ... An intelligent and sensitive person in the USSR had two options: either to do many things that were against his conscience (and abstain from doing things that he felt he must do) or to be sent, perhaps for years, to prison, to a labor camp, or to a psychiatric hospital."

According to Michael's son Gregory, his parents' decision to emigrate from the USSR came after they were visited by a pair of KGB agents and were told that he had two options: leave for the "West" (i.e. out of the country) or to the "East" (be sent to a labor camp). This was a typical Catch 22 situation. Not able to find a job because the government discriminated against Jews, and because it was known he openly spoke out against government policies, he would be subject to being sent to a labor camp. Taking the implied advice of the KGB agents, the Gelfonds immediately decided to apply for an exit visa. The visit from the KGB may have been caused partially by his attendance in the Maslov Seminar and by his openness with his high school students.

Neither Michael nor Lara could figure out what exactly prompted the KGB to pay them this critical visit. At the time Michael's conjecture was: when an investigative surveillance file reaches certain level - they need to do something. Since they started to collect information during his university years it probably was full and ready for an action. There were some events before this visit - a search warrant for underground literature at a friend's house. They were warned in advance and were able to load a suit case with all the relevant books and papers and take it to the "most neutral" friend that they had before the KGB visit so everything was "clean" when the KGB arrived. Lara believes that all the factors specified in this paragraph played a role leading to the KGB visit.

However, before they could apply for an exit visa, Larisa, who was still working, had to obtain an official statement from her employer stating that, in effect, she did not "owe the company" anything, such as overdue books, or checked out equipment. To obtain such a statement, she was forced to resign her job. Once that was done, they applied for their exit visas in the summer of 1977. Many Soviet Jews who wished to obtain exit visas were denied on their first attempts and could only emigrate after being refused over many years — they were referred to as "refuseniks". Michael and Larisa received their exit visas within six months, possibly with the aid of the KGB who may have wanted to rid the country of those who they deemed a thorn in their sides. Michael and Lara were fortunate that they never became refuseniks. They emigrated on December 28, 1977.

## 3 Michael and Larisa Gelfond's Early Years in the United States (1978-1980)

In 1972 there was a sharp increase of Soviet Jews who emigrated from the USSR, primarily because of anti-Semitism. The Brezhnev regime, to limit the "brain drain", imposed a prohibitively expensive exit tax on educated Jews who wanted to leave. The backlash on this policy by Western governments forced the Soviets to relent. By the time the Gelfonds applied, there was no exit tax. But, in its place, they had to "refuse/denounce" Soviet citizenship and pay 500 rubles per adult (a total of 1000 rubles) for the process. That was an enormous amount of money, but the Gelfonds had saved some kopeks, were fortunate to have won some money in a lottery and Michael's mother made up the difference.

Neither Gelfond knew much about the United States or Israel. However, some of their friends were already in the U.S. and they had neither family nor contacts in Israel. They were also concerned that their marriage would not be recognized in Israel since Lara was not Jewish and they were unaware of what it would take to be considered Jewish or at least to be married legally. The only foreign language Michael had studied in school was German. They believed it was possible, although difficult, to learn English, but no one on Michael's side of the family had known Hebrew for generations. For these reasons, they decided to try to go to the U.S. The USSR only allowed émigrés to exchange about a total of $200 when they left the country. Their journey to the U.S. took them through several places. The trip from Leningrad to Vienna, Austria was paid for from funds they were not allowed to take out of the country. Since they were short of funds when they arrived in Vienna, they applied to The Hebrew Immigrant Aid Society

(HIAS) for support. HIAS was well-known for its lifesaving services to world Jewry to rescue, reunite and resettle families over many generations. HIAS paid for their tickets for the rest of their journey and gave them money for food. The funding was an interest free loan, where they signed a promise to return it at some point. Naturally, the Gelfonds repaid the funds when they could afford it. After 10 days, they went from Vienna to Rome, stayed in Rome for about 3 months and then went to New York City.

In Rome, they applied to the U.S. on the ground of being refugees. They had been in contact with the Jewish community in San Jose, California, which guaranteed to the U.S. government that they would not be "public charges" and promised to support them until they could find jobs. In New York, they received social security numbers and permission to work. They stayed in New York for about a week with friends since the Jewish community in San Jose was busy with the Passover holidays. The symbolism of the holy day of Passover being a commemoration of the exodus of Jews from slavery in Egypt, took on a double meaning for the Gelfond family as their exodus and that of other Jews from the Soviet Union.

Their primary financial support in the U.S. was from the Jewish community in San Jose. They paid the rent and utility cost for the apartment in a modest housing complex. By Russian standards, their modest housing looked very good to them. They received about $180.00 a month for food. The Jewish community also provided them with the various essentials - beds, a dining table and other furniture. There were no luxury items, no new items, but there was much thought and care put into it and they felt very welcomed. There were even some toys for Yulia (Gregory had not been born yet). This kindness is appreciated to this day. To come from the "mother country" which did not want them, to an unknown place, and to feel so accepted by people who did not know them, made a lasting impression on them. Both Michael and Larisa also learned to drive in the U.S. and for the $200.00 they had brought with them from the USSR, they bought their first car, as it was impossible to travel to work in California without a car.

Although Michael had some private lessons in English in the USSR, he essentially knew no English. He learned by reading with a dictionary and listening to TV. But in some programs, such as the children's show, Mister Roger's Neighborhood, they spoke too fast to permit him to recognize words he already knew.

With limited ability in English, finding a job was not easy. He told his son many amusing stories common to many immigrants who were not fluent in English. At one interview he was asked a question along the following lines, "What happens when you hit the space bar on a keyboard?" He thought about it and to the amazement of the interviewer, started to try to describe how the signal travels through the bus etc. It never occurred to Michael to simply say that a "whitespace" comes up.

Six months after coming to San Jose, he obtained a job at Measurex, a company based in Cupertino, California. Measurex developed computer control systems for industry, primarily the paper-making industry. He was hired as an applications programmer. Although he had experience in programming in the USSR, it was a limited ability. Working with his coworkers, he learned how to systematically debug a program and other tools of the programming trade. After a while, one of his friends who worked for a company in San Jose, called Mohawk, convinced him to change jobs. Unfortunately, he did not realize that his insurance at the new company would not pay for hospitalization when his son was born two weeks later, since the insurance company at the new company did not pay for "pre-existing conditions".

## 4 Michael Gelfond's Scientific Contributions

In 1979, Vladimir Lifschitz was hired in his first tenure track position as an Assistant Professor in the Mathematics Department at the University of Texas at El Paso (UTEP). A short while after that he told a colleague at UTEP about Michael and recommended that they interview him for a faculty position. He was interviewed and accepted a job as an Assistant Professor in the Mathematics Department. As happened at many universities at that time, a short while after Michael arrived, UTEP turned its interdepartmental computer science program into a department, and Michael and Vladimir were among its first faculty members. During that time Michael, Vladimir and a friend became involved with writing a compiler from scratch.

Michael started his work in logic programming and nonmonotonic reasoning after he came to the University of Texas at El Paso. It is interesting to note how he came to work in these fields. As related by Michael in his paper [Gelfond, 2006], he wrote that in the early eighties he became interested in knowledge representation and reasoning.

> At that time Vladimir Lifschitz and I were involved in the development of the computer science program at the University of Texas at El Paso, and Vladimir decided to go to a Stanford Summer School to learn more about important CS topics. There he listened to John McCarthy's course on AI and came back with fascinated stories about defaults, non-monotonic logics and circumscription. The next year was my turn. I took the same course together with the course on Logic Programming taught by Kenneth Bowen. In that class I wrote my first Prolog program and got introduced to Prolog's treatment of negation.

It seems that everything in AI and nonmonotonic reasoning originates with John McCarthy.

A few years after Michael came to UTEP, Halina and Teodor Przymusinski were hired. Although they only worked together for a short period of time at the same university in logic programming and nonmonotonic reasoning, Michael, Vladimir, Halina and Teodor, became one of the strongest groups in the country in these fields. According to Halina,

> Michael's interest in non-monotonic reasoning and its formalizations came from his cooperation with Vladimir. When I joined the CS department at UTEP, Michael was already working on the relationship between the CWA and circumscription. He asked me to work with him on that topic. At that time our knowledge of the literature on the subject was quite limited. It was therefore not surprising that the first paper that Michael and I were to present at a conference turned out to contain results already published by Jack Minker. We both felt awful about this situation but Michael immediately thought about generalization of the original results to the case of prioritized circumscription. As a result our honor was saved and the tough lesson was learned.

I was always impressed that Halina and Michael, two immigrants new to the field of logic programming and the English language, were able independently to develop results on the Generalized Closed World Assumption (GCWA) that I had developed earlier. Both Halina and Michael went on to develop significant results following this impressive start.

It is not possible in the space allowed for this essay to do justice to all of Michael's scientific accomplishments. I shall highlight just a few: Stable Models/Answer Set Programming (with Lifschitz) and the introduction of logical negation into logic programs (with Lifschitz); knowledge bases (with Chitta Baral) and others; dynamic domains and applications (with Lifschitz and many others); and synthesizing Prolog with probabilistic reasoning.

The work that Michael and Vladimir Lifschitz did on stable models, which has been renamed answer set programming (ASP), is perhaps their major achievement to date. Their paper, "The Stable Model Semantics for Logic Programs" [Gelfond and Lifschitz, 1988], received the Most Influential Paper in 20 Years Award from the Association for Logic Programming in 2004. It is interesting to note that their first attempt to publish the paper was rejected from a conference. As noted by David Pearce in his excellent 2008 survey article [Pearce, 2008], they came to stable models through two different directions, "... Vladimir who was most closely associated with minimal model reasoning through his work on circumscription, while Michael Gelfond had noticed an interesting connection between logic programs and autoepistemic reasoning, where models are defined via a fixpoint construction." Their extension to include logical negation together with default negation was another significant contribution [Gelfond and Lifschitz, 1990]. Additionally, they extended stable models to apply to disjunctive theories [Gelfond and Lifschitz, 1991]— this paper, too, was first rejected from a conference. ASP has become the standard in the field. There has been an explosion of papers extending the work. Contributing to this explosion is the implementation and commercialization of logic programming systems that incorporate ASP that are now available and handle large databases and rules. Two such systems are Smodels [Niemelä and Simons, 1997]; DLV [Eiter et al., 2000]. CLASP [Gebser et al., 2007] is another system of interest. It is possibly the best current answer set solver. Although it is not a commercial product, it is available for free use in the public domain. All three systems have been expanded to include many features.

Another major contribution is Michael and Chitta Baral's survey paper on knowledge base systems [Baral and Gelfond, 1994]. It demonstrated conclusively how logic programming and nonmonotonic reasoning, using ASP, represented and reasoned about complex knowledge representation problems.

Michael has developed an impressive series of papers on dynamic domains which deal with practical problems that arise in real world applications. These domains are concerned with problems such as handling multiple agents, inter-agent communication, changing environments, exogenous actions and time that must be accounted for. He started this work with Vladimir Lifschitz [Gelfond and Lifschitz, 1992]. This paper was followed by the journal version [Gelfond and Lifschitz, 1993]. The paper was among the papers in the early 1990s that changed the direction of research in situation calculus and reasoning about actions. While in the past, research in this field was example based, this was one of the pioneer papers which brought a systematic approach to research in this field. Later, together with Chitta Baral, Michael defined the encoding of the same knowledge into ASP, and provided definitions of algorithms for planning and explanation of observations [Baral and Gelfond, 2000]. In his paper with Marcello Balduccini [Balduccini and Gelfond, 2001], they described an ASP-based architecture for intelligent agents capable of reasoning about and acting in changing environments. The design is based upon a description of the domain that is shared by all of the reasoning modules. The generation and execution of plans are interleaved with detecting, interpreting, and recovering from, unexpected observations. This is on-going work that has been expanded to other topics in dynamic domains.

Michael's work in dynamic systems led him to implement, the first practical industrial-sized application of ASP-based planning for NASA on the Reaction Control System of the Space Shuttle. The work was, however, not used in a space shuttle, but in simulation studies. His paper with Marcello Balduccini and Monica Nogueira, covers this work [Balduccini, Gelfond, and Nogueira, 2006].

Among Michael's current research is work on attempting to synthesize Prolog with probabilistic reasoning (what he calls P-log). His work with Chitta Baral and Nelson Rushton [Baral, Gelfond, and Rushton, 2009] was influenced by work of Judea Pearl [Pearl, 1988]. The work combines the ASP part of a P-log program to describe possible beliefs, while causal Bayes nets serve as a probabilistic foundation and allows knowledge engineers to quantify the degrees of these beliefs. It is important research, as a P-log program combines the logical knowledge representation power of answer set programming with causal probabilistic reasoning; the former, perhaps for the first time, allows non-monotonic specification of the set of possible worlds.

For his scientific achievements, in addition to his award in 2004 for the Most Influential Paper in 20 Years from the Association for Logic Programming, he was elected a Fellow of the American Association for Artificial Intelligence and a member of the European Academy of Sciences.

From the above brief description of Michael's contributions, it is clear that he is among the world leading researchers in our field.

## 5 Michael Gelfond the Person

Michael is not only an outstanding scientist, but also a warm, caring, nurturing individual. He inspires others with whom he works. He always has a smile on his face and kind words for those he knows or meets. As Halina Przymusinska wrote to me, "It was not just Michael a scientist that grabbed my interest in his research but it was primarily Michael—a wonderful human being and a friend who provided me and Teodor with a fascinating and very rewarding adventure—the study of non-monotonic reasoning and Logic Programming. We will be grateful to him for that forever."

Halina adds, "Michael continues to share his enthusiasm and knowledge and provides encouragement to many young (and not so young) computer scientists. Good scientists are rare, but people like Michael are truly exceptional even among good scientists." To Halina's comment, Chitta Baral wrote, "His mentoring and nurturing (of young researchers) in the two fields of answer set programming and reasoning about actions was an important reason why so many young researchers made their research careers in those two fields. He co-founded TAG (Texas Action Group) with Vladimir to have on-line discussion on those two topics, and it now has members from all over the world."

I share the comments made by Halina and Chitta. On a personal level, I have always been impressed and thankful that one of my outstanding Ph.D. students, Chitta Baral, was taken under Michael's wings after he left the University of Maryland. Chitta blossomed further to become one of the world leading researchers in our field. Michael contributed to the well-being of my scientific family. With Michael's son, Gregory, working for his Ph.D. with Chitta, I look forward to welcoming Grisha as part of my scientific family.

## 6 Concluding Remark

It is ironic that the Gelfonds, forced to leave the 'paradise' of the Former Soviet Union (FSU) after their families had lived there and in Greater Russia for several generations; where Michael and Lara's parents served their country valiantly during World War II; where Michael had been born on November 7th, the date that used to be celebrated in Russia as the "October revolution" (in 1917); where both Lara and Michael excelled as students and had graduate degrees—Michael a Ph.D. and Lara an M.S.; and yet the FSU did not want and could not take advantage of these productive scientists. Michael, unwelcome in the country of his birth, was then able to thrive in a welcoming, tolerant country, open to all religions, races and creeds that accepted him and his family as full citizens whose religion was Judaism, and gave them an opportunity to use their science in the United States. Despite having a difficult life in the FSU, Michael succeeded in this land of freedom, where one need not be afraid to speak one's mind openly, and became an internationally renowned researcher in logic programming, nonmonotonic reasoning, knowledge representation and reasoning, and artificial intelligence. It is a tribute to Michael that he was able to overcome all impediments to his career.

Mazal Tov, Misha, it is an honor to have you as a friend and colleague, and to celebrate you on your $65^{th}$ birthday. May you have many more productive years and may you and your family continue to thrive. Zolst leben biz ein hundert und zvanzig (May you live until 120)!

## Acknowledgments

I especially want to thank the Gelfond family (of course Michael was not involved) and Vladimir Lifschitz for their assistance. I learned a great deal of information from the Gelfonds about their lives in the USSR and their early years following their emigration. Vladimir provided me information about his and Michael's challenges in their education in the USSR and made several constructive suggestions about the entire essay. I am also grateful to Lara for the material she provided on how she and Michael met and to the entire Gelfond family who provided valuable information and reviewed the entire essay. I greatly appreciate Halina and Teodor Przymusinski's contribution with information about Michael the person and their early work with him at UTEP. I wish to thank Son

Tran and Marcello Balduccini for inviting me to write this essay. Together with Chitta Baral, they provided valuable suggestions about Michael's research and some aspects of his life.

## References

[Balduccini and Gelfond, 2001] Balduccini, M., Gelfond, M.: The Autonomous Agent Architecture. The Newsletter of the Association of Logic Programming 23 (2010)

[Balduccini, Gelfond, and Nogueira, 2006] Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Ann. Math. Artif. Intell. 47(1-2), 183–219 (2006)

[Baral and Gelfond, 1994] Baral, C., Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming 19/20, 73–148 (1994)

[Baral, Gelfond, and Rushton, 2009] Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theory and Practice of Logic Programming 9(1), 57–144 (2009)

[Baral and Gelfond, 2000] Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 257–279. Kluwer Academic Publishers, Dordrecht (2000)

[Eiter et al., 2000] Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem- Solving Using The DLV System. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 79–103. Kluwer Academic Publishers, Boston (2000)

[Gebser et al., 2007] 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)

[Gelfond, 2006] Gelfond, M.: Answer Sets in KR: a Personal Perspective. The Association for Logic Programming Newsletter 23 (2006)

[Gelfond and Lifschitz, 1988] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programs. In: Bowen, K., Kowalski, R. (eds.) Proc. 5th International Conference on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)

[Gelfond and Lifschitz, 1990] Gelfond, M., Lifschitz, V.: Logic Programs with Classical Negation. In: Warren, D., Szeredi, P. (eds.) Proceedings of ICLP 1990, pp. 579–597. MIT Press, Cambridge (1990)

[Gelfond and Lifschitz, 1991] Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)

[Gelfond and Lifschitz, 1992] Gelfond, M., Lifschitz, V.: Representing actions in extended logic programs. In: Apt, K. (ed.) Joint International Conference and Symposium on Logic Programming, pp. 559–573. MIT Press, Cambridge (1992)

[Gelfond and Lifschitz, 1993] Gelfond, M., Lifschitz, V.: Representing actions and change by logic programs. Journal of Logic Programming 17(2,3,4), 301–323 (1993)

[Niemelä and Simons, 1997] Niemelä, I., Simons, P.: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In: Proceedings of the 4th International Conference on on Logic Programming and Nonmonotonic Reasoning, pp. 421–430. Springer, Heidelberg (2003)

[Pearce, 2008] Pearce, D.: Sixty years of Stable Models. In: Proceedings of the International Conference on Logic Programming, vol. 52 (2008)

[Pearl, 1988] Pearl, J.: Probabilistic reasoning in intelligent systems: networks of plausible Inference. Morgan Kaufmann, San Francisco (1988)

# Answer Set Programming's Contributions to Classical Logic

## An Analysis of ASP Methodology

Marc Denecker, Joost Vennekens, Hanne Vlaeminck,
Johan Wittocx, and Maurice Bruynooghe

Department of Computer Science, K.U. Leuven

**Abstract.** Much research in logic programming and non-monotonic reasoning originates from dissatisfaction with classical logic as a knowledge representation language, and with classical deduction as a mode for automated reasoning. Discarding these classical roots has generated many interesting and fruitful ideas. However, to ensure the lasting impact of the results that have been achieved, it is important that they should not remain disconnected from their classical roots. Ultimately, a clear picture should emerge of what the achievements of answer set programming mean in the context of classical logic, so that they may be given their proper place in the canon of science. In this paper, a look at different aspects of ASP, in an effort to identify precisely the limitations of classical logic that they exposed and investigate how the ASP approaches can be transferred back to the classical setting. Among the issues we thus address are the closed world assumption, "classical" and default negation, default reasoning with exceptions, definitions, lp-functions and the interpolation technique and the strong introspection operator. We investigate the ASP-methodology to encode knowledge using these language constructs and come across a dichotomy in the ASP-methodology.

## 1 Introduction

In the late fifties, a thesis of logic-based AI was that classical first order logic (FO) and deductive theorem provers for FO would play a key role in building intelligent systems [21]. In the late sixties and early seventies, the disappointing results of this project led a number of new research directions. In logic-based AI, two areas that arose as a reaction were logic programming (LP) and non-monotonic reasoning. In logic programming, the idea was that FO was too expressive and had to be restricted to Horn clause logic to allow efficient theorem proving [16]. The origin of non-monotonic reasoning lies in the dissatisfaction about the use of FO for representing common sense knowledge. To overcome the limitations of FO, new logics and logic principles were studied. Examples are the Closed World Assumption [26], Circumscription [22] and Default logic [28]. Although having completely different points of departure, both areas converged in the late eighties and early nineties mainly under the influence of Michael Gelfond and Vladimir Lifschitz who sought to explain logic programming with negation as failure using non-monotonic reasoning principles [12], and turn it into an expressive knowledge representation language [13]. Michael and Vladimir thus became founding fathers of the field of Answer Set Programming (ASP) [1], which has developed into a flourishing, dynamic field with many applications and powerful systems [9,8].

Currently, there is undeniably a large conceptual gap between ASP and classical logic. As a long-term situation, this is undesirable. First of all, despite its limitations, FO provides a set of essential connectives, together with a well-understood methodology for their use, which makes it an invaluable kernel for KR languages. Second, the conceptual gap also hampers recognition of ASP's contributions to AI and computational logic, thus compromising the use of the ASP languages and tools in other fields.

This paper will try to identify limitations of classical logic exposed by Gelfond's work and investigate how the ASP solutions can be transferred back into classical logic. This boils down to identifying important forms of knowledge, to study how they are represented in ASP and to investigate how they can be represented in (extensions of) FO. Thus, this paper also studies the methodology of ASP, and compares it with the methodology of FO. It identifies also key contributions of the ASP language and integrates them with FO. Among the issues we thus address are the closed world assumption, "classical" and default negation, default reasoning with exceptions, definitions, lp-functions and the interpolation technique and the strong introspection operator.

Integrating classical logic and ASP is not a trivial task, as research on rule languages for the semantic web is now also discovering [25]. Expanding the stable semantics to the syntax of FO as in [10] is obviously *redefining* FO, not *integrating* ASP and FO. Among subtle problems and tricky mismatches, one glaring difference clearly stands out: ASP uses the *answer set* as its basic semantic construct, whereas FO of course uses *interpretations* (or *structures*). There is a profound difference between the two concepts. As defined in [13], an answer set is a set of literals that represents a possible state of belief that a rational agent might derive from the logic program. In other words, an answer set is *subjective*; it does not talk directly about the world itself, but only about the beliefs of an agent about this world. This conception of answer sets is not a philosophical afterthought, but is tightly connected with the view of negation as failure as a modal, epistemic or default operator, and therefore truly lies at the heart of ASP. By contrast, FO has a *possible world* semantics, in which each model of a theory represents a state of the world that is possible according to this theory. In other words, FO is concerned with *objective* forms of knowledge, that do not involve propositional attitudes such as the belief or knowledge of an epistemic agent, etc. Modal extensions of classical logic that can express such forms of subjective knowledge typically use *Kripke structures* (or the special case of *sets of possible worlds*) as a semantic object.

Answer sets fall between classical interpretations and Kripke structures: an answer set represents a belief state, and thus differs from an interpretation which describes a possible state of the world, but it does so in a less complex and less accurate way than a Kripke structure or sets of possible worlds, since it only describes what literals are believed rather than arbitrary formulas. This mismatch between ASP's basic semantic constructs and the semantic constructs of FO and its modal extensions seriously complicates the task of formally comparing ASP with FO, and it will be the focus of much of this paper. We will come across a dichotomy in the use of ASP programs: in some, an answer set represents the belief state of an existing rational agent, in many others the programmer carefully crafts the mind of an imaginary agent so that solutions of a problem can be extracted from the belief sets of that agent. This dichotomy stands orthogonal to the (many) ways in which stable semantics can be formalised [18]. We will show that this leads to two different methodologies and, de facto, to two different KR languages, as pointed out long time ago by Gelfond and Lifschitz[13] and, from a different angle, in [6].

*Overview.* In Section 2 we briefly recall the relevant basics of answer set programming and first order logic. In Section 3, we discuss the basic semantic principles of knowledge representation in ASP and contrast them with those of FO. We distinguish two methodologies for knowledge representation in ASP. In Section 4 we introduce FO(ID), the extension of first order logic with inductive definitions. In Section 5 we take a close look at different forms of objective knowledge, how they are represented in ASP and what is their representation in FO(ID). We discuss unique names and domain closure assumptions, different forms of closed world assumptions, the representation of definitions in ASP and defaults. In Section 6, we extend FO with a simple epistemic operator with which we can simulate negation as failure and strong negation.

## 2 Preliminaries of ASP and FO

We briefly recall the basic concepts of Answer Set Programming and of classical logic. A vocabulary $\Sigma$ consists of a set $\Sigma_{Fun}$ of constant and function symbols and a set $\Sigma_{Pred}$ of predicate symbols. An answer set program $P$ over vocabulary $\Sigma$ is a collection of rules $\mathtt{l}\,\mathtt{:-}\,\mathtt{l_1}, \ldots, \mathtt{l_m}, \mathtt{not}\ \mathtt{l_{m+1}}, \ldots, \mathtt{not}\ \mathtt{l_n}$. and constraints: $\mathtt{:-}\,\mathtt{l_1}, \ldots, \mathtt{l_m}, \mathtt{not}\ \mathtt{l_{m+1}}, \ldots, \mathtt{not}\ \mathtt{l_n}$. where each $\mathtt{l}, \mathtt{l_i}$ is an atom $\mathtt{P}(\mathtt{t_1}, \ldots, \mathtt{t_k})$ or a strong negation literal $\neg\mathtt{P}(\mathtt{t_1}, \ldots, \mathtt{t_k})$. A constraint is seen as a special rule defining $\mathtt{F}$ as a new symbol: $\mathtt{F}\,\mathtt{:-}\,\mathtt{not}\ \mathtt{F}, \mathtt{l_1}, \ldots, \mathtt{l_m}, \mathtt{not}\ \mathtt{l_{m+1}}, \ldots, \mathtt{not}\ \mathtt{l_n}$. An answer set program with variables is seen usually as the set of all ground instances obtained by substituting ground terms of $\Sigma$ for all variables. A disjunctive answer set program is a set of rules where the head $l$ may be a disjunction $\mathtt{l'_1}\ \mathtt{v}\ \ldots\ \mathtt{v}\ \mathtt{l'_k}$ of atoms and strong negation literals.

An answer set $A$ of $P$ is a set of ground atoms and strong negation literals such that $A$ is a $\subseteq$-minimal element in the collection of all sets $S$ of such literals that have the property that, for each rule

$$\mathtt{l'_1} \mathtt{v} \ldots \mathtt{v} \mathtt{l'_k}\,\mathtt{:-}\,\mathtt{l_1}, \ldots, \mathtt{l_m}, \mathtt{not}\ \mathtt{l_{m+1}}, \ldots, \mathtt{not}\ \mathtt{l_n}.$$

if $l_1, \ldots, l_m \in S$ and $l_{m+1}, \ldots, l_n \notin A$, then at least one $l'_i \in S$.

Note that for a constraint, there exists at least one literal $l_i$ such that either $1 \leq i \leq m$ and $l_i \notin A$ or $m+1 \leq i \leq n$ and $l_i \in A$.

As for the informal semantics, an answer set program $P$ is seen as the representation of all knowledge of some rational introspective epistemic agent. A rule $\mathtt{l'_1 v \ldots v l'_k :- l_1, \ldots, l_m, not l_{m+1}, \ldots, not l_n}$. expresses that this agent believes one of $\mathtt{l'_k}$ if he believes $\mathtt{l_1, \ldots, l_m}$ to be true and considers it possible that each of $\mathtt{l_{m+1}, \ldots, l_n}$ is false. The agent obeys a *rationality principle*, that is that he does not believe an atom or a strong negation literal unless his theory gives him an argument for believing it. In the answer set semantics, this rationality principle is formalized by the minimality requirement. Each answer set represents the set of believed literals in one of the possible states of belief of the agent.

First order logic (FO) formulas $\varphi$ over vocabulary $\Sigma$ are constructed using standard inductive rules for $\wedge, \neg, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall$. A $\Sigma$-interpretation $I$ (or $\Sigma$-structure) consists of a domain $U_I$, for each function symbol $F/n$ an $n$-ary function $F^I : U_I^n \to U_I$ and for each predicate symbol $P/n$ an $n$-ary relation $P^I$ on $U_I$. The interpretation of a term and the satisfaction of a formula under $I$ and some variable assignment $\theta$ is defined by standard inductive rules, such as $(F(t_1, \ldots, t_n))^{I\theta} = F^{I\theta}(t_1^{I\theta}, \ldots, t_n^{I\theta})$ and the (non-monotonic) inductive rule for the truth of negated formulas: $I\theta \models \neg\varphi$ if $I\theta \not\models \varphi$. The interpretation of variable-free ground terms $t$ and sentences $\varphi$ does not depend on the variable assignment $\theta$. We use $t^I$ to denote $t$'s interpretation and write $I \models \varphi$ ($I$ is a *model*) to denote that $\varphi$ is satisfied in $I\theta$ for arbitrary $\theta$.

A Herbrand interpretation is one in which $U_I$ is the set of ground terms of $\Sigma$ (called the Herbrand universe) and $I$ interprets each ground term $t$ by itself, i.e., for each function symbol $F$ and all term tuples $t_1, \ldots, t_n$, $F^I(t_1, \ldots, t_n) = F(t_1, \ldots, t_n)$. An alternative representation of a Herbrand interpretation is as a set of its true ground atoms. In this representation, a Herbrand interpretation is mathematically identical to an answer set of $\Sigma$. However, this is accidental and should not be taken to mean that both concepts are the same: in Tarskian model semantics, an interpretation (Herbrand or otherwise) does not represent a state of belief of an agent.

## 3 Basic Principles of Knowledge Representation in ASP and FO

An exquisite property of FO is the clarity and precision of the informal semantics of its connectives and quantifiers, and consequently, of its formulas. Tarskian model semantics is a mathematical theory developed to formalize this informal semantics. In the Tarskian model semantics, a model of an FO theory $T$ is viewed as an abstract representation of a state of the world that satisfies all axioms in $T$, hence that is a *possible world* according to $T$. This makes FO an *objective* language in that the truth of its formulas can be evaluated in the context of an objective world state. This is in contrast to negation-as-failure literals `not P` in ASP rules, whose truth is not determined by the material world but by the state of belief of an epistemic agent.

An FO theory $T$ over $\Sigma$ can of course also be understood as representing a state of belief of some agent. The belief state of $T$ is formalized by the collection[1] of its models: a model is a state of the world that is *possible* in that state of belief. Such a collection of interpretations, henceforth called a *collection (or set, in case) of possible worlds*, offers an extremely detailed representation of a belief state. Therefore, sets of possible worlds are used in the semantics of (propositional) epistemic logics such as kd45 as the representation of the belief state of the agent; such sets correspond to Kripke structures with the total accessibility relation.

As a representation of a belief state of an agent, a collection of possible worlds is vastly more precise than a (single) answer set. From any set of possible worlds, the corresponding answer set is the set of literals that are satisfied in all. But vice versa, not much information can be derived from an answer set of an agent about his collection of possible worlds. For instance, take $\Sigma = \{P, Q\}$ and assume that all we know about some agent is that his answer set is $\emptyset$, i.e., he or she knows (can derive) no literals. There are many sets of possible worlds with that answer set; the agents possible world set could be the set of models of the tautology *true*, or of $P \vee Q$, or $\neg P \vee Q$, or $P \vee \neg Q$, or $\neg P \vee \neg Q$, or $P \Leftrightarrow Q$ and more. In each case, he would not be able to derive a literal. Knowing only this answer set, all we can tell about the possible worlds is that there is at least one in which $P$ is true and one in which $P$ is false, and the same for $Q$. There is not a single interpretation that can be decided to be a possible world or rejected to be one. Hence, an answer set is a very coarse representation of a belief state. As we illustrate now, this has major impact on ASP methodology.

An answer set of an ASP program represents the belief of a rational agent—but who is this agent? The rational agent could be an existing epistemic entity in the real world: it could be a knowledge base that is being queried, or it could be us, the people writing the ASP program. As an example of this, we take the following scenario from [13].

*Example 1.* Assume a database with complete knowledge about the GPA (Grade Point Average) of students and partial knowledge about whether they belong to a minority group. A student is eligible for a grant if his GPA is high, or if he has a fair GPA and belongs to a minority. The policy of the department is to interview students who may be (but are not necessarily) eligible. In ASP, incomplete databases can be represented by a theory consisting of ground literals, such as:

$$\texttt{FairGPA(Bob)}.\ \ \texttt{Minority(Bob)}.\ \ \neg\texttt{Minority(Dave)}.$$
$$\texttt{FairGPA(Ann)}.\ \ \texttt{HighGPA(John)}.$$

The partial completeness of this database is expressed by two CWA rules:

$$\neg\texttt{FairGPA(x)} \leftarrow \texttt{not FairGPA(x)}.$$
$$\neg\texttt{HighGPA(x)} \leftarrow \texttt{not HighGPA(x)}.$$

The next rule expresses who is to be interviewed:

$$\texttt{Interview(x)} \leftarrow \texttt{FairGPA(x)}, \texttt{not Minority(x)}, \texttt{not}\neg\texttt{Minority(x)}.$$

[1] In general, the collection of models of an FO theory is too large to be a set.

In the database, Ann is the only student with fair GPA and unknown minority status. This ASP program has a unique answer set including the literals in the database and the atom `Interview(Ann)`. This corresponds to our beliefs about this domain; hence, the rational agent can be viewed to be us, the ASP programmer. Our possible worlds are all Herbrand interpretations satisfying the literals in the answer set. In the CWA rules and the `Interview` rule, the epistemic nature of negation as failure is well exploited, in the latter to verify that the KB knows that person $x$ is neither a minority nor a non-minority. This is an interesting sort of application that cannot be directly expressed in classical logic.

Many present-day examples of ASP concern computational problems such as graph coloring, Hamiltonian path, planning and scheduling problems, and many others. The only epistemic entity around in such domains is us, the people writing the ASP programs. Our knowledge is about data and the properties of a correct solution. The solutions that we want our systems to compute are given by certain output predicates in worlds that are possible according to this knowledge – the coloring predicate in a state in which the graph is correctly colored, the action predicates in a state where the actions form a correct plan, and so on. Unfortunately, our knowledge in such domains is highly disjunctive and there is no way in which possible worlds could be reconstructed from the set of literals that we "believe", i.e., that can be derived from our knowledge. For such domains, ASP developed a successful alternative methodology that makes use of a "virtual" non-existent agent. The "mind" of this figment of our imagination has been craftily constructed by us, the answer set programmer, in such a way that each of his possible belief sets corresponds to a single correct solution of the problem, i.e., to a *possible world.*

*Example 2.* The following ASP program represents the graph coloring problem for a given graph:

```
color(x, R) v color(x, B) v color(x, G) :- node(x).
:- color(x,c), color(y,c), edge(x,y).
node(1).  node(2). node(3). edge(1,2). edge(2,3). edge(1,3).
```

This program produces 6 answer sets, each of which corresponds to a coloring of the graph. E.g., the answer set

```
{ node(1).  node(2). node(3). edge(1,2). edge(2,3). edge(1,3).
  color(1,R). color(2,B). color(3,G).}
```

tells us that the (unique) color of 1 is $R$, that of 2 is $B$, and that of 3 is $G$. Obviously, this one answer set does not describe *our* beliefs about the colorings of this graph, since we know that there also exists, e.g., a coloring in which node 1 is not $R$.

In this example, and in many other ASP applications, *our* knowledge is represented by the *set of possible worlds* consisting of all Herbrand interpretations that are mathematically identical to the answer sets of the virtual agent. And there is no way in which the possible worlds, and hence, the solutions to the problem, could be extracted from *our* belief set.

The above observations show us that in ASP two different methodologies are in use. That of Example 1, called henceforth the ASP-belief methodology, is to represent the belief state(s) of an existing epistemic agent in the field. The second, called ASP-world methodology (Example 2), aims to characterize a collection of possible worlds by an answer set program of a virtual agent whose states of belief correspond to possible worlds. There is a simple heuristic criterion to determine which of these methodologies is used in an ASP application. Indeed, a real epistemic agent such as us or an existing knowledge base may have incomplete knowledge about the application domain, but in practice it is clear what this agent believes. I.e., the epistemic agent has a unique state of belief. Hence, ASP-belief programs typically have a unique answer set while ASP-world programs often have multiple answer sets.

The distinction between the ASP-belief and ASP-world methodologies is reminiscent of a discussion in Michael Gelfond's and Vladimir Lifschitz's seminal 1991 paper [13]. In this paper, they state that a set of rules without strong negation and disjunction can be viewed either as a *general logic program* or as an answer set program. In both cases the semantics of such a rule set is determined by the same stable models. The crucial difference, dixit Gelfond and Lifschitz, is that a stable model of the rule set, viewed as a general logic program, represents a Herbrand interpretation, i.e., a possible world, while it represents an answer set in the second case. There is an evident match here between general logic programming and the ASP-world methodology: ASP-world is the KR methodology of general logic programming. The distinction between general logic programs and answer set programs lays purely on the informal level. This might explain why it did not enter the eye of the ASP community and why the concept of general logic programs has not catched on in the ASP community. Yet, at present, it seems that the vast majority of ASP applications follows the ASP-world methodology and these programs are therefore to be viewed as (extensions of) general logic programs.

The relevance of distinguishing between these two methodologies is that in each different style, different kinds of knowledge can be represented, or the same knowledge is to be represented in different ways. Which methodology is chosen depends on the available knowledge and on the problem to be solved, e.g., whether a solution is (part of) a possible world, or what a KB knows/believes. If we want to match ASP to FO, our effort will therefore need to be parameterized on the particular way in which ASP is used. In the remainder of this paper, we will investigate how different sorts of knowledge can be represented in ASP-belief or ASP-world and in (extensions of) FO.

## 4 Adding (Inductive) Definitions to FO

A prototypical ASP-world application is the Hamiltonian cycle problem. The goal is to compute a Hamiltonian cycle through the graph, i.e., a path that passes through each node exactly once and then returns to the initial node. A typical ASP encoding is given by:

```
in(X, Y) v out(X, Y) :- edge(X,Y).
reached(X) :- reached(Y), in(Y,X).
reached(X) :- start(Y), in(Y,X).
:- node(X), not reached(X).
:- in(X,Y), in(X,Z), Y != Z.
:- in(X,Y), in(Z,Y), X != Z.
node(1).  node(2). node(3). edge(1,2). edge(2,3). edge(1,3).
```

The reachability relation defined in this program is a well-known example of an inductively definable concept that cannot be expressed in FO[17]. Definitional knowledge is an important form of human expert knowledge [4,7]. While simple non-inductive definitions can be expressed through *explicit definitions* $\forall \bar{x}(P(\bar{x}) \Leftrightarrow \varphi_P[\bar{x}])$, FO needs to be extended for inductive definitions. In the logic FO(LFP) [17], FO is extended with a least fixpoint construct. The definition of the reached relation would be expressed as:

$$\forall x(Reached(x) \Leftrightarrow \mathbf{lfp}_{R,x}[(\exists y\ (Start(y) \vee R(y)) \wedge In(y,x))](x)$$

In FO(ID) [7], inductive definitions are represented in a rule-based syntax, similar to the ASP encoding.

$$\left\{ \begin{array}{l} \forall x\ Reached(x) \leftarrow \exists y\ Reached(y) \wedge In(y,x). \\ \forall x\ Reached(x) \leftarrow \exists y\ Start(y) \wedge In(y,x). \end{array} \right\}$$

A rule (over $\Sigma$) is an expression of the form $\forall \bar{x}\ P(\bar{t}) \leftarrow \varphi$ where $P(\bar{t})$ is an atomic formula and $\varphi$ an FO formula, and $\leftarrow$ is the *definitional implication*. A definition $\Delta$ is a set of rules. A predicate symbol $P$ in the head of a rule of $\Delta$ is called a *defined predicate* of $\Delta$; a symbol of $\Sigma$ that occurs in $\Delta$ but is not defined by it is called a *parameter* or *open symbol* of $\Delta$. The set of defined predicates is denoted $Def(\Delta)$, the remaining symbols $Param(\Delta)$. An FO(ID) theory is a set of FO sentences and definitions.

The satisfaction relation $\models$ of FO(ID) is defined through the standard inductive rules for FO with one additional rule:

– $I \models \Delta$, if $I|_{Def(\Delta)} = \mathrm{wfm}^{\Delta}_{I|_{Param(\Delta)}}$

where $\mathrm{wfm}^{\Delta}_{I|_{Param(\Delta)}}$ is the *parameterized well-founded model* of $\Delta$ in the interpretation $I|_{Param(\Delta)}$. That is, to check whether $I$ is a model, we restrict $I$ to the parameter symbols of $\Delta$, extend this interpretation by performing the well-founded model construction in [29][2] and verify whether this well-founded model coincides with the (2-valued) interpretation $I$ on the defined symbols. The well-founded semantics formalizes the most common forms of inductive definitions such as monotone inductive definitions (e.g., reachability) and (non-monotone) definitions by induction over a well-founded order (e.g., the satisfaction relation $I \models \varphi$) [7].

[2] The well-founded model construction of [29] extends the original one of [30] by allowing arbitrary FO bodies and parameter symbols (interpreted by an extensional database).

In the next section, we identify useful, objective forms of knowledge that can be nicely encoded in ASP and we investigate how to represent them in FO(ID).

## 5 Representing Objective Knowledge in ASP and FO(ID)

### 5.1 Representing UNA and DCA

Implicitly, ASP maintains the *Unique Names Assumption* (UNA($\Sigma$)) [27] that all ground terms of $\Sigma_{Fun}$ represent different objects, and the *Domain Closure Assumption* (DCA($\Sigma$)) [27] that each object in the universe is represented by at least one ground term of $\Sigma_{Fun}$. These assumptions hold in many applications (e.g., databases), but neither UNA nor DCA are imposed in FO. That is, in FO they should be explicitly expressed.

UNA($\Sigma$) is expressed by the FO theory $UNA_{FO}(\Sigma)$ that consists of, for each pair of different function symbols $F/n, G/m \in \Sigma$ [27]:

$$\begin{array}{l} \forall \bar{x}\bar{y}\ \neg(F(\bar{x}) = G(\bar{y})) \\ \forall \bar{x}\bar{y}\ F(\bar{x}) = F(\bar{y}) \Rightarrow \bar{x} = \bar{y} \end{array}$$

It follows from the compactness of FO, that DCA($\Sigma$) cannot be expressed in FO, but it can be expressed in FO(ID) through the following theory $DCA_{FO(ID)}(\Sigma)$:

$$\begin{array}{l} \left\{ \begin{array}{l} \dots \\ \forall \bar{x}\ U(F(\bar{x})) \leftarrow U(x_1) \land \dots \land U(x_n). \\ \dots \end{array} \right\} \\ \forall x\ U(x) \end{array}$$

where the definition consists of one rule for each constant or function symbol $F/n \in \Sigma$ and $U$ is a new predicate representing the universe. The models of $UNA_{FO}(\Sigma) \land DCA_{FO(ID)}(\Sigma)$ are the Herbrand interpretations (up to isomorphism).

In many applications, UNA and DCA are too strong. For instance, in the applications typically considered in description logics, neither holds. In other applications, these axioms are applicable to only a subset $\sigma$ of $\Sigma_{Fun}$. A (very) early example of this is Peano arithmetic, Peano's standard second order theory of the natural numbers where $\sigma = \{0, S/1\}$. This theory contains $UNA_{FO}(\sigma)$ and a second order axiom expressing DCA($\sigma$), i.e., the induction axiom. Thus, UNA and DCA are not applied to $+/2$ and $\times/2$, the two other function symbols of Peano's theory.

In view of this, a number of variants of the ASP language have been designed with relaxations of UNA and/or DCA. An early one in which UNA still holds but DCA is dropped was proposed by Gelfond and Przymusinska [14]. In [19], an extension of ASP with functions is defined. This ASP extension applies the kind of relaxed UNA+DCA($\sigma$) axioms as observed above in Peano arithemetic. As a consequence, other function symbols represent arbitrary functions in the Herbrand universe of $\sigma$. In Open Answer Set Programming [15], UNA and DCA are dropped altogether. Applications of all these extensions arguably follow the ASP-world methodology: answer sets represent belief states of a virtual agent but possible worlds of the human expert.

## 5.2 Different Forms of CWA in Complete and Partial Databases

The general idea of the Closed World Assumption (CWA) [26] in the context of a theory $T$ is the assumption that atoms that cannot be derived from $T$ are false. In the context of incomplete knowledge, this principle often leads to inconsistency and needs to be relaxed. There are different options to do that. In fact, ASP-belief and ASP-world provide *different* forms of CWA.

In ASP-belief, CWA is expressed through ASP formulas of the form:

```
¬P(x̄) :- not P(x̄).
```

Informally, the rule states that `P(x̄)` is false if it is consistent to assume so. Such rules can be used to represent databases with complete and partial knowledge. Thus, the epistemic `not` connective of ASP allows us to explicitate a local form of CWA.

*Example 3.* An ASP-belief representation of a complete database :

```
FairGPA(Bob). Minority(Bob). FairGPA(Ann). HighGPA(John).
¬FairGPA(x) ← not FairGPA(x).
¬HighGPA(x) ← not HighGPA(x).
¬Minority(x) ← not Minority(x).
```

By deleting the CWA rule for `Minority` and adding strong negation literal `¬Minority(Dave)` we obtain the partial database with incomplete knowledge of Example 1.

In ASP-world, no CWA axioms need to be added; on the contrary, axioms are needed to express where the CWA does not hold. Thus in ASP-world, (a form of) CWA is implicit and needs to be overruled when necessary.

*Example 4.* An ASP-world representation of the complete database of Example 3:

```
FairGPA(Bob). Minority(Bob). FairGPA(Ann). HighGPA(John).
```

The partial database of Example 1 is represented by extending this with a constraint and a disjunctive rule to open up `Minority`:

```
Minority(x) v Minority*(x).
:-Minority(Dave).
```

There are several alternative syntaxes to do the latter in ASP, such as the lparse syntax "`0{Minority(x)}1`." or the traditional cycle over negation:

```
Minority(x):-not Minority*(x).
Minority*(x):-not Minority(x).
```

Interestingly, the mathematical closure principle underlying stable semantics, according to which an atom belongs to a stable model only if it can be derived by a rule, acts as a rationality principle in ASP-belief, but as a form of CWA in ASP-world! This is a clear example of how the same form of knowledge is represented in different ways in ASP-belief or ASP-world. Note that adding CWA-rules in ASP-world such as $\neg$`FairGPA(x)` $\leftarrow$ `not FairGPA(x)` is of no use: it expresses that the virtual epistemic agent has CWA on this predicate but not that we, the ASP programmer, have CWA on $FairGPA$. The effect is merely to add strong negation literals such as $\neg$`FairGPA(John)` to all answer sets, but strong negation literals are irrelevant since they are ignored when computing the possible worlds from the virtual agent's answer sets.

Recapitulating, in ASP-belief, CWA is not imposed unless explicit rules are added; in ASP-world, CWA is imposed unless overridden explicitly by rules that open up an atom.

The CWA cannot be expressed directly in FO, due to FO's monotonicity. However, a local form of CWA can be simulated through predicate completion [5] , as in the axiom:

$$\forall x(FairGPA(x) \Leftrightarrow x = Bob \vee x = Ann)$$

An alternative solution in FO(ID) is to use definitions to close predicates. The above partial database is represented in FO(ID) as:

$$\begin{array}{l} \{\, FairGPA(Bob).\;\; FairGPA(Ann).\,\} \\ \{\, HighGPA(John).\,\} \\ Minority(Bob).\;\; \neg Minority(Dave). \end{array}$$

Inductive definitions in mathematics quite clearly comprise some form of CWA. Indeed, an object or tuple does not belong to an inductively defined set *unless it is explicitly derived by a rule application.* It is not uncommon to find inductive definitions in mathematical texts that contain explicit CWA-like statements, as in the following definition:

We define the set of natural numbers by induction:
- 0 is a natural number;
- if $n$ is a natural number then $n + 1$ is a natural number;
- **no other objects are natural numbers.**

Thus, definitions are an informal form of CWA that we understand with great precision. In one of the next sections, we will use this form of CWA to represent defaults.

The following example shows that the CWA's expressed in ASP-belief differ from the one implicit in ASP-world and in definitions.

*Example 5.* In the context of the complete student database of Example 3, the following ASP-belief rules express which students are eligible for a grant :

```
Eligible(x) :- HighGPA(x).
Eligible :- FairGPA(x), Minority(x).
¬Eligible(x) :- not Eligible(x)
```

There is a problem with this representation in the context of the incomplete database. Indeed, Ann has a fair GPA but an unknown Minority status, and hence, is unknown to be eligible. Hence, the CWA rule jumps to the wrong conclusion that she is not eligible[3]. An ASP-world representation of eligibility is obtained by dropping the CWA rule from this definition. This is similar to FO(ID), where the predicate *Eligible* can be defined as:

$$\left\{\begin{array}{l} \forall x(Eligible(x) \leftarrow HighGPA(x). \\ \forall x(Eligible \leftarrow FairGPA(x) \land Minority(x)). \end{array}\right\}$$

The ASP-world representation as well as the FO(ID) representation is correct whether the database is complete or only partially complete.

An ASP-belief CWA rule turns ignorance of an atom into falsity. Intuitively, if there is one possible world in the agents belief state where the atom is false, the atom is false in all possible worlds. In contrast, in ASP-world and in definitions of FO(ID), a defined atom is false in a possible world if it cannot be produced by rule application in that particular world. Thus, in one world, Ann might be eligible and in another she might not be. This is a weaker form of CWA than the one expressed in ASP-belief, but more useful in this application. This clearly shows that the form of CWA that can be expressed in ASP-belief differs from the form of CWA implicit in ASP-world and in FO(ID) definitions.

## 5.3 Representing Definitions in ASP

Definitions are important in KR [4]. Many applications, also in ASP, contain rules that make up for a definition of one or more predicates. Examples in this paper are the rule defining `Interview` in Example 1 and the rules for `Eligible` in the previous section. (Inductive) definitions can be correctly represented in ASP-world and of course in FO(ID). It is more difficult to represent them in ASP-belief, as we saw in the previous section.

As shown in Example 5, representing definitions in ASP-belief is simple as long as the ASP program has complete knowledge on the parameter predicates. In that case, it suffices to add a CWA rule to the rules expressing the definition. Such a representation unfortunately leads to errors in case of incomplete databases (The parameter predicate `Minority` is incomplete in Example 5). To solve this problem in ASP-belief, we can use the interpolation technique developed by Baral, Gelfond and Kosheleva in [2,3]. They develop a technique to approximate *lp-functions* by an answer set program. An lp-function consists of a set of rules (without strong negation) and a set of input and output parameters. There is a clear and intuitive match between definitions of FO(ID) and lp-functions: a definition can be seen as an lp-function with the definitions parameters matching the input parameters, and the defined predicates matching the output parameters of the corresponding lp-function. The algorithm that they propose can be used to translate the rules for *Eligible* into an interpolation: an

[3] We will discuss a solution for this problem in Section 5.3.

answer set program that approximates the original rules in the context of an incomplete database. Applying this algorithm on the rules of *Eligible* yields:

```
MayBeMinority(x) ← not¬Minority(x).
Eligible(x) ← HighGPA(x).
Eligible(x) ← FairGPA(x), Minority(x).
MaybeEligible(x) ← HighGPA(x).
MaybeEligible(x) ← FairGPA(x), MayBeMinority(x).
¬Eligible(x) ← notMaybeEligible(x).
```

The algorithm adds predicates representing students that might be minorities or might be eligible and defines `¬Eligible` as the complement of the *maybe eligible* predicate.

The algorithm depends on which predicates the database has incomplete knowledge about and needs to be modified when more parameter predicates are unknown (e.g., `FairGPA`, `HighGPA`). The correctness and the accuracy of the approach depends on certain conditions as expressed in the correctness theorem in [3]. In particular, if there is disjunctive knowledge on the parameters of the definition, accuracy will be lost. For instance, if we add to the database that also John has a fair GPA and either he or Ann is a minority but not both, then the natural conclusion that either John or Ann is not eligible cannot be drawn from an interpolation.

This shows that representing definitions in ASP-belief has its pitfalls. Also, it shows that introducing incomplete knowledge in ASP-belief has more impact on the representation than in ASP-world and FO(ID).

## 5.4 Representing Defaults

Nonmonotonic Reasoning has its motivation in the problems of FO for representing common sense knowledge [23]. One problem is the *qualification problem*: many universally quantified statements are subject to a very long, if not endless list of qualifications and refinements. The standard example is the statement that *birds can fly*. But what about ostriches? Chickens? Chicks? Birds with clipped or broken wings? Etc. All we can say is: *normally*, a bird can fly, or *most* birds can fly. This gave rise to the domain of *defaults* and *default reasoning*. Defaults are approximative statements and reason about normality (typically, normally, usually, often, most, few, ..).

An essential part of human intelligence is the ability to reason with incomplete, vague, and uncertain information. When such information is represented in classical logic, where sentences are either wholly true or wholly false, the result is a crisp approximation of the fuzzy reality. The knowledge of a human expert evolves with time, e.g., when he learns more about particular objects or comes across new special cases and exceptions. Hence, the theory has to be continually refined and revised. An important issue here is *elaboration tolerance*; the ease by which new knowledge can be integrated in an old theory.

Let us illustrate this in a traditional example scenario. Assume that we want to represent knowledge about locomotion of animals. We know that most birds fly and that Tweety and Clyde are birds. If that is all we know, it is reasonable to assume that both fly. One approximately correct FO theory that entails the desired conclusions is:

$$T = \Big\{ \forall x(Bird(x) \Rightarrow Fly(x)), Bird(Tweety) \land Bird(Clyde) \Big\}$$

Next, we find out that Tweety is a penguin and want to extend our theory to, e.g.,

$$\forall x(Penguin(x) \Rightarrow Bird(x)), \forall x(Penguin \Rightarrow \neg Fly(x)), Penguin(Tweety)$$

However, this theory is inconsistent, and we also need to weaken the axiom that birds fly:

$$\forall x(Bird(x) \land \neg Penguin(x) \Rightarrow Fly(x))$$

Unfortunately, we now have lost the desirable but defeasible conclusion that Clyde flies. Adding the reasonable assumption $\neg Penguin(Clyde)$ (since "most birds are not penguins") would solve the problem, at the risk of adding potentially false information, since our assumption may be wrong. And even if Clyde turns out to be an eagle, we might later find out that he is a chick, and these don't fly. Clearly, FO's lack of elaboration tolerance makes it quite messy to maintain our formula that most birds fly.

An important goal of nonmonotonic reasoning is to design *elaboration tolerant* logics and methodologies that support this process of representing and refining evolving expert knowledge. For defaults, ASP achieves elaboration tolerance by means of its non-monotonic negation `not`, which allows additional exceptions to existing rules to be added with minimal effort:

```
Flies(x) :- Bird(x), not Abnormal(x).
Abnormal(x) :- Penguin(x).
```

Adding an additional exception can now be done by the almost trivial operation of introducing a new rule with the abnormality predicate in its head.

In the context of FO, we can achieve a remarkably similar effect, by again turning to the inductive definition construct of FO(·). Indeed, inductive definitions are essentially a non-monotonic language construct, in the sense that the operation of adding a new definitional rule to an existing definition is a non-monotonic operation, due to the CWA embodied in a definition. We can leverage this property to represent a default $d$ stating that "$P$s are typically $Q$s" as follows:

$$\forall x\ P(x) \land \neg Ab_d(x) \Rightarrow Q(x)$$
$$\{\forall x\ Ab_d(x) \leftarrow \mathbf{f}.\}$$

This theory defines the abnormality predicate as empty, and is therefore equivalent to the sentence $\forall x\ P(x) \Rightarrow Q(x)$. However, when new exceptions to the default come to light, they can easily be added as new definitional rules. For instance, if we discover that the default does not apply to $R$s:

$$\forall x\ P(x) \land \neg Ab_d(x) \Rightarrow Q(x)$$
$$\left\{\begin{array}{l} \forall x\ Ab_d(x) \leftarrow \mathbf{f}. \\ \forall x\ Ab_d(x) \leftarrow R(x). \end{array}\right\}$$

This has the effect of turning the theory into one equivalent to $\forall x\ P(x) \land \neg R(x) \Rightarrow Q(x)$, but does so in an elaboration tolerant way.

In a sense, every definition includes a kind of default information, namely that the defined concept is false by default, where the rules express the exceptions. For instance, the definition:

$$\{\forall x\ Square(x) \leftarrow Rectangle(x) \land Rhombus(x).\}$$

expresses that objects typically are not squares, with those objects that are both a rectangle and a rhombus forming an exception to this general rule. This allows us to represent certain kinds of interplaying defaults in a quite compact but still elaboration tolerant way. For instance, the following definition expresses that "animals typically do not fly ($d_1$), but birds are an exception, in the sense that these typically do fly ($d_2$), unless they happen to be penguins, which typically do not fly ($d_3$)":

$$\left\{\begin{array}{l} \forall x\ Flies(x) \leftarrow Bird(x) \land \neg Ab_{d_2}(x). \\ \forall x\ Ab_{d_2}(x) \leftarrow Penguin(x) \land \neg Ab_{d_3}(x). \\ \forall x\ Ab_{d_3}(x) \leftarrow \mathbf{f}. \end{array}\right\}$$

A new exception to $d_1$ might be that all bats also fly: this is the simple update of adding the definitional rule $\forall x\ Flies(x) \leftarrow Bat(x) \land \neg Ab_{d4}(x)$ and an empty definition for the latter abnormality predicate. A new exception to $d_2$ might be a bird with a broken wing: $\forall x\ Ab_{d_2}(x) \leftarrow BrokenWing(x)$. Finally, a penguin with a jet pack: $\forall x\ Ab_{d_3}(x) \leftarrow JetPack(x)$.

## 6 Adding an Epistemic Operator to FO(·)

In this section, we are concerned with expressing inherently epistemic ASP applications of the kind that can be expressed in ASP-belief. For this, we return to Example 1, where we had an ASP-belief partial knwoledge base with the following rule to express that students with fair GPA and unknown minority status need to be interviewed:

$$\texttt{Interview(x)} \leftarrow \texttt{FairGPA(x)}, \texttt{not Minority(x)}, \texttt{not}\neg\texttt{Minority(x)}.$$

In ASP-world, the latter rule does not work for the simple reason that the virtual agent has different beliefs about minority students than the knowledge base (or we) has. E.g., in the context of the ASP-world partial database of Example 4, adding the above rule would lead to one answer set containing `Minority(Ann)` and no `Interview(Ann)` and another answer set without `Minority(Ann)` but with `Interview(Ann)`. This is not what we need.

To express this example in a natural way, an epistemic operator is needed as provided in autoepistemic logic (AEL [24]). Applying the translation from ASP to AEL [12], we obtain a correct AEL representation:

$$\begin{array}{l} FairGPA(Bob) \land Minority(Bob) \land \neg Minority(Dave) \land FairGPA(Ann) \\ \forall x(\neg FairGPA(x) \Leftarrow \neg K FairGPA(x)) \\ \forall x(Interview(x) \Leftarrow FairGPA(x) \land \neg K Minority(x) \land \neg K \neg Minority(x)) \end{array}$$

In general, due to its self-referential nature, an AEL theory may have multiple stable expansions, each describing a different state of belief. This would be highly undesirable in this application, since we want the knowledge base to come up with a single set of people to be interviewed. Fortunately, the above theory is a stratified one. Therefore, it has a unique stable expansion [20] which correctly captures what the knowledge base knows.

For applications of this kind, it will suit us to develop an approach that is both simpler and closer to FO than AEL. Given that we already have an FO theory defining our incomplete database and its associated set of possible worlds, it is natural to consider a layered approach, where we keep our original database theory and just add an additional layer on top of that. This new layer is again just another FO theory, with the exception that it may query the knowledge contained in the database theory. More generally, it is of course equally possible to consider many layers, each building on the knowledge contained in previous layers. This suggests the following straightforward multi-modal extension of FO for reasoning on knowledge bases composed from multiple knowledge sources.

**Definition 1.** *An ordered epistemic theory $\mathcal{T}$ over vocabulary $\Sigma$ is a finite set of multi-modal logic theories over $\Sigma$ that satisfies the following conditions:*

– *Each $T \in \mathcal{T}$ is a multi-modal logic theory, possibly using modal operators $K_{T'}$, where $T' \in \mathcal{T}$.*
– *The modal operators are used in a stratified way, i.e., there is a partial order $<$ on $\mathcal{T}$ and if the modal literal $K_T\varphi$ occurs in a formula of subtheory $T'$ or in the scope of another modal literal $K_{T'}\psi$, then $T < T'$.*

Intuitively, each $T \in \mathcal{T}$ is expressing the knowledge of a component of the knowledge base. Theories in higher levels possess the knowledge of lower theories and can query lower levels through expressions $K_T\varphi$. Note that if $T$ is minimal in $<$, then it is an FO theory. If $T$ is maximal, then $K_T$ occurs nowhere in $\mathcal{T}$.

Self-referential autoepistemic statements such as $T = \{\neg K_T P \Rightarrow P\}$ cannot be expressed in ordered epistemic logic. The benefit is that FO semantics extends easily. To keep things simple and standard, we consider only Herbrand interpretations of the vocabulary $\Sigma$. This effectively means that DCA($\Sigma$) and UNA($\Sigma$)

hold[4] and moreover, that all function and constant symbols $\sigma$ are *rigid*: they have the same interpretation in all possible worlds. Let $\mathcal{T}|_T$ denote the restriction of $\mathcal{T}$ to $\{T' \in \mathcal{T} | T' \leq T\}$.

For any (Herbrand) $\Sigma$-interpretation $M$ and variable assignment $\theta$, we define that $M\theta$ satisfies a formula $\varphi$, denoted $M\theta \models \varphi$, by extending the standard inductive rules of FO with one extra rule[5]:

– $M\theta \models K_T\varphi$ if for each interpretation $M'$ such that $M' \models \mathcal{T}|_T$, it holds that $M'\theta \models \varphi$.

As usual, we define $M \models \varphi$ for sentences $\varphi$ -without free variables- if for some variable assignment $\theta$, $M\theta \models \varphi$. $M$ satisfies an ordered epistemic theory $\mathcal{T}$ if it satisfies each sentence in each $T \in \mathcal{T}$. Thus, a theory $T$ can be seen as the distributed knowledge of a layered set of agents, where each higher agent posses the knowledge of its lower agents but not vice versa. Just like an FO theory, an ordered epistemic theory defines a unique possible world set, namely the set of all its models.

A useful feature of this logic is that it is straightforward to extend it with other objective language constructs in FO($\cdot$) such as definitions and aggregates. In the sequel, we will include definitions in the examples.

It is easy to see how our example fits into ordered epistemic logic. Let $DB$ be either our FO or FO(ID) representation—it does not matter which—of the incomplete database. We now build on top of this a second layer $T_I > DB$, consisting of a single definition:

$$T_I = \left\{ \left\{ \begin{array}{l} \forall x\ Interview(x) \leftarrow FairGPA(x) \land \\ \qquad\qquad \neg K_{DB}(Minority(x)) \land \neg K_{DB}(\neg Minority(x)). \end{array} \right\} \right\}$$

Alternatively, we could of course also have used an FO equivalence:

$$T_I' = \left\{ \begin{array}{l} \forall x\ Interview(x) \Leftrightarrow FairGPA(x) \land \\ \qquad\qquad \neg K_{DB}(Minority(x)) \land \neg K_{DB}(\neg Minority(x)). \end{array} \right\}$$

Note that we have strengthened the original *implication* specifying $Interview$ into an arguably more accurate *definition.*

It is easy to see that the ordered epistemic theory $\mathcal{T} = (DB, T_I)$ has two models: one in which Ann is a minority member, one in which she is not. Therefore, $\mathcal{T}$ entails that only Ann needs an interview.

While in many useful cases, an ordered epistemic theory can be viewed as an autoepistemic theory, we will not delve in this.

[4] A standard way in modal logic of relaxing the assumption of a fixed domain in all worlds is to introduce a unary universe predicate $U$ and allowing only relativized quantifier formulas $\exists x(U(x) \land \varphi)$ and $\forall x(U(x) \Rightarrow \varphi)$.

[5] Observe that this is the second time in this paper that we extend FO's satisfaction relation $\models$ by adding a rule to its definition (the first time was for formal FO(ID) definitions). This illustrates that the elaboration tolerant, non-monotonic update operations that we used in Section 5.4 in the FO(ID) encoding of defaults, occurs also in informal inductive definitions.

## Disjunction in ASP-Belief – A Epistemic Operator in ASP-World

As shown in the beginning of this section, negation as failure cannot be used as epistemic operator to query partial ASP-world knowledge bases. In [11], Gelfond investigated this problem in the context of disjunctive answer set programs. To motivate his approach, Gelfond illustrated the epistemic use of negation as failure with Example 1 including the ASP-belief rule:

$$\texttt{Interview(x)} \leftarrow \texttt{FairGPA(x)}, \texttt{not Minority(x)}, \texttt{not } \neg\texttt{Minority(x)}.$$

This rule will correctly identify the students to be interviewed in all simple ASP-belief databases. Next, Gelfond considered a database with the disjunctive information that either Ann or Mike are members of a minority group:

```
FairGPA(Ann). FairGPA(Mike).
Minority(Ann) v Minority(Mike).
```

In this case, we would like to infer that both Ann and Mike need to be interviewed. However, the program has two answer sets, one with `Interview(Ann)`, and another with `Interview(Mike)`, and therefore, the answer of this program to, e.g., the query `Interview(Mike)` is unknown.

To solve this problem, Gelfond proposed to add another epistemic operator $K$ to ASP, called the *strong introspection operator*, that queries whether a literal is present in *all* answer sets. Using this operator, we can correctly express the definition of $Interview$ by:

$$\texttt{Interview(x)} \leftarrow \texttt{FairGPA(x)}, \texttt{not K not Minority(x)}, \texttt{not K Minority(x)}.$$

The resulting program has two correct answer sets:

$$\{\texttt{FairGPA(Ann)}, \texttt{FairGPA(Mike)}, \texttt{Minority(Ann)}, \texttt{Interview(Ann)}, \texttt{Interview(Mike)}\}$$
$$\{\texttt{FairGPA(Ann)}, \texttt{FairGPA(Mike)}, \texttt{Minority(Mike)}, \texttt{Interview(Ann)}, \texttt{Interview(Mike)}\}$$

This representation not only works for Gelfonds disjunctive datatase but also for the ASP-world partial database of Example 4.

In ordered epistemic logic, the problem could be solved by the theory $\mathcal{T}$ consisting of $DB < T_1$ where $T_1$ is as before and

$$DB = \left\{ \begin{array}{l} FairGPA(Ann).\ FairGPA(Mike.) \\ Minority(Ann) \vee Minority(Mike). \end{array} \right\}$$

This is a correct representation of the scenario and entails that both will be interviewed.

What Gelfond basically observed here is that the use of disjunction in the head turns an ASP-belief program into an ASP-world one. Before adding the disjunction, the answer set represents a belief state of the knowledge base; after adding the disjunction, this is no longer the case. The belief state of the knowledge base is reflected by the set of possible worlds in which either Ann or Mike is a minority. This belief set does not correspond to any of the two answer sets of the disjunctive program. Thus, the use of disjunction in ASP leads here and in many other cases to ASP-world programs. What we can conclude here is as follows:

– Without strong introspection, ASP-world does not provide an autoepistemic operator. ASP-world is not suitable to express introspective statements.
– ASP-belief is not suitable to express disjunctive information.
– Introduction of disjunction (in any of the known ways: disjunctive rule, loop over negation, weight constraints) in a programs including epistemic operators may force us to modify the ASP program. In comparison, the representation in ordered epistemic FO(.) is more elaboration tolerant.

In conclusion, this section has considered an interesting class of epistemic ASP-belief programs, namely, those in which conclusions need to be drawn from incomplete knowledge bases. ASP's combination of strong negation and NAF allows such problems to be elegantly represented and most applications in which strong negation plays a significant role seem to be of this kind. These applications naturally exhibit a layered structure, in which each layer is a knowledge base with its own area of expertise, and different layers may build on each other's knowledge. This suggest the simple epistemic logic that we have presented in this section. This logic has the benefit of simplicity and, moreover, it integrates effortlessly into classical logic or its extensions. While the logic is weaker than AEL, in the sense that it does not allow self reference, it guarantees a unique epistemic model, and it is possible to prove that inference tasks have a lower complexity than in AEL and in disjunctive answer set programming.

## 7 Conclusion

Gelfond and Lifschitz have merged ideas of logic programming and non-monotonic reasoning to build a practically useful KR-language — answer set programming. In this paper, we have highlighted and analysed these contributions by investigating what they contribute to classical first order logic (FO). We studied how forms of knowledge for which ASP was designed can be represented in FO or in suitable extensions of FO.

A prime factor in our analysis turned out to be the dichotomy between the ASP-belief and ASP-world methodologies: whether a program is written so that an answer set represents a belief set of an existing agent or a possible world. This decision turns out to have an overwhelming effect on the knowledge representation process in ASP:

– Some knowledge principles are implicit in one and not in the other (e.g., CWA).
– Some forms of knowledge can only be expressed in one (e.g., disjunction, autoepistemic statements).
– Some forms of knowledge can be expressed in both, but in considerably different ways (e.g., definitions – with interpolation in ASP-belief).

In the development of ASP, Gelfond and Lifschitz were led by the ASP-belief view. In the current state of the art, it seems that by far most applications of ASP are developed according to ASP-world view. We have explained the reason for this: answer sets are simply too coarse grained to represent the belief state of a real agent or knowledge base in sufficient detail.

To consolidate the contributions of ASP to KR, we believe that the best strategy is to integrate its contributions to FO. The same holds also for other KR-extensions of logic programming such as abductive logic programming. In the past, we have been led by this idea in the development of FO(.). We have shown here how ASP's KR-contributions can be mapped to elaboration tolerant FO(.) theories. For the development of FO(.), we are indebted to Michael Gelfond and Vladimir Lifschitz, whose work, as can be seen in this paper, has been a continous source of inspiration.

## References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
2. Baral, C., Gelfond, M., Kosheleva, O.: Approximating general logic programs. In: ILPS, pp. 181–198 (1993)
3. Baral, C., Gelfond, M., Kosheleva, O.: Expanding queries to incomplete databases by interpolating general logic programs. J. Log. Program. 35(3), 195–230 (1998)
4. Brachman, R.J., Levesque, H.J.: Competence in knowledge representation. In: National Conference on Artificial Intelligence (AAAI 1982), pp. 189–192 (1982)
5. Clark, K.L.: Negation as failure. In: Logic and Data Bases, pp. 293–322. Plenum Press, New York (1978)
6. Denecker, M.: What's in a model? Epistemological analysis of logic programming. In: Dubois, D., Welty, C.A., Williams, M.-A. (eds.) KR, pp. 106–113. AAAI Press, Menlo Park (2004)
7. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Transactions on Computational Logic (TOCL) 9(2), Article 14 (2008)
8. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second Answer Set Programming competition. In: Erdem, E., et al. (eds.) [9], pp. 637–654
9. Erdem, E., Lin, F., Schaub, T. (eds.): LPNMR 2009. LNCS, vol. 5753. Springer, Heidelberg (2009)
10. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Veloso, M.M. (ed.) IJCAI, pp. 372–379 (2007)
11. Gelfond, M.: Strong introspection. In: AAAI, pp. 386–391 (1991)
12. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) ICLP/SLP, pp. 1070–1080. MIT Press, Cambridge (1988)
13. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3/4), 365–386 (1991)
14. Gelfond, M., Przymusinska, H.: Reasoning on open domains. In: LPNMR, pp. 397–413 (1993)
15. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Open answer set programming with guarded programs. ACM Trans. Comput. Log. 9(4) (2008)

16. Kowalski, R.A.: Predicate logic as programming language. In: IFIP Congress, pp. 569–574 (1974)
17. Libkin, L.: Elements of Finite Model Theory. Springer, Heidelberg (2004)
18. 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)
19. Lin, F., Wang, Y.: Answer set programming with functions. In: Brewka, G., Lang, J. (eds.) KR, pp. 454–465. AAAI Press, Menlo Park (2008)
20. Marek, V.W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM 38(3), 588–619 (1991)
21. McCarthy, J.: Programs with common sense. In: Teddington Conference on the Mechanization of Thought Processes (1958)
22. McCarthy, J.: Applications of circumscription to formalizing common-sense knowledge. Artificial Intelligence 28(1), 89–116 (1986)
23. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press, Edinburgh (1969)
24. Moore, R.C.: Possible-world semantics for autoepistemic logic. In: Proceedings of the Workshop on Non-Monotonic Reasoning, pp. 344–354 (1984); reprinted in: Ginsberg, M. (ed.) Readings on Nonmonotonic Reasoning, pp. 137–142. Morgan Kaufmann, San Francisco (1990)
25. Pührer, J., Heymans, S., Eiter, T.: Dealing with inconsistency when combining ontologies and rules using dl-programs. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 183–197. Springer, Heidelberg (2010)
26. Reiter, R.: On closed world data bases. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 55–76. Plenum Press, New York (1977)
27. Reiter, R.: Equality and domain closure in first-order databases. Journal of the ACM 27(2), 235–249 (1980)
28. Reiter, R.: A logic for default reasoning. Artif. Intell. 13(1-2), 81–132 (1980)
29. Van Gelder, A.: The alternating fixpoint of logic programs with negation. Journal of Computer and System Sciences 47(1), 185–221 (1993)
30. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)

# Closure and Consistency Rationalities in Logic-Based Argumentation

Phan Minh Dung and Phan Minh Thang

Department of Computer Science, Asian Institute of Technology
GPO Box 4, Klong Luang, Pathumthani 12120, Thailand
{dung,thangphm}@cs.ait.ac.th

**Abstract.** Caminada and Amgoud have argued that logic-based argumentation systems should satisfy the intuitive and natural principles of logical closure and consistency. Prakken has developed this idea further for a richer logic. A question arises naturally whether a general structure guaranteeing the logical closure and consistency properties could be identified that is common for all underlying logics. We explore this question by first defining a logic-based argumentation framework as combination of an abstract argumentation framework with a monotonic Tarski-like consequence operator. We then demonstrate that the logical closure and consistency properties are rested on a simple notion of a base of arguments from which the argument could be constructed in an indefeasible way (using the monotonic consequence operator) and the only way to attack an argument is to attack its base. We show that two natural properties of structural closure and consistency covering based on the idea of bases of arguments indeed guarantee the logical closure and consistency properties. We demonstrate how the properties of structural closure and consistency covering are captured naturally in argumentation systems of Caminada, Amgoud and Prakken as well as in assumption-based argumentation.

## 1 Introduction

How do we know whether an argumentation framework is appropriate for its purposes ? Caminada and Amgoud [3] have argued that for logic-based systems, they should at least satisfy two intuitive and natural principles of logical closure and consistency. Prakken [8] has developed this idea further for a richer logic. But as there are many logics, Caminada, Amgoud and Prakken's results do not cover all of them. As argumentation is charaterized by arguments and the attack relation between them, a natural question is whether the logical closure and consistency principles could be captured in abstract argumentation without associating to a specific logic?

Logic-based abstract argumentation is viewed as abstract argumentation equipped with a general Tarski-like (monotonic) consequence operator. We develop in this paper two general principles of structural closure and consistency covering in logic-based abstract argumentation and show that they indeed capture the intuitions of the logical closure and consistency principles. The principles of structural closure and consistency covering rest on a simple notion of a base of arguments from which the argument could be constructed in an indefeasible way (using the monotonic consequence operator) and the only way to attack an argument is to attack its base. In other words the principle of logical closure boils down to the idea that if an argument is considered to be a "indefeasible logical consequence" of a set of arguments then the former must be acceptable wrt the later. The principle of consistency covering reduces the logical consistency to a kind of a conflict-freeness.

## 2 Logic-Based Abstract Argumentation Theories

Argumentation is a form of reasoning, that could be viewed as a dispute resolution, in which the participants present their arguments to establish, defend, or attack certain propositions. An abstract argumentation framework [5] is defined simply by a pair $(\mathcal{AR}\ , att)$ where $\mathcal{AR}$ is a set of arguments and $att$ is a binary relation over $\mathcal{AR}$ representing the relation that an argument $A$ *attacks* an argument $B$ for $(A, B) \in att$. The semantics of abstract argumentation is determined by the acceptability of arguments and various associated notions of extensions. For the purpose of this paper, we introduce two of them. A set of argument $S$ attacks an argument $A$ if some argument in $S$ attacks $A$; $S$ is *conflict-free* if it does not attack itself. An agument A is *acceptable* wrt set of arguments S if S attacks each attack against A. $S$ is *admissible* if $S$ is conflict-free and it counter-attacks each attack against it. The *characteristic function* $\mathcal{F}$ assigns to each set of arguments S the set of arguments that are acceptable wrt S. As $\mathcal{F}$ is monotonic, $\mathcal{F}$ has a least fixed point. A *complete extension* is defined as a fixed point of $\mathcal{F}$ while the *grounded extension* is the least fixed point of $\mathcal{F}$. A *stable extension* is defined as a conflict-free set of arguments that attacks every argument not belonging to it. It is well-known that each stable extension is a complete extension but not vice versa. Stable extensions generalize the stable and answer set semantics of [6, 7].

Intuitively, an argument is a proof of some conclusion. In many cases, such proofs are constructed following some proof theory of some formal logics. Such logics could be nonmonotonic. The notion of closure is then defined accordingly as the set of consequences following from the monotonic parts of the underlying logics. For illustration, we use an example borrowed from [3].

*Example 1.* The logics in consideration consists of a set of strict rules $\mathcal{R}_0 = \{\rightarrow wr;\ \rightarrow go;\ b \rightarrow \neg hw;\ m \rightarrow hw\}$ and a set of defeasible rules $\mathcal{D} = \{wr \Rightarrow m;\ go \Rightarrow b\}$[1]. The monotonic logic is defined by the set of strict rules $\mathcal{R}_0$. There are 6 arguments[2]:

[1] wr = "John wears something that looks like a a wedding ring", m = "John is married", hw = "John has a wife", go = "John often goes out until late", b = "John is a bachelor".

[2] For a precise definition see defintion 7.

$A_1 : \rightarrow wr,\ \ A_3 : \rightarrow wr \Rightarrow m,\ \ A_5 : \rightarrow wr \Rightarrow m \rightarrow hw.$
$A_2 : \rightarrow go,\ \ A_4 : \rightarrow go \Rightarrow b,\ \ A_6 : \rightarrow go \Rightarrow b \rightarrow \neg hw.$

Arguments $A_3, A_5, A_4, A_6$ are also often written as $A_1 \Rightarrow m$, $A_3 \Rightarrow hw$, $A_2 \Rightarrow b$ and $A_4 \Rightarrow \neg hw$ respectively.

Attack relation: $A_5$ attacks $A_6$ and vice versa. There are no other attacks. Let $att_0 = \{(A_5, A_6)\}$. The grounded extension contains arguments $A_1, A_2, A_3, A_4$. Hence the conclusions of the arguments in the grounded extension are not consistent wrt (monotonic) logic defined by the set of strict rules $\mathcal{R}_0$.

There are two preferred extensions $\{A_1, A_2, A_3, A_4, A_5\}$ and $\{A_1, A_2, A_3, A_4, A_6\}$. The conclusions of the arguments of neither is closed wrt (monotonic) logic defined by $\mathcal{R}$.

In this paper, we are interested in argumentation frameworks whose arguments could be intuitively understood as proofs of some (possibly nonmonotonic) underlying logic over a language $\mathcal{L}$. The monotonic part of the underlying logic is assummed to be represented by a Tarski-like consequence operator $CN(X)$ for set of sentences X over $\mathcal{L}$ such that following properties are satisfied:

1. $X \subseteq CN(X)$
2. $CN(X) = CN(CN(X))$
3. $CN(X) = \bigcup\{CN(Y) \mid Y \subseteq X \text{ and } Y \text{ is finite}\}$
4. A notion of contradictory is introduced by a set $CONTRA$ of subsets of $\mathcal{L}$ ($CONTRA \subseteq 2^{\mathcal{L}}$) such that if $S \in CONTRA$ then each superset of S also belongs to $CONTRA$. A set belonging to $CONTRA$ is said to be *contradictory.*
   The set $CN(\emptyset)$ is not contradictory, i.e. $CN(\emptyset) \notin CONTRA$.

A set of sentences X is said to be **inconsistent** wrt CN if its closure $CN(X)$ is contradictory. X is said to be **consistent** if it is not inconsistent. X is closed if it coincides with its own closure.

The language in example 1 consists of literals whose atoms occur in the (strict and defeasible) rules. The consequence operator $CN_0$ is defined by the set $\mathcal{R}_0$ of strict rules, namely $CN_0(X)$ is the smallest (wrt set inclusion) set of literals satisfying the propositions that $X \subseteq CN_0(X)$ and for any strict rule $r \in \mathcal{R}_0$, if the premises of r are contained in $CN_0(X)$ than the head of r also belongs to $CN_0(X)$. For example $CN_0(\{m\}) = \{wr, go, m, hw\}$ and $CN_0(\{m, b\}) = \{wr, go, m, hw, b, \neg hw\}$ and $CN_0(\emptyset) = \{wr, go\}$. A contradictory set is any set containing a pair of literals $\{l, \neg l\}$. Hence the set $CN_0(\{m, b\})$ is contradictory while the set $\{m, b\}$ is inconsistent wrt $CN_0$.

**Definition 1.** *A logic-based abstract argumentation framework over a language $\mathcal{L}$ is a triple $(AF, CN, Cnl)$ where $AF$ is an abstract argumentation framework, $CN$ is a Tarski-like consequence operator over $\mathcal{L}$ and for each argument $A$, $Cnl(A)$ is the conclusion of $A$.*

For a set S of arguments, $Cnl(S)$ denotes the set of the conclusions of the arguments in S. The Tarski-consequence operator has been employed in [1] to give a general definition of a logic-based argument. In contrast, we use a Tarski-like consequence operator to only specify the logical consequences of arguments without any hint about how an argument is constructed.

From now on until the end of this section, we assume an arbitrary but fixed logic-based abstract argumentation framework $(AF, CN, Cnl)$ and often simply refer to it as an argumentation framework.

**Definition 2.** *Let $(AF, CN, Cnl)$ be a logic-based abstract argumentation framework.*

1. *$AF$ is said to satisfy the* **logical closure-property** *if for each complete extension $E$, $Cnl(E)$ is closed.*
2. *$AF$ is said to satisfy the* **logical consistency-property** *if for each complete extension $E$, $Cnl(E)$ is consistent.*

*Example 2.* (Continuation of example 1) The grounded extension of the argumentation framework in example 1 is $GE = \{A_1, A_2, A_3, A_4\}$. $Cnl(GE) = \{wr, go, m, b\}$ and $CN_0(Cnl(GE)) = Cnl(GE) \cup \{hw, \neg hw\}$. Hence the considered argumentation framework satisfies neither the logical consistency- nor the closure-property.

It turns out that the properties of logical closure and consistency of argumentation is based on an intuitive idea of a base of an argument.

**Definition 3.** *Given a logic-based argumentation framework $(AF, CN, Cnl)$. We say that a set of arguments $S$ is a* **base of an argument** *$A$ if the conclusion of $A$ is a consequence of the conclusions of $S$ (i.e. $Cnl(A) \in CN(Cnl(S))$) and each attack against $A$ is an attack against $S$ and vice versa.*

*We say that a set of arguments $S$ is a* **base of a set of arguments** *$R$ if $Cnl(R) \subseteq CN(Cnl(S))$ and each attack against $R$ is an attack against $S$ and vice versa.*

*We say that an argument $A$ is* **based** *in a set of arguments $S$ if $S$ contains a base of $A$.*

In example 1, though $Cnl(A_5) \in CN_0(Cnl(A_3))$, the set $\{A_3\}$ is not a base of $A_5$ since $A_6$ attacks $A_5$ but $A_6$ does not attack $A_3$. Note that $\emptyset$ is a base of $A_1$ and $A_2$ and the set $\{A_1, A_2\}$.

*Example 3.* Consider a modified version of the example 1 where the set $\mathcal{R}_1$ of strict rules is obtained by adding to $\mathcal{R}_0$ two more strict rules $\neg hw \rightarrow \neg m$; and $hw \rightarrow \neg b$. The corresponding consequence operator is denoted by $CN_1$. There are two more arguments: $A_7 : A_5 \rightarrow \neg b$ and $A_8 : A_6 \rightarrow \neg m$. The attack relation is defined by $att_1 = \{(A_7, A_4), (A_7, A_6), (A_7, A_8), (A_8, A_3), (A_8, A_5), (A_8, A_7)\}$. $\{A_3\}$ is now a base of $A_5$ and $A_7$ and $\{A_5, A_7\}$. It is also easy to see that $\{A_3, A_4\}$ is a base of $\{A_5, A_6\}$.

**Lemma 1.** *Let $E$ be a complete extension of a logic-based argumentation framework LAF. Further let $S$ be a base of a subset of $E$. Then $S \subseteq E$*

**Proof.** As each attack against S is an attack against E, each argument in S is acceptable wrt E. Hence $S \subseteq E$.

The imposition of the closure and consistency-properties on an argumentation framework wrt consequence operator suggests intuitively that if a sentence $\sigma$ follows from the conclusions of a set of arguments S wrt consequence operator $CN$ then there exists an argument A with conclusion $\sigma$ constructed from some arguments in S using the rules of the underlying monotonic logic. In other words, argument A is acceptable wrt S.

**Definition 4.** *We say that a logic-based argumentation framework* $(AF, CN, Cnl)$ *is* **structurally closed** *if for each set of arguments S, for each sentence* $\alpha \in CN(Cnl(S))$ *there exists an argument A based in S such that* $Cnl(A) = \alpha$.

The argumentation framework in example 1 is not structurally closed since although $hw \in CN_0(Cnl(A_3))$ and $A_5$ is the only argument with conclusion $hw$, $A_5$ is not based in $\{A_3\}$ as $A_6$ attacks $A_5$ but $A_6$ does not attack $A_3$. In contrast, the argumentation framwork generated by the set of strict rules $\mathcal{R}_1$ in example 3 together with the defeasible rules in $\mathcal{D}$ is structurally closed.

**Lemma 2.** *Suppose a logic-based abstract argumentation framework* $LAF = (AF, CN, Cnl)$ *is structural closed. Then LAF satisfies the logical closure-property.*

**Proof.** Let E be a complete extension. Let $\alpha \in CN(Cnl(E))$. Therefore from the structural closure, there is an argument A based in E such that $Cnl(A) = \alpha$ such that each attack against A is an attack against E. Because admissibility of E, E attacks each attack against A. Hence A is acceptable wrt E, i.e. $A \in E$. E is hence closed wrt CN.

We say that an argument A is **generated** by a set S of arguments if A is based in a base of S.

In example 3, $\{A_3\}$ is a base of both $A_5$ and $A_7$. Hence $A_7$ is generated by $\{A_5\}$.

We say that a set of arguments S **implicitly attacks** an argument B if there is an argument A generated by S such that A attacks B. S is said to **implictly attack itself** if S implicitly attacks an argument in S.

Consider again example 3. A base of $A_3, A_6$ is $\{A_3, A_4\}$. As $A_7$ is generated by $\{A_3, A_4\}$ and $A_7$ attacks $A_4$, $\{A_3, A_6\}$ implicitly attacks itself.

**Definition 5.** *A logic-based argumentation framework is said to be* **consistency covering** *if for each set of arguments S such that* $Cnl(S)$ *is inconsistent, S implicitly attacks itself.*

In the argumentation framework in 1, for the grounded extension $GE = \{A_1, A_2, A_3, A_4\}$, $Cnl(GE) = \{wr, go, b, m\}$ is inconsistent. I is not difficlt to see that $\{A_3, A_4\}$ is a base of $\{A_1, A_2, A_3, A_4\}$ and the arguments generated by $\{A_3, A_4\}$ are $\{A_1, A_2, A_3, A_4\}$. Hence $GE$ does not implicitly attack itself. The consistency covering property is not satisfied for this framework.

In contrast, in example 3, a base for $S = \{A_1, A_2, A_3, A_4\}$ is also $\{A_3, A_4\}$, and a base of $(A_7)$ is $\{A_3\}$. It follows that $A_7$ is based in $S$. Because $A_7$ attacks $A_4$, S implicitly attacks itself. Overall, the logic-based argumentation framework of this example satisfies the consistency covering property (see section 4 for precise proof).

**Theorem 1.** *Let $LAF$ be a structural closed and consistency covering argumentation framework. Then $LAF$ satisfies both the logical closure- and consistency-properties.*

**Proof.** From lemma 2, we need to prove only the consistency property. Let E be a complete extension of $LAF$. Suppose E is inconsistent. From the consistency covering of $LAF$, it follows that there is an argument A generated by E attacking some argument B in E. Therefore A attacks E. E hence attacks A. Since any base of E is a subset of E (lemma 1), A is based in E. Hence any attack against A is an attack against E. E hence attacks itself. Contradiction.

In the next sections, we present two different argumentation systems slightly generalizing similar systems from the literature to demonstrate how to capture the structural-closedness and consistency covering property.

## 3 Abstract Assumption-Based Argumentation

We assume a language $\mathcal{L}$, a set of assumptions $\mathcal{A} \subseteq \mathcal{L}$, a contrary operator $\overline{(.)} : \mathcal{A} \longrightarrow \mathcal{L}$, and a Tarski-like consequence operator CN with a set CONTRA of contradictory sets. Note that we do not assume that sets containing both $\alpha$ and $\overline{\alpha}$ for an assumption $\alpha \in \mathcal{A}$ belong to CONTRA. In case of normal logic programming [2, 5, 6], CONTRA is empty while for extended logic programming [7] CONTRA contains sets containing pair of literals $\{l, \neg l\}$ where $\neg$ is explicit negation[3].

An argument is a pair $(X, \sigma)$ where X is a finite set of assumption X and $\sigma \in CN(X)$. An argument $(X, \sigma)$ attacks an argument $(Y, \delta)$ if $\sigma = \overline{\alpha}$ for some $\alpha \in Y$.

The just defined logic-based argumentation framework is referred to in the rest of thic section as abstract assumption-based argumentation AAA.

*Example 4.* For illustration, consider the famous bird-fly example. Let CN be the consequence operator defined by the following set of strict rules $\{\rightarrow p;\ \ p \rightarrow b;\ \ p, np \rightarrow \neg f;\ \ b, nb \rightarrow f;\ \ p \rightarrow \neg nb\}$ with $np, nb$ (for normal penguin and normal bird respectively) being assumptions and $\overline{np} = \neg np$ and $\overline{nb} = \neg nb$. Let $A_1 = (\{np\}, \neg f)$, $A_2 = (\{\}, \neg nb)$, $A_3 = (\{nb\}, f)$. $A_2$ attacks $A_3$.

**Definition 6.** *The consequence operator CN is said to be* **assumption-discriminate** *if for each inconsistent set of assumptions $X \subseteq \mathcal{A}$, there is $\alpha \in X$ such that $\overline{\alpha} \in CN(X)$.*

[3] Negation as failure is denoted by not-l.

The argumentation framework in example 4 is assumption-discriminate. For illustration, the set $X = \{np, nb\}$ is inconsistent and $\neg nb \in CN(X)$.

**Lemma 3.** *Suppose $CN$ is assumption-discriminate. Then the abstract assumption-based argumentation framework is structurally closed and consistency-covering.*

**Proof.** We first show the structural closure. Let S be a set of arguments and $\sigma \in CN(Cnl(S))$. Let $X$ be a minimal subset of $Cnl(S)$ such that $\sigma \in CN(X)$. Further let $S_X$ be a minimal set of arguments from S whose conclusions belong to X. Let $A = (Y, \sigma)$ such that $Y = \bigcup\{Z \mid (Z, \delta) \in S_X\}$. It is obvious that A is an argument. We show now that $S_X$ is a base of A. Suppose B is an argument attacking $S_X$. Then there is $(X, \delta) \in S_X$ such that $Cnl(B) = \overline{\alpha}$ for some $\alpha \in X$. Hence B attacks A. Suppose now that B attacks A. Then $Cnl(B) = \overline{\alpha}$ for some $\alpha \in Y$. Hence there is $(X, \delta) \in S_X$ such that $Cnl(B) = \overline{\alpha}$ for some $\alpha \in X$. B therefore attacks $S_X$.

We have proved that that the abstract assumption-based argumentation framework is structurally closed. We show now that it is consistency covering. We need some new notations. For an argument $A = (X, \sigma)$, let $NB(A) = \{(\{\alpha\}, \alpha) \mid \alpha \in X\}$. It is easy to see that $NB(A)$ is a base of A. Similarly, for a set S of arguments, $NB(S) = \bigcup\{NB(A) \mid A \in S\}$ is a base of S.

Let S be a set of arguments such that $Cnl(S)$ is inconsistent. Let $Y = NB(S)$ Hence $Y$ is inconsistent. From the assumption-discrimination of CN, it follows that there is $Z \subseteq Y$ such that $A = (Z, \overline{\alpha})$ is an argument. As $NB(S)$ is a base of S, A is generated by S. Since A attacks each argument having $\alpha$ as an assumption, A attacks S. Hence S implicitly attacks itself.

It follows immediately from lemma 3 and theorem 1

**Theorem 2.** *Suppose $CN$ is assumption-discriminate. Then the abstract assumption-based argumentation framework satisfies both the logical closure- and consistency-properties.*

## 4 Argumentation with Strict and Defeasible Rules

In this section, we apply our results developed in previous section on a defeasible logic similar to the one studied by [3, 8, 10].

The language $\mathcal{L}$ is a set of literals. A set of literals is said to be contradictory if it contains a pair $\{l, \neg l\}$. The set of all contradictory sets is denoted by CONTRA. Arguments are built from strict rules and defeasible rules. The set of strict rules is denoted by $\mathcal{R}_s$ while the set of defeasible rules by $\mathcal{R}_d$. Strict rules are of the form $l_1, \ldots, l_n \longrightarrow h$ and defeasible rules of the form $l_1, \ldots, l_n \Rightarrow h$ where $l_1, \ldots, l_n, h$ are literals from $\mathcal{L}$.

**Definition 7.** *Let $\alpha_1, \ldots, \alpha_n \rightarrow \alpha$ (respectively $\alpha_1, \ldots, \alpha_n \Rightarrow \alpha$) be a strict (respectively defeasible) rule. Further suppose that $A_1, \ldots, A_n$ , $n \geq 0$, are arguments with $\alpha_i = Cnl(A_i)$, $1 \leq i \leq n$. Then $A_1, \ldots, A_n \rightarrow \alpha$ (respectively $A_1, \ldots, A_n \Rightarrow \alpha$ ) is an argument with conclusion $\alpha$.*

*Arguments of the form $A_1, \ldots, A_n \to \alpha$ or $A_1, \ldots, A_n \Rightarrow \alpha$ are also often viewed as proof trees with the root labelled by $\alpha$ and $A_1, \ldots, A_n$ are subtrees whose roots are children of the proof tree root.*

*A strict argument is an argument containing no defeasible rule.*

*B is a subargument of an argument A, denoted by $B \sqsubseteq A$ if $B = A$ or B is a subargument of some $A_i$ if A is of the form $A_1, \ldots, A_n \to \alpha$ or $A_1, \ldots, A_n \Rightarrow \alpha$ .*

The consequence operator $CN_{\mathcal{R}_s}(X)$ (or simply $CN(X)$ if no misunderstanding is possible) is defined by the set of conclusions of strict arguments over the set of rules $\mathcal{R}_s(X) = \mathcal{R}_s \cup \{\to \alpha \,|\, \alpha \in X\}$.

For a strict argument A over a set of rules $\mathcal{R}_s(X)$, the set of premises of A, denoted by $Prem(A)$, is the set of literals from X labelling the leaves of A (viewed as a proof tree).

**Basic arguments** are arguments whose last rule is a defeasible one. For a basic argument B, the last rule of B is denoted by $Lr(B)$.

The following notion of attack is adopted but slightly modified from the ones given in [3, 8–10].

An argument A attacks a argument B if one of the following conditions is satisfied:

1. *(Undercutting)* B is basic and $Cnl(A) = \neg Oj(Lr(B))$ where for a defeasible rule $r$, $Oj(r)$ is a literal denoting that the rule is applicable.
2. *(Rebutting)* B is basic and $Cnl(A) = \neg Cnl(B)$
3. A attacks a basic subargument of B.

An example of argumenttion based on strict and defeasible rules is given in example 3.

**Definition 8.** *The consequence operator $CN_{\mathcal{R}_s}$ is said to be* **discriminate** *if for each inconsistent set X of literals, there is a literal $\sigma \in X$ such that $\neg\sigma \in CN_{\mathcal{R}_s}(X)$ holds.*

**Theorem 3.** *Let $\mathcal{R}_s, \mathcal{R}_d$ be sets of strict and defeasible rules respetively. Let $\mathcal{AR}$ be the arguments built from these rules and att be the associated attack relation and $AF = (\mathcal{AR}, att)$. Then the logic-based argumentation framework $LAF = (AF, CN_{\mathcal{R}_s}, Cnl)$ is structurally closed and consistency covering if the consequence operator $CN_{\mathcal{R}_s}$ is discriminate.*

**Proof.** We first show that $LAF$ is structurally closed. Suppose now that $\sigma \in CN(Cnl(S))$. Let $X$ be a minimal subset of $Cnl(S)$ such that $\sigma \in CN(X)$. Hence there is a strict argument $A_0$ over $\mathcal{R}_s(X)$ with conclusion $\sigma$. Further let $S_X$ be a minimal set of arguments from S whose conclusions belong to X. Let A be the argument obtained by replacing each leave in $A_0$ (viewed as a proof tree) labelled by a literal $\alpha$ from X by an argument with conclusion $\alpha$ from $S_X$. It is obvious that the conclusion of A is $\sigma$. We show now that $S_X$ is a base of A. Suppose B is an argument attacking $S_X$. Then it is obvious that B attacks A. Suppose now that B attacks A. Then B attacks a basic argument of A. Since $A_0$ is a strict argument over $\mathcal{R}_s(X)$, B must attacks a basic subargument of some argument in $S_X$. Hence B attacks $S_X$.

We have proved that that the logic-based argumentation framework LAF is structurally closed. We show now that it is consistency covering. We need some new notations.

Among the bases of arguments, a special kind of base called normal base plays a key role. The **normal base** of argument A is defined by $NB(A) = \{B \mid B$ is a basic subargument of A and for each argument C, if $C \neq B$ and $B \sqsubseteq C \sqsubseteq A$ then C is strict $\}$. For a set S of arguments, $NB(S)$ is the union of the normal bases of elements of S. The following lemma shows that a normal base is indeed a base.

**Lemma 4.** *For any argument A, $NB(A)$ is a base of A.*

**Proof of Lemma 4.** From the definition of NB(A), it is obvious that $Cnl(A) \in CN_{\mathcal{R}_f}(NB(A))$. It is also obvious that each argument attacking $NB(A)$ also attacking A. Let B be an argument attacking A. From the definition of attack, B attacks a basic subargument C of A. From the definition of $NB(A)$, there is an argument $C' \in NB(A)$ such that $C \sqsubseteq C'$. Hence B attacks $C'$ and therefore $NB(A)$.

**Continuation of Proof of Theorem 3.** It is obvious that NB(S) is also a base of set of arguments S. Suppose now that $Cnl(S)$ is inconsistent. It follows immediately that the set $Cnl(NB(S))$ is also inconsistent. Let $X = Cnl(NB(S))$. From the definition 8, it follows that there is $\alpha \in X$ such that $\neg\alpha \in CN_{\mathcal{R}_s}(X)$. Since $\alpha \in X$, there is a basic argument $B \in NB(S)$ with conclusion $\alpha$. From the structural closure of LAF, there is an argument A with conclusion $\neg\alpha$ based in $NB(S)$. Hence A is generated by S and A attacks B. As $B \in NB(S)$, there is $C \in S$ s.t. $B \in NB(C)$. Hence A attacks C. Therefore S implicitly attacks itself.

It follows immediately from theorem 1

**Theorem 4.** *Let $\mathcal{R}_s, \mathcal{R}_d$ be sets of strict and defeasible rules respectively. Then the associated logic-based abstract argumentation framework $LAF = (AF, CN_{\mathcal{R}_s})$ satisfies the logical closure- and consistency-properties if $CN_{\mathcal{R}_s}$ is discriminate.*

We next show that theorem 4 generalizes the results in Caminada and Amgoud [3], Prakken [8].

A set of strict rules $\mathcal{R}_s$ is said to be closed under transposition if for each rule $\alpha_1, \ldots, \alpha_n \rightarrow \sigma$ in $\mathcal{R}_s$, all the rules of the form
$\alpha_1, \ldots, \alpha_{i-1}, \neg\sigma, \alpha_{i+1}, \alpha_n \rightarrow \neg\alpha_i$ also belong to $\mathcal{R}_s$.

A set of strict rules $\mathcal{R}_s$ is said to satisfy the contraposition-property if for each set of literals X, for each argument A (with conclusion $\sigma$) wrt $\mathcal{R}_s(X)$ and for each $\alpha \in Prem(A)$, there is an argument whose premises is $Prem(A) - \{\alpha\} \cup \{\neg\sigma\}$ and conclusion is $\neg\alpha$.

**Theorem 5.** *$CN_{\mathcal{R}_s}$ is discriminate if the set of strict rules $\mathcal{R}_s$ is closed under transposition or satisfies the contraposition-property.*

**Proof.** We first prove the following assertion.
**Assertion.** Let A be a strict argument over $\mathcal{R}_s(X)$ whose conclusion is $\sigma$ and $\emptyset \neq Prem(A) \subseteq X$. Then there is an argument B with premises in $Prem(A) \cup \{\neg\sigma\}$ and conclusion $\neg\alpha$ for some $\alpha \in Prem(A)$.
**Proof of Assertion.** The assertion holds immediately if $\mathcal{R}_s$ satisfies the contraposition-property.

Supoose that $\mathcal{R}_s$ is closed under transposition. We prove by induction on the height of A (as a proof tree).

If the heitght of A is 1, the theorem is obvious.

Suppose A is of the form $A_1, \ldots, A_n \rightarrow \sigma$ where $Cnl(A_i) = \alpha_i$. Suppose $Prem(A_n) \neq \emptyset$. From the closure under transposition, $\alpha_1, \ldots, \alpha_{n-1}, \neg\sigma \rightarrow \neg\alpha_n$ also belongs to $\mathcal{R}_s$. Let $A_0$ be the argument $A_1, \ldots, A_{n-1}, \neg\sigma \rightarrow \neg\alpha_n$.

From the induction hypothesis, there is a tree $C$ whose premises in $Prem(A_n)$ $\cup \{\neg\alpha_n\}$ and whose conclusion is $\neg\alpha$ for some $\alpha \in Prem(A_n)$.

Let $B$ be the tree obtained from $C$ by replacing each occurence of premise $\neg\alpha_n$ by the argument $A_0$. It is clear that $Prem(B) \subseteq Prem(A) \cup \{\neg\sigma\}$ and $Cnl(B) = \neg\alpha$. Note $\alpha \in Prem(A)$.

**Continuation of Proof of Theorem 5.** Let X be an inconsistent set of literals. Hence there are two arguments $A_0, A_1$ with premises in X and conclusions $\sigma, \neg\sigma$ respectively. From the above assertion, it follows that there exists an argument $B$ with conclusion $\neg\alpha$ for some $\alpha \in Prem(A_0)$ and $Prem(B) \subseteq Prem(A_0) \cup \{\neg\sigma\}$. Let A be the argument obtained by replacing leaves labelled by $\neg\sigma$ in $B$ by trees $A_1$. It is clear that $Prem(A) \subseteq X$ and the conclusion of A is labelled by $\neg\alpha$ for some $\alpha \in X$.

In [11], a generalized assumption-based argumentation framework combining assumption-based argumentation with strict and defeasible rule has been proposed and shown to be both consistent and closed. It would be interesting to see how this generalized framework behaves wrt the principles of structural closure and consistency covering.

## 5 Conclusion

In general, an argumentation system could take a large number of arguments, many of them could be redundant. For efficiency, many of the redundant arguments should be avoided. In [4], principles for dealing with redundant arguments have been studied. It would be interesting to see how such principles could be integrated with the concepts of structural closure and consistency covering for modeling practical argumentation.

## Acknowledgements

Many thanks to Marcello Balduccini and Tran Cao Son for the invitation to contribute to this event.

## References

1. Amgoud, L., Besnard, P.: Bridging the gap between abstract argumentation systems and logic. In: Godo, L., Pugliese, A. (eds.) SUM 2009. LNCS, vol. 5785, pp. 12–27. Springer, Heidelberg (2009)
2. Bondarenko, A., Dung, P.M., Kowalski, R., Toni, F.: An abstract argumentation-theoretic approach to default reasoning. Artificial Intelligence 93, 63–101 (1997)
3. Caminada, M., Amgoud, L.: On the evaluation of argumentation formalisms. Artificial Intelligence 171, 286–310 (2007)
4. Dung, P.M., Toni, F., Mancarella, P.: Some design guidelines fo practical argumentation systems. In: Third International Conference on Computational Models of Argument, Italy (2010)
5. Dung, P.M.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. Artificial Intelligence 77, 321–357 (1995)
6. Gelfond, M., Lifschitz, V.: The stable model semantics of logic programs. In: Proc. of ICLP 1988. MIT Press, Cambridge (1988)
7. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: Proc. of ICLP 1990. MIT Press, Cambridge (1990)
8. Prakken, H.: An abstract framework for argumentation with structured arguments. Journal of Argumentation and Computation 1 (2010)
9. Pollock, J.: Defeasible Reasoning. Cognitive Science 11, 481–518 (1987)
10. Prakken, H., Sartor, G.: Argument-based extended logic programming with defeasible priorities. J. of Applied Non-Classical Logics 7, 25–75 (1997)
11. Toni, F.: Assumption-based argumentation for closed and consistent defeasible reasoning. In: Satoh, K., Inokuchi, A., Nagao, K., Kawamura, T. (eds.) JSAI 2007. LNCS (LNAI), vol. 4914, pp. 390–402. Springer, Heidelberg (2008)

# Manifold Answer-Set Programs and Their Applications*

Wolfgang Faber[1] and Stefan Woltran[2]

[1] University of Calabria, Italy
wf@wfaber.com
[2] Vienna University of Technology, Austria
woltran@dbai.tuwien.ac.at

**Abstract.** In answer-set programming (ASP), the main focus usually is on computing answer sets which correspond to solutions to the problem represented by a logic program. Simple reasoning over answer sets is sometimes supported by ASP systems (usually in the form of computing brave or cautious consequences), but slightly more involved reasoning problems require external postprocessing. Generally speaking, it is often desirable to use (a subset of) brave or cautious consequences of a program $P_1$ as input to another program $P_2$ in order to provide the desired solutions to the problem to be solved. In practice, the evaluation of the program $P_1$ currently has to be decoupled from the evaluation of $P_2$ using an intermediate step which collects the desired consequences of $P_1$ and provides them as input to $P_2$. In this work, we present a novel method for representing such a procedure within a *single* program, and thus within the realm of ASP itself. Our technique relies on rewriting $P_1$ into a so-called *manifold program*, which allows for accessing all desired consequences of $P_1$ within a single answer set. Then, this manifold program can be evaluated jointly with $P_2$ avoiding any intermediate computation step. For determining the consequences within the manifold program we use *weak constraints*, which is strongly motivated by complexity considerations. As applications, we present encodings for computing the ideal extension of an abstract argumentation framework and for computing world views of a particular class of epistemic specifications.

## 1 Introduction

In the last decade, *Answer Set Programming* (ASP) [1,2], also known as A-Prolog [3,4], has emerged as a declarative programming paradigm. ASP is well suited for modelling and solving problems which involve common-sense reasoning, and has been fruitfully applied to a wide variety of applications including diagnosis (e.g. [5]), data integration (e.g. [6]), configuration (e.g. [7]), and many others. Moreover, the efficiency of the latest tools for processing ASP programs (so-called ASP solvers) reached a state that makes them applicable for problems of practical importance [8]. The basic idea of ASP

* This work was supported by the Vienna Science and Technology Fund (WWTF), grant ICT08-028, and by M.I.U.R. within the Italia-Austria internazionalization project "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione" and the PRIN project LoDeN. A preliminary version of this paper appeared in the Proceedings of the the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), pp. 115–128, Springer LNAI 5753, 2009.

is to compute answer sets of a logic program from which the solutions of the problem encoded by the program can be obtained.

However, frequently one is interested not only in the solutions per se, but rather in reasoning tasks that have to take some or even all solutions into account. As an example, consider the problem of database repair, in which a given database instance does not satisfy some of the constraints imposed in the database. One can attempt to modify the data in order to obtain a consistent database by changing as little as possible. This will in general yield multiple possibilities and can be encoded conveniently using ASP (see, e.g., [9]). However, usually one is not interested in the repairs themselves, but in the data which is present in *all* repairs. For the ASP encoding, this means that one is interested in the elements which occur in all answer sets; these are also known as *cautious consequences*. Indeed, ASP systems provide special interfaces for computing cautious consequences by means of query answering. But sometimes one has to do more, such as answering a complex query over the cautious consequences (not to be confused with complex queries over answer sets). So far, ASP solvers provide no support for such tasks. Instead, computations like this have to be done outside ASP systems, which hampers usability and limits the potential of ASP.

In this work, we tackle this limitation by providing a technique, which transforms an ASP program $P$ into a *manifold program* $M_P$ which we use to identify all consequences of a certain type (we consider here the well-known concepts of brave and cautious consequence, but also definite consequence [10]) within a *single* answer set. The main advantage of the manifold approach is that the resulting program can be extended by additional rules representing a query over the brave (or cautious, definite) consequences of the original program $P$, thereby using ASP itself for this additional reasoning. In order to identify the consequences, we use *weak constraints* [11], which are supported by the ASP-solver DLV [12]. Weak constraints have been introduced to prefer a certain subset of answer sets via penalization. Their use for computing consequences is justified by a complexity-theoretic argument: One can show that computing consequences is complete for the complexity classes $\mathrm{FP}_{||}^{\mathrm{NP}}$ or $\mathrm{FP}_{||}^{\Sigma_2^P}$ (depending on the presence of disjunction), for which also computing answer sets for programs with weak constraints is complete[1], which means that an equivalent compact ASP program without these extra constructs does not exist, unless the polynomial hierarchy collapses. In principle, other preferential constructs similar to weak constraints could be used as well for our purposes, as long as they meet these complexity requirements.

We discuss three particular applications of the manifold approach. First, we specify an encoding which decides the SAT-related *unique minimal model problem*, which is closely related to closed-world reasoning [13]. The second problem stems from the area of argumentation (cf. [14] for an overview) and concerns the computation of the ideal extension [15] of an argumentation framework. For both problems we make use of manifold programs of well-known encodings (computing all models of a CNF-formula

[1] The first of these results is fairly easy to see, for the second, it was shown [11] that the related decision problem is complete for the class $\Theta_2^P$ or $\Theta_3^P$, from which the $\mathrm{FP}_{||}^{\mathrm{NP}}$ and $\mathrm{FP}_{||}^{\Sigma_2^P}$ results can be obtained. Also note that frequently cited NP, $\Sigma_2^P$, and co-NP, $\Pi_2^P$ completeness results hold for brave and cautious query answering, respectively, but not for computing brave and cautious consequences.

for the former application, computing all admissible extensions of an argumentation framework for the latter) in order to compute consequences. Extensions by a few more rules then directly provide the desired solutions, requiring little effort in total. As a final application, we consider an encoding for (a certain subclass of) epistemic specifications as introduced by Gelfond [16]. In a nutshell, these specifications are extensions of ASP programs, which may include modal atoms to allow for reasoning over answer-sets within the object language, and thus are closely related to some of the ideas we present here. Epistemic specifications (already introduced in 1991 [17]) have received increasing interest only over the last years (see, e.g. [18,19,20,21]) but nowadays get more and more recognized as an highly expressive and important extension of standard answer-set programming.

**Organization and Main Results.** After introducing the necessary background in the next section, we

- introduce in Section 3 the concept of a manifold program for rewriting propositional programs in such a way that all brave (resp. cautious, definite) consequences of the original program are collected into a single answer set;
- lift the results to the non-ground case (Section 4); and
- present applications for our technique in Section 5. In particular, we provide ASP encodings for computing the ideal extension of an argumentation framework and for computing world views of a particular class of epistemic specifications.

The paper concludes with a brief discussion of related and further work.

## 2 Preliminaries

In this section, we review the basic syntax and semantics of ASP with weak constraints, following [12], to which we refer for a more detailed definition.

An *atom* is an expression $p(t_1, \ldots, t_n)$, where $p$ is a *predicate* of arity $\alpha(p) = n \geq 0$ and each $t_i$ is either a variable or a constant. A *literal*[2] is either an atom $a$ or its negation $\text{not } a$.

A *(disjunctive) rule* $r$ is of the form

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \text{:-}\ b_1, \ldots, b_k, \ \text{not } b_{k+1}, \ldots, \ \text{not } b_m$$

with $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$, and where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms.

The *head* of $r$ is the set $H(r) = \{a_1, \ldots, a_n\}$, and the *body* of $r$ is the set $B(r) = \{b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m\}$. Furthermore, $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. We will sometimes denote a rule $r$ as $H(r) \text{ :- } B(r)$.

A *weak constraint* [11] is an expression $wc$ of the form

$$:\sim \ b_1, \ldots, b_k, \ \text{not } b_{k+1}, \ldots, \ \text{not } b_m. \ [w : l]$$

where $m \geq k \geq 0$ and $b_1, \ldots, b_m$ are literals, while $\mathit{weight}(wc) = w$ (the *weight*) and $l$ (the *level*) are positive integer constants or variables. For convenience, $w$ and/or $l$ may

[2] For keeping the framework simple, we do not consider strong negation in this paper. However, the formalism can easily be adapted to deal with them.

be omitted and are set to 1 in this case. The sets $B(wc)$, $B^+(wc)$, and $B^-(wc)$ are defined as for rules. We will sometimes denote a weak constraint $wc$ as $:\sim B(wc)$.

A *program* $P$ is a finite set of rules and weak constraints. We will often use semicolons for separating rules and weak constraints in order to avoid ambiguities. With $Rules(P)$ we denote the set of rules in $P$ and $WC(P)$ denotes the set of weak constraints in $P$. $w^P_{max}$ and $l^P_{max}$ denote the maximum weight and maximum level over $WC(P)$, respectively. A program (rule, atom) is *propositional* or *ground* if it does not contain variables. A program is called *strong* if $WC(P) = \emptyset$, and *weak* otherwise.

For any program $P$, let $U_P$ be the set of all constants appearing in $P$ (if no constant appears in $P$, an arbitrary constant is added to $U_P$); let $B_P$ be the set of all ground literals constructible from the predicate symbols appearing in $P$ and the constants of $U_P$; and let $Ground(P)$ be the set of rules and weak constraints obtained by applying, to each rule and weak constraint in $P$ all possible substitutions from the variables in $P$ to elements of $U_P$. $U_P$ is usually called the *Herbrand Universe* of $P$ and $B_P$ the *Herbrand Base* of $P$.

A ground rule $r$ is *satisfied* by a set $I$ of ground atoms iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. $I$ satisfies a ground program $P$, if each $r \in P$ is satisfied by $I$. For non-ground $P$, $I$ satisfies $P$ iff $I$ satisfies $Rules(Ground(P))$. A ground weak constraint $wc$ is *violated* by $I$, iff $B^+(wc) \subseteq I$ and $B^-(wc) \cap I = \emptyset$; it is satisfied otherwise.

Following [22], a set $I \subseteq B_P$ of atoms is an *answer set* for a strong program $P$ iff it is a subset-minimal set that satisfies the *reduct*

$$P^I = \{H(r) \text{:-}\, B^+(r) \mid I \cap B^-(r) = \emptyset, r \in Ground(P)\}.$$

A set of atoms $I \subseteq B_P$ is an *answer set* for a weak program $P$ iff $I$ is an answer set of $Rules(P)$ and $H^{Ground(P)}(I)$ is minimal among all the answer sets of $Rules(P)$, where the penalization function $H^P(I)$ for weak constraint violation of a ground program $P$ is defined as follows:

$$\begin{array}{ll} H^P(I) = \sum_{i=1}^{l^P_{max}} \left(f_P(i) \cdot \sum_{w \in N^P_i(I)} weight(w)\right) & \\ f_P(1) \;\; = 1, \;\text{ and} & \\ f_P(n) \;\; = f_P(n-1) \cdot |WC(P)| \cdot w^P_{max} + 1 \;\text{ for } n > 1. & \end{array}$$

where $N^P_i(I)$ denotes the set of weak constraints of $P$ in level $i$ violated by $I$. For any program $P$, we denote the set of its answer sets by $AS(P)$. In this paper, we use only weak constraints with weight and level 1, for which $H^{Ground(P)}(I)$ amounts to the number of weak constraints violated in $I$.

A ground atom $a$ is a *brave* (sometimes also called credulous or possible) consequence of a program $P$, denoted $P \models_b a$, if $a \in A$ holds for at least one $A \in AS(P)$. A ground atom $a$ is a *cautious* (sometimes also called skeptical or certain) consequence of a program $P$, denoted $P \models_c a$, if $a \in A$ holds for all $A \in AS(P)$. A ground atom $a$ is a *definite* consequence [10] of a program $P$, denoted $P \models_d a$, if $a$ is a cautious consequence of $P$ and $AS(P) \neq \emptyset$. The sets of all brave, cautious, definite consequences of a program $P$ are denoted as $BC(P)$, $CC(P)$, $DC(P)$, respectively.

## 3 Propositional Manifold Programs

In this section, we present a translation which essentially creates a copy of a given strong propositional program for each of (resp. for a subset of) its atoms. Thus, we require several copies of the alphabet used by the given program.

**Definition 1.** *Given a set $I$ of literals, a collection $\mathcal{I}$ of sets of literals, and an atom $a$, define $I^a = \{p^a \mid$ atom $p \in I\} \cup \{\mathrm{not}\ p^a \mid \mathrm{not}\ p \in I\}$ and $\mathcal{I}^a = \{I^a \mid I \in \mathcal{I}\}$.*

The actual transformation to a manifold is given in the next definition. We copy a given program $P$ for each atom $a$ in a given set $S$, whereby the transformation guarantees the existence of an answer set by enabling the copies conditionally.

**Definition 2.** *For a strong propositional program $P$ and $S \subseteq B_P$, define its* manifold *(w.r.t. $S$) as*

$$P_S^{tr} = \bigcup_{r \in P} \{H(r)^a \mathrel{:\!-} \{c\} \cup B(r)^a \mid a \in S\} \cup \{c \mathrel{:\!-} \mathrm{not}\ i \ ;\ \ i \mathrel{:\!-} \mathrm{not}\ c\}.$$

*We assume $B_P \cap B_{P_S^{tr}} = \emptyset$, that is, all symbols in $P_S^{tr}$ are assumed to be fresh.*

*Example 1.* Consider

$$\Phi = \{p \vee q \mathrel{:\!-}\ \ ;\ \ r \mathrel{:\!-} p\ ;\ \ r \mathrel{:\!-} q\}$$

for which we have $AS(\Phi) = \{\{p, r\}, \{q, r\}\}$, and thus $BC(\Phi) = \{p, q, r\}$ and $CC(\Phi) = DC(\Phi) = \{r\}$. When forming the manifold for $B_\Phi = \{p, q, r\}$, we obtain

$$\Phi_{B_\Phi}^{tr} = \left\{ \begin{array}{llll} p^p \vee q^p \mathrel{:\!-} c\ ; & r^p \mathrel{:\!-} c, p^p\ ; & r^p \mathrel{:\!-} c, q^p\ ; & c \mathrel{:\!-} \mathrm{not}\ i\ ; \\ p^q \vee q^q \mathrel{:\!-} c\ ; & r^q \mathrel{:\!-} c, p^q\ ; & r^q \mathrel{:\!-} c, q^q\ ; & i \mathrel{:\!-} \mathrm{not}\ c\ ; \\ p^r \vee q^r \mathrel{:\!-} c\ ; & r^r \mathrel{:\!-} c, p^r\ ; & r^r \mathrel{:\!-} c, q^r & \end{array} \right\}$$

Note that given a strong program $P$ and $S \subseteq B_P$, the construction of $P_S^{tr}$ can be done in polynomial time (w.r.t. the size of $P$). The answer sets of the transformed program consist of all combinations (of size $|S|$) of answer sets of the original program (augmented by $c$) plus the special answer set $\{i\}$ which we shall use to indicate inconsistency of $P$.

**Proposition 1.** *For a strong propositional program $P$ and a set $S \subseteq B_P$, $AS(P_S^{tr}) = A \cup \{\{i\}\}$, where*

$$A = \{\bigcup_{i=1}^{|S|} A_i \cup \{c\} \mid \langle A_1, \ldots, A_{|S|} \rangle \in \prod_{a \in S} AS(P)^a\}.$$

Note that $\prod$ denotes the Cartesian product in Proposition 1.

*Example 2.* For $\Phi$ of Example 1, we obtain that $AS(\Phi_{B_\Phi}^{tr})$ consists of $\{i\}$ plus (copies of $\{q, r\}$ are underlined for readability)

$$\begin{array}{l} \{c, p^p, r^p, p^q, r^q, p^r, r^r\}, \{c, \underline{q^p, r^p}, \underline{q^q, r^q}, \underline{q^r, r^r}\}, \\ \{c, \underline{q^p, r^p}, p^q, r^q, p^r, r^r\}, \{c, p^p, r^p, \underline{q^q, r^q}, p^r, r^r\}, \{c, p^p, r^p, p^q, r^q, \underline{q^r, r^r}\}, \\ \{c, \underline{q^p, r^p}, \underline{q^q, r^q}, p^r, r^r\}, \{c, \underline{q^p, r^p}, p^q, r^q, \underline{q^r, r^r}\}, \{c, p^p, r^p, \underline{q^q, r^q}, \underline{q^r, r^r}\}. \end{array}$$

Using this transformation, each answer set encodes an association of an atom with some answer set of the original program. If an atom $a$ is a brave consequence of the original program, then a witnessing answer set exists, which contains the atom $a^a$. The idea is now to prefer those atom-answer set associations where the answer set is a witness. We do this by means of weak constraints and penalize each association where the atom is not in the associated answer set, that is, where $a^a$ is not in the answer set of the transformed program. Doing this for each atom means that an optimal answer set will not contain $a^a$ only if there is no answer set of the original program that contains $a$, so each $a^a$ contained in an optimal answer set is a brave consequence of the original program.

**Definition 3.** *Given a strong propositional program $P$ and $S \subseteq B_P$, let*

$$P_S^{bc} = P_S^{tr} \cup \{:\sim \text{not } a^a \mid a \in S\} \cup \{:\sim i\}$$

Observe that all weak constraints are violated in the special answer set $\{i\}$, while in the answer set $\{c\}$ (which occurs if the original program has an empty answer set) all but $:\sim i$ are violated.

**Proposition 2.** *Given a strong propositional program $P$ and $S \subseteq B_P$, for any $A \in AS(P_S^{bc})$, $\{a \mid a^a \in A\} = BC(P) \cap S$.*

This result would also hold without including $:\sim i$ in $P_S^{bc}$. It has been included for clarity and for making the encoding more uniform with respect to the encoding for definite consequences, which will be presented below.

*Example 3.* For the program $\Phi$ as given Example 1,

$$\Phi_{B_\Phi}^{bc} = \Phi_{B_\Phi}^{tr} \cup \{:\sim \text{not } p^p \ ; \ :\sim \text{not } q^q \ ; \ :\sim \text{not } r^r \ ; \ :\sim i\}.$$

We obtain that $AS(\Phi_{B_\Phi}^{bc}) = \{A_1, A_2\}$, where

$$A_1 = \{c, p^p, r^p, q^q, r^q, p^r, r^r\};$$
$$A_2 = \{c, p^p, r^p, q^q, r^q, q^r, r^r\},$$

as these two answer sets are the only ones that violate no weak constraint. We can observe that $\{a \mid a^a \in A_1\} = \{a \mid a^a \in A_2\} = \{p, q, r\} = BC(\Phi)$.

Concerning cautious consequences, we first observe that if a program is inconsistent (in the sense that it does not have any answer set), each atom is a cautious consequence. But if $P$ is inconsistent, then $P_S^{tr}$ will have only $\{i\}$ as an answer set, so we will need to find a suitable modification in order to deal with this in the correct way. In fact, we can use a similar approach as for brave consequences, but penalize those associations where an atom is contained in its associated answer set. Any optimal answer set will thus contain $a^a$ for an atom only if $a$ is contained in each answer set. If an answer set containing $i$ exists, it is augmented by all atoms $a^a$, which also causes all weak constraints to be violated.

**Definition 4.** *Given a strong propositional program $P$ and $S \subseteq B_P$, let*

$$P_S^{cc} = P_S^{tr} \cup \{:\sim a^a \mid a \in S\} \cup \{a^a :- i \mid a \in S\} \cup \{:\sim i\}$$

**Proposition 3.** *Given a strong propositional program $P$ and $S \subseteq B_P$, for any $A \in AS(P_S^{cc})$, $\{a \mid a^a \in A\} = CC(P) \cap S$.*

Similar to $P_S^{bc}$, this result also holds without including $:\sim i$.

*Example 4.* Recall program $\Phi$ from Example 1. We have

$$\Phi_{B_\Phi}^{cc} = \Phi_{B_\Phi}^{tr} \cup \{:\sim p^p \ ; \ :\sim q^q \ ; \ :\sim r^r \ ; \ p^p :- i \ ; \ q^q :- i \ ; \ r^r :- i \ ; \ :\sim i\}.$$

We obtain that $AS(\Phi_{B_\Phi}^{cc}) = \{A_3, A_4\}$, where

$$A_3 = \{c, q^p, r^p, p^q, r^q, p^r, r^r\};$$
$$A_4 = \{c, q^p, r^p, p^q, r^q, q^r, r^r\},$$

as these two answer sets are the only ones that violate only one weak constraint, namely $:\sim r^r$. We observe that $\{a \mid a^a \in A_3\} = \{a \mid a^a \in A_4\} = \{r\} = CC(\Phi)$.

We next consider the notion of definite consequences. Different to cautious consequences, we do not add the annotated atoms to the answer set containing $i$. However, this answer set should never be among the optimal ones unless it is the only one. Therefore we inflate it by new atoms $i^a$, all of which incur a penalty. This guarantees that this answer set will incur a higher penalty ($|B_P| + 1$) than any other ($\leq |B_P|$).

**Definition 5.** *Given a strong propositional program $P$ and $S \subseteq B_P$, let*

$$P_S^{dc} = P_S^{tr} \cup \{:\sim a^a;\ i^a :- i;\ :\sim i^a \mid a \in S\} \cup \{:\sim i\}$$

**Proposition 4.** *Given a strong propositional program $P$ and $S \subseteq B_P$, for any $A \in AS(P_S^{dc})$, $\{a \mid a^a \in A\} = DC(P) \cap S$.*

*Example 5.* Recall program $\Phi$ from Example 1. We have

$$\Phi_{B_\Phi}^{dc} = \Phi_{B_\Phi}^{tr} \cup \{:\sim p^p \ ; \ :\sim q^q \ ; \ :\sim r^r \ ;$$
$$i^p :- i \ ; \ i^q :- i \ ; \ i^r :- i \ ; \ :\sim i^p \ ; \ :\sim i^q \ ; \ :\sim i^r \ ; \ :\sim i\}.$$

As in Example 4, $A_3$ and $A_4$ are the only ones that violate only one weak constraint, namely $:\sim r^r$, and thus are the answer sets of $\Phi_{B_\Phi}^{dc}$.

Obviously, one can compute all brave, cautious, or definite consequences of a program by choosing $S = B_P$. We also note that the programs from Definitions 3, 4 and 5 yield multiple answer sets. However each of these yields the same atoms $a^a$, so it is sufficient to compute one of these. The programs could be extended in order to admit only one answer set by suitably penalizing all atoms $a^b$ ($a \neq b$). To avoid interference with the weak constraints already used, these additional weak constraints would have to pertain to a different level.

## 4 Non-ground Manifold Programs

We now generalize the techniques introduced in Section 3 to non-ground strong programs. The first step in Section 3 was to define the notion of annotation. There, we annotated propositional atoms with propositional atoms. Also in the non-ground case, we want to annotate atoms with atoms in some way, but it is not immediately clear what kind of atoms should be used for annotations — ground atoms or non-ground atoms?

The first thought would be to annotate using ground atoms, since after all the goal is to produce a copy of the program for each possible ground consequence. This would amount mean annotating each predicate (and thus also each atom) with ground atoms of some subset of the Herbrand Base. For example, annotating the rule $p(X,Y) \text{:-}\ q(X,Y)$ with the set $\{r(a), r(b)\}$ would yield the annotated rules $p^{r(a)}(X,Y) \text{:-}\ q^{r(a)}(X,Y)$ and $p^{r(b)}(X,Y) \text{:-}\ q^{r(b)}(X,Y)$. The tacit assumption here is that $r(a)$ and $r(b)$ are the only two ground instances of predicate $r$ which are of interest.

Since we want to keep our description as general as possible, we assume annotation using the full Herbrand Base. In this scenario it makes sense to annotate with non-ground atoms, in order to ease readability and reduce the size of the (non-ground) manifold program. In particular, the arguments of these non-ground atoms should be mutually different variables, in order to represent all possible ground instances of the atom. The idea is that we can then use the standard grounding definition also on the annotations.

In the example given earlier, we would annotate using $r(Z)$. In order to be able to fall back on the regular grounding defined for non-annotated programs, we will annotate using only the predicate $r$ and extend the arguments of $p$, yielding the rule $\mathrm{d}_p^r(X,Y,Z) \text{:-}\ \mathrm{d}_q^r(X,Y,Z)$ (we use predicate symbols $\mathrm{d}_p^r$ and $\mathrm{d}_q^r$ rather than $p^r$ and $q^r$ just for pointing out the difference between annotation by predicates versus annotation by ground atoms).

This notation is quite general, as it can restrict the annotations to ground atoms of special interest by adding appropriate atoms to the rule body. In our example, this amounts to writing $p^r(X,Y,Z) \text{:-}\ q^r(X,Y,Z), \mathit{rdom}(Z)$ where the predicate $\mathit{rdom}$ identifies the instances of $r$ for which annotations should be produced. In the following, recall that $\alpha(p)$ denotes the arity of a predicate $p$.

**Definition 6.** *Given an atom $a = p(t_1,\ldots,t_n)$ and a predicate $q$, let $a_q^{tr}$ be the atom $\mathrm{d}_p^q(t_1,\ldots,t_n,X_1,\ldots,X_{\alpha(q)})$ where $X_1,\ldots,X_{\alpha(q)}$ are fresh variables and $\mathrm{d}_p^q$ is a new predicate symbol with $\alpha(\mathrm{d}_p^q) = \alpha(p)+\alpha(q)$. Furthermore, given a set $\mathcal{L}$ of literals, and a predicate $q$, let $\mathcal{L}_q^{tr}$ be $\{a_q^{tr} \mid \text{atom } a \in \mathcal{L}\} \cup \{\text{not } a_q^{tr} \mid \text{not } a \in \mathcal{L}\}$.*

Note that we assume that even though the variables $X_1,\ldots,X_{\alpha(q)}$ are fresh, they will be the same for each occurrence of $a_q^{tr}$. We define the manifold program in analogy to Definition 2, the only difference being the different way of annotating.

**Definition 7.** *Given a strong program $P$ and a set $S$ of predicates, define its* manifold *as*

$$P_S^{tr} = \bigcup_{r \in P} \{H(r)_q^{tr} \text{:-}\ \{c\} \cup B(r)_q^{tr} \mid q \in S\} \cup \{c \text{:-}\ \text{not } i \ ; \ i \text{:-}\ \text{not } c\}.$$

*Example 6.* Consider program

$$\Psi = \{p(X) \vee q(X) \mathrel{:\!-} r(X) \ ; \ r(a) \mathrel{:\!-} \ ; \ r(b) \mathrel{:\!-} \}$$

for which

$$\begin{aligned} AS(\Psi) = \{ \ & \{p(a), p(b), r(a), r(b)\}, \\ & \{p(a), q(b), r(a), r(b)\}, \\ & \{q(a), p(b), r(a), r(b)\}, \\ & \{q(a), q(b), r(a), r(b)\}\}. \end{aligned}$$

Hence, we have $BC(\Psi) = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$ and moreover $CC(\Psi) = DC(\Psi) = \{r(a), r(b)\}$. Forming the manifold for $S = \{p\}$, we obtain

$$\Psi_S^{tr} = \left\{ \begin{array}{l} \mathrm{d}_p^p(X, X_1) \vee \mathrm{d}_q^p(X, X_1) \mathrel{:\!-} \mathrm{d}_r^p(X, X_1), c \ ; \\ \mathrm{d}_r^p(a, X_1) \mathrel{:\!-} c \ ; \ \mathrm{d}_r^p(b, X_1) \mathrel{:\!-} c \ ; \ c \mathrel{:\!-} \text{not } i \ ; \ i \mathrel{:\!-} \text{not } c \end{array} \right\}$$

$AS(\Psi_S^{tr})$ consists of $\{i\}$ plus 16 answer sets, corresponding to all combinations of the 4 answer sets in $AS(\Psi)$.

Now we are able to generalize the encodings for brave, cautious, and definite consequences. These definitions are direct extensions of Definitions 3, 4, and 5, the differences are only due to the non-ground annotations. In particular, the diagonalization atoms $a^a$ should now be written as $\mathrm{d}_p^p(X_1, \ldots, X_{\alpha(p)}, X_1, \ldots, X_{\alpha(p)})$ which represent the set of ground instances of $p(X_1, \ldots, X_{\alpha(p)})$, each annotated by itself. So, a weak constraint $:\sim \ \mathrm{d}_p^p(X_1, \ldots, X_{\alpha(p)}, X_1, \ldots, X_{\alpha(p)})$ gives rise to $\{:\sim \ \mathrm{d}_p^p(c_1, \ldots, c_{\alpha(p)}, c_1, \ldots, c_{\alpha(p)}) \mid c_1, \ldots, c_{\alpha(p)} \in U\}$ where $U$ is the Herbrand base of the program in question, that is one weak constraint for each ground instance annotated by itself.

**Definition 8.** *Given a strong program $P$ and a set $S$ of predicate symbols, let*

$$\begin{aligned} P_S^{bc} &= P_S^{tr} \cup \{:\sim \text{not } \Delta_q \mid q \in S\} \cup \{:\sim i\} \\ P_S^{cc} &= P_S^{tr} \cup \{:\sim \Delta_q \ ; \ \Delta_q \mathrel{:\!-} i \mid q \in S\} \cup \{:\sim i\} \\ P_S^{dc} &= P_S^{tr} \cup \{:\sim \Delta_q \ ; \ I_q \mathrel{:\!-} i \ ; \ :\sim I_q \mid q \in S\} \cup \{:\sim i\} \end{aligned}$$

*where $\Delta_q = \mathrm{d}_q^q(X_1, \ldots, X_{\alpha(q)}, X_1, \ldots, X_{\alpha(q)})$ and $I_q = i_q(X_1, \ldots, X_{\alpha(q)})$.*

**Proposition 5.** *Given a strong program $P$ and a set $S$ of predicates, for an arbitrary $A \in AS(P_S^{bc})$, (resp., $A \in AS(P_S^{cc})$, $A \in AS(P_S^{dc})$), the set $\{p(c_1, \ldots, c_{\alpha(p)}) \mid \mathrm{d}_p^p(c_1, \ldots, c_{\alpha(p)}, c_1, \ldots, c_{\alpha(p)}) \in A\}$ is the set of brave (resp., cautious, definite) consequences of $P$ with a predicate in $S$.*

*Example 7.* Consider again $\Psi$ and $S = \{p\}$ from Example 6. We obtain

$$\Psi_S^{bc} = \Psi_S^{tr} \cup \{:\sim \text{not } \mathrm{d}_p^p(X_1, X_1) \ ; \ :\sim i\}$$

and we can check that $AS(\Psi_S^{bc})$ consists of the sets

$$\begin{aligned}
&R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_q^p(b,a)\},\\
&R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_p^p(a,b), \mathrm{d}_q^p(b,a)\},\\
&R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_p^p(b,a)\},\\
&R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_p^p(b,a), \mathrm{d}_p^p(b,a)\};
\end{aligned}$$

where $R = \{\mathrm{d}_r^p(a,a), \mathrm{d}_r^p(a,b), \mathrm{d}_r^p(b,a), \mathrm{d}_r^p(b,b)\}$. For each $A$ of these answer sets we obtain $\{p(t) \mid \mathrm{d}_p^p(t,t) \in A\} = \{p(a), p(b)\}$ which corresponds exactly to the brave consequences of $\Psi$ with a predicate of $S = \{p\}$.

For cautious consequences, we have

$$\Psi_S^{cc} = \Psi_S^{tr} \cup \{:\sim \mathrm{d}_p^p(X_1,X_1) \ ; \ \mathrm{d}_p^p(X_1,X_1) \text{ :- } i \ ; \ :\sim i\}$$

and we can check that $AS(\Psi_S^{cc})$ consists of the sets

$$\begin{aligned}
&R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_q^p(b,a)\},\\
&R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_p^p(a,b), \mathrm{d}_q^p(b,a)\},\\
&R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_p^p(b,a)\},\\
&R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_p^p(b,a), \mathrm{d}_p^p(b,a)\};
\end{aligned}$$

where $R = \{\mathrm{d}_r^p(a,a), \mathrm{d}_r^p(a,b), \mathrm{d}_r^p(b,a), \mathrm{d}_r^p(b,b)\}$, as above. For each $A$ of these answer sets we obtain $\{p(t) \mid \mathrm{d}_p^p(t,t) \in A\} = \emptyset$ and indeed there are no cautious consequences of $\Psi$ with a predicate of $S = \{p\}$.

Finally, for definite consequences,

$$\Psi_S^{dc} = \Psi_S^{tr} \cup \{:\sim \mathrm{d}_p^p(X_1,X_1) \ ; \ i_p(X_1) \text{ :- } i \ ; \ :\sim i_p(X_1) \ ; \ :\sim i\}.$$

It is easy to see that $AS(\Psi_S^{dc}) = AS(\Psi_S^{cc})$ and so $\{p(t) \mid \mathrm{d}_p^p(t,t) \in A\} = \emptyset$ for each answer set $A$ of $\Psi_S^{dc}$, and indeed there is also no definite consequence of $\Psi$ with a predicate of $S = \{p\}$.

These definitions exploit the fact that the semantics of non-ground programs is defined via their grounding with respect to their Herbrand Universe. So the fresh variables introduced in the manifold will give rise to one copy of a rule for each ground atom.

In practice, ASP systems usually require rules to be safe, that is, that each variable occurs (also) in the positive body. The manifold for a set of predicates may therefore contain unsafe rules (because of the fresh variables). But this can be repaired by adding a *domain atom* $dom_q(X_1,\ldots,X_m)$ to a rule which is to be annotated with $q$. This predicate can in turn be defined by a rule $dom_q(X_1,\ldots,X_m) \text{ :- } u(X_1),\ldots,u(X_m)$ where $u$ is defined using $\{u(c) \mid c \in U_P\}$. One can also provide smarter definitions for $dom_q$ by using a relaxation of the definition for $q$.

We also observe that ground atoms that are contained in all answer sets of a program need not be annotated in the manifold. Note that these are essentially the cautious consequences of a program and therefore determining all of those automatically before rewriting does not make sense. But for some atoms this property can be determined by a simple analysis of the structure of the program. For instance, facts will be in all answer sets. In the sequel we will not annotate extensional atoms (those defined only by facts) in order to obtain more concise programs. One could also go further and omit the annotation of atoms which are defined using non-disjunctive stratified programs.

As an example, we present an ASP encoding for boolean satisfiability and then create its manifold program for resolving the following problem: Given a propositional formula in CNF $\varphi$, compute all atoms which are true in all models of $\varphi$. We provide a fixed program which takes a representation of $\varphi$ as facts as input. To apply our method we first require a program whose answer sets are in a one-to-one correspondence to the models of $\varphi$. To start with, we fix the representation of CNFs. Let $\varphi$ (over atoms $A$) be of the form $\bigwedge_{i=1}^{n} c_i$. Then, $D_\varphi = \{\mathrm{at}(a) \mid a \in A\} \cup \{\mathrm{cl}(i) \mid 1 \leq i \leq n\} \cup \{\mathrm{pos}(a,i) \mid$ atom $a$ occurs positively in $c_i\} \cup \{\mathrm{neg}(a,i) \mid$ atom $a$ occurs negatively in $c_i\}$. We construct program SAT as the set of the following rules.

$$\begin{array}{rlrl}
t(X) & \text{:- not } f(X), \mathrm{at}(X); & f(X) & \text{:- not } t(X), \mathrm{at}(X); \\
ok(C) & \text{:- } t(X), \mathrm{pos}(C,X); & ok(C) & \text{:- } f(X), \mathrm{neg}(C,X); \\
 & \text{:- not } ok(C), \mathrm{cl}(C). & &
\end{array}$$

It can be checked that the answer sets of $\mathrm{SAT} \cup D_\varphi$ are in a one-to-one correspondence to the models (over $A$) of $\varphi$. In particular, for any model $I \subseteq A$ of $\varphi$ there exists an answer set $M$ of $\mathrm{SAT} \cup D_\varphi$ such that $I = \{a \mid t(a) \in M\}$. We now consider $\mathrm{SAT}^{cc}_{\{t\}}$ which consists of the following rules.

$$\begin{array}{rlrl}
\mathrm{d}^t_t(X,Y) & \text{:- } c, \text{not } \mathrm{d}^t_f(X,Y), \mathrm{at}(X); & \mathrm{d}^t_f(X,Y) & \text{:- } c, \text{not } \mathrm{d}^t_t(X,Y), \mathrm{at}(X); \\
\mathrm{d}^t_{ok}(C,Y) & \text{:- } c, \mathrm{d}^t_t(X,Y), \mathrm{pos}(C,X); & \mathrm{d}^t_{ok}(C,Y) & \text{:- } c, \mathrm{d}^t_f(X,Y), \mathrm{neg}(C,X); \\
 & \text{:- } c, \text{not } \mathrm{d}^t_{ok}(C,Y), \mathrm{cl}(C); & \mathrm{d}^t_t(X,X) & \text{:- } i\ ; \\
c & \text{:- not } i; & i & \text{:- not } c; \\
 & :\sim \mathrm{d}^t_t(X,X); & & :\sim i.
\end{array}$$

Given Proposition 5, it is easy to see that, given some answer set $A$ of $\mathrm{SAT}^{cc}_{\{t\}} \cup D_\varphi$, $\{a \mid \mathrm{d}^t_t(a,a) \in A\}$ is precisely the set of atoms which are true in all models of $\varphi$.

## 5 Applications

In this section, we put our technique to work and show how to use meta-reasoning over answer sets for three application scenarios. The first one is a well-known problem from propositional logic, and we will reuse the example from above. The second example takes a bit more background, but presents a novel method to compute ideal extensions for argumentation frameworks which was also implemented in the logic-programming based argumentation system ASPARTIX [23].[3] Finally, we address Michael Gelfond's epistemic specification, a powerful extension of standard ASP with modal atoms which allow for meta-reasoning over answer sets. In particular, we will consider a certain subclass which is directly amenable to manifolds.

[3] For a web frontend, see `http://rull.dbai.tuwien.ac.at:8080/ASPARTIX`

## 5.1 The Unique Minimal Model Problem

As a first example, we show how to encode the problem of deciding whether a given propositional formula $\varphi$ has a unique minimal model. This problem is known to be in $\Theta_2^P$ and to be co-NP-hard (the exact complexity is an open problem). Let $I$ be the intersection of all models of $\varphi$. Then $\varphi$ has a unique minimal model iff $I$ is also a model of $\varphi$. We thus use our example from the previous section, and define the program $\mathrm{UNIQUE}$ as $\mathrm{SAT}^{cc}_{\{t\}}$ augmented by rules

$$\begin{aligned} \mathit{ok}(C) &\text{ :- } \mathrm{d}^t_t(X,X), \mathrm{pos}(C,X);\\ \mathit{ok}(C) &\text{ :- } \mathrm{not}\ \mathrm{d}^t_t(X,X), \mathrm{neg}(C,X);\\ &\text{ :- } \mathrm{not}\ \mathit{ok}(C), \mathrm{cl}(C). \end{aligned}$$

We immediately obtain the following result.

**Theorem 1.** *For any CNF formula $\varphi$, it holds that $\varphi$ has a unique minimal model, if and only if program* $\mathrm{UNIQUE} \cup D_\varphi$ *has at least one answer set.*

A slight adaption of this encoding allows us to formalize CWA-reasoning [13] over a propositional knowledge base $\varphi$, since the atoms $a$ in $\varphi$, for which the corresponding atoms $\mathrm{d}^t_t(a,a)$ are not contained in an answer set of $\mathrm{SAT}^{cc}_{\{t\}} \cup D_\varphi$, are exactly those which are added negated to $\varphi$ for CWA-reasoning.

## 5.2 Computing the Ideal Extension

Our second example is from the area of argumentation, where the problem of computing the ideal extension [15] of an abstract argumentation framework was recently shown to be complete for $\mathrm{FP}^{\mathrm{NP}}_{||}$ in [24]. Thus, this task cannot be compactly encoded via normal programs (under usual complexity theoretic assumptions). On the other hand, the complexity shows that employing disjunction is not necessary, if one instead uses weak constraints. We first give the basic definitions following [25].

**Definition 9.** *An* argumentation framework (AF) *is a pair $F = (A, R)$ where $A \subseteq \mathcal{U}$ is a set of arguments and $R \subseteq A \times A$. $(a, b) \in R$ means that $a$ attacks $b$. An argument $a \in A$ is* defended *by $S \subseteq A$ (in $F$) if, for each $b \in A$ such that $(b, a) \in R$, there exists a $c \in S$, such that $(c, b) \in R$. An argument $a$ is admissible (in $F$) w.r.t. a set $S \subseteq A$ if each $b \in A$ which attacks $a$ is defended by $S$.*

Semantics for argumentation frameworks are given in terms of so-called extensions. The next definitions introduce two such notions which also underlie the concept of an ideal extension.

**Definition 10.** *Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is said to be* conflict-free (in *$F$), if there are no $a, b \in S$, such that $(a, b) \in R$. A set $S$ is an* admissible extension *of $F$, if $S$ is conflict-free in $F$ and each $a \in S$ is admissible in $F$ w.r.t. $S$. The collection of admissible extensions is denoted by $\mathit{adm}(F)$. An admissible extension $S$ of $F$ is a* preferred extension *of $F$, if for each $T \in \mathit{adm}(F)$, $S \not\subset T$. The collection of preferred extensions of $F$ is denoted by $\mathit{pref}(F)$.*

The original definition of ideal extensions is as follows [15].

**Definition 11.** *Let $F$ be an AF. A set $S$ is called* ideal *for $F$, if $S \in adm(F)$ and $S \subseteq \bigcap_{T \in pref(F)} T$. A maximal (w.r.t. set-inclusion) ideal set of $F$ is called an* ideal extension *of $F$.*

It is known that each AF possesses a unique ideal extension. In [24], the following algorithm to compute the ideal extension of an AF $F = (A, R)$ is proposed. Let

$$X_F^- = A \setminus \bigcup_{S \in adm(F)} S \text{ and}$$
$$X_F^+ = \{a \in A \mid \forall b, c : (b, a), (a, c) \in R \Rightarrow b, c \in X_F^-\} \setminus X_F^-,$$

and define the AF $F^* = (X_F^+ \cup X_F^-, R^*)$ where the attack relation $R^*$ is given as $R \cap \{(a, b), (b, a) \mid a \in X_F^+, b \in X_F^-\}$. $F^*$ is a bipartite AF in the sense that $R^*$ is a bipartite graph.

**Proposition 6 ([24]).** *The ideal extension of AF $F$ is given by $\bigcup_{S \in adm(F^*)} (S \cap X_F^+)$.*

The set of all admissible atoms for a bipartite AF $F$ can be computed in polynomial time using Algorithm 1 of [26]. This is basically a fixpoint iteration identifying arguments in $X_F^+$ that cannot be in an admissible extension: First, arguments in $X_0 = X_F^+$ are excluded, which are attacked by unattacked arguments (which are necessarily in $X_F^-$), yielding $X_1$. Now, arguments in $X_F^-$ may be unattacked by $X_1$, and all arguments in $X_1$ attacked by such newly unattacked arguments should be excluded. This process is iterated until either no arguments are left or no more argument can be excluded. There may be at most $|X_F^+|$ iterations in this process.

We exploit this technique to formulate an ASP-encoding IDEAL. We first describe a program the answer sets of which characterize admissible extensions. Then, we use the brave manifold of this program in order to determine all arguments contained in some admissible extension. Finally, we extend this manifold program in order to identify $F^*$ and to simulate Algorithm 1 of [26].

The argumentation frameworks will be given to IDEAL as sets of input facts. Given an AF $F = (A, R)$, let $D_F = \{a(x) \mid x \in A\} \cup \{\mathrm{r}(x, y) \mid (x, y) \in R\}$. The program ADM, given by the rules below, computes admissible extensions (cf. [27,23]):

$$\begin{aligned}
\mathrm{in}(X) &\text{ :- } \mathrm{not\ out}(X), a(X);\\
\mathrm{out}(X) &\text{ :- } \mathrm{not\ in}(X), a(X);\\
&\text{ :- } \mathrm{in}(X), \mathrm{in}(Y), \mathrm{r}(X, Y);\\
\mathrm{def}(X) &\text{ :- } \mathrm{in}(Y), \mathrm{r}(Y, X);\\
&\text{ :- } \mathrm{in}(X), \mathrm{r}(Y, X), \mathrm{not\ def}(Y).
\end{aligned}$$

Indeed one can show that, given an AF $F$, the answer sets of $\mathrm{ADM} \cup D_F$ are in a one-to-one correspondence to the admissible extensions of $F$ via the $\mathrm{in}(\cdot)$ predicate. In order to determine the brave consequences of ADM for predicate in, we form $\mathrm{ADM}^{bc}_{\{\mathrm{in}\}}$, and extend it by collecting all brave consequences of $\mathrm{ADM} \cup D_F$ in predicate $\mathrm{in}(\cdot)$, from which we can determine $X_F^-$ (represented by $\mathrm{in}^-(\cdot)$), $X_F^+$ (represented by $\mathrm{in}^+(\cdot)$, using auxiliary predicate $\mathrm{not\_in}^+(\cdot)$), and $R^*$ (represented by $\mathrm{q}(\cdot,\cdot)$).

$$\begin{aligned}
\mathrm{in}(X) &\text{ :- } \mathrm{d}_{\mathrm{in}}^{\mathrm{in}}(X,X);\\
\mathrm{in}^-(X) &\text{ :- } a(X), \mathrm{not\ in}(X);\\
\mathrm{in}^+(X) &\text{ :- } \mathrm{in}(X), \mathrm{not\ not\_in}^+(X);\\
\mathrm{not\_in}^+(X) &\text{ :- } \mathrm{in}(Y), r(X,Y);\\
\mathrm{not\_in}^+(X) &\text{ :- } \mathrm{in}(Y), r(Y,X);\\
\mathrm{q}(X,Y) &\text{ :- } \mathrm{r}(X,Y), \mathrm{in}^+(X), \mathrm{in}^-(Y);\\
\mathrm{q}(X,Y) &\text{ :- } \mathrm{r}(X,Y), \mathrm{in}^-(X), \mathrm{in}^+(Y).
\end{aligned}$$

In order to simulate Algorithm 1 of [26], we use the elements in $X_F^+$ for marking the iteration steps. To this end, we use an arbitrary order $<$ on ASP constants (all ASP systems provide such a predefined order) and define successor, infimum and supremum among the constants representing $X_F^+$ w.r.t. the order $<$.

$$\begin{aligned}
\mathrm{nsucc}(X,Z) &\text{ :- } \mathrm{in}^+(X), \mathrm{in}^+(Y), \mathrm{in}^+(Z), X < Y, Y < Z;\\
\mathrm{succ}(X,Y) &\text{ :- } \mathrm{in}^+(X), \mathrm{in}^+(Y), X < Y, \mathrm{not\ nsucc}(X,Y);\\
\mathrm{ninf}(Y) &\text{ :- } \mathrm{in}^+(X), \mathrm{in}^+(Y), X < Y;\\
\mathrm{nsup}(X) &\text{ :- } \mathrm{in}^+(X), \mathrm{in}^+(Y), X < Y;\\
\mathrm{inf}(X) &\text{ :- } \mathrm{in}^+(X), \mathrm{not\ ninf}(X);\\
\mathrm{sup}(X) &\text{ :- } \mathrm{in}^+(X), \mathrm{not\ nsup}(X).
\end{aligned}$$

We now use this to iteratively determine arguments that are not in the ideal extension, using $\mathrm{nid}(\cdot,\cdot)$, where the first argument is the iteration step. In the first iteration (identified by the infimum) all arguments in $X_F^+$ which are attacked by an unattacked argument are collected. In subsequent iterations, all arguments from the previous steps are included and augmented by arguments that are attacked by an argument not attacked by arguments in $X_F^+$ that were not yet excluded in the previous iteration. Finally, arguments in the ideal extension are those that are not excluded from $X_F^+$ in the final iteration (identified by the supremum).

$$\begin{aligned}
\mathrm{att}_0(X) &\text{ :- } \mathrm{q}(Y,X);\\
\mathrm{att}_i(J,Z) &\text{ :- } q(Y,Z), \mathrm{in}^+(Y), \mathrm{not\ nid}(J,Y), \mathrm{in}^+(J);\\
\mathrm{ideal}(X) &\text{ :- } \mathrm{in}^+(X), \mathrm{sup}(I), \mathrm{not\ nid}(I,X);\\
\mathrm{nid}(I,Y) &\text{ :- } \mathrm{succ}(J,I), \mathrm{nid}(J,Y);\\
\mathrm{nid}(I,Y) &\text{ :- } \mathrm{inf}(I), \mathrm{q}(Z,Y), \mathrm{in}^+(Y), \mathrm{not\ att}_0(Z);\\
\mathrm{nid}(I,Y) &\text{ :- } \mathrm{succ}(J,I), q(Z,Y), \mathrm{in}^+(Y), \mathrm{not\ att}_i(J,Z).
\end{aligned}$$

If we put $\mathrm{ADM}^{bc}_{\{\mathrm{in}\}}$ and all of these additional rules together to form the program IDEAL, we obtain the following result:

**Theorem 2.** *Let $F$ be an AF and $A \in AS(\mathrm{IDEAL} \cup D_F)$. Then, the ideal extension of $F$ is given by* $\{a \mid \mathrm{ideal}(a) \in A\}$.

## 5.3 Epistemic Specifications

*Epistemic Specifications* have been defined in [16], and are an extension of programs as defined in Section 2 by the possible occurrence of epistemic operators $\mathrm{K}$ and $\mathrm{M}$. In this paper, we will consider a simple class of epistemic specifications, which includes the main motivating example of [16][4].

A *simple epistemic literal* is one of $\mathrm{K}a$, $\neg\mathrm{K}a$, $\mathrm{M}a$ or $\neg\mathrm{M}a$, where $a$ is an atom as in Section 2. A *simple epistemic specification* is a set of epistemic rules

$$a_1 \lor \cdots \lor a_n \text{ :- } B_1, \ldots, B_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m \qquad (1)$$

where $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$, $B_1, \ldots, B_k$ are atoms or simple epistemic literals and $a_1, \ldots, a_n, b_{k+1}, \ldots, b_m$ are atoms. We say that an atom $a$ directly modally depends on an atom $b$ if $a$ is one of $a_1, \ldots, a_n$ and $b$ occurs in a simple epistemic literal of $B_1, \ldots, B_k$ in a rule of the form (1). A simple epistemic specification is *modally acyclic* if no atom depends modally on itself in the transitive closure of the direct modal dependency relation. A specification is *one-step modal* if each atom $a$ directly modally depends only on atoms which do not depend modally on other atoms.

Herbrand Universe and Base are defined as for standard logic programs, considering also atoms in simple epistemic literals (but no modal operators). In the context of epistemic specifications, collections of interpretations are called *world views*. Satisfaction of standard atoms by interpretations is defined as usual. Satisfaction of simple epistemic literals is defined with respect to world views: A world view $W$ satisfies $\mathrm{K}a$, written $W \models \mathrm{K}a$, iff $\forall B \in W : a \in B$. $W$ satisfies $\mathrm{M}a$, written $W \models \mathrm{M}a$, iff $\exists B \in W : a \in B$. Moreover, $W \models \neg\mathrm{K}a$ iff $W \not\models \mathrm{K}a$ and $W \models \neg\mathrm{M}a$ iff $W \not\models \mathrm{M}a$.

The *modal reduct* of a simple epistemic specification $\Pi$ with respect to a world view $W$, denoted $\Pi^W$, is obtained by deleting all epistemic rules of $\Pi$ of the form (1) where $W \not\models B_i$ for some simple epistemic literal $B_i$, and by deleting all simple epistemic literals of the remaining rules. Note that $\Pi^W$ is a standard program without epistemic literals. $W$ is a world view of $\Pi$ iff $W = AS(\Pi^W)$.

Observe that standard programs without weak constraints are epistemic specifications, and their modal reduct is equal to the original program. These programs therefore have a single world view, the collection of the answer sets of the program.

A one-step modal epistemic specification $\Pi$ can be split into two specifications $\Pi_1$ (the lower part) and $\Pi_2$ (the upper part), where $\Pi_1 \cap \Pi_2 = \emptyset$ and $\Pi_1 \cup \Pi_2 = \Pi$, such that $\Pi_1$ does not contain $\mathrm{K}$ or $\mathrm{M}$, and no head atom of $\Pi_2$ occurs in $\Pi_1$. This is similar to, and in fact a special case of, splitting epistemic specifications as defined in [28].

In order to be able to compute world views of one-step modal epistemic specifications by means of manifold programs, we would like them to have a single world view. The reason is that it is not clear how to differentiate between a specification having multiple world views and a specification having a single world view that contains all sets of the multiple world views of the first specification. The issues are best explained by an example.

[4] Here we do not consider strong negation (except for negating epistemic operators) in order to keep the framework simple. It can be extended without major efforts to incorporate also strong negation.

*Example 8.* Consider the following one-step modal epistemic specification

$$\begin{aligned} &:\!-\ a, \mathrm{K}b;\\ &:\!-\ b, \mathrm{K}a;\\ &:\!-\ \mathrm{M}a, \mathrm{M}b;\\ a \vee b\ &:\!-\ . \end{aligned}$$

It has two world views, $\{\{a\}\}$ and $\{\{b\}\}$. It is not clear how to find a manifold encoding for this specification which lets one differentiate its output from a manifold encoding of a specification having one world view $\{\{a\}, \{b\}\}$ (for example the specification consisting only of $a \vee b$). The difficulty is that one would have to encode also an indicator in which world view(s) an interpretation occurs, which appears to be a hard, if not impossible, task.

The important observation in the example is that the upper part of the specification can trigger incoherences (in this case because of constraint violations), and for this reason not all answer sets of the lower part necessarily have a corresponding answer set in a world view of the complete specification. A similar issue has been described in [28], where specifications are called *safe* if (for the special case of one-step modal epistemic specifications) the modal reduct of the upper part with respect to the collection of answer sets of the lower part has answer sets when any answer set of the lower part is added as a set of facts, that is if

$$\forall A \in AS(\Pi_1) : AS(\Pi_2^{AS(\Pi_1)} \cup A) \neq \emptyset.$$

For any safe one-step modal epistemic specification $\Pi$ and one of its world views $W$, any $A \in W$ extends an $A' \in AS(\Pi_1)$ and, vice versa, each $A' \in AS(\Pi_1)$ is contained in some $A \in W$. Therefore, for any epistemic literal $\ell$ in $\Pi$ and any world view $W$ of $\Pi$, we have that $W \models \ell$ if and only if $AS(\Pi_1) \models \ell$, and as a consequence $\Pi^W = \Pi^{AS(\Pi_1)}$ and so $W = AS(\Pi^{AS(\Pi_1)})$ is unique.

*Example 9.* Consider the following variant $\Pi^g$ of the main motivating example of [16].

$$\begin{aligned} \mathit{eligible}(X)\ &:\!-\ \mathit{highGPA}(X);\\ \mathit{eligible}(X)\ &:\!-\ \mathit{minority}(X), \mathit{fairGPA}(X);\\ \mathit{notEligible}(X)\ &:\!-\ \mathit{notFairGPA}(X), \mathit{notHighGPA}(X);\\ \mathit{interview}(X)\ &:\!-\ \neg\mathrm{K}\mathit{eligible}(X), \neg\mathrm{K}\mathit{notEligible}(X). \end{aligned}$$

This (and any extensions by facts) is a safe one-step modal epistemic specification: The first three rules form the lower part $\Pi_1$ and the last rule forms the upper part $\Pi_2$.

Moreover, observe that due to the considerations above, for the lower part $\Pi_1$ of a one-step modal epistemic specification $\Pi$, $AS(\Pi_1) \models \mathrm{M}a$ iff $\Pi_1 \models_b a$ ($AS(\Pi_1) \models \neg\mathrm{M}a$ iff $\Pi_1 \not\models_b a$) and $AS(\Pi_1) \models \mathrm{K}a$ iff $\Pi_1 \models_c a$ ($AS(\Pi_1) \models \neg\mathrm{K}a$ iff $\Pi_1 \not\models_c a$) for epistemic literals $\mathrm{M}a, \neg\mathrm{M}a, \mathrm{K}a, \neg\mathrm{K}a$ in $\Pi$.

We can then use Proposition 5 in order to simulate the modal reduct $\Pi^W$ of the unique world view $W$ of $\Pi$. In particular,

$$W \models \mathrm{M}p(t_1, \ldots, t_n) \quad \text{iff} \quad \mathrm{d}_p^p(t_1, \ldots, t_n, \boldsymbol{X}) \in AS({\Pi_1}_p^{bc}),$$

(with $\boldsymbol{X}$ being a sequence of suitably chosen variables, cf. Section 4) and

$$W \models \mathrm{K}p(t_1,\dots,t_n) \quad \text{iff} \quad \mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X}) \in AS({\Pi_1}^{cc}_p).$$

Moreover, we have $W \models \neg\mathrm{M}p(t_1,\dots,t_n)$ iff $\mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X}) \notin AS({\Pi_1}^{bc}_p)$, and $W \models \neg\mathrm{K}p(t_1,\dots,t_n)$ iff $\mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X}) \notin AS({\Pi_1}^{cc}_p)$. Making sure that all ${\Pi_1}^{cc}_p$ and ${\Pi_1}^{bc}_p$ use distinct symbols, different also from those in $\Pi$, we can form the union of all of these programs.

That means that we can replace each occurrence of $\mathrm{K}p(t_1,\dots,t_n)$ in $\Pi$ by the manifold atom $\mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X})$ (and $\neg\mathrm{K}p(t_1,\dots,t_n)$ by the corresponding default negated atom, i.e. not $\mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X})$) and add ${\Pi_1}^{cc}_{\{p\}}$; symmetrically, we can replace each occurrence of $\mathrm{M}p(t_1,\dots,t_n)$ by $\mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X})$ (and $\neg\mathrm{M}p(t_1,\dots,t_n)$ by not $\mathrm{d}^p_p(t_1,\dots,t_n,\boldsymbol{X})$) and add ${\Pi_1}^{bc}_{\{p\}}$. Let us call the program obtained in this way $\overline{\Pi}$. $\overline{\Pi}$ can be split such that ${\Pi_1}^{bc}_{\{p\}}$ and ${\Pi_1}^{cc}_{\{p\}}$ form the bottom program $\overline{\Pi}_1$, and the partial evaluation $\overline{\Pi}'$ of $\overline{\Pi}$ with respect to $AS(\overline{\Pi}_1)$ coincides with $\Pi^W$ for the unique world view $W$ of $\Pi$. It follows that the restriction of each $A \in AS(\overline{\Pi})$ to the symbols of $\Pi$ is in $W$, and for each $A \in W$, an $A' \in AS(\overline{\Pi})$ exists, such that the restriction of $A'$ to symbols in $\Pi$ is $A$.

*Example 10.* Reconsider the main motivating example $\Pi^g$ of [16] as reported in Example 9. $\overline{\Pi^g}$ is:

$$\begin{aligned}
\mathit{eligible}(X) &\coloneq \mathit{highGPA}(X);\\
\mathit{eligible}(X) &\coloneq \mathit{minority}(X), \mathit{fairGPA}(X);\\
\mathit{notEligible}(X) &\coloneq \mathit{notFairGPA}(X), \mathit{notHighGPA}(X);\\
\mathit{interview}(X) &\coloneq \text{not } \mathrm{d}^{\mathit{eligible}}_{\mathit{eligible}}(X,X), \text{not } \mathrm{d}^{\mathit{notEligible}}_{\mathit{notEligible}}(X,X);\\
\mathrm{d}^{\mathit{eligible}}_{\mathit{eligible}}(X,X_1) &\coloneq c_1, \mathrm{d}^{\mathit{eligible}}_{\mathit{highGPA}}(X,X_1);\\
\mathrm{d}^{\mathit{eligible}}_{\mathit{eligible}}(X,X_1) &\coloneq c_1, \mathrm{d}^{\mathit{eligible}}_{\mathit{minority}}(X,X_1), \mathrm{d}^{\mathit{eligible}}_{\mathit{fairGPA}}(X,X_1);\\
\mathrm{d}^{\mathit{eligible}}_{\mathit{notEligible}}(X,X_1) &\coloneq c_1, \mathrm{d}^{\mathit{eligible}}_{\mathit{notFairGPA}}(X,X_1), \mathrm{d}^{\mathit{eligible}}_{\mathit{notHighGPA}}(X,X_1);\\
\mathrm{d}^{\mathit{eligible}}_{\mathit{eligible}}(X_1,X_1) &\coloneq i_1;\\
c_1 &\coloneq \text{not } i_1;\\
i_1 &\coloneq \text{not } c_1;\\
&:\sim \mathrm{d}^{\mathit{eligible}}_{\mathit{eligible}}(X_1,X_1);\\
&:\sim i_1;\\
\mathrm{d}^{\mathit{notEligible}}_{\mathit{eligible}}(X,X_1) &\coloneq c_2, \mathrm{d}^{\mathit{notEligible}}_{\mathit{highGPA}}(X,X_1);\\
\mathrm{d}^{\mathit{notEligible}}_{\mathit{eligible}}(X,X_1) &\coloneq c_2, \mathrm{d}^{\mathit{notEligible}}_{\mathit{minority}}(X,X_1), \mathrm{d}^{\mathit{notEligible}}_{\mathit{fairGPA}}(X,X_1);\\
\mathrm{d}^{\mathit{notEligible}}_{\mathit{notEligible}}(X,X_1) &\coloneq c_2, \mathrm{d}^{\mathit{notEligible}}_{\mathit{notFairGPA}}(X,X_1), \mathrm{d}^{\mathit{notEligible}}_{\mathit{notHighGPA}}(X,X_1);\\
\mathrm{d}^{\mathit{notEligible}}_{\mathit{notEligible}}(X_1,X_1) &\coloneq i_2;\\
c_2 &\coloneq \text{not } i_2;\\
i_2 &\coloneq \text{not } c_2;\\
&:\sim \mathrm{d}^{\mathit{notEligible}}_{\mathit{notEligible}}(X_1,X_1);\\
&:\sim i_2.
\end{aligned}$$

This rewriting can be extended to safe modally acyclic epistemic specifications essentially by a repeated application, but special care must be taken of the involved weak constraints.

## 6 Conclusion

In this paper, we provided a novel method to rewrite ASP programs in such a way that reasoning over all answer sets of the original program can be formulated within the same program. Our method exploits the well-known concept of weak constraints. We illustrated the impact of our method by encoding the problems of (i) deciding whether a propositional formula in CNF has a unique minimal model, (ii) computing the ideal extension of an argumentation framework. For (i) and (ii), known complexity results witness that our encodings are adequate in the sense that efficient ASP encodings without weak constraints or similar constructs are assumed to be infeasible. As a final application we considered (iii) epistemic specifications, where we used our concepts to simulate the semantics of epistemic literals within a single world view (thus we had to restrict ourselves to a particular subclass of epistemic specifications). Our encodings provide evidence that the class of disjunctive (non-disjunctive) safe one-step modal epistemic specifications is easier to evaluate (in $\Theta_3^P$ resp. $\Theta_2^P$) as the respective general class of disjunctive (non-disjunctive) epistemic specifications (which have been shown to be hard for $\Sigma_3^P$ resp. $\Sigma_2^P$ in [21]).

Concerning related work, we remark that the manifold program for cautious consequences is closely related to the concept of data disjunctions [29] (this paper also contains a detailed discussion about the complexity class $\Theta_2^P$ and related classes for functional problems). Concepts similar to manifold programs have also been studied in the area of default logic, where a method for reasoning within a single extension has been proposed [30]. That method uses set-variables which characterize the set of generating defaults of the original extensions. However, such an approach differs considerably from ours as it encodes certain aspects of the semantics (which ours does not), which puts it closer to meta-programming (cf. [31]).

As future work, we intend studying the use of alternative preferential constructs in place of weak constraints. Moreover, we are currently developing a suitable language for expressing reasoning with brave, cautious and definite consequences, allowing also for mixing different reasoning modes. This language should serve as a platform for natural encodings of problems in complexity classes $\Theta_2^P$, $\Theta_3^P$, $\mathrm{FP}_{||}^{\mathrm{NP}}$, and $\mathrm{FP}_{||}^{\Sigma_2^P}$. A first step towards this direction has already been undertaken in [32]; such extensions should also pave the way to simulate a broader class of epistemic specifications.

## References

1. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K., Marek, V.W., Truszczyński, M., Warren, D.S. (eds.) The Logic Programming Paradigm – A 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)
2. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. 25(3-4), 241–273 (1999)

3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2002)
4. Gelfond, M.: Representing knowledge in A-prolog. In: Kakas, A., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS (LNAI), vol. 2408, pp. 413–451. Springer, Heidelberg (2002)
5. 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)
6. 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: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), pp. 915–917. ACM Press, New York (2005)
7. Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R.: Representing configuration knowledge with weight constraint rules. In: Provetti, A., Son, T.C. (eds.) Proceedings of the 1st International Workshop on Answer Set Programming (2001)
8. 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)
9. Bravo, L., Bertossi, L.E.: Logic programs for consistently querying data integration systems. In: Gottlob, G., Walsh, T. (eds.) Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 10–15. Morgan Kaufmann, San Francisco (2003)
10. Saccà, D.: Multiple total stable models are definitely needed to solve unique solution problems. Inf. Process. Lett. 58(5), 249–254 (1996)
11. Buccafurri, F., Leone, N., Rullo, P.: Enhancing disjunctive datalog by constraints. IEEE Trans. Knowl. Data Eng. 12(5), 845–860 (2000)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)
13. Reiter, R.: On closed world data bases. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 55–76. Plenum Press, New York (1978)
14. Bench-Capon, T.J.M., Dunne, P.E.: Argumentation in artificial intelligence. Artif. Intell. 171(10-15), 619–641 (2007)
15. Dung, P.M., Mancarella, P., Toni, F.: Computing ideal sceptical argumentation. Artif. Intell. 171(10-15), 642–674 (2007)
16. Gelfond, M.: Logic programming and reasoning with incomplete information. Annals of Mathematics and Artificial Intelligence 12(1-2), 89–116 (1994)
17. Gelfond, M.: Strong introspection. In: Proceedings of the 9th National Conference on Artificial Intelligence (AAAI 1991), pp. 386–391. AAAI Press / The MIT Press (1991)
18. Wang, K., Zhang, Y.: Nested epistemic logic programs. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 279–290. Springer, Heidelberg (2005)
19. Zhang, Y.: Computational properties of epistemic logic programs. In: Doherty, P., Mylopoulos, J., Welty, C.A. (eds.) Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), pp. 308–317. AAAI Press, Menlo Park (2006)
20. Zhang, Y.: Updating epistemic logic programs. J. Log. Comput. 19(2), 405–423 (2009)
21. Truszczyński, M.: Revisiting epistemic specifications. In: Balduccini, M., Son, T.C. (eds.) Gelfond Festschrift. LNCS (LNAI), vol. 6565, pp. 315–333. Springer, Heidelberg (2011)
22. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3/4), 365–386 (1991)

23. Egly, U., Gaggl, S.A., Woltran, S.: Answer-set programming encodings for argumentation frameworks. Argument and Computation 1(2), 144–177 (2010)
24. Dunne, P.E.: The computational complexity of ideal semantics. Artif. Intell. 173(18), 1559–1591 (2009)
25. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77(2), 321–358 (1995)
26. Dunne, P.E.: Computational properties of argument systems satisfying graph-theoretic constraints. Artif. Intell. 171(10-15), 701–729 (2007)
27. Osorio, M., Zepeda, C., Nieves, J.C., Cortés, U.: Inferring acceptable arguments with answer set programming. In: Proceedings of the 6th Mexican International Conference on Computer Science (ENC 2005), pp. 198–205. IEEE, Los Alamitos (2005)
28. Watson, R.: A splitting set theorem for epistemic specifications. In: Baral, C., Truszczyński, M. (eds.) Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, NMR 2000 (2000), http://arxiv.org/abs/cs/0003038
29. Eiter, T., Veith, H.: On the complexity of data disjunctions. Theor. Comput. Sci. 288(1), 101–128 (2002)
30. Delgrande, J.P., Schaub, T.: Reasoning credulously and skeptically within a single extension. Journal of Applied Non-Classical Logics 12(2), 259–285 (2002)
31. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Computing preferred answer sets by meta-interpretation in answer set programming. TPLP 3(4-5), 463–498 (2003)
32. Faber, W., Woltran, S.: A framework for programming with module consequences. In: de Vos, M., Schaub, T. (eds.) Proceedings of the LPNMR 2009 Workshop on Software Engineering for Answer Set Programming, SEA 2009, pp. 34–48 (2009), http://sea09.cs.bath.ac.uk/downloads/sea09proceedings.pdf

# On the Minimality of Stable Models

Paolo Ferraris[1] and Vladimir Lifschitz[2]

[1] Google, USA
[2] Department of Computer Science, University of Texas at Austin, USA

**Abstract.** The class of logic programs covered by the original definition of a stable model has the property that all stable models of a program in this class are minimal. In the course of research on answer set programming, the concept of a stable model was extended to several new programming constructs, and for some of these extensions the minimality property does not hold. We are interested in syntactic conditions on a logic program that guarantee the minimality of its stable models. This question is addressed here in the context of the general theory of stable models of first-order sentences.

## 1 Introduction

A Prolog program with negation, viewed as a logical formula, usually has several minimal Herbrand models, and only one of them may reflect the actual behavior of Prolog. For instance, the propositional rule

$$p \leftarrow not\ q,$$

viewed as the formula

$$\neg q \rightarrow p \tag{1}$$

written in logic programming notation, has two minimal models, $\{p\}$ and $\{q\}$; the first of them is the "intended" model. Early research on the semantics of negation as failure [Bidoit and Froidevaux, 1987, Gelfond, 1987, Apt *et al.*, 1988, Van Gelder, 1988, Van Gelder *et al.*, 1988] was motivated by the need to distinguish between the intended model of a logic program and its other minimal models.

The definition of a stable model proposed in [Gelfond and Lifschitz, 1988] had a similar motivation. According to Theorem 1 from that paper, every stable model of a logic program is minimal. The converse, for programs with negation, is usually not true. One corollary to the fact that all stable models are minimal is that the collection of stable models of a program is an antichain: one stable model cannot be a subset of another.

In the course of research on answer set programming, the concept of a stable model was extended to several new programming constructs, and for some of these extensions the antichain property does not hold. Take, for instance, choice rules, which play an important role in the language of LPARSE.[1] The set of stable models of the program consisting of the single choice rule

$$\{p\}$$

is $\{\emptyset, \{p\}\}$. It is not an antichain. If we identify this choice rule with the formula

$$p \vee \neg p, \tag{2}$$

as proposed in [Ferraris, 2005], then we can say that the singleton $\{p\}$ is a stable but nonminimal model of (2).

The situation is similar for some cardinality constraints containing negation as failure. The one-rule LPARSE program

$$p \leftarrow \{\mathit{not}\ p\}\ \mathit{0}$$

has the same stable models as the choice rule above, $\emptyset$ and $\{p\}$. According to [Ferraris, 2005], this program can be identified with the formula

$$\neg\neg p \rightarrow p. \tag{3}$$

The singleton $\{p\}$ is a nonminimal stable model of (3).

Under what syntactic conditions on a logic program can we assert that every stable model of the program is minimal? What is the essential difference, for instance, between formula (1) on the one hand, and formulas (2) and (3) on the other? In this note we address this question in the context of the general theory of stable models proposed in [Ferraris *et al.*, 2010]. The main definition of that paper, reproduced in the next section, describes the "stable model operator" $\mathrm{SM}_{\mathbf{p}}$, where $\mathbf{p}$ is a tuple of predicate constants. This operator turns any first-order sentence $F$ into a conjunction of $F$ with a second-order sentence. The stable models of $F$ relative to the given choice of "intensional" predicates $\mathbf{p}$ are defined in [Ferraris *et al.*, 2010] as models of $\mathrm{SM}_{\mathbf{p}}[F]$ in the sense of classical logic. The definition of $\mathrm{SM}_{\mathbf{p}}[F]$ is very similar to the definition of the parallel circumscription of $\mathbf{p}$ in $F$ [McCarthy, 1986], which we will denote by $\mathrm{CIRC}_{\mathbf{p}}[F]$. The circumscription formula characterizes the models of $F$ in which the extents of the predicates $\mathbf{p}$ are minimal. Thus the question that we are interested in can be stated as follows: Under what conditions is $\mathrm{CIRC}_{\mathbf{p}}[F]$ entailed by $\mathrm{SM}_{\mathbf{p}}[F]$?

## 2 Review: Circumscription and Stable Models

This review follows [Ferraris *et al.*, 2010]. We assume that, in the definition of a formula, the propositional connectives $\bot$ (falsity), $\wedge$, $\vee$, $\rightarrow$ are chosen as primitives, and $\neg F$ is treated as an abbreviation for $F \rightarrow \bot$.

Notation: if $p$ and $q$ are predicate constants of the same arity then $p \leq q$ stands for the formula $\forall \mathbf{x}(p(\mathbf{x}) \rightarrow q(\mathbf{x}))$, where $\mathbf{x}$ is a tuple of distinct object

[1] See `http://www.tcs.hut.fi/Software/smodels/lparse.ps` for a description of the language.

variables. If $\mathbf{p}$ and $\mathbf{q}$ are tuples $p_1, \dots, p_n$ and $q_1, \dots, q_n$ of predicate constants then $\mathbf{p} \leq \mathbf{q}$ stands for the conjunction

$$(p_1 \leq q_1) \wedge \cdots \wedge (p_n \leq q_n),$$

and $\mathbf{p} < \mathbf{q}$ stands for $(\mathbf{p} \leq \mathbf{q}) \wedge \neg(\mathbf{q} \leq \mathbf{p})$. In second-order logic, we apply the same notation to tuples of predicate variables.

Let $\mathbf{p}$ be a list of distinct predicate constants.[2] The *circumscription operator with the minimized predicates* $\mathbf{p}$, denoted by $\mathrm{CIRC}_{\mathbf{p}}$, is defined as follows: for any first-order sentence $F$, $\mathrm{CIRC}_{\mathbf{p}}[F]$ is the second-order sentence

$$F \wedge \neg\exists\mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F(\mathbf{u})),$$

where $\mathbf{u}$ is a list of distinct predicate variables of the same length as $\mathbf{p}$, and $F(\mathbf{u})$ is the formula obtained from $F$ by substituting the variables $\mathbf{u}$ for the constants $\mathbf{p}$. Models of $\mathrm{CIRC}_{\mathbf{p}}[F]$ will be called $\mathbf{p}$-*minimal* models of $F$.

Let $\mathbf{p}$ be a list of distinct predicate constants $p_1, \dots, p_n$. The *stable model operator with the intensional predicates* $\mathbf{p}$, denoted by $\mathrm{SM}_{\mathbf{p}}$, is defined as follows: for any first-order sentence $F$, $\mathrm{SM}_{\mathbf{p}}[F]$ is the second-order sentence

$$F \wedge \neg\exists\mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})),$$

where $\mathbf{u}$ is a list of $n$ distinct predicate variables $u_1, \dots, u_n$, and $F^*(\mathbf{u})$ is defined recursively:

– $p_i(\mathbf{t})^* = u_i(\mathbf{t})$ for any tuple $\mathbf{t}$ of terms;
– $F^* = F$ for any atomic formula $F$ that does not contain members of $\mathbf{p}$;[3]
– $(F \wedge G)^* = F^* \wedge G^*$;
– $(F \vee G)^* = F^* \vee G^*$;
– $(F \rightarrow G)^* = (F^* \rightarrow G^*) \wedge (F \rightarrow G)$;
– $(\forall x F)^* = \forall x F^*$;
– $(\exists x F)^* = \exists x F^*$.

Models of $\mathrm{SM}_{\mathbf{p}}[F]$ will be called $\mathbf{p}$-*stable* models of $F$.

It is clear that if $F$ does not contain implication then $F^*(\mathbf{u})$ is identical to $F(\mathbf{u})$, $\mathrm{SM}_{\mathbf{p}}[F]$ is identical to $\mathrm{CIRC}_{\mathbf{p}}[F]$, and the class of $\mathbf{p}$-stable models of $F$ is identical to the class of $\mathbf{p}$-minimal models of $F$.

**Example 1.** If $F$ is (1) then $\mathrm{CIRC}_{pq}[F]$ is

$$(\neg q \rightarrow p) \wedge \neg\exists uv(((u,v) < (p,q)) \wedge (\neg v \rightarrow u))$$

($u$, $v$ are propositional variables). This formula is equivalent to

$$(p \wedge \neg q) \vee (\neg p \wedge q),$$

[2] In this note, equality is not considered a predicate constant, so that it is not allowed to be a member of $\mathbf{p}$.

[3] This includes the case when $F$ is $\bot$.

so that the $pq$-minimal models of (1) are $\{p\}$ and $\{q\}$. To apply the operator $\mathrm{SM}_{pq}$ to formula (1) we need to remember that this formula is shorthand for

$$(q \to \bot) \to p.$$

So $F^*(u,v)$ is

$$(((v \to \bot) \wedge (q \to \bot)) \to u) \wedge ((q \to \bot) \to p),$$

which can be abbreviated as

$$((\neg v \wedge \neg q) \to u) \wedge (\neg q \to p).$$

Then $\mathrm{SM}_{pq}[F]$ is

$$(\neg q \to p) \wedge \neg \exists uv(((u,v) < (p,q)) \wedge ((\neg v \wedge \neg q) \to u) \wedge (\neg q \to p)).$$

This formula is equivalent to $p \wedge \neg q$, so that the only $pq$-stable model of (1) is $\{p\}$.

**Example 2.** Let $F$ be the formula

$$\forall xy(p(x,y) \to q(x,y)) \wedge \forall xyz(q(x,y) \wedge q(y,z) \to q(x,z)).$$

Then $\mathrm{CIRC}_q[F]$ is the conjunction of $F$ with the formula

$$\neg \exists u((u < q) \wedge \forall xy(p(x,y) \to u(x,y)) \wedge \forall xyz(u(x,y) \wedge u(y,z) \to u(x,z)))$$

($u$ is a binary predicate variable). This conjunction expresses that $q$ is the transitive closure of $p$. Furthermore, $\mathrm{SM}_q[F]$ is the conjunction of $F$ with the formula

$$\begin{aligned}\neg \exists u((u < q) & \wedge \forall xy(p(x,y) \to u(x,y)) \\ & \wedge \forall xy(p(x,y) \to q(x,y)) \\ & \wedge \forall xyz(u(x,y) \wedge u(y,z) \to u(x,z)) \\ & \wedge \forall xyz(q(x,y) \wedge q(y,z) \to q(x,z))).\end{aligned}$$

This conjunction is equivalent to $\mathrm{CIRC}_q[F]$, and consequently it expresses the same relationship: $q$ is the transitive closure of $p$.[4]

## 3 Critical Subformulas

Recall that an occurrence of a symbol in a formula is called *strictly positive* if it does not belong to the antecedent of any implication.[5]

About an occurrence of a formula $G$ in a first-order formula $F$ we say that it is *critical* if it is the antecedent of an implication $G \to H$ in $F$, and this implication

[4] Formula $F$ in this example is a canonical theory in the sense of [Kim *et al.*, 2009, Section 4]. The fact that the stable models of this formula are identical to its minimal models is a special case of Proposition 2 from that paper.

[5] We do not add "and is not in a scope of a negation" because $\neg F$ is treated here as an abbreviation.

(i) is in the scope of a strictly positive $\vee$, or
(ii) is in the scope of a strictly positive $\exists$, or
(iii) belongs to the antecedents of at least two other implications.

**Theorem 1.** *For any first-order sentence $F$ and any list $\mathbf{p}$ of predicate constants, if members of $\mathbf{p}$ do not occur in critical subformulas of $F$ then every $\mathbf{p}$-stable model of $F$ is $\mathbf{p}$-minimal.*

In other words, under the condition on critical subformulas above, $\mathrm{SM}_{\mathbf{p}}[F]$ entails $\mathrm{CIRC}_{\mathbf{p}}[F]$.

For example, formula (1) has no critical subformulas, so that the condition in the statement of the theorem holds in this case trivially. The same can be said about the formula from Example 2.[6] The second occurrence of $p$ in (2) is critical (recall that $\neg p$ is shorthand for $p \to \bot$), as well as the first occurrence of $p$ in (3). Consequently neither (2) nor (3) is covered by our theorem, which could be expected: each of these formulas has a nonminimal stable model.

Logic programs in the sense of [Gelfond and Lifschitz, 1988] are, from the perspective of [Ferraris *et al.*, 2010], conjunctions of the universal closures of formulas of the form

$$L_1 \wedge \cdots \wedge L_m \to A,$$

where $L_1, \ldots, L_m$ $(m \geq 0)$ are literals, and $A$ is an atom. These formulas do not have critical subformulas. (Thus our Theorem 1 can be viewed as a generalization of Theorem 1 from [Gelfond and Lifschitz, 1988].) The same can be said about "disjunctive logic programs"—conjunctions of the universal closures of formulas of the form

$$L_1 \wedge \cdots \wedge L_m \to A_1 \vee \cdots \vee A_n, \tag{4}$$

where $L_1, \ldots, L_m$ are literals, and $A_1, \ldots, A_n$ are atoms $(m, n \geq 0)$.

Proposition 1 below gives an example of a formula without critical subformulas that is "very different" from disjunctive logic programs. Recall that first-order formulas $F$ and $G$ are *strongly equivalent* to each other if, for any formula $H$, any occurrence of $F$ in $H$, and any list $\mathbf{p}$ of distinct predicate constants, $\mathrm{SM}_{\mathbf{p}}[H]$ is equivalent to $\mathrm{SM}_{\mathbf{p}}[H']$, where $H'$ is obtained from $H$ by replacing the occurrence of $F$ by $G$ [Ferraris *et al.*, 2010, Section 5.2].

**Proposition 1.** *No conjunction of propositional formulas of the form (4) is strongly equivalent to $(p \to q) \to q$.*

If we drop any of conditions (i)–(iii) from the definition of a critical subformula then the assertion of the theorem will become incorrect. The need to include condition (i) is illustrated by formula (2). Formula (3) shows that (iii) is required. The need for (ii) follows from the following proposition:

[6] More generally, the examination of the definition of a canonical theory from [Kim *et al.*, 2009] shows that our condition on critical subformulas holds for all canonical theories.

**Proposition 2.** *Formula*

$$p(a) \wedge (q(a) \rightarrow p(b)) \wedge \exists x(p(x) \rightarrow q(x)) \tag{5}$$

*has a pq-stable model that is not pq-minimal.*

## 4 Proofs

In Lemmas 1–4, $F$ is a first-order formula, $\mathbf{p}$ is a tuple of distinct predicate constants, and $\mathbf{u}$ is a tuple of distinct predicate variables of the same length as $\mathbf{p}$.

**Lemma 1.** *If $F$ does not contain members of $\mathbf{p}$ then $F^*(\mathbf{u})$ is equivalent to $F$.*

*Proof.* Immediate by structural induction.

**Lemma 2.** *If all occurrences of members of $\mathbf{p}$ in $F$ are strictly positive then the formula*

$$(\mathbf{u} \leq \mathbf{p}) \wedge F(\mathbf{u}) \rightarrow F^*(\mathbf{u})$$

*is logically valid.*

*Proof.* By structural induction. The only nontrivial case is when $F$ has the form $G \rightarrow H$; $G$ does not contain members of $\mathbf{p}$, and all occurrences of members of $\mathbf{p}$ in $H$ are strictly positive. By the induction hypothesis, the formula

$$(\mathbf{u} \leq \mathbf{p}) \wedge H(\mathbf{u}) \rightarrow H^*(\mathbf{u}) \tag{6}$$

is logically valid. Assume $(\mathbf{u} \leq \mathbf{p}) \wedge F(\mathbf{u})$, that is,

$$(\mathbf{u} \leq \mathbf{p}) \wedge (G \rightarrow H(\mathbf{u})). \tag{7}$$

We need to derive $F^*(\mathbf{u})$, that is,

$$(G^*(\mathbf{u}) \rightarrow H^*(\mathbf{u})) \wedge (G \rightarrow H).$$

In view of Lemma 1, this formula is equivalent to

$$G \rightarrow (H^*(\mathbf{u}) \wedge H).$$

Assume $G$. Then, by (7),

$$(\mathbf{u} \leq \mathbf{p}) \wedge H(\mathbf{u}),$$

and, by (6), $H^*(\mathbf{u})$. The formula

$$(\mathbf{u} \leq \mathbf{p}) \wedge H^*(\mathbf{u}) \rightarrow H \tag{8}$$

is logically valid [Ferraris *et al.*, 2010, Lemma 5]. Consequently $H$ follows as well.

**Lemma 3.** *If no occurrence of any member of* $\mathbf{p}$ *in* $F$ *belongs to the antecedent of more than one implication then the formula*

$$(\mathbf{u} \leq \mathbf{p}) \wedge F^*(\mathbf{u}) \rightarrow F(\mathbf{u})$$

*is logically valid.*

*Proof.* By structural induction. The only nontrivial case is when $F$ has the form $G \rightarrow H$; all occurrences of members of $\mathbf{p}$ in $G$ are strictly positive, and no occurrence of any member of $\mathbf{p}$ in $H$ belongs to the antecedent of more than one implication. By Lemma 2, the formula

$$(\mathbf{u} \leq \mathbf{p}) \wedge G(\mathbf{u}) \rightarrow G^*(\mathbf{u}) \tag{9}$$

is logically valid. By the induction hypothesis, the formula

$$(\mathbf{u} \leq \mathbf{p}) \wedge H^*(\mathbf{u}) \rightarrow H(\mathbf{u}) \tag{10}$$

is logically valid. Assume $(\mathbf{u} \leq \mathbf{p}) \wedge F^*(\mathbf{u})$, that is,

$$(\mathbf{u} \leq \mathbf{p}) \wedge (G^*(\mathbf{u}) \rightarrow H^*(\mathbf{u})) \wedge (G \rightarrow H). \tag{11}$$

Our goal is to prove $G(\mathbf{u}) \rightarrow H(\mathbf{u})$. From $G(\mathbf{u})$, the first conjunctive term of (11), and (9), $G^*(\mathbf{u})$. Then, by the second conjunctive term of (11), $H^*(\mathbf{u})$. Then $H(\mathbf{u})$ follows by (10).

**Lemma 4.** *If members of* $\mathbf{p}$ *do not occur in critical subformulas of* $F$ *then the formula*

$$\mathbf{u} \leq \mathbf{p} \wedge F \wedge F(\mathbf{u}) \rightarrow F^*(\mathbf{u})$$

*is logically valid.*

*Proof.* By induction on $F$. There are three nontrivial cases: when $F$ is $G \vee H$, $G \rightarrow H$, or $\exists x G(x)$. If $F$ is $G \vee H$ or $\exists x G(x)$ then the antecedents of all implications occurring in $F$ are critical and consequently do not contain members of $\mathbf{p}$. Thus all occurrences of members of $\mathbf{p}$ in $F$ are strictly positive, and the assertion to be proved follows from Lemma 2. Let $F$ be $G \rightarrow H$. In formula $G$, no occurrence of any member of $\mathbf{p}$ belongs to the antecedent of more than one implication, because otherwise the antecedent of the innermost implication containing that occurrence would be critical in $F$. By Lemma 3, it follows that the formula

$$(\mathbf{u} \leq \mathbf{p}) \wedge G^*(\mathbf{u}) \rightarrow G(\mathbf{u}) \tag{12}$$

is logically valid. By the induction hypothesis, the formula

$$\mathbf{u} \leq \mathbf{p} \wedge H \wedge H(\mathbf{u}) \rightarrow H^*(\mathbf{u}) \tag{13}$$

is logically valid. Assume

$$\mathbf{u} \leq \mathbf{p} \wedge F \wedge F(\mathbf{u}); \tag{14}$$

our goal is to derive $F^*(\mathbf{u})$, that is,

$$(G^*(\mathbf{u}) \rightarrow H^*(\mathbf{u})) \wedge F.$$

The second conjunctive term is immediate from (14). To prove the first conjunctive term, assume $G^*(\mathbf{u})$. Then, by the first conjunctive term of (14) and (12), $G(\mathbf{u})$. Consequently, by the third conjunctive term of (14), $H(\mathbf{u})$. On the other hand, the formula

$$(\mathbf{u} \leq \mathbf{p}) \wedge G^*(\mathbf{u}) \rightarrow G$$

is logically valid [Ferraris *et al.*, 2010, Lemma 5]; hence $G$, and, by the second conjunctive term of (14), $H$. Then, by (13), $H^*(\mathbf{u})$.

*Proof of Theorem 1.* Take a sentence $F$ such that members of $\mathbf{p}$ do not occur in critical subformulas of $F$. We need to show that $\mathrm{SM}_{\mathbf{p}}[F]$ entails $\mathrm{CIRC}_{\mathbf{p}}[F]$. Assume that

$$F \wedge \neg\exists\mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u})) \tag{15}$$

but

$$(\mathbf{u} < \mathbf{p}) \wedge F(\mathbf{u}).$$

Then, by Lemma 4, $F^*(\mathbf{u})$, which contradicts (15).

The proof of Proposition 1 below refers to reducts in the sense of [Ferraris, 2005]. It uses two facts about them. One is a characterization of strong equivalence in terms of reducts:

**Lemma 5.** *Propositional formulas $F$ and $G$ are strongly equivalent to each other iff, for every set $X$ of atoms, the reducts $F^X$ and $G^X$ are equivalent to each other in the sense of classical logic.*

This is part of the statement of Proposition 2 from [Ferraris, 2005].[7]

The second fact is a property of disjunctive logic programs:

**Lemma 6.** *If $\Pi$ is a conjunction of propositional formulas of form (4) then, for any sets $X$, $Y$ and $Z$ of atoms such that $X \subseteq Y \subseteq Z$, if $Z \models \Pi$ and $X \models \Pi^Y$ then $X \models \Pi^Z$.*

This observation is made in [Eiter *et al.*, 2005] (and expressed there using somewhat different terminology, in terms of SE-models instead of reducts).

*Proof of Proposition 1.* Let $F$ stand for $(p \rightarrow q) \rightarrow q$, and assume that $\Pi$ is a conjunction of propositional formulas of form (4) that is strongly equivalent to $F$. It is easy to check that $\{p, q\} \models F$ and that

$$\emptyset \models F^{\{p\}},\ \emptyset \not\models F^{\{p,q\}}.$$

[7] The definition of strong equivalence in [Ferraris, 2005] is somewhat different from the definition given above, which is taken from [Ferraris *et al.*, 2010]. But in application to propositional formulas the two definitions are equivalent to each other, because, as discussed in these papers, each definition is equivalent in this case to the provability of $F \leftrightarrow G$ in the logic of here-and-there.

It follows that $\{p, q\} \models \Pi$ (because strongly equivalent formulas are classically equivalent) and, by Lemma 5, that

$$\emptyset \models \Pi^{\{p\}},\ \emptyset \not\models \Pi^{\{p,q\}}.$$

But this is impossible by Lemma 6: take $X = \emptyset$, $Y = \{p\}$, and $Z = \{p, q\}$.

*Proof of Proposition 2 (Hint).* The Herbrand interpretation $\{p(a), p(b), q(a)\}$ is a $pq$-stable model of (5) that is not $pq$-minimal.

## 5 Conclusion

In this note, we gave a syntactic condition that ensures the minimality of all stable models of a first-order sentence. The condition is expressed in terms of critical subformulas. It shows that in the propositional case all possible exceptions to the general principle that stable models are minimal are similar to the examples given in the introduction: they contain an implication in the scope of a disjunction, as (2), or three implications nested within each other, as (3). In the presence of variables, the restriction on disjunctions has to be extended to existential quantifiers.

## Acknowledgements

We are grateful to Dan Lessin for useful discussions on the topic of this paper, and to Gerhard Brewka for comments on the previous version. The second author was supported by the National Science Foundation under grant IIS-0712113.

## References

[Apt *et al.*, 1988]Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann, San Mateo (1988)

[Bidoit and Froidevaux, 1987]Bidoit, N., Froidevaux, C.: Minimalism subsumes default logic and circumscription in stratified logic programming. In: Proceedings LICS 1987, pp. 89–97 (1987)

[Eiter *et al.*, 2005]Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer-set programming. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp. 97–102 (2005)

[Ferraris *et al.*, 2010]Ferraris, P., Lee, J., Lifschitz, V.: Stable models and circumscription[8]. Artificial Intelligence 175, 236–263 (2011)

[Ferraris, 2005]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)

[8] `http://peace.eas.asu.edu/joolee/papers/smcirc.pdf`

[Gelfond and Lifschitz, 1988]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)

[Gelfond, 1987]Gelfond, M.: On stratified autoepistemic theories. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp. 207–211 (1987)

[Kim *et al.*, 2009]Kim, T.-W., Lee, J., Palla, R.: Circumscriptive event calculus as answer set programming. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp. 823–829 (2009)

[McCarthy, 1986]McCarthy, J.: Applications of circumscription to formalizing common sense knowledge. Artificial Intelligence 26(3), 89–116 (1986)

[Van Gelder *et al.*, 1988]Van Gelder, A., Ross, K., Schlipf, J.: Unfounded sets and well-founded semantics for general logic programs. In: Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas, March 21-23, pp. 221–230. ACM Press, New York (1988)

[Van Gelder, 1988]Van Gelder, A.: Negation as failure using tight derivations for general logic programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 149–176. Morgan Kaufmann, San Mateo (1988)

# Challenges in Answer Set Solving

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub*

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

**Abstract.** Michael Gelfond's application of Answer Set Programming (ASP) in the context of NASA's space shuttle has opened the door of the ivory tower. His project has not only given our community self-confidence and served us as a reference for grant agencies and neighboring fields, but ultimately it helped freeing the penguins making them exclaim *"Yes, we can* [fly] *!"*. The community has taken up this wonderful assist to establish ASP as a prime tool for declarative problem solving in the area of Knowledge Representation and Reasoning. Despite this success, however, ASP has not yet attracted broad attention outside this area. This paper aims at identifying some current challenges that our field has to overcome in the mid-run to ultimately become a full-fledged technology in Informatics.

## 1 Introduction

A central goal of the field of Knowledge Representation and Reasoning is to furnish methods for automated yet declarative problem solving. Unlike programming, where the idea is to use programs to specify how a problem is to be solved, the idea is to view a program as a formal representation of the problem as such. Accordingly, the mere execution of a traditional program is replaced by an automated search through the solution space spanned by the problem's representation. Looking at chess, this amounts to specifying the rules of chess and searching in the resulting state space rather than writing a chess program. In the latter case, the intelligence lies with the programmer, while in the former it is the system that cares about finding a solution in a smart way. Also, the procedural approach is bound to playing chess, while the problem-solving oriented approach is free to play any other game that can be specified in the realm of the input language, as witnessed in the area of General Game Playing [1].

Answer Set Programming (ASP; [2]) is nowadays one of the most popular approaches to declarative problem solving. This is due to its appealing combination of a rich yet simple modeling language with high-performance solving capacities. ASP has its roots in Knowledge Representation and (Nonmonotonic) Reasoning, Logic Programming (with negation), Databases, and Boolean Constraint Solving. ASP allows for solving all search problems in *NP* (and $NP^{NP}$) in a uniform way, offering more succinct problem representations than propositional logic [3]. Meanwhile, ASP has been used in many application areas, among them, product configuration [4], decision support for NASA shuttle controllers [5], composition of Renaissance music [6], synthesis of multiprocessor systems [7], reasoning tools in systems biology [8, 9], team-building [10],

* Affiliated with the School of Computing Science at Simon Fraser University, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Australia.

M. Balduccini and T.C. Son (Eds.): Gelfond Festschrift, LNAI 6565, pp. 74–90, 2011.

and many more.[1] The success story of ASP has its roots in the early availability of ASP solvers, beginning with the *smodels* system [11], followed by *dlv* [12], SAT-based ASP solvers, like *assat* [13] and *cmodels* [14], and the conflict-driven ASP solver *clasp*, demonstrating the performance and versatility of ASP solvers by winning first places at international competitions like ASP'09, PB'09, and SAT'09.

Despite this success, ASP has not yet become an out-of-the-box technology like, for instance Prolog, or even a natural subject of undergraduate teaching, like relational databases or functional programming. A major prerequisite for this ambitious goal is to enable the scalability of ASP, even when used by non-expert users. However, ASP's declarative methodology for problem posing and solving does not scale for free. Actually, the decent performance of ASP systems often conceals a lack of scalability and has so far impeded the community's awareness of this limitation. In what follows, we aim at sharpening the community's awareness of the challenges ahead of us and to motivate a joined effort in addressing them. Enjoy!

## 2 ASP Solving

As with traditional computer programming, the ASP solving process amounts to a closed loop. Its steps can be roughly classified into

1. Modeling,
2. Grounding,
3. Solving,
4. Visualizing, and
5. Software Engineering.

We have illustrated this process in Figure 1 by giving the associated components. It all starts with a modeling phase, which results in a first throw at a representation of the given problem in terms of logic programming rules. The resulting program[2] is usually formulated by means of first-order variables, which are systematically replaced by elements of the Herbrand universe in a subsequent grounding phase. This yields a finite propositional program that is then fed into the actual ASP solver. The output of the solver varies depending on the respective reasoning mode. Often, it consists of a textual representation of a sequence of answer sets. Depending on the quality of the resulting answer, one then either refines the last version of the problem representation or not.

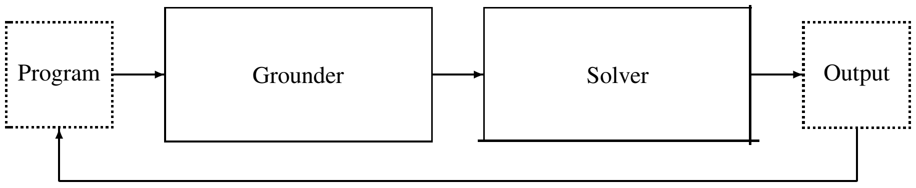


**Fig. 1.** ASP Solving Process

[1] See `http://www.cs.uni-potsdam.de/~torsten/asp` for an extended listing of ASP applications.

[2] This is of course a misnomer but historically too well established to be dropped.

As pointed out in the introductory section, the strongholds of ASP are usually regarded to be its rich modeling language as well as its high-performance solving capacities. We revisit both topics, Modeling and Solving, in the following sections and close by commenting briefly on the remaining issues.

Grounding, or more generally propositionalization, is common to all approaches exploiting the computational advantages of propositional representations; besides ASP this includes, for instance, Satisfiability Checking (SAT;[15]) and Planning [16]. Unlike the latter, however, ASP is the only area having widely used, highly optimized grounding systems. Even so, some problems are as a matter of fact prone to a combinatorial blow-up in space and thus beyond the realm of any efficient grounding procedure. Given that such problems usually comprise large but flat domains, there is definitely a need to further investigate database techniques for grounding or finite-domain constraint processing techniques for hybrid solving, as done in [17–19].

Visualization becomes indispensable when problems and their resulting answer sets involve a high number of entities and relations among them. Although the relational nature of ASP suggests tables as an obvious means of visualization, at the end of the day, tables are just another domain-independent way of characterizing the actual objects of interest. But despite this application-specific nature of visualization, it remains a great challenge whether a highly declarative, general-purpose paradigm such as ASP can be lifted from a purely textual level towards a more appealing graphical stage. A first approach in this direction is described in [20].

Software-engineering becomes more and more important to ASP in view of its increasing range of applications. Among others, this involves effective development tools, including editors and debuggers, as well as the dissemination of (open-source) tools and libraries connecting ASP to other computing paradigms. For instance, classical debugging techniques do often not apply to ASP because of its high degree of declarativity, or in other words, its lack of a procedural semantics that could be subject to debugging and tracing. This "curse of declarativity" is well recognized withing the ASP community and addressed within a dedicated workshop series [21, 22]; first approaches can be found in [23–26].

## 3 Modeling

ASP Modeling is an art; it requires craft, experience, and knowledge. Although the resulting problem specifications are usually quite succinct and easy to understand, crafting such beautiful specification that also shows its best performance is not as obvious as it might seem. To illustrate this, we conduct a little case study in the next section.

All experiments were conducted with `gringo` (3.0.0[3]) and `clasp` (1.3.4).

### 3.1 A Case-Study

Let us consider the well-known $n$-Queens problem that consists of placing $n$ queens on an $n \times n$-square board such that no queen may attacks another one.

[3] The release candidate was referred to as `bingo`.

Following the common *generate-and-test* methodology of ASP, this problem can be specified in four rules, the first providing a generator positioning $n$ queens on the $n \times n$ board, and the three remaining ones excluding two queens on the same row, column, and diagonal, respectively. The first throw at a formalization of these constraints in ASP is given in Table 1.

**Table 1.** $n$-Queens problem, first throw

```
% place n queens on the chess board
n { q(1..n,1..n) } n.

% at most one queen per row/column
:- q(X,Y1), q(X,Y2), Y1 != Y2.
:- q(X1,Y), q(X2,Y), X1 != X2.

% at most one queen per diagonal
:- q(X1,Y1), q(X2,Y2), X1 != X2, #abs(X1-X2) == #abs(Y1-Y2).
```

A first improvement is to eliminate symmetric ground rules, expressing the same constraint. For example, rule `:- q(X,Y1), q(X,Y2), Y1 != Y2.` gives rise to ground instances `:- q(3,1), q(3,2).` and `:- q(3,2), q(3,1).` both of which prohibit the same placements of queens. This redundancy can be removed by some simple symmetry breaking. In our example, it suffices to replace inequality `Y1 != Y2` by `Y1 < Y2`. Globally applying this simple way of symmetry breaking to the encoding in Table 1 yields the one in Table 2. The latter encoding strictly halves the number of ground instances obtained from the three integrity constraints. For instance, on the 10-Queens problem, the number of ground rules drops from 2941 to 1471. Despite this reduction, the improved encoding still scales poorly, as witnessed by the 1646701 rules obtained after 28.26s on the 100-Queens problem (cf. Table 5 at the end of this section).

**Table 2.** $n$-Queens problem, second throw

```
% place n queens on the chess board
n { q(1..n,1..n) } n.

% at most one queen per row/column
:- q(X,Y1), q(X,Y2), Y1 < Y2.
:- q(X1,Y), q(X2,Y), X1 < X2.

% at most one queen per diagonal
:- q(X1,Y1), q(X2,Y2), X1 < X2, #abs(X1-X2) == #abs(Y1-Y2).
```

Analyzing the encoding in Table 2 a bit further reveals that all three integrity constraints give rise to a cubic number of ground instances, that is, on the $n$-Queens problem they produce $O(n^3)$ ground rules. This can be drastically reduced by replacing the rule restricting placements in rows, viz. `:- q(X,Y1), q(X,Y2), Y1 < Y2.`, by[4]

```
:- X = 1..n, not 1 { q(X,Y) } 1.
```

asserting that there is exactly one queen in a row. One rule per row, results in $O(n)$ rules (each of size $O(n)$) rather than $O(n^3)$ as before. Clearly, the same can be done for columns, yielding `:- Y = 1..n, not 1 { q(X,Y) } 1.` Note that the new rules imply that there is *exactly one* queen per row and column, respectively. Hence, we may replace the cardinality constraint `n { q(1..n,1..n) } n.` by the unconstrained choice `{ q(1..n,1..n) }`. This is advantageous because it constitutes practically no constraint for `clasp`. Finally, what can we do about the integrity constraint controlling diagonal placements? It fact, the same aggregation can be done for the diagonals, once we have an enumeration scheme. The idea is to enumerate diagonals in two ways, once from the upper left corner to the lower right corner, and similarly from the upper right corner to the lower left corner. Let us illustrate this for $n = 4$:

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 |
| 2 | 2 | 3 | 4 | 5 |
| 3 | 3 | 4 | 5 | 6 |
| 4 | 4 | 5 | 6 | 7 |

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 4 | 3 | 2 | 1 |
| 2 | 5 | 4 | 3 | 2 |
| 3 | 6 | 5 | 4 | 3 |
| 4 | 7 | 6 | 5 | 4 |

These two enumeration schemes can be captured by the equations $D = X + Y - 1$ and $D = X - Y + n$, respectively. For instance, the first equation tells us that diagonal 6 consists of positions $(4, 3)$ and $(3, 4)$. Given both equations, we may replace the rule restricting placements in diagonals by the two following rules:

```
:- D = 1..n*2-1, not { q(X,Y) : D==X-Y+n } 1.
:- D = 1..n*2-1, not { q(X,Y) : D==X+Y-1 } 1.
```

As above, we thus obtain one rule per diagonal, inducing $O(n)$ ground rules (each of size $O(n)$). The resulting encoding is given in Table 3.

For 10 and 100 queens, the encoding of Table 3 yields 55 and 595 ground rules, respectively, in contrast to the 1471 and 1646701 rules obtained with the encoding in Table 2. Despite the much smaller grounding size, however, the grounding time does not scale as expected. To see this, note that grounding the encoding in Table 3 for 100 queens takes less than second, while 500 queens require more than 100 seconds of grounding time (although only 2995 ground rules are produced).

Further investigations[5] reveal that the last two rules in Table 3 are the source of the problem. In fact, it turns out that during grounding the tests `D==X-Y+n` and `D==X-Y-1` are repeated over and over. This can be avoided by pre-calculating both conditions. To this end, we add the rules

[4] The construct `X = 1..n` can be read as $X \in \{1, \ldots, n\}$.

[5] This can be done with `gringo`'s debug option `--verbose`.

```
d1(X,Y,X-Y+n) :- X = 1..n, Y = 1..n.
d2(X,Y,X+Y-1) :- X = 1..n, Y = 1..n.
```

and replace the two conditions `D==X-Y+n` and `D==X-Y-1` by `d1(X,Y,D)` and `d2(X,Y,D)`, respectively. The resulting encoding is given in Table 4. Although this encoding adds a quadratic number of facts, their computation is linear and exploits indexing techniques known from database systems.

**Table 3.** $n$-Queens problem, third throw

```
% place n queens on the chess board
{ q(1..n,1..n) }.

% exactly one queen per row/column
:- X = 1..n, not 1 { q(X,Y) } 1.
:- Y = 1..n, not 1 { q(X,Y) } 1.

% at most one queen per diagonal
:- D = 1..n*2-1, not { q(X,Y) : D==X-Y+n } 1.
:- D = 1..n*2-1, not { q(X,Y) : D==X+Y-1 } 1.
```

**Table 4.** $n$-Queens problem, fourth throw

```
% place n queens on the chess board
{ q(1..n,1..n) }.

% exactly one queen per row/column
:- X = 1..n, not 1 { q(X,Y) } 1.
:- Y = 1..n, not 1 { q(X,Y) } 1.

% pre-calculate the diagonals
d1(X,Y,X-Y+n) :- X = 1..n, Y = 1..n.
d2(X,Y,X+Y-1) :- X = 1..n, Y = 1..n.

% at most one queen per diagonal
:- D = 1..n*2-1, not { q(X,Y) : d1(X,Y,D) } 1.
:- D = 1..n*2-1, not { q(X,Y) : d2(X,Y,D) } 1.
```

**Table 5.** Experiments contrasting different encodings of the $n$-Queens problem. All runs conducted with `clasp --heuristic=vsids --quiet`.

| $n$ | Encoding 1 | | Encoding 2 | | Encoding 3 | | Encoding 4 | |
|---|---|---|---|---|---|---|---|---|
| 50 | 2.95 | 42.10 | 1.95 | 41.16 | 0.12 | 0.04 | 0.05 | 0.05 |
| 100 | 41.50 | — | 28.26 | — | 0.81 | 0.16 | 0.13 | 0.18 |
| 500 | — | — | — | — | 96.91 | 16.34 | 3.60 | 16.84 |
| 1000 | — | — | — | — | 767.70 | 166.80 | 20.98 | 168.75 |

### 3.2 Some Hints on (Manual) Modeling

Finally, let us give some hints on modeling based upon our experience.

1. Keep the grounding compact
   – If possible, use aggregates
   – Try to avoid combinatorial blow-up
   – Project out unused variables
   – But don't remove too many inferences!
2. Add additional constraints to prune the search space
   – Consider special cases
   – Break symmetries
   – ...
   – Test whether the additional constraints really help
3. Try different approaches to model the problem
   – Problems involving time steps might be parallelized
4. It (still) helps to know the systems
   – `gringo` offers options to trace the grounding process
   – `clasp` offers many options to configure the search[6]

### 3.3 Non-ground Pre-processing

We have conducted a preliminary case-study illustrating the potential of non-ground pre-processing techniques. To this end, we explored two simple techniques.[7]

**Concretion** The idea of concretion is to replace overly general rules by their effectively used partial instantiations. In other words, concretion eliminates redundant rule instances from the program whenever their contribution is re-constructable from an answer set and not needed otherwise. Consider the following simple program.

```
q(X,Y) :- p(X), p(Y).
r(X)   :- q(X,X).
```

Given that the binary predicate `q` is only used with identical arguments, concretion replaces the first rule by

```
q(X,X) :- p(X).
```

Similarly, concretion replaces the first rule in

```
q(X,X) :- p(X).
r(X)   :- q(X,2).
```

by

```
q(2,2) :- p(2).
```

[6] `clasp` was run with `--heuristic=vsids` to solve the large $n$-Queens problem.
[7] We are grateful to Michael Grosshans and Arne König for accomplishing this case-study!

Note that concretion does not preserve answer sets. However, the original answer sets can be reconstructed from the resulting ones by means of the original program.

**Projection** aims at reducing the number of variables in a rule in order to scale down the number of its ground instances. To this end, one eliminates variables with singleton (or say "isolated") occurrences and replaces the encompassing literal(s) with a new literal only containing the remaining variables.

For illustration, consider the rule

```
q(X) :- z(X,W), v(X,Y,Z,0), u(Z,W).
```

In this rule, variable `Y` is irrelevant to the remainder of the rule and can thus be eliminated by projection. As a result, projection replaces the above rule by the two following ones.

```
q(X) :- z(X,W), v_new(X,Z), u(Z,W).
v_new(X,Z) :- v(X,Y,Z,0).
```

The predicate `v_new` yields the value combinations necessary for deriving instances of `q`. Note that this reduces the number of variables from four to three, which may significantly reduce the number of ground instances depending on the size of the respective domains.

Projection was first applied to ASP in [27], tracing back to well-known database techniques [28, p. 176].

Similar and often much more refined techniques can be found in the literature, however, frequently in different research areas, like (deductive) databases, traditional logic programming, automated planning, etc.

As a proof-of-concept, we have implemented both techniques in the prototypical grounder *pyngo*[8] [29] and conducted some preliminary experiments. First and foremost, it is worth mentioning that both techniques are useless for common benchmarks because most of them have been designed by experts in ASP. For instance, both techniques cannot really improve the encodings furnished during the last modeling-oriented ASP competition [30]. Hence, our experiment design rather aimed at use-cases covering non-expert usage of ASP.

The two use-cases envisaged for concretion are the usage of library programs in more specific contexts. To this end, we took encodings necessitating complex sub-problems. The first benchmark set computes for each element of the residue class ring modulo $n$ the multiplicative inverse, while the second one takes numbers $n$ and $a$ and conducts the Solovay-Strassen primality test. Note that the benchmarks involve computing the greatest common divisor, quadratic remainders, and potencies in the residue class rings.

Table 6(a) summarizes the results obtained on both benchmarks. We measured the run-time of *clingo* restricted to 120sec on a 3.6GHz PC running Linux. Comparing the run-times on the original encoding with those obtained after applying concretion

[8] *pyngo* is a Python-based grounder, developed by Arne König for rapid prototyping of grounder features.

**Table 6.** Experimental results applying concretion

(a) Multiplicative Inverse

| $n$ | Original | Transform |
|---|---|---|
| 50 | 0.680 | 0.010 |
| 100 | 10.690 | 0.060 |
| 200 | – | 0.210 |
| 500 | – | 2.030 |
| 1000 | – | 15.490 |
| 1500 | – | 51.000 |
| 2000 | – | 114.240 |
| 2500 | – | – |

(b) Solovay-Streets Test

| $n$ | Original | Transform |
|---|---|---|
| 100 | 0.130 | 0.000 |
| 500 | 12.650 | 0.050 |
| 1000 | 97.880 | 0.120 |
| 2000 | – | 0.390 |
| 5000 | – | 2.410 |
| 10000 | – | 9.590 |
| 20000 | – | 37.370 |
| 50000 | – | – |

(indicated as ‘Transform’ in Table 6(a)), we observe a clear improvement after pre-processing. In this case, this betterment is due to the fact that all of the aforementioned sub-problems were subject to concretion.

Our second experiment deals with single-player games from the area of General Game Playing [1] and aims at illustrating the potential of projection. All benchmarks were obtained through automatic transformations from original specifications in the Game Description Language [31] into ASP. This provides us with a benchmark set not at all designed for ASP and rather comparable to Prolog programs (over a finite Herbrand base).

Table 7 summarizes our results. This time we distinguish between grounding and solving time, and provide as well the size of the ground program. This is interesting because the program modifications of projection are more substantial, and may thus have different effects. As before, all benchmarks were run on a 3.6GHz PC under Linux yet now with a timeout of 1200sec; the grounding size is given in MB. We observe in 22 out of 32 benchmarks an improvement that usually affected both grounding as well as solving time. A remarkable decrease was observed on *Sudoku*, where grounding time and size was reduced by two orders of magnitude and solving time dropped from over 20min, viz. the cut-off time, to a bit more than 4sec. But despite these ameliorations, projection can also lead to a deterioration of performance, as drastically shown on *God*, where projection increased the grounding by an order of magnitude and pushed solving time beyond the cut-off.

All in all, our preliminary case-study demonstrates the great potential of automatic non-ground pre-processing techniques for improving ASP code. Moreover, it revealed significant research challenges in identifying not only more such pre-processing techniques but furthermore in gearing them towards true improvements.

## 4 Solving

Advanced Boolean Constraint Solving is sensitive to parameter tuning. Clearly, this carries over to modern ASP Solving. In fact, an ASP Solver like `clasp` offers an arsenal of parameters for controlling the search for answer sets. Choosing the right parameters often makes the difference between being able to solve a problem or not.

**Table 7.** Experimental results applying projection

| | Original | | | Transform | | |
|---|---|---|---|---|---|---|
| Game | Grounding | Size | Solving | Grounding | Size | Solving |
| 8puzzle | 1.59 | 9.8 | 74.37 | 0.19 | 1.4 | 6.90 |
| aipsrovers | 0.23 | 1.9 | 1.16 | 0.20 | 1.9 | 1.08 |
| asteroids | 0.07 | 0.5 | 1.15 | 0.12 | 0.8 | 1.73 |
| asteroidsparallel | 0.17 | 1.2 | 53.38 | 0.21 | 1.6 | 39.02 |
| asteroidsserial | 0.40 | 2.8 | 14.62 | 0.49 | 4.0 | 4.81 |
| blocksworldparallel | 0.11 | 0.8 | 0.08 | 0.04 | 0.3 | 0.01 |
| brainteaser | 0.03 | 0.1 | 0.02 | 0.07 | 0.2 | 0.23 |
| chinesecheckers | 12.35 | 96.9 | 21.71 | 11.36 | 92.9 | 12.23 |
| circlesolitaire | 0.06 | 0.4 | 0.07 | 0.05 | 0.3 | 0.04 |
| coins | 0.04 | 0.2 | 0.02 | 0.03 | 0.2 | 0.03 |
| firefighter | 0.09 | 0.7 | 0.04 | 0.05 | 0.4 | 0.02 |
| god | 38.76 | 354.8 | 349.8 | 1151.91 | 426.5 | – |
| hanoi6 | 1.29 | 9.6 | - | 1.36 | 11.0 | – |
| hanoi7(1) | 7.83 | 44.0 | - | 8.04 | 52.0 | – |
| hanoi7(2) | 7.84 | 44.0 | 4.83 | 8.14 | 52.0 | 7.42 |
| hanoi | 0.20 | 1.8 | - | 0.26 | 2.2 | 16.03 |
| incredible | 0.31 | 2.2 | 2.28 | 0.08 | 0.6 | 0.12 |
| knightmove | 0.24 | 1.7 | - | 0.25 | 1.8 | 1031.80 |
| lightsout | 0.04 | 0.1 | 2.16 | 0.02 | 0.2 | 15.73 |
| maxknights | 1.10 | 7.6 | 0.79 | 0.58 | 3.7 | 0.42 |
| pancakes6 | 39.99 | 325.7 | 737.33 | 40.11 | 325.8 | 631.70 |
| pancakes | 43.72 | 325.7 | 556.22 | 39.28 | 325.8 | 465.92 |
| peg(1) | 64.42 | 509.2 | - | 3.43 | 30.7 | – |
| peg(2) | 66.68 | 509.2 | - | 3.24 | 30.7 | – |
| queens | 2.36 | 13.3 | 36.58 | 5.81 | 34.3 | 29.26 |
| slidingpieces | 3.93 | 24.6 | 2.79 | 3.53 | 32.5 | 4.66 |
| snake2008 | 0.59 | 4.2 | 27.18 | 0.66 | 4.8 | 5.50 |
| snake2009 | 0.94 | 6.5 | 542.57 | 0.85 | 6.3 | – |
| sudoku | 221.94 | 1643.8 | - | 3.64 | 34.1 | 4.30 |
| tpeg | 65.84 | 509.1 | - | 3.10 | 31.5 | – |
| troublemaker | 0.03 | 0.1 | 0.04 | 0.01 | 0.1 | 0.00 |
| twistypassage | 0.93 | 5.9 | 1.27 | 0.83 | 6.8 | 0.66 |

### 4.1 Another Case-Study

Let us analyze the performance of `clasp` in the context of the NP problems used at the 2009 ASP Solver Competition [30]. To this end, we begin with contrasting the default configuration of `clasp` with a slightly changed configuration denoted by `clasp`$^+$. The latter invokes `clasp` with options `--sat-prepro` and `--trans-ext=dynamic`. For comparison, we also give results for ASP solvers `cmodels` [32] and `smodels` [11].[9] All experiments were run on an Intel Quad-Core Xeon E5520, possessing 2.27GHz processors, under Linux. Each benchmark instance was run three times with every solver, each individual run restricted to 600 seconds

[9] Version information is given below Table 8.

**Table 8.** Solving the 2009 ASP Competition (NP problems)

| Benchmark | # | | clasp | | clasp$^+$ | | cmodels[m] | | smodels | |
|---|---|---|---|---|---|---|---|---|---|---|
| *15Puzzle* | 16 | (16/0) | 33.01 | (0) | 20.18 | (0) | 31.36 | (0) | 600.00 | (48) |
| *BlockedNQueens* | 29 | (15/14) | 5.09 | (0) | 4.91 | (0) | 9.04 | (0) | 29.37 | (0) |
| *ChannelRouting* | 10 | (6/4) | 120.13 | (6) | 120.14 | (6) | 120.58 | (6) | 120.90 | (6) |
| *EdgeMatching* | 29 | (29/0) | 0.23 | (0) | 0.41 | (0) | 59.32 | (0) | 60.32 | (0) |
| *Fastfood* | 29 | (10/19) | 1.17 | (0) | 0.90 | (0) | 29.22 | (0) | 83.93 | (3) |
| *GraphColouring* | 29 | (9/20) | 421.55 | (60) | 357.88 | (39) | 422.66 | (57) | 453.77 | (63) |
| *Hanoi* | 15 | (15/0) | 11.76 | (0) | 3.97 | (0) | 2.92 | (0) | 523.77 | (39) |
| *HierarchicalClustering* | 12 | (8/4) | 0.16 | (0) | 0.17 | (0) | 0.76 | (0) | 1.56 | (0) |
| *SchurNumbers* | 29 | (13/16) | 17.44 | (0) | 49.60 | (0) | 75.70 | (0) | 504.17 | (72) |
| *Solitaire* | 27 | (22/5) | 204.78 | (27) | 162.82 | (21) | 175.69 | (21) | 316.96 | (36) |
| *Sudoku* | 10 | (10/0) | 0.15 | (0) | 0.16 | (0) | 2.55 | (0) | 0.25 | (0) |
| *WeightBoundedDomSet* | 29 | (29/0) | 123.13 | (15) | 102.18 | (12) | 300.26 | (36) | 400.84 | (51) |
| $\varnothing(\Sigma)$ *(tight)* | 264 | (182/82) | *78.22* | *(108)* | *68.61* | *(78)* | 102.50 | (120) | 257.99 | (318) |
| *ConnectedDomSet* | 21 | (10/11) | 40.42 | (3) | 36.11 | (3) | 7.46 | (0) | 183.76 | (15) |
| *GeneralizedSlitherlink* | 29 | (29/0) | 0.10 | (0) | 0.22 | (0) | 1.92 | (0) | 0.16 | (0) |
| *GraphPartitioning* | 13 | (6/7) | 9.27 | (0) | 7.98 | (0) | 20.19 | (0) | 92.10 | (3) |
| *HamiltonianPath* | 29 | (29/0) | 0.07 | (0) | 0.06 | (0) | 0.21 | (0) | 2.22 | (0) |
| *KnightTour* | 10 | (10/0) | 124.29 | (6) | 91.80 | (3) | 242.48 | (12) | 150.55 | (3) |
| *Labyrinth* | 29 | (29/0) | 123.82 | (12) | 82.92 | (6) | 142.24 | (6) | 594.10 | (81) |
| *MazeGeneration* | 29 | (10/19) | 91.17 | (12) | 89.89 | (12) | 90.41 | (12) | 293.62 | (42) |
| *Sokoban* | 29 | (9/20) | 0.73 | (0) | 0.80 | (0) | 3.39 | (0) | 176.01 | (15) |
| *TravellingSalesperson* | 29 | (29/0) | 0.05 | (0) | 0.06 | (0) | 317.82 | (7) | 0.22 | (0) |
| *WireRouting* | 23 | (12/11) | 42.81 | (3) | 36.36 | (3) | 175.73 | (12) | 448.32 | (45) |
| $\varnothing(\Sigma)$ *(nontight)* | 241 | (173/68) | *43.27* | *(36)* | *34.62* | *(27)* | 100.19 | (49) | 194.11 | (204) |
| $\varnothing(\Sigma)$ | 505 | (355/150) | *62.33* | *(144)* | *53.16* | *(105)* | 101.45 | (169) | 228.95 | (522) |

clasp (1.3.1)
clasp$^+$ = clasp --sat-prepro --trans-ext=dynamic
cmodels[m] (3.79 with minisat 2.0)
smodels (2.34 with option -restart)

and 2GB RAM. Our experiments are summarized in Table 8, giving average runtimes in seconds (and numbers of timed-out runs in parentheses) for every solver on each benchmark class, with timeouts taken as 600 seconds. The table gives in the column headed by # the number of instances per benchmark class. In addition, Table 8 provides the respective partition into satisfiable and unsatisfiable instances in parentheses. The rows marked with $\varnothing(\Sigma)$ provide the average runtimes and number of timeouts wrt the considered collection of benchmark classes.

We see that the performance of clasp's default configuration is quite inferior to that of clasp$^+$ on this set of benchmarks. In fact, almost all benchmark classes contain extended rules. However, not all of them are substantial enough to warrant a dedicated treatment. This situation is accounted for by the configuration of clasp$^+$, using a hybrid treatment of extended rules. The option --trans-ext=dynamic excludes "small" extended rules from an intrinsic treatment (cf. [33]) and rather transforms them into normal ones. This results in a higher number of Boolean constraints, which is counterbalanced by invoking option `--sat-prepro` that enables Resolution-based pre-processing [34]. This greatly reduces the number of timeouts, namely, from 144 benchmarks unsolved by the default configuration to 105 unsolved ones by `clasp`$^+$.

Let us get a closer look at three classes that were difficult for `clasp`. To this end, it is important to get a good idea about the features of the considered benchmark class. This involves static properties of the benchmark as such as well as dynamic features reflecting its solving process.

The *WeightBoundedDomSet* benchmark class consists of tight, rather small, unstructured logic programs having many solutions. The two latter features often suggest a more aggressive restart strategy, making the solver explore an increased number of different locations in the search space rather than performing fewer yet more exhaustive explorations.

Indeed, invoking `clasp` (1.3.1) with `--restarts=256`, indicated by `clasp`$^\star$ below, yields an average Time of 4.64s (versus 123.13s) and makes the number of timeouts drop from 15 to zero.

| Benchmark | # | `clasp` | `clasp`$^+$ | `clasp`$^\star$ | `cmodels[m]` | `smodels` |
|---|---|---|---|---|---|---|
| *WBDS* | 29 (29/0) | 123.13(15) | 102.18(12) | 4.64 (0) | 300.26 (36) | 400.84(51) |

The *ConnectedDomSet* benchmark class consists of non-tight, rather small logic programs containing a single large integrity cardinality constraint. The difficulty in solving this class lies in the latter integrity constraint. In fact, the default configuration of `clasp` treats extended rules as special Boolean constraints rather then initially compiling them into normal rules. However, on this benchmark the conflict learning scheme of `clasp` spends a lot of time extracting all implicit conflicts comprised in this constraint. Unlike `clasp` (and `smodels`), `cmodels` unwraps these conflicts when compiling them into normal rules, so that its solving process needs not spend any time in recovering them.

This behavior is nicely reflected by invoking `clasp` (1.3.1) with Option `--trans-ext=weight`[10], indicated by `clasp`$^\star$ below, yielding an average time of 4.19s (versus 40.42s) and no timeouts (versus 3).

| Benchmark | # | `clasp` | `clasp`$^+$ | `clasp`$^\star$ | `cmodels[m]` | `smodels` |
|---|---|---|---|---|---|---|
| *ConnectedDomSet* | 21 (10/11) | 40.42(3) | 36.11 (3) | 4.19 (0) | 7.46 (0) | 183.76(15) |

The *KnightTour* benchmark class consists of non-tight logic programs containing many large cardinality constraints and exhibiting many large back-jumps during solving. The latter runtime feature normally calls for progress saving [35] enforcing the same truth assignment to atoms chosen on subsequent descents into the search space. Also, it is known from the SAT literature that this works best in combination with an aggressive restart strategy.

In fact, invoking `clasp` (1.3.1) with `--restarts=256` and `--save-progress`, indicated by `clasp`$^\star$ below, reduces the average time to 1.47s (versus 124.29s) and leaves us with no timeouts (versus 6).

[10] `--trans-ext=integrity`, if using `clasp` 1.3.3 or later versions.

| Benchmark | # | clasp | clasp$^+$ | clasp$^\star$ | cmodels[m] | smodels |
|---|---|---|---|---|---|---|
| *KnightTour* | 10 (10/0) | 124.29(6) | 91.80 (3) | 1.47 (0) | 242.48 (12) | 150.55 (3) |

## 4.2 Some Hints on (Manual) Solving

The question then arises how to deal with this vast "configuration space" and how to conciliate it with the idea of declarative problem solving. Currently, there seems to be no true alternative to manual fine-tuning when addressing highly demanding application problems.

As rules of thumb, we usually start by investigating the following options:

- `--heuristic`: Try *vsids* instead of `clasp`'s default *berkmin*-style heuristic.
- `--trans-ext`: Applicable if a program contains extended rules, that is, rules including cardinality and weight constraints. Try at least the *dynamic* transformation.
- `--sat-prepro`: Resolution-based preprocessing works best on tight programs with few cardinality and weight constraints. It should (almost) always be used if extended rules are transformed into normal ones (via `--trans-ext`).
- `--restarts`: Try aggressive restart policies, like *Luby-256* or the *nested policy*, or try disabling restarts whenever a problem is deemed to be unsatisfiable.
- `--save-progress`: Progress saving typically works nicely if the average backjump length (or the #choices/#conflicts ratio) is high ($\geq$10). It usually performs best if combined with aggressive restarts.

## 4.3 Portfolio-Based Solving

A first step to overcome the sensitivity of modern ASP solvers to parameter settings and thus to regain a certain degree of declarativity in solving is to use a portfolio-based approach to ASP solving. The general idea is to identify a set of different solving approaches, either different systems, configurations, or both, and to harness existing machine learning techniques to build a classifier, mapping benchmark instances to the putatively best solver configurations. Such approaches have already shown their versatility in neighboring areas such as Constraint and SAT solving [36, 37].

This idea resulted in the portfolio-based ASP-solver `claspfolio` [29], winning the first place in the category "Single-System Teams" at the Second ASP competition [30].[11] Unlike other heterogeneous approaches using distinct systems in their portfolio, `claspfolio` takes advantage of `clasp`'s manifold search gearing options to identify a portfolio of different configurations. This has originally resulted in 30 different `clasp` configurations that were run on an extensive set of benchmarks. These results are then used to select a reduced portfolio of "interesting" settings by eliminating configurations whose exclusion does not significantly decrease the quality of the overall selection. This resulted in 12 configurations on which a support-vector machine is trained to predict the potentially fastest configuration for an arbitrary benchmark instance. To this end, we extract from each training instance 140 static and dynamic

[11] `claspfolio` was developed by Stefan Ziller.

features by appeal to `claspre`. While the static features, like number of rule types or tightness, are obtained while processing the input benchmark, the dynamic ones are obtained through a limited run of `clasp`, observing features like number of learned nogoods or average length of back-jumps. Once the support-vector machines are established, a typical run of `claspfolio` starts by launching `claspre` for feature extraction upon which the support-vector machines select the most promising solver configuration to launch `clasp`. The option `--mode=su` also allows for a two-step classification by first predicting whether the instance is SAT, UNSAT, or unknown and then selecting the best configuration among specific SAT-, UNSAT-, and remaining portfolios, respectively.

The computational impact of this approach can be seen by contrasting the performance of `claspfolio` (0.8) with that of `clasp`'s default (1.3.4), its best but fixed configuration, a random selection among the portfolio, and the virtually best `clasp` version obtained my taking for each benchmark the minimum run-time among all solvers in the portfolio. Considering all systems on a set of 2771 benchmark instances restricted to 1200s, we observed an average run-time of 87.95s for `clasp`'s default configuration (and 127 timeouts); `clasp`'s best configuration took 70,42s (79 timeouts), while the virtually best solver among all 30 configurations spent 20,46s and that among the portfolio 24.61s on average (both trivially without timeouts).[12] While a random selection among the portfolio configurations run 97.89s on average (145 timeouts), the trained approach of `claspfolio` used 38.85s of which it spent 37.38s on solving only (with 22 timeouts). `claspfolio` has thus a clear edge over `clasp`'s best but rigid configuration.

All in all, `claspfolio` takes some burden of parameter tuning away from us and lets us concentrate more on problem posing. Nonetheless real applications still need manual interference. In this respect, `claspfolio` can be used as a first guide indicating which search parameters are most promising for attacking an application at hand.[13] The true research challenge however lies in getting a solid understanding in the link between problem features and search parameters.

## 5 Conclusion

ASP has come a long way. Having its roots in Nonmonotonic Reasoning [38], we can be proud of having taught Tweety how to fly. We have build impressive systems by drawing on solid formal foundations. And although there is still a long way to go to establish ASP among the standard technologies in Informatics, the future is bright and conceals many interesting research challenges.

Thank you Michael (and Vladimir) for putting us on the right track!

*Acknowledgments.* This work was supported by the German Science Foundation (DFG) under grants SCHA 550/8-1 and -2.

[12] Learning is restricted to benchmarks solvable by at least one configuration.

[13] For instance, `claspfolio --dir=<dir> --skip-solving --fstats` provides a ranking of the best `clasp` configuration on the benchmark instances in directory `<dir>`.

## References

1. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
3. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic 7(2), 261–268 (2006)
4. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 305–319. Springer, Heidelberg (1999)
5. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-prolog decision support system for the space shuttle. In: Ramakrishnan, I. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)
6. Boenn, G., Brain, M., de Vos, M., Fitch, J.: Automatic composition of melodic and harmonic music by answer set programming. In: [39], pp. 160–174
7. Ishebabi, H., Mahr, P., Bobda, C., Gebser, M., Schaub, T.: Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. Journal of Reconfigurable Computing (2009)
8. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. In: [40], pp. 436–441
9. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. Theory and Practice of Logic Programming 11(2), 1–38 (2011)
10. Grasso, G., Iiritano, S., Leone, N., Lio, V., Ricca, F., Scalise, F.: An ASP-based system for team-building in the gioia-tauro seaport. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 40–42. Springer, Heidelberg (2010)
11. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
13. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence 157(1-2), 115–137 (2004)
14. 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)
15. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
16. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers, San Francisco (2004)
17. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence 53(1-4), 251–287 (2008)
18. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: [41], pp. 235–249
19. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. In: Theory and Practice of Logic Programming. Twenty-sixth International Conference on Logic Programming (ICLP 2010) Special Issue, vol. 10(4-6), pp. 465–480. Cambridge University Press, Cambridge (2010)

20. Cliffe, O., de Vos, M., Brain, M., Padget, J.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: [39], pp. 724–728
21. de Vos, M., Schaub, T. (eds.): Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA 2007). Number CSBU-2007-05 in Department of Computer Science, University of Bath, Technical Report Series (2007) ISSN 1740-9497
22. de Vos, M., Schaub, T. (eds.): Proceedings of the Second Workshop on Software Engineering for Answer Set Programming (SEA 2009). Department of Computer Science, University of Bath, Technical Report Series (2009)
23. Brain, M., de Vos, M.: Debugging logic programs under the answer set semantics. In: de Vos, M., Provetti, A. (eds.) Proceedings of the Third International Workshop on Answer Set Programming (ASP 2005). CEUR Workshop Proceedings (CEUR-WS.org), vol. 142, pp. 141–152 (2005)
24. Pontelli, E., Son, T.C.: Justifications for logic programs under answer set semantics. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 196–210. Springer, Heidelberg (2006)
25. 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)
26. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: [40], pp. 448–453
27. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using database optimization techniques for nonmonotonic reasoning. In: Proceedings of the Seventh International Workshop on Deductive Databases and Logic Programming (DDLP 1999), pp. 135–139 (1999)
28. Ullman, J.: Principles of Database and Knowledge-Base Systems. Computer Science Press, Rockville (1988)
29. http://potassco.sourceforge.net/
30. 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)
31. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Technical Report LG-2006-01, Stanford University (March 2008)
32. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36(4), 345–377 (2006)
33. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. In: [41], pp. 250–264
34. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
35. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
36. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Bridge, D., Brown, K., O'Sullivan, B., Sorensen, H. (eds.) Proceedings of the Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science, AICS 2008 (2008)
37. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32, 565–606 (2008)

38. Ginsberg, M. (ed.): Readings in Nonmonotonic Reasoning. Morgan Kaufmann, San Francisco (1987)
39. Garcia de la Banda, M., Pontelli, E. (eds.): ICLP 2008. LNCS, vol. 5366. Springer, Heidelberg (2008)
40. Fox, D., Gomes, C. (eds.): Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI 2008). AAAI Press, Menlo Park (2008)
41. Hill, P.M., Warren, D.S. (eds.): ICLP 2009. LNCS, vol. 5649. Springer, Heidelberg (2009)

# Exploring Relations between Answer Set Programs[⋆]

Katsumi Inoue[1] and Chiaki Sakama[2]

[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
[2] Department of Computer and Communication Sciences, Wakayama University
Sakaedani, Wakayama 640-8510, Japan

**Abstract.** Equivalence and generality relations over logic programs have been proposed in answer set programming to semantically compare information contents of logic programs. In this paper, we overview previous relations of answer set programs, and propose a general framework that subsumes previous relations. The proposed framework allows us to compare programs possibly having non-minimal answer sets as well as to explore new relations between programs. Such new relations include relativized variants of generality relations over logic programs. By selecting contexts for comparison, the proposed framework can represent weak, strong and uniform variants of generality, inclusion and equivalence relations. These new relations can be applied to comparison of abductive logic programs and coordination of multiple answer set programs.

## 1 Introduction

Relations between theories have attracted a lot of interests in design and maintenance of knowledge bases. In particular, the notion of *equivalence* is important for verification, simplification and optimization of logic programs. Maher [26] introduced the notion of *equivalence as program segments* for definite programs, which has later been explored as *strong equivalence* for answer set programming by Lifschitz *et al.* [21]. Sagiv [37] defined the notion of *uniform equivalence* of Datalog programs as equivalence of the outputs of programs for any set of input atoms, which has also been extended for answer set programming by Eiter and Fink [4]. These two notions of equivalence are then unified in a more general concept called *relative/relativized equivalence* [25,15,49,7,31,50,48], which characterizes equivalence with respect to *contexts*. By changing contexts, we can consider various forms of equivalence relations in answer set programming.

Inter-theory relations between logic programs are not necessarily limited to equivalence. We also would like to know what kind of relations hold between programs when they are not equivalent. Eiter *et al.* [7] define a general correspondence framework to compare answer sets of programs not only under

⋆ This research is supported in part by the 2008-2011 JSPS Grant-in-Aid for Scientific Research (A) No. 20240016.

equivalence but under *inclusion*, and Oetsch *et al.* [31] extend the framework to the uniform variant. Inoue and Sakama [17] define another inter-theory relation called *generality* for answer set programming. Their main concern is to compare the amounts of information brought by logic programs and thus to assess the relative value of each program. It turns out that inclusion by [7] is a special case of generality.

Generality relations have many important applications. First, generality has been central in the theory of *inductive logic programming* (ILP) [34,28,29] as a criterion to rank hypotheses whenever alternative hypotheses can be considered. Intuitively, a program $P_1$ is considered *more general* than a program $P_2$ if $P_1$ brings more information than $P_2$. In *learning nonmonotonic theories* [13,32,39,45], hypotheses are constructed in the form of logic programs with negation as failure. Then, we need to rank such hypotheses according to some generality relations. Secondly, answer set programming is used for describing and refining knowledge bases related to *ontologies* [6]. In this process, generality relations can be used as criteria to judge whether descriptions are really refined [18]. Thirdly, integration, merging, and coordination of multiple descriptions are important in *multi-agent systems*; one may modify her own description in accordance with those from other agents by constructing a *minimal generalization* or a *maximal specialization* of multiple descriptions [44], or may *compose a new description* by gathering agents' descriptions [42], or may build a *consensus* between agents' beliefs [43]. These combination operations can be formalized as *minimal upper* or *maximal lower bounds* of multiple programs under generality relations [17].

In this paper, we will consider stronger versions of generality. The idea is to introduce some focused contexts for comparison, which makes two programs comparable under particular program extensions. As in the case of equivalence, *strong* or *uniform* variants of generality is considered to represent a situation that a generality relation between two programs is preserved after adding any set of rules or facts, respectively. Moreover, these two notions of generality can be unified into the notion of *relativized generality* as in the case of relativized equivalence. Motivation behind this relativized generality lies in the fact that we learn from it *robustness* of generality between programs: one program brings more information than another whatever rules in a focused context are added to those programs. More justifications for consideration of relativized generality will be discussed from the viewpoint of *inductive generality.* As an application of relativized generality, we will show that explanatory power of *abductive logic programs* [20] can be compared using the proposed framework. Although the notion of *abductive generality* is defined in [19] from the viewpoint of abductive explanations, we will see that it can be characterized as a special case of relativized generality. Further applications of this generality would be logical foundations of coordination between multiple *abductive agents*.

Note that the generality relations are defined in [17] for *extended disjunctive programs* using the Smyth ($\sharp$) and Hoare ($\flat$) orderings from domain theory [11], which respectively reflect orderings on inclusion relations of the *skeptical* and *credulous* consequences. Those previous results require an anti-chain property for the answer sets of programs, which limits the applicability of the framework to EDPs. In this paper, these previous orderings as well as newly proposed orderings can be applied to more extended classes of programs that may possess non-minimal answer sets [24,40,14,23,8], as well as logic programs with aggregates [46,27,47]. Abductive logic programs [20] can also be considered to have non-minimal answer sets due to augmentation with abducibles even when the background program is an EDP.

The rest of this paper is organized as follows. Section 2 defines $\sharp$- and $\flat$-generality relations over the class of all sets of literal sets. Section 3 applies these generality relations to ordering logic programs. Section 4 considers strong, uniform and relativized variants of generality relations and analyzes their properties. Section 5 relates relativized generality with abductive generality. Section 6 gives related work and Section 7 summarizes the paper.

## 2 Generality Relations over Semantic Structures

This section presents generality relations defined over the sets of literal sets. We recall some mathematical definitions about domains [11]. A *pre-order* (or *quasi-order*) $\succcurlyeq$ is a binary relation which is reflexive and transitive. A pre-order $\succcurlyeq$ is a *partial order* if it is also anti-symmetric. A *pre-ordered set* (resp. *partially ordered set*; *poset*) is a set $D$ with a pre-order (resp. partial order) $\succcurlyeq$ on $D$. For a pre-ordered set $\langle D, \succcurlyeq \rangle$ and $x, y \in D$, we write $x \succ y$ if $x \succcurlyeq y$ and $y \not\succcurlyeq x$.

For a pre-ordered set $\langle D, \succcurlyeq \rangle$ and any set $X \subseteq D$, we denote the minimal and maximal elements of $X$ as follows.

$$min_{\succcurlyeq}(X) = \{\, x \in X \mid \neg\exists y \in X.\, x \succ y \,\},$$
$$max_{\succcurlyeq}(X) = \{\, x \in X \mid \neg\exists y \in X.\, y \succ x \,\}.$$

We often denote these as $min(X)$ and $max(X)$ by omitting $\succcurlyeq$. We also assume that the relation $\succcurlyeq$ is well-founded (resp. upwards well-founded) on $D$[1] whenever $min_{\succcurlyeq}(X)$ (resp. $max_{\succcurlyeq}(X)$) is concerned in order to guarantee the existence of a minimal (resp. maximal) element of any $X \subseteq D$.[2] Note that, when $D$ is finite, any pre-order is both well-founded and upwards well-founded on $D$.

For any set $D$, let $\mathcal{P}(D)$ be the powerset of $D$. Given a pre-ordered set $\langle D, \succcurlyeq \rangle$ and $X, Y \in \mathcal{P}(D)$, the *Smyth order* is defined as

$$X \succeq^{\sharp} Y \quad \text{iff} \quad \forall x \in X\, \exists y \in Y.\, x \succcurlyeq y\,,$$

and the *Hoare order* is defined as

$$X \succeq^{\flat} Y \quad \text{iff} \quad \forall y \in Y\, \exists x \in X.\, x \succcurlyeq y\,.$$

[1] A relation $R$ is *well-founded* on a class $D$ iff every non-empty subset of $D$ has a minimal element with respect to $R$. A relation $R$ is *upwards well-founded* on $D$ iff the inverse relation $R^{-1}$ is well-founded on $D$.

[2] For example, Theorems 2.1 (2), 2.2 (2) and 3.2 (2) assume the upwards well-foundedness. Note that the relation $\subseteq$ is well-founded on $D$ even if $D$ is infinite.

The relations $\succeq^\sharp$ and $\succeq^\flat$ are pre-orders on $\mathcal{P}(D)$, and both $\langle \mathcal{P}(D), \succeq^\sharp \rangle$ and $\langle \mathcal{P}(D), \succeq^\flat \rangle$ are pre-ordered sets. Note that the orderings $\succeq^\sharp$ and $\succeq^\flat$ are slightly different from those in domain theory:[3] we allow the empty set $\emptyset$ ($\in \mathcal{P}(D)$) as both the top element $\top^\sharp$ in $\langle \mathcal{P}(D), \succeq^\sharp \rangle$ and the bottom element $\bot^\flat$ in $\langle \mathcal{P}(D), \succeq^\flat \rangle$.

**Example 2.1.** Consider the poset $\langle \mathcal{P}(\{p,q\}), \supseteq \rangle$. Then, we have $\{\{p,q\}\} \succeq^\sharp \{\{p\}\} \succeq^\sharp \{\{p\},\{q\}\}$. On the other hand, $\{\{p,q\}\} \succeq^\flat \{\{p\},\{q\}\} \succeq^\flat \{\{p\}\}$. Since $\{\emptyset, \{p\}\} \succeq^\sharp \{\emptyset, \{q\}\} \succeq^\sharp \{\emptyset, \{p\}\}$ holds, $\succeq^\sharp$ is not a partial order.

We now define orderings over the sets of literal sets. We assume a first-order language, and denote by *Lit* the set of all ground literals in the language. A literal set $T \subseteq Lit$ is *inconsistent* if $T$ contains a pair of complementary literals $L, \neg L$; otherwise, $T$ is *consistent*. The *(logical) closure* of $T$ is defined as

$$cl(T) = \begin{cases} T\,, & \text{if } T \text{ is consistent;} \\ Lit\,, & \text{if } T \text{ is inconsistent.} \end{cases}$$

A literal set $T$ is *closed* if $T = cl(T)$ holds.

We define a semantic structure called a *composite set* as an element in $\mathcal{P}(\mathcal{P}(Lit))$, i.e., a class of sets of ground literals from *Lit*. A composite set is *consistent* if it contains a consistent literal set; otherwise, it is *inconsistent*. Hence, if a composite set $\Sigma$ is inconsistent, then either $\Sigma = \emptyset$ or every set $T \in \Sigma$ is inconsistent. We denote the closures of a composite set $\Sigma$ as $Cl(\Sigma) = \{cl(T) \mid T \in \Sigma\}$. Two composite sets $\Sigma_1$ and $\Sigma_2$ are *equivalent*, denoted as $\Sigma_1 \equiv \Sigma_2$, if $Cl(\Sigma_1) = Cl(\Sigma_2)$. A composite set $\Sigma$ is *closed* if $\Sigma = Cl(\Sigma)$ holds. Given a pre-order $\succcurlyeq$, a composite set $\Sigma$ is *irredundant (with respect to $\succcurlyeq$)* if $\Sigma$ is an anti-chain on the set $\langle \mathcal{P}(Lit), \succcurlyeq \rangle$, that is, for any $S, T \in \Sigma$, $S \succcurlyeq T$ implies $T \succcurlyeq S$.

To define generality ordering over composite sets, let us assume a pre-ordered set $\langle D, \succcurlyeq \rangle$ such that the domain $D$ is $\mathcal{P}(Lit)$, i.e., the class of sets of ground literals in the language, and the pre-order $\succcurlyeq$ is the *inclusion* relation $\supseteq$ over $\mathcal{P}(Lit)$. We denote by $\mathcal{LS}$ the class of all composite sets which can be constructed in the language. That is, $\mathcal{LS} = \mathcal{P}(\mathcal{P}(Lit))$. Then, the Smyth and Hoare orderings on $\mathcal{P}(\mathcal{P}(Lit))$ can be defined, which enable us to order composite sets.

**Definition 2.1.** Assume a pre-ordered set $\langle \mathcal{P}(Lit), \supseteq \rangle$, and let $\Sigma_1$ and $\Sigma_2$ be any two composite sets in $\mathcal{LS}$. $\Sigma_1$ is *more $\sharp$-general than (or equal to)* $\Sigma_2$, written $\Sigma_1 \models^\sharp \Sigma_2$, if $Cl(\Sigma_1) \succeq^\sharp Cl(\Sigma_2)$. $\Sigma_1$ is *more $\flat$-general than (or equal to)* $\Sigma_2$, written $\Sigma_1 \models^\flat \Sigma_2$, if $Cl(\Sigma_1) \succeq^\flat Cl(\Sigma_2)$.

By definition, $\sharp$- and $\flat$-generality reflect the following situations. $\Sigma_1 \models^\sharp \Sigma_2$ means that any set in $\Sigma_1$ has logically more information than (or equal to) some set in $\Sigma_2$. On the other hand, $\Sigma_1 \models^\flat \Sigma_2$ means that any set in $\Sigma_2$ has logically less information than (or equal to) some set in $\Sigma_1$. For notational convenience, we

[3] This is because we enable comparison of all classes of programs by associating $\emptyset$ with the class of programs having no answer set [17].

write $\models^{\sharp/\flat}$ to represent the $\sharp$- or $\flat$-generality relations together, which denotes either $\models^{\sharp}$ or $\models^{\flat}$. It is easy to see that $\langle \mathcal{LS}, \models^{\sharp/\flat} \rangle$ is a pre-ordered set.

In the following theorems, we assume a pre-ordered set $\langle \mathcal{P}(Lit), \supseteq \rangle$, and let $\Sigma_1, \Sigma_2 \in \mathcal{LS}$. The next theorem shows that the minimal (resp. maximal) elements determine the $\models^{\sharp}$ (resp. $\models^{\flat}$) relation between composite sets.

**Theorem 2.1.** (1) *$\Sigma_1 \models^{\sharp} \Sigma_2$ iff $min(\Sigma_1) \models^{\sharp} min(\Sigma_2)$.*
(2) *$\Sigma_1 \models^{\flat} \Sigma_2$ iff $max(\Sigma_1) \models^{\flat} max(\Sigma_2)$.*

*Proof.* We prove (1) but (2) can be proved in the same way. Suppose that $\Sigma_1 \models^{\sharp} \Sigma_2$. Then, $\forall S_1 \in min(Cl(\Sigma_1))$, $\exists S_2 \in Cl(\Sigma_2)$ such that $S_1 \supseteq S_2$, and then $\exists S_2' \in min(Cl(\Sigma_2))$ such that $S_2 \supseteq S_2'$. Hence, $min(\Sigma_1) \models^{\sharp} min(\Sigma_2)$. Conversely, suppose that $min(\Sigma_1) \models^{\sharp} min(\Sigma_2)$. For any $T_1 \in Cl(\Sigma_1)$, $\exists T_1' \in min(Cl(\Sigma_1))$ such that $T_1 \supseteq T_1'$. Then by the supposition, for $T_1'$, $\exists T_2 \in min(Cl(\Sigma_2))$ such that $T_1' \supseteq T_2$. Then, $T_2 \in Cl(\Sigma_2)$ by the definition of $min$. Hence, $\Sigma_1 \models^{\sharp} \Sigma_2$. □

**Theorem 2.2.** (1) *$\Sigma_1 \models^{\sharp} \Sigma_2$ and $\Sigma_2 \models^{\sharp} \Sigma_1$ iff $min(\Sigma_1) \equiv min(\Sigma_2)$.*
(2) *$\Sigma_1 \models^{\flat} \Sigma_2$ and $\Sigma_2 \models^{\flat} \Sigma_1$ iff $max(\Sigma_1) \equiv max(\Sigma_2)$.*

*Proof.* (1) By Theorem 2.1 (1), $\Sigma_1 \models^{\sharp} \Sigma_2$ and $\Sigma_2 \models^{\sharp} \Sigma_1$ iff (i) $min(\Sigma_1) \models^{\sharp} min(\Sigma_2)$ and (ii) $min(\Sigma_2) \models^{\sharp} min(\Sigma_1)$. By (i), $\forall S_1 \in Cl(min(\Sigma_1))$, $\exists S_2 \in Cl(min(\Sigma_2))$ such that $S_1 \supseteq S_2$. Then by (ii), $\exists S_1' \in Cl(min(\Sigma_1))$ such that $S_2 \supseteq S_1'$. Then, $S_1 \supseteq S_1'$ holds, but both belong to $Cl(min(\Sigma_1)) = min(Cl(\Sigma_1))$. Hence, $S_1 = S_1' = S_2$. That is, $S_1 \in Cl(min(\Sigma_1))$ implies $S_1 \in Cl(min(\Sigma_2))$. Hence, $Cl(min(\Sigma_1)) \subseteq Cl(min(\Sigma_2))$. Similarly, $Cl(min(\Sigma_2)) \subseteq Cl(min(\Sigma_1))$.

(2) can also be proved in a similar way. □

**Corollary 2.3.** *For irredundant composite sets $\Sigma_1$ and $\Sigma_2$, the following three are equivalent:* (i) $\Sigma_1 \models^{\sharp} \Sigma_2 \models^{\sharp} \Sigma_1$; (ii) $\Sigma_1 \models^{\flat} \Sigma_2 \models^{\flat} \Sigma_1$; (iii) $\Sigma_1 \equiv \Sigma_2$.

*Proof.* If $\Sigma$ is an irredundant composite set, $max(\Sigma) = min(\Sigma) = \Sigma$ holds. Then the corollary holds by Theorem 2.2. □

The next property states that $\sharp$- and $\flat$-generality relations are *monotonic* with respect to addition of literal sets.

**Proposition 2.4.** *If $\Sigma_1 \supseteq \Sigma_2$, then $\Sigma_2 \models^{\sharp} \Sigma_1$ and $\Sigma_1 \models^{\flat} \Sigma_2$.*

In the following propositions, we assume a pre-ordered set $\langle \mathcal{P}(Lit), \supseteq \rangle$.

**Proposition 2.5 (Independence).** *Let $\Sigma_1$ and $\Sigma_2$ be composite sets, and $T_1$ and $T_2$ literal sets. If $\Sigma_1 \models^{\sharp/\flat} \Sigma_2$ and $T_1 \supseteq T_2$, then $\Sigma_1 \cup \{T_1\} \models^{\sharp/\flat} \Sigma_2 \cup \{T_2\}$.*

**Corollary 2.6 (Refinement).** *Let $\Sigma$ be a composite set, and $T$ a literal set such that $T \in \Sigma$. If $S$ is a literal set such that $S \supseteq T$, then $(\Sigma \setminus \{T\}) \cup \{S\} \models^{\sharp/\flat} \Sigma$.*

**Proposition 2.7 (Robustness).** *Let $\Sigma_1$, $\Sigma_2$ and $\Sigma_3$ be composite sets. If $\Sigma_1 \models^{\sharp/\flat} \Sigma_2$ and $\Sigma_1 \models^{\sharp/\flat} \Sigma_3$, then $\Sigma_1 \models^{\sharp/\flat} \Sigma_2 \cup \Sigma_3$.*

**Proposition 2.8 (Extended Equivalence).** *Let $\Sigma_1$, $\Sigma_2$, $\Pi_1$ and $\Pi_2$ be composite sets. If $\Sigma_1 \equiv \Sigma_2$ and $\Pi_1 \equiv \Pi_2$, then $\Sigma_1 \cup \Pi_1 \equiv \Sigma_2 \cup \Pi_2$.*

## 3 Ordering Logic Programs

This section applies the Smyth and Hoare orderings over $\mathcal{LS}$ to order logic programs. Generalizing the results for the class of extended disjunctive programs [17], we allow any class of programs possibly having non-minimal answer sets.

In this paper, we can compare any class of logic programs as long as the semantics of a program is defined as a *closed composite set*, which is a collection of literal sets called *answer sets* [10] or *stable models* [9] of the program.[4] For example, here we can consider a *general extended disjunctive program* (GEDP) [24,14], which is a set of *rules* of the form:

$$L_1 \,;\, \cdots \,;\, L_k \,;\, not\, L_{k+1} \,;\, \cdots \,;\, not\, L_l \;\leftarrow\; L_{l+1},\, \ldots,\, L_m,\, not\, L_{m+1},\, \ldots,\, not\, L_n \tag{1}$$

where $0 \leq k \leq l \leq m \leq n$, each $L_i$ is a literal, *not* is *negation as failure* (NAF), and ";" represents disjunction. A rule of the form (1) is an *(integrity) constraint* if $k = 0$, and is a *fact* if $k = l = m = n \geq 1$. A fact is *disjunctive* if $k > 1$. A fact is often denoted by omitting $\leftarrow$. A more general class of programs called *nested programs* [23] can be considered here, although any nested program is equivalent to some GEDP [23]. Similarly, any *propositional formula* can be used as a logic program [8]. Another important class of programs we can compare includes *logic programs with aggregates* [46,27,47]. Examples of subclasses of GEDPs are as follows. If any rule of the form (1) in a GEDP $P$ satisfies $k = l$, $P$ is called an *extended disjunctive program* (EDP) [10]. If any rule of the form (1) in an EDP $P$ satisfies $k = l \leq 1$, $P$ is called an *extended logic program* (ELP). An ELP $P$ is a *normal logic program* (NLP) if every literal appearing in $P$ is a positive literal. A program $P$ is *NAF-free* if every rule of the form (1) in $P$ never contains *not*, i.e., $k = l$ and $m = n$. A NAF-free program $P$ is a *positive disjunctive program* (PDP) if every literal appearing in $P$ is a positive literal. A PDP $P$ is a *definite program* if $k = 1$ holds for any rule in $P$.

The definition of *answer sets* for each class of programs can be referred to the original papers mentioned above, so we will omit it here. The set of all answer sets of a program $P$ is written as $A(P)$. A program $P$ is *consistent* if $A(P)$ is consistent. It is known that $A(P)$ is irredundant for any EDP $P$, that is, every answer set of an EDP is *minimal* [10]. On the other hand, answer sets of GEDPs, nested programs and logic programs with aggregates are non-minimal in general. Notice also that we can consider any semantics other than the original answer set semantics for a logic program as long as it is defined as a closed composite set. Then, this setting allows comparison of EDPs under the *possible model semantics* [40] or the *supported model semantics* [14,48], which has representation of non-minimal literal sets. Moreover, when PDPs $P_1$ and $P_2$ are compared, we can even consider their *classical models* $Mod(P_i)$ $(i = 1, 2)$ instead of their minimal models that are answer sets.

In the answer set semantics, a program containing variables is a shorthand of its ground instantiation. In the case of non-ground programs, however, we need

[4] *Stable models* are used to refer to answer sets of logic programs in which explicit negation ($\neg$) does not appear, and are sets of atoms instead of literals in this paper.

to assume that the relation $\supseteq$ is upwards well-founded on $\mathcal{P}(Lit)$ whenever the maximal elements are concerned (see Footnotes 1 and 2 in Section 2). Also the issue of decidability is of concern for non-ground programs [5,22,30]. To simplify the discussion, we assume that programs in this paper are finite propositional.

We denote by $\mathcal{LP}$ the class of all programs which can be constructed in the language. A subclass of programs is then denoted as $\mathcal{C} \subseteq \mathcal{LP}$. Examples of such subclasses $\mathcal{C}$ are the class of all programs $\mathcal{LP}$, the class of EDPs, the class of programs constructed with a subset of $Lit$, a set of programs, and a set of literal sets. The following correspondence notions have been proposed in the literature.

**Definition 3.1.** Let $P_1, P_2 \in \mathcal{LP}$ be programs, and $\mathcal{C} \subseteq \mathcal{LP}$.

(1) $P_1$ and $P_2$ are *(weakly) equivalent*, written $P_1 \equiv P_2$, if $A(P_1) = A(P_2)$ holds.
(2) $P_1$ and $P_2$ are *strongly equivalent*, written $P_1 \equiv_s P_2$, if $A(P_1 \cup R) = A(P_2 \cup R)$ holds for any program $R \in \mathcal{LP}$ [21].
(3) $P_1$ and $P_2$ are *uniformly equivalent*, written $P_1 \equiv_u P_2$, if $A(P_1 \cup U) = A(P_2 \cup U)$ holds for any literal set $U \in \mathcal{P}(Lit)$ [4].
(4) $P_1$ and $P_2$ are *(strongly) equivalent with respect to* $\mathcal{C}$, written $P_1 \equiv_{\mathcal{C}} P_2$, if $A(P_1 \cup R) = A(P_2 \cup R)$ holds for any $R \in \mathcal{C}$ [15].
(5) $P_1$ *ans-includes* $P_2$, written $P_1 \sqsupseteq P_2$, if $A(P_1) \supseteq A(P_2)$ holds [7].
(6) $P_1$ *strongly ans-includes* $P_2$, written $P_1 \sqsupseteq_s P_2$, if $A(P_1 \cup R) \supseteq A(P_2 \cup R)$ holds for any $R \in \mathcal{LP}$ [7]. $P_1$ *(strongly) ans-includes* $P_2$ *with respect to* $\mathcal{C}$, written $P_1 \sqsupseteq_{\mathcal{C}} P_2$, if $A(P_1 \cup R) \supseteq A(P_2 \cup R)$ holds for any $R \in \mathcal{C}$.

*Relativized (strong) equivalence* in Definition 3.1 (4) is the most general notion of equivalence in the sense that the other notions can be regarded as its special cases with particular contexts $\mathcal{C}$ [15].[5] That is, (1) weak, (2) strong and (3) uniform equivalence are the cases of $\mathcal{C} = \{\emptyset\}$, $\mathcal{C} = \mathcal{LP}$, and $\mathcal{C} = \mathcal{P}(Lit)$, respectively. The notion of *relativized (strong) inclusion* is likewise, and in particular we can define that $P_1$ *uniformly ans-includes* $P_2$ [31] iff $P_1$ strongly ans-includes $P_2$ with respect to $\mathcal{P}(Lit)$. We can further define the notions of *relativized uniform equivalence* [15,49] and *relativized uniform inclusion* [31] as special cases of (4) relativized equivalence and (6) relativized inclusion, respectively, such that $\mathcal{C} = \mathcal{P}(U)$ for some literal set $U \subseteq Lit$.

Assuming a poset $\langle \mathcal{P}(Lit), \supseteq \rangle$, the Smyth and Hoare orderings have been defined on $\mathcal{P}(\mathcal{P}(Lit))$ in Section 2. Then, a program $P$ is associated with its answer sets $A(P)$ to define generality orderings over the programs in $\mathcal{LP}$.

**Definition 3.2.** Assume the poset $\langle \mathcal{P}(Lit), \supseteq \rangle$, and let $P_1, P_2 \in \mathcal{LP}$. $P_1$ is *more (or equally)* $\sharp$*-general than* $P_2$, written $P_1 \models^\sharp P_2$, if $A(P_1) \models^\sharp A(P_2)$. $P_1$ is *more (or equally)* $\flat$*-general than* $P_2$, written $P_1 \models^\flat P_2$, if $A(P_1) \models^\flat A(P_2)$.

[5] This representation with a context $\mathcal{C}$ in Definition 3.1 (4) is similar to that in [48]. This is a slightly more general definition than that in [15], which can be represented by setting $\mathcal{C} = \mathcal{P}(R)$ for some rule set $R \in \mathcal{LP}$. Likewise, this is more general than the definition in [49,50], which can be expressed by setting $\mathcal{C}$ as the class of programs constructed with some literal set $U \subseteq Lit$ [49] or as the class of programs that use possibly different alphabets for the heads and bodies of rules [50].

In Definition 3.2, because $A(P_i)$ is closed for each program $P_i$, it holds that, $P_1 \models^{\sharp/\flat} P_2$ iff $A(P_1) \succeq^{\sharp/\flat} A(P_2)$. Note that Definition 3.2 is essentially the same as [17, Definition 2.2], but now we allow comparison of programs having non-minimal answer sets, whereas [17] only considers EDPs.

**Theorem 3.1.** *Let $P_1, P_2 \in \mathcal{LP}$. If $P_1 \sqsupseteq P_2$, then $P_2 \models^{\sharp} P_1$ and $P_1 \models^{\flat} P_2$.*

Theorem 3.1 follows from Proposition 2.4, and shows that the inclusion relation [7] can be interpreted in terms of generality relations.

An equivalence class under the $\models^{\sharp/\flat}$ relation can be characterized by the next proposition, which follows from Theorem 2.2.

**Theorem 3.2.** *Let $P_1, P_2 \in \mathcal{LP}$.*

(1) *$P_1 \models^{\sharp} P_2$ and $P_2 \models^{\sharp} P_1$ iff $min(A(P_1)) = min(A(P_2))$.*
(2) *$P_1 \models^{\flat} P_2$ and $P_2 \models^{\flat} P_1$ iff $max(A(P_1)) = max(A(P_2))$.*

In the case of EDPs, as shown in [17], each equivalence class precisely characterizes the set of weakly equivalent EDPs.

**Corollary 3.3.** *Let $P_1$ and $P_2$ be EDPs. Then, the following three are equivalent:* (1) *$P_1 \models^{\sharp} P_2 \models^{\sharp} P_1$;* (2) *$P_1 \models^{\flat} P_2 \models^{\flat} P_1$;* (3) *$P_1 \equiv P_2$.*

*Proof.* This has been proved in [17], but can be easily proved as follows. For any EDP $P$, $A(P)$ is irredundant. Then the result follows from Corollary 2.3. $\square$

**Example 3.1.** Consider the following programs:

$$\begin{aligned}
P_1 &= \{\, p \leftarrow not\, q \,\},\\
P_2 &= \{\, p \leftarrow not\, q, \quad q \leftarrow not\, p \,\},\\
P_3 &= \{\, p\,;\, q \leftarrow \,\},\\
P_4 &= \{\, p\,;\, q \leftarrow, \quad p \leftarrow q, \quad q \leftarrow p \,\},\\
P_5 &= \{\, p\,;\, not\, p \leftarrow, \quad q\,;\, not\, q \leftarrow, \quad \leftarrow not\, p,\, not\, q \,\}.
\end{aligned}$$

Then, $P_4 \models^{\sharp} P_1 \models^{\sharp} P_2$ and $P_4 \models^{\flat} P_2 \models^{\flat} P_1$ hold (see Example 2.1). $P_2 \equiv P_3$, and thus $P_2 \models^{\sharp} P_3 \models^{\sharp} P_2$ and $P_2 \models^{\flat} P_3 \models^{\flat} P_2$.

For $P_5$, $A(P_5) = \{\{p\}, \{q\}, \{p, q\}\}$ is not irredundant, where $\{p, q\}$ is non-minimal. Then, $P_1 \models^{\sharp} P_5 \models^{\sharp} P_2 \models^{\sharp} P_5$ and $P_5 \models^{\flat} P_4 \models^{\flat} P_5 \models^{\flat} P_2$ hold.

*Minimal upper bounds* (mubs) and *maximal lower bounds* (mlbs) have been important in the theory of generalization and in ILP [34,28], and can be applied to coordination [44], composition [42] and consensus [43] of multiple agents [17]. We can show that both a minimal upper bound and a maximal lower bound of any pair of programs exist with respect to $\sharp$- and $\flat$-generality orderings, but omit the details here.

As a result, we can induce the poset $\langle \mathbf{LP}^{\sharp/\flat}, \geqq^{\sharp/\flat} \rangle$, where $\mathbf{LP}^{\sharp/\flat}$ is the equivalence classes from $\langle \mathcal{LP}, \models^{\sharp/\flat} \rangle$ and $\geqq^{\sharp/\flat}$ is a partial order satisfying that $[P_1] \geqq^{\sharp/\flat} [P_2]$ if $P_1 \models^{\sharp/\flat} P_2$ for any $P_1, P_2 \in \mathcal{LP}$ and $[P_1], [P_2] \in \mathbf{LP}^{\sharp/\flat}$. Then, this poset constitutes a complete lattice such that (1) a program $P$ is in the top (resp. bottom) element of $\langle \mathbf{LP}^\sharp, \gtreqqless^\sharp \rangle$ iff $A(P) = \emptyset$ (resp. $A(P) = \{\emptyset\}$), and that (2) a program $P$ is in the top (resp. bottom) element of $\langle \mathbf{LP}^\flat, \geqq^\flat \rangle$ iff $A(P) = \{Lit\}$ (resp. $A(P) = \emptyset$).

When a logic program has multiple answer sets, the $\sharp$- and $\flat$-generality relations can be connected with skeptical and credulous entailment, respectively.

**Definition 3.3.** Let $P \in \mathcal{LP}$ be a program, and $\psi$ a set of literals, which is interpreted as the conjunction of literals in $\psi$.[6] Then, $\psi$ is a *skeptical consequence* of $P$ if $\psi \subseteq S$ for every $S \in A(P)$. $\psi$ is a *credulous consequence* of $P$ if $\psi \subseteq S$ for some $S \in A(P)$. The sets of skeptical and credulous consequences of $P$ are denoted as $Skp(P)$ and $Crd(P)$, respectively.

**Proposition 3.4.** *Let $P \in \mathcal{LP}$. If $P$ is consistent, $Skp(P) = \bigcap_{S \in A(P)} \mathcal{P}(S)$ and $Crd(P) = \bigcup_{S \in A(P)} \mathcal{P}(S)$. If $A(P) = \emptyset$, $Skp(P) = \mathcal{P}(Lit)$ and $Crd(P) = \emptyset$. If $A(P) = \{Lit\}$, $Skp(P) = Crd(P) = \mathcal{P}(Lit)$.*

**Theorem 3.5.** *Let $P_1, P_2 \in \mathcal{LP}$.*

(1) *If $P_1 \models^\sharp P_2$ then $Skp(P_1) \supseteq Skp(P_2)$.*
(2) *$P_1 \models^\flat P_2$ iff $Crd(P_1) \supseteq Crd(P_2)$.*

*Proof.* (1) Assume that $P_1 \models^\sharp P_2$. If $P_1$ is inconsistent, then $Skp(P_1) = \mathcal{P}(Lit)$ and thus $Skp(P_1) \supseteq Skp(P_2)$. Suppose that $P_1$ is consistent and $\psi \in Skp(P_2)$. Then, $\psi \subseteq T$ for every $T \in A(P_2)$. By $P_1 \models^\sharp P_2$, for any $S \in A(P_1)$, there is $T' \in A(P_2)$ such that $S \supseteq T'$. Since $\psi \subseteq T'$, $\psi \subseteq S$ too. That is, $\psi \in Skp(P_1)$, and thus $Skp(P_1) \supseteq Skp(P_2)$.

(2) The only-if part can be proved in a similar way as (1). To prove the if part, suppose that $P_1 \not\models^\flat P_2$. Then, $A(P_1) \neq \{Lit\}$ and $A(P_2) \neq \emptyset$. Then, there is a set $T \in A(P_2)$ such that $S \not\supseteq T$ for any $S \in A(P_1)$. That is, for each $S \in A(P_1)$, we can pick a literal $\varphi_S \in (T \setminus S)$. Then, $\psi = \{\varphi_S \mid S \in A(P_1)\} \subseteq T$, whereas $\psi \not\subseteq S$ for any $S \in A(P_1)$. Hence, $\psi \in Crd(P_2)$ and $\psi \notin Crd(P_1)$, and thus $Crd(P_1) \not\supseteq Crd(P_2)$. □

By Theorem 3.5, the more $\sharp$-general (resp. $\flat$-general) a program is, the more it entails skeptically (resp. credulously). Hence, the Smyth and Hoare orderings over programs reflect the amount of information by skeptical and credulous entailment, respectively. Moreover, $\flat$-generality precisely reflects informativeness of credulous entailment. On the other hand, the converse of Theorem 3.5 (1) does not hold for $\sharp$-generality.[7] For example, for $A(P_1) = \{\{p\}\}$ and $A(P_2) = \{\{q\}, \{r\}\}$, $Skp(P_1) = \{\emptyset, \{p\}\}$ and $Skp(P_2) = \{\emptyset\}$ hold, and thus $Skp(P_1) \supset Skp(P_2)$ but $P_1 \not\models^\sharp P_2$.

[6] This definition extends the previous one in [17, Definition 4.1], in which a consequence was defined for each single literal instead of a conjunction of literals.

[7] The converse of Theorem 3.5 (1) would also hold if we could include *disjunctive facts* along with conjunctions of literals in the consequences. For example, for $A(P_2) = \{\{q\}, \{r\}\}$, the extended set of skeptical consequences of $P_2$ contains $q; r$.

**Example 3.2.** In Example 3.1, $A(P_3) = \{\{p\}, \{q\}\}$ and $A(P_4) = \{\{p, q\}\}$, and thus $P_4 \models^\sharp P_3$ and $P_4 \models^\flat P_3$. By Definition 3.3, $Skp(P_3) = \emptyset$, $Crd(P_3) = \{\emptyset, \{p\}, \{q\}\}$, and $Skp(P_4) = Crd(P_4) = \mathcal{P}(\{p, q\})$. Correspondingly, $Skp(P_4) \supset Skp(P_3)$ and $Crd(P_4) \supset Crd(P_3)$, which verify Theorem 3.5. Note here that $Crd(P_3) \neq Crd(P_4)$, because $\{p, q\}$ is only included in the latter. Note also that $Crd(P_4) = Crd(P_5)$, and correspondingly $P_5 \models^\flat P_4 \models^\flat P_5$.

## 4 Strong, Uniform and Relativized Generality

A possible issue in generality by Definition 3.2 is that generality of programs is determined solely by their answer sets. However, since the set of answer sets of a program does not monotonically decrease as the program grows, we often want to know how the generality relation between programs changes when some rules are added to those programs. In this section, we examine such context-dependent versions of generality relations. Then, *robustness* of generality between programs can be discussed by the property that one program brings more information than another whatever any set of rules in a focused context are added to those programs.

Another concern on generality is the question of whether or not a generality relation under consideration agrees with the property of classical generality relation defined in ILP for first-order logic. In first-order logic, a theory $T_1$ is defined to be *more (or equally) general* than $T_2$ if every formula derived from $T_2$ is derived from $T_1$, i.e., $T_1 \models T_2$ [34,29]. Hence, classical generality is defined as the first-order entailment. Although the syntax of logic programming is different from first-order logic, it has been a convention that the class of PDPs and definite programs can also be regarded as a set of first-order formulas. We say that a generality relation $\succeq$ satisfies the *classical inductive generality* if it holds that $P_1 \succeq P_2$ iff $P_1 \models P_2$ for all PDPs $P_1$ and $P_2$. With this regard, the generality relation $\models^{\sharp/\flat}$ cannot make programs with the same answer sets different, hence does not satisfy the classical inductive generality. This property is often inconvenient from the viewpoint of program development.

**Example 4.1.** Suppose the three programs containing single rules, $P_1 = \{r_1\}$, $P_2 = \{r_2\}$ and $P_3 = \{r_3\}$, where

$$\begin{aligned} r_1 &: \ p \leftarrow q, \\ r_2 &: \ p \leftarrow q, r, \\ r_3 &: \ p \leftarrow q, \, not\, r. \end{aligned}$$

Because the answer sets of all programs are $\{\emptyset\}$, they are weakly equivalent, i.e., $P_1 \equiv P_2 \equiv P_3$. In ILP, however, $P_1$ is regarded as more general than $P_2$ because $P_1 \models P_2$ but $P_2 \not\models P_1$. In this case, under the classical models, $Mod(P_1) \models^\sharp Mod(P_2)$ and $Mod(P_2) \not\models^\sharp Mod(P_1)$ hold by $Mod(P_2) \supsetneq Mod(P_1)$ and Proposition 2.4. Similarly, $P_1$ is considered more general than $P_3$ in ILP. In program development, $r_1$ is often called a *generalization* of both $r_2$ and $r_3$, and conversely $r_2$ and $r_3$ are called *specializations* of $r_1$.

Example 4.1 indicates that, even in definite programs ($P_1$ and $P_2$), the generality relation $\models^{\sharp/\flat}$ does not satisfy the classical inductive generality as long as only the answer sets (i.e., minimal models) are concerned. This is because information contents brought by the collection of answer sets represents only the "solutions" of the current program, and "predictive (or deductive) power" of the program does not appear in the resultant answer sets but is implicit in the program itself. In this respect, too, we need to consider more context-sensitive notions of generality.

The notion of *strong generality* has already been introduced briefly in [17] as a counterpart of *strong equivalence* [21]. In this section, we firstly recall *strong generality* and newly consider *uniform* and *relativized* variants of generality relations in answer set programming. These three notions of generality are all context-sensitive variants of generality, but the strong and uniform versions can be regarded as special cases of relativized generality.

**Definition 4.1.** Let $P_1, P_2 \in \mathcal{LP}$ be programs, and $\mathcal{C} \subseteq \mathcal{LP}$.

(1) $P_1$ is *strongly more $\sharp$-general than* $P_2$, written $P_1 \trianglerighteq^{\sharp}_{s} P_2$, if $P_1 \cup R \models^{\sharp} P_2 \cup R$ for any program $R \in \mathcal{LP}$. $P_1$ is *strongly more $\flat$-general than* $P_2$, written $P_1 \trianglerighteq^{\flat}_{s} P_2$, if $P_1 \cup R \models^{\flat} P_2 \cup R$ for any program $R \in \mathcal{LP}$ [17].
(2) $P_1$ is *uniformly more $\sharp$-general than* $P_2$, written $P_1 \trianglerighteq^{\sharp}_{u} P_2$, if $P_1 \cup U \models^{\sharp} P_2 \cup U$ for any literal set $U \in \mathcal{P}(Lit)$. $P_1$ is *uniformly more $\flat$-general than* $P_2$, written $P_1 \trianglerighteq^{\flat}_{u} P_2$, if $P_1 \cup U \models^{\flat} P_2 \cup U$ for any literal set $U \in \mathcal{P}(Lit)$.
(3) $P_1$ is *(strongly) more $\sharp$-general than* $P_2$ *with respect to* $\mathcal{C}$, written $P_1 \trianglerighteq^{\sharp}_{\mathcal{C}} P_2$, if $P_1 \cup R \models^{\sharp} P_2 \cup R$ for any $R \in \mathcal{C}$. $P_1$ is *(strongly) more $\flat$-general than* $P_2$ *with respect to* $\mathcal{C}$, written $P_1 \trianglerighteq^{\flat}_{\mathcal{C}} P_2$, if $P_1 \cup R \models^{\flat} P_2 \cup R$ for any $R \in \mathcal{C}$.

The notions of *strong $\sharp/\flat$-generality* and *uniform $\sharp/\flat$-generality* are represented in Definition 4.1 (1) and (2), which can be expressed as special cases of Definition 4.1 (3) such that $\mathcal{C} = \mathcal{LP}$ and $\mathcal{C} = \mathcal{P}(Lit)$, respectively. Similarly, we can define the notion of *relativized uniform $\sharp/\flat$-generality* as a special case of Definition 4.1 (3) such that $\mathcal{C} = \mathcal{P}(U)$ for some $U \subseteq Lit$. In a special case, the context can be given as a set consisting of a single program as $\mathcal{C} = \{R\}$ for some $R \in \mathcal{LP}$. This corresponds to the concept of *relative generalization* in ILP [34], in which $P_1$ is called *more general than (or equal to)* $P_2$ *relative to background knowledge* $R$. Other relations among these definitions hold as follows.

**Proposition 4.1.** (1) $P_1 \trianglerighteq^{\sharp/\flat}_{s} P_2$ *implies* $P_1 \trianglerighteq^{\sharp/\flat}_{u} P_2$.
(2) $P_1 \trianglerighteq^{\sharp/\flat}_{u} P_2$ *implies* $P_1 \models^{\sharp/\flat} P_2$*, and hence* $P_1 \trianglerighteq^{\sharp/\flat}_{s} P_2$ *implies* $P_1 \models^{\sharp/\flat} P_2$.
(3) *For any* $\mathcal{C} \subseteq \mathcal{LP}$, $P_1 \trianglerighteq^{\sharp/\flat}_{s} P_2$ *implies* $P_1 \trianglerighteq^{\sharp/\flat}_{\mathcal{C}} P_2$.
(4) *For any* $\mathcal{C} \subseteq \mathcal{LP}$ *such that* $\emptyset \in \mathcal{C}$, $P_1 \trianglerighteq^{\sharp/\flat}_{\mathcal{C}} P_2$ *implies* $P_1 \models^{\sharp/\flat} P_2$.

Note that the condition $\emptyset \in \mathcal{C}$ is indispensable in Proposition 4.1 (4). For example, when $P_1 = \{p \leftarrow q\}$ and $P_2 = \{p \leftarrow\ \}$ are compared in the context $\mathcal{C} = \{\{q \leftarrow\ \}\}$, we have $P_1 \trianglerighteq^{\sharp}_{\mathcal{C}} P_2$ but $P_1 \not\models^{\sharp} P_2$.

It has been shown in [17] that programs belonging to the same equivalence class induced by strong generality become strongly equivalent in the case of EDPs. We here see that similar results also hold for uniform and relativized generality.

**Proposition 4.2.** *Let $P_1$ and $P_2$ be EDPs, and $\mathcal{C}$ a set of EDPs. Then, in each of the following, the statements* (i), (ii) *and* (iii) *are equivalent.*

(1) [17] (i) $P_1 \trianglerighteq^\sharp_s P_2 \trianglerighteq^\sharp_s P_1$; (ii) $P_1 \trianglerighteq^\flat_s P_2 \trianglerighteq^\flat_s P_1$; (iii) $P_1 \equiv_s P_2$.
(2) (i) $P_1 \trianglerighteq^\sharp_u P_2 \trianglerighteq^\sharp_u P_1$; (ii) $P_1 \trianglerighteq^\flat_u P_2 \trianglerighteq^\flat_u P_1$; (iii) $P_1 \equiv_u P_2$.
(3) (i) $P_1 \trianglerighteq^\sharp_{\mathcal{C}} P_2 \trianglerighteq^\sharp_{\mathcal{C}} P_1$; (ii) $P_1 \trianglerighteq^\flat_{\mathcal{C}} P_2 \trianglerighteq^\flat_{\mathcal{C}} P_1$; (iii) $P_1 \equiv_{\mathcal{C}} P_2$.

**Theorem 4.3.** *Let $P_1, P_2 \in \mathcal{LP}$. Suppose that Lit is finite.*
(1) *If $P_1 \trianglerighteq^\sharp_s P_2$ then $P_2 \sqsupseteq P_1$.* (2) *If $P_1 \trianglerighteq^\flat_s P_2$ then $P_1 \sqsupseteq P_2$.*

*Proof.* We prove (1), but (2) can be proved in a similar way.

Assume that $P_2 \not\sqsupseteq P_1$. Then, $\mathcal{A}(P_2) \not\supseteq \mathcal{A}(P_1)$, and hence $D = \mathcal{A}(P_1) \setminus \mathcal{A}(P_2) \neq \emptyset$. If there is an answer set $S \in D$ ($\subseteq \mathcal{A}(P_1)$) such that $S \subset T$ for any answer set $T \in \mathcal{A}(P_2)$, then $P_1 \not\models^\sharp P_2$ and thus $P_1 \not\trianglerighteq^\sharp_s P_2$. Otherwise, $S \not\subset T$ for any $S \in D$ and any $T \in \mathcal{A}(P_2)$, and moreover $S \not\subseteq T$ by the fact that $T \notin D$ and hence $S \neq T$. Then, $S \setminus T \neq \emptyset$. Let $C = \{ \leftarrow \mathit{not}\, L_1, \ldots, \mathit{not}\, L_n \mid L_i \in (S_i \setminus T),\ D = \{S_1, \ldots, S_n\},\ T \in \mathcal{A}(P_2) \}$. Each constraint in $C$ is always satisfied by some $S_i \in \mathcal{A}(P_1)$, but can never be satisfied by any $T \in \mathcal{A}(P_2)$. Hence, $\mathcal{A}(P_1 \cup C) \neq \emptyset$ and $\mathcal{A}(P_2 \cup C) = \emptyset$. Therefore, $P_1 \cup C \not\models^\sharp P_2 \cup C$, and hence $P_1 \not\trianglerighteq^\sharp_s P_2$. □

Theorem 4.3 states that *strong $\sharp/\flat$-generality implies inclusion.* This rather unexpected result can be explained by tracking the proof as follows. Simply assume that both $P_1$ and $P_2$ have single answer sets, $\mathcal{A}(P_1) = \{S\}$ and $\mathcal{A}(P_2) = \{T\}$. If $S \supset T$, the relation $P_1 \models^\sharp P_2$ holds. However, adding to $P_1$ and $P_2$ the constraint $C = \{ \leftarrow \mathit{not}\, L\}$ for some $L \in S \setminus T$ makes $P_2 \cup C$ inconsistent, and hence $P_2 \cup C \models^\sharp P_1 \cup C$ holds. Therefore, for $P_1$ to be strongly more $\sharp$-general than $P_2$, it is necessary that $S \setminus T = \emptyset$. From Theorem 4.3, we can derive the next stronger result:[8] *strong generality is equivalent to strong inclusion.*

**Theorem 4.4.** *Let $P_1, P_2 \in \mathcal{LP}$, and suppose that Lit is finite. Then, $P_1 \trianglerighteq^\sharp_s P_2$ iff $P_2 \trianglerighteq^\flat_s P_1$ iff $P_2 \sqsupseteq_s P_1$.*

*Proof.* We prove ($P_1 \trianglerighteq^\sharp_s P_2$ iff $P_2 \sqsupseteq_s P_1$), but ($P_1 \trianglerighteq^\flat_s P_2$ iff $P_1 \sqsupseteq_s P_2$) can be proved in a similar way. Suppose $P_1 \trianglerighteq^\sharp_s P_2$. By definition, $P_1 \cup R \models^\sharp P_2 \cup R$ for any $R \in \mathcal{LP}$. Then, $(P_1 \cup R_1) \cup R_2 \models^\sharp (P_2 \cup R_1) \cup R_2$ for any $R_1, R_2 \in \mathcal{LP}$. By definition, $P_1 \cup R_1 \trianglerighteq^\sharp_s P_2 \cup R_1$ for any $R_1 \in \mathcal{LP}$. By Theorem 4.3 (1), $\mathcal{A}(P_2 \cup R_1) \supseteq \mathcal{A}(P_1 \cup R_1)$ for any $R_1 \in \mathcal{LP}$. Hence, $P_2 \sqsupseteq_s P_1$. Conversely, suppose $P_2 \sqsupseteq_s P_1$. By definition, $\mathcal{A}(P_2 \cup R) \supseteq \mathcal{A}(P_1 \cup R)$ for any $R \in \mathcal{LP}$. By Theorem 3.1, $P_1 \cup R \models^\sharp P_2 \cup R$ for any $R \in \mathcal{LP}$. Hence, $P_1 \trianglerighteq^\sharp_s P_2$. □

**Example 4.2.** Consider $P_1$, $P_2$ and $P_3$ in Example 4.1. For $R_1 = \{ q \leftarrow \mathit{not}\, p \}$, $\mathcal{A}(P_1 \cup R_1) = \emptyset$, while $\mathcal{A}(P_2 \cup R_1) = \{\{q\}\}$. By contrast, for $R_2 = \{ q \leftarrow,\ \leftarrow \mathit{not}\, p \}$, $\mathcal{A}(P_1 \cup R_2) = \{\{p, q\}\}$, while $\mathcal{A}(P_2 \cup R_2) = \emptyset$. Hence, $P_1 \not\trianglerighteq^\sharp_s P_2$ and $P_2 \not\trianglerighteq^\sharp_s P_1$. Similarly, for $R_3 = \{ r \leftarrow,\ q \leftarrow \mathit{not}\, p \}$, $\mathcal{A}(P_1 \cup R_3) = \emptyset$ and $\mathcal{A}(P_3 \cup R_3) = \{\{q, r\}\}$. However, for $R_4 = \{ q \leftarrow,\ r \leftarrow,\ \leftarrow \mathit{not}\, p \}$, $\mathcal{A}(P_1 \cup R_4) = \{\{p, q, r\}\}$ and $\mathcal{A}(P_3 \cup R_4) = \emptyset$. Hence, $P_1 \not\trianglerighteq^\sharp_s P_3$ and $P_3 \not\trianglerighteq^\sharp_s P_1$.

[8] This property was pointed out to the authors by Jianmin Ji for EDPs.

Example 4.2 indicates that strong generality does not satisfy the classical inductive generality even for definite programs.

On the other hand, we see that both $P_1 \trianglerighteq^{\sharp}_{u} P_2$ and $P_1 \trianglerighteq^{\sharp}_{u} P_3$ hold. Uniform generality thus satisfies the classical inductive generality for Example 4.1. Indeed, the next result states that uniform generality collapses to classical entailment in the case of PDPs. Related results are stated in [26,4] such that uniform equivalence coincides with classical equivalence for PDPs.

**Proposition 4.5.** *Let $P_1$ and $P_2$ be PDPs. Then, $P_1 \trianglerighteq^{\sharp}_{u} P_2$ iff $P_1 \models P_2$, that is, $P_1$ classically entails $P_2$.*

*Proof.* It is easy to see that $P_1 \models P_2$ implies $P_1 \trianglerighteq^{\sharp}_{u} P_2$. Conversely, suppose that $P_1 \not\models P_2$. Then, there is a set $M$ of ground atoms such that $M \in Mod(P_1)$ and $M \notin Mod(P_2)$. Let $M' = M \cup \{\neg A \in Lit \mid A \notin M\}$. Since $M$ is not a model of $P_2$, $P_2 \cup M'$ is inconsistent. By Herbrand's theorem, there is a finite subset $U \subseteq M'$ of ground literals such that $P_2 \cup U$ is inconsistent. Since $M$ is a model of $P_1$, $P_1 \cup U$ is consistent. Then, there is a consistent answer set of $P_1 \cup U$, but $Lit$ is the unique inconsistent answer set of $P_2 \cup U$. Obviously, $P_1 \cup U \not\models^{\sharp} P_2 \cup U$. Hence, $P_1 \not\trianglerighteq^{\sharp}_{u} P_2$. □

**Proposition 4.6.** *Let $P_1$ and $P_2$ be PDPs without integrity constraints. Then, $P_1 \trianglerighteq^{\flat}_{u} P_2$ iff $P_1 \models P_2$.*

*Proof.* The property (if $P_1 \trianglerighteq^{\flat}_{u} P_2$ then $P_1 \models P_2$) can be proved in the same way as Proposition 4.5. To show that (if $P_1 \models P_2$ then $P_1 \trianglerighteq^{\flat}_{u} P_2$), we need the condition that $P_1$ does not contain integrity constraints. When $P_1$ has no constraint, $P_1 \cup U$ always has an answer set for any literal set $U \in \mathcal{P}(Lit)$. Then, $P_1 \cup U \models^{\flat} P_2 \cup U$. □

**Example 4.3.** Consider $P_1$, $P_2$ and $P_3$ in Example 4.1. We see that both $P_1 \trianglerighteq^{\sharp}_{\mathcal{C}} P_2$ and $P_1 \trianglerighteq^{\sharp}_{\mathcal{C}} P_3$ hold for $\mathcal{C} = \mathcal{P}(R)$, where $R = \{p,\ q,\ r,\ p;q,\ p;r,\ q;r,\ p;q;r\}$.

Example 4.3 strengthens the classical inductive property for uniform generality to that for relativized generality with the context $\mathcal{C}$ including disjunctive facts. Moreover, relativized generality also holds when the context $\mathcal{C}$ is the class of PDPs.

**Proposition 4.7.** *Let $\mathcal{C}$ be the class of PDPs. Then, for any two PDPs $P_1, P_2 \in \mathcal{C}$, $P_1 \trianglerighteq^{\sharp}_{\mathcal{C}} P_2$ iff $P_1 \models P_2$.*

*Proof.* $P_1 \models P_2$ implies $P_1 \cup R \models^{\sharp} P_2 \cup R$ for any $R \in \mathcal{C}$. Hence, $P_1 \models P_2$ implies $P_1 \trianglerighteq^{\sharp}_{\mathcal{C}} P_2$. The converse property ($P_1 \not\models P_2$ implies $P_1 \not\trianglerighteq^{\sharp}_{\mathcal{C}} P_2$) can be proved in the same way as Proposition 4.5. □

**Proposition 4.8.** *Let $\mathcal{C}$ be the class of PDPs having no integrity constraint. Then, for any two PDPs $P_1, P_2 \in \mathcal{C}$, $P_1 \trianglerighteq^{\flat}_{\mathcal{C}} P_2$ iff $P_1 \models P_2$.*

*Proof.* Let $R$ be any PDP in $\mathcal{C}$. Since $P_1$, $P_2$ and $R$ are PDPs containing no constraints, $P_1 \cup R$ has always an answer set, and thus $P_1 \models P_2$ implies $P_1 \cup R \models^{\flat} P_2 \cup R$. Hence, $P_1 \models P_2$ implies $P_2 \trianglerighteq^{\flat}_{\mathcal{C}} P_1$. The converse direction can be proved in the same way as Proposition 4.6. □

In both Propositions 4.6 and 4.8, the condition that $P_1$ does not contain an integrity constraint is necessary to establish the relations that $P_1 \models P_2$ implies $P_1 \trianglerighteq^{\flat}_{u} P_2$ and $P_1 \trianglerighteq^{\flat}_{\mathcal{C}} P_2$. For example, when $P_1 = \{\leftarrow a\}$ and $P_2 = \emptyset$, $P_1 \models P_2$ holds. However, for $U = \{a \leftarrow\}$, $\mathcal{A}(P_1 \cup U) = \emptyset$ and $\mathcal{A}(P_2 \cup U) = \{\{a\}\}$, and hence $P_1 \not\trianglerighteq^{\flat}_{u} P_2$ and $P_1 \not\trianglerighteq^{\flat}_{\mathcal{C}} P_2$.

## 5 Abductive Generality

In this section, we show that relativized generality in the previous section can be well applied to generality relations for *abductive logic programming* (ALP) [20]. Then, such generality relations could be applied to give logical foundations of coordination between multiple *abductive agents* by extending results of [44,42,43] to combination of abductive programs.

A semantics for ALP is given by extending answer sets of the background program with addition of abducibles. Such an extended answer set is called a *generalized stable model* [20] or a *belief set* [16].

**Definition 5.1.** An *abductive (logic) program* is a pair $\langle P, \Gamma \rangle$, where $P \in \mathcal{LP}$ is a logic program and $\Gamma \subseteq Lit$ is a set of literals called *abducibles*.[9] Put $\mathbf{\Gamma} = \mathcal{P}(\Gamma)$.

Let $A = \langle P, \Gamma \rangle$ be an abductive program, and $E \in \mathbf{\Gamma}$, i.e., $E \subseteq \Gamma$. A *belief set* of $A$ (*with respect to* $E$) is a consistent answer set of the logic program $P \cup E$. The set of all belief sets of $A$ is denoted as $\mathcal{B}(A)$. A set $S \in \mathcal{B}(A)$ is often denoted as $S_E$ when $S$ is a belief set with respect to $E$.

An *observation* $G$ is defined as a set of ground literals, which is interpreted as the conjunction of its literals. A set $E \in \mathbf{\Gamma}$ is a *brave* (resp. *cautious*) *explanation* of $G$ *in* $A$ if $G$ is true in some (resp. every) belief set of $A$ with respect to $E$.[10] When $G$ has a brave (resp. cautious) explanation in $A$, $G$ is *bravely* (resp. *cautiously*) *explainable* in $A$.

Note that $\mathcal{B}(A)$ is a composite set not containing $Lit$. Inoue and Sakama [19] introduce two measures for comparing (brave) explanation power of abductive programs. One is aimed at comparing *explainability* for observations in different theories, while the other is aimed at comparing *explanation contents* for observations.

**Definition 5.2.** $A_1$ is *more (or equally) bravely* (resp. *cautiously*) *explainable than* $A_2$, written $A_1 \geqslant^b A_2$, (resp. $A_1 \geqslant^c A_2$), if every observation bravely (resp. cautiously) explainable in $A_2$ is also bravely (resp. cautiously) explainable in $A_1$.

[9] The abducibles can be defined as $\Gamma \in \mathcal{LP}$ by allowing rules, but such an extended form of abductive program $\langle P, \Gamma \rangle$ can be reduced to an abductive program in Definition 5.1 by moving each rule in $\Gamma$ to $P$ with a new abuducible literal added to the body so that adding the new abducible enables the rule in abduction [12].

[10] Brave and cautious explanations are also called *credulous* and *skeptical* explanations, respectively. The properties of cautious explanations have not been studied in [19] and are newly investigated in this paper.

On the other hand, $A_1$ is *more (or equally) bravely* (resp. *cautiously*) *explanatory than* $A_2$, written $A_1 \Supset^b A_2$ (resp. $A_1 \Supset^c A_2$), if, for any observation $G$, every brave (resp. cautious) explanation of $G$ in $A_2$ is also a brave (resp. cautious) explanation of $G$ in $A_1$.

Note that $A_1 \Supset^b A_2$ implies $A_1 \geqslant^b A_2$ and that $A_1 \Supset^c A_2$ implies $A_1 \geqslant^c A_2$. To simplify the problem, we hereafter assume that $A_1 = \langle P_1, \Gamma\rangle$ and $A_2 = \langle P_2, \Gamma\rangle$ are abductive programs with the same abducibles $\Gamma$.

**Example 5.1.** Let $A_1 = \langle P_1, \Gamma\rangle$ and $A_2 = \langle P_2, \Gamma\rangle$ be abductive programs, where $P_1 = \{\, p \leftarrow a,\ a \leftarrow b \,\}$, $P_2 = \{\, p \leftarrow a,\ p \leftarrow b \,\}$, and $\Gamma = \{a, b\}$. Then, $A_1 \geqslant^b A_2$ and $A_2 \geqslant^b A_1$, while $A_1 \Supset^b A_2$ but $A_2 \not\Supset^b A_1$. In fact, $\{b\}$ is an explanation of $a$ in $A_1$, but is not in $A_2$. So with the relations $\geqslant^c$ and $\Supset^c$.

The relationships between the notions of abductive generality and (relativized) generality relations for logic programs can be established as follows. Note that these relationships are not obvious from the definitions.

**Theorem 5.1.** *$A_1 \geqslant^b A_2$ iff $B(A_1) \models^\flat B(A_2)$.*

*Proof.* This result follows from the property proved in [19] that, $A_1 \geqslant^b A_2$ holds iff for any belief set $S_2$ of $A_2$, there is a belief set $S_1$ of $A_1$ such that $S_1 \supseteq S_1$. □

An abductive program $A = \langle P, \Gamma\rangle$ can be translated to a logic program as follows [14]. Let $Cons = \{\, \leftarrow L, \neg L \mid L \in Lit \,\}$ [21], and put $P^+ = P \cup Cons$. Next, let $Abd(\Gamma) = \{\, L; not\, L \leftarrow \ \mid L \in \Gamma \,\}$, and put $P^+_\Gamma = P^+ \cup Abd(\Gamma)$. Then, $B(A) = A(P^+_\Gamma)$ holds [14]. With this result we get the next.

**Corollary 5.2.** *Let $A_1 = \langle P_1, \Gamma\rangle$ and $A_2 = \langle P_2, \Gamma\rangle$ be abductive programs. Then, $A_1 \geqslant^b A_2$ iff ${P_1}^+_\Gamma \models^\flat {P_2}^+_\Gamma$.*

**Theorem 5.3.** *Let $A_1 = \langle P_1, \Gamma\rangle$ and $A_2 = \langle P_2, \Gamma\rangle$ be abductive programs. Then, $A_1 \Supset^b A_2$ iff ${P_1}^+ \trianglerighteq^\flat_{\mathbf{\Gamma}} {P_2}^+$.*

*Proof.* $A_1 \Supset^b A_2$ iff for any $E \in \mathbf{\Gamma}$ and any $S_E \in B(A_2)$, there is $T_E \in B(A_1)$ such that $T_E \supseteq S_E$ [19] iff for any $E \in \mathbf{\Gamma}$, for any $S \in A(P^+_2 \cup E)$ there is $T \in A(P^+_1 \cup E)$ such that $T \supseteq S$. Hence, the result follows. □

Corollary 5.2 and Theorem 5.3 give us a better intuition on how explainable and explanatory generality are different in brave abduction: the former is characterized by $\flat$-generality, while the latter by relativized $\flat$-generality.

By contrast, explanatory (and explainable) generality in cautious abduction can be characterized by relativized strong $\sharp$-generality, but only as a sufficient condition.

**Theorem 5.4.** *Let $A_1 = \langle P_1, \Gamma\rangle$ and $A_2 = \langle P_2, \Gamma\rangle$ be abductive programs. If ${P_1}^+ \trianglerighteq^\sharp_{\mathbf{\Gamma}} {P_2}^+$ then $A_1 \Supset^c A_2$ (and then $A_1 \geqslant^c A_2$).*

*Proof.* If ${P_1}^+ \trianglerighteq^\sharp_{\mathbf{\Gamma}} {P_2}^+$ then ${P_1}^+ \cup E \models^\sharp {P_2}^+ \cup E$ for any $E \in \mathbf{\Gamma}$. Then, $Skp({P_1}^+ \cup E) \supseteq Skp({P_2}^+ \cup E)$ for any $E \in \mathbf{\Gamma}$ by Theorem 3.5 (1). Hence, $A_1 \Supset^c A_2$. □

Note that the converse of Theorem 5.4 does not hold as the converse of Theorem 3.5 (1) used in the proof does not hold.

Inoue and Sakama [16] study two equivalence relations in abduction called *explainable/explanatory equivalence*. Pearce *et al.* [33] characterize a part of these problems in the context of equilibrium logic. Since these abductive equivalence notions are defined in terms of brave explanations, they are related with brave abductive generality notions in [19]. Formally, two abductive programs $A_1$ and $A_2$ are *explainably equivalent* if, for any observation $O$,[11] $O$ is explainable in $A_1$ iff $O$ is explainable in $A_2$. On the other hand, $A_1$ and $A_2$ are *explanatorily equivalent* if, for any observation $O$, $E$ is an explanation of $O$ in $A_1$ iff $E$ is an explanation of $O$ in $A_2$.

Characterization of abductive equivalence has already been investigated in depth in [16,19]. Using results in Sections 2 and 4, these relations can easily be derived as corollaries of Theorems 5.1 and 5.3 as follows.

**Corollary 5.5.** *Let* $A_1 = \langle P_1, \Gamma\rangle$ *and* $A_2 = \langle P_2, \Gamma\rangle$ *be abductive programs.*

(1) [19] $A_1$ *and* $A_2$ *are explainably equivalent iff* $max(B(A_1)) = max(B(A_2))$.
(2) [19] $A_1$ *and* $A_2$ *are explanatorily equivalent iff* ${P_1}^+ \trianglerighteq^\flat_{\mathbf{\Gamma}} {P_2}^+ \trianglerighteq^\flat_{\mathbf{\Gamma}} {P_1}^+$ *iff* $max(A(P_1^+ \cup E)) = max(A(P_2^+ \cup E))$ *for any* $E \in \mathbf{\Gamma}$.
(3) [16] *Suppose that both* $P_1$ *and* $P_2$ *are EDPs. Then,* $A_1$ *and* $A_2$ *are explanatorily equivalent iff* $P_1^+ \equiv_{\mathbf{\Gamma}} P_2^+$.

## 6 Discussion

The inclusion relation in answer set programming was firstly considered in Eiter *et al.* [7]. It is argued in [7] that, if every answer set of a program $P$ is also an answer set of a program $Q$, i.e., $Q \sqsupseteq P$, then $Q$ can be viewed as a *skeptically sound approximation* of $P$, meaning that $Skp(P) \supseteq Skp(Q)$. However, a program which has no inclusion relation with $P$ can be a skeptically sound approximation of $P$ as well. For example, suppose two programs $P = \{p \leftarrow\ ,\ q \leftarrow p\}$ and $Q = \{p \leftarrow\ \}$. Then, $A(P) = \{\{p, q\}\}$ and $A(Q) = \{\{p\}\}$, and hence $Q$ is sound with respect to skeptical reasoning from $P$ although the inclusion relation $\sqsupseteq$ does not hold between $P$ and $Q$. Using generality, we can conclude that $P \models^\sharp Q$.

Section 2 gives a fairly general framework for comparing semantic structures called *composite sets*. This notion can be further extended by allowing any theory as an element of a composite set. For example, if a composite set is defined as a set of closed first-order theories, it can be applied to represent the semantic structure of *default logic* [36]. In Example 4.1, we have seen weakly equivalent programs, $P_1$, $P_2$ and $P_3$. As long as each rule in a program is interpreted as a first-order formula, the models of $P_1$ include those of $P_2$, and hence satisfy the classical inductive generality. With this regard, generality relations over default theories, which allow first-order formulas along with default rules, are proposed in [18]. Then, $P_1$ is properly ordered to be more general than $P_2$ if these programs are regarded as default theories. Unfortunately, the approach of [18] cannot

[11] This definition of explainable equivalence for ALP is not exactly the same as that in [16, Definition 4.3]. In [16] an observation is a single ground literal, while we allow a conjunction of ground literals as an observation.

differentiate $P_1$ and $P_3$ because $r_1$ can be interpreted as either a propositional formula or a default but $r_3$ can only be regarded as a default. We have seen that they are properly ordered using uniform generality and relativized generality.

We have introduced the notion of composite sets and their orderings to compare answer set programs in this paper. Since any pre-order $\succcurlyeq$ can be considered in an underlying pre-ordered set, an application of ranking or preferring composite sets to *decision theory* would be interesting in AI and economics. In this context, the $\flat$-ordering has been used to extend a preference relation over a possibly infinite set $X$ to its powerset $\mathcal{P}(X)$ [1]. The comparison framework proposed in this paper could be the basis for ordering answer set programs in such a decision theoretic context. For example, we can compare two *prioritized logic programs* [41] by setting the pre-order $\succcurlyeq$ as the preference relations between answer sets.

There are some other approaches to order logic programs in the literature. In [38], answer set programs are ordered according to multi-valued interpretations for programs. A promising approach is to order programs according to the implication relation with *HT-models* based on the logic of here-and-there [2,52]. A related approach is to use *SE-models* instead of HT-models for defining entailment and consequence relations [51,3]. As is stated in [52], any generality relation in [17] does not coincide with any of [52]. Considering relativized versions of generality in this paper, however, it is worth investigating precise relations between these approaches and ours. In fact, as far as consistent programs are compared, the ordering based on HT-models results in a similar ordering to our strong generality.

## 7 Conclusion

In this paper, we have extended the $\sharp$- and $\flat$-generality orderings of [17] in various ways. The original contributions in this paper are summarized as follows.

- Applicability of comparison under generality is extended from EDPs to any class of logic programs including nested programs and logic programs with aggregates. In comparison under $\sharp$- (resp. $\flat$-) generality, it is shown that the minimal (resp. maximal) answer sets determine the generality relation.
- Strong, uniform and relativized generality for logic programs are proposed as context-sensitive versions of generality, and several relations between them are investigated. For example, we have shown that:
  - Every generality relation is an instance of relativized generality.
  - Strong generality coincides with strong inclusion.
  - Uniform generality for PDPs collapses to classical entailment.
  - Relativized equivalent EDPs can be characterized as an equivalence class induced by relativized $\sharp$ or $\flat$-generality.
- The notions of explainable and explanatory generality in abductive logic programming are characterized using $\flat$-generality and relativized $\flat$-generality, which provide us a better intuition on how they are different.

As for the computational complexity of the proposed framework, we can apply several previously known results to establish complexity results for our framework. Since our comparison framework is general enough to include many known equivalence and inclusion relations, hardness results can be easily obtained from complexity results for such subproblems, e.g., [4,7,31,19]. On the other hand, those necessary and sufficient conditions such as Theorem 4.4, Propositions 4.5 and 4.7, Corollary 5.2 and Theorem 5.3 can be used to establish completeness results. Roughly speaking, the complexity classes range from coNP to $\Pi_3^P$ in the polynomial hierarchy, and more precise characterizations will be reported elsewhere.

Other important future work includes incorporation of the notion of *projection* used in the framework of [7,31,35] into our generality framework and more investigation of generality in the non-ground case like studies for equivalence in [25,5,22,30].

## References

1. Ballester, M.A., De Miguel, J.R.: Extending an order to the power set: the leximax criterion. Social Choice and Welfare 21, 63–71 (2003)
2. Cabalar, P., Pearce, D.J., Valverde, A.: Minimal Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 104–118. Springer, Heidelberg (2007)
3. Delgrande, J., Schaub, T., Tompits, H., Woltran, S.: Merging Logic Programs under Answer Set Semantics. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 160–174. Springer, Heidelberg (2009)
4. Eiter, T., Fink, M.: Uniform Equivalence of Logic Programs under the Stable Model Semantics. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 224–238. Springer, Heidelberg (2003)
5. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Strong and uniform equivalence in answer-set programming: characterizations and complexity results for the non-ground case. In: Proceedings AAAI 2005, pp. 695–700 (2005)
6. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. In: Proceedings KR 2004, pp. 141–151. AAAI Press, Menlo Park (2004)
7. Eiter, T., Tompits, H., Woltran, S.: On solution correspondences in answer-set programming. In: Proceedings IJCAI 2005, pp. 97–102 (2005)
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. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings ICLP 1988, pp. 1070–1080. MIT Press, Cambridge (1988)
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
11. Gunter, C.A., Scott, D.S.: Semantic domains. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 633–674. North-Holland, Amsterdam (1990)
12. Inoue, K.: Hypothetical reasoning in logic programs. J. Logic Programming 18, 191–227 (1994)

13. Inoue, K., Kudoh, Y.: Learning extended logic programs. In: Proceedings IJCAI 1997, pp. 176–181. Morgan Kaufmann, San Francisco (1997)
14. Inoue, K., Sakama, C.: Negation as failure in the head. J. Logic Programming 35, 39–78 (1998)
15. 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)
16. Inoue, K., Sakama, C.: Equivalence in abductive logic. In: Proceedings IJCAI 2005, pp. 472–477 (2005)
17. Inoue, K., Sakama, C.: Generality Relations in Answer Set Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 211–225. Springer, Heidelberg (2006)
18. Inoue, K., Sakama, C.: Generality and equivalence relations in default logic. In: Proceedings AAAI 2007, pp. 434–439 (2007)
19. Inoue, K., Sakama, C.: Comparing abductive theories. In: Proceedings 18th European Conference on Artificial Intelligence, pp. 35–39 (2008)
20. Kakas, A., Kowalski, R., Toni, F.: The role of abduction in logic programming. In: Gabbay, D., Hogger, C., Robinson, J. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 235–324. Oxford University Press, Oxford (1998)
21. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2, 526–541 (2001)
22. 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)
23. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence 25, 369–389 (1999)
24. Lifschitz, V., Woo, T.Y.C.: Answer sets in general nonmonotonic reasoning (preliminary report). In: Proceedings KR 1992, pp. 603–614. Morgan Kaufmann, San Francisco (1992)
25. Lin, F.: Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: Proceedings KR 2002, pp. 170–176 (2002)
26. Maher, M.J.: Equivalence of logic programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 627–658 (1988)
27. Marek, V.W., Niemelä, I., Truszczyński, M.: Logic programs with monotone abstract constraint atoms. Theory and Practice of Logic Programming 8, 167–199 (2007)
28. Muggleton, S., de Raedt, L.: Inductive logic programming: theory and methods. J. Logic Programming 19/20, 629–679 (1994)
29. Nienhuys-Cheng, S.-H., de Wolf, R.: Foundations of Inductive Logic Programming. LNCS (LNAI), vol. 1228. Springer, Heidelberg (1997)
30. Oetsch, J., Tompits, H.: Program correspondence under the answer-set semantics: the non-ground case. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 591–605. Springer, Heidelberg (2008)
31. Oetsch, J., Tompits, H., Woltran, S.: Facts do not cease to exist because they are ignored: relativised uniform equivalence with answer-set projection. In: Proceedings AAAI 2007, pp. 458–464 (2007)
32. Otero, R.: Induction of stable models. In: Rouveirol, C., Sebag, M. (eds.) ILP 2001. LNCS (LNAI), vol. 2157, pp. 193–205. Springer, Heidelberg (2001)

33. Pearce, D., Tompits, H., Woltran, S.: Relativised equivalence in equilibrium logic and its applications to prediction and explanation: preliminary report. In: Proceedings LPNMR 2007 Workshop on Correspondence and Equivalence for Nonmonotonic Theories, pp. 37–48 (2007)
34. Plotkin, G.D.: A note on inductive generalization. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 5, pp. 153–163. Edinburgh University Press, Edinburgh (1970)
35. Pührer, J., Tompits, H.: Casting Away Disjunction and Negation under a Generalisation of Strong Equivalence with Projection. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 264–276. Springer, Heidelberg (2009)
36. Reiter, R.: A logic for default reasoning. Artificial Intelligence 13, 81–132 (1980)
37. Sagiv, Y.: Optimizing datalog programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 659–668 (1988)
38. Sakama, C.: Ordering default theories and nonmonotonic logic programs. Theoretical Computer Science 338, 127–152 (2005)
39. Sakama, C.: Induction from answer sets in nonmonotonic logic programs. ACM Transactions on Computational Logic 6, 203–221 (2005)
40. Sakama, C., Inoue, K.: An alternative approach to the semantics of disjunctive logic programs and deductive databases. J. Automated Reasoning 13, 145–172 (1994)
41. Sakama, C., Inoue, K.: Prioritized logic programming and its application to commonsense reasoning. Artificial Intelligence 123, 185–222 (2000)
42. Sakama, C., Inoue, K.: Combining Answer Sets of Nonmonotonic Logic Programs. In: Toni, F., Torroni, P. (eds.) CLIMA 2005. LNCS (LNAI), vol. 3900, pp. 320–339. Springer, Heidelberg (2006)
43. Sakama, C., Inoue, K.: Constructing Consensus Logic Programs. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 26–42. Springer, Heidelberg (2007)
44. Sakama, C., Inoue, K.: Coordination in answer set programming. ACM Transactions on Computational Logic 9(2), A9 (2008)
45. Sakama, C., Inoue, K.: Brave induction: a logical framework for learning from incomplete information. Machine Learning 76(1), 3–35 (2009)
46. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138, 181–234 (2002)
47. Son, T.C., Pontelli, E., Tu, P.H.: Answer sets for logic programs with arbitrary abstract constraint atoms. J. Artificial Intelligence Research 29, 353–389 (2007)
48. Truszczyński, M., Woltran, S.: Hyperequivalence of logic programs with respect to supported models. In: Proceedings AAAI 2008, pp. 560–565 (2008)
49. Woltran, S.: Characterizations for Relativized Notions of Equivalence in Answer Set Programming. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 161–173. Springer, Heidelberg (2004)
50. Woltran, S.: A common view on strong, uniform, and other notions of equivalence in answer set programming. Theory and Practice of Logic Programming 8, 217–234 (2008)
51. Wong, K.-S.: Sound and complete inference rules for SE-consequences. Journal of Artificial Intelligence Research 31, 205–216 (2008)
52. 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)

# Compact Translations of Non-disjunctive Answer Set Programs to Propositional Clauses

Tomi Janhunen and Ilkka Niemelä

Aalto University
Department of Information and Computer Science
P.O. Box 15400, FI-00076 Aalto, Finland
{Tomi.Janhunen,Ilkka.Niemela}@aalto.fi

**Abstract.** Propositional satisfiability (SAT) solvers provide a promising computational platform for logic programs under the stable model semantics. Computing stable models of a logic program using a SAT solver presumes translating the program into a set of clauses in the DIMACS format which is accepted by most SAT solvers as input. In this paper, we present succinct translations from programs with choice rules, cardinality rules, and weight rules—also known as SMODELS programs—to sets of clauses. These translations enable us to harness SAT solvers as black boxes to the task of computing stable models for logic programs generated by any SMODELS compatible grounder such as LPARSE or GRINGO. In the experimental part of this paper, we evaluate the potential of SAT solver technology in finding stable models using NP-complete benchmark problems employed in the Second Answer Set Programming Competition.

## 1 Introduction

Logic programs under the stable model semantics [9] provide an interesting basis for solving search problems using the *answer set programming* (ASP) paradigm [20,16,22]. In ASP a problem is solved by devising a logic program such that the stable models of the program provide the answers to the problem, i.e., solving the problem is reduced to a stable model computation task. For the success of ASP, efficient solver technology is a key issue. Typically, ASP solvers use a two level architecture where a *grounder* takes an input program with variables and compiles it to a ground program which preserves the stable models of the original program, and then a *model finder* is used for computing stable models of the resulting ground program. Current ASP systems such as SMODELS [24], DLV [14], and CLASP [8] provide efficient grounder and model finder components. In some systems the components are separate and can be combined in various ways. This is the case, for example, for the grounder LPARSE and the model finder SMODELS and similarly for the grounder GRINGO and the model finder CLASP. In addition to these native model finders, a number of other systems have been developed to exploit SAT solver technology in the computation of stable models. Naturally, the idea is to benefit from the rapid development of SAT solvers.

In this paper we investigate this approach further with the goal of developing a compact translation of ground ASP programs to propositional clauses so that a *SAT solver can be used unmodified* as the model finder component of an ASP system.

For a number of subclasses of normal programs the translation to clauses can be done using Clark's completion $\mathrm{Comp}(P)$ [3]. The most well-known class is that without positive recursion, so-called *tight* [6] logic programs. For a tight program $P$, the respective sets of stable models $\mathrm{SM}(P)$, *supported models* $\mathrm{SuppM}(P)$, and *classical models* of $\mathrm{Comp}(P)$ coincide. To deal with non-tight programs in a similar fashion further constraints become necessary in addition to $\mathrm{Comp}(P)$. One possibility is to introduce *loop formulas* [19] on the fly to exclude models of the completion (i.e., supported models) which are not stable models. As witnessed by the ASSAT system [19] this strategy can be highly efficient. A drawback of the approach is that, at worst, the solver might introduce exponentially many loop formulas in program length (denoted $\|P\|$). Although the number of loop formulas often stays reasonably low, e.g., when computing just one stable model, they can become a bottleneck when finding all models—or proving their non-existence—is of interest. There is a follow-up system CMODELS [15] which implements an improved *ASP-SAT algorithm* detailed in [10]. As demonstrated therein, there are logic programs for which the number of supported models is exponentially greater than that of stable models. Therefore, extra bookkeeping is required in order to avoid repetitive disqualification of supported but non-stable models.

The translations from normal programs into propositional theories [2,18,11] provide a way to circumvent the worst-case behavior of the strategy based on loop formulas. However, the translation in [2] does not yield a one-to-one correspondence of models, and the one in [18] is quadratic which makes it infeasible for larger program instances. The most compact proposal [11] exploits a characterization of stable models in terms of *level numberings*—leading to a translation of the order of $\|P\| \times \log_2 |\mathrm{At}(P)|$ where $\mathrm{At}(P)$ is the set of atoms appearing in $P$. Additionally, the idea of the translation is to remove all positive body literals from a normal program which makes Clark's completion sound for a program transformed in this way. There is also an implementation of this transformation, a translator called LP2SAT, evaluated in [11]. Quite recently, it has also been established that stable models of normal programs can be captured in an extension of propositional logic, called difference logic, using level rankings [23] which are quite close to the level numbering technique in [11]. Difference logic can be seen as an instance of the *satisfiability modulo theories* (SMT) framework and practically all major SMT solvers support it. The translation is very compact, linear size, and when using current SMT solvers it leads to very competitive performance [13]. Our objective is to take a similar approach when translating programs into propositional logic.

The goal of this paper is to develop effective compact translations from ground logic programs to propositional clauses based on results in [11,23,13] in such a way that a SAT solver can be used *unmodified* as the model finder component in an ASP system. The developed translation has a number of novel features. It handles not just normal programs but also programs with cardinality and weight constraints (so-called SMODELS programs) in a compact way. The main part of the translation is done by logic program level transformations from an SMODELS program $P$ to a normal program $P'$ in such a way that the stable models of $P$ correspond to the supported models of $P'$. In this way ASP techniques can be applied for succinctly representing the additional constraints and to optimize the encoding. Moreover, a standard Clark's completion based technique can be used directly to $P'$ to generate a set of clauses capturing the supported models of $P'$ and, hence, the stable models of $P$. The rest of the paper is structured as follows. In the next section we explain briefly the necessary basics of logic programs. In Section 3, the translation from logic programs to clauses is developed. Section 4 provides an experimental evaluation of the translations compared to state-of-the-art techniques for computing stable models. Finally, we present our conclusions in Section 5.

## 2 Preliminaries

A propositional *normal program* $P$ is a set of *normal rules* of the form

$$a \leftarrow b_1,\, \ldots,\, b_n,\, {\sim}c_1,\, \ldots,\, {\sim}c_m. \tag{1}$$

where $a$, $b_1, \ldots, b_n$, and $c_1, \ldots, c_m$ are propositional atoms, and $\sim$ denotes *default negation*. In the sequel, propositional atoms are called just *atoms* for short. The intuition behind a normal rule $r$ of the form (1) is that the *head* atom $\mathrm{H}(r) = a$ is true whenever the *body* $\mathrm{B}(r) = \{b_1,\, \ldots,\, b_n,\, {\sim}c_1,\, \ldots,\, {\sim}c_m\}$ of the rule is satisfied. This is the case exactly when the *positive body atoms* in $\mathrm{B}^+(r) = \{b_1, \ldots, b_n\}$ are necessarily true by the other rules in the program whereas none of the *negative body atoms* in $\mathrm{B}^-(r) = \{c_1, \ldots, c_m\}$ is satisfied in this sense. We define the positive part $r^+$ of a rule $r$ as $\mathrm{H}(r) \leftarrow \mathrm{B}^+(r)$ hence omitting negative body conditions. A normal program $P$ is called *positive*, if $r = r^+$ holds for every rule $r \in P$. Occasionally, we also use *integrity constraints* which are rules (1) without the head $a$ and denoting the falsity of the body. Such constraints can expressed using a new head atom $f$ and the literal ${\sim}f$ in the body.

To define the semantics of normal programs, we write $\mathrm{At}(P)$ for the set of atoms that appear in a program $P$. An *interpretation* $I \subseteq \mathrm{At}(P)$ of $P$ determines which atoms $a \in \mathrm{At}(P)$ are *true* ($a \in I$) and which atoms are *false* ($a \in \mathrm{At}(P) \setminus I$). A rule $r$ is satisfied in $I$, denoted by $I \models r$ iff $I \models \mathrm{H}(r)$ is implied by $I \models \mathrm{B}(r)$ where $\sim$ is treated classically, i.e., $I \models {\sim}c_i$ iff $I \not\models c_i$. An interpretation $I$ is a (*classical*) *model* of $P$, denoted $I \models P$ iff $I \models r$ for each $r \in P$. A model $M \models P$ is a *minimal model* of $P$ iff there is no model $M' \models P$ such that $M' \subset M$. In particular, every positive normal program $P$ has a unique minimal model, i.e., the *least model* $\mathrm{LM}(P)$ of $P$.

The least model semantics can be extended to cover an arbitrary normal program $P$ [9] by reducing $P$ into a positive program $P^M = \{r^+ \mid r \in P \text{ and } M \cap \mathrm{B}^-(r) = \emptyset\}$ with respect to any *model candidate* $M \subseteq \mathrm{At}(P)$. In particular, an interpretation $M \subseteq \mathrm{At}(P)$ is a *stable model* of $P$ iff $M = \mathrm{LM}(P^M)$. The number of stable models can vary in general and we write $\mathrm{SM}(P)$ for the set of stable models associated with $P$. The stable model semantics was preceded by an alternative semantics, namely the one based on *supported models* [1]. A classical model $M$ of a normal program $P$ is a supported model of $P$ iff for every atom $a \in M$ there is a rule $r \in P$ such that $\mathrm{H}(r) = a$ and $M \models \mathrm{B}(r)$. Inspired by this idea, we define for any program $P$ and interpretation $I \subseteq \mathrm{At}(P)$, the set of *supporting rules* $\mathrm{SuppR}(P, I) = \{r \in P \mid I \models \mathrm{B}(r)\}$. As shown in [21], stable models are also supported models, but not necessarily vice versa.

The *positive dependency graph* $\mathrm{DG}^+(P)$ of a normal program $P$ is formally a pair $\langle \mathrm{At}(P), \leq \rangle$ where $b \leq a$ iff there is a rule $r \in P$ such that $\mathrm{H}(r) = a$ and $b \in \mathrm{B}^+(r)$. A *strongly connected component* (SCC) of $\mathrm{DG}^+(P)$ is a non-empty subset $S \subseteq \mathrm{At}(P)$ such that (i) $a \leq^* b$ holds for each $a, b \in S$ and the *reflexive* and *transitive* closure $\leq^*$

of $\leq$ and (ii) $S$ is maximal with this property. The set of rules associated with an SCC $S$ is $\mathrm{Def}_P(S) = \{r \in P \mid \mathrm{H}(r) \in S\}$. The same notation is also overloaded for single atoms by setting $\mathrm{Def}_P(a) = \mathrm{Def}_P(\{a\})$.

The *completion* of a normal program $P$, denoted by $\mathrm{Comp}(P)$, contains a formula

$$a \leftrightarrow \bigvee_{r \in \mathrm{Def}_P(a)} \Big( \bigwedge_{b \in \mathrm{B}^+(r)} b \wedge \bigwedge_{c \in \mathrm{B}^-(r)} \neg c \Big) \tag{2}$$

for each atom $a \in \mathrm{At}(P)$. For a propositional theory $T$ and its signature $\mathrm{At}(T)$, we define the set of *classical models* by setting $\mathrm{CM}(T) = \{M \subseteq \mathrm{At}(T) \mid M \models T\}$. It is well-known that $\mathrm{SuppM}(P) = \mathrm{CM}(\mathrm{Comp}(P))$ holds for any normal program $P$.

## 3 Translations

The aim of this section is to provide a compact translation of a ground logic program $P$ into a set $S$ of clauses such that the stable models of $P$ are captured as classical models of $S$. In addition to normal rules we show how to treat also choice rules and rules with cardinality and weight constraints, i.e., the class of programs supported by the SMODELS system [24]. Such a transformation will be devised in three steps:

1. First we show how to translate SMODELS programs into normal programs in a semantics preserving way (Section 3.2).
2. Then we develop a translation which extends a normal program $P$ in such a way that the *supported* models of the transformed program coincide with the stable models of $P$ (Section 3.3).
3. The final step is to map the transformed program to a set of clauses. This can be implemented directly with Clark's completion [3] because the classical models of the completion capture the supported models of the program (Section 3.4).

Step 2 is rather involved and it is based on a characterization of stable models with *level rankings* [23]. A level ranking $l(\cdot)$ is a function that maps an atom $a$ to a natural number $\mathrm{l}(a)$, i.e., its level rank. The translation in Step 2 is given using only two primitive operations for comparing the level ranks associated with atoms $a$ and $b$:

– $l(b) < l(a)$, denoted by a new atom $\mathbf{lt}(b, a)$, and
– $l(a) = l(b) + 1$, analogously denoted by an atom $\mathbf{succ}(b, a)$.

In Section 3.1 we show how to define $\mathbf{lt}(b, a)$ and $\mathbf{succ}(b, a)$ capturing conditions $l(b) < l(a)$ and $l(a) = l(b) + 1$, respectively. This is based on the idea of representing level ranks in binary using vectors of new propositional atoms following the approach of [11]. Next we outline the main ideas of the transformation in Step 2 where the aim is to devise a translation for any given normal program $P$ so that the *supported* models of the transformed program correspond to the stable models of the original program $P$.

Roughly speaking, we resort to the supported models of the original program $P$ augmented by further rules to exclude non-stable models. These rules are used to represent conditions for checking the existence of a *level ranking* [23] of atoms in the program which is closely related to a *level numbering* [11] of atoms and rules of the program. It turns out that a supported model $M$ of a program of $P$ is a stable model of $P$ iff it has a *level ranking* such that each atom $a \in M$ has a defining supporting rule whose positive body literals have a level rank strictly lower than that of $a$ [23]. To capture the existence of a level ranking the additional rules include for each normal rule (1), a rule

$$\mathbf{just}(a) \leftarrow b_1,\, \ldots,\, b_n,\, {\sim}c_1,\, \ldots,\, {\sim}c_m, \mathbf{lt}(b_1, a), \ldots, \mathbf{lt}(b_n, a). \tag{3}$$

and moreover for each atom $a$, a constraint

$$\leftarrow a,\, {\sim}\mathbf{just}(a). \tag{4}$$

where $\mathbf{just}(a)$ is a new atom and $\mathbf{lt}(b_i, a)$ is a new atom implementing the comparison $\mathrm{l}(b_i) < \mathrm{l}(a)$ between the level ranks $\mathrm{l}(b_i)$ and $\mathrm{l}(a)$ of atoms $b_i$ and $a$, respectively (see Section 3.1 for the definition of the atoms $\mathbf{lt}(b_i, a)$) .The idea of the translation is that

– a supported model of the translation gives a supported model of the original program which has a level ranking, i.e. is a stable model of the original program [23];
– a stable model of the original program can be extended to a supported model of the translation.

Hence, the translation preserves existence of models and can be used in standard reasoning tasks such as brave and cautious reasoning, and checking the existence of stable models. It can be extended to guarantee that there is a one-to-one correspondence between the stable models of the original program and the supported models of the translation by adding *strong ranking constraints* [23]. There are two variants of strong ranking constraints as put forth in [13]. The local strong ranking constraints add the requirement that each supported rule (1) has a positive body literal $b_i$ whose level ranking is at most one lower than that of the head $a$, i.e., $\mathrm{l}(b_i) + 1 \geq \mathrm{l}(a)$ holds [23]. Global strong level ranking constraints, on the other hand, concern the support of $a$ globally and require the existence of at least one supporting rule (1) where $\mathrm{l}(a) = \mathrm{l}(b_i) + 1$ holds for some $b_i$. Using rules these constraints are encoded as follows.

– The *local* strong ranking constraints include for every normal rule (1), the constraint

$$\begin{aligned}\leftarrow\, & b_1,\, \ldots,\, b_n,\, {\sim}c_1,\, \ldots,\, {\sim}c_m,\\ & \mathbf{lt}(b_1, a), {\sim}\mathbf{succ}(b_1, a), \ldots, \mathbf{lt}(b_n, a), {\sim}\mathbf{succ}(b_n, a).\end{aligned} \tag{5}$$

where each $\mathbf{lt}(b_i, a)$ is a new atom implementing the comparison $\mathrm{l}(b_i) < \mathrm{l}(a)$ and $\mathbf{succ}(b_i, a)$ the comparison $\mathrm{l}(a) = \mathrm{l}(b_i)+1$ as to be detailed in Section 3.1. Rule (5) formalizes the ranking constraint in a denial form, i.e., it cannot be the case that for a supported rule the condition $\mathrm{l}(b_i)+1 \geq \mathrm{l}(a)$ is false for every positive body literal $b_i$. Notice that the condition that $\mathrm{l}(b_i)+1 \geq \mathrm{l}(a)$ does not hold, i.e. $\mathrm{l}(b_i)+1 < \mathrm{l}(a)$ holds, is equivalently stated as the conjunction of $\mathrm{l}(b_i) < \mathrm{l}(a)$ and $\mathrm{l}(a) \neq \mathrm{l}(b_i)+1$.
– The *global* strong ranking constraints contain for each rule of the form (1), the rules

$$\begin{aligned}&\mathbf{next}(a) \leftarrow b_1,\, \ldots,\, b_n,\, {\sim}c_1,\, \ldots,\, {\sim}c_m,\, \mathbf{succ}(b_1, a).\\ &\ldots\\ &\mathbf{next}(a) \leftarrow b_1,\, \ldots,\, b_n,\, {\sim}c_1,\, \ldots,\, {\sim}c_m,\, \mathbf{succ}(b_n, a).\end{aligned} \tag{6}$$

and then for each atom $a$ a constraint

$$\leftarrow a, \sim\mathbf{next}(a). \tag{7}$$

where $\mathbf{next}(a)$ is a new atom and $\mathbf{succ}(b_i, a)$ is an atom implementing the comparison $\mathrm{l}(a) = \mathrm{l}(b_i) + 1$ given in Section 3.1.

The translation can be optimized considerably by exploiting the *strongly connected components* (SCCs) of the positive dependency graph of a program in two ways:

- The translation can be done one SCC at a time reducing substantially the number of bits needed in the binary representation of level ranks.
- The definition of non-circular justification for an atom $a$, as captured by the definition (3) of $\mathbf{just}(a)$ above, can be further separated to *external support* coming from outside of the SCC of $a$ and *internal support* originating from the SCC of $a$ and then utilizing the relationship of external and internal support.

Moreover, the translation can be made more compact by introducing new atoms to exploit the shared substructure of the rules. For example, the condition that the body $b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m$ of a rule $r$ is satisfied appears multiple times in the translation and can be distinguished by introducing a new atom $\mathbf{bt}(r)$ and a defining rule

$$\mathbf{bt}(r) \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m. \tag{8}$$

Section 3.3 describes the further details of an optimized translation.

## 3.1 Primitives for Representing Level Rankings

In this section we present primitives for handling level rankings needed in the second step of our translation from SMODELS programs into a set of clauses. In that step we are going to extend a normal program $P$ in such a way that the *supported* models of the transformed program coincide with the stable models of $P$ (see Section 3.3). The translation is based on level rankings and is stated using two primitive operations for comparing the level ranks $l(a)$ and $l(b)$ associated with atoms $a$ and $b$: $\mathbf{lt}(a, b)$ and $\mathbf{succ}(a, b)$ capturing conditions $l(a) < l(b)$ and $l(b) = l(a) + 1$, respectively.

In order to encode these operations, we closely follow the approach of [11] and use a vector $a_1 \ldots a_n$ of new propositional atoms to encode the level rank $\mathrm{l}(a)$ in binary. In such a vector, the atoms $a_1$ and $a_n$, respectively, capture the most and the least significant bits of $l(a)$. On the semantical side, the precise idea will be that the $i^{\text{th}}$ bit of the number $\mathrm{l}(a)$ equals to $1$ iff the proposition $a_i$ is true in a supported model $M$, i.e.,

$$l(a) = \sum\{2^{n-i} \mid 1 \leq i \leq n \text{ and } M \models a_i\}. \tag{9}$$

The number $n$ of bits required depends on the size of the SCC $S$ in which $a$ resides and this number equals to $\lceil \log_2 |S| \rceil$. To further optimize our encodings, we represent level ranks using off-by-one values $0, \ldots, |S| - 1$ rather than $1, \ldots, |S|$ used in [11,23].

Table 1 introduces the subprograms for defining the atoms $\mathbf{lt}(b, a)$ and $\mathbf{succ}(b, a)$. These subprograms concern two distinct atoms $a$ and $b$ involved in a non-trivial SCC

**Table 1.** Subprograms related with bit vectors

| Primitive | Definition using normal rules |
|---|---|
| $\mathbf{SEL}_n(a)$ | $\{a_i \leftarrow {\sim}\overline{a_i}.\ \ \overline{a_i} \leftarrow {\sim}a_i.\ \mid 1 \leq i \leq n\}$ |
| $\mathbf{CLR}_n(a)$ | $\{\leftarrow a_i.\ \mid 1 \leq i \leq n\}$ |
| $\mathbf{LT}_n(a,b)$ | $\{\mathbf{lt}(a,b)_i \leftarrow {\sim}a_i,\, b_i.\ \mid 1 \leq i \leq n\} \cup$ |
| | $\{\mathbf{lt}(a,b)_i \leftarrow {\sim}a_i,\, {\sim}b_i,\, \mathbf{lt}(a,b)_{i+1}.\ \mid 1 \leq i < n\} \cup$ |
| | $\{\mathbf{lt}(a,b)_i \leftarrow a_i,\, b_i,\, \mathbf{lt}(a,b)_{i+1}.\ \mid 1 \leq i < n\}$ |
| $\mathbf{EQ}_n(a,b)$ | $\{\mathbf{eq}(a,b)_1 \leftarrow a_1,\, b_1.\ \ \mathbf{eq}(a,b)_1 \leftarrow {\sim}a_1,\, {\sim}b_1.\ \mid 1 < n\} \cup$ |
| | $\{\mathbf{eq}(a,b)_i \leftarrow \mathbf{eq}(a,b)_{i-1},\, a_i,\, b_i.\ \mid 2 \leq i < n\} \cup$ |
| | $\{\mathbf{eq}(a,b)_i \leftarrow \mathbf{eq}(a,b)_{i-1},\, {\sim}a_i,\, {\sim}b_i.\ \mid 2 \leq i < n\}$ |
| $\mathbf{SUCC}_n(a,b)$ | $\{\mathbf{succ}(a,b)_1 \leftarrow {\sim}a_1,\, b_1.\ \} \cup$ |
| | $\{\mathbf{succ}(a,b)_i \leftarrow \mathbf{eq}(a,b)_{i-1},\, {\sim}a_i,\, b_i.\ \mid 2 \leq i \leq n\} \cup$ |
| | $\{\mathbf{succ}(a,b)_i \leftarrow \mathbf{succ}(a,b)_{i-1},\, a_i,\, {\sim}b_i.\ \mid 2 \leq i \leq n\}$ |

$S$ with $|S| > 1$ and their level ranks $\mathrm{l}(a)$ and $\mathrm{l}(b)$ which are represented by vectors of new atoms $a_1 \dots a_n$ and $b_1 \dots b_n$ where $n = \lceil \log_2 |S| \rceil$. For the sake of flexibility, we allow for adding extra positive/negative conditions to control the activation of these subprograms. To this end, we use a notation similar to that of normal rules (1). For example, the program $\mathbf{CLR}_n(a) \leftarrow a,\, \mathbf{ext}(a)$ would consist of integrity constraints

$$\leftarrow a_1,\, a,\, \mathbf{ext}(a).\ \dots \leftarrow a_n,\, a,\, \mathbf{ext}(a).$$

These rules illustrate how the extra conditions are added to all rules of the program subject to this construction. The primitives listed in Table 1 serve the following purposes.

1. The program $\mathbf{SEL}_n(a)$ selects a level rank $0 \leq \mathrm{l}(a) < 2^n$ for the atom $a$. Since $a_1 \dots a_n$ do not have any other defining rules we call them *input atoms*. They can be treated very succinctly in the completion step (see Section 3.4 for details). The new atoms $\overline{a_1}, \dots, \overline{a_n}$ act as complements of $a_1, \dots, a_n$ and they are only required to define input atoms with normal rules and do not appear elsewhere in our translation.
2. The role of $\mathbf{CLR}_n(a)$ is to *clear* the rank $\mathrm{l}(a)$, i.e., to assign $\mathrm{l}(a) = 0$ for atom $a$.
3. The program $\mathbf{LT}_n(a,b)$ checks whether $\mathrm{l}(a) < \mathrm{l}(b)$. To keep the length of this program linear in $n$, a vector of new atoms $\mathbf{lt}(a,b)_1 \dots \mathbf{lt}(a,b)_n$ is introduced. The informal reading of $\mathbf{lt}(a,b)_i$ is that $\mathrm{l}(a) < \mathrm{l}(b)$ holds if only bits in positions $i, \dots, n$ are taken into account. Hence, the atom $\mathbf{lt}(a,b)_1$ related with the first (most significant) bits of $\mathrm{l}(a)$ and $\mathrm{l}(b)$ captures the overall result of the comparison.
4. The purpose of $\mathbf{EQ}_n(a,b)$ is to test whether the first $i$ bits of $\mathrm{l}(a)$ and $\mathrm{l}(b)$ coincide and this is also the informal reading of the new atom $\mathbf{eq}(a,b)_i$. The case $i = n$ is not needed by the program $\mathbf{SUCC}_n(a,b)$ below and is hence omitted.

5. The program $\mathbf{SUCC}_n(a,b)$ is used to check whether $\mathrm{l}(b) = \mathrm{l}(a) + 1$ holds. This can be achieved recursively: the informal reading of $\mathbf{succ}(a,b)_i$ is that the first $i$ bits of $\mathrm{l}(a)$ and $\mathrm{l}(b)$ constitute successive numbers in binary. In particular, rules of the second kind break down recursion as they detect a trivial successor relationship for the first $i$ bits using $\mathbf{eq}(a,b)_{i-1}$ from the subprogram $\mathbf{EQ}_n(a,b)$.

## 3.2 Translating SMODELS Programs into Normal Programs

In addition to normal rules of the form (1), also known as *basic rules*, answer set programs in the SMODELS system are based on three extended rule types

$$\{a_1,\ldots,a_h\} \leftarrow b_1,\ldots,b_n,{\sim}c_1,\ldots,{\sim}c_m. \tag{10}$$

$$a \leftarrow l \leq \{b_1,\ldots,b_n,{\sim}c_1,\ldots,{\sim}c_m\}. \tag{11}$$

$$a \leftarrow w \leq \{b_1 = w_{b_1},\ldots,b_n = w_{b_n},{\sim}c_1 = w_{c_1},\ldots,{\sim}c_m = w_{c_m}\}. \tag{12}$$

where $a$, $a_i$'s, $b_j$'s, and $c_k$'s are propositional atoms. The head $\{a_1,\ldots,a_h\}$ of a *choice rule* (10) denotes a choice to be made when the body of the rule is satisfied: any of $a_i$'s can be true. The body of a *cardinality rule* (11) is satisfied when the number of satisfied literals is at least $l$. More generally, the body of a *weight rule* (12) is satisfied if the sum of weights (denoted by $w_{b_j}$'s and $w_{c_k}$'s above) of satisfied literals is at least $w$.

Since our goal is to translate any SMODELS program $P$ into a program solely consisting of normal rules (1), denoted $\mathrm{Normal}(P)$, the heads of choice rules (10) and the bodies of cardinality rules (11) and weight rules (12) have to be dealt with. A choice rule $r$ of the form (10) can be turned into a normal rule (1) by replacing its head by a new atom $\mathbf{bt}(r)$, denoting that the body of $r$ is true, and by adding normal rules

$$\begin{array}{lcl} a_1 \leftarrow \mathbf{bt}(r),\ {\sim}\overline{a_1}. & \quad\ldots\quad & a_h \leftarrow \mathbf{bt}(r),\ {\sim}\overline{a_h}. \\ \overline{a_1} \leftarrow {\sim}a_1. & \quad\ldots\quad & \overline{a_h} \leftarrow {\sim}a_h. \end{array}$$

with new atoms $\overline{a_1},\ldots,\overline{a_h}$ that denote the complements of $a_1,\ldots,a_h$, respectively.

It is quite demanding to represent the body of a cardinality rule (11) using normal rules succinctly.[1] To obtain a polynomial encoding, we resort to a representation which is formulated by Eén and Sörensson [5] when translating pseudo-Boolean constraints. In the general case, $l \times (n+m-l)$ new atoms of the form $\mathbf{cnt}(i,j)$ are needed. Here $1 \leq i \leq l$ counts the number of satisfied literals and $l-i+1 \leq j \leq n+m-i+1$ selects the $j^{\text{th}}$ literal (either $b_j$ or ${\sim}c_{j-n}$) amongst $b_1$, …, $b_n$, ${\sim}c_1$, …, ${\sim}c_m$ in the body of (11). The intuitive reading of $\mathbf{cnt}(i,j)$ is that, from the $j^{\text{th}}$ up to the $(n+m)^{\text{th}}$ literal, the number of true literals in the body of the rule is at least $i$. Hence, the atom $\mathbf{cnt}(l,1)$ captures the intended condition for making the head $a$ of (11) true. Given these considerations, a cardinality rule (11) translates into following rules:

[1] In general, there are $\binom{n+m}{l}$ ways to satisfy the body—suggesting that an exponential number of normal rules would be required to represent a cardinality rule in the worst case.

$$\begin{tikzcd}
a & \mathbf{cnt}(3,1) \arrow[l] & \mathbf{cnt}(3,2) \arrow[l, "\sim b_1"'] & \mathbf{cnt}(3,3) \arrow[l, "\sim b_2"'] \\
 & \mathbf{cnt}(2,2) \arrow[u, "b_1"'] & \mathbf{cnt}(2,3) \arrow[l, "\sim b_2"'] \arrow[u, "b_2"'] & \mathbf{cnt}(2,4) \arrow[l, "\sim b_3"'] \arrow[u, "b_3"'] \\
 & \mathbf{cnt}(1,3) \arrow[u, "b_2"'] & \mathbf{cnt}(1,4) \arrow[l, "\sim b_3"'] \arrow[u, "b_3"'] & \mathbf{cnt}(1,5) \arrow[l, "\sim \overline{c_1}"'] \arrow[u, "\sim c_1"'] \\
 & {} \arrow[u, "b_3"'] & {} \arrow[u, "\sim c_1"'] & {} \arrow[u, "\sim c_2"']
\end{tikzcd}$$

**Fig. 1.** Counting grid capturing a cardinality constraint

| | |
|---|---|
| $\mathbf{cnt}(1,j) \leftarrow b_j.$ | $(l \leq j \leq n)$ |
| $\mathbf{cnt}(1,j) \leftarrow \sim c_{j-n}.$ | $(\max(l, n+1) \leq j \leq n+m)$ |
| $\mathbf{cnt}(i,j) \leftarrow \mathbf{cnt}(i-1,j),\ b_j.$ | $(1 < i \leq l$ and $l-i+1 \leq j \leq n)$ |
| $\mathbf{cnt}(i,j) \leftarrow \mathbf{cnt}(i-1,j),\ \sim c_{j-n}.$ | $(1 < i \leq l$ and $\max(l,n+1) \leq j \leq n+m-i+1)$ |
| $\mathbf{cnt}(i,j) \leftarrow \mathbf{cnt}(i,j+1),\ \sim b_j.$ | $(1 < i \leq l$ and $l-i+1 \leq j < n)$ |
| $\mathbf{cnt}(i,j) \leftarrow \mathbf{cnt}(i,j+1),\ \sim \overline{c_{j-n}}.$ | $(1 < i \leq l$ and $\max(l,n+1) \leq j < n+m-i+1)$ |
| $\overline{c_{j-n}} \leftarrow \sim c_{j-n}.$ | $(\max(l,n+1) \leq j < n+m-i+1)$ |
| $a \leftarrow \mathbf{cnt}(l,1).$ | |

These rules give rise to a kind of a *counting grid*. One such grid is illustrated in Figure 1 in the case of parameter values $l = 3$, $n = 3$, and $m = 2$. As regards the soundness of the overall transformation, the complementary atoms $\overline{c_{j-n}}$ used in the translation deserve special attention. They pre-empt the creation of new *positive dependencies*[2] between the head atom $a$ and the negative body atoms $c_1, \ldots, c_n$ and are therefore crucial under stable model semantics. This aspect does not arise in Eén and Sörensson's translation [5] due to the symmetry of truth values under classical semantics.

*Example 1.* Consider a cardinality rule $a \leftarrow 3 \leq \{b_1, b_2, b_3, \sim c_1, \sim c_2\}$ and its translation as illustrated by the grid structure in Figure 1. Literals marked along the arrows give the conditions on which *truth* can be propagated for atoms appearing in the nodes of the grid. For instance, if $\sim c_2$ and $\sim c_1$ are satisfied, atoms $\mathbf{cnt}(1,5)$ and $\mathbf{cnt}(2,4)$ can be inferred to be true. If $b_3$ and $b_2$ are not satisfied, then $\sim b_3$ and $\sim b_2$ are. Consequently, atoms $\mathbf{cnt}(2,3)$ and $\mathbf{cnt}(2,2)$ become true. If, in addition, $b_1$ is satisfied, then $\mathbf{cnt}(3,1)$ and eventually $a$ can be inferred to be true. In this way, the levels 1, 2, and 3 count the number of satisfied literals. There are also alternative ways to infer $a$ true. Intuitively, each vertical "move" in the grid captures the satisfaction of one further literal whereas any horizontal one leaves the count of true literals unchanged. ■

The treatment of weight rules (12) is much more challenging in the general case. To simplify the setting, we concentrate on the case $m = 0$. This is without loss of generality because an arbitrary weight rule can be transformed into such a form by introducing complementary atoms defined in terms of normal rules $\overline{c_1} \leftarrow \sim c_1, \ldots, \overline{c_m} \leftarrow \sim c_m$ and by rewriting (12) as a purely *positive* weight rule

$$a \leftarrow w \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, \overline{c_1} = w_{c_1}, \ldots, \overline{c_m} = w_{c_m}\}. \tag{13}$$

[2] The notion of positive dependency straightforwardly generalizes for the rule types (10)–(12).

where the weights $w_{c_1}, \ldots, w_{c_m}$ remain unchanged. This step is also beneficial for the forthcoming translation because the definitions of $\overline{c_1}, \ldots, \overline{c_m}$ block any positive dependencies from the head $a$ to the negative body atoms $c_1, \ldots, c_m$.

As regards the evaluation of the rule body (12) in the case $m = 0$, we take a quite straightforward approach and count the sum of weights associated with satisfied body literals dynamically. The intermediate results of summing up these numbers are represented using vectors of propositional atoms in analogy to Section 3.1. The number of bits $b$ required is determined by the bound $w$ and the unconditional (maximum) sum of weights $\sum_{1 \leq i \leq n} w_{b_i}$. Suppose that the vector $y_1 \ldots y_b$ encodes the sum up to first $i-1$ atoms and $x_1 \ldots x_b$ is the new value to be computed using $b_i$ in the body of (12) with $m = 0$. If $b_i$ is not satisfied, then $x_1 \ldots x_b$ is simply made a copy of $y_1 \ldots y_b$ as follows:

$$x_j \leftarrow y_j, {\sim}b_i. \quad (1 \leq j \leq b \text{ and } 1 \leq i \leq n) \tag{14}$$

Otherwise, the vector $x_1 \ldots x_b$ should represent the sum of $y_1 \ldots y_b$ and the weight $w_1 \ldots w_b$ of $b_i$ in binary. This is achieved by the following rules conditioned by $b_i$:

| $w_j = 0$ | $w_j = 1$ |
|---|---|
| $x_j \leftarrow b_i,\ y_j,\ {\sim}\mathbf{carry}(x_{j+1}, y_{j+1}).$<br>$x_j \leftarrow b_i,\ {\sim}y_j,\ \mathbf{carry}(x_{j+1}, y_{j+1}).$<br>$\mathbf{carry}(x_j, y_j) \leftarrow$<br>$\quad b_i,\ y_j,\ \mathbf{carry}(x_{j+1}, y_{j+1}).$ | $x_j \leftarrow b_i,\ y_j,\ \mathbf{carry}(x_{j+1}, y_{j+1}).$<br>$x_j \leftarrow b_i,\ {\sim}y_j,\ {\sim}\mathbf{carry}(x_{j+1}, y_{j+1}).$<br>$\mathbf{carry}(x_j, y_j) \leftarrow b_i,\ y_j.$<br>$\mathbf{carry}(x_j, y_j) \leftarrow b_i,\ \mathbf{carry}(x_{j+1}, y_{j+1}).$ |
| $x_b \leftarrow b_i,\ y_b.$ | $x_b \leftarrow b_i,\ {\sim}y_b.$<br>$\mathbf{carry}(x_b, y_b) \leftarrow b_i,\ y_b.$ |

In the rules above, the index $j$ varies in the range $1 \leq j < b$ and the new atom $\mathbf{carry}(x_j, y_j)$ represents the carry bit associated with $x_j$ and $y_j$. The translation depends statically on the value of $j^{\text{th}}$ bit $w_j$ of the weight of $b_i$. Now, if $x$ represents the result of all additions for $b_1, \ldots, b_n$ and $w$ is the overall limit, then $a \leftarrow \mathbf{gte}(x, w)_1$ captures the weight rule (12) with $m = 0$ in the presence of the subprogram $\mathbf{GTE}_b(x, w)$:

$$\begin{array}{ll}
\mathbf{gte}(x, w)_j \leftarrow x_j. & (w_j = 0 \text{ and } 1 \leq j \leq b) \\
\mathbf{gte}(x, w)_j \leftarrow {\sim}x_j,\ \mathbf{gte}(x, w)_{j+1}. & (w_j = 0 \text{ and } 1 \leq j < b) \\
\mathbf{gte}(x, w)_j \leftarrow x_j,\ \mathbf{gte}(x, w)_{j+1}. & (w_j = 1 \text{ and } 1 \leq j < b)
\end{array}$$

This program formalizes the test for *greater than or equal* and is in this sense complementary to $\mathbf{LT}_b(x, w)$ given in Table 1. However, static role of the weight $w$ is fully taken into account. More importantly, we cannot use $a \leftarrow {\sim}\mathbf{lt}(x, w)_1$ as the overall translation of (12) with $m = 0$ because the positive dependency of $a$ on $b_1, \ldots, b_n$ would be lost—accordingly endangering the soundness of our transformation.

**Theorem 1.** *Let $P$ be an* SMODELS *program.*

1. *If $M \in \mathrm{SM}(P)$, then there is a unique stable model $N \in \mathrm{SM}(\mathrm{Normal}(P))$ such that $M = N \cap \mathrm{At}(P)$.*
2. *If $N \in \mathrm{SM}(\mathrm{Normal}(P))$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

Using the terminology from [12], the outcome of Theorem 1 is that if new atoms are hidden, any SMODELS program $P$ is *visibly equivalent* to the result $\mathrm{Normal}(P)$ of normalization. The normalization procedure as described above is also highly modular as it can be constructed on a rule-by-rule basis. As regards the soundness of normalization (Theorem 1), it is important that the positive dependencies between atoms in $\mathrm{At}(P)$ are preserved in spite of the new atoms involved in $\mathrm{Normal}(P)$.

Normalization procedures have also been implemented in other ASP solvers. In the CMODELS system, a reduction [7] from weight rules to *nested rules* [17] is applied. Basically, such a reduction can be exponential in the worst case but it remains polynomial if new atoms are introduced [7]. Optionally, also CLASP can remove extended rule types from its input program if supplied the option flag `-trans-ext=all` from the command line. According to the source code of CLASP both a worst-case quadratic and a worst-case exponential transformation of cardinality/weight rules are implemented.

## 3.3 Capturing Stability Using Supported Models

We proceed to the details of a translation function $\mathrm{LP2LP}^w(\cdot)$ sketched in the beginning of Section 3. It is a mapping within the class of normal programs—aiming at a semantical shift from stable models to supported models. Our first mapping will be based on weak ranking constraints only, hence the superscript $w$ in the notation.

The translation $\mathrm{LP2LP}^w(P)$ exploits the SCCs of the positive dependency graph of the input program $P$ in order to reduce the number of bits required to represent level ranks of atoms. Essentially, the treatment of an SCC $S \subseteq \mathrm{At}(P)$ involves the translation of the atoms in $S$ and their defining rules, i.e., the subprogram $\mathrm{Def}_P(S)$. By a slight abuse of notation, we write $\mathrm{LP2LP}^w(S)$ to denote the translation $\mathrm{LP2LP}^w(\mathrm{Def}_P(S))$ of the component $S$. Given a partitioning of $\mathrm{At}(P)$ into SCCs $S_1, \ldots, S_m$ and the respective subprograms $\mathrm{Def}_P(S_1), \ldots, \mathrm{Def}_P(S_m)$ of $P$, the translation $\mathrm{LP2LP}^w(P)$ is feasible component-by-component. Although $\mathrm{LP2LP}^w(\cdot)$ is *non-modular* in general, the translation $\mathrm{LP2LP}^w(P)$ can be realized as the union

$$\textstyle\bigcup_{i=1}^{m} \mathrm{LP2LP}^w(S_i).$$

This program is typically different from $\mathrm{LP2LP}^w(P)$ because level ranks are representable with fewer bits. This also reveals the essential source of non-modularity in the translation. The case of a trivial (singleton) SCC $\{a\}$ is straightforward: it is sufficient to remove all tautological rules $r \in \mathrm{Def}_P(a)$ for which $a \in \mathrm{B}^+(r)$. Thus, we define

$$\mathrm{LP2LP}^w(\{a\}) = \{r \in \mathrm{Def}_P(a) \mid a \notin \mathrm{B}^+(r)\}.$$

Let us then consider any *non-trivial* SCC $S$ satisfying $|S| > 1$ and any atom $a \in S$ involved in this component. Following an existing translation [13] into difference logic, we introduce two new atoms $\mathbf{ext}(a)$ and $\mathbf{int}(a)$: these formalize the *external* and *internal* support for the atom $a$ via its definition $\mathrm{Def}_P(a) \subseteq \mathrm{Def}_P(S)$. The atom $a$ contributes the following parts to the translation $\mathrm{LP2LP}^w(S)$ where $n = \lceil \log_2 |S| \rceil$:

1. Each rule $r \in \mathrm{Def}_P(a)$ is split into two normal rules

$$a \leftarrow \mathbf{bt}(r). \qquad \mathbf{bt}(r) \leftarrow \mathrm{B}(r). \tag{15}$$

   The latter rule coincides with (8) and the intermediate new atom $\mathbf{bt}(r)$ will control the activation of other rules which depend on the satisfaction of this rule body.

2. For each rule $r \in \mathrm{Def}_P(a)$ such that $\mathrm{B}^+(r) \cap S = \emptyset$, a rule for external support:

$$\mathbf{ext}(a) \leftarrow \mathbf{bt}(r). \tag{16}$$

3. The program $\mathbf{SEL}_n(a)$ from Table 1 is used to select a level rank $\mathrm{l}(a)$. Additionally, it is important to assign $\mathrm{l}(a) = 0$ using subprograms $\mathbf{CLR}_n(a) \leftarrow {\sim}a$ and $\mathbf{CLR}_n(a) \leftarrow a, \mathbf{ext}(a)$. These rules decrease degrees of freedom and they become highly relevant when a one-to-one correspondence of models is sought later.
4. For each rule $r \in \mathrm{Def}_P(a)$ such that $\mathrm{B}^+(r) \cap S = \{b_1, \ldots, b_m\} \neq \emptyset$, a rule

$$\mathbf{int}(a) \leftarrow \mathbf{bt}(r), \mathbf{lt}(b_1, a)_1, \ldots, \mathbf{lt}(b_m, a)_1, {\sim}\mathbf{ext}(a). \tag{17}$$

which captures internal support for $a$ together with the subprograms $\mathbf{LT}_n(b_1, a) \leftarrow \mathbf{bt}(r), \ldots, \mathbf{LT}_n(b_m, a) \leftarrow \mathbf{bt}(r)$. These programs are used to evaluate the conditions $\mathrm{l}(b_1) < \mathrm{l}(a), \ldots, \mathrm{l}(b_m) < \mathrm{l}(a)$ on level ranks associated with the rule $r$.
5. Finally, we require that a true atom in a component must have either external or internal support by including an integrity constraint of the form

$$\leftarrow a, {\sim}\mathbf{ext}(a), {\sim}\mathbf{int}(a). \tag{18}$$

Roughly speaking, the rules of the translation $\mathrm{LP2LP}^w(S)$ aim to express that if an atom $a \in S$ of a non-trivial SCC $S$ is true in a stable model $M \in \mathrm{SM}(P)$, then it must have either external or internal support by (18). This is how (4) is implemented when these types of support are distinguished. External support can only be provided by a rule $r \in \mathrm{Def}_P(a)$ which does not positively depend on $S$ and whose body is true under $M$, i.e., $\mathrm{B}^+(r) \cap S = \emptyset$ and $r \in \mathrm{SuppR}(\mathrm{Def}_P(a), M)$. If none of the rules of the forms (16) and (15) enable the derivation of $\mathbf{ext}(a)$, then $\mathbf{int}(a)$ must be derivable by (17). This means that $a$ must have internal support: a rule $r \in \mathrm{SuppR}(\mathrm{Def}_P(a), M)$ such that $\mathrm{B}^+(r) \cap S \neq \emptyset$ and $\mathrm{l}(a) > \mathrm{l}(b)$ for every $b \in \mathrm{B}^+(r) \cap S$. This captures (3) but restricted to positive body atoms within the SCC in which $a$ resides. The net effect of the rules introduced above is that the non-stable supported models of $P$ are eliminated.

*Example 2.* Consider a normal logic program $P$ consisting of the following six rules:

$$\begin{array}{lll} r_1 : a \leftarrow b, c. & r_3 : c \leftarrow {\sim}d. & r_5 : a \leftarrow d. \\ r_2 : b \leftarrow a, {\sim}d. & r_4 : d \leftarrow {\sim}c. & r_6 : b \leftarrow a, {\sim}c. \end{array}$$

The graph $\mathrm{DG}^+(P)$ contains three SCCs, viz. $S_1 = \{a, b\}$, $S_2 = \{c\}$, and $S_3 = \{d\}$. Hence, the rules $r_3$ and $r_4$, related with the trivial SCCs of $P$ are left untouched. On the other hand, the translation of $S_1$ requires the distinction of internal and external support for $a$ and $b$. To this end, we rewrite the rules in $\mathrm{Def}_P(S_1) = \{r_1, r_2, r_5, r_6\}$ by (15):

$$\begin{array}{llll} a \leftarrow \mathbf{bt}(r_1). & \mathbf{bt}(r_1) \leftarrow b, c. & a \leftarrow \mathbf{bt}(r_5). & \mathbf{bt}(r_5) \leftarrow d. \\ b \leftarrow \mathbf{bt}(r_2). & \mathbf{bt}(r_2) \leftarrow a, {\sim}d. & b \leftarrow \mathbf{bt}(r_6). & \mathbf{bt}(r_6) \leftarrow a, {\sim}c. \end{array}$$

This is the first part of the translation $\mathrm{LP2LP}^w(S_1)$. The second part captures the potential external support for $a$ in terms of a rule of the form (16):

$$\mathbf{ext}(a) \leftarrow \mathbf{bt}(r_5).$$

No such rules are introduced for $b$ as it does not have a defining rule $r$ such that $\mathrm{B}^+(r) \cap S_1 = \emptyset$. The rest deals with level rankings that are representable with $n = \lceil \log_2 2 \rceil = 1$ bit, i.e., only ranks 0 and 1 are essentially required. The third part chooses level ranks using subprograms $\mathbf{SEL}_1(a)$ and $\mathbf{SEL}_1(b)$ as well as subprograms

$$\begin{array}{ll} \mathbf{CLR}_1(a) \leftarrow {\sim}a. & \mathbf{CLR}_1(b) \leftarrow {\sim}b. \\ \mathbf{CLR}_1(a) \leftarrow a,\, \mathbf{ext}(a). & \mathbf{CLR}_1(b) \leftarrow b,\, \mathbf{ext}(b). \end{array}$$

i.e., the following rules altogether:

$$\begin{array}{ll} a_1 \leftarrow {\sim}\overline{a_1}. \quad \overline{a_1} \leftarrow {\sim}a_1. & b_1 \leftarrow {\sim}\overline{b_1}. \quad \overline{b_1} \leftarrow {\sim}b_1. \\ \leftarrow a_1,\, {\sim}a. & \leftarrow b_1,\, {\sim}b. \\ \leftarrow a_1,\, a,\, \mathbf{ext}(a). & \leftarrow b_1,\, b,\, \mathbf{ext}(b). \end{array}$$

The weak ranking constraints about $a$ and $b$ are captured by rules of the form (17) and subprograms $\mathbf{LT}_1(b,a) \leftarrow \mathbf{bt}(r_1)$, $\mathbf{LT}_1(a,b) \leftarrow \mathbf{bt}(r_2)$, and $\mathbf{LT}_1(a,b) \leftarrow \mathbf{bt}(r_6)$:

$$\begin{array}{ll} \mathbf{int}(a) \leftarrow \mathbf{bt}(r_1),\, \mathbf{lt}(b,a)_1,\, {\sim}\mathbf{ext}(a). & \mathbf{lt}(b,a)_1 \leftarrow {\sim}b_1,\, a_1,\, \mathbf{bt}(r_1). \\ \mathbf{int}(b) \leftarrow \mathbf{bt}(r_2),\, \mathbf{lt}(a,b)_1,\, {\sim}\mathbf{ext}(b). & \mathbf{lt}(a,b)_1 \leftarrow {\sim}a_1,\, b_1,\, \mathbf{bt}(r_2). \\ \mathbf{int}(b) \leftarrow \mathbf{bt}(r_6),\, \mathbf{lt}(a,b)_1,\, {\sim}\mathbf{ext}(b). & \mathbf{lt}(a,b)_1 \leftarrow {\sim}a_1,\, b_1,\, \mathbf{bt}(r_6). \end{array}$$

Finally, the relationship of internal and external support must be constrained by (18).

$$\leftarrow a,\, {\sim}\mathbf{ext}(a),\, {\sim}\mathbf{int}(a). \qquad \leftarrow b,\, {\sim}\mathbf{ext}(b),\, {\sim}\mathbf{int}(b).$$

The supported models of the translation $\mathrm{LP2LP}^w(P)$ introduced so far are

$$M_1 = \{c\} \text{ and } M_2 = \{a, b, d, b_1, \overline{a_1}, \mathbf{int}(b), \mathbf{bt}(r_6), \mathbf{lt}(a,b)_1, \mathbf{ext}(a), \mathbf{bt}(r_5)\}.$$

In particular, there is no *supported* model $M$ extending $\{a, b, c\}$ simply because this would imply the falsity of $\mathbf{ext}(a)$ and $\mathbf{ext}(b)$ which, in turn, necessitates the truth of $\mathbf{int}(a)$ and $\mathbf{int}(b)$ in $M$. But then both $\mathbf{lt}(a,b)_1$ and $\mathbf{lt}(b,a)_1$ ought to be true under $M$ by the defining rules of $\mathbf{int}(a)$ and $\mathbf{int}(b)$. This is clearly impossible and so is $M$. Moreover, we deduce $l(a) = 0$ and $l(b) = 1$ by applying Equation (9) to $M_2$. ■

**Theorem 2.** *Let $P$ be a normal logic program.*

1. *If $M \in \mathrm{SM}(P)$, then there is $N \in \mathrm{SuppM}(\mathrm{LP2LP}^w(P))$ so that $M = N \cap \mathrm{At}(P)$.*
2. *If $N \in \mathrm{SuppM}(\mathrm{LP2LP}^w(P))$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

However, the relationship of stable and supported models established in Theorem 2 need not be one-to-one although this happens to be the case in Example 2. The situation would already be different if further bits were introduced to select ranks $\mathrm{l}(a)$ and $\mathrm{l}(b)$ leading to a one-to-many relationship of models. Given these observations about model correspondence, the translation $\mathrm{LP2LP}^w(P)$ will be sufficient when the goal is to compute only one stable model, or to check the existence of stable models. However, if the goal is to compute *all* stable models, or to *count* the number of stable models, further constraints have to be expressed (using normal rules subject to supported models) in order to avoid generation of stable models $N_1, N_2 \in \mathrm{SuppM}(\mathrm{LP2LP}^w(P))$ satisfying $N_1 \neq N_2$, $N_1 \cap \mathrm{At}(P) = M = N_2 \cap \mathrm{At}(P)$, and $M \in \mathrm{SM}(P)$ by Theorem 2. In what follows, we strengthen the conditions on level rankings by introducing rules similar to (5), (6), and (7), but which take the SCC of $a$ properly into account and try to share primitives from Table 1 as far as possible:

6. For each rule $r \in \mathrm{Def}_P(a)$ such that $\mathrm{B}^+(r) \cap S = \{b_1, \ldots, b_m\} \neq \emptyset$, a constraint

$$\leftarrow \mathbf{bt}(r),\ \mathbf{lt}(b_1, a)_1,\ {\sim}\mathbf{succ}(b_1, a)_n,\ \ldots,\ \mathbf{lt}(b_m, a)_1,\ {\sim}\mathbf{succ}(b_m, a)_n \tag{19}$$

7. For each rule $r \in \mathrm{Def}_P(a)$ such that $\mathrm{B}^+(r) \cap S = \{b_1, \ldots, b_m\} \neq \emptyset$, rules

$$\mathbf{next}(a) \leftarrow \mathbf{bt}(r),\ \mathbf{succ}(b_1, a)_n. \quad \ldots \quad \mathbf{next}(a) \leftarrow \mathbf{bt}(r),\ \mathbf{succ}(b_m, a)_n. \tag{20}$$

together with one global constraint

$$\leftarrow \mathbf{int}(a),\ {\sim}\mathbf{next}(a). \tag{21}$$

These rules must be accompanied by the respective subprograms

$$\mathbf{SUCC}_n(b_1, a) \leftarrow \mathbf{bt}(r), \ldots, \mathbf{SUCC}_n(b_m, a) \leftarrow \mathbf{bt}(r) \text{ and}$$
$$\mathbf{EQ}_n(b_1, a) \leftarrow \mathbf{bt}(r), \ldots, \mathbf{EQ}_n(b_m, a) \leftarrow \mathbf{bt}(r)$$

required by them. Items 6 and 7 above give rise three new variants of our translation:

- $\mathrm{LP2LP}^{wl}(P)$ extends $\mathrm{LP2LP}^{w}(P)$ with constraints from Item 6,
- $\mathrm{LP2LP}^{wg}(P)$ extends it with rules and the global constraint from Item 7, and
- $\mathrm{LP2LP}^{wlg}(P)$ extends $\mathrm{LP2LP}^{w}(P)$ using both.

As a result of these additional constraints, if it is possible to assign a level rank $\mathrm{l}(a)$ to each atom residing in a non-trivial SCC of $\mathrm{DG}^+(P)$ and true in a supported model $M$ of any translation $\mathrm{LP2LP}^{w*}(P)$ above, this can be done only in a unique way.

*Example 3.* For the program $P$ of Example 2, the translation $\mathrm{LP2LP}^{w}(P)$ was already presented. As $n = 1$, the constraints and rules arising from (19) are simply as follows:

$$\begin{array}{ll}
\leftarrow \mathbf{bt}(r_1),\ \mathbf{lt}(b, a)_1,\ {\sim}\mathbf{succ}(b, a)_1. & \mathbf{succ}(b, a)_1 \leftarrow {\sim}b_1,\ a_1,\ \mathbf{bt}(r_1).\\
\leftarrow \mathbf{bt}(r_2),\ \mathbf{lt}(a, b)_1,\ {\sim}\mathbf{succ}(a, b)_1. & \mathbf{succ}(a, b)_1 \leftarrow {\sim}a_1,\ b_1,\ \mathbf{bt}(r_2).\\
\leftarrow \mathbf{bt}(r_6),\ \mathbf{lt}(a, b)_1,\ {\sim}\mathbf{succ}(a, b)_1. & \mathbf{succ}(a, b)_1 \leftarrow {\sim}a_1,\ b_1,\ \mathbf{bt}(r_6).
\end{array}$$

On the other hand, the global formulations are based on (20) and (21):

$$\begin{array}{ll}
\mathbf{next}(a) \leftarrow \mathbf{bt}(r_1),\ \mathbf{succ}(b, a)_1. & \mathbf{next}(b) \leftarrow \mathbf{bt}(r_2),\ \mathbf{succ}(a, b)_1.\\
 & \mathbf{next}(b) \leftarrow \mathbf{bt}(r_6),\ \mathbf{succ}(a, b)_1.\\
\leftarrow \mathbf{int}(a),\ {\sim}\mathbf{next}(a). & \leftarrow \mathbf{int}(b),\ {\sim}\mathbf{next}(b).
\end{array}$$

The stable models of the resulting translations are essentially the same (recall $M_1$ and $M_2$ from Example 2) except that $M_2$ has to be extended by the following new atoms related with the additional rules and supposed to be true in $M_2$: $\mathbf{succ}(a, b)_1$ in any case and $\mathbf{next}(b)$ if rules of the form (20) are present. Given these additions to $M_2$, it is clear that none of the constraints listed above is violated by $M_2$. ■

**Theorem 3.** *Let $P$ be a normal logic program.*

1. *If $M \in \mathrm{SM}(P)$, then there is a unique model $N \in \mathrm{SuppM}(\mathrm{LP2LP}^{wl}(P))$ such that $M = N \cap \mathrm{At}(P)$.*
2. *If $N \in \mathrm{SuppM}(\mathrm{LP2LP}^{wl}(P))$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

The analogs of Theorem 3 hold for translations $\mathrm{LP2LP}^{wg}(P)$ and $\mathrm{LP2LP}^{wlg}(P)$.

## 3.4 Completion and Clausification

The final translation presented in this section aims to capture supported models of a normal logic program $P$ using Clark's completion [3] and classical models. For the completion, we basically apply (2). However, in order to keep the resulting clausal representation of the completion, denoted by $\mathrm{CCC}(P)$, linear in the length of $P$, we introduce a new atom for each rule body—also known as the Tseitin transformation [25]. To define the contribution of an atom $a \in \mathrm{At}(P)$ in $\mathrm{CCC}(P)$, we refer to its definition $\mathrm{Def}_P(a) = \{r_1, \ldots, r_k\}$ and introduce a new atom $\mathbf{bt}(r_i)$ for each rule $r_i$ involved. An individual rule $r_i$ of the form (1) is encoded by

$$\mathbf{bt}(r_i) \leftrightarrow b_1 \wedge \cdots \wedge b_n \wedge \neg c_1 \wedge \cdots \wedge \neg c_m. \tag{22}$$

Then the definition $\mathrm{Def}_P(a)$ is essentially captured by an equivalence

$$a \leftrightarrow \mathbf{bt}(r_1) \vee \cdots \vee \mathbf{bt}(r_k). \tag{23}$$

Each equivalence (22) with $1 \leq i \leq k$ contributes to $\mathrm{CCC}(P)$ the following clauses:

$$\begin{array}{l}\mathbf{bt}(r_i) \vee \neg b_1 \vee \cdots \vee \neg b_n \vee c_1 \vee \cdots \vee c_m, \\ \neg\mathbf{bt}(r_i) \vee b_1, \ \ldots, \ \neg\mathbf{bt}(r_i) \vee b_n, \\ \neg\mathbf{bt}(r_i) \vee \neg c_1, \ \ldots, \ \neg\mathbf{bt}(r_i) \vee \neg c_m.\end{array}$$

Finally, the formula (23) clausifies into

$$\neg a \vee \mathbf{bt}(r_1) \vee \cdots \vee \mathbf{bt}(r_k), \text{ and } a \vee \neg\mathbf{bt}(r_1), \ \ldots, \ a \vee \neg\mathbf{bt}(r_k).$$

We distinguish certain special cases in the implementation, though. If $k = 1$, $\mathbf{bt}(r_1)$ is not introduced and $a$ is substituted for $\mathbf{bt}(r_1)$ in (22) which replaces (23). If $k = 0$, then a unit clause $\neg a$ is created to falsify $a$. This clause is omitted if $a$ happens to be an *input atom* and is supposed to vary freely. Last, if the truth values of certain atoms are known in advance, it is possible to simplify resulting clauses using them. However, we leave any further propagation of truth values as the SAT solvers' responsibility.

**Theorem 4.** *Let $P$ be a normal logic program.*

1. *If $M \in \mathrm{SuppM}(P)$, then there is a unique $N \in \mathrm{CM}(\mathrm{CCC}(P))$ such that $M = N \cap \mathrm{At}(P)$.*
2. *If $N \in \mathrm{CM}(\mathrm{CCC}(P))$, then $M \in \mathrm{SuppM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

**Corollary 1.** *Let $P$ be an* SMODELS *program and $S$ the set of clauses*

$$\mathrm{CCC}(\mathrm{LP2LP}^{wl}(\mathrm{Normal}(P))).$$

1. *If $M \in \mathrm{SM}(P)$, then there is a unique interpretation $N \subseteq \mathrm{At}(S)$ such that $N \models S$ and $M = N \cap \mathrm{At}(P)$.*
2. *If $N \models S$ for $N \subseteq \mathrm{At}(S)$, then $M \in \mathrm{SM}(P)$ for $M = N \cap \mathrm{At}(P)$.*

An analogous corollary holds for the respective translation obtained using strong global ranking constraints or both, i.e., translation $\mathrm{LP2LP}^{wg}(\cdot)$ or $\mathrm{LP2LP}^{wlg}(\cdot)$.

```
gringo program.lp \
| smodels -internal -nolookahead \
| lpcat \
| lp2normal \
| igen \
| smodels -internal -nolookahead \
| lpcat -s=symbols.lp \
| lp2lp2 \
| lp2sat -n \
| minisat
```

**Fig. 2.** Unix shell pipeline for running LP2SAT2

## 4 Implementation and Experiments

In this section, we present a prototype implementation of the program transformations described in Section 3 and report the results of a performance analysis based on the benchmark instances used in the Second Answer Set Programming Competition [4].

The main components of our preliminary implementation[3] are three translators, namely LP2NORMAL (v. 1.11), LP2LP2 (v. 1.17), and LP2SAT (v. 1.15), corresponding to the respective program transformations $\mathrm{Normal}(\cdot)$, $\mathrm{LP2LP}^{w*}(\cdot)$, and $\mathrm{CCC}(\cdot)$ described in Section 3. Each tool expects to receive its input in the SMODELS file format. The first translator, LP2NORMAL, produces only basic rules (1) as required by the last two translators, i.e., LP2LP2 and LP2SAT. The latter of these produces a CNF in the DIMACS format as its output. Hence, typical SAT solvers can be used as model finders for its output and we tried out MINISAT[4] (v. 1.14 and v. 2.2.0), PRECOSAT[5] (v. 570), and CLASP[6] (v. 1.3.5 in SAT solver mode) in our experiments as back-ends to our translators. We call the combination of the three translators LP2SAT2 that can be considered as a new version of the overall system LP2SAT[7] proposed in [11]. However, this system employs a different translation and also a translator LP2ATOMIC instead of LP2LP2. The transformation implemented by LP2ATOMIC effectively removes all positive subgoals from a normal logic program given as its input: an *atomic* normal program consists of rules of the form (1) with $n = 0$ which makes Clark's completion trivially sound with respect to the stable model semantics.

Figure 2 demonstrates how LP2SAT2 is used in our tests in terms of a shell command line. The first step uses GRINGO (v. 2.0.5) to ground the answer set program `program.lp` in question. The resulting ground program is then forwarded for SMODELS[8] (v. 2.34) for simplification using the principles of the well-founded semantics [26] among others. As the next step, we compress the ground program after simplification using LPCAT (v. 1.18). As a result of a *relocation* step, only consecutive atom numbers appear in the resulting program. Then the normalization takes place using LP2NORMAL

[3] All of our tools are available under http://www.tcs.hut.fi/Software/
[4] http://minisat.se/
[5] http://fmv.jku.at/precosat/
[6] http://www.cs.uni-potsdam.de/clasp/
[7] http://www.tcs.hut.fi/Software/lp2sat/
[8] http://www.tcs.hut.fi/Software/smodels/

**Table 2.** Numbers of solved problem instances

| Benchmark / Number of Instances | | CLASP | CMODELS | LP2DIFF | LP2SAT | LP2SAT2 | LP2SAT2 -l | LP2SAT2 -g | LP2SAT2 -lg |
|---|---|---|---|---|---|---|---|---|---|
| 15Puzzle | 16 | 16 | 11 | 0 | 16 | 16 | 16 | 16 | 16 |
| BlockedNQueens | 29 | 29 | 28 | 29 | 29 | 29 | 29 | 29 | 29 |
| ChannelRouting | 11 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| ConnectedDominatingSet | 21 | 20 | 21 | 18 | 21 | 21 | 21 | 20 | 21 |
| DisjunctiveScheduling | 10 | 5 | 5 | 1 | 5 | 5 | 5 | 5 | 5 |
| EdgeMatching | 29 | 29 | 29 | 5 | 29 | 29 | 29 | 29 | 29 |
| Fastfood | 29 | 29 | 16 | 29 | 28 | 29 | 29 | 29 | 29 |
| GeneralizedSlitherlink | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 |
| GraphColouring | 29 | 9 | 5 | 9 | 9 | 10 | 10 | 10 | 10 |
| GraphPartitioning | 13 | 13 | 10 | 8 | 5 | 6 | 6 | 6 | 6 |
| HamiltonianPath | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 | 29 |
| Hanoi | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 | 15 |
| HierarchicalClustering | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| KnightTour | 10 | 10 | 8 | 6 | 1 | 1 | 1 | 1 | 1 |
| Labyrinth | 29 | 26 | 13 | 0 | 0 | 23 | 20 | 22 | 17 |
| MazeGeneration | 29 | 25 | 25 | 27 | 15 | 19 | 17 | 16 | 17 |
| SchurNumbers | 29 | 29 | 25 | 28 | 29 | 28 | 28 | 28 | 28 |
| Sokoban | 29 | 29 | 27 | 29 | 29 | 29 | 29 | 29 | 28 |
| Solitaire | 27 | 19 | 15 | 22 | 20 | 22 | 22 | 22 | 22 |
| Sudoku | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| TravellingSalesperson | 29 | 29 | 24 | 29 | 27 | 0 | 29 | 29 | 29 |
| WeightBoundedDominatingSet | 29 | 27 | 16 | 25 | 15 | 28 | 28 | 28 | 28 |
| WireRouting | 23 | 23 | 6 | 10 | 6 | 6 | 7 | 5 | 6 |
| TOTAL | 516 | 470 | 387 | 378 | 387 | 404 | 429 | 427 | 424 |

and IGEN (v. 1.5) is used to remove the input interface of the program for backward compatibility reasons. Then the program can be passed on for another round of simplification using SMODELS. After that LPCAT is used to compress the program again and to extract symbolic names of atoms so that the actual answer set can be later recovered. Having recorded the symbols, the ground program is ready to be translated by LP2LP2 and LP2SAT. The outcome is a valid input for MINISAT which is used to compute one model or to show non-existence of models. The strong ranking constraints involved in the translation function $\mathrm{LP2LP}^{w*}(\cdot)$ can be activated by supplying option flag `-l` (local) and/or option flag `-g` (global) to the translator `lp2lp2`.

The goal of our experiment was to compare the performance of four systems based on the pipeline shown in Figure 2 with a number of reference systems—both a native ASP solver and other translation-based approaches for finding stable models:

1. A native state-of-the-art solver CLASP (v. 1.3.5) based on conflict-driven clause learning [8].
2. A solver CMODELS[9] (v. 3.79) [15] that uses loop formulas on top of Clark's completion in order to exclude non-stable supported models computed by SAT solvers.
3. The previous version of LP2SAT [11] with MINISAT (v. 1.14) as its back-end.

[9] `http://www.cs.utexas.edu/users/tag/cmodels.html`

4. The approach of translating SMODELS programs into difference logic [13] using LP2DIFF[10] (v. 1.26) and a state-of-the-art SMT solver Z3[11] (v. 2.11) as its back-end. We tried out strong ranking constraints but obtained best performance without.

As benchmarks, we used the original sets of problem instances from the Second Answer Set Programming Competition [4] which was run under the auspices of KU Leuven in Belgium. More precisely, we used the 516 instances submitted to the category of NP-complete problems. A summary of contributors can be found in [4]. Most of the ASP encodings used to solve the problems in question are due to the Potassco[12] team and published under the Asparagus site[13]. We used GRINGO to ground each problem instance accompanied by the respective encoding of the problem. In this way, each system is provided an identical ground program as its input. Moreover, we decided to run each solver using its default parameters.[14] As hardware we used Intel Q9550 Dual Core CPUs with 2.8 Ghz clock rate and 8 GB of main memory. The running time of each solver was limited to 600 seconds as in the second ASP competition. This limit applies to all processing steps from grounding to printing an answer set or declaring its non-existence. Moreover, we constrained the amount of available memory to 2.79 GBs in order to simulate the arrangements of the actual competition. The models computed by various systems were verified using SMODELS (v. 2.34) as a reference.

Table 2 collects the numbers of solved problem instances—both satisfiable and unsatisfiable ones. The approaches based on the translations presented in this paper scale very similarly. Strong ranking constraints improved the performance of MINISAT and, in particular, for satisfiable problem instances. The best performance was obtained using strong local ranking constraints (option `-l` of LP2LP2). The state-of-the-art ASP system CLASP outperforms these by roughly 40–65 solved instances within the given time and memory limits. This difference is mainly due to problems GraphPartitioning, KnightTour, WireRouting, and partly TravellingSalesperson. The improvement with respect to the old version, viz. LP2SAT [11] is also clear: 17–42 solved instances. We also tried other SAT solvers as the back-end of our translators but the overall performance slightly degraded in this way. For PRECOSAT, the numbers of solved instances were 419, 415 (option `-l`), 420 (option `-g`), and 410 (options `-l` and `-g`). Hence strong local ranking constraints slowed down PRECOSAT slightly. The respective numbers for CLASP were 398, 410, 417, and 409 that also indicate the positive effect of strong global ranking constraints. The numbers obtained for MINISAT2 were alike: 401, 407, 411, and 410 solved instances. Thus, using the latest version of MINISAT does not seem to improve performance in the computation of answer sets with our translators.

An account of running times is included in Table 3. These results parallel the ones obtained in terms of numbers of solved instances. The performance difference with respect to CLASP is emphasized by time spent on solved instances. Perhaps it should be noted that timeouts sanctioned by 40–65 unsolved instances amount to 6.7–10.8 hours.

[10] http://www.tcs.hut.fi/Software/lp2diff/
[11] http://research.microsoft.com/en-us/um/redmond/projects/z3/
[12] http://potassco.sourceforge.net/
[13] http://asp.haiti.cs.uni-potsdam.de/
[14] To the contrary, the rules of the ASP competition allowed benchmark-specific encodings as well as tuning the parameters of the solver in question to obtain the best possible performance.

**Table 3.** Total running times in hours

| System | CLASP | CMODELS | LP2DIFF | LP2SAT | LP2SAT2 | -l | -g | -lg |
|---|---|---|---|---|---|---|---|---|
| On solved instances | 1.5 | 4.0 | 5.6 | 3.7 | 4.8 | 5.6 | 5.1 | 5.2 |
| Timeouts | 7.7 | 21.5 | 23.0 | 21.5 | 18.7 | 14.5 | 14.8 | 15.3 |
| TOTAL | 9.2 | 25.5 | 28.6 | 25.2 | 23.5 | 20.1 | 19.9 | 20.5 |

## 5 Conclusions

SAT solver technology has been developing rapidly and provides a very promising computational platform for implementing also ASP systems. The goal of the paper is to develop a compact translation from ground programs with cardinality and weight constraints to clauses so that a SAT solver can be used unmodified for the task of computing stable models of SMODELS programs. Based on the results in [11,23,13] we present a compact translation from SMODELS programs to propositional clauses using normal programs subject to the supported model semantics [1] as an intermediary representation. This enables the direct exploitation of improving SAT solvers as stable model finders without any changes. The translation technique is novel as it exploits program transformations and ASP encoding techniques for achieving a compact and computationally efficient overall translation. Our preliminary experimental results indicate that this translation provides the most effective way of using current state-of-the-art SAT solvers for stable model computation with a performance which is surprisingly close to that of the top state-of-the-art ASP solver.

There are a number of interesting topics for future research including, for example, developing preprocessing techniques for optimizing the translation, investigating alternative ways of translating cardinality and weight constraints, and evaluating more systematically the performance of different SAT solver techniques for SAT instances resulting from the translation.

*Acknowledgments.* The authors would like to thank Martin Gebser for his insightful comments and observations about a draft of this paper. This research has been partially funded by the Academy of Finland under project #122399.

## References

1. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann, San Francisco (1988)
2. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. Annals of Mathematics and Artificial Intelligence 12(1-2), 53–87 (1994)
3. Clark, K.: Negation as failure. In: Logic and Data Bases, pp. 293–322. Plenum Press, New York (1978)
4. 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)

5. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2(1-4), 1–26 (2006)
6. Erdem, E., Lifschitz, V.: Tight logic programs. Theory and Practice of Logic Programming 3(4-5), 499–518 (2003)
7. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming 5(1-2), 45–74 (2005)
8. 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)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP 1988, pp. 1070–1080 (1988)
10. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36(4), 345–377 (2006)
11. Janhunen, T.: Representing normal programs with clauses. In: Proceedings of ECAI 2004, pp. 358–362 (2004)
12. 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)
13. 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)
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
15. Lierler, Y.: Cmodels – SAT-based disjunctive answer set solver. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 447–451. Springer, Heidelberg (2005)
16. Lifschitz, V.: Answer set planning. In: Proceedings of ICLP 1999, pp. 23–37 (1999)
17. Lifschitz, V., Tang, L.R., Turner, H.: Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence 25(3-4), 369–389 (1999)
18. Lin, F., Zhao, J.: On tight logic programs and yet another translation from normal logic programs to propositional logic. In: Proceedings of IJCAI 2003, pp. 853–858 (2003)
19. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence 157(1-2), 115–137 (2004)
20. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)
21. Marek, V.W., Subrahmanian, V.S.: The relationship between stable, supported, default and autoepistemic semantics for general logic programs. Theoretical Computer Science 103, 365–386 (1992)
22. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4), 241–273 (1999)
23. Niemelä, I.: Stable models and difference logic. Annals of Mathematics and Artificial Intelligence 53(1-4), 313–329 (2008)
24. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2), 181–234 (2002)
25. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Siekmann, J., Wrightson, G. (eds.) Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970, pp. 466–483. Springer, Heidelberg (1983)
26. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)

# Effectively Reasoning about Infinite Sets in Answer Set Programming*

Victor Marek[1] and Jeffrey B. Remmel[2]

[1] Department of Computer Science,
University of Kentucky, Lexington, KY 40506
marek@cs.uky.edu.edu

[2] Departments of Mathematics and Computer Science,
University of California at San Diego, La Jolla, CA 92903
jremmel@ucsd.edu

**Abstract.** In answer set programming (ASP), one does not allow the use of function symbols. Disallowing function symbols avoids the problem of having logic programs which have stable models of excessively high complexity. For example, Marek, Nerode, and Remmel showed that there exist finite predicate logic programs which have stable models but which have no hyperarithmetic stable model. Disallowing function symbols also avoids problems with the occurs check that can lead to wrong answers in logic programs. Of course, by eliminating function symbols, one loses a lot of expressive power in the language. In particular, it is difficult to directly reason about infinite sets in ASP.

Blair, Marek, and Remmel [BMR08] developed an extension of logic programming called *set based logic programming*. In the theory of set based logic programming, the atoms represent subsets of a fixed universe $X$ and one is allowed to compose the one-step consequence operator with a monotonic idempotent operator (miop) $O$ so as to ensure that the analogue of stable models are always closed under $O$. We let $\mathcal{S}_P$ denote the set of fixed points of finite unions of the sets represented by the atoms of $P$ under the miops associated with $P$. We shall show that if there is a coding scheme which associates to each element $A \in \mathcal{S}_P$ a code $c(A)$ such that there are effective procedures, which given two codes $c(A)$ and $c(B)$ of elements $A, B \in \mathcal{S}_P$, will (i) decide if $A \subseteq B$, (ii) decide if $A \cap B = \emptyset$, and (iii) produce the codes of the closures of $A \cup B$ and of $A \cap B$ under the miop operators associated with $P$, then we can effectively decide whether an element $A \in \mathcal{S}_P$ is a stable model of $P$. Thus in such a situation, we can effectively reason about the stable models of $P$ even though $\mathcal{S}_P$ contains infinite sets. Our basic example is the case where all the sets represented by the atoms of $P$ are regular languages but many other examples are possible such as when the sets involved are certain classes of convex sets in $\mathbb{R}^n$.

* This is an updated and expanded version of [MR09].

## 1 Introduction

Computer Science for the most part reasons about *finite* sets, relations and functions. There are many examples in computer science where adding symbols for infinite sets or arbitrary function symbols into programming languages results in big jumps in the complexity of models of programs. For example, finding the least model of a finite Horn program with no function symbols can be done in linear time [DG82] while the least model of finite predicate logic Horn program with function symbols can be an arbitrary recursively enumerable set [Sm68]. If we consider logic programs with negation, Marek and Truszczyński [MT93] showed that the question of whether a finite propositional logic program has a stable model is NP-complete. However Marek, Nerode, and Remmel [MNR94] showed that the question of whether a finite predicate logic program with function symbols possesses a stable model is $\Sigma_1^1$ complete. Similarly, the stable models of logic programs that contain function symbols can be quite complex. Starting with [AB90] and continuing with [BMS95] and [MNR94], a number of results showed that the stable models of logic programs that allow function symbols can be exceedingly complex, even in the case where the program has a unique stable model. For example, Apt and Blair [AB90] have shown that arithmetic sets can be defined with stable models of stratified programs and Marek, Nerode and Remmel [MNR94] showed that there exist finite predicate logic programs which have stable models but which have no hyperarithmetic stable model.

While these type of results may at first glance appear negative, they had a positive effect in the long run since they forced researchers and designers to limit themselves to cases where programs can be actually processed. The effect was that processing programs called *solvers* such as cmodels [BL02, GLM06], smodels [SNS02], *clasp* [GKN+], ASSAT [LZ02], and `dlv` [LPF+06] had to focus on finite programs that do not admit function symbols (`dlv` allows for use of a very limited class of programs with function symbols). The designers of solvers have also focused on the issues of both improving processing of the logic programs (i.e. searching for a stable model) and improving the use of logic programs as a programming language. The latter task consists of extending the constructs available to the programmer to make programming easier and more readable. This development resulted in a class of solvers that found use in combinatorial optimization [MT99, Nie99], hardware verification [KN03], product configuration [SNTS01], and other applications.

Of course, by eliminating function symbols, one loses a lot of expressive power in the language. One of the motivations of this paper was to find ways to extend the ASP formalism to allow one to reason *directly* about infinite sets yet still allow the programs to be processed in an effective manner. This requires a very careful analysis of the complexity issues involved in the formalisms as well as developing various ways to code the infinite sets involved in any given application so that one can process information effectively. Part of our motivation is that with the rise of the Internet, there are now many tools which use the Internet as a virtual data base. While all the information on the Internet at any given point of time is a finite object, it is constantly changing and it would be nearly impossible to characterize the totality of information available in any meaningful way. Thus, for all practical purposes, one can consider the information on the Internet as an infinite set of information items. Hence we need to consider ways in which one can extend various formalisms in computer science to reason about infinite objects.

The main goal of this paper is to show that there are extensions of the ASP formalism where one can effectively reason about infinite languages which are accepted by deterministic finite automata (DFAs). In particular, we shall show that in a recent extension of logic programming due to Blair, Marek, and Remmel [BMR08], one can effectively reason about languages which are accepted by finite automaton. We will also show that under reasonable assumptions, this approach can be lifted to other areas as well.

In [BMR01], Blair, Marek, and Remmel developed an extension of the logic programming paradigm called *spatial logic programming* in which one can directly reason about regions in space and time as might be required in applications like graphics, image compression, or job scheduling. In spatial logic programming, one has some fixed space $X$ be the intended universe of the program rather than having the Herbrand base be the intended underlying universe of the program and one has each atom of the language of the program specify a subset of $X$, i.e. an element of the set $2^X$.

As pointed out in [BMR08], if one reflects for a moment on the basic aspects of logic programming with an Herbrand model interpretation, a slight change in one's point of view shows that it is natural to interpret atoms as subsets of the Herbrand base. In ordinary logic programming, we determine the truth value of an atom $p$ in an Herbrand interpretation $I$ by declaring $I \models p$ if and only if $p \in I$. However, this is equivalent to defining the *sense*, $[\![p]\!]$, of an atom $p$ to be the set $\{p\}$ and declaring that $I \models p$ if and only if $[\![p]\!] \subseteq I$. By this simple move, we have permitted ourselves to interpret the sense of an atom as a subset of a set $X$ rather than the literal atom itself in the case where $X$ is the Herbrand base of the language of the program.

It turns out that if the underlying space $X$ has structure such as a topology or an algebraic structure such as a group or vector space, then a number of other natural options present themselves. That is, Blair, Marek, and Remmel [BMR08] extended the theory of spatial logic programming to what they called *set based logic programming* where one composes the one-step consequence operator of spatial logic programing with a monotonic idempotent operator. For example, if we are dealing with a topological space, one can construct a new one-step consequence operator $T$ by composing the one-step consequence operator for spatial logic programming with an operator that produces the topological closure of a set or the interior of a set. In such a situation, we can ensure that the new one-step consequence operator $T$ *always* produces a closed set or always produces an open set. Similarly, if the underlying space is a vector space, one might construct a new one-step consequence operator $T$ by composing the one-step consequence operator for spatial logic programming with the operator that produces the smallest subspace containing a set, the *span* operator, or with the operator that produces the smallest convex closed set containing a set, the *convex closure* operator. In this way, one can ensure that the new one-step consequence operator $T$ always produces a subspace or always produces a convex closed set. More generally, We say that an operator $O : 2^X \rightarrow 2^X$ is *monotonic* if for all $Y \subseteq Z \subseteq X$, we have $O(Y) \subseteq O(Z)$ and we say that $O$ is *idempotent* for all $Y \subseteq X$, $O(O(Y)) = O(Y)$. Specifically, many familiar operators such as *closure*, *interior*, or the *span* and *convex-closure* operators in vector spaces over the rationals and other fields are monotonic idempotent operators. We call a monotonic idempotent operator a *miop*. We say that a set $Y$ is *closed* with respect to miop $O$ if and only if $Y = O(Y)$. Besides of examples listed above, in the context or regular languages, we discuss a number of monotone idempotent operators in Example 3, Section 4. By composing the one-step consequence operator for spatial logic programs with the operator $O$, we can ensure that the resulting one-step consequence operator always produces a fixed point of $O$. We can then think of the operator $O$ as a parameter. This naturally leads us to a situation where we have a natural polymorphism for set based logic programming. That is, one can use the same logic program to produce stable models with different properties depending on how the operator $O$ is chosen.

Moreover, in such a setting, one also has a variety of options for how to interpret negation. In normal logic programming, a model $M$ satisfies $\neg p$ if $p \notin M$. From the spatial logic programming point of view, when $p$ is interpreted as a singleton $\{p\}$, this would be equivalent to saying that $M$ satisfies $\neg p$ if (i) $\{p\} \cap M = \emptyset$, or (equivalently) (ii) $\{p\} \not\subseteq M$. When the sense of $p$ is a set with more than one element, it is easy to see that saying that $M$ satisfies $\neg p$ if $[\![p]\!] \cap M = \emptyset$ (strong negation) is different from saying that $M$ satisfies $\neg p$ if $[\![p]\!] \not\subseteq M$ (weak negation). This leads to two natural interpretations of the negation symbol which are compatible with the basic logic programming paradigm. When the underlying space has a miop $cl$, one can get even more subsidiary types of negation by taking $M$ to satisfy $\neg p$ if $cl([\![p]\!]) \cap M \subseteq cl(\emptyset)$ (strong negation) or by taking $M$ to satisfy $\neg p$ if $cl([\![p]\!]) \not\subseteq M$ (weak negation).

Blair, Marek, and Remmel [BMR08] showed that set based logic programing provides the foundations and basic techniques for crafting applications in the *answer set paradigm* as described in [MT99, Nie99] and then [GL02, Ba03]. That is, if in a given application, topological, linear algebraic, or similar constructs are either necessary or at least natural, then one can construct an answer set programming paradigm whose models correspond to natural closed structures. The expressive power of miops allows us to capture functions and relations intrinsic to the domain of a spatial logic program, but independent of the program. This permits set based logic programs to seamlessly serve as front-ends to other systems. Miops play the role of back-end, or “behind-the-scenes”, procedures and functions.

We let $\mathcal{S}_P$ denote the set of least fixpoints with respect to the miops associated with $P$ containing all finite unions and intersections of sets represented by the atoms of a finite set based logic program $P$. Here the elements of $\mathcal{S}_P$ may be finite or infinite. The main goal of this paper is find conditions on $\mathcal{S}_P$ which ensure that we can effectively decide whether a given element of $\mathcal{S}_P$ is a stable model of $P$. We shall show that if there is a way of associating codes $c(A)$ to the elements of $A \in \mathcal{S}_P$ such that there are effective procedures which, given codes $c(A)$ and $c(B)$ for elements of $A, B \in \mathcal{S}_P$, will (i) decide if $A \subseteq B$, (ii) decide if $A \cap B = \emptyset$, and (iii) produce of the codes of closures of $A \cup B$ and $A \cap B$ under miop operators associated with $P$, then we can effectively decide whether a code $c(A)$ is the code of a stable model of $P$. Our running example will be the case where $P$ is a finite set-based logic program over a universe $X = \Sigma^*$ where $\Sigma$ is a finite alphabet and the sets represented by atoms in $P$ are languages contained in $X$ which are accepted by finite automaton and the miops $O$ involved in $P$ preserve regular languages, i.e, if $A$ is an automata such that the language $L(A)$ accepted by $A$ is contained in $X$, then we can effectively construct an automaton $B$ such that the language $L(B)$ accepted by $B$ equals $O(L(A))$. Then, we shall show that the stable models of $P$ are languages accepted by finite automaton and one can effectively check whether a language accepted by finite automaton is a stable model. Thus in this setting, one can effectively reason about an important class of infinite sets. However, it will be clear from our proofs that the only properties that we use for regular languages coded by their accepting DFAs are that the procedures for (i), (ii), and (iii) are effective.

The outline of this paper is as follows. In Section 2, we shall give the basic definitions of set based logic programming with miops. as developed by Blair, Marek, and Remmel [BMR08]. In Section 3, we shall review that basic properties of languages accepted by finite automata that we shall need. In Section 4, we shall show how the formalisms of finite automata can be seamlessly incorporated into the set based logic programming formalism. Finally, in Section 5, we give conclusions and directions for further research.

## 2 Set Logic Programs: Syntax, Miops, and Semantics

We review the basic definitions of set based logic programming as introduced by Blair, Marek, and Remmel [BMR08]. The syntax of set based logic programs will essentially be the syntax of DATALOG programs with negation.

Following [BMR08], we define a **set based augmented first-order language** (**set based language**, for short) $\mathcal{L}$ as a triple $(L, X, [\![\cdot]\!])$, where

(1) $L$ is a language for first-order predicate logic (without function symbols other than constants),
(2) $X$ is a nonempty (possibly infinite) set, called the **interpretation space**, and
(3) $[\![\cdot]\!]$ is a mapping from the atoms of $L$ to the power set of $X$, called the *sense assignment*. If $p$ is an atom, then $[\![p]\!]$ is called the *sense* of $p$.

In our setting, a **set based logic program** has three components.

1) The language $\mathcal{L}$ which includes the interpretation space and the sense assignment.

2) The IDB (**Intentional Database**): A finite set of program clauses, each of the form $A \leftarrow L_1, \ldots, L_n$, where each $L_i$ is a *literal*, i.e. an atom or the negation of an atom, and $A$ is an atom.
3) The EDB (**Extensional Database**): A finite set of clauses of the form $A \leftarrow$ where $A$ is an atom.

Given a set based logic program $P$, the *Herbrand base* of $P$ is the Herbrand base of the smallest set based language over which $P$ is a set based logic program.

We shall assume that the classes of set based logic programs that we consider are always over a language for first-order logic $L$ with no function symbols except constants, and a fixed set $X$. We let $\mathrm{HB}_L$ denote the Herbrand base of $L$, i.e. the set of atoms of $L$. We omit the subscript $L$ when the context is clear. Thus we allow clauses whose instances are of the following form:

$$\mathcal{C} = A \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m. \tag{1}$$

where $A$, $B_i$, and $C_j$ are atoms for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. We let $head(\mathcal{C}) = A$, $Body(\mathcal{C}) = B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m$, and $PosBody(\mathcal{C}) = \{B_1, \ldots, B_m\}$, and $NegBody(\mathcal{C}) = \{C_1, \ldots, C_m\}$.

We let $2^X$ be the powerset of $X$. Given $[\![\cdot]\!] : \mathrm{HB}_L \longrightarrow 2^X$, an *interpretation* $I$ of the set based language $\mathcal{L} = (L, X, [\![\cdot]\!])$ is a subset of $X$.

## 2.1 Examples of Monotonic Idempotent Operators

A second component of a set based logic program is one or more monotonic idempotent operators $O : 2^X \to 2^X$ that are associated with the program. Recall that an operator $O : 2^X \to 2^X$ is *monotonic* if for all $Y \subseteq Z \subseteq X$, we have $O(Y) \subseteq O(Z)$ and we say that $O$ is *idempotent* for all $Y \subseteq X$, $O(O(Y)) = O(Y)$. We call a monotonic idempotent operator a *miop* (pronounced "my op"). We say that a set $Y$ is *closed* with respect to miop $O$ if and only if $Y = O(Y)$.

For example, suppose that the interpretation space $X$ is either $\mathbf{R}^n$ or $\mathbf{Q}^n$ where $\mathbf{R}$ is the reals and $\mathbf{Q}$ is the rationals. Then, $X$ is a topological vector space under the usual topology so that we have a number of natural miop operators:

1. $op_{id}(A) = A$, i.e. the identity map is simplest miop operator,
2. $op_c(A) = \bar{A}$ where $\bar{A}$ is the smallest closed set containing $A$,
3. $op_{int}(A) = int(A)$ where $int(A)$ is the interior of $A$,
4. $op_{convex}(A) = K(A)$ where $K(A)$ is the convex closure of $A$, i.e. the smallest set $K \subseteq X$ such that $A \subseteq K$ and whenever $x_1, \ldots, x_n \in K$ and $\alpha_1, \ldots, \alpha_n$ are elements of the underlying field ($\mathbf{R}$ or $\mathbf{Q}$) such that $\sum_{i=1}^n \alpha_i = 1$, then $\sum_{i=1}^n \alpha_i x_i$ is in $K$, and
5. $op_{subsp}(A) = (A)^*$ where $(A)^*$ is the subspace of $X$ generated by $A$.

We should note that (5) is a prototypical example if we start with an *algebraic* structure. That is, in such cases, we can let $op_{substr}(A) = (A)^*$ where $(A)^*$ is the substructure of $X$ generated by $A$. Examples of such miops include the following:

**(a)** if $X$ is a group, we can let $op_{subgrp}(A) = (A)^*$ where $(A)^*$ is the subgroup of $X$ generated by $A$,

**(b)** if $X$ is a ring, we can let $op_{subrg}(A) = (A)^*$ where $(A)^*$ is the subring of $X$ generated by $A$,

**(c)** if $X$ is a field, we can let $op_{subfld}(A) = (A)^*$ where $(A)^*$ is the subfield of $X$ generated by $A$,

**(d)** if $X$ is a Boolean algebra, we can let $op_{subalg}(A) = (A)^*$ where $(A)^*$ is the subalgebra of $X$ generated by $A$ or we can let $op_{ideal}(A) = Id(A)$ where $Id(A)$ is the ideal of $X$ generated by $A$, and

**(e)** if $(X, \leq_X)$ is a partially ordered set, we can let $op_{uideal}(A) = Uid(A)$ where $Uid(A)$ is the upper order ideal of $X$ (that is, the least subset $S$ of $X$ containing $A$ such that whenever $x \in S$ and $x \leq_X y$, then $y \in S$).

## 2.2 Set Based Logic Programming with Miops

Now suppose that we are given a miop $op^+ : 2^X \to 2^X$ and Horn set based logic program $P$ over $X$. Here we say that a set based logic program is *Horn* if its IDB is Horn. Blair, Marek, and Remmel [BMR08] generalized the one-step consequence-operator of ordinary logic programs with respect to 2-valued logic to set based logic programs relative to a miop operator $op^+$ as follows. First, for any atom $A$ and $I \subseteq X$, we say that $I \models_{[\![\cdot]\!],op^+} A$ if and only if $op^+([\![A]\!]) \subseteq I$. Then, given a set based logic program $P$ with IDB $P$, let $P'$ be the set of instances of a clauses in $P$ and let

$$T_{P,op^+}(I) = op^+(I_1 \cup I_2)$$

where $I_1 = \bigcup\{[\![a]\!] \mid a \leftarrow L_1, \ldots, L_n \ \in P', I \models_{[\![\cdot]\!],op^+} L_i, i = 1, \ldots, n\}$ and $I_2 = \bigcup\{[\![a]\!] \mid a \leftarrow \ $ is an instance of a clause in the EDB of $P\}$.
We then say that a *supported model relative to $op^+$* of $P$ is a fixed point of $T_{P,op^+}$.

We iterate $T_{P,op^+}$ according to the following.

$$\begin{aligned} T_{P,op^+} \uparrow^0 (I) &= I \\ T_{P,op^+} \uparrow^{\alpha+1} (I) &= T_{P,op^+}(T_{P,op^+} \uparrow^\alpha (I)) \\ T_{P,op^+} \uparrow^\lambda (I) &= op^+(\bigcup_{\alpha<\lambda} \{T_{P,op^+} \uparrow^\alpha (I)\}),\ \lambda \text{ limit} \end{aligned}$$

It is easy to see that if $P$ is a Horn spatial logic program and $op^+$ is a miop, then $T_{P,op^+}$ is monotonic. Blair, Marek, and Remmel [BMR08] proved the following.

**Theorem 1.** Given a miop $op^+$, the least model of a Horn set based logic program $P$ exists and is closed under $op^+$ , is supported relative $op^+$, and is given by $\mathbf{T}_{P,op^+} \uparrow^\alpha (\emptyset)$ for the least ordinal $\alpha$ at which a fixed point is obtained.

We note, however, that if the Herbrand universe of a set based logic program is infinite (contains infinitely many constants) then, unlike the situation with ordinary Horn programs, $T_{P,op^+}$ will not in general be upward continuous even in the case where $op^+(A) = A$ for all $A \subseteq X$. That is, consider the following example which was given in [BMR08].

*Example 1.* Assume that $op^+$ is the identity operator on $2^X$. To specify a set based logic program, we must specify the language, EDB and IDB. Let $\mathcal{L} = (L, X, [\![\cdot]\!])$ where $L$ has four unary predicate symbols: $p$, $q$, $r$ and $s$, and countably many constants $e_0, e_1, \ldots,$. $X$ is the set $\mathbf{N} \bigcup \{\mathbf{N}\}$ where $\mathbf{N}$ is the set of natural numbers, $\{0, 1, 2, \ldots\}$. $[\![\cdot]\!]$ is specified by $[\![q(e_n)]\!] = \{0, \ldots, n\}$, $[\![p(e_n)]\!] = \{0, \ldots, n+1\}$, $[\![r(e_n)]\!] = \mathbf{N}$, and $[\![s(e_n)]\!] = \{\mathbf{N}\}$.

The EDB is $q(e_0) \leftarrow$ and the IDB is: $p(X) \leftarrow q(X)$ and $s(e_0) \leftarrow r(e_0)$.

Now, after $\omega$ iterations upward from the empty interpretation, $r(e_0)$ becomes satisfied. One more iteration is required to reach an interpretation that satisfies $s(e_0)$, where the least fixed point is attained. $\square$

Next we consider how we should deal with negation in the setting of miop operators. Suppose that we have a miop operator $op^-$ on the space $X$. We do not require that $op^-$ is the same as that miop $op^+$ but it may be. Our goal is to define two different satisfaction relations for negative literals relative to the miop operator $op^-$ which are called strong and weak negation in [BMR08] [1].

For the rest of this paper, we shall think of a set based logic program $P$ as a set of clauses of the form (1) where it may be that either $n$ or $m$ equals 0. We let $horn(P)$ denote the set of all Horn clauses in $P$ and $nohorn(P) = P \setminus horn(P)$.

**Definition 1.** Suppose that $P$ is a set based logic program over $X$ and $op^+$ and $op^-$ are miops on $X$ and $a \in \{s, w\}$.

$(I)$ Given any atom $A$ and set $J \subseteq X$, then we say
$J \models^a_{[\![\cdot]\!],op^+,op^-} A$ if and only if $op^+([\![A]\!]) \subseteq J$.

$(\mathrm{II})_s$ (Strong negation) Given any atom $A$ and set $J \subseteq X$, then we say
$J \models^s_{[\![\cdot]\!],op^+,op^-} \neg A$ if and only if $op^-([\![A]\!]) \cap J \subseteq op^-(\emptyset)$.

$(\mathrm{II})_w$ (Weak negation) Given any atom $A$ and set $J \subseteq X$, then we say
$J \models^w_{[\![\cdot]\!],op^+,op^-} \neg A$ if and only if $op^-([\![A]\!]) \nsubseteq J$.

**Definition 2.** For any given set $J \subseteq X$ we define the *strong Gelfond-Lifschitz transform*, $GL^s_{J,[\![\cdot]\!],op^+,op^-}(P)$, of a program $P$ with respect to miops $op^+$ and $op^-$ on $2^X$, in two steps. First, we consider all clauses in $P$,

$$\mathcal{C} = A \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, C_m \tag{2}$$

where $A, B_1, \ldots, B_n, C_1, \ldots, C_m$ are atoms. If for some $i$, it is *not* the case that $J \models^s_{[\![\cdot]\!],op^+,op^-} \neg C_i$, then we eliminate clause $\mathcal{C}$. Otherwise we replace $\mathcal{C}$ by the Horn clause

$$A \leftarrow B_1, \ldots, B_n. \tag{3}$$

Then, $GL^s_{J,[\![\cdot]\!],op^+,\mathcal{R}}(P)$ consists of the set of all Horn clauses produced by this two step process.

[1] Lifschitz [Li94] observed that different modalities, thus different operators, can be used to evaluate positive and negative part of bodies of clauses of normal programs.

We define the *weak Gelfond-Lifschitz transform*, $GL^w_{J,[\![\cdot]\!],op^+,op^-}(P)$, of a program $P$ with respect to miops $op^+$ and $op^-$ on $2^X$ in a similar manner except that we use $\models^w_{[\![\cdot]\!],op^+,op^-}$ in place of $\models^s_{[\![\cdot]\!],op^+,op^-}$ in the definition.

Note that since $GL^a_{J,[\![\cdot]\!],op^+,op^-}(P)$ is a Horn set based logic program for either $a = s$ or $a = w$, the least model of $GL^a_{J,[\![\cdot]\!],op^+,op^-}(P)$ relative to $op^+$ is defined. We then define the $a$-stable model semantics for a set based logic program $P$ over $X$ relative to the miops $op^+$ and $op^-$ on $X$ for $a \in \{s, w\}$ as follows.

**Definition 3.** $J$ is an *$a$-stable* model of $P$ *relative* to $op^+$ and $op^-$ if and only if $J$ is the least fixed point of $T_{GL^a_{J,[\![\cdot]\!],op^+,op^-}(P),op^+}$.

Next we give a simple example to show that there is a difference between $s$-stable and $w$-stable models.

*Example 2.* Suppose that the space $X = \mathcal{R}^2$ is the real plane. Our program will have two atoms $\{a, b\}, \{c, d\}$ where $a, b, c$ and $d$ are reals. We let $[a, b]$ and $[c, d]$ denote the line segments connecting $a$ to $b$ and $c$ to $d$ respectively. We let the sense of the these atoms be the corresponding subsets, i.e. we let $[\![\{a, b\}]\!] = \{a, b\}$ and $[\![\{c, d\}]\!] = \{c, d\}$. We let $op^+ = op^- = op_{convex}$. The consider the following program $\mathcal{P}$.

(1) $\{a, b\} \leftarrow \neg\{c, d\}$
(2) $\{c, d\} \leftarrow \neg\{a, b\}$

There are four possible candidate for stable models in this case, namely (i) $\emptyset$, (ii) $[a, b]$, (iii) $[c, d]$, and (iv) $op_{convex}\{a, b, c, d\}$. Let us recall that $op_{convex}(X)$ is the convex closure of $X$ which, depending on $a, b, c,$ and $d$ may be either a quadrilateral, triangle, or a line segment.

If we are considering $s$-stable models where $J \models^s_{[\![\cdot]\!],op^+,op^-} \neg C$ if and only if $op^-(C) \cap J = op^-(\emptyset) = \emptyset$, then the only case where there are $s$-stable models if $[a, b]$ and $[c, d]$ are disjoint in which (ii) case and (iii) are $s$-stable models.

If we are considering $w$-stable models where $J \models^w_{[\![\cdot]\!],op^+,op^-} \neg C$ if and only if $op^-(C) \not\subseteq J$, then there are no $w$-stable models if $[a, b] = [c, d]$, (ii) is a $w$-stable model if $[a, b] \not\subseteq [c, d]$, (iii) is $w$-stable model if $[c, d] \not\subseteq [a, b]$ and (ii) and (iii) are $w$-stable models if neither $[a, b] \subseteq [c, d]$ nor $[c, d] \subseteq [a, b]$. □

It is still the case that the $a$-stable models of a set based logic program $P$ form an antichain for $a \in \{s, w\}$. That is, we have the following result.

**Theorem 2.** *Suppose that $P$ is a set based logic program over $X$, $op^+$ and $op^-$ are miops on $X$, and $a \in \{s, w\}$. If $M$ and $N$ are $a$-stable models of $P$ and $M \subseteq N$, then $M = N$.*

**Proof.** It is easy to see that in general if $M \subseteq N$, then

$$GL^a_{N,[\![\cdot]\!],op^+,op^-}(P) \subseteq GL^a_{M,[\![\cdot]\!],op^+,op^-}(P).$$

Hence the least fixed point of $T_{GL^a_{N,[\![\cdot]\!],op^+,op^-}(P),op^+}$ is a subset of the least fixed point of $T_{GL^x_{M,[\![\cdot]\!],op^+,op^-}(P),op^+}$. But if $M \subseteq N$ and $M$ and $N$ are $a$-stable models, then $N$ equals the least fixed point of $T_{GL^a_{N,[\![\cdot]\!],op^+,op^-}(P),op^+}$ and $M$ equals the least fixed point of $T_{GL^a_{M,[\![\cdot]\!],op^+,op^-}(P),op^+}$ so that $N \subseteq M$. □

## 3 Languages Accepted by Finite Automaton

In this section, we shall briefly list some of the basic properties of languages accepted by finite automaton that we shall need.

Recall that a deterministic finite automaton (DFA) M is specified by a quintuple $M = (Q, \Sigma, \delta, s, F)$ where

$Q$ is a finite alphabet of state symbols,
$\Sigma$ is finite alphabet of input symbols,
$\delta : Q \times \Sigma \to Q$ is a transition function,
$s$ in $Q$ is the start state, and
$F \subseteq Q$ is the set of final (accepting) states.

We let $L(M)$ denote the set of all words $w$ accepted by $M$. A nondeterministic automaton (NFA) $M = (Q, \Sigma, \delta, s, F)$ is specified by similar 5-tuple except that in this case $\delta \subseteq Q \times \Sigma \times Q$. It is well known that for any fixed finite alphabet $\Sigma$, the set of languages $L \subseteq \Sigma^*$ accepted by DFAs and the set languages of $L \subseteq \Sigma^*$ accepted by NFA's are the same. Moreover, given any two DFAs $M_1$ and $M_2$, there are standard constructions of DFAs $M_3$, $M_4$, and $M_5$ such that

$L(M_3) = L(M_1) \cap L(M_2)$,
$L(M_4) = L(M_1) \cup L(M_2)$, and
$L(M_5) = \Sigma^* - L(M_1)$.

We shall denote these three DFAs by $M_3 = M_1 \cap M_2$, $M_4 = M_1 \cup M_2$, and $M_5 = \bar{M}_1$. We notice that in such setting the automaton accepting the language $L$ is a *code* for $L$. It is a well-known fact that instead of DFA one can consider a different class of codes for regular languages, namely regular expressions.

A crucial property of DFAs is the pumping lemma of [BPS61].

**Lemma 1.** *Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $p = |Q|$. Then for all words $w \in L(M)$ such that $|w| \geq p$, we can write $w = xyz$ for some $x, y, z \in \Sigma^*$ such that*

1. *$|xy| \leq p$,*
2. *$|y| \geq 1$, and*
3. *$xy^iz \in L(M)$ for all $i \geq 0$.*

One immediate consequence of the pumping lemma is that we can effectively decide whether $L(M)$ is empty or finite. That is, we have the following lemmas.

**Lemma 2.** *Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA. Then, $L(M)$ is empty if and only if for every $w \in \Sigma^*$ such that $|w| < |Q|$, $w$ is not accepted by $L(M)$.*

**Lemma 3.** *Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA. Then, $L(M)$ is finite if and only if for every $w \in \Sigma^*$ such that $|Q| \leq |w| < 2|Q|$, $w$ is not accepted by $L(M)$.*

Thus the complexity of the decision procedure to decide whether $L(M)$ is empty or finite depends directly on $|Q|$ and $|\Sigma|$. The fact that we can effectively decide if $L(M) = \emptyset$ also means that we can decide for any given DFAs $M_1$ and $M_2$ whether

1. $L(M_1) \subseteq L(M_2)$ since $L(M_1) \subseteq L(M_2)$ if and only if $L(M_1 \cap \bar{M_2}) = \emptyset$,
2. $L(M_1) = L(M_2)$ since $L(M_1) = L(M_2)$ if and only if $L(M_1) \subseteq L(M_2)$ and $L(M_2) \subseteq L(M_1)$, and
3. $L(M_1) \cap L(M_2) = \emptyset$ since $L(M_1) \cap L(M_2) = \emptyset$ if and only if $L(M_1 \cap M_2) = \emptyset$.

## 4 Set Based Logic Programming with Automata

In this section, we shall consider finite set based logic programs $P$ over $\mathcal{L} = (L, X, [\![\cdot]\!])$ where $X = \Sigma^*$ for some finite alphabet $\Sigma$. Thus $P$ consists of clauses of the form

$$\mathcal{C} = A \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, C_m \tag{4}$$

where $A, B_1, \ldots, B_n, C_1, \ldots, C_m$ are atoms. We shall assume that $X = \Sigma^*$ for some finite alphabet $\Sigma$ and that for any clause of the form (4) in $P$,

$$[\![A]\!], [\![B_1]\!], \ldots, [\![B_n]\!], [\![C_1]\!], \ldots, [\![C_m]\!]$$

are all accepted by DFAs whose alphabet of symbols is $\Sigma$. **For the moment, assume also that $op^+$ and $op^-$ are the identity operators.** For ease of notation, we shall assume that for any atom $A$ that appears in $P$, $A$ is a DFA whose over the alphabet $\Sigma$ and that $[\![A]\!] = L(A)$.

**Proposition 1.** *For every finite set based program $P$ where $op^+ = op_{id}$, every weak or strong stable model of $P$ is a finite union of the sense assignments of the heads of clauses in $P$.*

Thus any weak or strong stable model of $P$ must be a finite union of languages in $\Sigma^*$ which are accepted by DFAs and, hence, the stable model itself is accepted by a DFA since languages accepted by DFAs are closed under union. We claim that if $M$ is a DFA whose alphabet of symbols is $\Sigma$, then we can effectively decide whether $L(M)$ is a weak or strong stable model of $P$.

The first thing to observe is that we can effectively find the weak or strong Gelfond-Lifschitz transform of $P$. That is, under our assumptions for any atom $A$ and any $a \in \{s, w\}$,

1. $L(M) \models^a_{[\![\cdot]\!], op^+, op^-} A$ if and only if $L(A) \subseteq L(M)$,
2. $L(M) \models^s_{[\![\cdot]\!], op^+, op^-} \neg A$ if and only if $L(A) \cap L(M) = \emptyset$, and
3. $L(M) \models^w_{[\![\cdot]\!], op^+, op^-} \neg A$ if and only if $L(A) \not\subseteq L(M)$.

It follows from the results in Section 3, that we can effectively decide whether $L(M) \models^a_{[\![\cdot]\!],op^+,op^-} A$, $L(M) \models^s_{[\![\cdot]\!],op^+,op^-} \neg A$, and $L(M) \models^w_{[\![\cdot]\!],op^+,op^-} \neg A$. Hence, we can effectively construct $GL^s_{L(M),[\![\cdot]\!],op^+,op^-}(P)$ and $GL^w_{L(M),[\![\cdot]\!],op^+,op^-}(P)$.

Now suppose that $Q$ is a finite Horn set based logic program over $\mathcal{L} = (L, X, [\![\cdot]\!])$ where $X = \Sigma^*$ for some finite alphabet $\Sigma$ and $op^+$ and $op^-$ are the identity operators. Moreover, assume that for any atom $A$ which appears in $Q$, $[\![A]\!]$ is a language accepted by a DFA whose alphabet is $\Sigma$. Again, for ease of notation, we shall assume that for any atom $A$ that appears in $P$, $A$ is a DFA whose alphabet is $\Sigma$ and that $[\![A]\!] = L(A)$. Then, we claim that we can effectively construct a DFA $M$ such that $L(M)$ is the least model of $Q$. First, we shall show that for all $n \geq 1$, we can effectively construct a DFA $M^n$ such $T^n_{Q,op^+}(\emptyset) = L(M^n)$. Note that $T_{Q,op^+}(\emptyset)$ is equal to $\bigcup\{L(A) : A \leftarrow \ \in Q\}$. Now if $\{A \leftarrow \ \in Q\}$ is empty, then $T_{Q,op^+}(\emptyset) = \emptyset$ and the least model of $Q$ equals $\emptyset$ so that we simply let $M^1$ be the one state DFA which has no accepting state. Otherwise, suppose

$$\{A : A \leftarrow \ \in Q\} = \{A^0_1, \ldots, A^0_{n_0}\}.$$

Then, we set $M^1 = A^0_1 \cup \cdots \cup A^0_{n_0}$. Now assume that we have constructed a DFA $M^n$ such that $T^n_{Q,op^+}(\emptyset) = L(M^n)$. Then,

$$T_{Q,op^+}(L(M^n)) = op^+(I_1 \cup I_2)$$

where $I_1 = \bigcup\{[\![A]\!] \mid A \leftarrow B_1, \ldots, B_m \ \in Q, L(M^n) \models_{[\![\cdot]\!],op^+} B_i, i = 1, \ldots, n\}$ and $I_2 = \bigcup\{[\![A]\!] \mid A \leftarrow \ \text{is a clause in the EDB of } Q\}$.

Note that $I_1 \cup I_2$ is finite since $Q$ is finite. Since we can effectively decide whether $L(N) \subseteq L(M^n)$ for any DFA $N$, we can effectively decide whether $L(M^n) \models_{[\![\cdot]\!],op^+} B_i$ for any atom $B_i$ and hence we can effectively compute $I_1$ and $I_2$. Then we simply let $L(M^{n+1})$ be the DFA whose language is the union of all the $L(A)$ such that $A \in I_1 \cup I_2$.

Finally, we can effectively check whether $L(M^{n+1}) = L(M^n)$. Since the least model of $Q$ equals $L(M^n)$ where $n$ is the least integer such that $L(M^{n+1}) = L(M^n)$, we can effectively construct a DFA $R$ such that $L(R)$ is the least model of $Q$.

It follows that we can effectively construct DFAs $M_s$ and $M_w$ such that $L(M_s)$ is the least model of $GL^s_{L(M),[\![\cdot]\!],op^+,op^-}(P)$ and $L(M_w)$ is the least model of $GL^w_{L(M),[\![\cdot]\!],op^+,op^-}(P)$. Since we can effectively check whether $L(M) = L(M_s)$ and whether $L(M) = L(M_w)$, it follows that we can effectively decide if $L(M)$ is a weak or strong stable model of $P$.

We can extend our analysis to finite set based logic programs $P$ with miops assuming that the miops for $P$ satisfy the following property.

**Definition 4.** We say that a miop $op : 2^{\Sigma^*} \rightarrow 2^{\Sigma^*}$ is *effectively automata preserving* if for any DFA $M$ whose underlying alphabet of symbols is $\Sigma$, we can effectively construct a DFA $N$ whose underlying alphabet of symbols is $\Sigma$ such that $L(N) = op(L(M))$.

We will now give a number of examples of miops on regular languages.

*Example 3.* Suppose that $\Sigma = \{0, 1, \ldots, m\}$. Then, the following are effectively automata preserving operators.

1. If $N$ is a DFA whose underlying set of symbols is $\Sigma$, then we can define $op : 2^{\Sigma^*} \to 2^{\Sigma^*}$ by setting $op(S) = S \cup L(N)$ for any $S \subseteq \Sigma^*$. Clearly if $S = L(M)$ for some DFA $M$ whose underlying set of symbols is $\Sigma$, then $op(L(M)) = L(M \cup N)$ so $op$ is effectively automaton preserving.
2. If $N$ is a DFA whose underlying set of symbols is $\Sigma$, then we can define $op : 2^{\Sigma^*} \to 2^{\Sigma^*}$ by setting $op(S) = S \cap L(N)$ for any $S \subseteq \Sigma^*$. Clearly if $S = L(M)$ for some DFA $M$ whose underlying set of symbols is $\Sigma$, then $op(L(M) = L(M \cap N)$ so $op$ is effectively automata preserving.
3. If $T$ is any subset of $\Sigma$, we can let $op(S) = S(T^*)$. Again $op$ will be an effectively automata preserving miop since if $M$ is DFA whose underlying set of symbols is $\Sigma$, then let $N$ be NFA constructed from $M$ by adding loops on all the accepting states labeled with letters from $T$. It is easy to see that $N$ accepts $L(M)T^*$ and then one can use the standard construction to find a DFA $N'$ such that $L(N') = L(N)$. Note that in the special case where $T$ equals $\Sigma$, we can think of $op$ as constructing the upper ideal of $S$ in $\Sigma^*$ relative to the partial order $\sqsubseteq$ where for any words $u, v \in \Sigma^*$, $u \sqsubseteq v$ if and only if $u$ is prefix of $v$, i.e. $v$ is of the form $uw$ for some $w \in \Sigma^*$. For any poset $(P, \leq_P)$, we say that a set $U \subseteq P$ is an *upper ideal* in $P$, if whenever $x \leq_P y$ and $x \in P$, then $y \in P$. Clearly, for the poset $(\Sigma^*, \sqsubseteq)$, $op(S)$ is the upper ideal of $(\Sigma^*, \sqsubseteq)$ generated by $S$.
4. Let $\mathcal{P} = (\Sigma, \leq)$ be a partially-ordered set. For any $w, w' \in \Sigma^*$, we say that $w'$ is a factor of $w$ if there are words $u, v \in \Sigma^*$ with $w = uw'v$. Define the *generalized factor order* on $P^*$ by letting $u \leq w$ if there is a factor $w'$ of $w$ having the same length as $u$ such that $u \leq w'$, where the comparison of $u$ and $w'$ is done componentwise using the partial order in $\mathcal{P}$. Again we can show that if $op(S)$ is the upper ideal generated by $S$ the generalized factor order relative to $P^*$, then $op$ is an effectively automata preserving miop. That is, if we start with a DFA $M = (Q, \Sigma, \delta, s, F)$, then we can modify $M$ to an NFA that accepts $op(L(M))$ as follows. Think of $M$ as a digraph with edges labeled by elements of $\Sigma$ in the usual manner. First, we add a new start state $s_0$. There are loops from $s_0$ labeled with all letters in $\Sigma$. There is also a $\lambda$-transition from $s_0$ to the old start state $s$. We then modify the transitions in $M$ so that if there is an edge from state $q$ to $q'$ labeled with symbol $r$, then we add an edge from $q$ to $q'$ with any symbol $s$ such that $r \leq s$. Finally we add loops to all accepting states such that labeled with all letters in in $\Sigma$.
5. If we allow multiple representations of the infinite dimensional vector space $V_\infty$ for the field $GF_q$ where $q$ is prime, then the operator $op_{subsp}$ can be thought of an automaton preserving miop. Let $\Sigma = \{0, \ldots, q-1\}$. The standard way to represent the elements of $V_\infty$ is to let $\mathbf{0} = 0$ and think of a non-zero element of $V_\infty$ as a finite sequence $\sigma_1 \ldots \sigma_n$ where $\sigma_n \neq 0$. The operations of scalar multiplication and addition are then performed

componentwise. In our case, we will let any element $\sigma \in V_\infty$ have multiple representations, namely, $\sigma$ can be represented by $\sigma 0^n$ for any $n \geq 0$. Then, we let $op_{subsp}(S)$ be the set of all representatives of the subspace of $V_\infty$ generated by $S$. In what follows, we shall only describe how to construct NFA's that accept the desired languages since the Myhill-Nerode Theorem allows us to construct in a uniform manner, for any NFA $M$, a DFA $D$ such that $L(M) = L(D)$. First, consider miop $op_1$ such that $op_1(S)$ is the set of all representations of elements of $S$. If $M$ is a DFA whose underlying alphabet is $\Sigma$, then we can modify $M$ to an NFA $N$ that accepts $op_1(S)$ as follows. First, any state $q$ such that there is an $n$ such that the word $0^n$ starting at state $q$ ends in an accepting state is an accepting state of $N$. In particular, every accepting state of $M$ is an accepting state of $N$. In addition, we add loops labeled with 0 to all the accepting states of $N$.

Next we let $op_2(S)$ denote the set of all representations of any element which is a scalar multiple of an element of $S$. We claim $op_2$ is also an automaton preserving miop. That is, if $M$ is a DFA whose underlying alphabet is $\Sigma$, then we can modify $M$ to an NFA $\bar{N}$ that accepts $op_2(S)$ as follows. First, let $N$ be the NFA such that $op_1(L(M)) = L(N)$. The for each $a \in \{0, \dots, q-1\}$, let $aN$ be the NFA that is constructed from $N$ by replacing each edge labeled with the letter $x$ by an edge labeled $ax$. Then, it is clear that $L(aN) = \{(a\sigma_1)\dots(a\sigma_n) : \sigma_1 \dots \sigma_n \in L(N)\}$ so that $op_2(L(M)) = L(\bar{N})$ where $\bar{N} = 0N \cup 1N \cup \dots \cup (q-1)N$.

Finally for any $a, b \in \{0, \dots, q-1\}$, we let $op_{a,b}(S)$ denote the set of all representatives of the form $a\sigma + b\tau$ such that $\sigma, \tau$ are in $S$ and $|\sigma| = |\tau|$. $op_{a,b}$ is not a miop, but nevertheless for any DFA $M$, we can construct an NFA $R_{a,b}$ such that $L(R_{a,b}) = op_{a,b}(L(M))$. First, let $N = (Q, \Sigma, \delta, s, F)$ be the DFA such that $L(N) = op_2(L(M))$. Then, the set of states of $R_{a,b}$ will be $Q \times Q$, $(s, s)$ will be the start state of $R_{a,b}$, and $F \times F$ will be the set of final states of $R_{a,b}$. Now suppose that there are edges from $p_0$ to $p_1$ labeled with $\alpha$ and from $q_0$ to $q_1$ labeled with $\beta$ in $N$. Then, we will have an edge in $R_{a,b}$ from $(p_0, q_0)$ to $(p_1, q_1)$ labeled with $a\alpha + b\beta$. It is easy to see that $L(R_{a,b}) = op_{a,b}(L(M))$. and hence if we let $R$ be the DFA such that $R = \bigcup_{(a,b) \in \Sigma \times \Sigma} R_{a,b}$, then $S \subseteq L(R) \subseteq op_{subsp}(S)$ and $L(R)$ has the property that if $s_1, s_2 \in S$, then $as_1 + bs_2 \in L(R)$ for any $a, b \in GF_q$. By a similar argument, we can construct for any finite sequence of distinct elements $a_1, \dots, a_r$ from $GF_q$, a DFA $U_{a_1,\dots,a_r}$ such that $L(U_{a_1,\dots,a_r})$ equals the set of all $a_1 t_1 + \dots + a_r t_r$ such that $t_1, \dots t_r \in L(R)$. It then follows that $op_{subsp}(S)$ equal the union of $L(U_{a_1,\dots,a_r})$ over all possible finite sequence of distinct elements from $GF_q$ and hence is we can construct a DFA $U$ which accepts $op_{subsp}(S)$. $\square$

It is then easy to check that if $op^+ : 2^{\Sigma^*} \to 2^{\Sigma^*}$, then for any Horn set based logic program $Q$ with the properties described above, we can construct a DFA $M^n$ such that $T^n_{Q,op^+}(\emptyset) = L(M^n)$ and, hence, we can effectively construct the least model of $Q$. Thus we have the following result.

**Theorem 3.** *Suppose that $P$ is a finite set based logic program over $\mathcal{L} = (L, X, [\![\cdot]\!])$ where $X = \Sigma^*$ for some finite alphabet $\Sigma$ and $op^+ : 2^{\Sigma^*} \to 2^{\Sigma^*}$ and $op^- : 2^{\Sigma^*} \to 2^{\Sigma^*}$ are effectively automaton preserving miops. Moreover, assume that for any atom $A$ which appears in $Q$, $[\![A]\!]$ is a language accepted by a DFA whose underlying set of symbols is $\Sigma$. Then:*

1. *Every weak (strong) stable model of $P$ is a language accepted by a DFA.*
2. *For any DFA $M$ whose underlying set of symbols is $\Sigma$, we can effectively decide whether $L(M)$ is a weak or strong stable model of $P$.*

Note that under the assumptions of Theorem 3, there are only finitely many possible strong or weak stable models the program $P$, namely a union of the sense of the head of certain clauses, and these are all recognizable by DFAs. Hence it is decidable whether such a set based logic program has a weak or strong stable model and there is an algorithm to find all such weak or strong stable models.

## 5 Conclusions

We showed in Theorem 3 that if the senses of the atoms of a finite set based logic program $P$ are all regular languages over some fixed finite alphabet and the miops involved are all automaton preserving miops, then we can effectively decide if $P$ has weak or strong stable model and there is an algorithm to find all weak and strong stable models. In fact, it is not difficult to see that all the operations in searching for either a weak or strong stable model of such programs are effective so that it is possible to extend existing search engines to produce either weak or strong stable models of such programs. However, we suspect that the problem of how to optimize such extensions of existing search engines will be an interesting and challenging research problem. Finite automaton are useful for carrying out a lot of recognition tasks such as search for keywords or ensuring documents or strings have a proper form so that our results show that we can add ASP programming on top of such recognition tasks.

If we examine the proof of Theorem 3, it is clear that we used DFAs as codes for set of regular languages $\mathcal{S}_P$ that arise by taking the closures under $op^+$ and $op^-$ of finite unions of the languages associated with atoms of $P$. Here we consider the empty set as the empty union so that the emptyset is in $\mathcal{S}_P$. Then the only properties of such regular languages that were necessary to prove Theorem 3 was that we have effective procedures which, given codes for $A, B \in mathcalS_P$, (i) decide if $A \subseteq B$, (ii) decide $A \cap B = \emptyset$, and (iii) produce the codes of $op^+(A \cup B)$, $op^+(A \cap B)$, $op^-(A \cup B)$ and $op^+(A \cup B)$.

For any finite set based logic program $P$, we let $\mathcal{S}_P$ denote set of fix points of all finite unions of sets represented by the atoms of a finite set based logic program $P$ of the miops associated with $P$. If we can associate a code $c(A)$ to each elements of $A \in \mathcal{S}_P$ such that there are effective procedures which, given codes $c(A)$ and $c(B)$ for elements of $A, B \in \mathcal{S}_P$, will (i) decide if $A \subseteq B$, (ii) decide if $A \cap B = \emptyset$, and (iii) produce of the codes of closures of $A \cup B$ and

$A \cap B$ under miop operators associated with $P$, then we can prove the analogue of Theorem 3 for $P$. We have shown that the case where the code of an atom $A$ is a DFA which accepts $A$ (alernatively a regular expression describing $A$) then we have such procedures. However, such codes and procedures are available in many other cases. For example, if all sets involved are the convex closures of a finite set of points in $\mathbb{R}^n$ and $op^+ = op_{convex}$ and $op^- = op_{id}$ or if all sets involved are finite dimensional vector spaces over a computable fiel and $op^+ = op^- = op_{subsp}$, then such codes and procedures as described above can be constructed.

**Acknowledgments.** The first author was partially supported by NASA-JLP contract 1401954. The second author has been partially supported by NSF grant DMS 0654060.

## References

[AB90] Apt, K., Blair, H.A.: Arithmetic Classification of Perfect Models of Stratified Programs. Fundamenta Informaticae 13, 1–17 (1990)

[BL02] Babovich, Y., Lifschitz, V.: Cmodels (2002), http://www.cs.utexas.edu/users/tag/cmodels.html

[Ba03] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)

[BPS61] Bar-Hillel, Y., Perles, M.A., Shamir, E.: On formal properties of simple phrase structure grammars. Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung 14, 143–172 (1961)

[BMR01] Blair, H.A., Marek, V.W., Remmel, J.B.: Spatial Logic Programming. In: Proceedings SCI 2001, Orlando, FL (July 2001)

[BMR08] Blair, H.A., Marek, V.W., Remmel, J.B.: Set Based Logic Programming. Annals of Mathematics and Artificial Intelligence 52, 81–105 (2008)

[BMS95] Blair, H.A., Marek, V.W., Schlipf, J.S.: The Expressivness of Locally Stratified Programs. Annals of Mathematics and Artificial Intelligence 15, 209–229 (1995)

[BG02] Blumensath, A., Grädel, E.: Automatic Structures. In: Proceedings of the 15th Symposium on Logic in Computer Science, LICS 2000, pp. 51–62 (2000)

[Den00] Denecker, M.: Extending classical logic with inductive definitions. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 703–717. Springer, Heidelberg (2000)

[DG82] Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing satisfiability of propositional Horn formulae. Journal of Logic Programming 3, 267–284 (1984)

[GKN+] 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)

[KN03] Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming 3, 519–550 (2003)

[GL02] Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – A-Prolog perspective. Artificial Intelligence Journal 138, 3–38 (2002)

[GL88] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the International Joint Conference and Symposium on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)

[GLM06] Giunchiglia, E., Lierer, Y., Maratea, M.: Answer Set Programming Based on Propositional Satisfiability. Journal of Automated Reasoning 36, 345–377 (2006)

[KN95] Khoussainov, B., Nerode, A.: Automatic Presentations of Structures. In: Leivant, D. (ed.) LCC 1994. LNCS, vol. 960, pp. 367–392. Springer, Heidelberg (1995)

[KNRS07] Khoussainov, B., Nies, A., Rubin, S., Stephan, F.: Automatic structures: richness and limitations. Logical Methods of Computer Science 3(2), 18(electronic) (2007)

[LPF+06] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7, 499–562 (2006)

[Li94] Lifschitz, V.: Minimal belief and negation as failure. Artificial Intelligence Journal 70, 53–72 (1994)

[LZ02] Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. In: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI 2002), pp. 112–117. AAAI Press, Menlo Park (2002)

[MNR94] Marek, W., Nerode, A., Remmel, J.B.: The stable models of predicate logic programs. Journal of Logic Programming 21(3), 129–154 (1994)

[MR09] Marek, V.W., Remmel, J.B.: Automata and Answer Set Programming. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 323–337. Springer, Heidelberg (2008)

[MT93] Marek, W., Truszczyński, M.: Nonmonotonic Logic – Context-Dependent Reasoning. Springer, Heidelberg (1993)

[MT99] Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm. AI, pp. 375–398. Springer, Heidelberg (1999)

[Nie99] Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)

[SNS02] Simons, P., Niemelä, I., Soininen, T.: Extending and implementing stable semantics of logic programs. Artificial Intelligence Journal 138, 181–234 (2002)

[SNTS01] Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R.: Representing Configuration Knowledge with Weight Constraint Rules. In: Answer Set Programming 2001 (2001)

[Sm68] Smullyan, R.: First-order Logic. Springer, Heidelberg (1968)

# Inspecting Side-Effects of Abduction in Logic Programs

Luís Moniz Pereira and Alexandre Miguel Pinto

Centro de Inteligência Artificial (CENTRIA)
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{lmp,amp}@di.fct.unl.pt

**Abstract.** In the context of abduction in Logic Programs, when finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, that is without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. We show how this type of reasoning requires a new mechanism, not provided by others already available. To achieve it, we present the concept of Inspection Point in Abductive Logic Programs, and show, by means of examples, how one can employ it to investigate side-effects of interest (the *inspection points*) in order to help choose among abductive solutions. We show how to implement inspection points on top of already existing abduction solving systems — ABDUAL and XSB-XASP — in a way that can be adopted by other systems too.

**Keywords:** Logic Programs, Abduction, Side-Effects.

## 1 Introduction

Abductive logic programming offers a formalism to declaratively express and solve problems in areas such as decision-making, diagnosis, planning, belief revision and hypothetical reasoning.

When finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, i.e. without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. We show how this type of reasoning requires a new abduction mechanism, that of *Inspection Points* (IPs).

Electing a specific abducible occurrence as an inspection point can be afforded by using an intentional abduction device, for convenience dubbed “meta-abduction" or “conditional abduction”; that is, in lieu of abducing that occurrence, one instead (meta-) abduces just the *intent* to simply check that the abducible’s actual abduction occurs somewhere in the abductive solution, by virtue of some other occurrence of it. Consequently, as we shall see, inspecting the side-effects of abduction is achievable by using abduction itself.

We begin by presenting the motivation, plus some background notation and definitions follow. Then issues of reasoning with logic programs are addressed in section 2, in particular, we take a look at abductive reasoning and the nature of backward and forward chaining and their relationship to query answering in an abductive framework. In section 3 we introduce inspection points, illustrate their need and their use with examples, and provide a declarative semantics. In section 4 we describe in detail our implementation of inspection points and illustrate its workings with an example. We close with conclusions, comparisons, and future work.

## 1.1 Motivation

Often, besides needing to abductively discover which hypotheses to assume in order to satisfy some condition, we may also want to know some of the side-effects of those assumptions; in fact, this is rather a rational thing to do. But, most of the time, we do not wish to know *all* possible side-effects of our assumptions, as some of them may be irrelevant to our concern, e.g. in decision-making. Likewise, the side-effects inherent in abductive explanations might not all be of interest, e.g. in model-based fault-diagnosis. Another common application of abductive reasoning is that of finding which actions to perform, action names being coded as abducibles; again, only some of an action's side-effects may be of interest. A simple example will help bring out the abduction side-effect issue and our approach to it.

*Example 1.* **Relevant and irrelevant side-effects.** Consider this logic program where $drink\_water$ and $drink\_beer$ are abducibles. Suppose we want to satisfy the Integrity Constraint (IC), and also to check if we get drunk or not. However, we do not care about the glass becoming wet — that being completely irrelevant to our current concern. Thus, in general, computation of whole models can be a waste of time since we are normally only interested, as for side-effects, in some subset of the program's literals.

$$\begin{array}{ll}
\leftarrow thirsty, not\ drink. & \text{\% this is an Integrity Constraint} \\
wet\_glass \leftarrow use\_glass. & use\_glass \leftarrow drink. \\
drink \leftarrow drink\_water. & drink \leftarrow drink\_beer. \\
thirsty. & drunk \leftarrow drink\_beer. \\
unsafe\_drive \leftarrow drunk. &
\end{array}$$

Moreover, in this example, we may wish to decide a possible action (whether to drive or not) only **after** we know which side-effects are true. In such cases, we do not want simply to introduce an extra IC expressed as $\leftarrow not\ unsafe\_drive$, because that would always impose abducing $not\ drink\_beer$, irrespective of whether we are not even considering to drive. We want to allow all possible abductive solutions for the single IC $\leftarrow thirsty, not\ drink$ and only then check for the side-effects of each solution, in order to then decide the driving action.

What we need is an inspection mechanism that permits checking the truth value of given side-effect literals (like $drunk$) as a consequence of abductions made to satisfy a given query and the program's ICs, but without further abducing whilst checking. This is achieved simply via our $inspect/1$ meta-predicate, by introducing instead the extra IC $\leftarrow inspect(not\ unsafe\_drive)$, rather than just $\leftarrow not\ unsafe\_drive$. The so-formulated (passive) IC is not allowed to be met by actively introducing abductions to that effect, but only by consuming abductions introduced to satisfy the query and other (active) ICs, like $\leftarrow thirsty, not\ drink$.

### 1.2 Background Notation and Definitions

**Definition 1.** ***Logic Rule.*** *A Logic Rule has the general form*
$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$
*where $H$ is an atom, and the $B_i$ and $C_j$ are atoms.*

$H$ is the head of the rule, and $B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ is its body, where any rule variables are deemed universally quantified. Throughout, we use '$not$ ' to denote default negation. When the body is empty, we say its head is a fact and write the rule just as $H$. If the head is empty, the rule is said to be an Integrity Constraint (IC). The atoms $true$ and $false$ are by definition respectively true and false in every interpretation.

**Definition 2.** ***Logic Program.*** *A Logic Program (LP for short) P is a (possibly infinite) set of Logic Rules, where non-ground rules stand for all their ground instances.*

In this paper, we consider solely so-called Normal LPs (NLPs), those whose heads of rules are positive literals, i.e. positive atoms, or empty, as per the above definition of rules. We focus furthermore on abductive logic programs, i.e. NLPs allowing for *abducibles*—user-specified positive literals without rules, whose truth-value is **not** assumed initially. Abducible instances or their default negations may appear in bodies of rules, like any other literal. They stand for hypotheses, each of which may independently be assumed true, in positive literal or default negation form, as the case may be, in order to produce an abductive solution to a query.

**Definition 3.** ***Abductive Solution.*** *An abductive solution is a consistent set of abducible instances or their negations that, when replaced by $true$ everywhere in P, or equivalently simply omitted, affords a (Herbrand) model of P that satisfies the query and (of course) the ICs—a so-called abductive model, for the specific semantics being used on P.*

We replace abducibles or their negations into $P$—instead of the more standard adding of abducibles as facts to $P$—because we also may abduce negations of abducibles, since the latter are **not** assumed *false* by default to begin with.

## 2 Abductive Reasoning with Logic Programs

Logic Programs have been used for a few decades now in knowledge representation and reasoning. Amongst the most common kinds of reasoning performed using them, one can find deduction, induction and abduction. Abduction, or inference to the best explanation, is a reasoning method whereby one chooses those hypotheses that would, if true, best explain the observed evidence—by meeting the corresponding ICs—and satisfy some query. Within deduction, and its abduction counterpart, so-named "brave" and "cautious" reasoning varieties are distinguished. "Brave" reasoning consists in finding if there exists at least one consistent model of the program—according to some pre-established semantics—which entails the query. "Cautious" reasoning demands that every model of the program entail the query.

In LPs, abductive hypotheses (or *abducibles*) are named literals of the program which have no rules. They can be considered *true* or *false* for the purpose of answering a query.

Abduction in LPs ([1,5,6,10,11]) can naturally be used in top-down query-oriented proof-procedures to find an (abductive) solution to a query, where the abducibles in the solution are leaves in the procedure's query-rooted call-graph—that is the graph recursively engendered by the procedure calls from literals in bodies of rules to heads of rules, and thence from the literals in a rule's body.

When query-answering, wherein abduction is enjoined as needed, if we know the underlying semantics is *relevant*, i.e. guarantees it is enough to use only the rules relevant to the query (those in its call-graph) to assess its truthfulness, then we need not compute a whole model in order to find an answer to a query: it suffices just to use the call-graph or relevant part of the program and determine the truth of a subset of the program's literals, those in the query's call-graph. Thus, top-down finding a solution to a query, dubbed "backward chaining", is possible only when the underlying semantics is relevant, in the above sense, because then the extension of that subset to a full model is guaranteed.

When performing abductive reasoning, we typically wish to find by need only—via backward chaining—the abductive solutions to a query. However, sometimes we also want to know which are some of the consequences (or side-effects) of such abductive solutions. I.e., we desire to know the truth value of some other literals, not part of the query's call-graph, whose truth-value may be determined by a found abductive solution. In some cases, we might be interested in knowing every possible side-effect—the truth-value of every literal in a complete model satisfying the query. In other situations though, our focus is frequently just on some specific side-effects.

In our approach, the side-effects of interest are explicitly indicated by the user, by wrapping the corresponding goals within the reserved construct $inspect/1$.

## 2.1 Abductive Logic Program Procedures

Currently, the standard 2-valued semantics used by the logic programming community is Stable Model (SM) semantics [9]. Its properties are well known and there are efficient implementations (such as *DLV* and *SModels* [4,12]). However, SM misses out on some important properties, both from the theoretical and practical perspectives: guarantee of model existence for every NLP, relevance and cumulativity—though the latter will not be of concern in the present context. Most importantly, since SM do not enjoy relevance they cannot just use backward chaining for query answering, irrespective of whether abduction is involved— indeed, odd-loops over default negation, outside the query's call-graph, may prevent model existence. This means SM implementations need to compute whole models, and so one will waste computational resources, because extra time and memory are required to compute parts of the model which are irrelevant to the query and ICs, i.e. outside their call-graph. The problem becomes compounded in abductive reasoning, because then the truth-value combinations of every abducible must be considered in order to provide complete models, even where abducibles are irrelevant to the query at hand. Moreover, such irrelevant abducibles and their combinations must subsequently be weeded out from the abductive models, at additional computational cost. On the other hand, because whole models are computed side-effects of abductive choices are computed too.

The Well-Founded Semantics (WFS) [8]—which enjoys model existence, relevance, and cumulativity—allows for top-down abductive query answering. Whole models need not to be computed, but then testing for side-effects involves extra querying about side-effected literals. One important issue we address with the introduction of inspection points is how to query the side-effects of a given abductive solution without performing additional abductions in the process. In so doing we avoid producing whole models still. We used WFS in the specific implementation described in section 4 based on ABDUAL [1]. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued.

Because they do not depend on any other literal in the program, abducibles can be modeled in a LP system without specific abduction mechanisms by automatically including for each *abducible* an even loop over default negation, e.g.,

$$abducible \leftarrow not\ neg\_abducible. \qquad neg\_abducible \leftarrow not\ abducible.$$

where $neg\_abducible$ is a new abducible atom, representing the (abducible) negation of the abducible. This way, under the SM semantics, a program may have models where some $abducible$ is *true* and another where it is *false*, i.e. $neg\_abducible$ is *true*. If there are $n$ abducibles in the program, there will be $2^n$ models corresponding to all the possible combinations of *true* and *false* for each. Under the WFS without a specific abduction mechanism, both $abducible$ and $neg\_abducible$ remain *undefined* in the Well-Founded Model (WFM), but may hold (as alternatives) in Partial Stable Models. In ABDUAL, however, a specific distinct mechanism is employed: abducibles and their negations are explicitly collected during search.

Using the SM semantics, unless the program is stratified, abduction must be done by guessing the truth-value of each abducible, providing the whole model and testing it for stability; whereas using WFS, even for non-stratified programs, abduction can be performed *by need*, induced by the top-down query solving procedure, solely for the relevant abducibles—i.e., irrelevant abducibles are left unconsidered. Thus, top-down abductive query answering is a means of finding those abducible values one might commit to in order to satisfy a query.

A new additional procedural preoccupation, addressed in this paper, is when one wishes to only passively determine which abducibles would be sufficient to satisfy some goal but without actually abducing them, just consuming other goals' needed and produced abductions. The difference is subtle but of importance, and it requires a new construct. Its mechanism, of *inspecting without abducing*, can be conceived and implemented through *meta-abduction*, or conditional abduction, as discussed in detail in the sequel.

## 3 Inspection Points

When faced with some situation where several alternative courses of actions are available a rational agent must decide and choose which action to take. *A priori* preferences can be applied before choosing in order to reduce the number of considerable possible actions curtailing the explosion of irrelevant combinations of choices, but still several (possibly exclusive) may remain available.

To make the best possible informed decision, and commit to a course of action, the agent must be enabled to foresee the consequences of its actions and then prefer on the basis of those consequences (with *a posteriori* preferences). Choosing which set of consequences is most preferred corresponds to an implicit choice on restricting which course of action to take. But only consequences relevant to the *a posteriori* preferences should be calculated: there are virtually infinitely many consequences of a given action, most of which are completely irrelevant to the preference-based decision making. Other consequences may be just predictions about the present state of the world, and observing whether they are verified can eliminate hypothetical scenarios where certain decisions would appear to make sense.

Not all consequences are experimentally observable though, hence Inspection Points (IPs) may serve to focus on the ones that are, and thus guide the experimentation required to decide among competing hypotheses. That is, IPs can be put to the service of sifting through competing explanations. In science, such decisive consequences are known as "crucial" side-effects, because they exclude untoward hypotheses. However, this is not the place to discuss the varied uses of abduction and its pragmatics. Instead, we direct the reader to Robert Kowalski's online book draft, available at his home page.

## 3.1 Backward and Forward Chaining

Abductive query-answering is intrinsically a backward-chaining process, a top-down dependency-graph oriented proof-procedure. Finding the side-effects of a set of abductive assumptions may be conceptually envisaged as forward-chaining, as it consists of progressively deriving consequences from the assumptions until the truth value of the chosen side-effect literals is determined.

The problem with full-fledged forward-chaining is that too many (often irrelevant) conclusions of a model are derived. Wasting time and resources deriving them only to be discarded afterwards is a flagrant setback. Even worse, in combinatorial problems, there may be many alternative solutions whose differences repose just on irrelevant conclusions. So, the unnecessary computation of irrelevant conclusions in full forward-chaining may be multiplied, leading to immense waste.

A more rational solution, when one is focused on some specific conclusions from a set of premises, is afforded by a selective top-down ersatz forward-chaining. In this setting, the user can specify the conclusions she is focused on, and only those are computed in a backward-chaining fashion, checking whether they are consequences of desired abductions, but without further abducing. Combining backward-chaining with such ersatz forward-chaining allows for a greater precision in specifying what we wish to know, and altogether improve efficient use of computational resources, because focusing on the points of interest.

Crucially, if abduction is enabled, the computation of side-effects should take place without further abduction, passively —but not destructively— just "consuming" abducibles "produced" elsewhere by abduction, for the top query.

In the sequel, we show how such ersatz forward-chaining from a set of hypotheses can be achieved by backward chaining from the consequences focused on—the inspection points—by virtue of a controlled form of abduction.

## 3.2 Meta-abduction for Side-Effects Inspection

"Meta-abduction" is used in *abduction inhibited inspection*. Intuitively, when an abducible is considered under mere inspection, meta-abduction abduces only the intention to *a posteriori* check for its abduction elsewhere, i.e. it abduces the intention of verifying that the abducible is indeed adopted—that is, it abduces on condition. In practice, when we want to meta-abduce some abducible '$X$', we abduce a literal '$consume(X)$' (or '$abduced(X)$'), which represents the intention that '$X$' is eventually abduced elsewhere in the process of finding an abductive solution. The pairing check is performed after a complete abductive answer to the top query is found. Meta-abduction, by its very nature, can be supported by any abduction capable system.

In the examples below, we are not propounding a methodology for using abduction, but simply illustrating the concepts we have introduced.

*Example 2.* **Police and Tear Gas Issue.** Consider this NLP, where '$tear\_gas$', '$fire$', and '$water\_cannon$' are the only abducibles. Notice the two rules for '$smoke$'. The first states that one explanation for smoke is fire, when assuming the hypothesis '$fire$'. The second states '$tear\_gas$' is also a possible explanation for smoke. However, the presence of tear gas is a much more unlikely situation than the presence of fire; after all, tear gas is only used by police to contain riots and that is truly an exceptional situation. Fires are much more common and spontaneous than riots. For this reason, '$fire$' is a much more plausible explanation for '$smoke$' and, therefore, in order to let the explanation for '$smoke$' be '$tear\_gas$', there must be a plausible reason—imposed by some other likely phenomenon. This is represented by $inspect(tear\_gas)$ instead of simply '$tear\_gas$'.

$$\begin{array}{ll}
\leftarrow police, riot, not\ contain. & \text{\% this is an Integrity Constraint} \\
contain \leftarrow tear\_gas. & contain \leftarrow water\_cannon. \\
smoke \leftarrow fire. & smoke \leftarrow inspect(tear\_gas). \\
police. & riot.
\end{array}$$

The '$inspect$' construct disallows regular abduction—allowing only a conditional meta-abduction to be performed whilst trying to solve '$tear\_gas$'. I.e., if we take tear gas as an abductive solution for smoke, this rule imposes that the step where we abduce '$tear\_gas$' is performed elsewhere, not under the derivation tree for '$smoke$'. Thus, '$tear\_gas$' is an *inspection point*. The IC, because there is '$police$' and a '$riot$', forces '$contain$' to be *true*, and hence, '$tear\_gas$' or '$water\_cannon$' or both must be abduced. '$smoke$' is only explained if, at the end of the day, '$tear\_gas$' is abduced to enact containment. Abductive solutions should be plausible, and '$smoke$' is plausibly explained by '$tear\_gas$' if there is a reason, a best explanation, that makes the presence of tear gas plausible; in this case the riot and the police. Crucially, if the police were not around, or there was no riot, '$tear\_gas$' could not be abduced to explain '$smoke$'. Plausibility is an important concept in science, for lending credibility to hypotheses. Assigning plausibility measures to situations is an orthogonal issue though.

*Example 3.* **Nuclear Power Plant Decision Problem.** This example was extracted from [13] and adapted to our current designs, and its abducibles do not represent actions. In a nuclear power plant there is decision problem: cleaning staff will dust the power plant on cleaning days, but only if there is no alarm sounding. The alarm sounds when the temperature in the main reactor rises above a certain threshold, or if the alarm itself is faulty. When the alarm sounds everybody must evacuate the power plant immediately! Abducible literals are $cleaning\_day$, $temperature\_rise$ and $faulty\_alarm$.

$$\begin{array}{ll} dust & \leftarrow cleaning\_day, inspect(not\ sound\_alarm) \\ sound\_alarm & \leftarrow temperature\_rise \\ sound\_alarm & \leftarrow faulty\_alarm \\ evacuate & \leftarrow sound\_alarm \\ & \leftarrow not\ cleaning\_day \end{array}$$

Satisfying the unique IC imposes $cleaning\_day$ *true* (we may not employ a fact as $cleaning\_day$ is an abducible and these may not have rules), and that gives us three minimal abductive solutions to what happens on a cleaning day:

$S_1 = \{dust, cleaning\_day\}$,
$S_2 = \{cleaning\_day, sound\_alarm, temperature\_rise, evacuate\}$, and
$S_3 = \{cleaning\_day, sound\_alarm, faulty\_alarm, evacuate\}$.

If we pose the query $? - not\ dust$ we want to know what could justify the cleaners dusting not to occur given that it is a cleaning day (enforced by the IC). However, we do not want to abduce the rise in temperature of the reactor nor to abduce the alarm to be faulty in order to prove $not\ dust$. Any of these justifying two abductions must result as a side-effect of the need to explain something else, for instance the observation of the sounding of the alarm, expressible by adding the IC $\leftarrow not\ sound\_alarm$, which would then abduce one or both of those two abducibles as plausible explanations. Hence $S_2$ and $S_3$ are not solutions to the query, as intended in [13]. They would be, however, if the query were $? - not\ dust, evacuate$. The $inspect/1$ in the body of the rule for $dust$ prevents any abduction below $sound\_alarm$ to be made just to make $not\ dust$ true. One other possibility would be for two observations, coded by ICs $\leftarrow not\ temperature\_rise$ or $\leftarrow not\ faulty\_alarm$, to be present in order for $not\ dust$ to be true as a side-effect. A similar argument can be made about evacuating: one thing is to explain why evacuation takes place, another altogether is to justify it as necessary side-effect of root explanations for the alarm to go off. These two pragmatic uses correspond to different queries: $? - evacuate$ and $? - inspect(evacuate)$, respectively.

## 3.3 Declarative Semantics of Inspection Points

A simple transformation $\Pi$ maps any NLP $P$, with possibly nested inspection points—that is inspection points under the scope of other ones—into a NLP $TP$ without them.

**Definition 4.** ***Abductive Models.*** *Abductive Models are those models obtained by the abductive solutions—according to the base semantics which is applied to the transformed program* $TP$—in which each $abduced(X)$ is required to be matched by the corresponding $X$. *Thus the transformation* $\Pi$ *provides a definitional transformative declarative semantics for* $P$*, no matter what the base semantics chosen and its actual implementation*

Both the Stable Models or the Well-Founded Semantics are used in this paper, corresponding to different implementations naturally. For instance, the Abductive Stable Models of some $TP$, are the stable models for its abductive solutions, with respect to the source abducibles for $P$ plus those abducibles introduced by the transformation. Likewise for the Abductive Well-Founded Models.

In essence, $TP$ adds to $P$ duplicates of its rules, wrapping each literal with $inspect/1$, except for the abducibles, which are treated differently. Mark, below, that the abductive Stable Models of the transform $TP$—in which, by definition, each $abduced(X)$ is required to be matched by the corresponding $X$—clearly correspond to the intended meaning ascribed to the inspection points of the original program, as the example illustrates.

**Definition 5.** ***Transforming Inspection Points.*** *Let $P$ be a program containing rules whose body possibly contains inspection points. The program $\Pi(P)$ consists of:*

1. *all the rules obtained from the rules in $P$ by systematically replacing:*
   - *$inspect(not\ L)$ with $not\ inspect(L)$;*
   - *$inspect(L)$ with $abduced(a)$*
     *if $L$ is an abducible $a$, and keeping $inspect(L)$ otherwise.*
2. *plus, for each rule $A \leftarrow L_1, \ldots, L_t$ in the replaced rules of $P$ from step 1, the additional rule:*
   *$inspect(A) \leftarrow L_1', \ldots, L_t'$ where for every $1 \le i \le t$:*

$$L_i' = \begin{cases} abduced(L_i) & \text{if } L_i \text{ is an abducible} \\ inspect(X) & \text{if } L_i \text{ is } inspect(X) \\ inspect(L_i) & \text{otherwise} \end{cases}$$

*The semantics of the $inspect/1$ predicate is exclusively given by the generated rules for $inspect/1$. Moreover, '$abduced/1$' is an abducible, joining the original abducibles.*

*Example 4.* **Transforming a Program P with Nested Inspection Points.**

$$\begin{array}{ll} x \leftarrow a, inspect(y), b, c, not\ d & y \leftarrow inspect(not\ a) \\ z \leftarrow d & y \leftarrow b, inspect(not\ z), c \end{array}$$

where the abducibles are $a, b, c, d$. Then, $\Pi(P)$ is:

$$\begin{array}{lll} x & \leftarrow a, inspect(y), b, c, not\ d & \\ inspect(x) & \leftarrow abduced(a), inspect(y), abduced(b), abduced(c), not\ abduced(d) & \\ y & \leftarrow not\ inspect(a) & \\ y & \leftarrow b, not\ inspect(z), c & \\ inspect(y) & \leftarrow not\ abduced(a) \qquad \%\ by\ two\ rewrites & \\ inspect(y) & \leftarrow abduced(b), not\ inspect(z), abduced(c) & \\ z & \leftarrow d & \\ inspect(z) & \leftarrow abduced(d) & \end{array}$$

The single abductive stable model of $\Pi(P)$—that its stable model for its single abductive solution—respecting the meaning of the inspection points declarations in $P$ is:
$\{x, a, b, c, abduced(a), abduced(b), abduced(c), inspect(y)\}$.
Note that indeed for each $abduced(X)$ the corresponding $X$ is in the model.

## 4 Implementation

We based our practical work on a formally defined, XSB-implemented, true and tried abduction system—ABDUAL [1]. ABDUAL lays the foundations for efficiently computing queries over ground 3-valued abductive frameworks for extended logic programs with integrity constraints, on the well-founded semantics and its partial stable models.

The query processing technique in ABDUAL relies on an admixture of program transformation and tabled evaluation. A transformation removes default negative literals (by making them positive) from both the program and the integrity rules. Specifically, a dual transformation is used, that defines for each objective literal $O$ (i.e. an atom or explicit negated atom) and its set of rules $R$, a dual set of rules whose conclusions $not\ (O)$ are true if and only if $O$ is false in $R$. Tabled evaluation of the resulting program turns out to be much simpler than for the original program, whenever abduction over negation is needed. At the same time, termination and complexity properties of tabled evaluation of extended programs are preserved by the transformation, when abduction is not needed. Regarding tabled evaluation, ABDUAL is in line with SLG [15] evaluation, which computes queries to normal programs according to the well-founded semantics. To it, ABDUAL tabled evaluation adds mechanisms to handle abduction and deal with the dual programs.

ABDUAL is composed of two modules: the preprocessor which transforms the original program by adding its dual rules, plus specific abduction-enabling rules; and a meta-interpreter allowing for top-down abductive query solving. When solving a query, abducibles are dealt with by means of extra rules the preprocessor added to that effect. These rules just add the name of the abducible (or its negation) to an ongoing list of current abductions, unless the negation of the abducible was added before to the lists, then failing in order to ensure abduction consistency. Our conditional meta-abduction is implemented adroitly by means of a reserved predicate, '$inspect/1$' taking some literal $L$ as its argument, which engages the abduction mechanism to try and discharge any conditional meta-abductions performed under $L$ by matching with the corresponding abducibles, adopted elsewhere outside from under any '$inspect/1$' call. The approach taken can easily be adopted by other abductive systems, albeit in part—e.g. inspecting only abducibles directly, and so omitting inspection nesting too—as we had the occasion to check, namely with the authors of system [3]. We have also enacted an alternative implementation, relying on XSB-XASP and the declarative semantics transformation above, which is reported further below.

Procedurally, in the ABDUAL implementation, the checking of an inspection point corresponds to performing a top-down query-proof for the inspected literal, but with the specific proviso of disabling new abductions during that proof. The proof for the inspected literal will succeed only if the abducibles needed for it were already adopted, or will be adopted, in the present ongoing solution search for the top query. Consequently, this check is performed after a solution for the query has been found, except for "quick-kill" cases, as when the opposite abduction has already been collected in the ongoing solution. At "inspection-point-top-down-proof-mode", whenever an abducible is encountered, instead of adopting it, we simply adopt the intention to *a posteriori* check if the abducible is part of the answer to the query. That is, one conditionally (meta-) abduces the checking of some abducible $A$, and the check consists in confirming that $A$ is part of the abductive solution by matching it with the object of the check. According to our method, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals, those to be subject to inspection mode, with the reserved construct '$inspect/1$'.

## 4.1 ABDUAL with Inspection Points—Details

Inspection points in ABDUAL function mainly by means of controlling the general abduction step, which involves very few changes, both in the pre-processor and the meta-interpreter, that might be imported into other abduction systems. Whenever an '$inspect(X)$' literal is found in the body of a rule, where '$X$' is a goal, a meta-abduction-specific counter—the '$inspect\_counter$', initialized with zero—is increased by one, in order to keep track of the allowed character, active or passive, of ongoing abduction performing. The top-down evaluation of the query for '$X$' then proceeds normally. Active abductions are only allowed if the counter is set to zero, otherwise only meta-abductions are permitted. After finding an abductive solution to query '$X$', the counter is decreased by one, since that inspection execution of $X$ has been completed. Backtracking over counter assignations is duly accounted for. Of course, this way of implementing the inspection points (with a single '$inspect\_counter$') presupposes the abductive query answering process is carried out "depth-first", guaranteeing that the order of the literals in the bodies of rules actually corresponds to the order they are processed in. For simplicity of description, we assume such a "depth-first" discipline in the implementation of inspection points, described in detail below. We then lift this restriction at the end of the subsection.

**Changes to the pre-processor:**

1. A new dynamic predicate was added: the '$inspect\_counter/1$'. This is initialized to zero ('$inspect\_counter(0)$') via an assert, before a top-level query is launched.
2. The original rules for the normal abduction step are now preceded by an additional condition checking that the '$inspect\_counter$' is indeed set to zero.
3. Extra rules for the "inspection" abduction step are added, preceded by a condition checking the '$inspect\_counter$' is set to greater than zero. When these rules are called, the corresponding abducible '$A$' is not abduced as it would happen in the original rules; instead, '$consume(A)$' (or '$abduced(A)$') is abduced. This corresponds to the conditional meta-abduction: we abduce the need to abduce '$A$', the need to 'consume' the abduction of '$A$', which is finally checked when derivation for the very top goal is finished.

**Changes to the meta-interpreter:** The changes to the meta-interpreter include all the remaining processing needed to correctly implement inspection points, namely the matching of the abduction of '$consume(X)$' against the abduction of '$X$'. If a conditional meta-abduction on '$X$' (producing '$consume(X)$') is not matched by an actual abduction on '$X$' when the end of solving the top query is reached, the candidate abductive answer is considered invalid and the attempted query solving fails. On backtracking, an alternative abductive solution (possibly with other meta-abductions) will be sought.

In detail, the changes to the meta-interpreter include:

1. Two "quick-kill" rules for improved efficiency that detect and immediately solve trivial cases for conditional meta-abduction:
   - When literal '$X$' about to be meta-abduced ('$consume(X)$' about to be added to the abductions list) has actually been abduced already ('$X$' is in the abductions list) the meta-abduction succeeds immediately and '$consume(X)$' is not added to the abductions list;
   - When the situation in the previous point occurs, but with '$not\ X$' already abduced instead, the meta-abduction immediately fails.
2. Two new rules for the general case of meta-abduction, that now specifically treat the '$inspect(not\ X)$' and '$inspect(X)$' literals. In either rule, first we increase the '$inspect\_counter$' mentioned before, then proceed with the usual meta-interpretation for '$not\ X$' ('$X$', respectively), and, when this evaluation succeeds, we then decrease '$inspect\_counter$'.
3. After an abductive solution is found to the top query, ensure that every meta-abduction, i.e. every '$consume(X)$' literal abduced, is indeed matched by a corresponding and consistent abduction, i.e. that it is matched by the abducible '$X$' in the abductions list; otherwise the tentative solution found fails.

A counter—'$inspect\_counter$'—is employed instead of a simple toggle because several '$inspect(X)$' literals may appear at different graph-depth levels under one another, and resetting a toggle after solving a lower-level meta-abduction would enable producer abductions under the higher-level meta-abduction. An example clarifies this.

*Example 5.* **Nested Inspection Points.** Consider again the program of the previous example, where the abducibles are $a, b, c, d$:

$$\begin{array}{ll} x \leftarrow a, inspect(y), b, c, not\ d. & y \leftarrow inspect(not\ a). \\ z \leftarrow d. & y \leftarrow b, inspect(not\ z), c. \end{array}$$

When we want to find an abductive solution for $x$—skipping over the low-level technical details—we proceed as follows:

1. $a$ is an abducible and since the '$inspect\_counter$' is still set initially to 0 we can abduce $a$ by adding it to the running abductions list;
2. $y$ is not an abducible and so we cannot use any "quick-kill" rule on it. We increase the '$inspect\_counter$'—which now takes the value 1—and proceed to find an abductive solution to $y$;
3. Since the '$inspect\_counter$' is different from 0, only meta-abductions are allowed;
4. Using the first rule for $y$ we need to '$inspect(not\ a)$', but since we have already abduced $a$, a "quick-kill" is applicable here: we already know that this '$inspect(not\ a)$' will fail. The value of the '$inspect\_counter$' will remain 1;
5. On backtracking, the second rule for $y$ is selected, and now we meta-abduce $b$ by adding '$consume(b)$' to the ongoing abductions list;
6. Increase the '$inspect\_counter$' again, making it take the value 2, and continue on searching for an abductive solution to $not\ z$;

7. The only solution to $not\ z$ is by abducing $not\ d$, but since the '$inspect\_counter$' is greater than 0, we can only meta-abduce $not\ d$, i.e. '$consume(not\ d)$' is added to the running abductions list;
8. Returning to $y$'s rule: the meta-interpretation of '$inspect(not\ z)$' succeeds and so we decrease the '$inspect\_counter$' by one—it takes the value 1 again. Now we proceed and attempt to solve $c$;
9. $c$ is an abducible, but since the $inspect\_counter$ is set to 1, we only meta-abduce $c$ by adding '$consume(c)$' to the running abductions list;
10. Returning to $x$'s rule: the meta-interpretation of '$inspect(y)$' succeeds and so we decrease the '$inspect\_counter$' once more, and it now takes the value 0. From this point onwards regular abductions will take place instead of meta-abductions;
11. We abduce $b$, $c$, and $not\ d$ by adding them to the abductions list;
12. A tentative abductive solution is found to the initial query. It consists of the abductions list: $[a, consume(b), consume(not\ d), consume(c), b, c, not\ d]$;
13. The abductive solution is now checked for matches between meta-abductions and producer abductions.
    In this case, for every '$consume(A)$' in the abduction list there is actually an $A$ also in the abduction list, i.e. each abduction intention '$consume(A)$' is satisfied by a producer abduction $A$, where the $A$ in $consume(A)$ is just any abducible literal $a$ or its default negation $not\ a$. It is irrelevant in which order a '$consume(A)$' and the corresponding $A$ appear or were placed in the abductions list. Because this final checking step succeeds, the abductive solution is actually accepted.

In this example, we can clearly see that the $inspect$ predicate can be used on any arbitrary literal, and not just on abducibles.

The correctness of this implementation against the declarative semantics provided before can be sketched by noticing that whenever the $inspect\_counter$ is set to 0 the meta-interpreter performs actual abduction, which corresponds to the use of the original program rules; whenever the $inspect\_counter$ is set to some value greater than 0, the meta-interpreter just abduces $consume(A)$—where $A$ is the abducible being checked for its abduction being produced elsewhere—and that corresponds to the use of the transformed program rules for the $inspect$ predicate.

The implementation of ABDUAL with inspection points is available on request.

**More general query solving.** In case the "depth-first" discipline is not followed, either because goal delaying is taking place, or multi-threading, or co-routining, or any other form of parallelism is being exploited, then each queried literal will need to carry its own list of ancestors with their individual '$inspect\_counters$'. This is necessary so as to have a means, in each literal, to know which and how many $inspect$s there are between the root node and the currently being processed literal, and which $inspect\_counter$ to update; otherwise there would be no way to know if abductions or meta-abductions should be performed.

## 4.2 Alternative Implementation Method

The method presented forthwith is an avenue for the implementation of the inspection points mechanism through a simple syntactic transformation that can readily be employed by any SMs system, like SModels or DLV. Using a SMs implementation alone,

one can get the abductive SMs of some program $P$ by computing the SMs of $P'$, where $P'$ is obtained from $P$ by applying the program transformation we presented before for the declarative semantics of the inspection points, and then adding an even loop over negation for each declared abducible—as shown in section 2.1. When using XSB-Prolog's XSB-XASP interface, the process method is the same as for when using a SMs implementation alone, but instead of sending the whole $P'$ to the SMs engine, only the residual or remainder program [2], the one that results from a query evaluated in XSB using tabling [14], relevant for the query at hand, is sent. This way, abductive reasoning may benefit from the relevance property enjoyed by the Well-Founded Semantics implemented in XSB-Prolog's SLG-WAM.

Given the top-down proof procedure for abduction, implementing inspection points for program $P$ becomes just a matter of adapting the evaluation of derivation subtrees falling under '$inspect/1$' literals, at meta-interpreter level, subsequent to performing the transformation $\Pi(P)$ presented before, which actually defines the declarative semantics. Basically, any considered abducibles evaluated under '$inspect/1$' subtrees, say $A$, are codified as '$abduced(A)$', where, as in section 2.1:

$$\begin{aligned} abduced(A) &\leftarrow not\ neg\_abduced(A) \\ neg\_abduced(A) &\leftarrow not\ abduced(A) \end{aligned}$$

All $abduced/1$ literals collected during computation of the residual program are later checked against the stable models themselves. Every '$abduced(A)$' in a model must pair with a corresponding abducible $A$ for the model to be accepted.

## 5 Conclusions, Comparisons, and Future Work

In the context of abductive logic programs, we have presented a new mechanism of inspecting literals that can be used to check for side-effects, by relying on conditional meta-abduction. We have implemented the inspection mechanism within the Abdual [1] meta-interpreter, as well as in XSB-XASP. We have further checked that our approach can easily be adopted, in part, by other systems [3] with the help of these cited authors.

HyProlog [3] is an abduction/assumption system which allows for the user to specify if an abducible is to be consumed only once or many times. In HyProlog, as the query solving proceeds, when abducible/assumption consumptions take place, they are executed by storing the corresponding consumption intention in a store. After an abductive solution for a query is found, the actual abductions/assumptions are matched against the consumption intentions. Overall, there is not such a big gap between the operational semantics of HyProlog and the inspection points implementation we present; however, there is a major functional difference: in HyProlog we can only specify consumption directly on abducibles, whereas in our more general inspection points approach we can declare inspection of any literal (not just abducibles)—meaning any abducible found below an inspect-wrapped literal call is automatically just inspected.

In [13], the authors detect a problem with the IFF abductive proof procedure [7] of Fung and Kowalski, in what concerns the treatment of negated abducibles in integrity constraints (e.g. in their examples 2 and 3). They then specialize IFF to avoid such problems, which arise only in ICs, and prove correctness of the new procedure. The detected problem refers to the active use of an IC comprising in its body some $notA$, where $A$ is an abducible, whereas the intended use should be a passive one, simply checking whether some A is proved in the abductive solution found. To that effect, by means of an inference rule used during query evaluation, it's as if they replaced such occurrences of $notA$ by '$not\ provable(A)$', before moving each as a disjunct '$provable(A)$' to the IC head along with other disjuncts, so as to ensure that no new abductions are allowed during IC checking, by virtue of '$provable/1$'. For a detailed exposition the reader is referred to their section 4.2. Our own work generalizes the scope of the problem they solved, and solves the problems arising in this wider scope. For one, we abduce both positive and negative literals, and the latter are not true by default. Moreover, we allow for passive checking not just of negated abducibles but also of positive ones, as well as passive checking of any literal, whether or not abducible and whether in ICs or other rules. Furthermore, we allow to single out which specific occurrences are passive or active. Thus, we can cater for both passive and active ICs, depending on the desired usage. Our solution uses abduction itself to solve the problem, making it general for deployment in other abductive frameworks and procedures.

A future application of inspection points is planning in a multi-agent setting. An agent may have abduced a plan and, in the course of carrying out its abduced actions, it may find that another agent has undone some of its already executed actions. So, before executing an action, the agent should check all necessary preconditions still hold. Note that it should only *check*, thereby avoiding abducing again a plan for them: this way, should the preconditions hold, the agent can continue and execute the planned action. The agent should only take measures to enforce the preconditions again whenever the check fails. Clearly, an "inspection" of the preconditions is what is needed here.

More generally, inspection points afford us with the ability to avoid having to generate complete abductive models in order to glean the consequences of interest of abductive solutions. The developed techniques can be employed too for permitting passive ICs, which are not allowed to actively abduce but only to verify their satisfaction with regard to given abductions, in contrast to active ICs that can further abduce in order to be satisfied. Plus, of course, to enable ICs which contain a combination of both active and passive literals.

Another future use concerns the computation of inspected consequences of partially defined 2-valued models, obtained by top-down querying of NLPs, wherein the abducibles are the default $not$s themselves, plus appropriate ICs to enforce consistency. Once again, the computation of complete models can thus be avoided. A 2-valued semantics which enjoys relevance must then be used, or otherwise a guarantee that the NLP is stratified or does not contain loops over default negation via an odd number of $not$s.

## Acknowledgements

We thank Robert Kowalski, Verónica Dahl and Henning Christiansen for discussions, Pierangelo Dell'Acqua for the declarative semantics, and Gonçalo Lopes for help with the XSB-XASP implementation. A special thanks to our helpful reviewer, Francesca Toni.

## References

1. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. Theory and Practice of Logic Programming 4(4), 383–428 (2004)
2. Brass, S., Dix, J., Freitag, B., Zukowski, U.: Transformation-based bottom-up computation of the well-founded model. TPLP 1(5), 497–538 (2001)
3. Christiansen, H., Dahl, V.: HyProlog: A new logic programming language with assumptions and abduction. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 159–173. Springer, Heidelberg (2005)
4. Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The dlv system: Model generator and advanced frontends (system description). In: 12th Workshop on Logic Programming (1997)
5. Denecker, M., De Schreye, D.: SLDNFA: An abductive procedure for normal abductive programs. In: Apt, K. (ed.) Proceedings of the Joint International Conference and Symposium on Logic Programming, Washington, USA, pp. 686–700. The MIT Press, Cambridge (1992)
6. Eiter, T., Gottlob, G., Leone, N.: Abduction from logic programs: semantics and complexity. Theoretical Computer Science 189(1-2), 129–177 (1997)
7. Fung, T.H., Kowalski, R.: The IFF proof procedure for abductive logic programming. J. Log. Prog. 33(2), 151–165 (1997)
8. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. of ACM 38(3), 620–650 (1991)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080. MIT Press, Cambridge (1988)
10. Inoue, K., Sakama, C.: A fixpoint characterization of abductive logic programs. Journal of Logic Programming 27(2), 107–136 (1996)
11. Kakas, A., Kowalski, R., Toni, F.: The role of abduction in logic programming. In: Handbook of Logic in AI and LP, vol. 5, pp. 235–324. Oxford University Press, Oxford (1998)
12. Niemelä, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS (LNAI), vol. 1265, pp. 420–429. Springer, Heidelberg (1997)
13. Sadri, F., Toni, F.: Abduction with negation as failure for active and reactive rules. In: Lamma, E., Mello, P. (eds.) AI*IA 1999. LNCS (LNAI), vol. 1792, pp. 49–60. Springer, Heidelberg (2000)
14. Swift, T.: Tabling for non-monotonic programming. AMAI 25(3-4), 201–240 (1999)
15. Swift, T., Warren, D.S.: An abstract machine for SLG resolution: definite programs. In: Procedings of the 1994 International Symposium on Logic Programming, ILPS 1994, Symposia, Melbourne, pp. 633–652. MIT Press, Cambridge (1994) ISBN 0-262-52191-1

# Argumentation and Answer Set Programming

Francesca Toni and Marek Sergot

Department of Computing,
Imperial College London, UK
{ft,mjs}@imperial.ac.uk

**Abstract.** Argumentation and answer set programming are two of the main knowledge representation paradigms that have emerged from logic programming for non-monotonic reasoning. This paper surveys recent work on using answer set programming as a mechanism for computing extensions in argumentation. The paper also indicates some possible directions for future work.

## 1 Introduction

Argumentation was developed, starting in the early '90s [9,16,8], as a computational framework to reconcile and understand common features and differences amongst most existing approaches to non-monotonic reasoning. These include various alternative treatments of negation as failure in logic programming [32,26,52,19], Theorist [43], default logic [45], autoepistemic logic [38], non-monotonic modal logic [37] and circumscription [36]. Argumentation relies upon

- the representation of knowledge in terms of an argumentation framework; defining *arguments* and a binary *attack* relation between the arguments;
- *dialectical semantics* for determining *acceptable* sets of arguments;
- a computational machinery for determining the acceptability of a given (set of) argument(s) or for computing all acceptable sets of arguments (also referred to as *extensions*), according to some dialectical semantics.

Answer set programming (ASP) [31] constitutes one of the main current trends in logic programming and non-monotonic reasoning. ASP relies upon

- the representation of knowledge in terms of (possibly disjunctive) logic programs with negation as failure (possibly including explicit negation, various forms of constraints, aggregates etc);
- the interpretation of these logic programs under the stable model/answer set semantics [32,33] and its extensions (to deal with explicit negation, constraints, aggregates etc);
- efficient computational mechanisms (ASP solvers) to compute answer sets for grounded logic programs, and efficient 'grounders' to transform non-ground logic programs into ground (variable-free) ones.

Standard computational mechanisms for argumentation are defined using trees (e.g. [18]) or disputes (e.g. [20]) and only construct relevant parts of extensions. ASP can instead be used to support the full computation of extensions.

This paper provides a survey of recent work using ASP for computing extensions in abstract argumentation frameworks [16] and some other forms of argumentation. It also indicates possible directions for future work and cross-fertilisation between ASP and argumentation.

The paper is organised as follows. Section 2 gives some background on argumentation (focusing on abstract argumentation [16]) and ASP. Section 3 surveys existing approaches using ASP to compute extensions in argumentation (again focusing on abstract argumentation). Section 4 indicates some possible directions for future work. Section 5 concludes.

## 2 Background

### 2.1 Argumentation

An *abstract argumentation (AA) framework* [16] is a pair $\langle Arg, att\rangle$ where $Arg$ is a finite set, whose elements are referred to as *arguments*, and $att \subseteq Arg \times Arg$ is a binary relation over $Arg$. Given $\alpha, \beta \in Arg$, $\alpha$ *attacks* $\beta$ iff $(\alpha, \beta) \in att$. Given sets $X, Y \subseteq Arg$ of arguments, $X$ *attacks* $Y$ iff there exists $x \in X$ and $y \in Y$ such that $(x, y) \in att$. A set of arguments is referred to as an *extension*. An extension $X \subseteq Arg$ is

– *conflict-free* iff it does not attack itself;
– *stable* iff it is conflict-free and it attacks every argument it does not contain;
– *acceptable wrt* a set $Y \subseteq Arg$ of arguments iff for each $\beta$ that attacks an argument in $X$, there exists $\alpha \in Y$ such that $\alpha$ attacks $\beta$;
– *admissible* iff $X$ is conflict-free and $X$ is acceptable wrt itself;
– *preferred* iff $X$ is (subset) maximally admissible;
– *complete* iff $X$ is admissible and $X$ contains all arguments $\alpha$ such that $\{\alpha\}$ is acceptable wrt $X$;
– *grounded* iff $X$ is (subset) minimally complete.

In addition, an extension $X \subseteq Arg$ is

– *ideal* [18] iff $X$ is admissible and it is contained in every preferred set of arguments;
– *semi-stable* [12] iff $X$ is complete and $X \cup X^+$ is (subset) maximal, where $X^+ = \{\beta \mid (\alpha, \beta) \in att \text{ for some } \alpha \in X\}$.

These notions of extensions constitute different alternative dialectical semantics, giving different approaches for determining what makes arguments dialectically viable. Arguments can be deemed to hold *credulously* wrt a given dialectical semantics if they belong to an extension sanctioned by that semantics. Arguments can be deemed to hold *sceptically* wrt a given dialectical semantics if they belong to all extensions sanctioned by that semantics. In some cases credulous and sceptical reasoning coincide, e.g. for grounded and ideal extensions, since these are unique.

For $AF = \langle Arg, att \rangle$, the *characteristic function* $\mathcal{F}_{AF}$ is such that $\mathcal{F}_{AF}(X)$ is the set of all acceptable arguments wrt $X$. Then, a conflict-free $X \subseteq Arg$ is

– an admissible extension iff $X \subseteq \mathcal{F}_{AF}(X)$,
– a complete extension iff it is a fixpoint of $\mathcal{F}_{AF}$, and
– a grounded extension iff $X$ is the least fixpoint of $\mathcal{F}_{AF}$.

Several other argumentation frameworks have been given in the literature, concretely specifying arguments and attacks, some instantiating abstract argumentation, e.g. assumption-based argumentation [9,8,17] and logic programming-based argumentation frameworks such as [44], some equipped with dialectical semantics other than the ones proposed for abstract argumentation, e.g. [7,30]. Moreover, extensions of abstract argumentation have been proposed, e,g, value-based argumentation [5].

## 2.2 Answer Set Programming (ASP)

A logic program is a set of clauses of the form

$$p_1 \vee \ldots \vee p_k \leftarrow q_1 \wedge \ldots \wedge q_m \wedge not\, q_{m+1} \wedge \ldots \wedge not\, q_{m+n}$$

for $k \geq 0$, $m \geq 0$, $n \geq 0$, $k + m + n > 0$, where the $p_i$ and $q_j$ are *literals*, that is, of the form $p$ or $\neg p$ where $p$ is an atom. *not* denotes negation as failure. Expressions of the form $not\, q_j$ where $q_j$ is a literal are called 'negation as failure literals', or *nbf-literals* for short. We will refer to $\{p_1, \ldots, p_k\}$ as the *head*, $\{q_1, \ldots, q_m, not\, q_{m+1}, \ldots not\, q_{m+n}\}$ as the *body* and $\{not\, q_{m+1}, \ldots not\, q_{m+n}\}$ as the *negative body* of a clause. We will also refer to clauses with $k = 0$ as *denial clauses*, clauses with $k > 1$ as *disjunctive* clauses, and clauses with $n = 0$ as *positive clauses.*

All variables in clauses in a logic program are implicitly universally quantified, with scope the individual clauses. A logic program stands for the set of all its ground instances over a given Herbrand universe. The semantics of logic programs is given for their grounded version over this Herbrand universe.

The answer sets of a (grounded) logic program are defined as follows [33].

Let $X$ be a set of literals (that is, expressions $p$ or $\neg p$ where $p$ is an atom). A literal is true in $X$ if it belongs to $X$. A nbf-literal $not\, p$ is true in $X$ if the literal $p$ does not belong to $X$. A clause is true in $X$ if its head is true in $X$ (there exists a literal in the head that is true in $X$) whenever its body is true in $X$ (i.e., when all the literals and all the nbf-literals in that body are true in $X$). Thus, denial clauses are true in $X$ if and only if their body is false in $X$ (i.e., some literal in the body is not true in $X$). A set of literals $X$ is closed under a set of clauses $P$ if every clause in $P$ is true in $X$. $X$ is 'consistent' if it contains no pair of complementary literals $p$ and $\neg p$ for some atom $p$; $X$ is logically closed if it is consistent or if it is the set of all literals otherwise.

If $P$ is a set of positive clauses, that is, a set of clauses containing no occurrences of negation as failure *not*, then $X$ is an *answer set* of $P$ if it is a (subset) minimal set of literals that is logically closed and closed under the clauses in $P$.

If $P$ is any set of clauses (not necessarily positive ones) the Gelfond-Lifschitz reduct $P^X$ of $P$ wrt to the set of literals $X$ is obtained from $P$ by deleting (1) all clauses in $P$ containing a nbf-literal $not\, p$ in the body where $p$ is true in $X$, and (2) the negative body from all remaining clauses. $P^X$ is a set of positive clauses. Then $X$ is an *answer set* of $P$ when $X$ is an answer set of $P^X$.

None of the logic programs presented in section 3 of this paper contain occurrences of classical negation ($\neg$): all of the literals in all of the clauses are atoms. For programs $P$ of this type, the answer sets of $P$ are also the *stable models* [32] of $P$.

Several very efficient ASP solvers are widely available and can be used to compute answer sets and/or perform query answering wrt answer sets. These solvers include Smodels[1], DLV[2] and clasp[3]. These solvers incorporate, or are used in combination with, *grounders*, programs whose function is to generate, prior to computing answer sets, a finite ground logic program from a non-ground logic program (over a given, not necessarily finite Herbrand Universe).

## 3 ASP for Argumentation

Several approaches have been proposed for computing (several kinds of) extensions of *abstract argumentation* (AA) frameworks using ASP solvers [39,53,23,27]. All rely upon the mapping of an AA framework into a logic program whose answer sets are in one-to-one correspondence with the extensions of the original AA framework. All those summarised below use the DLV solver to compute these answer sets (and thus the extensions). The approaches differ in the kinds of extensions they focus on and in the mappings and correspondences they define, as we will see below. They fall into two groups: those which result in an AA framework-dependent logic program, and those which result in a logic program with an AA framework-dependent component and an AA framework-independent (meta-)logic program.

### 3.1 Nieves, Cortés and Osorio [39]: Preferred Extensions

Nieves et al [39] focus on the computation of **preferred** extensions. Their mapping relies upon the method of [6] using propositional formulas to express conditions for sets of arguments to be extensions of AA frameworks. The mapping produces a disjunctive logic program defining a predicate *def*, where $def(\alpha)$ can be read as 'argument $\alpha$ is defeated'. Intuitively,

– each pair $(\alpha, \beta)$ in the *att* component of an AA framework $(Arg, att)$ is mapped to a disjunctive clause $def(\alpha) \vee def(\beta) \leftarrow$ ;

[1] http://www.tcs.hut.fi/Software/smodels/
[2] http://www.dbai.tuwien.ac.at/proj/dlv/
[3] http://potassco.sourceforge.net/

– for each pair $(\beta, \alpha) \in \mathit{att}$, a clause $\mathit{def}(\alpha) \leftarrow \mathit{def}(\gamma_1) \wedge \ldots \wedge \mathit{def}(\gamma_k)$ $(k \geq 0)$ is introduced, where $\gamma_1, \ldots, \gamma_k$ are all the 'defenders' of $\alpha$ against $\beta$ (that is, where $(\gamma_1, \beta), \ldots, (\gamma_k, \beta) \in \mathit{att}$, and there are no other attacks against $\beta$ in $\mathit{att}$).

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \end{tikzcd}$$

**Fig. 1.** Graph representation for the AA framework $(\{a, b, c\}, \{(a, b), (b, c)\})$

For the AA framework of figure 1, the mapping returns

$$\begin{aligned} &\mathit{def}(a) \vee \mathit{def}(b) \leftarrow \\ &\mathit{def}(b) \vee \mathit{def}(c) \leftarrow \\ &\mathit{def}(c) \leftarrow \mathit{def}(a) \\ &\mathit{def}(b) \leftarrow \end{aligned}$$

The answer sets of the disjunctive logic program $P^{\mathit{pref}}_{NCO}$ thus obtained are in one-to-one correspondence with the preferred extensions of the original AA framework $\langle \mathit{Arg}, \mathit{att} \rangle$, in that the complement

$$\mathcal{C}(AS) \ =_{\text{def}} \ \{\alpha \in \mathit{Arg} \mid \mathit{def}(\alpha) \notin AS\}$$

of each answer set $AS$ of $P^{\mathit{pref}}_{NCO}$ is a preferred extension of $\langle \mathit{Arg}, \mathit{att} \rangle$.

In the example of the AA framework of figure 1 and the resulting $P^{\mathit{pref}}_{NCO}$ given earlier, the only answer set is $\{\mathit{def}(b)\}$, corresponding to the only preferred extension $\{a, c\} = \mathcal{C}(\{\mathit{def}(b)\})$ of the original AA framework.

$$\begin{tikzcd} a \arrow[loop left] \end{tikzcd}$$

**Fig. 2.** Graph representation for the AA framework $(\{a\}, \{(a, a)\})$

In the example of figure 2, $P^{\mathit{pref}}_{NCO}$ is

$$\begin{aligned} &\mathit{def}(a) \vee \mathit{def}(a) \leftarrow \\ &\mathit{def}(a) \leftarrow \mathit{def}(a) \end{aligned}$$

with answer set $\{\mathit{def}(a)\}$ corresponding to the (only) preferred extension $\{\}$ of the original AA framework. In the case of the AA framework of figure 3, $P^{\mathit{pref}}_{NCO}$ is

$$\begin{aligned} &\mathit{def}(a) \vee \mathit{def}(b) \leftarrow \\ &\mathit{def}(a) \leftarrow \mathit{def}(a) \\ &\mathit{def}(b) \leftarrow \mathit{def}(b) \end{aligned}$$

with answer sets $\{\mathit{def}(a)\}$ and $\{\mathit{def}(b)\}$ corresponding to the preferred extensions $\{b\}$ and $\{a\}$ (respectively) of the original AA framework.

$$\begin{tikzcd} a \arrow[r, leftrightarrow] & b \end{tikzcd}$$

**Fig. 3.** Graph representation for the AA framework $(\{a, b\}, \{(a, b), (b, a)\})$

### 3.2 Wakaki and Nitta [53]: Complete, Stable, Preferred, Grounded, and Semi-stable Extensions

Wakaki and Nitta [53] focus on the computation of complete, stable, preferred, grounded, and semi-stable extensions. Their mappings rely upon Caminada's *reinstatement labellings* [11] and correspondences between various kinds of constraints on such labellings and various notions of extensions in abstract argumentation. Intuitively, a reinstatement labelling is a total function from arguments to labels $\{in, out, undec\}$ such that (i) an argument is labelled *out* iff some argument attacking it is labelled *in*, and (ii) an argument is labelled *in* iff all arguments attacking it are labelled *out*.

All mappings given by Wakaki and Nitta result in a logic program that contains, for a given AA framework $\langle Arg, att\rangle)$, a set $P_{\langle Arg,att\rangle}$ of clauses $arg(\alpha) \leftarrow$ for all arguments $\alpha \in Arg$ and clauses $att(\alpha, \beta) \leftarrow$ for all pairs $(\alpha, \beta) \in att$. For example, in the case of the AA framework of figure 1, one obtains:

$$\begin{aligned} &arg(a) \leftarrow \\ &arg(b) \leftarrow \\ &arg(c) \leftarrow \\ &att(a, b) \leftarrow \\ &att(b, c) \leftarrow \end{aligned}$$

In the case of **complete** extensions, the logic program $P^{compl}_{WN}$ resulting from the mapping includes in addition to $P_{\langle Arg,att\rangle}$ the following (AA framework-independent) clauses (directly corresponding to the notion of reinstatement labelling):

$$\begin{aligned} &in(X) \leftarrow arg(X) \wedge not\, ng(X) \\ &ng(X) \leftarrow in(Y) \wedge att(Y, X) \\ &ng(X) \leftarrow undec(Y) \wedge att(Y, X) \\ &out(X) \leftarrow in(Y) \wedge att(Y, X) \\ &undec(X) \leftarrow arg(X) \wedge not\, in(X) \wedge not\, out(X) \end{aligned}$$

The answer sets of $P^{compl}_{WN}$ thus obtained are in one-to-one correspondence with the original AA framework $\langle Arg, att\rangle$, in that the *in* arguments

$$\mathcal{I}(AS) \;=_{\text{def}}\; \{\alpha \in Arg \mid in(\alpha) \in AS\}$$

of each answer set $AS$ of $P^{compl}_{WN}$ is a complete extension of $\langle Arg, att\rangle$.

In the example AA framework of figure 1, there is just one answer set of $P^{compl}_{WN}$: $\{in(a), in(c), out(b)\}$, corresponding to the only complete extension $\{a, c\} = \mathcal{I}(\{in(a), in(c), out(b)\})$ of the original AA framework. In the example

AA framework of figure 2, there is just one answer set of $P_{WN}^{compl}$: $\{undec(a)\}$, corresponding to the only complete extension $\{\} = \mathcal{I}(\{undec(a)\})$ of the original AA framework. In the example AA framework of figure 3, there are three answer sets of $P_{WN}^{compl}$: $\{in(a), out(b)\}$, $\{in(b), out(a)\}$, and $\{undec(a), undec(b)\}$, corresponding to the three complete extensions $\{a\}$, $\{b\}$ and $\{\}$, respectively, of the original AA framework.

In the case of **stable** extensions, the logic program defined by Wakaki and Nitta is $P_{WN}^{stable}$ obtained by extending $P_{WN}^{compl}$ with the clause

$$\leftarrow undec(X)$$

which imposes the further requirement that reinstatement labellings have an empty *undec* component. Thus, in the case of the AA framework of figure 3, there are only two answer sets of $P_{WN}^{stable}$, since $\{undec(a), undec(b)\}$ is not an answer set in this case. In the case of the AA framework of figure 2, there is also no answer set of $P_{WN}^{stable}$, since $\{undec(a)\}$ is no longer an answer set. The answer sets of $P_{WN}^{stable}$ are in one-to-one correspondence with the original AA framework $\langle Arg, att\rangle$, in that the *in* arguments $\mathcal{I}(AS)$ of each answer set $AS$ of $P_{WN}^{stable}$ is a stable extension of $\langle Arg, att\rangle$, similarly to complete extensions.

Caminada [11] has proven that reinstatement labellings with a minimal *in* component, a maximal *in* component, and a minimal *undec* component correspond, respectively, to grounded, preferred and semi-stable extensions. In order to impose these maximality/minimality conditions and obtain logic programs with answer sets corresponding to grounded, preferred and semi-stable extensions, Wakaki and Nitta extend $P_{WN}^{compl}$ to include meta-logic programs to be used to check answer sets of $P_{WN}^{compl}$ (and thus reinstatement labellings). Such answer sets are determined in a 'guess & check' fashion [25]. The meta-logic programs are different for the three notions of extensions, but include a common core $MP_{WN}$ consisting of the following (meta-)clauses

$$m_1(in_t(X)) \leftarrow in(X) \wedge arg(X)$$
$$m_1(undec_t(X)) \leftarrow undec(X) \wedge arg(X)$$

where $in_t(\alpha)$ and $undec_t(\alpha)$ are terms corresponding to atoms $in(\alpha)$ and $undec(\alpha)$ in $P_{WN}^{compl}$ and $m_1$ is a meta-predicate expressing the candidate reinstatement labelling to be checked, as well as (meta-)clauses, for all answer sets $AS$ of $P_{WN}^{compl}$:

$$m_2(in_t(X), \psi(AS)) \leftarrow in(X) \in AS$$
$$m_2(undec_t(X), \psi(AS)) \leftarrow undec(X) \in AS$$

$\psi$ is a function assigning a unique natural number to every answer set of $P_{WN}^{compl}$ and $m_2$ is a meta-predicate expressing alternative reinstatement labellings to be compared with the candidate reinstatement labelling being checked.

Then, $P_{WN}^{pref}$ is $P_{WN}^{compl} \cup MP_{WN}$ extended with

$$\leftarrow d(Z) \wedge not\, c(Z)$$
$$d(\psi(AS)) \leftarrow m_2(in_t(X), \psi(AS)) \wedge not\, m_1(in_t(X))$$
$$c(\psi(AS)) \leftarrow m_1(in_t(X)) \wedge not\, m_2(in_t(X), \psi(AS))$$

$P^{grounded}_{WN}$ is $P^{compl}_{WN} \cup MP_{WN}$ extended with

$$\begin{aligned}
&\leftarrow c(Z) \wedge not\, d(Z)\\
&d(\psi(AS)) \leftarrow m_2(in_t(X), \psi(AS)) \wedge not\, m_1(in_t(X))\\
&c(\psi(AS)) \leftarrow m_1(in_t(X)) \wedge not\, m_2(in_t(X), \psi(AS))
\end{aligned}$$

Finally, $P^{semi}_{WN}$ is $P^{compl}_{WN} \cup MP_{WN}$ extended with

$$\begin{aligned}
&\leftarrow d(Z) \wedge not\, c(Z)\\
&d(\psi(AS)) \leftarrow m_2(undec_t(X), \psi(AS)) \wedge not\, m_1(undec_t(X))\\
&c(\psi(AS)) \leftarrow m_1(undec_t(X)) \wedge not\, m_2(undec_t(X), \psi(AS))
\end{aligned}$$

As in the case of complete and stable extensions, preferred, grounded and semi-stable extensions correspond to the *in* arguments $\mathcal{I}(AS)$ of answer sets $AS$ of the respective logic programs.

### 3.3 Egly, Gaggl and Woltran [23,24]: Conflict-Free, Admissible, Preferred, Stable, Semi-stable, Complete, Grounded Extensions

Egly et al [23] deal with the computation of conflict-free, admissible, preferred, stable, complete, and grounded extensions. Like Wakaki and Nitta [53] summarised in the previous section, an AA framework $\langle Arg, att\rangle$ is first mapped to a set of clauses $P_{\langle Arg,att\rangle}$ to be included in logic programs defined for computing the various notions of extension.

For **conflict-free** extensions, the logic program $P^{cf}_{EGW}$ consists of $P_{\langle Arg,att\rangle}$ together with[4]

$$\begin{aligned}
&\leftarrow in(X) \wedge in(Y) \wedge att(X,Y)\\
&in(X) \leftarrow not\, out(X) \wedge arg(X)\\
&out(X) \leftarrow not\, in(X) \wedge arg(X)
\end{aligned}$$

The answer sets of $P^{cf}_{EGW}$ are in one-to-one correspondence with the conflict-free extensions of the AA framework $\langle Arg, att\rangle$ mapped onto the $P_{\langle Arg,att\rangle}$ component of $P^{cf}_{EGW}$, in the same sense as in [53] (namely that the *in* arguments $\mathcal{I}(AS)$ of the answer sets $AS$ correspond to conflict-free extensions).

A similar correspondence exists for the other kinds of extensions and the answer sets of the logic programs given below.

For **stable** extensions, the logic program $P^{stable}_{EGW}$ consists of $P^{cf}_{EGW}$ and

$$\begin{aligned}
&\leftarrow out(X) \wedge not\, defeated(X)\\
&defeated(X) \leftarrow in(Y) \wedge att(Y,X)
\end{aligned}$$

[4] Note that predicates *in* and *out* here are different from those used in [53] and, in particular, do not refer to the reinstatement labelling of [11].

For **admissible** extensions, the logic program $P^{adm}_{EGW}$ consists of $P^{cf}_{EGW}$ and

$$\begin{aligned}
&\leftarrow in(X) \wedge not\_defended(X)\\
&not\_defended(X) \leftarrow att(Y,X) \wedge not\ defeated(Y)\\
&defeated(X) \leftarrow in(Y) \wedge att(Y,X)
\end{aligned}$$

For **complete** extensions, the logic program $P^{compl}_{EGW}$ consists of $P^{adm}_{EGW}$ and

$$\leftarrow out(X) \wedge not\ not\_defended(X)$$

For **grounded** extensions, the logic program $P^{grounded}_{EGW}$ is obtained by mirroring the characteristic function presentation of this semantics (see section 2.1). The program makes use of an arbitrary ordering $<$ over arguments assumed to be given *a priori*. The program consists of three components. The first component $P^{<}_{EGW}$ uses the given ordering $<$ over arguments to define notions of infimum *inf*, supremum *sup* and successor *succ* over arguments, as follows:

$$\begin{aligned}
&succ(X,Y) \leftarrow lt(X,Y) \wedge not\ nsucc(X,Y)\\
&nsucc(X,Z) \leftarrow lt(X,Y) \wedge lt(Y,Z)\\
&lt(X,Y) \leftarrow arg(X) \wedge arg(Y) \wedge X < Y\\
&inf(X) \leftarrow arg(X) \wedge not\ ninf(X)\\
&ninf(Y) \leftarrow lt(X,Y)\\
&sup(X) \leftarrow arg(X) \wedge not\ nsup(X)\\
&nsup(X) \leftarrow lt(X,Y)
\end{aligned}$$

The second component computes all arguments defended (by all arguments currently *in*) in the layers obtained using *inf*, *sup* and *succ*, as follows:

$$\begin{aligned}
&defended(X) \leftarrow sup(Y) \wedge defended\_up\_to(X,Y)\\
&defended\_up\_to(X,Y) \leftarrow inf(Y) \wedge arg(X) \wedge not\ att(Y,X)\\
&defended\_up\_to(X,Y) \leftarrow inf(Y) \wedge in(Z) \wedge att(Z,Y) \wedge att(Y,X)\\
&defended\_up\_to(X,Y) \leftarrow succ(Z,Y) \wedge defended\_up\_to(X,Z) \wedge not\ att(Y,X)\\
&defended\_up\_to(X,Y) \leftarrow succ(Z,Y) \wedge defended\_up\_to(X,Z) \wedge in(V) \wedge\\
&\qquad\qquad att(V,Y) \wedge att(Y,X)
\end{aligned}$$

The third component of $P^{grounded}_{EGW}$ simply imposes that all defended arguments should be *in*:

$$in(X) \leftarrow defended(X)$$

Further, for **preferred** extensions, $P^{pref}_{EGW}$ is $P^{adm}_{EGW} \cup P^{<}_{EGW}$ extended with a further component incorporating a maximality check on the *in* arguments. This is done by guessing a larger extension with more *in* arguments than the current extension, and checking that this is not admissible, again in a 'guess & check' fashion [25]. Membership in the guessed larger extension is defined using a new predicate *inN* (and corresponding new predicate *outN*). This additional component in $P_{EGW}^{pref}$ is:

$$\begin{aligned}
&\leftarrow \mathit{not\,spoil}\\
&\mathit{spoil} \leftarrow \mathit{eq}\\
&\mathit{eq} \leftarrow \mathit{sup}(Y) \wedge \mathit{eq\_up\_to}(Y)\\
&\mathit{eq\_up\_to}(Y) \leftarrow \mathit{inf}(Y) \wedge \mathit{in}(Y) \wedge \mathit{inN}(Y)\\
&\mathit{eq\_up\_to}(Y) \leftarrow \mathit{inf}(Y) \wedge \mathit{out}(Y) \wedge \mathit{outN}(Y)\\
&\mathit{eq\_up\_to}(Y) \leftarrow \mathit{succ}(Z,Y) \wedge \mathit{in}(Y) \wedge \mathit{inN}(Y) \wedge \mathit{eq\_up\_to}(Z)\\
&\mathit{eq\_up\_to}(Y) \leftarrow \mathit{succ}(Z,Y) \wedge \mathit{out}(Y) \wedge \mathit{outN}(Y) \wedge \mathit{eq\_up\_to}(Z)\\
&\mathit{spoil} \leftarrow \mathit{inN}(X) \wedge \mathit{inN}(Y) \wedge \mathit{att}(X,Y)\\
&\mathit{spoil} \leftarrow \mathit{inN}(X) \wedge \mathit{outN}(Y) \wedge \mathit{att}(Y,X) \wedge \mathit{undefeated}(Y)\\
&\mathit{undefeated}(X) \leftarrow \mathit{sup}(Y) \wedge \mathit{undefeated\_up\_to}(X,Y)\\
&\mathit{undefeated\_up\_to}(X,Y) \leftarrow \mathit{inf}(Y) \wedge \mathit{outN}(X) \wedge \mathit{outN}(Y)\\
&\mathit{undefeated\_up\_to}(X,Y) \leftarrow \mathit{inf}(Y) \wedge \mathit{outN}(X) \wedge \mathit{not\,att}(Y,X)\\
&\mathit{undefeated\_up\_to}(X,Y) \leftarrow \mathit{succ}(Z,Y) \wedge \mathit{undefeated\_up\_to}(X,Z) \wedge \mathit{outN}(Y)\\
&\mathit{undefeated\_up\_to}(X,Y) \leftarrow \mathit{succ}(Z,Y) \wedge \mathit{undefeated\_up\_to}(X,Z) \wedge \mathit{not\,att}(Y,X)\\
&\mathit{inN}(X) \leftarrow \mathit{spoil} \wedge \mathit{arg}(X)\\
&\mathit{outN}(X) \leftarrow \mathit{spoil} \wedge \mathit{arg}(X)
\end{aligned}$$

Finally, for **semi-stable** extensions, Egly et al define $P_{EGW}^{semi}$ as a variant of $P_{EGW}^{pref}$ (see [24] for details).

### 3.4 Faber and Woltran [27]: Ideal Extensions

Faber and Woltran present an encoding of the computation of ideal extensions into so-called *manifold* answer set programs [27]. These programs allow various forms of meta-reasoning to be implemented within ASP, including credulous and sceptical reasoning. The manifold answer set program used for computing ideal extensions follows the algorithm of [21], which works as follows.

- Let *adm* be the set of all the admissible extensions of a given argumentation framework $\langle Arg, att\rangle$.
- Let $X^- = Arg \setminus \bigcup_{S\in adm} S$.
- Let $X^+ = \{\alpha \in Arg \mid \forall\beta,\gamma \in Arg : (\beta,\alpha),(\alpha,\gamma) \in att \Rightarrow \beta,\gamma \in X^-\} \setminus X^-$.
- Let $\langle Arg^*, att^*\rangle$ be the argumentation framework with $Arg^* = X^+ \cup X^-$ and $att^* = att \cap \{(\alpha,\beta),(\beta,\alpha) \mid \alpha \in X^+, \beta \in X^-\}$.
- Let $adm^*$ be the set of all admissible extensions of $\langle Arg^*, att^*\rangle$.

Then, the ideal extension of $\langle Arg, att\rangle$ is $\bigcup_{S\in adm^*} S \cap X^+$.

The admissible extensions of $(Arg^*, att^*)$ can be computed using a fixpoint iteration (which can be done in polynomial time since this argumentation framework is *bipartite* [21]). At the first iteration, $X_1$ is generated by eliminating all arguments in $Arg^*$ that are attacked by unattacked arguments. At the second iteration, $X_2$ is $X_1$ minus all arguments that are attacked by arguments unattacked by $X_1$, and so on, until no more arguments can be eliminated (after at most $|X^+|$ iterations).

The logic program whose answer sets correspond to ideal extensions is obtained by using the manifold for credulous reasoning of the logic program for admissible extensions given by [23], further extended to identify $(Arg^*, att^*)$ and to simulate the fixpoint algorithm outlined above. Details of this logic program can be found in [27].

## 3.5 DLV for ASP for Abstract Argumentation

All approaches described above have been implemented using the DLV ASP solver.

For the method of [39] (section 3.1) DLV can be used to perform credulous and sceptical reasoning under preferred extensions as follows. Given the logic program $P^{pref}_{NCO}$ for an AA framework $(Arg, att)$ and the query $\alpha$? for an argument $\alpha \in Arg$, DLV used in `-brave` mode determines whether $\alpha$ belongs to a preferred extension of $(Arg, att)$, and in `-cautious` mode whether $\alpha$ belongs to all preferred extensions of $(Arg, att)$.

DLV is employed in a system[5] that can perform credulous and sceptical reasoning under the various semantics considered by [53] (section 3.2 above). DLV is also the core of the ASPARTIX system[6] [23,22]. This system supports the computation of admissible, stable, complete, grounded, preferred and ideal extensions, following the work of [23,27] (see section 3.3 above), as well as semi-stable extensions [12] and cf2 extensions [3] (see section 3.6 below) following encodings given in [24].

## 3.6 ASP for Other Forms of Argumentation

ASP has been used as a computational tool for abstract argumentation under other semantics, notably the cf2 extensions semantics [3], as well as for forms of argumentation other than abstract argumentation. In particular:

– Thimm and Kern-Isberner [48] present mappings of the DeLP argumentation framework [30] onto ASP.
– Egly et al [23] define mappings for value-based argumentation [5], a form preference-based abstract argumentation [1] and bipolar argumentation [2].
– Wakaki and Nitta [54] use ASP for computing extensions of a form of abductive argumentation they define.
– Devred et al [15] use ASP to compute extensions of abstract argumentation frameworks extended with constraints in the form of propositional formulas. Their mapping of constrained argumentation onto ASP uses lists and, as

[5] `http://www.ailab.se.shibaura-it.ac.jp/compARG.html`

[6] `http://rull.dbai.tuwien.ac.at:8080/ASPARTIX/`

a consequence, only ASP solvers capable of dealing with lists can be used with the outcome of this mapping. These include DLV-complex[7] [10] and ASPerIX[8] [35].
- Gaggl and Woltran [29] and Egly et al [24] propose encodings for abstract argumentation under the cf2 extensions semantics [3]. These are incorporated within the ASPARTIX system (see section 3.5).
- Osorio et al [40] propose an alternative mapping for abstract argumentation under the cf2 extensions semantics and use the DLV system to compute these extensions under the given mapping.

## 4 Some Future Directions

The approaches presented in section 3 employ the DLV system as the underlying ASP solver of choice. The approach of [39] (section 3.1) makes use of the capability of DLV to deal with disjunctive clauses and deploy the `brave` and `cautious` modes of DLV. The approach of [23,24] (section 3.3) makes use of the $<$ ordering that is built into DLV. The approach of [53] (section 3.2) uses the `brave` and `cautious` modes of DLV. It would be interesting to see whether other ASP solvers, e.g. the clasp and claspD (for disjunctive clauses)[9] solvers, could be beneficially used to support the computation of extensions and query answering in abstract argumentation. It would also be interesting to perform a comparative performance analysis of the methods presented in section 3 to identify the most efficient method to support various kinds of applications.

With the exception of a few works (see section 3.6) ASP has been deployed to support *abstract* argumentation. However, the majority of applications of argumentation, e.g. medical decision making [28] and legal reasoning [4], require concrete argumentation frameworks, where arguments and attacks are built from knowledge bases of rules and facts and are constructed on demand, that is to say, as determined by the needs of a given query/claim to be argued for or against. In particular, there are a number of approaches to argumentation based on logic programming (e.g [44,47,42]), extending logic programming (e.g. assumption-based argumentation [9,8,17]), or based upon classical logic (e.g. [7]). It would be useful to see whether these other forms of argumentation could be fruitfully computed using ASP.

Evidently this can always be done in the sense that these concrete forms of argumentation are (usually) also instances of abstract argumentation frameworks: having generated the relevant arguments and attacks, one could compute the extensions of the resulting abstract argumentation framework using any of the methods described in previous sections. What we have in mind however is rather different. What we would really like is a mapping from the rule base $R$ used to construct arguments and attacks to a logic program $P$ used to compute extensions, but one which retains some well defined correspondence between the argument-generating rules in $R$ and (some of) the clauses in $P$.

[7] `http://www.mat.unical.it/dlv-complex`

[8] `http://www.info.univ-angers.fr/pub/claire/asperix/`

[9] `http://potassco.sourceforge.net/`

As has often been remarked, ASP computations are oriented primarily towards the generation of models (answer sets) rather than to proofs or chains of reasoning as in other forms of logic programming. In several of the existing applications of argumentation, however, it is precisely the explanation/justification of answers to queries, in the form of arguments for and against a particular claim, that matters more than the answers themselves. This is an essential feature, for example, when argumentation is used in a collaborative setting, e.g. as in [46,13], where arguments are constructed and evaluated across agents with possibly different knowledge bases, and explanations serve to inform and share information. In applications concerned with the (internal) resolution of dilemmas, conflicts between defeasible rules of conduct, decisions about vague or uncertain outcomes, it is often the explanations/justifications that are of primary interest. In an argumentation setting, explanations/justifications take the form of a *dialectical structure* which presents the relevant arguments, identifies the attacks between them, possibly classifying them into different types, and provides some representation of how arguments are defended against attacks. The specific details vary according to the concrete form of argumentation employed and the chosen dialectical semantics.

It would be extremely valuable to investigate to what extent dialectical explanations could be meaningfully extracted from extensions computed by means of ASP. A step in this direction has been made with the ASPARTIX system (see section 3.5) which includes a graphical interface labelling nodes of argumentation graphs to provide a visualisation of abstract argumentation frameworks. In general, however, and especially when the graph is large, only relevant parts of the graph should be presented (as the presentation of full extensions tends to obscure matters).

The survey presented in this paper was motivated in part by an interest in how argumentation frameworks could be extended with *priorities* (also referred to as 'preferences' in the literature). We are interested in how priorities (relative strengths of rules and arguments) could be used in the resolution of conflicts between defeasible rules of belief, and between defeasible rules of conduct in practical reasoning. There is reason to think that these forms of reasoning, though similar, might nevertheless be different in some important respects. See, for example, the recent discussion in [41] (though that is not presented in argumentation terms). There is some existing work on incorporating forms of priority/preference in (non-abstract) argumentation. See for example [44,34,49,51,50]. It is fair to say that it is still fragmented, however, certainly when compared to the long standing investigations of priorities/preferences in nonmonotonic reasoning. See [14] for a survey and classification of approaches, including the computation of answer sets. As observed in [14], a major difficulty in adding a treatment of priority into a (rule-based) argumentation framework lies in defining a suitable ordering on the strength of arguments based on a priority ordering of the rules from which they are constructed. It may be that it is better to approach the problem by ordering the strengths of attacks instead of the strengths of arguments, or some combination of the two. These questions remain to be investigated, as well as possible relationships to existing work on priorities/preferences in ASP. There is then the further issue, essential for the kind of applications we have in mind, of finding some way of extracting dialectical explanations from the computed extensions.

## 5 Conclusions

Argumentation and answer set programming are two of the main knowledge representation paradigms that have emerged from logic programming for non-monotonic reasoning.

We have surveyed a number of existing approaches to using ASP for computing extensions in argumentation. The majority of these approaches focus on abstract argumentation, and rely upon mapping abstract argumentation frameworks onto logic programs whose answer sets correspond to (various kinds of) extensions for abstract argumentation. We have seen that approaches exist for computing the majority of existing notions of extensions (including conflict-free, admissible, stable, preferred, complete, semi-stable, and ideal extensions). All presented approaches have been implemented in DLV.

We have also indicated some possible directions for future research on using ASP for argumentation, including: (i) deploying ASP solvers other than DLV, e.g. claspD, (ii) considering concrete (rather than abstract) argumentation frameworks in support of applications, and (iii) developing methods for explaining answers to queries (claims) in dialectical terms, drawing from relevant parts of answer sets corresponding to extensions where the claims hold true.

## Acknowledgements

We thank Luis Moniz Pereira for useful comments on an earlier version of this paper.

## References

1. Amgoud, L., Cayrol, C.: A reasoning model based on the production of acceptable arguments. Annals of Mathematics and Artificial Intelligence 34(1-3), 197–215 (2002)
2. Amgoud, L., Cayrol, C., Lagasquie-Schiex, M.-C., Livet, P.: On bipolarity in argumentation frameworks. International Journal of Intelligent Systems 23(10), 1062–1093 (2008)
3. Baroni, P., Giacomin, M., Guida, G.: SCC-recursiveness: a general schema for argumentation semantics. Artificial Intelligence 168(1-2), 162–210 (2005)
4. Bench-Capon, T., Prakken, H., Sartor, G.: Argumentation in legal reasoning. In: Rahwan, I., Simari, G. (eds.) Argumentation in Artificial Intelligence, pp. 363–382. Springer, Heidelberg (2009)
5. Bench-Capon, T.J.M.: Persuasion in practical argument using value-based argumentation frameworks. Journal of Logic and Computation 13(3), 429–448 (2003)

6. Besnard, P., Doutre, S.: Characterization of semantics for argument systems. In: Dubois, D., Welty, C.A., Williams, M.-A. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR 2004), pp. 183–193. AAAI Press, Menlo Park (2004)
7. Besnard, P., Hunter, A.: Elements of Argumentation. MIT Press, Cambridge (2008)
8. Bondarenko, A., Dung, P., Kowalski, R., Toni, F.: An abstract, argumentation-theoretic approach to default reasoning. Artificial Intelligence 93(1-2), 63–101 (1997)
9. Bondarenko, A., Toni, F., Kowalski, R.: An assumption-based framework for non-monotonic reasoning. In: Nerode, A., Pereira, L. (eds.) Proc. 2nd International Workshop on Logic Programming and Non-monotonic Reasoning, pp. 171–189. MIT Press, Cambridge (1993)
10. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in asp: Theory and implementation. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 407–424. Springer, Heidelberg (2008)
11. Caminada, M.: On the issue of reinstatement in argumentation. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 111–123. Springer, Heidelberg (2006)
12. Caminada, M.: Semi-stable semantics. In: Dunne, P.E., Bench-Capon, T.J.M. (eds.) Proceedings of the Second International Conference on Computational Models of Argument (COMMA 2006). Frontiers in Artificial Intelligence and Applications, vol. 144, pp. 121–130. IOS Press, Amsterdam (2006)
13. de Almeida, I.C., Alferes, J.J.: An argumentation-based negotiation for distributed extended logic programs. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 191–210. Springer, Heidelberg (2007)
14. Delgrande, J., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence 20(2), 308–334 (2004)
15. Devred, C., Doutre, S., Lefèvre, C., Nicolas, P.: Dialectical proofs for constrained argumentation. In: Baroni, P., Cerutti, F., Giacomin, M., Simari, G. (eds.) Proceedings of the Third International Conference on Computational Models of Argument (COMMA 2010), vol. 216. IOS Press, Amsterdam (2010)
16. Dung, P.: On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. Artificial Intelligence 77, 321–357 (1995)
17. Dung, P., Kowalski, R., Toni, F.: Assumption-based argumentation. In: Rahwan, I., Simari, G. (eds.) Argumentation in Artificial Intelligence, pp. 199–218. Springer, Heidelberg (2009)
18. Dung, P., Mancarella, P., Toni, F.: Computing ideal sceptical argumentation. Artificial Intelligence, Special Issue on Argumentation in Artificial Intelligence 171(10-15), 642–674 (2007)
19. Dung, P.M.: Negations as hypotheses: An abductive foundation for logic programming. In: Proceedings of 8th International Conference on Logic Programming, ICLP 1991, pp. 3–17 (1991)
20. Dung, P.M., Thang, P.M.: A unified framework for representation and development of dialectical proof procedures in argumentation. In: Boutilier, C. (ed.) Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, pp. 746–751 (2009)
21. Dunne, P.E.: The computational complexity of ideal semantics. Artificial Intelligence 173(18), 1559–1591 (2009)

22. Egly, U., Gaggl, S.A., Wandl, P., Woltran, S.: ASPARTIX conquers the web. In: Baroni, P., Giacomin, M., Simari, G. (eds.) Proceedings of the Second International Conference on Computational Models of Argument (COMMA 2010). Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam (2010)
23. Egly, U., Gaggl, S.A., Woltran, S.: ASPARTIX: Implementing argumentation frameworks using answer-set programming. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 734–738. Springer, Heidelberg (2008)
24. Egly, U., Gaggl, S.A., Woltran, S.: Answer-set programming encodings for argumentation frameworks. In: Argument and Computation (2010) (accepted for publication)
25. Eiter, T., Polleres, A.: Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. Theory and Practice of Logic Programming 6(1-2), 23–60 (2006)
26. Eshghi, K., Kowalski, R.A.: Abduction compared with negation by failure. In: Proceedings of 6th International Conference on Logic Programming, ICLP 1989, pp. 234–254 (1989)
27. Faber, W., Woltran, S.: Manifold answer-set programs for meta-reasoning. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 115–128. Springer, Heidelberg (2009)
28. Fox, J., Glasspool, D., Grecu, D., Modgil, S., South, M., Patkar, V.: Argumentation-based inference and decision making–a medical perspective. IEEE Intelligent Systems 22(6), 34–41 (2007)
29. Gaggl, S., Woltran, S.: CF2 semantics revisited. In: Baroni, P., Cerutti, F., Giacomin, M., Simari, G. (eds.) Proceedings of the Third International Conference on Computational Models of Argument (COMMA 2010), vol. 216, pp. 243–254. IOS Press, Amsterdam (2010)
30. Garcia, A., Simari, G.: Defeasible logic programming: An argumentative approach. Theory and Practice of Logic Programming 4(1-2), 95–138 (2004)
31. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, ch. 7, pp. 285–316. Elsevier, Amsterdam (2007)
32. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the Fifth International Conference and Symposium Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
33. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3/4), 365–386 (1991)
34. Kowalski, R.A., Toni, F.: Abstract argumentation. Journal of Artificial Intelligence and Law, Special Issue on Logical Models of Argumentation 4(3-4), 275–296 (1996)
35. 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)
36. McCarthy, J.: Circumscription—a form of non-monotonic reasoning. Artificial Intelligence 13, 27–39 (1980)
37. McDermott, D.V.: Nonmonotonic logic ii: Nonmonotonic modal theories. J. ACM 29(1), 33–57 (1982)
38. Moore, R.C.: Semantical considerations on nonmonotonic logic. Artif. Intell. 25(1), 75–94 (1985)
39. Nieves, J.C., Cortés, U., Osorio, M.: Preferred extensions as stable models. TPLP 8(4), 527–543 (2008)

40. Osorio, M., Nieves, J.C., Gómez-Sebastià, I.: CF2-extensions as answer-set models. In: Baroni, P., Cerutti, F., Giacomin, M., Simari, G. (eds.) Proceedings of the Third International Conference on Computational Models of Argument (COMMA 2010), vol. 216. IOS Press, Amsterdam (2010)
41. Parent, X.: Moral particularism and deontic logic. In: Governatori, G., Sartor, G. (eds.) DEON 2010. LNCS, vol. 6181, pp. 84–97. Springer, Heidelberg (2010)
42. Pereira, L.M., Pinto, A.M.: Approved models for normal logic programs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 454–468. Springer, Heidelberg (2007)
43. Poole, D.: A logical framework for default reasoning. Artificial Intelligence 36(1), 27–47 (1988)
44. Prakken, H., Sartor, G.: Argument-based extended logic programming with defeasible priorities. Journal of Applied Non-classical Logics 7, 25–75 (1997)
45. Reiter, R.: A logic for default reasoning. Artificial Intelligence 13(1-2), 81–132 (1980)
46. Schroeder, M., Schweimeier, R.: Fuzzy argumentation for negotiating agents. In: Proceedings of the First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, pp. 942–943. ACM, New York (2002)
47. Schweimeier, R., Schroeder, M.: A parameterised hierarchy of argumentation semantics for extended logic programming and its application to the well-founded semantics. Theory and Practice of Logic Programming 5(1-2), 207–242 (2005)
48. Thimm, M., Kern-Isberner, G.: On the relationship of defeasible argumentation and answer set programming. In: Besnard, P., Doutre, S., Hunter, A. (eds.) Proceedings of the Second International Conference on Computational Models of Argument (COMMA 2008). Frontiers in Artificial Intelligence and Applications, vol. 172, pp. 393–404. IOS Press, Amsterdam (2008)
49. Toni, F.: Assumption-based argumentation for closed and consistent defeasible reasoning. In: Satoh, K., Inokuchi, A., Nagao, K., Kawamura, T. (eds.) JSAI 2007. LNCS (LNAI), vol. 4914, pp. 390–402. Springer, Heidelberg (2008)
50. Toni, F.: Assumption-based argumentation for epistemic and practical reasoning. In: Casanovas, P., Sartor, G., Casellas, N., Rubino, R. (eds.) Computable Models of the Law. LNCS (LNAI), vol. 4884, pp. 185–202. Springer, Heidelberg (2008)
51. Toni, F.: Assumption-based argumentation for selection and composition of services. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 231–247. Springer, Heidelberg (2008)
52. Van Gelder, A., Ross, K., Schlifp, J.: The well-founded semantics for general logic programs. Journal of ACM 38(3), 620–650 (1991)
53. Wakaki, T., Nitta, K.: Computing argumentation semantics in answer set programming. In: Hattori, H., Kawamura, T., Idé, T., Yokoo, M., Murakami, Y. (eds.) JSAI 2008. LNCS, vol. 5447, pp. 254–269. Springer, Heidelberg (2009)
54. Wakaki, T., Nitta, K., Sawamura, H.: Computing abductive argumentation in answer set programming. In: McBurney, P., Rahwan, I., Parsons, S., Maudet, N. (eds.) ArgMAS 2009. LNCS, vol. 6057, pp. 195–215. Springer, Heidelberg (2010)

# Cantor's Paradise Regained: Constructive Mathematics from Brouwer to Kolmogorov to Gelfond

Vladik Kreinovich

Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968 USA
vladik@utep.edu

**Abstract.** Constructive mathematics, mathematics in which the existence of an object means that that we can actually construct this object, started as a heavily restricted version of mathematics, a version in which many commonly used mathematical techniques (like the Law of Excluded Middle) were forbidden to maintain constructivity. Eventually, it turned out that not only constructive mathematics is not a weakened version of the classical one – as it was originally perceived – but that, vice versa, classical mathematics can be viewed as a particular (thus, weaker) case of the constructive one. Crucial results in this direction were obtained by M. Gelfond in the 1970s. In this paper, we mention the history of these results, and show how these results affected constructive mathematics, how they led to new algorithms, and how they affected the current activity in logic programming-related research.

**Keywords:** constructive mathematics; logic programming; algorithms.

*Science and engineering: a brief reminder.* One of the main objectives of *science* is to find out how the world operates, to be able to predict what will happen in the future. Science predicts the future positions of celestial bodies, the future location of a spaceship, etc.

From the practical viewpoint, it is important not only to passively predict what will happen, but also to decide what to do in order to achieve certain goals. Roughly speaking, decisions of this type correspond not to science but to *engineering*. For example, once we have come up with the design of a spaceship, once we know how exactly it will be launched, Newton's equations can predict where exactly it will be at any future moment of time – and if our goal is to reach the Moon, whether it will reach it or not. If we happen to stumble upon the design and initial conditions that satisfy our goal, great, but in practice, such situations are rare. Usually, we need to *find* the initial conditions (and the design) for which the spaceship will safely reach the Moon.

Similarly, once the bridge is designed, we can use the known equations of mechanics to predict its stability and vulnerability to winds. However, a more important problem is to design a bridge that will withstand the expected loads under prevailing winds.

*Resulting need for constructive mathematics.* The ultimate objective of a scientific analysis is to formulate an exact mathematical model of the corresponding physical phenomena. Once the corresponding physical model is formulated in precise mathematical terms, the practical (engineering) problem can be also formulated in mathematical terms: find an object $x$ (design, trajectory, etc.) that satisfies the given precisely formulated condition $P(x)$.

At first glance, it may seem that similar statement exist in mathematics: for example, mathematicians can prove statements of the type $\exists x\, P(x)$. Intuitively, to claim to prove that something exists often means to actually construct the corresponding object, Indeed, for many centuries, most mathematical proofs of the existence statements were based on the actual construction. Once in a while there were proofs from contradiction, but they were rare. The situation changed drastically at the end of the 19th century. The first important result for which an existence proof was provided without an explicit construction was David Hilbert's 1888 proof of the Finite Basis Theorem [10, 16–18]. This proof answered an important question (raised by a mathematician Paul Gordan from Göttingen) about the invariants of homogeneous polynomials. A polynomial $P(x_1, \ldots, x_n)$ is called *homogeneous* if all its monomials are of the same total degree $d$: e.g., we can have $\sum\limits_{i=1}^{n} a_i \cdot x_i$ or $\sum\limits_{i,j} a_{ij} \cdot x_i \cdot x_j$, with $a_{ij} = a_{ji}$. A polynomial function $f(a_\alpha)$ of the coefficients $a_\alpha$ of this polynomial is called *invariant* it for every linear transformation of the unknowns $x_i \to x'_i \stackrel{\text{def}}{=} \sum c_{ij} \cdot x_j$, this function changes by a multiplicative constant $f(a'_\alpha) = \lambda(\{c_{ij}\}) \cdot f(a_\alpha)$. For example, for quadratic forms of two variables $a_{11} \cdot x_1^2 + 2a_{12} \cdot x_1 \cdot x_2 + a_{22} \cdot x_2^2$, a discriminant $D = a_{11} \cdot a_{22} - a_{12}^2$ is an invariant. It was known that for the case of $n \le 8$ variables, for every degree $d$, invariants have a finite basis in the sense that we can select finitely many invariants so that every other invariant is a polynomial of the selected ones. For example, for $n = 2$ and $d = 2$, every invariant is a power of the discriminant $D$. Hilbert proved that such a finite basis exists for all $n$ and all $d$, but – in contrast to the previously known proofs for $n \le 8$ – his by-contradiction proof did not provide any actual construction of the corresponding finite set. After reading this proof, Paul Gordan himself said: "This is not mathematics; it is theology"; see, e.g., [33].

Hilbert himself provided later a constructive proof of this result [19, 20], but the floodgates were opened for non-constructive proofs. After Hilbert's theorem, came numerous such proofs, including proofs from Cantor's set theory. The most well-known of them was the diagonal proof that there exist irrational and transcendental numbers – the "father" of all the modern diagonalization proofs.

Indirect proofs of existence became fully accepted in the mainstream mathematics – because, strictly speaking, the existence of an object does not mean that we must be able to actually construct it. The famous topologist L. E. J. Brouwer agreed with this statement – that the existence in traditional mathematics does not imply constructibility – but he made a different conclusion: since we *do* need to construct objects, we must hence change the meaning of the existential quantifier in mathematics. He therefore proposed a new mathematics

– what we now call *constructive mathematics* – in which the only way to prove a statement $\exists x\, P(x)$ is to produce a construction of an object $x$ for which $P(x)$ is true; see, e.g., [7, 8]. Brouwer argued that this new understanding of the existential quantifiers is in better accordance with our intuition – at least with the intuition of applied mathematicians who want, e.g., to *solve* equations, not just "prove" that the solution exists. Because of this argument, he called his approach to mathematics *intuitionism* – instead of blindly following formal constructions, even when we start deviating of our intuitive meaning of the corresponding notions, we should also listen to our intuition when formal constructions lead us astray.

To illustrate Brouwer's point, it is actually not necessary to consider the case when we have infinitely many possible objects $x$. The same idea can be illustrated on the example of a simple disjunction $A \vee B$, when we have only two alternatives $A$ and $B$. For example, in classical mathematics, the Law of Excluded Middle holds, according to which every statement is either true or false $A \vee \neg A$. However, a construction interpretation of a disjunction $P \vee Q$ means that we know either $P$ or $Q$. Since we do not know the truth value of a generic statement $A$, in intuitionistic logic, the Law of Excluded Middle is not generally true.

*First years of constructive mathematics: constructive mathematics as a straightjacket.* Several mathematicians whose interests were close to applications agreed with (at least some of) Brouwer's ideas, the most famous of them Hermann Weyl, whose interest in application of mathematics to space-time has led him to interesting intuitionistic ideas [35].

However, such converts were rare. The main reason for this rarity is that to avoid non-constructive existence proofs, Brouwer proposed to restrict allowed logical constructions – so that proofs by contradiction become impossible. These restrictions severely limited the ability of mathematicians to prove new results. Not surprisingly, most mathematicians did not want to place themselves under such severe restrictions. The general opinion was best expressed by David Hilbert himself. By then, he has become the leading mathematician of that time. In 1900, he was tasked, by the world mathematics community, to prepare the list of most important problem that the 19 century mathematics should leave for the 20 century mathematicians to solve.

In regards to constructive mathematics, Hilbert famously said: "No one shall drive us from the paradise that Cantor has created for us."

*Kolmogorov: it is the classical logic that is a straightjacket, not the constructive one.* For several years, constructive mathematics and the related constructive logic were viewed as severely restricted version of the traditional ones. In logic, this view changed drastically in 1925, when the famous Russian mathematician A. N. Kolmogorov showed that classical logic can be interpreted as a subset of the constructive one. To perform such a translation, we need to interpret each classical statement $A$ as a double negation $\neg\neg A$. Correspondingly, e.g., a classical disjunction $A \vee B$ must be interpreted as $\neg\neg(\neg\neg A \vee \neg\neg B)$, etc.

Because of Kolmogorov's result, the constructive logic was no longer perceived as a poor, limited version of the classical one: vice versa, the classical logic is a particular case of the constructive one. Thus, the constructive logic was shown to be richer and more versatile than the classical one: it allowed, e.g., in addition to the "classical" disjunction $\neg\neg(\neg\neg A \vee \neg\neg B)$, to also consider a different "constructive" disjunction $A \vee B$.

Constructive logic became a widely used, widely studied, and well-respected part of logic. But not yet constructive mathematics.

*From the ubiquity of constructive logic to the ubiquity of constructive mathematics: Gelfond's groundbreaking results.* In the beginning, constructive mathematics was perceived as a kind of a limited version of the classical one – the tools are limited and thus, was the argument, it is understandable that the results are limited. Even constructive mathematicians themselves originally believed in this tradeoff: yes, our theorems are not as plentiful and not as sophisticated, but they are deeper: every time we prove existence, we have a construction. Of course, the need to provide constructions limits us – but makes our results more useful in applications.

As more and more research was done in constructive mathematics, more and more results became constructively proven – and these results became more and more sophisticated. The real breakthrough came with the 1967 book of E. Bishop, a classical mathematician who "saw the light" of constructive ideas and transformed a large portion of basic math into constructive language [3].

With all these results, it became clearer that for every mathematical statement that does not explicitly contain a construction, classical truth implies constructive truth. Clearer, but still, every time, we needed to re-do the original classical proof by meticulously avoiding constructively "non-kosher" ideas (like the Law of Excluded Middle) that might have been used in the original classical proof. This re-doing was taking a lot of time and effort. A general, "meta"-result of this type was badly needed to save all this time. And this general result was proven by M. G. Gelfond in his groundbreaking papers [11–13].

Similarly to the way Kolmogorov interpreted a classical logical statement $A$ as $\neg\neg A$ and each logical connective $A \odot B$ as $\neg\neg(\neg\neg A \odot \neg\neg B)$, Gelfond interpreted classical real numbers (= classically converging sequences of integers) in the constructive terms. Specifically, a classical sequence $r_n$, which can be equivalently reformulated as a predicate $P(n, r)$ for which for every $n$, there is exactly one $r$ satisfying this property ($\forall n \exists! r\, P(r, n)$, where ! stands for uniqueness) was constructivized into a logical property $P(n, r)$ for which $\forall n \neg\neg\exists! r\, P(n, r)$ (a *filling* in the sense of [32]). This enabled Michael Gelfond to prove a meta-result about real numbers, continuous functions, etc.

Later, with V. Lifschitz, they extended this result to a large portion of set theory – and thus working mathematics – by designing a constructive version of set theory that allowed high-level constructions similar to the classical set theories like ZF.

Because of their results, the constructive mathematics can no longer be perceived as a poor, limited version of the classical one: vice versa, the classical mathematics is a particular (weaker) case of the constructive one. The constructive mathematics was shown to be richer and more versatile than the classical one: it allowed, e.g., in addition to the "classical" non-constructive existential quantifier, to also consider a different "constructive" one.

Cantor's paradise was regained – and an even better constructive one was built on top of it.

*How these exciting results influenced my own research.* In the early 1970s, when Michael Gelfond and Vladimir Lifschitz developed their excising results in St. Petersburg, Russia, I was a student attending the logical seminars where their presented different stage of their research – and attending seminars led by them, e.g., the seminar on set theory and its possible constructivization. I learned from them (and they gave me an official A for their seminar :-), I ran my ideas by them, they helped me present my ideas and edit my papers.

My interest at that time was in using Gelfond's theorem to prove new results about constructive existence. I concentrated on three related directions.

First, since it is well known that it is not possible, in general, to compute the exact solutions to a system of equation or the exact location of a maximizing point, in practice – since measurements and implementations are approximate anyway – we only need $\varepsilon$-approximate solutions, for an appropriate accuracy $\varepsilon$. For many practical problems like solving systems of equations and finding locations of maxima, the algorithmic computability of such approximate solutions was well known. I used Gelfond's theorem to extend these results to more general problems involving integration, minimization, and maximization.

My second direction was related to the fact that the proofs of most algorithmic non-computability results essentially use functions which have several maxima and/or equations which have several solutions. It turned out that this is not an accident: uniqueness actually implies algorithmic computability. Such a result was first proven by Lacombe [31] who designed an algorithm that inputs a constructive function of one or several real variables on a bounded set that attains its maximum on this set at exactly one point – and computes this global maximum point. V. Lifschitz extended this result to constructive functions on general constructive compact spaces [32].

In my paper [25] and in my dissertation [27], I showed how this result can be applied to design many algorithms: from optimal approximation of functions to designing a convex body from its metric to constructive a shortest path in a curved space to designing a Riemannian space most tightly enclosing unit spheres in a given Finsler space [9].

The third direction was to search for other classes of mathematical statements for which classical validity automatically implies their constructive validity. I found a few such classes (e.g., functions attaining maxima at two points, with a known lower bound on the distance between them), but mostly, the results turned out to be negative. For example, Lacombe-Lifschitz-style algorithm is not possible for functions that have exactly two global maxima, not possible for functions on non-compact sets, etc. [25–29].

In the Appendix, we provide an informal explanation of what is so special about uniqueness, and how classes like this can be constructed.

*Constructive mathematics after Gelfond: very briefly.* Since the 1970s, many interesting results were constructively proved within constructive mathematics; see, e.g., [1, 2, 4–6, 29, 30, 34]. It is impossible to enumerate all such results in this short paper. Let us just mention that the uniqueness-implies-computability results were effectively used by Kohlenbach and his school to come up with new *efficient* algorithms; see, e.g., [21–23].

*Gelfond's research after constructive mathematics.* While I was working on Michael Gelfond's constructive ideas, Michael himself moved further, in the direction that very naturally follows from his constructive research. Indeed, as we have mentioned earlier, constructive mathematics started with an observation that the meaning of disjunction $A \vee B$ ("$A$ or $B$") in mathematics is sometimes different from the intuitive meaning of "or". A thorough analysis of this distinction led to the realization that to adequately represent our intuition about "or", we need (at least) two *different* disjunction operations: the constructive disjunction and the classical disjunction.

This distinction – and constructive mathematics in general – got a great boost in the 1950s and 1960s, when computers became ubiquitous and construction became not just a theoretical concept but rather an everyday task. This boost led to the urgent need for translating the existing knowledge into algorithmic computer-understandable terms. The experience of such translation showed that "or" is not the only logical connective whose intuitive and mathematical meanings sometime differ; the same turned out to be for other connectives as well.

This realization started with the implication $A \rightarrow B$ ("$A$ implies $B$"). In classical mathematics, $A \rightarrow B$ simply means that either $B$ is true or $A$ is false. As a result, in mathematics, statements like "if $2 + 2 = 5$ then witches can fly" make perfect sense. From the intuitive viewpoint, however, "$A$ implies $B$" means that the statement $A$ was actually used in proving the statement $B$ – i.e., crudely speaking, that the statement $B$ would be not true without $A$. This intuitive meaning of implication was partly captured by if-then rules of *logic programming.*

For logic programs without negation, the intuitive meaning of commonsense implication has been captured by the notion of a *minimal model*, i.e., a model in which the smallest possible set of atoms is true. This means, in particular, that if $A$ is true, then either it was assumed to be true from the very beginning, or it follows from a rule of the type $A \leftarrow B, C, \ldots$ in which all the conditions $B$, $C$, etc. are already proven to be true. However, until the late 1980s, there was no similar intuitively clear semantics of logic program with negation. Such a semantics – called *stable semantics* – was provided by M. Gelfond and V. Lifschitz in their 1988 paper [14]. This semantics – in essence, formalizing the above intuitive meaning of implication – indeed proved to be very adequate in capturing the intuitive meaning of if-then rules, and thus, in transforming abstract commonsense and expert knowledge into a set of constructive rules – rules enabling us to algorithmically solve problems.

After the success with implication, came another realization: that the mathematical and intuitive meanings of negation are also slightly different, and that in order to capture this intuitive meaning, we need two *different* negations: classical negation (meaning that $A$ is known to be false) and "negation as failure" (meaning that we cannot prove $A$) [15].

This was just the beginning. Then came the use of disjunctions in logic programming, the use of sets, and – slowly but surely – we seem to be arriving at a situation where classical mathematics will become a particular case of this generalized "logic programming" – like it became, in effect, a particular case of constructive mathematics.

And when we reach this stage – sometime in the next century – then no one shall drive us from the paradise that Brouwer, Kolmogorov, Gelfond, Lifschitz, and others have created for us!

*Acknowledgments.* This work was supported in part by the National Science Foundation grants HRD-0734825 and DUE-0926721 and by Grant 1 T36 GM078000-01 from the National Institutes of Health. The author is greatly thankful to Michael Gelfond for his guidance and help, and to Tran Cao Son and Marcello Balduccini for their organization of Gelfond Festschrift.

## References

1. Aberth, O.: Precise Numerical Analysis Using C++. Academic Press, New York (1998)
2. Beeson, M.J.: Foundations of Constructive Mathematics. Springer, New York (1985)
3. Bishop, E.: Foundations of Constructive Analysis. McGraw-Hill, New York (1967)
4. Bishop, E., Bridges, D.S.: Constructive Analysis. Springer, New York (1985)
5. Bridges, D.S.: Constructive Functional Analysis. Pitman, London (1979)
6. Bridges, D.S., Via, S.L.: Techniques of Constructive Analysis. Springer, New York (2006)
7. Brouwer, L.E.J.: Over de Grondslagen der Wiskunde, Ph.D. thesis, Universiteit van Amsterdam (1907); in Dutch, English translation in Heyting, A. (ed.), Collected Works of L.E.J. Brouwer. I: Philosophy and Foundations of Mathematics, pp. 11–101. North-Holland, Amsterdam (1975)
8. Brouwer, L.E.J.: Über die Bedeutung des Satzes vom ausgeschlossenen Dritten in der Mathematik, insbesondere in der Funktionentheorie. Journal für die reine und angewandte Mathematik 154, 1–7 (1924) (in German); English translation: On the significance of the principle of excluded middle in mathematics, especially in function theory. In: van Heijenoort, J. (ed.), A Source Book in Mathematical Logic, 1879-1931, From Frege to Gödel, pp. 334–345. Harvard University Press, Cambridge (1967)
9. Busemann, H.: The Geometry of Geodesics. Dover Publ., New York (2005)
10. Cox, D.A., Little, J.B., O'Shea, D.: Ideals, Varieties, and Algorithms. Springer, New York (1997)
11. Gel'fond, M.G.: On constructive pseudofunctions. Proceedings of the Leningrad Mathematical Institute of the Academy of Sciences 16, 20–27 (1969) (in Russian); English translation in: Seminars in Mathematics Published by Consultants Bureau (New York-London) 16, 7–10 (1971)

12. Gel'fond, M.G.: Relationship between the classical and constructive developments of mathematical analysis. Proceedings of the Leningrad Mathematical Institute of the Academy of Sciences 32, 5–11 (1972) (in Russian); English translation in Journal of Soviet Mathematics 6(4), 347–352 (1976).
13. Gelfond, M.G.: Classes of formulas of classical analysis which are consistent with the constructive interpretation, PhD Dissertation, Leningrad Mathematical Institute of the Academy of Sciences (1975) (in Russian)
14. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference on Logic Programming ICLP, pp. 1070–1080 (1988)
15. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
16. Hilbert, D.: Zur Theorie der algebraischen Gebilde. Nachrichten von der Königlichen Gesellschaft der Wissenschaften und der Georg- Augusts-Universität zu Göttingen, 450–457 (1988); 25–34, 423–430 (1889) (in German)
17. Hilbert, D.: Über die Theorie der algebraischen Formen. Mathematische Annalen 36, 473–534 (1890) (in German)
18. Hilbert, D.: Über die theorie der algebraischen invarianten. Nachrichten von der Königlichen Gesellschaft der Wissenschaften und der Georg- Augusts-Universität zu Göttingen, 232–241 (1891); 6–16, 439–448 (1892) (in German)
19. Hilbert, D.: Über die vollen Invariantensysteme. Mathematische Annalen 42, 313–370 (1893) (in German)
20. Hilbert, D.: Theory of Algebraic Invariants. Cambrdige University Press, Cambridge (1993) (Lecture Notes from 1897)
21. Kohlenbach, U.: Theorie der majorisierbaren und stetigen Funktionale und ihre Anwendung bei der Extraktion von Schranken aus inkonstruktiven Beweisen: Effektive Eindeutigkeitsmodule bei besten Approximationen aus ineffektiven Eindeutigkeitsbeweisen, Ph.D. Dissertation, Frankfurt am Main (1990) (in German)
22. Kohlenbach, U.: Effective moduli from ineffective uniqueness proofs. An unwinding of de La Vallée Poussin's proof for Chebycheff approximation. Annals for Pure and Applied Logic 64(1), 27–94 (1993)
23. Kohlenbach, U.: Applied Proof Theory: Proof Interpretations and their Use in Mathematics. Springer, Heidelberg (2008)
24. Kolmogorov, A.N.: O printsipe tertium non datur. Matematicheskij Sbornik 32, 646–667 (1925) (in Russian); English translation: On the principle of excluded middle. In: A Source Book in Mathematical Logic, 1879-1931, van Heijenoort, J. (ed.), From Frege to Gödel, pp. 414–437. Harvard University Press, Cambridge (1967)
25. Kreinovich, V.: Uniqueness implies algorithmic computability. In: Proceedings of the 4th Student Mathematical Conference, pp. 19–21. Leningrad University, Leningrad (1975) (in Russian)
26. Kreinovich, V.: Reviewer's remarks in a review of Bridges. In: D.S.: Constrictive Functional Analysis. Pitman, London (1979); Zentralblatt für Mathematik 401, 22–24 (1979)
27. Kreinovich, V.: Categories of space-time models, Ph.D. dissertation, Novosibirsk, Soviet Academy of Sciences, Siberian Branch, Institute of Mathematics (1979) (in Russian)
28. Kreinovich, V.: Physics-motivated ideas for extracting efficient bounds (and algorithms) from classical proofs: beyond local compactness, beyond uniqueness. In: Abstracts of the Conference on the Methods of Proof Theory in Mathematics, Max-Planck Institut für Mathematik, Bonn, Germany, June 3-10, p. 8 (2007)

29. Kreinovich, V., Lakeyev, A., Rohn, J., Kahl, P.: Computational complexity and feasibility of data processing and interval computations. Kluwer, Dordrecht (1998)
30. Kushner, B.A.: Lectures on Constructive Mathematical Analysis. Amer. Math. Soc., Providence (1984)
31. Lacombe, D.: Les ensembles récursivement ouvert ou fermés, et leurs applications à l'analyse récurslve. Compt. Rend. 245(13), 1040–1043 (1957)
32. Lifschitz, V.A.: Investigation of constructive functions by the method of fillings. J. Soviet Math. 1, 41–47 (1973)
33. Noether, M.: Paul Gordan. Mathematisce Annalen 75, 1–41 (1914)
34. Pour-El, M., Richards, J.: Computability in Analysis and Physics. Springer, New York (1989)
35. Weyl, H.: Das Kontinuum. Veyt, Leipzig (1918) (in German); English translation: The Continuum: A Critical Examination of the Foundation of Analysis. Dover Publ., New York (1994)

## Appendix

*Explanation of why uniqueness naturally appears and how other possible classes of statements can be thus generated*

Let us explain why uniqueness naturally appears in our attempts to describe classes of properties $P(x)$ for which the classical validity of the existence statement $\exists x\, P(x)$ implies its constructive validity.

To find such classes, let us try to describe all possible reasonable classes of properties $P(x)$. It is, in general, algorithmically impossible to construct an object $x$ that satisfies the given property $P(x)$; thus, instead, we may look for an object that satisfies the given property only "approximately" (in some reasonable sense). To formally describe the notion of an approximation, we also need relations like $d(x, y) \le \varepsilon$, meaning that the two points $x$ and $y$ are $\varepsilon$-close. To describe a general class, we can combine the atomic properties $P(x)$ and $d(x, y) \le \varepsilon$ by using propositional connectives and quantifiers.

We need at least two different variables $x$ and $y$ to meaningfully use the formula $d(x, y) \le \varepsilon$. For simplicity, let us restrict ourselves to the case when there are no quantifiers (other than implicitly assumed universal quantifiers in front of the formula) and that we have exactly two different variables in a formula describing the class.

We are looking for classes of classical (non-constructive) formulas, so the propositional connectives should also be understood in terms of the classical logic. In the classical logic, every propositional statement can be described in a CNF form, as a conjunction $C_1 \,\&\, C_2 \,\&\, \ldots$ of clauses $C_i$, and every clause $C_i$ is a disjunction of literals (i.e., atomic statements or their negations). Thus, every possible class of properties described by such formulas is an intersection of classes corresponding to clauses. So, to study general classes, it is sufficient to study classes described by individual clauses.

In the classical logic, every clause $a \vee b \vee \ldots \vee c$ can be equivalently described as a rule $\neg a \,\&\, \neg b \,\&\, \ldots \rightarrow c$. So, instead of studying clauses, we will study possible rules.

Rules must relate to the original property, thus, one of the literals in the clause must be $P(x)$ or $\neg P(x)$. Since we must have an approximation – otherwise, no general algorithm is possible – at least one other literal must come from the atomic statement $d(x, y) \leq \varepsilon$. For this literal to be meaningful, we must have at least one literal with the variable $x$ and at least one with the variable $y$. These clauses can only contain literals coming from the atoms $P(x)$, $P(y)$, and $d(x, y) \leq \varepsilon$. Let us classify the corresponding rules.

One of these rules to uniqueness: the rule $P(x) \,\&\, P(y) \rightarrow d(x, y) \leq \varepsilon$. Indeed, this rule means that all the solution to our problem are $\varepsilon$-close to each other – i.e., that, in effect, with the accuracy $\varepsilon$, we have a unique solution. Let us show that other possible rules do not lead to meaningful classes.

If one of the literals corresponding to $P(x)$ is positive, we can make it a conclusion of the corresponding if-then rule. Depending on whether each of the remaining two literals is positive or negative, we have four possibilities:

- The first possibility is $P(x) \,\&\, d(x, y) \leq \varepsilon \rightarrow P(y)$; in this case, once $P(x)$ holds for some object $x$, it holds for all $\varepsilon$-close values $y$. In the usual case when the set of objects is a connected set (e.g., the set of real numbers, $R^n$, the class of all continuous or differentiable functions), in which we can get from each point $x$ to every other point $y$ by a sequence of $\varepsilon$-neighbors, this means that if $P(x)$ holds for one object $x$, it holds for every $x$ as well. Thus, this case covers only two trivial properties $P(x)$: the property that is always true and the property that is always false.
- The second possibility is the rule $\neg P(x) \,\&\, d(x, y) \leq \varepsilon \rightarrow P(y)$. In this case, for $y = x$, we conclude that $\neg P(x) \rightarrow P(x)$ thus, that $\neg P(x)$ is impossible, and $P(x)$ holds for all $x$ – also a trivial case.
- The third possibility is $P(x) \,\&\, d(x, y) > \varepsilon \rightarrow P(y)$. Similarly to the first possibility, we can also usually connect every two elements by a sequence in which every next one is $\varepsilon$-far from the previous one, so we also only get two trivial cases.
- The fourth possibility is $\neg P(x) \,\&\, d(x, y) > \varepsilon \rightarrow P(y)$, i.e., equivalently, $\neg P(x) \,\&\, \neg P(y) \rightarrow d(x, y) \leq \varepsilon$. In this case, all the objects that do not satisfy the property $P(x)$ are $\varepsilon$-close. So, with the exception of this small vicinity, every object satisfies the property $P(x)$. In this sense, this case is "almost" trivial.

Finally, let us consider the clauses in which literals corresponding to both atoms $P(x)$ and $P(y)$ are negative. In this case, we have two possibilities:

- the possibility $P(x) \,\&\, P(y) \rightarrow d(x, y) \leq \varepsilon$ that we already considered, and
- the possibility $P(x) \,\&\, P(y) \rightarrow d(x, y) > \varepsilon$; in this case, taking $x = y$, we conclude that $P(x)$ is impossible, so the property $P(x)$ is always false.

For the case of two variables $x$ and $y$, the (informal) statement is proven.

For a larger number of variables, we can have clauses of the type

$$P(x) \,\&\, P(y) \,\&\, P(z) \rightarrow (d(x, y) \leq \varepsilon \vee d(y, z) \leq \varepsilon \vee d(x, z) \leq \varepsilon)$$

corresponding to the assumption that there are exactly two objects satisfying the property $P(x)$, etc.

# Recollections on Michael Gelfond's 65th Birthday

Alessandro Provetti

Dept. of Physics - Informatics section, University of Messina.
Viale F. Stagno d'Alcontres, 31. I-98166 Messina, Italy
ale@unime.it

Having been invited to write an essay about Michael, both as a person and as a researcher, I welcome the opportunity, forewarning the reader, however, that in so doing I will have to write about myself, also as a person and as a researcher, through my experience as one of Michaels first PhD students. So, walk with me for a while.

Michael and I first met in mid-June 1992 when he was invited by Stefania Costantini and Elio Lanzarone to Milan for a seminar. His presentation was about the then-novel Stable model semantics for (propositional) logic programs with negation. I thoroughly enjoyed the talk, both the speaker and the subject seeming, strangely, both rational and friendly. It was only at the end of the seminar, hearing the repeated and somewhat pointed questions from the skilled logicians who had attended the presentation that I came to understand that something big was going on, and that events had been set in motion. Right after the seminar Stefania took us to a café nearby where, during a lively and pleasant technical discussion that went on for some time, I had the chance to speak to Michael about some *-probably very half-baked-* idea of mine on reasoning about actions. Whereas Italian academics in general are often careful and sometimes slightly alarmed when young people come up with ideas, to my surprise, Michael encouraged me to try harder in that direction, adding that reasoning about actions had become one of his interests and that he would like to hear more about it. That was very motivating for a first-year PhD student who was still only half-sure about his vocation and skills. Subsequent talks during the GULP Conference on Lake Como only reinforced the -I believe mutual- appreciation and before long, July 1992, I was on a plane to El Paso, to work under Michael for the rest of the summer.

It turned out to be a life-changing experience that has motivated and sustained me ever since to purse the research topics Michael pointed us students towards. On the face of it, what a hot El Pasoan summer that was: in the middle of the Chihuahua desert, a group of heavily-accented expats from almost everywhere (and that includes the USA, represented by Bonnie Traylor and Richard Watson, both Michaels Masters students at the time) were taking on some of the hardest topics of Logical AI. Michaels group worked inside what I thought one of the most bizarre buildings on the University of Texas-El Paso campus: a windowless, keystone-shaped, concrete structure whose concrete walls were so thick that one could only assume the building had been designed to resist a nuclear explosion from nearby Trinity point NM! Never mind the location, and the heat outside, the energy, clarity and focus of Michaels research and supervision easily made up for it all. Memorable were the seminars, where discussions with Prof. Vladik Kreinovich and Prof. Amine Khamsi were as lively as a (legal) academic exchange can get.

One important feature of Michael's research and teaching is the ability to have a focused, even narrow, area of interest, yet to nurture several of these interests at the same time and find the conceptual connections and synergies between them. In this sense I see his works on Stable model semantics and formal models of reasoning about actions, as well as his application projects, viz. the *USA Advisor* project, as coexisting and complementing each other seamlessly. In this propitious environment, I myself became interested and active in these three areas, even though I freely admit it has proved a tall order to replicate such insight and effectiveness. Michaels ability to work with diverse students and colleagues and to help them, all of them, to develop research skills is well-known and the present volume, filled by accomplished academicians and scientists from industry, bears witness to it.

Having been one of the first PhD students he mentored, I believe I also had a unique opportunity to share time with Michael and even to be a guest of his family repeatedly for long periods. So, Lara, Greg, Yulia and Patrick became my family in the US. And thanks, Lara, for those wonderful, hearty dinners. Which were followed, and complemented, by wide-ranging discussions on Philosophy, religion, democracy, literature, all the way to the history of Mathematics and the scientific method. All of Michael's students and colleagues joined in at one time or another: Chitta, Olga, Son, Monica, Alfredo, Marcello, Ramon, Pedro, Ann, Mary, Veena, Joel and more than I can mention here. These days, immersed in the rarefied formality of Italian academia, I fondly remember those hot summers with you-all.

To sum up, on the personal level, you will forgive me for feeling that I have received so much in the way of a scientific and personal relationship that I seriously doubt I will ever be able to give it back to my students in equal measure. On the scientific level, I am still and always grateful for the opportunity to be exposed to Michaels research, as I was as recently as last March at the Datalog 2.0 event in Oxford. The work on probabilities he presented there came across as ambitious, deeply conceptual and yet easy-to-grasp, thanks to Michaels rational and gentle presentation style. Indeed, some colleagues later commented to me that they thought they were finally starting to enjoy probabilities through Michaels work!

# Evolving Logic Programs with Temporal Operators

José Júlio Alferes, Alfredo Gabaldon, and João Leite

CENTRIA – Center for Artificial Intelligence,
Universidade Nova de Lisboa, Portugal

**Abstract.** Logic Programming Update Languages have been proposed as extensions of logic programming that allow specifying and reasoning about knowledge bases where both extensional knowledge (facts) as well as intentional knowledge (rules) may change over time as a result of updates.

Despite their generality, these languages are limited in that they do not provide a means to directly access past states of the evolving knowledge. They only allow for so-called Markovian change, i.e. change which is entirely determined by the current state of the knowledge base.

After motivating the need for non-Markovian change, we extend the language EVOLP – a Logic Programming Update Language – with Linear Temporal Logic-like operators, which allow referring to the history of an evolving knowledge base. We then show that it is in fact possible to embed the extended EVOLP into the original one, thus demonstrating that EVOLP itself is expressive enough to encode non-Markovian dynamic knowledge bases. This embedding additionally sheds light on the relationship between Logic Programming Update Languages and Modal Temporal Logics. The embedding is also the starting point of our implementation.

## 1 Introduction

While belief update in the context of classical knowledge bases has traditionally received significant devotion [15], only in the last decade have we witnessed increasing attention to this topic in the context of non-monotonic knowledge bases, notably using logic programming (LP) [25, 5, 24, 10, 11, 31, 35, 30, 27, 3, 6, 2, 34]. Chief among the results of such efforts are several semantics for sequences of logic programs (dynamic logic programs) with different properties [25, 5, 24, 10, 31, 35, 30, 2, 34], and the so-called LP Update Languages: LUPS [6], EPI [9, 11], KABUL [24] and EVOLP [3].

LP Update Languages are extensions of LP designed for modeling dynamic, non-monotonic knowledge bases represented by logic programs. In these knowledge bases, both the extensional part (a set of facts) and the intentional part (a set of rules) may change over time due to updates. In these languages, special types of rules are used to specify updates to the current knowledge base leading to a subsequent knowledge base. LUPS, EPI and KABUL offer a very diverse set of update commands, each specific for one particular kind of update (assertion, retraction, etc). On the other hand, EVOLP follows a simpler approach, staying closer to traditional LP.

In a nutshell, *EVOLP* is a simple though quite powerful extension of ASP [18, 17] that allows for the specification of a program's evolution, in a single unified way, by permitting rules to indicate assertive conclusions in the form of program rules. Syntactically, evolving logic programs are just generalized logic programs[1]. But semantically, they permit to reason about updates of the program itself. The language of *EVOLP* contains a special predicate $assert/1$ whose sole argument is a full-blown rule. Whenever an assertion $assert\,(r)$ is true in a model, the program is updated with rule $r$. The process is then further iterated with the new program. These assertions arise both from self (i.e. internal to the program) updating, and from external updating (e.g. originating in the environment). EVOLP can adequately express the semantics resulting from successive updates to logic programs, considered as incremental knowledge specifications, and whose effect can be contextual. Whenever the program semantics allows for several possible program models, evolution branching occurs, and several evolution sequences are made possible. This branching can be used to specify the evolution of a situation in the presence of incomplete information. Moreover, the ability of *EVOLP* to nest rule assertions within assertions allows rule updates to be themselves updated down the line. Furthermore, the *EVOLP* language can express self-modifications triggered by the evolution context itself, present or future—*assert* literals included in rule bodies allow for looking ahead on some program changes and acting on that knowledge before the changes occur. In contradistinction to other approaches, *EVOLP* also automatically and appropriately deals with the possible contradictions arising from successive specification changes and refinements (via Dynamic Logic Programming[2] [5, 24, 2]).

Update languages have been applied in areas such as Multi-Agent Systems to represent the dynamics of both beliefs and capabilities of agents [23, 21], Legal Reasoning to represent the way rules and regulations change over time and how they update existing ones [29], Role Playing Games to represent the dynamic behavior of Non-Playing Characters [22] and Knowledge Bases to describe their update policies [9]. Furthermore, it was shown that Logic Programming Update Languages are able to capture Action Languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ [19], making them suitable for describing domains of actions [1].

[1] Logic programs that allow for rules with default negated literals in their heads.

[2] *Dynamic Logic Programming* determines the semantics of sequences of generalized logic programs representing states of the world at different time periods, i.e. knowledge undergoing successive updates. As individual theories may comprise mutually contradictory as well as overlapping information, the role of *DLP* is to employ the mutual relationships among different states to determine the declarative semantics, at each state, for the combined theory comprised of all individual theories. Intuitively, one can add newer rules at the end of the sequence, leaving to *DLP* the task of ensuring that these rules are in force, and that previous ones are valid (by inertia) only so far as possible, i.e. they are kept for as long as they are not in conflict with newly added ones, these always prevailing.

Despite their generality, none of the update languages mentioned above provides a means to directly access past states of an evolving knowledge base. These languages were designed for domains where updates are Markovian, meaning that their dynamics are entirely determined by the current state of the knowledge base. Although in many cases this is a reasonable assumption, there are also many scenarios where the behavior of the knowledge base is complex and depends on the particular way it has evolved. In order to capture such complex behavior, it is necessary to be able to refer to properties of the knowledge base at previous states in its "history." This is the motivation behind our proposed language, which we call EVOLP$_T$. The following example attempts to illustrate this.

Consider a knowledge base of user access policies for a number of computers at different locations. A login policy may say, e.g., that after the first failed login a user receives a warning by sms, and if there is another failed login attempt the account is blocked. Moreover, if there are two simultaneous failed login attempts from different ip addresses of the same domain, then the domain is considered suspect. This policy could be expressed by the following rules:

$$\begin{aligned}
sms(User) &\leftarrow \Box(not\, sms(User)), fLogin(User, IP).\\
assert(block(User)) &\leftarrow \Diamond(sms(User)), fLogin(User, IP).\\
assert(suspect(D)) &\leftarrow fLogin(User1, IP_1), fLogin(User2, IP_2),\\
&\qquad domain(D, IP_1), domain(D, IP_2), IP_1 \neq IP_2.
\end{aligned}$$

where $fLogin(User, IP)$ represents the external event of a failed login attempt by the user $User$ at the ip $IP$. The ability to represent such external influence on the contents of a knowledge base is one of the features of EVOLP (and of EVOLP$_T$, which we present below). The symbols $\Diamond$ and $\Box$ represent Past Linear Temporal Logic (Past LTL) like operators. Intuitively, $\Diamond\varphi$ means that there is a past state where $\varphi$ was true and $\Box\varphi$ means that $\varphi$ was true in all past states. The *assert* construct is one of the main features of EVOLP. It allows the user to specify updates to a knowledge base by declaring new rules to be asserted without worrying about conflicts with rules already in the knowledge base.

Referring back to the example, the first rule above specifies that, at any given state, if there is a failed login attempt, $fLogin(User, IP)$, and the user has not received an SMS warning in the past, $\Box(not\, sms(User))$, then the user is sent a warning, $sms(User)$. The second rule specifies that if there is a failed login attempt, $fLogin(User, IP)$, and the user has received an SMS warning sometime in the past, $\Diamond(sms(User))$, then the user is blocked, $assert(block(User))$. Notice that in this case the fact $block(User)$ is asserted while in the case of the SMS warning the atom $sms(User)$ holds in the current state but is not asserted. Intuitively, the atom $sms(User)$ only holds in the state where the rule fires. On the other hand, the fact $block(User)$, which is asserted, persists by inertia until possibly a subsequent update falsifies it.

Now suppose we want to model some updates made later on by the system administrator. Consider for instance that the *sys admin* decides that for those domains which have remained suspect since the last failed login attempt, from now on, i) immediately block any user failing a login from such a domain and ii) not send sms warnings to users failing logins from such domains.

This may be captured by the rules:

$$\begin{aligned} assert(\ assert(block(User)) \leftarrow fLogin(User, IP), domain(IP, Dom).\ ) &\leftarrow \\ \mathbf{S}(suspect(Dom),\ fLogin(Usr2, IP2)), \\ domain(IP2, Dom). \\ assert(\ not\, sms(User) \leftarrow fLogin(User, IP), domain(IP, Dom).\ ) &\leftarrow \\ \mathbf{S}(suspect(Dom),\ fLogin(Usr2, IP2)), \\ domain(IP2, Dom). \end{aligned}$$

The symbol **S** represents an operator similar to the Past LTL operator "since". The intuitive meaning of $\mathbf{S}(\psi, \varphi)$ is: at some point in the past $\varphi$ was true, and $\psi$ has always been true since then.

The ability to refer to the past, such as referring to an sms never having been sent in the past or that a domain has remained suspect since a failed login occurred, is lacking in EVOLP and other similar update languages. For instance, in [9], Eiter et al. discuss that a possible useful extension of their EPI language would be to add a set of constructs for expressing complex behavior that takes into account the full temporal evolution of a knowledge base, and mention as an example a construct *prev*() for referring to the previous stage of the knowledge base.

In this paper, we introduce EVOLP$_T$, an extension of EVOLP with LTL-like operators which allow more flexibility in referring to the history of the evolving knowledge base.[3] We proceed by showing that, with a suitable introduction of new propositional variables, it is possible to embed EVOLP$_T$ into the original EVOLP, thus demonstrating that EVOLP is expressive enough to capture non-Markovian knowledge base dynamics. In addition to this expressivity result, the embedding proves interesting also for shedding light into the relationship between Logic Programming Update Languages and modal temporal logics. The embedding is also a starting point for an implementation, which we also describe in this paper, and that is freely available from `http://centria.di.fct.unl.pt/~jja/updates/`.

## 2 Preliminaries

EVOLP [3] is a logic programming language that includes the special predicate *assert*/1 for specifying updates. An EVOLP program consists of a set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_n$$

where $L_0, L_1 \ldots, L_n$ are literals, i.e., propositional atoms possibly preceded by the negation-as-failure operator *not* . Notice that EVOLP allows *not* to appear in

[3] EVOLP$_T$ was first introduced in [4], a preliminary shorter version of this paper.

the head of rules. The predicate *assert*/1 takes a rule as an argument. Intuitively, *assert*($R$) means that the current knowledge base will be updated by adding rule $R$.

An EVOLP program containing rules with *assert* in the head is capable of going through a sequence of changes even without influence from outside. External influence is captured in EVOLP by means of a sequence of programs each of which represents an update due to external events. In other words, using predicate *assert* one can specify self-modifying behavior in an EVOLP knowledge base, while updates issued externally are specified as a sequence of programs in the same language. The following definitions make these intuitions precise.

**Definition 1.** *Let $\mathcal{L}$ be any propositional language (not containing the predicate assert/1). The* extended language $\mathcal{L}_{assert}$ *is defined inductively as follows:*

- *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assert}$;*
- *If each of $L_0, \ldots, L_n$ is a literal in $\mathcal{L}_{assert}$ (i.e. a propositional atom $A$ or its default negation not $A$), then $L_0 \leftarrow L_1, \ldots, L_n$ is a generalized logic program rule over $\mathcal{L}_{assert}$;*
- *If $R$ is a rule over $\mathcal{L}_{assert}$ then assert($R$) is a propositional atom of $\mathcal{L}_{assert}$;*
- *Nothing else is a propositional atom in $\mathcal{L}_{assert}$.*

*An* evolving logic program *over a language $\mathcal{L}$ is a (possibly infinite) set of logic program rules over $\mathcal{L}_{assert}$.*

By nesting *assert*/1 one can specify knowledge base updates that may in turn introduce further updates. As mentioned earlier, in addition to these "internal updates," one can specify "external events" intuitively of two types: observation (of facts or rules) events that occur at a given state, and direct assertion commands which specify an update to the knowledge base with new rules. The main difference between the two types of event is that observations only hold in the state where they occur, while rules that are asserted persist in the knowledge base. Syntactically, observations are represented in EVOLP by rules without the assert predicate in the head, and assertions by rules with it. A sequence of external events is represented as a sequence of EVOLP programs:

**Definition 2.** *Let $P$ be an evolving program over the language $\mathcal{L}$. An* event sequence *over $P$ is a sequence of evolving programs over $\mathcal{L}$.*

Given an initial EVOLP program and an event sequence, the semantics dictates what holds and what does not after each of the events in the sequence. More precisely, the meaning of a sequence of EVOLP programs is given by a set of *evolution stable models*, each of which is a sequence of interpretations $\langle I_1, \ldots, I_n \rangle$. Each evolution stable model describes some possible evolution of an initial program after a number $n$ of evolution steps, given the events in the sequence. Each evolution is represented by a sequence of programs $\langle P_1, \ldots, P_n \rangle$, where each $P_i$ corresponds to a knowledge base constructed as follows: regarding head asserts, whenever the atom *assert*(*Rule*) belongs to an interpretation in a sequence, i.e. belongs to a model according to the stable model semantics of the current program, then *Rule* must belong to the program in the next state. Assert literals in the bodies of rules are treated as are any other literals.

**Definition 3.** *An* evolution interpretation *of length $n$ over $\mathcal{L}$ is a finite sequence $\mathcal{I} = \langle I_1, I_2, \ldots, I_n \rangle$ of sets of propositional atoms of $\mathcal{L}_{assert}$. The* evolution trace *of $P$ under $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where*

- $P_1 = P$ *and*
- $P_i = \{R \mid assert(R) \in I_{i-1}\}$ *for* $2 \le i \le n$.

Sequences of programs are then treated as in *dynamic logic programming* [5], where the most recent rules are set in force, and previous rules are valid (by inertia) insofar as possible, i.e. they are kept for as long as they do not conflict with more recent ones. The semantics of dynamic logic programs [2] is a generalization of the answer-set semantics of [26] in the sense that if the sequence consists of a single program, the semantics coincide.

Before we define this semantics, we need to introduce some notation. Let $\rho_s(P_1, ..., P_n)$ denote the multiset of all rules appearing in the programs $P_1, ..., P_s$. If $r$ is a rule of the form $L_0 \leftarrow L_1, \ldots, L_n$, then $H(r) = L_0$ (dubbed the head of the rule) and $B(r) = L_1, \ldots, L_n$ (dubbed the body of the rule). Two rules $r, r'$ are said to be *conflicting rules*, denoted by $r \bowtie r'$, iff $H(r) = A$ and $H(r') = not\, A$ or $H(r) = not\, A$ and $H(r') = A$. By $least(P)$ we denote the least model of the definite program obtained from the argument program $P$ by replacing every default literal $not\, A$ by a new atom $not\_A$. For every set of propositional atoms $M$ in $\mathcal{L}_{assert}$, we define $M' = M \cup \{not\_A \mid A \notin M\}$. Finally, we need to define the following two sets. Given a program sequence $\mathcal{P}$ and a set of atoms $M$, the first set denotes the set of negated atoms $not\, A$ that hold wrt a state $s$:

$$Def_s(\mathcal{P}, M) \stackrel{\text{def}}{=} \{not\, A \mid \nexists r \in \rho_s(\mathcal{P}), H(r) = A, M \vDash B(r)\} .$$

The second is the set of rules that are "rejected" because of the appearance of a conflicting rule later in the sequence and whose body is satisfied by $M$:

$$Rej_s(\mathcal{P}, M) \stackrel{\text{def}}{=} \{r \mid r \in P_i, \exists r' \in P_j, i \le j \le s, r \bowtie r', M \vDash B(r')\} .$$

**Definition 4.** *Let $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ be a sequence of programs over $\mathcal{L}$. A set, $M$, of propositional atoms in $\mathcal{L}_{assert}$ is a dynamic stable model of $\mathcal{P}$ at state $s$, $1 \le s \le n$, iff*

$$M' = least\left([\rho_s(\mathcal{P}) - Rej_s(\mathcal{P}, M)] \cup Def_s(\mathcal{P}, M)\right) .$$

*Whenever $s = n$, we simply say that $M$ is a stable model of $\mathcal{P}$.*

Going back to EVOLP, the events received at each state must be added to the corresponding program of the trace before testing the stability condition of stable models of the evolution interpretation.

**Definition 5 (Evolution Stable Model).** *Let $\mathcal{I} = \langle I_1, ..., I_n \rangle$ be an evolution interpretation of an EVOLP program $P$ and $\langle P_1, P_2, \ldots, P_n \rangle$ be the corresponding execution trace. Then $\mathcal{I}$ is an* evolution stable model *of $P$ given event sequence $\langle E_1, E_2, \ldots, E_n \rangle$ iff for every $i$ ($1 \leq i \leq n$), $I_i$ is a stable model of $\langle P_1, P_2, \ldots, (P_i \cup E_i) \rangle$.*

## 3 EVOLP with Temporal Operators

EVOLP programs have the limitation that rules cannot refer to past states in the evolution of a program. In other words, they do not allow one to specify behavior that is conditional on the full evolution of the system being modeled. Despite the fact that the whole evolution is available as a sequence of evolving programs, the body of a rule at any state is always evaluated only with respect to that state. In fact, a careful look at the above definition of the semantics of dynamic logic programs makes this evident: in the definitions of both $Def_s(\mathcal{P}, M)$ and $Rej_s(\mathcal{P}, M)$, rules in previous states are indeed taken into account, but the rule bodies are always evaluated with respect to model $M$ which is the model defined for the knowledge base at state $s$.

Our goal here is to extend the syntax and semantics of EVOLP to overcome this limitation, defining a new language called EVOLP$_T$. Our approach is similar to the approach in [12, 14] where action theories in the Situation Calculus are generalized with non-Markovian control. In particular, we extend the syntax of EVOLP with Past LTL modalities $\bigcirc(G)$, $\Diamond(G)$ $\Box(G)$, and $\mathbf{S}(G_1, G_2)$, which intuitively mean, respectively: $G$ holds in the previous state; $G$ holds in some past state; $G$ holds in all past states; and $G_1$ holds since $G_2$ holds.

Moreover, we allow arbitrary nesting of these operators as well as negation-as-failure in front of their arguments. Unlike *not*, however, temporal operators are not allowed in the head of rules. The only restriction on the body of rules is that negation is allowed to appear in front of atoms and temporal operators only. The formal definition of the language and programs in EVOLP$_T$ is as follows.

**Definition 6 (EVOLP with Temporal Operators).** *Let $\mathcal{L}$ be any propositional language (not containing the predicates $assert/1$, $\bigcirc/1$, $\Diamond/1$, $\mathbf{S}/2$ and $\Box/1$). The* extended temporal language $\mathcal{L}_{assertT}$ *and the* set of b-literals[4] $\mathcal{G}$ *are defined inductively as follows:*

- *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assertT}$ and b-literals in $\mathcal{G}$.*
- *If $G_1$ and $G_2$ are b-literals in $\mathcal{G}$ then $\bigcirc(G_1)$, $\Diamond(G_1)$, $\mathbf{S}(G_1, G_2)$ and $\Box(G_1)$ are t-formulae[5], and are also b-literals in $\mathcal{G}$.*
- *If $G$ is a t-formula or an atom in $\mathcal{L}_{assertT}$ then $not\, G$ is a b-literal in $\mathcal{G}$.*
- *If $G_1$ and $G_2$ are b-literals in $\mathcal{G}$, then $(G_1, G_2)$ is a b-literal in $\mathcal{G}$.*

[4] Intuitively, b-literal stands for body-literal.

[5] Intuitively, t-formula stands for temporal-formula.

- *If $L_0$ is a propositional atom $A$ in $\mathcal{L}_{assertT}$ or its default negation not $A$, and each of $G_1, \ldots G_n$ is a b-literal, then $L_0 \leftarrow G_1, \ldots, G_n$ is a generalized logic program rule over $\mathcal{L}_{assertT}$ and $\mathcal{G}$.*
- *If $R$ is a rule over $\mathcal{L}_{assertT}$ then $assert(R)$ is a propositional atom of $\mathcal{L}_{assertT}$.*
- *Nothing else is a propositional atom in $\mathcal{L}_{assert}$ or a b-literal in $\mathcal{G}$.*

*An* evolving logic program with temporal operators *over a language $\mathcal{L}$ is a (possibly infinite) set of generalized logic program rules over $\mathcal{L}_{assertT}$ and $\mathcal{G}$.*

For example, under this definition the following is a legal EVOLP$_T$ rule:

$$assert(a \leftarrow not\, \Diamond(b)) \leftarrow not\, \Box(not\, \Diamond((b, not\, assert(c \leftarrow d)))).$$

Notice the nesting of temporal operators and the appearance of negation, conjunction and assert under the scope of the temporal operators, which is all allowed.

On the other hand, the following are examples of rules that are not legal according to the definition:

$$\begin{array}{l} assert(\Box(b) \leftarrow a) \leftarrow b. \\ a \leftarrow \Diamond(not\,(a, b)). \\ a \leftarrow not\, not\, b. \end{array}$$

In the first rule, $\Box(b)$ appears in the argument rule $\Box(b) \leftarrow a$, but temporal operators are not allowed in the head of rules. The second rule applies negation to a conjunctive b-literal, and the third rule has double negation. But negation is only allowed in front of atoms and t-formulae.

As in EVOLP, the definition of the semantics is based on sequences of interpretations $\langle I_1, \ldots, I_n \rangle$ (evolution interpretations). Each interpretation in a sequence stands for the propositional atoms (of $\mathcal{L}_{assertT}$) that are true at the corresponding state, and a sequence stands for a possible evolution of an initial program after a given number $n$ of evolution steps. However, whereas in the original EVOLP language the satisfiability of rule bodies in one such interpretation $I_i$ can easily be defined in terms of set inclusion—all the positive atoms must be included in $I_i$, all the negative ones excluded—in EVOLP$_T$ satisfiability is more elaborate as it must account for the Past LTL modalities.

**Definition 7 (Satisfiability of b-literals).** *Let $\mathcal{I} = \langle I_1, ..., I_n \rangle$ be an evolution interpretation of length $n$ of a program $P$ over $\mathcal{L}_{assertT}$, and let $G$ and $G'$ be any b-literals in $\mathcal{G}$. The satisfiability relation is defined as:*

$$\begin{array}{ll}
\mathcal{I} \models A & \textit{iff}\ A \in I_n \wedge A \in \mathcal{L}_{assertT} \\
\mathcal{I} \models not\, G & \textit{iff}\ \langle I_1, ..., I_n \rangle \not\models G \\
\mathcal{I} \models G, G' & \textit{iff}\ \langle I_1, ..., I_n \rangle \models G \wedge \langle I_1, ..., I_n \rangle \models G' \\
\mathcal{I} \models \bigcirc(G) & \textit{iff}\ n \geq 2 \wedge \langle I_1, ..., I_{n-1} \rangle \models G \\
\mathcal{I} \models \Diamond(G) & \textit{iff}\ n \geq 2 \wedge \exists i < n : \langle I_1, ..., I_i \rangle \models G \\
\mathcal{I} \models \mathbf{S}(G, G') & \textit{iff}\ n > 2 \wedge \exists i < n : \langle I_1, ..., I_i \rangle \models G' \wedge \forall i < k < n : \langle I_1, ..., I_k \rangle \models G \\
\mathcal{I} \models \Box(G) & \textit{iff}\ \forall i < n : \langle I_1, ..., I_i \rangle \models G
\end{array}$$

Given an evolution interpretation, an *evolution trace* (defined below) represents one of the possible evolutions of the knowledge base. In EVOLP$_T$, whether an evolution trace is one of these possible evolutions additionally depends on the satisfaction of the t-formulae that appear in rules. Towards formally defining evolution traces, we first define an elimination procedure which evaluates satisfiability of t-formulae and replaces them with a corresponding truth constant.

**Definition 8 (Elimination of Temporal Operators).** *Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation and $L_0 \leftarrow G_1, \ldots, G_n$ a generalized logic program rule. The rule resulting from the elimination of temporal operators given $\mathcal{I}$, denoted by $El(\mathcal{I}, L_0 \leftarrow G_1, \ldots, G_n)$, is obtained by:*

- *replacing by* **true** *every t-formula $G_t$ in the body such $\mathcal{I} \models G_t$; and*
- *by replacing all remaining t-formulae by* **false**

*where constants* **true** *and* **false** *are defined, as usual, such that the former is true in every interpretation and the latter is not true in any interpretation.*

*The program resulting from the elimination of temporal operators given $\mathcal{I}$, denoted by $El(\mathcal{I}, P)$, is obtained by applying $El(\mathcal{I}, r)$ to each rule $r$ of $P$.*

Evolution traces are defined as in Def. 3 except that t-formulae are eliminated by applying *El*:

**Definition 9 (Evolution Trace of EVOLP$_T$ Programs).** *Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of an EVOLP$_T$ program $P$. The* evolution trace *of $P$ under $\mathcal{I}$ is the sequence of programs $\langle P_1, P_2, \ldots, P_n \rangle$ where:*

- *$P_1 = El(\langle I_1 \rangle, P)$ and*
- *$P_i = El(\langle I_1, \ldots, I_i \rangle, \{R \mid assert(R) \in I_{i-1}\})$ for $2 \leq i \leq n$.*

Since the programs in an evolution trace do not mention t-formulae, evolution stable models can be defined in a similar way as in Def. 5, only taking into account that the temporal operators must also be tested for satisfiability, and eliminated accordingly, from the evolution trace and also from the external events. Here, events are simply sequences of EVOLP$_T$ programs.

**Definition 10 (Evolution Stable Models of EVOLP$_T$ Programs).** *Let $\mathcal{I} = \langle I_1, \ldots, I_n \rangle$ be an evolution interpretation of an EVOLP$_T$ program $P$ and $\langle P_1, P_2, \ldots, P_n \rangle$ be the corresponding execution trace. Then $\mathcal{I}$ is an* evolution stable model *of $P$ given event sequence $\langle E_1, E_2, \ldots, E_n \rangle$ iff $I_i$ is a stable model of*

$$\langle P_1, P_2, \ldots, (P_i \cup E_i^*) \rangle$$

*for every $i$ $(1 \leq i \leq n)$, where $E_i^* = \{El(\langle I_1, \ldots, I_i \rangle, r) \mid r \in E_i\}$.*

Since various evolutions may exist for a given length, evolution stable models alone do not determine a truth relation. But one such truth relation can be defined, as usual, based on the intersection of models:

**Definition 11 (Stable Models after $n$ Steps of EVOLP$_T$ Programs).** *Let $P$ be an EVOLP$_T$ program over the language $\mathcal{L}$. We say that a set of propositional atoms $M$ over $\mathcal{L}_{assertT}$ is a* stable model of $P$ after $n$ steps given the sequence of events $\mathcal{E}$ *iff there exist $I_1, \ldots, I_{n-1}$ such that $\langle I_1, \ldots, I_{n-1}, M\rangle$ is an evolution stable model of $P$ given $\mathcal{E}$.*

*We say that propositional atom $A$ of $\mathcal{L}_{assertT}$ is:*

– true after $n$ steps given $\mathcal{E}$ *iff all stable models after $n$ steps contain $A$;*
– false after $n$ steps given $\mathcal{E}$ *iff no stable model after $n$ steps contains $A$;*
– unknown after $n$ steps given $\mathcal{E}$ *otherwise.*

It is worth noting that basic properties of Past LTL operators carry over to EVOLP$_T$. In particular, in EVOLP$_T$, as in LTL, some of the operators are not strictly needed, since they can be express in terms of other operators:

**Proposition 1.** *Let $\mathcal{I} = \langle I_1, \ldots, I_n\rangle$ be an evolution stable model of an EVOLP$_T$ program given a sequence of events $\mathcal{E}$. Then, for every $G \in \mathcal{G}$:*

– $\mathcal{I} \models \Box(G)$ *iff* $\mathcal{I} \models not\, \Diamond(not\, G)$;
– $\mathcal{I} \models \Diamond(G)$ *iff* $\mathcal{I} \models \mathbf{S}(\mathbf{true}, G)$

Moreover, it should also be noted that EVOLP$_T$ is an extension of EVOLP in the sense that when no temporal operators appear in the program and in the sequence of events, then evolution stable models coincide with those of the original EVOLP. As an immediate consequence of this fact, it can also be noted that EVOLP$_T$ coincides with answer-sets when, moreover, the sequence of events is empty and predicate *assert*/1 does not occur in the program.

We end this section by illustrating the results of EVOLP$_T$ semantics on the motivating example of the Introduction.

*Example 1.* The initial program $P$ is made of the rules specifying the initial policy plus, for the example, some facts about ips and domains:

$$\begin{aligned} sms(User) &\leftarrow \Box(not\, sms(User)), fLogin(User, IP).\\ assert(block(User)) &\leftarrow \Diamond(sms(User)), fLogin(User, IP).\\ assert(suspect(D)) &\leftarrow fLogin(User1, IP_1), fLogin(User2, IP_2),\\ &\qquad domain(D, IP_1), domain(D, IP_2), IP_1 \neq IP_2.\\ domain(ip_1, d_1). \quad & domain(ip_2, d_2). \quad domain(ip_3, d_2). \quad domain(ip_4, d_2).\end{aligned}$$

At this initial state, assuming an empty $E_1$, the only evolution stable model is $\langle I_1\rangle$ with $I_1$ only including the *domain*/2 facts.

Now suppose that there is an event of a failed login from John at $ip_1$, represented by the program $E_2$ with the single fact $fLogin(john, ip_1)$.

The only evolution stable model of $P$ given $\langle E_1, E_2\rangle$ is the interpretation $\langle I_1, I_2\rangle$ where $I_1$ includes the *domain*/2 facts and $I_2$ includes, besides the *domain*/2 facts, $sms(john)$, which intuitively means that upon receiving the event of the failed login from John, an sms is sent to him. Indeed, the evolution trace is $\langle P_1, P_2 \rangle$ where $P_2$ is empty (since $I_1$ has no *domain*/1 predicates) and $P_1 = El(\langle I_1, I_2 \rangle, P)$ is:

$$\begin{aligned}
sms(User) &\leftarrow \mathbf{true}, fLogin(User, IP).\\
assert(block(User)) &\leftarrow \mathbf{false}, fLogin(User, IP).\\
assert(suspect(D)) &\leftarrow fLogin(User1, IP_1), fLogin(User2, IP_2),\\
&\qquad domain(D, IP_1), domain(D, IP_2), IP_1 \neq IP_2.
\end{aligned}$$

$$domain(ip_1, d_1). \quad domain(ip_2, d_1). \quad domain(ip_3, d_2). \quad domain(ip_4, d_2).$$

It is easy to verify that $I_1$ is a stable model of $\langle P_1 \cup E_1^*) \rangle$ and $I_2$ a stable model of $\langle P_1, (P_2 \cup E_2^*) \rangle$.

Continuing the example, suppose that there is an event of another failed login from John, this time at $ip_3$, and of a failed login from Eva at $ip_4$ (represented by the program $E_3$ with the facts $fLogin(john, ip_3)$ and $fLogin(eva, ip_4)$). Now the evolution stable model is $\langle I_1, I_2, I_3 \rangle$ were $I_1$ and $I_2$ are as above, $I_3$ includes $sms(eva)$, and also $assert(block(john))$ and $assert(suspect(d_2))$. As such, $block(john)$ and $suspect(d_2)$ will be added to the evolution trace in the subsequent step.

Suppose now that the system administrator issues the update with the rules shown in the Introduction, including them in $E_4$, and subsequently, in $E_5$, there are failed logins from Carl at $ip_4$ and from Vic at $ip_1$. In this case, after these 5 steps, $assert(block(carl))$ becomes true (blocking Carl's access), and no sms is sent to him because the more recent rule $not\, sms(User) \leftarrow fLogin(User, IP), domain(IP, d_2)$ (belonging to $P_5$ because of the assertion in $E_4$) overrides the first rule in the initial program. The same does not happen for Vic, since this rule is not asserted for domain $d_1$, as it was not suspect.

## 4 Embedding Temporal Operators in EVOLP

In this section we show that it is possible to transform EVOLP$_T$ programs into regular EVOLP programs. This transformation is important for at least two reasons. On one hand, it shows that EVOLP is expressive enough to deal with non-Markovian conditions, although not immediately nor easily. On the other hand, given the existing implementations of EVOLP, the transformation readily provides a means to implement EVOLP$_T$, as we describe below.

Transforming EVOLP$_T$ programs and sequences of events into EVOLP mainly amounts to eliminating the t-formulae by introducing new propositional atoms that will represent the t-formulae, and rules that will model how the truth value of the t-formulae changes as the knowledge base evolves. We start by defining the target language of the resulting EVOLP programs.

**Definition 12 (Target language).** *Let $P$ and $\mathcal{E}$ be an EVOLP$_T$ program and a sequence of events, respectively, in a propositional language $\mathcal{L}$. Let $\mathcal{G}(P, \mathcal{E})$ be the set of all non-atomic b-literals that appear in $P$ or $\mathcal{E}$.*

*The EVOLP target language is*

$$\mathcal{L}_E(\mathcal{L}, P, \mathcal{E}) \stackrel{\text{def}}{=} \mathcal{L} \cup \{'L' \mid L \in \mathcal{G}(P, \mathcal{E})\}$$

*where by $'L'$ we mean a propositional variable whose name is the (atomic) string of characters that compose the formula $L$ (which is assumed not to occur in $\mathcal{L}$).*

The transformation takes the rules in both program and events, and replaces all occurrences of t-formulas and conjunctions in their bodies by the corresponding new propositional variables in the target language. Moreover, extra rules are added to the program for encoding the behavior of the operators.

**Definition 13 (Transformation).** Let $P$ be an EVOLP$_T$ program and $\mathcal{E} = \langle E_1, E_2, \ldots, E_n\rangle$ be a sequence of events in a propositional language $\mathcal{L}$. Then $Tr_E(P, \mathcal{E}) = (T_P, \langle T_{E_1}, \ldots, T_{E_n}\rangle)$ is a pair consisting of an EVOLP program (i.e., without temporal operators) and a sequence of events, both in the language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, defined as follows:[6]

1. **Rewritten program rules.** For every rule $r$ in $P$ (resp. each of the $E_i$), $T_P$ (resp. $T_{E_i}$) contains a rule obtained from $r$ by replacing every t-formula $G$ in its body by the new propositional variable $'G'$;
2. **Previous-operator rules.** For every propositional variable of the form $'\bigcirc(G)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, $T_P$ contains:
$$assert('\bigcirc(G)') \leftarrow 'G'.$$
$$assert(not\, '\bigcirc(G)') \leftarrow not\, 'G'.$$
3. **Sometimes-operator rule.** For every propositional variable of the form $'\Diamond(G)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, $T_P$ contains:
$$assert('\Diamond(G)') \leftarrow 'G'.$$
4. **Since-operator rules.** For every propositional variable of the form $'\mathbf{S}(G_1, G_2)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, $T_P$ contains:
$$assert('\mathbf{S}(G_1, G_2)') \leftarrow 'G_1', '\bigcirc(G_2)'.$$
$$assert(assert(not\, '\mathbf{S}(G_1, G_2)') \leftarrow not\, 'G_1') \leftarrow assert('\mathbf{S}(G_1, G_2)').$$
5. **Always-operator rules.** For every propositional variable of the form $'\Box(G)'$, appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, $T_P$ contains:
$$'\Box(G)' \leftarrow 'G',\ not\ \bigcirc\,\mathbf{true}.$$
$$assert(not\, '\Box(G)') \leftarrow not\, 'G'.$$
6. **Conjunction and negation rules.** For every propositional variables of the form $'not\, G'$, or of the form $'G_1, G_2'$ appearing in $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$, $T_P$ contains, respectively:
$$'not\, G' \leftarrow not\, 'G'$$
$$'G_1, G_2' \leftarrow 'G_1', 'G_2'.$$
$$'G_1, G_2' \leftarrow 'G_2', 'G_1'.$$

[6] When $G$ is an atom, $'G'$ stands for the same atom $G$.

Before establishing the correctness of this transformation with respect to EVOLP$_T$, it is worth giving some intuition on how the rules added in the transformed program indeed capture the meaning of the temporal operators.

The rule for $\Diamond(G)$ guarantees that whenever $'G'$ is true at some state, the fact $'\Diamond(G)'$ is added to the subsequent program. So, since no rule for *not* $'\Diamond(G)'$ is ever added in the transformation and rules with $\Diamond(G)$ in the head are not allowed in EVOLP$_T$ programs, $'\Diamond(G)'$ will continue to hold in succeeding states. The first rule for $\bigcirc(G)$ is similar to the one for $\Diamond(G)$. But the second one adds the fact *not* $\bigcirc(G)$ in case *not* $G$ is true. So $'\bigcirc(G)'$ will be true in the state after the one in which $'G'$ is true, and will become false in a state immediately after one in which $'G'$ is false, as desired.

The rules for $\Box(G)$ are also easy to explain, and in fact result from the dualization of the $\Diamond(G)$ operator. More interesting are the rules for $\mathbf{S}(G_1, G_2)$. The first simply captures the condition where $'\mathbf{S}(G_1, G_2)'$ first becomes true: it adds a fact for it, in the state immediately after one in which $'G_1'$ is true and which is preceded by one in which $'G_2'$ is true. With the addition of this fact, according to the semantics of EVOLP, $'\mathbf{S}(G_1, G_2)'$ will remain true by inertia in subsequent states until a rule that asserts *not* $'\mathbf{S}(G_1, G_2)'$ fires. We want this to happen only until a state immediately after one in which $'G_1'$ becomes false. This effect is obtained with the second rule by adding, at the same time as the fact $'\mathbf{S}(G_1, G_2)'$ is asserted, a rule stating that the falsity of $'G_1'$ leads to the assertion of *not* $'\mathbf{S}(G_1, G_2)'$.

These intuitions form the basis of the proof of the following theorem, which we sketch below.

**Theorem 1 (Correctness).** *Let $P$ be an evolving logic program with temporal operators over language $\mathcal{L}$, and let $Tr(P)$ be the transformed evolving logic program over language $\mathcal{L}_E(\mathcal{L}, P, \langle\rangle)$. Then $M = \langle I_1, \ldots, I_n\rangle$ is an evolving stable model of $P$ iff there exists an evolving stable model $M' = \langle I'_1, \ldots, I'_n\rangle$ of $Tr(P)$ such that*

$$I_1 = (I'_1 \cap \mathcal{L}_{assert}), \ldots, I_n = (I'_n \cap \mathcal{L}_{assert}).$$

*Proof.* (Sketch) The proof proceeds by induction on the length of the sequence of interpretations, showing that the transformed atoms corresponding to t-formulae satisfied in each state, and some additional assert-literals guarantying the assertion of t-formulae, belong to the interpretation state.

For the induction step, the rules of the transformation are used to guarantee that the new propositional variables belong to the interpretation of the transformed program whenever the corresponding temporal b-literals belong to the interpretation of the original EVOLP$_T$ program. For example, if some $G \in I_i$ then, according to the EVOLP$_T$ semantics, for every $j > i$, $\Diamond(G) \in I_j$; by induction hypothesis, $'G' \in I'_i$ and the "sometime-operator rule" guarantees that $'\Diamond(G)'$ is added to the subsequent program and so, since no rule for *not* $'\Diamond(G)'$ is added in the transformation, for every $j > i$, $'\Diamond(G)' \in I'_j$. As another example, note that the first "previous-operator" rule is similar to the "sometime-operator rule" and the second adds the fact *not* $'\bigcirc(G)'$ in case *not* $'G'$ is true; so, as in the EVOLP$_T$ semantics, $'\bigcirc(G)' \in I'_{i+i}$. A similar reasoning is applied also for the since-operator and always-operator rules.

To account for the nesting of temporal operators first note that the transformation adds the above rules for all possible nestings. However, since this nesting can be combined with conjunction and negation, as per the definition of the syntax of EVOLP$_T$ (Def. 6), care must be taken with the new propositional variables that stand for those conjunctions and negations. The last rules of the transformation guarantee that a new atom for a conjunction is true in case the b-literals in the conjunction are true, and that a new atom for the negation of a b-literal is true in case the negation of the b-literal is true. □

Since events are also EVOLP$_T$ programs, we can easily deal with events by applying the same transformation. First, when transforming the main program $P$, we take into account the t-formulae in the event sequence. Then the transformation is applied to the events themselves.

**Definition 14 (Transformation of EVOLP$_T$ with Event Sequence).** *Let $P$ be an evolving program with temporal operators and $\mathcal{E} = \langle E_1, \ldots, E_k \rangle$ be an event sequence, both over $\mathcal{L}$. Then $Tr(P, \langle E_1, \ldots, E_k \rangle)$ is an EVOLP program (without temporal operators) over language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$ obtained from $P$ by applying exactly the same procedure as in Def. 13, only replacing* "appearing in $P$" *by* "either appearing in $P$ or in any of the $E_i$'s"*.*

**Theorem 2.** *Let $P$ be an evolving logic program with temporal operators over language $\mathcal{L}$, and let $Tr(P)$ be the transformed evolving logic program over language $\mathcal{L}_E(\mathcal{L}, P, \mathcal{E})$.*

*Then $M = \langle I_1, \ldots, I_n \rangle$ is an evolving stable model of $P$ given $\langle E_1, \ldots, E_k \rangle$ iff there exists an evolving stable model $M' = \langle I'_1, \ldots, I'_n \rangle$ of $Tr(P, \langle E_1, E_2, \ldots, E_k \rangle)$ given the sequence $\langle Tr(E_1), \ldots, Tr(E_k) \rangle$ such that*

$$I_1 = (I'_1 \cap \mathcal{L}_{assert}), \ldots, I_n = (I'_n \cap \mathcal{L}_{assert}).$$

## 5 EVOLP$_T$ Implementations

We have developed two implementations of EVOLP$_T$. One follows the evolution stable models semantics defined above, while the second one computes answers to existential queries under the well-founded semantics [16], i.e., it is sound, though not complete, with respect to the truth relation of Def. 11. The implementations rely on two consecutive program transformations: the first is the transformation from EVOLP$_T$ into EVOLP described above which eliminates temporal operators. The second transformation is based on previous work [33] and takes the result of the first and generates a normal logic program.

### 5.1 Evolution SM Semantics Implementation

Here we discuss some of the intuitions behind the second transformation and then describe the actual implementation and its usage. We then describe our implementation of query answering under the well-founded semantics.

The implementation proceeds by using the transformation presented in [33] which converts the resulting EVOLP program and sequence of events into a normal logic program over an extended language whose stable models have a one-to-one correspondence with the evolution stable models of the original EVOLP program and sequence of events.

As described in [33], the extended language over which the resulting program is constructed is defined as follows:

$$\begin{aligned}\mathcal{L}_{\text{trans}} \overset{\text{def}}{=} &\left\{ A^j, A^j_{\text{neg}} \;\middle|\; A \in \mathcal{L}_{\text{assert}} \wedge 1 \le j \le n \right\}\\ &\cup \left\{ \text{rej}(A^j, i), \text{rej}(A^j_{\text{neg}}, i) \;\middle|\; A \in \mathcal{L}_{\text{assert}} \wedge 1 \le j \le n \wedge 0 \le i \le j \right\}\\ &\cup \{ u \} \;\; .\end{aligned}$$

Atoms of the form $A^j$ and $A^j_{\text{neg}}$ in the extended language allow us to compress the whole evolution interpretation (consisting of $n$ interpretations of $\mathcal{L}_{\text{assert}}$, see Def. 3) into just one interpretation of $\mathcal{L}_{\text{trans}}$. The superscript $j$ then encodes the index of these interpretations. Atoms of the form $\text{rej}(A^j, i)$ and $\text{rej}(A^j_{\text{neg}}, i)$ are needed for simulating rule rejection. Roughly, an atom $\text{rej}(A^j, i)$ intuitively means that, at step $j$, rules with head $A$ that were added at the earlier step $i$ have been rejected and thus cannot be used to derive $A$. Similarly for $\text{rej}(A^j_{\text{neg}}, i)$. The atom $u$ will serve to formulate constraints needed to eliminate some unwanted models of the resulting program.

To simplify the notation in the transformation's definition, we will use the following conventions: Let $L$ be a literal over $\mathcal{L}_{\text{assert}}$, *Body* a set of literals over $\mathcal{L}_{\text{assert}}$ and $j$ a natural number. Then:

– If $L$ is an atom $A$, then $L^j$ is $A^j$ and $L^j_{\text{neg}}$ is $A^j_{\text{neg}}$.
– If $L$ is a default literal **not** $A$, then $L^j$ is $A^j_{\text{neg}}$ and $L^j_{\text{neg}}$ is $A^j$.
– $\mathit{Body}^j = \{ L^j \mid L \in \mathit{Body} \}$.

**Definition 15.** *[33] Let $P$ be an evolving logic program and $\mathcal{E} = (E_1, E_2, \ldots, E_n)$ an event sequence. By a* transformational equivalent *of $P$ given $\mathcal{E}$ we mean the normal logic program $P_\mathcal{E} = P^1_\mathcal{E} \cup P^2_\mathcal{E} \cup \ldots \cup P^n_\mathcal{E}$ over $\mathcal{L}_{\text{trans}}$, where each $P^j_\mathcal{E}$ consists of these six groups of rules:*

1. ***Rewritten program rules.*** *For every rule $(L \leftarrow \mathit{Body}.) \in P$, $P^j_\mathcal{E}$ contains the rule*
$$L^j \leftarrow \mathit{Body}^j, \textbf{not}\, \text{rej}(L^j, 1).$$
2. ***Rewritten event rules.*** *For every rule $(L \leftarrow \mathit{Body}.) \in E_j$, $P^j_\mathcal{E}$ contains the rule*
$$L^j \leftarrow \mathit{Body}^j, \textbf{not}\, \text{rej}(L^j, j).$$
3. ***Assertable rules.*** *For every rule $r = (L \leftarrow \mathit{Body}.)$ over $\mathcal{L}_{\text{assert}}$ and all $i$, $1 < i \le j$, such that $(\text{assert}(r))^{i-1}$ is in the head of some rule of $P^{i-1}_\mathcal{E}$, $P^j_\mathcal{E}$ contains the rule*
$$L^j \leftarrow \mathit{Body}^j, (\text{assert}(r))^{i-1}, \textbf{not}\, \text{rej}(L^j, i).$$

4. ***Default assumptions.*** *For every atom $A \in \mathcal{L}_{\text{assert}}$ such that $A^j$ or $A^j_{\text{neg}}$ appears in some rule of $P^j_{\mathcal{E}}$ (from the previous groups of rules), $P^j_{\mathcal{E}}$ also contains the rule*
$$A^j_{\text{neg}} \leftarrow \textbf{not}\,\text{rej}(A^j_{\text{neg}}, 0).$$
5. ***Rejection rules.*** *For every rule of $P^j_{\mathcal{E}}$ of the form*
$$L^j \leftarrow \mathit{Body}, \textbf{not}\,\text{rej}(L^j, i).^{[7]}$$
*$P^j_{\mathcal{E}}$ also contains the rules*
$$\text{rej}(L^j_{\text{neg}}, p) \leftarrow \mathit{Body}. \tag{1}$$
$$\text{rej}(L^j, q) \leftarrow \text{rej}(L^j, i). \tag{2}$$
*where:*
   *(a) $p \leq i$ is the largest index such that $P^j_{\mathcal{E}}$ contains a rule with the literal $\textbf{not}\,\text{rej}(L^j_{\text{neg}}, p)$ in its body. If no such $p$ exists, then (1) is not in $P^j_{\mathcal{E}}$.*
   *(b) $q < i$ is the largest index such that $P^j_{\mathcal{E}}$ contains a rule with the literal $\textbf{not}\,\text{rej}(L^j, q)$ in its body. If no such $q$ exists, then (2) is not in $P^j_{\mathcal{E}}$.*
6. ***Totality constraints.*** *For all $i \in \{1, 2, \ldots, j\}$ and every atom $A \in \mathcal{L}_{\text{assert}}$ such that $P^j_{\mathcal{E}}$ contains rules of the form*
$$A^j \leftarrow \mathit{Body}_p, \textbf{not}\,\text{rej}(A^j, i).$$
$$A^j_{\text{neg}} \leftarrow \mathit{Body}_n, \textbf{not}\,\text{rej}(A^j_{\text{neg}}, i).$$
*$P^j_{\mathcal{E}}$ also contains the constraint*
$$u \leftarrow \textbf{not}\,u, \textbf{not}\,A^j, \textbf{not}\,A^j_{\text{neg}}.$$

A thorough explanation of this transformation can be found in [33].

Also in [33] it is proved that there is a one-to-one relation between the stable models of the so transformed normal program and the stable models of the original EVOLP program. From that result and Theorem 2, it follows that the composition of these two transformations is correct, i.e. that the stable models of the resulting normal logic program correspond to the stable models of the original EVOLP$_T$ program plus events.

Our implementation relies on the composed transformation described above. More precisely, the basic implementation takes an EVOLP$_T$ program and a sequence of events and preprocesses it into a normal logic program.

The preprocessor available at `http://centria.di.fct.unl.pt/~jja/updates/` is implemented in Prolog. It takes a file that starts with an EVOLP$_T$ program, where the syntax is just as described in Def. 6, except that the rule

[7] It can be a rewritten program rule, a rewritten event rule or an assertable rule (default assumptions never satisfy the further conditions). The set *Body* contains all literals from the rule's body except the $\textbf{not}\,\text{rej}(L^j, i)$ literal.

symbol is `<-` and the symbols `previous/1`, `sometime/1`, `since/2`, `always/1` are used instead of $\bigcirc(G)$, $\Diamond(G)$, $\mathbf{S}(G_1, G_2)$ and $\Box(G)$, respectively. The EVOLP$_T$ program is ended by a fact `newEvents`. The rules after this fact constitute the first event, which is again ended by a fact `newEvents`, after which the rules for the second event follow, etc.

For efficiency reasons, the preprocessor applies both transformations simultaneously, rather than applying them in sequence as described in the previous subsection. This is done by a simple combination of the sequences of steps from each of the transformations. Moreover, instead of creating new atomic names, as done in both transformations, the preprocessor uses Prolog terms for the new propositions accounting for the b-literals (e.g. it uses a term `sometime(p)` instead of $'\Diamond(p)'$ or $'sometime(p)'$), which eases the processing of nested temporal operators. Similarly, it adds an argument $j$ to atoms, instead of creating new propositions of the form $A^j$, as in the second transformation.

The programs obtained by the Prolog preprocessor can then run in any answer-set solver to obtain the set of the stable models of the original EVOLP$_T$ program and events. We have tested the implementation using the lparse grounder and the smodels solver(`http://www.tcs.hut.fi/Software/smodels/`). The implementation can also take advantage of the implementation of EVOLP and interface described in [32]. For this, we provide a version that starts by applying the first transformation, then feeds the result to the (java-based) implementation from [32] which in turn performs the second transformation and computes the stable models (using smodels).

## 5.2 Query-Answering under WF Semantics Implementation

Instead of computing the stable models of the resulting normal program, one may compute its well-founded model [16]. This provides a (3-valued) model which is sound, though not complete, w.r.t. the intersection of all stable models and thus also with respect to the truth relation of Def. 11. That is, for programs with stable models, if an atom $A(n)$ belongs to the well founded model then $A$ is true in all stable models of the program and events after $n$ steps; if $not\, A(n)$ belongs to the well founded model then $A$ belongs to no stable models after $n$ steps; if neither $A(n)$ nor $not\, A(n)$ belong to the well founded model, then nothing can be concluded.

Despite the incompleteness, the well founded model has the advantage of having polynomial complexity and allowing for (top-down) query-driven procedures. With this in mind, we have done another implementation, also available online, that besides the preprocessor also includes a meta-interpreter that answers existential queries under the well founded semantics. The meta-interpreter is implemented in XSB-Prolog, and relies on its tabling mechanisms for computing the well founded model. For top-down querying we provide a top goal predicate `G after I in (N1,N2)` which, given a goal `G` and two integers `N1` and `N2`, returns in `I` all integers between `N1` and `N2` such that in all stable models after `I` steps, `G` is true. This XSB-Prolog implementation also allows for a more interactive usage, e.g. allowing to add events as they occur (and adjusting the transformation on the fly), separately from the initial programs, and querying the current program (after as many steps as the number of events given).

## 6 Related Work and Conclusions

We have introduced the language EVOLP$_T$ for representing and reasoning about evolving knowledge bases with non-Markovian dynamics. The language generalizes its predecessor EVOLP by providing rules that may refer to the past states in a knowledge base evolution through Past LTL modalities. In addition to defining a syntax and semantics for the new language, we show, through a syntactic transformation, that an evolving logic program in EVOLP$_T$ can be compiled into a regular program in EVOLP. The latter is thus proved to be expressive enough to capture non-Markovian evolving knowledge bases as defined above.

The use of temporal logic in computer science is widespread. Here we would like to mention some of the most closely related work. Eiter et al. [11] present a very general framework for reasoning about evolving knowledge bases. This abstract framework allows the study of different approaches to logic programming knowledge base update, including those specified in LUPS, EPI, and KABUL. For the purpose of verifying properties of evolving knowledge bases in this language, they define a syntax and semantics for Computational Tree Logic (CTL), a branching temporal logic, modalities. While in [11] temporal logic is only used for verifying meta-level properties, in EVOLP$_T$ temporal operators are used in the object language to specify the behavior of an evolving knowledge base.

In the area of reasoning about actions, [28] and [20] describe extensions of the action language $\mathcal{A}$ with Past LTL operators, which allows formalizing actions whose effects depend on the evolution of the described domain. On a similar vein but in the more expressive situation calculus, [12, 14] shows a generalization of Reiter's Basic Action Theories for systems with non-Markovian dynamics and [13] introduces a transformation, somewhat similar to ours, of non-Markovian Basic Action Theories into traditional Markovian ones. Both the action language and the situation calculus based formalisms provide languages that can refer to past states in the evolution of a dynamic system. However, the focus of these formalisms is on solving the *projection problem*, i.e., reasoning about what will be true in the resulting state after executing a sequence of actions. On the other hand, the focus in the EVOLP$_T$ language is specifying updates to the system's knowledge base itself due to internal or external influence. For example, a system formalized in EVOLP$_T$ would be able to modify the description of its own behavior, which is not possible in $\mathcal{A}$ or in Basic Action Theories.

Also designed for specifying dynamic systems using temporal logic is the multi-agent language MetateM [8]. A program in this language consists of rules of the form $P \Rightarrow F$, where $P$ is a Past LTL formula and $F$ is a Future LTL formula. Intuitively, such a rule evaluated in a state specifies that if the evolution of the system up to this state satisfies $P$, then the system must proceed in such a way that $F$ be satisfied. EVOLP$_T$ does not include Future LTL connectives (our future work) so METATEM is more expressive in that sense. On the other hand, METATEM does not have a construct for updates and it is monotonic, unlike EVOLP$_T$. In [7] the authors propose a non-monotonic extension of LTL with the purpose of specifying agent's goals. Whereas [7] share with our work the the use of LTL operators and non-monotonicity, like METATEM it provides future operators, but the non-monotonic character in [7] is given by limited explicit exceptions to rules, thus appearing to be less general than our proposal.

## References

1. Alferes, J.J., Banti, F., Brogi, A.: From logic programs updates to action description updates. In: Leite, J., Torroni, P. (eds.) CLIMA 2004. LNCS (LNAI), vol. 3487, pp. 52–77. Springer, Heidelberg (2005)
2. 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)
3. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 50–61. Springer, Heidelberg (2002)
4. Alferes, J.J., Gabaldon, A., Leite, J.: Evolving logic programming based agents with temporal operators. In: IEEE/WIC/ACM Int'l Conf. on Intelligent Agent Technology (2008)
5. 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)
6. Alferes, J.J., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: LUPS – a language for updating logic programs. Artificial Intelligence 138(1&2) (June 2002)
7. Baral, C., Zhao, J.: Non-monotonic temporal logics for goal specification. In: IJCAI 2007, pp. 236–242 (2007)
8. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: Metatem: A framework for programming in temporal logic. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 94–129. Springer, Heidelberg (1990)
9. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: A framework for declarative update specifications in logic programs. In: IJCAI 2001, pp. 649–654 (2001)
10. 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) (2002)
11. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: Reasoning about evolving nonmonotonic knowledge bases. ACM Trans. Comput. Log. 6(2), 389–440 (2005)
12. Gabaldon, A.: Non-markovian control in the situation calculus. In: AAAI 2002, pp. 519–524. AAAI Press, Menlo Park (2002)
13. Gabaldon, A.: Compiling control knowledge into preconditions for planning in the situation calculus. In: IJCAI 2003, pp. 1061–1066 (2003)
14. Gabaldon, A.: Non-Markovian Control in the Situation Calculus. Artificial Intelligence 175(1), 25–48 (2011)
15. Gabbay, D., Smets, P. (eds.): Handbook of Defeasible Reasoning and Uncertainty Management Systems. Belief Change, vol. 3. Kluwer, Dordrecht (1998)
16. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)

17. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, ch. 7, pp. 285–316. Elsevier, Amsterdam (2008)
18. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: 7th Int'l Conf. on Logic Programming (1990)
19. Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on Artificial Intelligence 3, 195–210 (1998)
20. Gonzalez, G., Baral, C., Gelfond, M.: Alan: An action language for non-markovian domains. In: NonMon. Reasoning, Action and Change Workshop (2003)
21. Leite, J.: Playing with rules. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 1–19. Springer, Heidelberg (2010)
22. Leite, J., Soares, L.: Evolving characters in role-playing games. In: EMCSR 2006, vol. 2, pp. 515–520 (2006)
23. Leite, J., Soares, L.: Adding evolving abilities to a multi-agent system. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 246–265. Springer, Heidelberg (2007)
24. Leite, J.A.: Evolving Knowledge Bases. IOS Press, Amsterdam (2003)
25. Leite, J., Moniz Pereira, L.: Generalizing updates: From models to programs. In: Dix, J., Moniz Pereira, L., Przymusinski, T.C. (eds.) LPKR 1997. LNCS (LNAI), vol. 1471, p. 224. Springer, Heidelberg (1998)
26. Lifschitz, V., Woo, T.: Answer sets in general nonmonotonic reasoning (preliminary report). In: KR 1992 (1992)
27. Marek, V., Truszczynski, M.: Revision programming. Theor. Comput. Sci. 190(2), 241–277 (1998)
28. Mendez, G., Lobo, J., Llopis, J., Baral, C.: Temporal logic and reasoning about actions. In: 3rd Symp. on Logical Formalizations of Commonsense Reasoning (1996)
29. Saias, J., Quaresma, P.: A methodology to create legal ontologies in a lp based web information retrieval system. Artif. Intell. Law 12(4), 397–417 (2004)
30. Sakama, C., Inoue, K.: Updating extended logic programs through abduction. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) LPNMR 1999. LNCS (LNAI), vol. 1730, p. 147. Springer, Heidelberg (1999)
31. Šefránek, J.: Irrelevant updates and nonmonotonic assumptions. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS (LNAI), vol. 4160, pp. 426–438. Springer, Heidelberg (2006)
32. Slota, M., Leite, J.: Evolp: An implementation. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 288–298. Springer, Heidelberg (2008)
33. Slota, M., Leite, J.: Evolp: Tranformation-based semantics. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 117–136. Springer, Heidelberg (2008)
34. Slota, M., Leite, J.: On semantic update operators for answer-set programs. In: ECAI 2010, pp. 957–962. IOS Press, Amsterdam (2010)
35. Zhang, Y., Foo, N.Y.: Updating logic programs. In: ECAI 1998. John Wiley & Sons, Chichester (1998)

# On Representing Actions in Multi-agent Domains

Chitta Baral and Gregory Gelfond

Department of Computer Science and Engineering
Arizona State University

**Abstract.** Reasoning about actions forms the foundation of prediction, planning, explanation, and diagnosis in a dynamic environment. Most of the research in this field has focused on domains with a single agent, albeit in a dynamic environment, with considerably less attention being paid to multi-agent domains. In a domain with multiple agents, interesting issues arise when one considers the knowledge of various agents about the world, as well about as each other's knowledge. This aspect of multi-agent domains has been studied in the field of dynamic epistemic logic. In this paper we review work by Baltag and Moss on multi-agent reasoning in the context of dynamic epistemic logic, extrapolate their work to the case where agents in a domain are classified into three types and suggest directions for combining ideas from dynamic epistemic logic and reasoning about actions and change in order to obtain a unified theory of multi-agent actions.

## 1 Introduction

Actions, when executed, often change the state of the world. Reasoning about actions enables us to predict whether a given sequence of actions is indeed going to achieve some goal; it allows us to plan, or obtain a sequence of actions that would achieve a particular goal; it grants us the ability to explain observations in terms of actions that may have taken place; and it allows us to diagnose faults in a system by determining those actions whose consequences may have lead to them. When actions have non-deterministic effects, higher level reasoning about them is needed to verify and construct policies, enabling us to maintain various properties in addition to achieving certain goals.

As the number of states within a domain is often exponential in terms of the number of fluents (individual properties of the world), a central aspect in reasoning about actions is to develop languages which enable the concise representation of actions and their effects, and whose semantics define transitions between "states" due to their execution. In a single-agent domain, if we assume that an agent has complete knowledge of the values of the fluents, states may be thought of as *states of the world.* In the presence of incompleteness and *sensing actions*, states may be thought of as pairs which combine states of the world, together with an agent's *knowledge state.*

Considerable research has been done on the development of a class of languages, called *action languages*, that allow one to describe the world and the effects of various actions, and in using them for various tasks such as prediction, planning, counterfactual reasoning, as well as in proving the correctness of plans, policies and execution programs. The importance of reasoning about actions in the context of AI was mentioned as early as 1969 by McCarthy [10]. A systematic approach has evolved over the past twenty years which has been guided and influenced by the high level action language approach (beginning with the language $\mathcal{A}$ [7]), the approach of Sandewall [12], and the approach of the Toronto school [9,11].

With few exceptions, [13,8,6], the majority of these approaches have assumed that actions were performed by a single agent. Even in instances where multiple agents have been referred to, their interactions have been kept simple [3] (e.g., two agents simultaneously lifting a large table). In the real world however, interactions between multiple agents are more involved. One key issue that presents itself in multi-agent domains is the potential for discrepancies between the beliefs of various agents due to their different *frames of reference*.

While this issue has not been studied thoroughly within the reasoning about actions community, it has been explored in a somewhat different setting by the dynamic epistemic logic community [4,1]. In this paper we review work by Baltag and Moss [1] and discuss how some of its ideas can be applied to the reasoning about actions setting. We then pose some questions that need to be addressed in order to perform various reasoning tasks in a multi-agent domain.

The rest of the paper is structured as follows. In Sect. 2, we give a bit of background on modal logic and Kripke models. In Sect. 3, we introduce Baltag and Moss's action models and present a few examples (taken from the literature) of their use for modeling multi-agent actions. In Sect. 4 we show how Baltag and Moss's action models can be used to express multi-agent actions involving three classes of agents. In Sect. 5 we discuss some of our concerns with this approach and suggest some ways to overcome them and present an avenue for developing a high-level language to represent and reason about actions in a multi-agent setting. We then conclude with some final thoughts.

## 2 Background: Modal Logic and Kripke Models

We begin our discussion with an overview of modal logic, which forms the foundation of many of the concepts discussed in this work.

**Definition 1 (Multi-Agent Domain).** *A* multi-agent domain *is defined as a triple, $(\mathcal{AG}, \mathcal{F}, \mathcal{A})$, where $\mathcal{AG}$ is a finite, non-empty set of agent names, $\mathcal{F}$ is a finite, non-empty set of propositional atoms, and $\mathcal{A}$ is a finite, non-empty set of action names.*

Intuitively, $\mathcal{AG}$ defines the set of agents who are operating in the domain, $\mathcal{F}$ defines the physical properties of the domain, and $\mathcal{A}$ denotes the actions which the agents are capable of performing. Given a multi-agent domain, various properties of the domain are described by modal formulae.

**Definition 2 (Modal Formula).** *Given a multi-agent domain, $(\mathcal{AG}, \mathcal{F}, \mathcal{A})$, a* modal formula *is defined as follows:*

- *if $\phi \in \mathcal{F}$, then $\phi$ is modal formula*
- *if $\phi$ is a modal formula, then $\neg\phi$ is a modal formula*
- *if $\phi$ is a modal formula, then $\mathbf{K}_i\phi$, where $i \in \mathcal{AG}$, is a modal formula*
- *if $\phi$ is a modal formula, then $\mathbf{E}_\alpha\phi$, where $\alpha \subseteq \mathcal{AG}$, is a modal formula*
- *if $\phi$ is a modal formula, then $\mathbf{C}_\alpha\phi$, where $\alpha \subseteq \mathcal{AG}$, is a modal formula*
- *if $\phi$ and $\psi$ are modal formulae, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \Rightarrow \psi$, and $\phi \Leftrightarrow \psi$ are modal formulae*

Intuitively, a modal formula of the form $\mathbf{K}_i\phi$ means that "agent $i$ knows $\phi$". Modal formulae of the form $\mathbf{E}_\alpha\phi$, mean that "every agent in the group $\alpha$ knows $\phi$." Lastly, modal formulae of the form $\mathbf{C}_\alpha\phi$, mean that "$\phi$ is common knowledge amongst the agents in $\alpha$."[1] Before defining the entailment relation, it is necessary to define the notions of a *Kripke structure*, and a *Kripke world*.

**Definition 3 (Kripke Structure).** *A* Kripke structure*, $M$, over a set of agents $\mathcal{AG}$, and a set of fluents $\mathcal{F}$, is defined as a tuple $(S, \pi, \{R_i \mid i \in \mathcal{AG}\})$ where:*

- *$S$ is a non-empty set of state symbols (or possible worlds)*
- *$\pi$ is a function mapping elements of $S$ onto interpretations of $\mathcal{F}$*
- *each $R_i$ is a binary relation on $S$ (called an accessibility relation)*

Intuitively, a pair $(\sigma, \tau) \in R_i$ is understood to mean that from within possible world $\sigma$, agent $i$ cannot distinguish between $\sigma$ and $\tau$. Depending on the modality of discourse, the accessibility relations have different properties, which correspond to axiom systems associated with that modality. For example, the axiom system S5 is often used with the *knowledge* modality, in which case the accessibility relations must be *reflexive*, *symmetric*, and *transitive*. On the other hand, the axiom system KD45 is often used with the *belief* modality, in which case the accessibility relations are *euclidean*, *serial*, and *transitive*.

**Definition 4 (Kripke World).** *A* Kripke world *is defined as a pair $(M, \sigma)$, where $M$ is a Kripke structure, and $\sigma$ is a state symbol of $M$. $\sigma$ is said to be the state which designates* the real physical state of the world.

Having defined the notions of a Kripke structure and a Kripke world, we can define the semantics of modal logic.

**Definition 5 (Entailment Relation for Modal Formulae).** *Let $\phi$ and $\psi$ be modal formulae, and let $(M, \sigma)$ be a Kripke world over a multi-agent domain $D = (\mathcal{AG}, \mathcal{F}, \mathcal{A})$. The* entailment relation for modal formulae *is defined as follows:*

- *if $\phi \in \mathcal{F}$, then $(M, \sigma) \models \phi$ if and only if $\pi(\sigma)(\phi) = \top$*
- *$(M, \sigma) \models \neg\phi$ if and only if $(M, \sigma) \not\models \phi$*
- *$(M, \sigma) \models \phi \wedge \psi$ if and only if $(M, \sigma) \models \phi$ and $(M, \sigma) \models \psi$*

[1] The readings of the various modal formulae use the term *knows* in order to remain consistent with prior work done in the field of dynamic epistemic logic.

- $(M,\sigma) \models \phi \lor \psi$ *if and only if* $(M,\sigma) \models \phi$ *or* $(M,\sigma) \models \psi$
- $(M,\sigma) \models \phi \Rightarrow \psi$ *if and only if* $(M,\sigma) \models \neg\phi \lor \psi$
- $(M,\sigma) \models \phi \Leftrightarrow \psi$ *if and only if* $(M,\sigma) \models (\phi \Rightarrow \psi) \land (\psi \Rightarrow \phi)$
- $(M,\sigma) \models \mathbf{K}_i\phi$ *if and only if* $(M,\tau) \models \phi$ *for all* $\tau$ *such that* $(\sigma,\tau) \in R_i$
- $(M,\sigma) \models \mathbf{E}_\alpha\phi$ *if and only if* $(M,\sigma) \models \mathbf{K}_i\phi$ *for every agent* $i \in \alpha$

*Let* $\mathbf{E}^0_\alpha\phi$ *be an abbreviation for* $\phi$*, and let* $\mathbf{E}^{k+1}_\alpha\phi = \mathbf{E}^k_\alpha\mathbf{E}_\alpha\phi$*.*

- $(M,\sigma) \models \mathbf{C}_\alpha\phi$ *if and only if* $(M,\sigma) \models \mathbf{E}^k_\alpha\phi$ *for* $k = 0, 1, 2, \ldots$

*Example 1 (The Strongbox Domain of [1]).* Consider a domain in which two agents, $A$ and $B$, are together in a room containing a strongbox in which there is a coin. This fact is common knowledge amongst the agents, as is the fact that none of them knows which face of the coin is showing. Suppose that the coin is actually lying heads up. The Kripke world shown in Fig. 1 represents the initial configuration of the world and the agents' knowledge about the world. In the figure, circles represent states and a double circle represents the real physical state of the world. The Kripke world in Fig. 1 entails the following modal formulae (among others):

- $(M,\sigma) \models H$
- $(M,\sigma) \models \neg\mathbf{K}_A H \land \neg\mathbf{K}_A\neg H$
- $(M,\sigma) \models \neg\mathbf{K}_B H \land \neg\mathbf{K}_B\neg H$
- $(M,\sigma) \models \mathbf{C}_{\{A,B\}}(\neg\mathbf{K}_A H \land \neg\mathbf{K}_A\neg H \land \neg\mathbf{K}_B H \land \neg\mathbf{K}_B\neg H)$

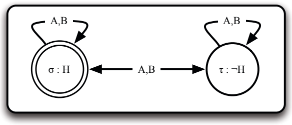


**Fig. 1.** Kripke world for the Strongbox Domain. Neither A nor B know the real world; the external observer knows that $H$ is true in the real world.

## 3 Baltag-Moss Action Models

In a multi-agent setting, the agents in the domain may have differing frames of reference with respect to an action. Continuing with the Strongbox Domain, agent $A$ may peek into the box to find that the coin is showing heads with agent $B$ watching this take place. Baltag and Moss [1] describe a construct called an *action model* which they use to represent these differences of perspective. Moreover, they define a means by which an action model may be used to *update* a Kripke world in order to obtain a successor world modeling the effects of the execution of the underlying action.

Intuitively, an action model may be thought of as a directed graph, whose nodes are referred to as simple actions, and are labeled by modal formulae representing their preconditions. The edges of an action model are labeled by the names of the agents in the domain, and are meant to convey the frames of reference of the agents with respect to the complex action described by the model.

**Definition 6 (Action Model).** *An* action model*, $\mathbf{\Sigma}$, over a set of agents $\mathcal{AG}$, is defined as a tuple $(\Sigma, s, \{R_i \mid i \in \mathcal{AG}\}, pre)$*

- *$\Sigma$ is a set whose elements are called simple actions*
- *$s \subseteq \Sigma$ is a set of designated simple actions*
- *each $R_i$ is a binary relation on $\Sigma$*
- *pre is a function mapping each simple action $\alpha \in \Sigma$ onto a modal formula representing its preconditions*

Action models correspond to individual occurrences of a complex, knowledge producing action[2], and induce an *update operation* on the Kripke worlds describing the *state of the world* prior to its execution. Within this paper they are represented as directed graphs, whose nodes correspond to the set of simple actions (and are labeled by the modal formulae describing their preconditions), and whose arcs are defined by the binary relations over $\Sigma$. Prior to defining the update operation, we introduce the notion of a *knowledge state.*

**Definition 7 (Knowledge State).** *A* knowledge state *in a multi-agent domain is defined as a set of Kripke worlds that have the same underlying Kripke structure and differ only in the state representing the physical state of the world. The different physical state of the world are from the frame of reference of an external observer.*

*Example 2 (Knowledge State in the Strongbox Domain).* Consider a domain in which two agents, $A$ and $B$, are together in a room containing a strongbox in which there is a fair coin. This fact is common knowledge amongst the agents, as is the fact that none of them knows which face of the coin is showing. The graph shown in Fig. 2 shows the knowledge state of the agents $A$ and $B$. Notice

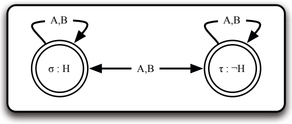


**Fig. 2.** A knowledge state in the Strongbox Domain; neither $A$, nor $B$ and nor the external observer know the real world

that from our frame of reference as external observers, we ourselves do not know which state corresponds to the real physical state of the world, and consequently both $\sigma$ and $\tau$ are marked as possibilities.

For notational convenience, given a knowledge state $\mathbf{S}$, let $\mathbf{S}[S]$ denote the set of state symbols in $\mathbf{S}$; $\mathbf{S}[s]$ denote the set of state symbols in $\mathbf{S}$ describing the possible real physical states of the world; $\mathbf{S}[\pi]$ denote the interpretation function

[2] Baltag and Moss limit their discussion in [1] to actions which affect the knowledge of the agents, as opposed to the physical state of the domain, hence the term *knowledge producing action.*

in $\mathbf{S}$; and $\mathbf{S}[R_i]$ denote the accessibility relation for agent $i$ in $\mathbf{S}$. Similarly, given an action model $\mathbf{\Sigma}$, let $\mathbf{\Sigma}[S]$ denote the set of action symbols in $\mathbf{\Sigma}$; $\mathbf{\Sigma}[s]$ denote the set of action symbols in $\mathbf{\Sigma}$ describing the designated simple actions; and $\mathbf{\Sigma}[R_i]$ denote the relation for agent $i$ in $\mathbf{\Sigma}$.

**Definition 8 (Update Operation Induced by an Action Model).** *Let* $\mathbf{S}$ *denote a knowledge state within a multi-agent domain, and let* $\mathbf{\Sigma}$ *be an action model for a knowledge producing action. The* update operation, *induced by* $\mathbf{\Sigma}$, *is defined as a new knowledge state,* $O(\mathbf{S}) = \mathbf{S} \otimes \mathbf{\Sigma}$ *where:*

- $O(\mathbf{S})[S] = \{(\sigma, \tau) : \sigma \in \mathbf{S}[S], \tau \in \mathbf{\Sigma}[S]$ *and* $(M, \sigma) \models pre(\tau)\}$, *where* $M$ *is the underlying Kripke structure of* $\mathbf{S}$
- $((\sigma, \tau), (\sigma', \tau')) \in O(\mathbf{S})[R_i]$ *if and only if* $(\sigma, \sigma') \in \mathbf{S}[R_i]$ *and* $(\tau, \tau') \in \mathbf{\Sigma}[R_i]$
- $(\sigma, \tau) \in O(\mathbf{S})[s]$ *if and only if* $(\sigma, \tau) \in O(\mathbf{S})[S]$ *and* $\sigma \in \mathbf{S}[s]$, $\tau \in \mathbf{\Sigma}[s]$
- $O[\pi]((\sigma, \tau)) = \mathbf{S}(\pi)(\sigma)$

We now present several action models from [1] and their impact on the knowledge state of the Strongbox Domain shown in Fig. 1. Throughout this paper the examples of successor knowledge states obtained as a consequence of the update operation have been simplified (for example by removing states from $O(\mathbf{S})[S]$, along with their incoming and outgoing arcs, which are not reachable from those in $O(\mathbf{S})[s]$). Due to space considerations a full description of these simplification actions has been omitted.

## 3.1 Global Announcement Actions

The first class of an action scenario that we examine is that of *global announcements*, actions in which a new piece of information is made common knowledge amongst the agents of the domain[3].

*Example 3 (Global Announcement Actions).* Consider the variant of the Strongbox Domain mentioned in Ex. 2. Suppose that it is announced to both $A$ and $B$ that the coin is showing heads. The corresponding action model, $\mathbf{\Sigma_1}$, is shown in Fig. 3. Note that the intuitive reading of the model is that the action causes

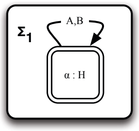


**Fig. 3.** Baltag-Moss action model for the global announcement of $H$ to $A$ and $B$

$H$ to be common knowledge to both $A$ and $B$. Furthermore, the real state of the world is the one in which $H$ is true. Application of the update operation yields the transition shown in Fig. 4.

The action model from Fig. 3 may be generalized to the global announcement of an arbitrary modal formula as shown in Fig. 5.

[3] It is assumed that global announcements only convey factually correct information.

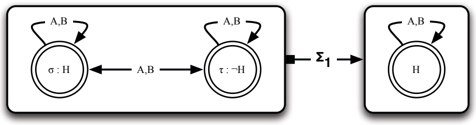


**Fig. 4.** Transition defined by $\mathbf{\Sigma_1}$, the global announcement that $H$ is true

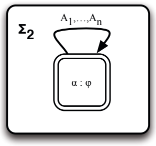


**Fig. 5.** Global announcement of a modal formula $\varphi$ to agents $\{A_1, \ldots, A_n\}$

## 3.2 Private Announcement Actions

In the previous subsection we considered announcement actions that were made globally. In private announcements the announcement is only made to a selected group of agents. The rest of the agents are oblivious about the announcement.

*Example 4 (Private Announcement Actions).* As before, we begin by considering the variant of the Strongbox Domain mentioned in Ex. 2. Suppose that agent $A$ peeks into the strongbox, learning that the coin is showing heads, and that agent $B$ is unaware of this. The corresponding action model, $\mathbf{\Sigma_3}$, is shown in Fig. 6. The intuitive reading of this model is that agent $A$ knows $H$ to be true in the

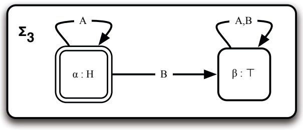


**Fig. 6.** Action model for agent $A$ observing $H$ unbeknownst to agent $B$

real physical state of the world, and also correctly believes that the knowledge of agent $B$ is unchanged (represented by the state labeled $\top$). The transition defined by application of the update operation is shown in Fig. 7.

The action model in Fig. 6 may be viewed as a special case of an action in which differing pieces of information are declared to the agents. Specifically, the model may be interpreted as the composition of two simple actions, the first of which announces a formula, $\varphi = H$ to agent $A$, while the second essentially models an announcement by agent $A$ that nothing has changed to agent $B$ (represented by the formula $\psi = \top$). With this reading in mind, the action model may be generalized in a fashion as shown in Fig. 8.

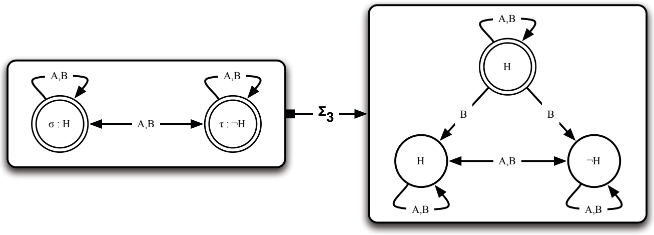


**Fig. 7.** Transition defined by $\mathbf{\Sigma_3}$

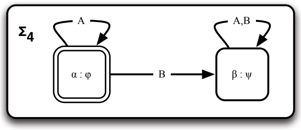


**Fig. 8.** Action model, $\mathbf{\Sigma_4}$: Agent $A$ learns $\varphi$, but agent $B$ believes that $\psi$ was announced to $A$ instead and $A$ is aware of $B$'s mistaken belief

Notice that in both Fig. 6 and Fig. 8, the *external observer* knows the actual state of the world (in the former, we are directly told that agent $A$ observes $H$, in the later this is extended to the formula $\varphi$).

## 3.3 Sensing Actions

In addition to acquiring new information about the domain through announcements, agents may perform *sensing actions* in order to learn more about their surroundings. The effect of sensing actions is not known in advance, consequently, the external observers do not a-priori know the result.

*Example 5 (Sensing Actions).* As before, consider the variant of the Strongbox Domain in which we have $n$ agents, $A_1, \dots, A_n$. Now suppose that agents $A_1, \dots, A_m$ peek into the box together, thereby learning which side of the coin is facing up while agents $A_{m+1}, \dots, A_n$ remain unaware of what has transpired. The corresponding action model $\mathbf{\Sigma_5}$ is shown in Fig. 9.

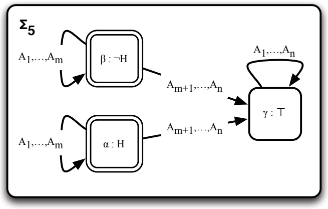


**Fig. 9.** Agents $A_1, \dots, A_m$ secretly sense value of $H$; agents $A_{m+1}, \dots, A_n$ are oblivious

All of the previous examples of sensing actions have assumed that agents within a domain could be partitioned into two categories: *actors*, and *oblivious agents*. Baltag and Moss also consider a different class of non-actors, who are not directly sensing but are (partially) observing the occurrence of the sensing actions. We call such agents *partial-observers* as they do not know the result of such an action, but do know that it occurred.

*Example 6 (Sensing Actions with Partial Observers).* Consider the variant of the Strongbox Domain from Ex. 5. Suppose however that agents $A_1, \ldots, A_m$ peek into the box together with the knowledge of agents $A_{m+1}, \ldots, A_n$ (e.g., agents $A_{m+1}, \ldots, A_n$ are watching this take place). The corresponding action model $\mathbf{\Sigma_6}$ is shown in Fig. 10.

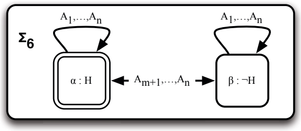


**Fig. 10.** Agents $A_1, \ldots, A_m$ sense $H$ to be true; agents $A_{m+1}, \ldots, A_n$ are aware of $H$ being sensed but not the value that was sensed

In addition to the deterministic sensing actions discussed before, there are also *nondeterministic sensing actions*. Such actions represent the notion of an agent possibly performing a sensing action.

*Example 7 (Nondeterministic Sensing Actions).* Consider the Strongbox Domain from Ex. 5. Suppose that agents $A_1, \ldots, A_m$ may have peeked into the box, *possibly determining* the face of the coin, and that the remaining agents $A_{m+1}, \ldots, A_n$, are aware of this possibility. The corresponding action model $\mathbf{\Sigma_7}$ is shown in Fig. 11.

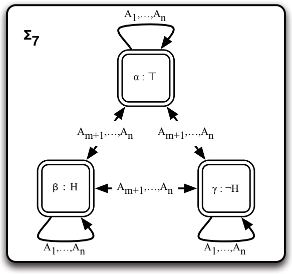


**Fig. 11.** Agents $A_1, \ldots, A_m$ may have peeked into the box; agents $A_{m+1}, \ldots, A_n$, are aware of this possibility

A further variant of sensing deals with the case where one agent believes that some other agent may have sensed the value of a fluent. The actions models representing two instances of this scenario are given in Fig. 12 and Fig. 13.

Fig. 12 deals with the case where the mistaken belief of one agent is later proven to be true, while Fig. 13 deals with the case in which one agent has lied to another. Both scenarios are modeled as the sequential composition of two separate actions, the first of which (given by $\mathbf{\Sigma_8}$ and $\mathbf{\Sigma_{10}}$ respectively) is common to both of them. This action in essence *sets the stage* for its successor, by allowing the possibility for one agent to have potentially erroneous information about the domain.

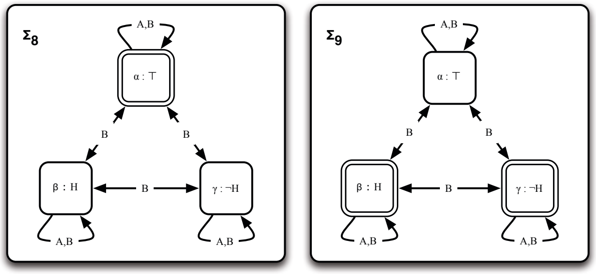


**Fig. 12.** $\mathbf{\Sigma_8}$: Agent $B$ mistakenly believes that agent $A$ has peeked into the box. $\mathbf{\Sigma_9}$: Agent $A$ has indeed peeked and agent $B$ correctly believes this to be the case.

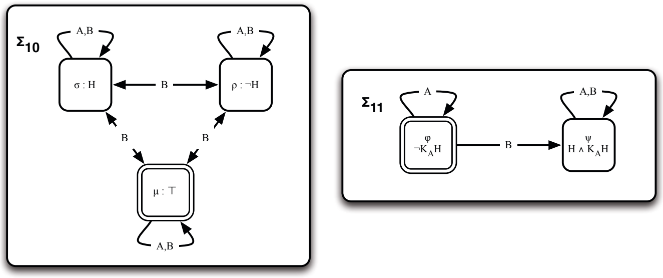


**Fig. 13.** $\mathbf{\Sigma_{10}}$: Agent $B$ mistakenly believes that agent $A$ may have peeked. $\mathbf{\Sigma_{11}}$: Agent $A$ does not know the value of $H$ but successfully lies to $B$ that it knows and the value is $H$.

## 4 Using Baltag-Moss Action Models to Express Three Classes of Agents

The various action models in the previous section are from [1] and other works. They propose a notion of action signatures generalizing those action models. They also propose a notion of program models built using action models and action signatures. In this section we do a generalization of their actions in a different dimension.

Broadly speaking, the action models described by Baltag and Moss separate the agents into two groups. Each action defines a set of *actors*, representing the set of agents who are "recipients" of the action's direct effects. In addition, an action either defines a set of agents who are *observers*, i.e., agents who are aware that some action has occurred, and as a consequence receive the action's indirect effects; or who are *oblivious*, i.e., are completely unaware that the action has transpired, and whose knowledge is therefore unchanged.

It is often the case however that two classes of agents are not enough to fully express the consequences of a knowledge producing action. Consider for example a variant of the Strongbox Domain consisting of three agents, $A$, $B$, and $C$. In this case it is quite natural to envision a scenario in which agent $A$ performs some action, with agent $B$ observing, and agent $C$ being completely oblivious as to what has transpired.

## 4.1 Extending Announcement Actions

The global announcements described by Baltag and Moss may be thought of as a special case of an *announcement*, where all of the agents within the domain are actors. Suppose however, that only a specific subset of the agents receives the announcement. In this case the action is more complex, with differing effects depending upon how the agents are grouped with respect to their *awareness of the action occurrence.*

*Example 8 (Announcements with Partial Observers).* Consider a variant of the Strongbox Domain comprised of three agents, $A$, $B$, and $C$, and in which the initial knowledge state is unchanged. Suppose that it is announced to agent $A$ that the coin is showing heads. Furthermore, let us suppose that agent $B$ is aware that some piece of information concerning the coin has been announced to agent $A$, and that agent $C$ is oblivious to what has transpired. The action model, $\mathbf{\Sigma_{12}}$, is shown in Fig. 14.

Applying the update yields the transition shown in Fig. 15.

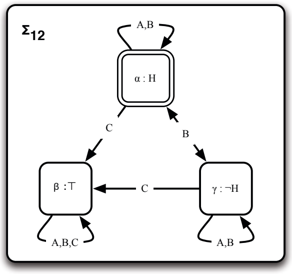


**Fig. 14.** Action model for the announcement of $H$ to $A$ with agent $B$ as an observer, and an oblivious agent $C$

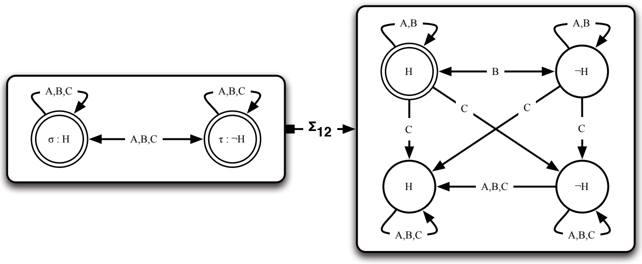


**Fig. 15.** Transition defined by $\mathbf{\Sigma_{12}}$

Note that in the resulting Kripke world, agent $A$ knows that $H$ is true; agent $B$ knows that agent $A$ knows the value of $H$ but does know what the value is; however, as far as agent $C$ is concerned, he believes that none of the agents know the value of $H$.

## 4.2 Sensing Actions with Three Classes of Agents

As with communication actions, sensing actions also need to be extended to the case where all three classes of agents are present.

*Example 9 (Extended Sensing Actions).* Consider the the Strongbox Domain in which we have three agents, $A$, $B$, and $C$. Suppose that agent $A$ performs a *sensing action* (as originally introduced in Ex. 5), with agent $B$ observing the occurrence but not the result, and agent $C$ oblivious to what has transpired. Fig. 16 shows the corresponding action model, $\mathbf{\Sigma_{13}}$, for this action occurrence. Applying the update yields the transition shown in Fig. 17.

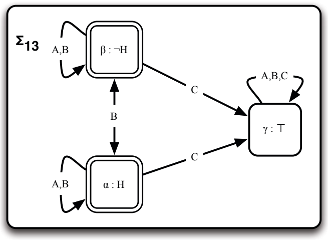


**Fig. 16.** Action model for a sensing action performed by agent $A$, with agent $B$ as a partial observer, and an oblivious agent $C$

As with sensing actions, we also extend nondeterministic sensing actions to the case where all three classes of agents are represented.

*Example 10 (Nondeterministic Sensing Actions).* As with Ex. 9, we begin with the Strongbox Domain comprised of three agents, $A$, $B$, and $C$. Suppose that

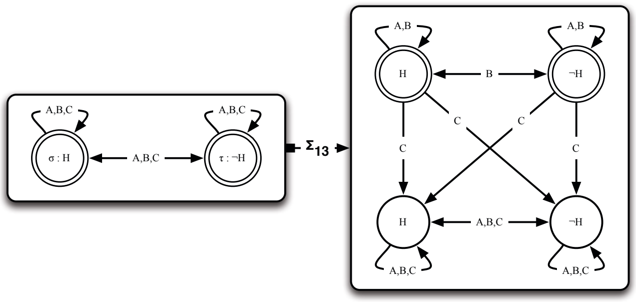


**Fig. 17.** Transition defined by $\mathbf{\Sigma_{13}}$

agent $A$ attempts to use a sensor (which may or may not work properly) to discern which face of the coin is showing. As before, let us suppose that agent $B$ watches this occur (but does not know the outcome) and that agent $C$ is oblivious. The action model, $\mathbf{\Sigma_{14}}$, for this scenario is shown in Fig. 18, and yields a considerably more complex successor state shown in Fig. 19.

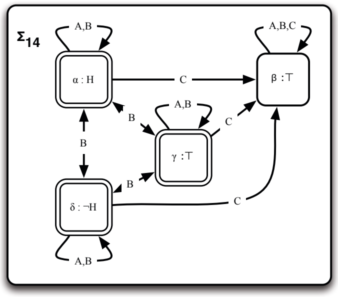


**Fig. 18.** Action model for agent $A$ attempting to sense the value of $H$ while partially observed by agent $B$ with $C$ unaware of what has happened

## 4.3 Actions of Misdirection with Three Classes of Agents

Lastly we consider the class of actions dealing with lying and misdirection in which all three categories of agents are present.

*Example 11 (Lying About the World).* Consider the Strongbox Domain comprised of three agents, $A$, $B$, and $C$, and suppose that as before, it is common knowledge that none of them know which face of the coin is showing. Suppose that agent $A$ lies to $B$, convincing that he knows the coin is truly showing heads, while agent $C$ is oblivious to what has occurred. As with the case in which we have two classes of agents, such an action is modeled by the *sequential composition* of two simpler actions: the first in which agent $B$ develops a suspicion that agent $A$ may know which face of the coin is showing; and the second in

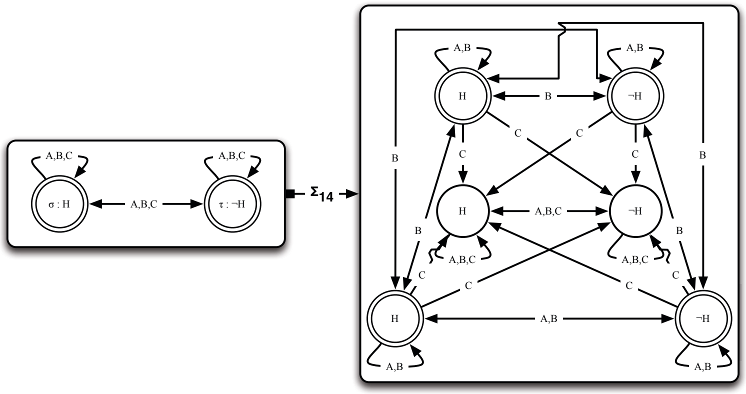


**Fig. 19.** Transition defined by $\mathbf{\Sigma_{14}}$

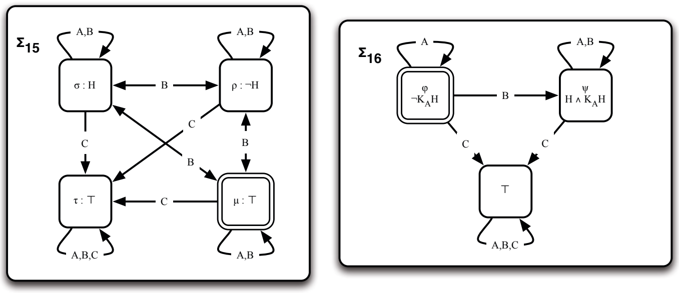


**Fig. 20.** Action models for lying about knowledge with three class of agents. $\mathbf{\Sigma_{15}}$: Agent $B$ believes that agent $A$ may have peeked. Agent $C$ is oblivious. $\mathbf{\Sigma_{16}}$: Agent $A$ does not know the value of $H$ but successfully lies to $B$ that it knows and the value is $H$. Agent $C$ is oblivious.

which $A$ actually lies. The action models for both of these actions, $\mathbf{\Sigma_{15}}$ and $\mathbf{\Sigma_{16}}$ respectively, are shown in Fig. 20 below. Applying the update operation induced by $\mathbf{\Sigma_{15}}$ to our initial state defines the intermediate transition shown Fig. 21. Once this intermediate state has been obtained, we apply the update operation induced by $\mathbf{\Sigma_{16}}$ in order to obtain our final successor state shown in Fig. 22.

In addition to lying about the value of a particular fluent, "The coin is facing heads.", agents may mislead each other about their knowledge with respect to a particular fluent, "I know which face of the coin is showing". The latter is captured in Ex. 12.

*Example 12 (Lying About Knowledge).* As with Ex. 11, we consider the Strongbox Domain consisting of three agents $A$, $B$, and $C$. Suppose that agent $A$ lies to agent $B$ by telling him that he knows which face of the coin is showing, while agent $C$ remains oblivious. As before, this scenario is modeled by the sequential

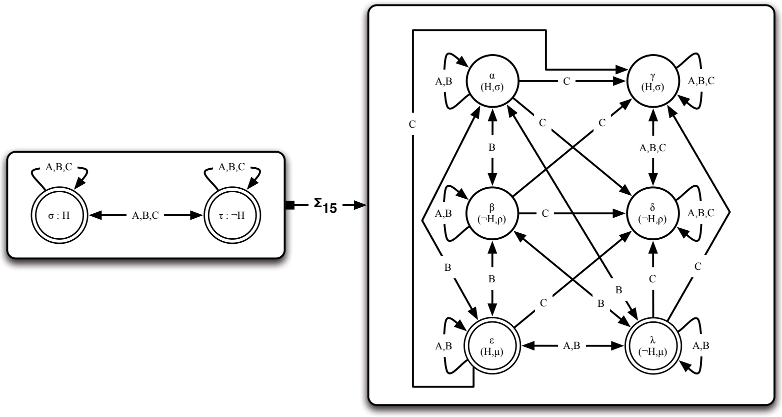


**Fig. 21.** Intermediate transition corresponding to agent $B$ suspecting that $A$ knows which face of the coin is showing, while agent $C$ is oblivious

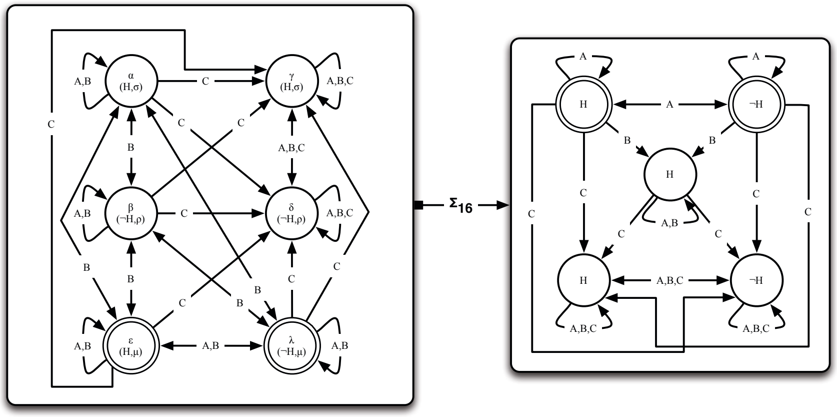


**Fig. 22.** Final transition for lying about the world with three classes of agents

composition of two simpler actions: the first being identical to the one discussed in Ex. 11, as is the intermediate transition; and the second being almost identical to the one discussed there as well, with the only change being in the modal formulae associated as preconditions to the component simple actions. The action model for this second action, $\mathbf{\Sigma_{17}}$, is shown in Fig. 23, and the final transition is shown in Fig. 24.

## 5 From Action Models to High Level Action Languages

In Sect. 3 we presented the Baltag-Moss action models and several examples of their use. We also presented several action signatures as defined by Baltag

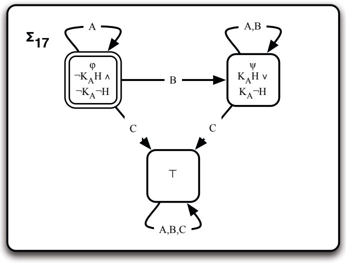


**Fig. 23.** Agent $A$ does not know the value of $H$ but successfully lies to $B$ that it may know. Agent $C$ is oblivious.

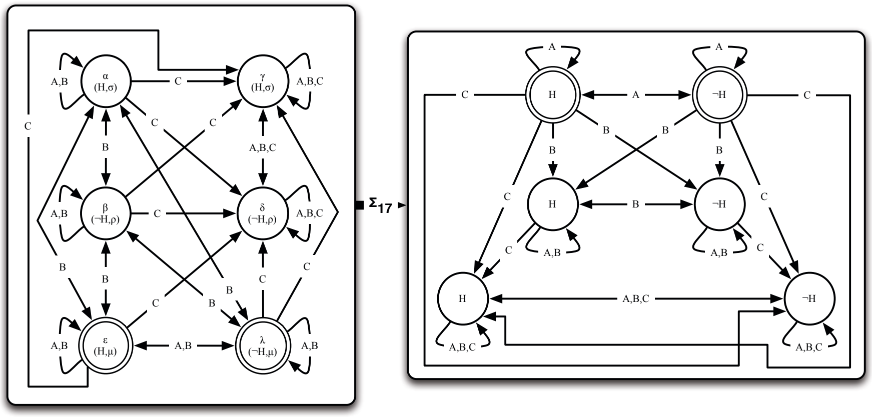


**Fig. 24.** Final transition when lying about knowledge with three classes of agents

and Moss. Action models and signatures seem to be good technical devices for expressing certain kinds of multi-agent actions. The update operations induced by action models are simple to understand and seem to be a natural extension of the set product operator to graphs with nodes labeled by propositional formulae. Baltag and Moss also give a language to construct programs, which they call program models, from action models.

We believe that although action models, action signatures, and program models are a good start, further work needs to be done to express multi-agent action scenarios.

## 5.1 Future Directions on Action Models

Currently, even though one could give meaning to actions models that are constructed from the given action signatures, we could find no work in the literature that gives meaning to individual nodes and edges of action models in such a way that they can be used to build up the overall meaning of an action model. Discovering or developing such a semantics would be very helpful.

Alternatively, developing a general algorithm to construct action models for certain kinds of action scenarios would also be helpful. This seems to be a promising direction as we were able to generalize the examples in Sect. 3 to come with action models for various interesting multi-agent action scenarios when there are three classes of agents. We presented those examples in Sect. 4. Our action model examples of Sect. 4 are simple extensions of the known action models; yet they are novel in the sense that they do not follow any of the known action signatures and lead to new action signatures.

### 5.2 A High Level Action Language That Uses Action Models as a Semantic Tool

Another representational aspect of action models that needs to be enhanced is that there is often a dependency between an agent's role with respect to an action scenario and the state of the world. For example, consider an enhancement of the Strongbox Domain, where agent $A$ has various actions at her disposal such as: (a) an action to distract another agent who is watching the strongbox and (b) an action to signal an agent that is not watching the strongbox to pay attention. Such actions when executed change subsequent action scenarios. For example, suppose that agent $C$ is initially watching the strongbox and agent $A$ distracts him. If agent $A$ then peeks into the strongbox, $C$ will be oblivious.

Such knowledge can be expressed in the style of high-level action language $\mathcal{A}$ [7] by *agent-role statements* of the following kinds:

– $Y$ **observes** $peek(X)$ **if** $lookingAt(Y,X), near(Y,X)$
– $Y$ **partially observes** $peek(X)$ **if** $lookingAt(Y,X), \neg near(Y,X)$

Using statements of the above kind one can determine who are the observers, partial observers, and oblivious agents with respect to an action like *peek*. Once that is determined one can then construct the appropriate action model and compute the transition due to that action model.

One can then borrow the other constructs of high level languages to express world changing actions as well as relationship between properties of the world. Some of those constructs are the following:

– Dynamic Causal Laws: $a$ **causes** $\phi$ **if** $\psi$
– Executability Conditions: **executable** $a$ **if** $\phi$
– Sensing Actions: $a$ **determines** $f$
– Non-deterministic Sensing Actions: $a$ **may determine** $\phi$
– Static Causal Laws: $\psi$ **if** $\phi$
– Initial State Axioms: **initially** $\psi$

Intuitive readings of the aforementioned statements are as follows. Dynamic causal laws state that when an action $a$ is performed in a state where $\phi$ is true, $\psi$ becomes true in the resulting state. Executability conditions state that an action $a$ is only executable in states where $\phi$ is true. Sensing actions state that after the performing the action $a$, the agent knows the value of the fluent $f$. Nondeterministic sensing actions are taken to mean that after performing the action $a$, the agent may know the value of the fluent $f$. Static causal laws state that all states which satisfy $\phi$ must also satisfy $\psi$. Initial state axioms state that $\psi$ is true in the initial state.

The above readings are appropriate for domains with one active agent. In the presence of multiple agents however, they may change slightly. For example, in a multi-agent domain the meaning of a sensing action is that after an occurrence of the action $a$, the value of the fluent $f$ becomes known to those agents who are observing that action, the fact that the action occurred is known to the agents who are partially observing the action, while all other agents are oblivious.

Thus a high level action language for multi-agent domains can be defined by combining existing constructs from action languages with new ones such as *agent-role statements*. The semantics of such a language would then be defined by generalizing the transition semantics of existing action languages with the addition that in the presence of multiple agents, an action is enhanced to an action model by the application of agent-role statements, and then the action model is used to compute the corresponding transition.

The above is a good first step in the development of a high level language for multi-agent domains. However, several other concerns need to be addressed.

Various reasoning about action tasks, such as planning, require the notion of an initial state. Since in a multi-agent domain the initial state axioms may contain modal formulae, and the Kripke worlds satisfying them may have an infinite number of states, there is a need to identify a subset of modal logic for use in these axioms such that: the Kripke worlds satisfying them are finite; can be constructed easily; and are able to express interesting and important domains from the multi-agent literature (for example, the muddy-children domain [5,2]). One such sub-language is where $\psi$ is a formula without modal operators, or has the form $\mathbf{C}\varphi$ or $\mathbf{C}(\mathbf{K}_i\varphi \vee \mathbf{K}_i\neg\varphi)$, where $\varphi$ is a formula without modal operators.

## 5.3 Knowledge and Belief

A major concern that needs to be addressed in the future is that even though we may begin with the knowledge modality, after a number of actions occur we begin to have a mixture of belief and knowledge. For example, consider $\mathbf{\Sigma_3}$ and its associated transition as shown in Fig. 7. Prior to the execution of that action, it is common knowledge among agents $A$ and $B$ that neither knows the value of $H$. After the execution of $\mathbf{\Sigma_3}$, $A$ knows that $H$ is true. But what about agent $B$'s knowledge? Per Baltag and Moss, the given reading of the successor state shown in Fig. 7 is that $B$ knows that $A$ does not know the value of $H$. However, it is often argued that one may only *know* truths, yet *believe* falsehoods. In this case, a more appropriate reading would be that $B$ believes that $A$ does not know the value of $H$. To be able to do this, one needs to capture both the knowledge and belief modalities. One possible approach would be to treat the Kripke models in [1] and this paper as a short hand for a more complex Kripke model that has accessibility relations covering both knowledge and belief. We are currently working on formalizing this structure.

## 6 Final Thoughts

As was mentioned in the introduction, considerable bodies of research with respect to both single, and multi-agent reasoning have been developed. With a few exceptions however ([13,8,6]), there has not been much crossover between these two areas. With regards to single-agent domains, a myriad of techniques has arisen, involving the use of action languages for reasoning about actions with application in planning, diagnosis, and other reasoning tasks. From the multi-agent standpoint, the use of modal logics and other methods has produced a body of work for reasoning about the knowledge and beliefs of the agents present in a domain.

The work done by Baltag and Moss begins to provide a framework for describing the effects of knowledge producing actions within a multi-agent setting. While somewhat limited from the standpoint of knowledge representation, we believe that a higher level action language can be developed by adding *agent-role statements* to existing action languages and characterizing the new language by translating the higher level language of causal laws to their corresponding action model representations. Successor states could then be defined in terms of the update operation induced by these action models. Once this has been accomplished, one can extend the work done with respect to various reasoning about action tasks such as planning and diagnosis from single, to multi-agent domains.

## References

1. Baltag, A., Moss, L.: Logics for epistemic programs. Synthese (2004)
2. Baral, C., Gelfond, G., Son, T., Pontelli, E.: Using answer set programming to model multi-agent scenarios involving agents' knowledge about other's knowledge. In: AAMAS 2010 (2010)
3. Baral, C., Son, T., Pontelli, E.: Modeling multi-agent domains in an action language: An empirical study using C. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 409–415. Springer, Heidelberg (2009)
4. van Ditmarsch, H., van der Hoek, W., Kooi, B.: Dynamic Epistemic Logic. Springer, Heidelberg (2007)
5. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press, Cambridge (1995)
6. Gelfond, G.: A declarative framework for modeling multi-agent systems. Masters Thesis, Texas Tech. University (2007)
7. Gelfond, M., Lifschitz, V.: Representing actions in extended logic programs. In: Apt, K. (ed.) Joint International Conference and Symposium on Logic Programming, pp. 559–573. MIT Press, Cambridge (1992)
8. Ghaderi, H., Levesque, H., Lespérance, Y.: A logical theory of coordination and joint ability. In: AAAI 2007, pp. 421–426 (2007)

9. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming 31(1-3), 59–84 (1997)
10. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press, Edinburgh (1969)
11. Reiter, R.: Knowledge in action: logical foundation for describing and implementing dynamical systems. MIT Press, Cambridge (2001)
12. Sandewall, E.: The range of applicability of some non-monotonic logics for strict inertia. Journal of Logic and Computation 4(5), 581–616 (1994)
13. Shapiro, S., Lespérance, Y., Levesque, H.: The cognitive agents specification language and verification environment for multiagent systems. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (2002)

# Nonmonotonic Multi-Context Systems: A Flexible Approach for Integrating Heterogeneous Knowledge Sources*

Gerhard Brewka[1], Thomas Eiter[2], and Michael Fink[2]

[1] Universität Leipzig, Augustusplatz 10-11, 04109 Leipzig, Germany
brewka@informatik.uni-leipzig.de
[2] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink}@kr.tuwien.ac.at

**Abstract.** In this paper we give an overview on multi-context systems (MCS) with a special focus on their recent nonmonotonic extensions. MCS provide a flexible, principled account of integrating heterogeneous knowledge sources. By a knowledge source we mean a knowledge base formulated in any of the typical knowledge representation languages, including classical logic, description logics, modal or temporal logics, but also nonmonotonic formalisms like logic programs under answer set semantics or default logic. We will motivate the need for such systems, describe what has been achieved in this area, but we also discuss work in progress and introduce generalizations of the existing framework which we consider useful.

## 1 Introduction

In this paper we give an overview on multi-context systems (MCS) with a special focus on their recent nonmonotonic extensions. MCS provide a flexible, principled account of integrating heterogeneous knowledge sources. By a knowledge source we mean a knowledge base formulated in any of the typical knowledge representation languages, including classical logic, description logics, modal or temporal logics, but also non-monotonic formalisms like logic programs under answer set semantics or default logic.

There are several reasons why the integration of multiple knowledge representation formalisms, logics in our terminology, is highly important. First of all, larger and larger bodies of knowledge are being formalized, by different groups of researchers and with different intended uses. It is a matter of fact that these efforts, which have been particularly fueled by the vision of the semantic web, are based on a variety of different representation languages. Re-representing all this knowledge in a single "standard" formalism is out of the question for practical reasons. Secondly, and may be even more importantly, a single universal knowledge representation language is not in sight, and

* This work has been supported by the Austrian Science Fund (FWF) projects P20840 & P20841, the EC ICT Integrated Project Ontorule (FP7 231875), and the Vienna Science and Technology Fund (WWTF) project ICT08-020. The first author acknowledges funding from Deutsche Forschungsgemeinschaft under grant Br 1817/3.

there are good reasons to doubt that such a language exists. The needs of different applications in terms of expressiveness and/or efficiency vary tremendously. For this reason we strongly believe that the plurality of formalisms we see today is necessary and will not disappear in the future.

If this analysis is correct, then the integration of formalisms is an issue that needs to be addressed urgently. MCS provide a principled approach to such an integration. A context here is a knowledge base represented in a logical formalism. A multi-context system describes the information available in a number of contexts (i.e., to a number of people/agents/databases/modules, etc.) and specifies the information flow between those contexts. The contexts themselves may be heterogeneous in the sense that they can use different logical languages and different inference systems, and no notion of global consistency is required. The information flow is modeled via so-called *bridge rules* which can refer in their premises to information from other contexts.

The history of MCS started in Trento in the Nineties of the last century. Furthering work in [35,42], the Trento School developed monotonic heterogeneous multi-context systems [36] with the aim to integrate different inference systems; informally, contexts were viewed as pairs $Context_i = (\{T_i\}, \Delta_{br})$ where each $T_i = (L_i, \Omega_i, \Delta_i)$ is a formal system consisting of a language $L_i$, a set of axioms $\Omega_i \subseteq L_i$, and a set of inference rules $\Delta_i$. $\Delta_{br}$ consists of *bridge rules* of the form

$$(c_1 : p_1), \ldots, (c_k : p_k) \Rightarrow (c_j : q_j)$$

using labeled formulas $(c : p)$ where $p$ is from the language $L_c$. Giunchiglia and Serafini gave a collection of such contexts a semantics in terms of local models plus compatibility conditions (see also [34]), which respects information flow across contexts via bridge rules. Noticeably, reasoning within/across contexts is monotonic.

The first, still somewhat limited attempts to include nonmonotonic reasoning are [43] and [11]. To allow for reasoning based on the *absence* of information from a context, in both papers default negation is allowed in the rules. In this way contextual and default reasoning are combined. The former paper is based on a model theoretic approach where so-called information chains are manipulated. The latter is based on a multi-context variant of default logic (respectively its specialization to logic programs under answer set semantics).

The MCS of [9] substantially generalized these approaches, by accommodating *heterogeneous* and both *monotonic* and *nonmonotonic* contexts. They are thus capable of integrating "typical" monotonic KR logics like description logics or temporal logics, and nonmonotonic logics like Reiter's Default Logic, Answer Set Programming, circumscription, defeasible logic, or theories in autoepistemic logic; in several of the latter, a knowledge base gives rise to multiple belief sets in general. Since we consider nonmonotonic formalisms as essential in knowledge representation we focus entirely on this type of MCS here.

It is far from incidental that this paper appears in a collection honoring Michael Gelfond's 65th birthday. To the contrary, there are many intimate connections to his work, and it is more than fair to say that without Michael's seminal contributions this research area would still be in its (monotonic) infancy. The following connections are the most obvious ones:

1. Although MCS are "pluralistic"[1] in admitting a variety of different knowledge representation languages, we foresee that logic programs under answer set semantics, as developed by Gelfond and Lifschitz [32,33,31], will play a highly prominent role as particular context languages in MCS.
2. Nonmonotonic rules and their semantics provide the formal tools for integrating knowledge in MCS and thus play a crucial role in our framework.
3. The construction used to define grounded equilibria (see Sect. 2.2) is a direct generalization of the Gelfond-Lifschitz reduct [32] to multiple contexts.
4. Mediators extend the basic MCS framework with revision and consistency handling methods. Michael Gelfond has made numerous contributions to these topics which we believe will prove highly fruitful. As an example let us just mention his work on consistency restoring rules [3].

The outline of the rest of the paper is as follows. We first present (heterogeneous nonmonotonic) MCS [9], recalling their basic definitions. We also discuss ways of implementing such systems. The next section presents techniques for handling inconsistencies in MCS. Such inconsistencies arise naturally whenever different knowledge bases, often developed by different researchers, are being integrated. Consequently, there is a need for adequate consistency restoring methods.

We then discuss argumentation context systems (ACS) [10]. These system are less general than MCS in one respect: they are homogeneous and use Dung-style argumentation frameworks [19] as their single context logics. However, they go beyond MCS in two important aspects: (a) they allow contexts not only to augment other contexts, but also to update (e.g. revise) other contexts, and (b) they introduce so-called mediators which resolve conflicts among the update information which is obtained from other contexts through (generalized) bridge rules.

After, we report about some ongoing and future work. Here, a natural question to ask is then whether both ACS and MCS can be generalized to a framework which combines the additional functionality of ACS with the wide coverage of heterogeneous formalisms of MCS. We will address this in Sect. 5.4, where we outline one possible way to realize such a framework. A brief discussion of related work concludes the paper.

## 2 Heterogeneous Nonmonotonic MCS

As mentioned in the Introduction, a multi-context system describes the information available in a number of contexts and specifies the information flow between those contexts via bridge rules. In this section we will first present the basic notions underlying MCS, leading to the equilibrium semantics [9]. We then discuss groundedness of equilibria, applying a generalization of the Gelfond-Lifschitz reduct to MCS to solve a potential problem for a certain subset of MCS. We finally discuss implementation issues.

### 2.1 Formal Concepts

A "logic" is, very abstractly, a tuple $L = (\mathbf{KB}_L, \mathbf{BS}_L, \mathbf{ACC}_L)$, where

[1] One might even say "promiscuous".

– $\mathbf{KB}_L$ is a set of well-formed knowledge bases, each being a set (of formulas),
– $\mathbf{BS}_L$ is a set of possible belief sets, each being a set (of formulas), and
– $\mathbf{ACC}_L : \mathbf{KB}_L \rightarrow 2^{\mathbf{BS}_L}$ assigns to each $kb \in \mathbf{KB}_L$ a set of acceptable belief sets.

$L$ is *monotonic*, if $\mathbf{ACC}_L$ assigns to each $kb$ a single belief set (denoted $S_{kb}$), and $kb \subseteq kb'$ implies $S_{kb} \subseteq S_{kb'}$. Examples of knowledge bases as logic programs, default theories, description logic ontologies, classical propositional or first order theories, etc. The possible belief sets are those which are syntactically admissible (e.g., deductively closed sets of sentences, sets of literals, etc). Moreover, $\mathbf{ACC}_L$ respects that a knowledge base might have one, multiple, or even no acceptable belief set in the logic.

Access to other contexts is facilitated via bridge rules for heterogenous logics. Given logics $L = L_1, \ldots, L_n$, an $L_i$-bridge rule over $L$, $1 \leq i \leq n$, is of the form

$$s \leftarrow (r_1 : p_1), \ldots, (r_j : p_j), \mathbf{not}\ (r_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (r_m : p_m) \qquad (1)$$

where $r_k \in \{1 \ldots, n\}$ and $p_k$ is an element of some belief set of $L_{r_k}$, $1 \leq k \leq m$, and $kb \cup \{s\} \in \mathbf{KB}_i$ for each $kb \in \mathbf{KB}_i$.

Multi-context systems are then defined as follows.

**Definition 1.** *A* multi-context system $M = (C_1, \ldots, C_n)$ *consists of contexts* $C_i = (L_i, kb_i, br_i)$*, where* $L_i = (\boldsymbol{KB}_i, \boldsymbol{BS}_i, \boldsymbol{ACC}_i)$ *is a logic,* $kb_i \in \boldsymbol{KB}_i$ *is a knowledge base, and* $br_i$ *is a set of* $L_i$*-bridge rules over* $L = L_1, \ldots, L_n$*,* $1 \leq i \leq n$*.*

*Example 1.* As a simple example, we consider $M = (C_1, C_2)$, where the contexts are different views of a paper by its co-authors $A_1$ and $A_2$ who reason in different logics. In $C_1$, we have Classical Logic as $L_1$, the knowledge base $kb_1 = \{\, \mathit{unhappy} \supset \mathit{revision} \,\}$, and the bridge rules $br_1 = \{\, \mathit{unhappy} \leftarrow (2 : \mathit{work}) \,\}$. Intuitively, if $A_1$ is unhappy about the paper, then she wants a revision, and if $A_2$ finds that the paper needs more work, then $A_1$ feels unhappy. In $C_2$, we have Answer Set Programming as $L_2$, the knowledge base $kb_2 = \{\, \mathit{accepted} \leftarrow \mathit{good}, \mathrm{not}\ \neg \mathit{accepted} \,\}$ and bridge rules $br_2 = \{\, \mathit{work} \leftarrow (1 : \mathit{revision});\ \mathit{good} \leftarrow \mathbf{not}\ (1 : \mathit{unhappy}) \}$. Intuitively, $A_2$ thinks that the paper, if good, is usually accepted; moreover, she infers that more work is needed if $A_1$ wants a revision, and that the paper is good if there is no evidence that $A_1$ is unhappy.

The semantics of an MCS is defined in terms of special belief states, which are sequences $S = (S_1, \ldots, S_n)$ such that each $S_i$ is an element of $\mathbf{BS}_i$. Intuitively, $S_i$ should be a belief set of the knowledge base $kb_i$; however, also the bridge rules must be respected; to this end, $kb_i$ is augmented with the conclusions of its bridge rules that are applicable. More precisely, a bridge rule $r$ of form (1) is *applicable in* $S$, if $p_i \in S_{r_i}$, for $1 \leq i \leq j$, and $p_k \notin S_{r_k}$, for $j + 1 \leq k \leq m$. Denote by $\mathit{head}(r)$ the head of $r$ and by $\mathit{app}(R, S)$ the set of bridge rules $r \in R$ that are applicable in $S$. Then,

**Definition 2.** *A belief state* $S = (S_1, \ldots, S_n)$ *of a multi-context system* $M$ *is an* equilibrium *iff* $S_i \in \boldsymbol{ACC}_i(kb_i \cup \{\mathit{head}(r) \mid r \in \mathit{app}(br_i, S)\})$*,* $1 \leq i \leq n$*.*

An equilibrium thus is a belief state which contains for each context an acceptable belief set, given the belief sets of the other contexts.

*Example 2 (ctd).* Reconsidering $M = (C_1, C_2)$ from Example 1, we find that $M$ has two equilibria, viz.

- $E_1 = (Cn(\{unhappy, revision\}), \{work\})$ and
- $E_2 = (Cn(\{unhappy \supset revision\}), \{good, accepted\})$,

where $Cn(\cdot)$ is the set of all classical consequences. As for $E_1$, the bridge rule of $C_1$ is applicable in $E_1$, and $Cn(\{unhappy, revision\})$ is the (single) acceptable belief set of $kb_i \cup \{unhappy\}$; the first bridge rule of $C_2$ is applicable in $E_1$, but not the second; clearly, $\{work\}$ is the single answer set of $kb_2 \cup \{\, work \,\}$.

As for $E_2$, the bridge rule of $C_1$ is not applicable in $E_1$, and $Cn(\{unhappy \supset revision\} = Cn(kb_1)$; now the second bridge rule of $C_2$ is applicable but not the first, and $\{good, accepted\}$ is the single answer set of $kb_2 \cup \{good\}$.

The notion of equilibrium may remind the reader of similar game-theoretic concepts, and in fact we may view each context $C_i$ as a player in an $n$-person game where players choose belief sets; we will discuss this more in detail in Section 6.

## 2.2 Groundedness

Equilibria suffer, similar as the answer sets of modular logic programs in [24], from groundedness problems due to cyclic justifications. Informally, the reason is that bridge rules might be applied unfoundedly. E.g., in Example 1, the formula $unhappy$ has only a cyclic justification in the equilibrium

$$E_1 = (Cn(\{unhappy, revision\}), Cn(\{work\})).$$

The formula is accepted in $C_1$ via the bridge rule, as $work$ is accepted in $C_2$; the latter is also accepted via a bridge rule, as $revision$ is accepted in $C_1$ (by modus ponens from $unhappy \supset revision$ and $unhappy$). Here, the application of the bridge rules is unfounded.

Inspired by the definition of answer set semantics, [9] proposed grounded equilibria to overcome this. They are defined in terms of a GL-style reduct which transforms $M = (C_1, \ldots, C_n)$, given a belief state $S = (S_1, \ldots, S_n)$, into another MCS $M^S = (C_1^S, \ldots, C_n^S)$ that behaves monotonically, such that a unique minimal equilibrium exists; if it coincides with $S$, we have groundedness.

Formally, $C_i^S = (L_i, red_i(kb_i, S), br_i^S)$, where $red_i(kb_i, S)$ maps $kb_i$ and $S$ to a monotonic core of $L_i$ and $br_i^S$ is the GL-reduct of $br_i$ w.r.t. $S$, i.e., contains $s \leftarrow (r_1 : p_1), \ldots, (r_j : p_j)$ for each rule of form (1) in $br_i$ such that $p_k \notin S_{r_k}$, $k = j+1, \ldots, m$.

In addition, the following *reducibility conditions* are assumed: (i) $red_i(kb_i, S_i)$ is antimonotonic in $S_i$, (ii) $S_i$ is acceptable for $kb_i$ iff $\mathbf{ACC}_i(red_i(kb_i, S_i)) = \{S_i\}$, and (iii) $red_i(kb_i, S) \cup H_i = red_i(kb_i \cup H, S)$, for each $H_i \subseteq \{head(r) \mid r \in br_i\}$. This condition is trivially satisfied by all monotonic logics, by Reiter's Default Logic, answer set programs, etc. Grounded equilibria are then defined as follows.

**Definition 3.** *A belief state $S = (S_1, \ldots, S_n)$ is a grounded equilibrium of $M$ iff $S$ is the unique minimal equilibrium of $M^S$, where minimality is componentwise w.r.t. $\subseteq$.*

*Example 3 (ctd.).* In our review example, naturally $red(kb_i, S)$ is identity and $red(kb_2, S)$ the GL-reduct. Then $E_1$ is not a grounded equilibrium: $M^{E_1}$ has the single minimal equilibrium $(Cn(\{unhappy \supset revision\}), \emptyset)) \neq E_1$. On the other hand, $E_2$ is a grounded equilibrium of $M$.

Grounded equilibria are in fact equilibria of $M$, as well as minimal ones. Similar as for answer sets, the grounded equilibrium of $M^S$ can be characterized as the least fixpoint of an operator [9].

Brewka and Eiter also introduced a wellfounded semantics for MCS. The reader is referred to [9] for the details.

### 2.3 Implementing MCS

As for implementing MCS, we assume that the belief sets under consideration are uniquely identified by finite sets of beliefs, which serve as kernels (cf. [9]) that represent (potentially infinite) belief sets. Suppose also that the alphabets $\Sigma_i$, $1 \leq i \leq n$ for these beliefs are pairwise disjoint for contexts (which can always be achieved, e.g., by prefixing). Moreover, let us consider two different scenarios: taking a *centralized stance*, one is interested in computing equilibria as global views on the system, while taking a *decentralized stance* the goal is to compute partial global views (*partial equilibria*) by a distributed computation which is initiated at a particular context. In the following, we give a brief account of the main ideas underlying implementations that have been developed for both application scenarios.

*Centralized MCS implementation.* The computation of equilibria for a given MCS has been realized by a declarative implementation using HEX-programs [26] which can be evaluated using the dlvhex system.[2] HEX-programs extend disjunctive logic programs with access to external information by means of so-called *external atoms* (and with higher order features, which we disregard here).

Focusing on ground (variable-free) HEX-programs, we say that an *ordinary atom* is a predicate $p(c_1, \ldots, c_n)$ where $p$, and $c_1, \ldots, c_n$ are constants. An *external atom* is of the form

$$\&g[\boldsymbol{v}](\boldsymbol{w}),$$

where $\boldsymbol{v}$, and $\boldsymbol{w}$ are fixed length lists of constants, and $\&g \in \mathcal{G}$ is an external predicate name. Intuitively, an external atom provides a way for deciding the truth value of $g(\boldsymbol{w})$ in an external source which is accessed providing the extension of predicates $\boldsymbol{v}$ as input.

A HEX *rule* $r$ is of the form

$$\alpha_1 \vee \ldots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \text{not}\, \beta_{m+1}, \ldots, \text{not}\, \beta_n \tag{2}$$

$m, k \geq 0$, where all $\alpha_i$ are ordinary atoms and all $\beta_j$ are ordinary or external atoms. As usual, a rule $r$ is a constraint, if $k = 0$. Furthermore, a HEX-*program* (or *program*) is a finite set of HEX rules.

The semantics of HEX-programs is defined considering interpretations $I$ over the ordinary *Herbrand base* $HB_P$ of a program $P$ and a set of constants $\mathcal{C}$. An interpretation

[2] http://www.kr.tuwien.ac.at/research/systems/dlvhex/

$I$ satisfies an external atom $\alpha = \&g[\boldsymbol{v}](\boldsymbol{w})$ (denoted $I \models \alpha$), if $f_{\&g}(I, \boldsymbol{v}, \boldsymbol{w}) = 1$, where $\boldsymbol{v} \in \mathcal{C}^n$, $\boldsymbol{w} \in \mathcal{C}^m$, and $f_{\&g}$ is a (fixed) function $f_{\&g} : 2^{HB_P} \times \mathcal{C}^{n+m} \rightarrow \{0, 1\}$, representing the (semantics of the) corresponding external source. For an ordinary atom $\alpha$, a rule $r$, or a program $P$, the satisfaction relation $I \models \alpha$ (respectively $I \models r$ or $I \models P$) is defined as usual.

The *FLP-reduct* [27] of $P$ wrt. $I$ is the set $fP^I \subseteq P$ of all rules $r$ of form (2) in $P$ such that $I \models \beta_i$, for all $i \in \{1, \ldots, m\}$ and $I \not\models \beta_j$ for all $j \in \{m+1, \ldots, n\}$. Eventually, $I$ is an *answer set* of $P$, if $I$ is a $\subseteq$-minimal model of $fP^I$.

Note that this semantics is another generalization of the Gelfond-Lifschitz reduct [32], i.e., on programs without external atoms the semantics coincide. For a more detailed account of HEX and its relation to MCS see [20].

Concerning our main purpose, given an MCS $M$, we assemble a program $P(M)$ for computing equilibria of $M$ as follows, where $1 \leq i \leq n$. An arbitrary truth assignment to beliefs is guessed:

$$a_i(p) \vee \bar{a}_i(p). \qquad \forall p \in \Sigma_i \tag{3}$$

Each bridge rule (1) is evaluated by corresponding HEX rules, wrt. the guess:

$$\begin{aligned} b_i(s) \leftarrow\ & a_{c_1}(p_1), \ldots, a_{c_j}(p_j), \\ & \text{not } a_{c_{j+1}}(p_{j+1}), \ldots, \text{not } a_{c_m}(p_m). \end{aligned} \tag{4}$$

Finally, constraints ensure that answer sets of the program correspond to equilibria:

$$\leftarrow \text{not } \&\mathit{con\_out}_i[a_i, b_i](). \tag{5}$$

Given an interpretation $I$, let $A_i^I = \{p \mid a_i(p) \in I\}$, $1 \leq i \leq n$, denote a belief set for context $C_i$ (corresponding to the guess on $\Sigma_i$ in (3)), and let $B_i^I = \{s \mid b_i(s) \in I\}$ denote the set of bridge rule heads, from bridge rules $br_i$, which are applicable wrt. the guessed belief state. Each external atom in (5) represents $\mathbf{ACC}_i$: it returns true iff context $C_i$ accepts a belief set upon input of $B_i^I$, which corresponds to the guessed $A_i^I$. Formally, we define $f_{\&con\_out_i}(I, a_i, b_i) = 1$ iff there exists an $S \in \mathbf{ACC}_i(kb_i \cup B_i^I)$ such that $S = A_i^I$.

**Proposition 1.** *Answer sets $I$ of $P(M)$ correspond 1-1 to equilibria $S^I$ of $M$, where $I \leftrightharpoons S^I = (S_1^I, \ldots, S_n^I)$ and $S_i^I = \{p \mid a_i(p) \in I\}$, $i = 1, \ldots, n$.*

For further details on a concrete implementation we refer the reader to the MCS-IE system [5].

*Decentralized MCS implementation.* For the remainder of this section, we make our assumption on the representation of belief sets slightly more precise, supposing that they are represented by truth assignments $v_{S_i} : \Sigma_i \rightarrow \{0, 1\}$ to a finite set $\Sigma_i$ of propositional atoms. Furthermore, let $\Sigma = \bigcup_i \Sigma_i$.

In a decentralized setting, given an MCS $M$ and a starting context $C_k$, we aim at finding so-called *partial equilibria* of $M$ wrt. $C_k$ in a distributed way. For this purpose, an algorithm called DMCS has been developed, whose instances run independently at each context node and communicate with other instances for exchanging sets of partial belief states. It provides a method for distributed model building, and the DMCS algorithm can be applied to any MCS such that appropriate solvers for the respective context logics are available. As a main feature, DMCS can also compute *projected partial equilibria*, i.e., partial equilibria projected to a relevant portion of the signature of the so-called import closure of the starting context. This can be exploited for specific tasks like, e.g., local query answering or consistency checking.

The notion of import closure formally captures contexts that are 'reachable' from a given context according to the MCS topology given by its bridge rules. It essentially defines a subsystem $M'$ of $M$ that is connected by bridge rules. Let $In(k) = \{c_i \mid (c_i : p_i) \in B(r), r \in br_k\}$, for a given context $C_k$ of an MCS $M$. Then, the *import closure* $IC(k)$ *of* $C_k$ is the smallest set $S$ such that (i) $k \in S$ and (ii) $In(i) \subseteq S$, for all $i \in S$. The import closure provides a syntactic notion of 'relevance' to consider partial equilibria which are local to $M'$. (Note however, that the MCS semantics does not satisfy a relevance property akin to the one in [17] in general.) Based on the import closure partial equilibria are defined as follows.

**Definition 4 (Partial Belief States and Equilibria)**
*Let* $M = (C_1, \ldots, C_n)$ *be an MCS, and let* $\epsilon \notin \bigcup_{i=1}^{n} \boldsymbol{BS}_i$. *A* partial belief state *of* $M$ *is a sequence* $S = (S_1, \ldots, S_n)$, *such that* $S_i \in \boldsymbol{BS}_i \cup \{\epsilon\}$, *for* $1 \leq i \leq n$.

*A partial belief state* $S = (S_1, \ldots, S_n)$ *of* $M$ *is a* partial equilibrium of $M$ w.r.t. a context $C_k$ *iff* $i \in IC(k)$ *implies* $S_i \in \boldsymbol{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$, *and if* $i \notin IC(k)$, *then* $S_i = \epsilon$, *for all* $1 \leq i \leq n$.

For combining partial belief states $S = (S_1, \ldots, S_n)$ and $T = (T_1, \ldots, T_n)$, their *join* $S \bowtie T$ is defined as the partial belief state $(U_1, \ldots, U_n)$ with (i) $U_i = S_i$, if $T_i = \epsilon$ or $S_i = T_i$, and (ii) $U_i = T_i$, if $T_i \neq \epsilon$ and $S_i = \epsilon$, for all $1 \leq i \leq n$. This notion is extended to sets $\mathcal{S}$ and $\mathcal{T}$ of partial belief states in the obvious way: $\mathcal{S} \bowtie \mathcal{T} = \{S \bowtie T \mid S \in \mathcal{S}, T \in \mathcal{T}\}$.

In the sequel, we present the main idea of a basic version of the DMCS algorithm, which assumes that the topology of the overall MCS is not known at context nodes (enhancements are possible given topology information): starting from context $C_k$, the import closure of $C_k$ is visited by expanding $In(k)$ at each context like in a depth-first search, maintaining visited contexts in a set $hist$, until a leaf context is reached, or a cycle is detected (by noticing the presence of the current context in $hist$). A leaf context simply computes its local belief sets, transforms them into partial belief states, and returns this result to its parent (invoking context). In case of a cycle, the context detecting the cycle, say $C_i$, must also break it, by

1. guessing belief sets for the "export" interface of $C_i$,
2. transforming the guesses into partial belief states, and
3. returning them to the invoking context.

The results of intermediate contexts are partial belief states, which can be joined, i.e., consistently combined, with partial belief states from their neighbors; a context $C_k$ returns its local belief sets, joined with the results from its neighbors, as final result.

For computing projected partial equilibria, the algorithm offers a parameter $V$, the *relevant interface*. Given a (partial) belief state $S$ and a set $V \subseteq \Sigma$ of variables, the *restriction of* $S$ *to* $V$, denoted $S|_V$, is the (partial) belief state $S' = (S_1|_V, \dots, S_n|_V)$, where $S_i|_V = S_i \cap V$ if $S_i \neq \epsilon$, and $\epsilon|_V = \epsilon$; the restriction of a set of (partial) belief states $\mathcal{S}$ to $V$ is $\mathcal{S}|_V = \{S|_V \mid S \in \mathcal{S}\}$. Let $V(k) = \{p_i \mid (c_i : p_i) \in B(r), r \in br_k\}$ denote the *import interface* of context $C_k$. By $V^*(k) = \bigcup_{i \in IC(k)} V(i)$, the *recursive import interface* of $C_k$, we refer to the interface of the import closure of $C_k$.

For a context $C_k$, we have two extremal cases: 1. $V = V^*(k)$ and 2. $V = \Sigma$. In Case 1, DMCS basically checks for consistency on the import closure of $C_k$ by computing partial equilibria projected to interface variables only. In Case 2, the algorithm computes partial equilibria w.r.t. $C_k$. Between these two, by providing a fixed interface $V$, problem-specific knowledge (such as query variables) and/or infrastructure information can be exploited to keep computations focused on relevant projections of partial belief states. For further details and a concrete implementation see [15,1] and the DMCS system [2].

## 3 Inconsistency Handling in MCS

*Inconsistency* in an MCS is the lack of an equilibrium. As the combination and interaction of heterogeneous systems can easily have unforeseen and intricate effects, inconsistency is a major problem in MCS. The eventual aim of understanding and giving reasons for inconsistency is to provide support for restoring consistency. We assume that every context is consistent if no bridge rules apply, and thus, reasons of inconsistency can fully be captures in terms of bridge rules.

*Example 4.* Consider an extension of Example 1 given by MCS $M' = (C_1, C_2', C_3)$, where the third context represents the view of a mentor of co-author $A_2$. For simplicity, let $C_3$ also have Answer Set Programming as $L_3$, and assume that $kb_3 = \{\, comments \leftarrow done \,\}$ and that $br_3 = \{\, r_3 \,\}$, where $r_3 = done \leftarrow \textbf{not}\,(2\!:\!work)$. Intuitively, the mentor provides comments as soon as author $A_2$ considers the paper finished, i.e., she does not believe that the paper needs more work. The author $A_2$ takes this information into account via the extended set of bridge rules $br_2' = br_2 \cup \{\, r_1, r_2 \,\}$, where $r_1 = advice \leftarrow (3\!:\!comments)$, and $r_2 = ack \leftarrow \textbf{not}\,(3\!:\!comments)$. Thus, either there is advice (comments) from the mentor, or the mentor acknowledges the current status of the paper (no comments). Moreover, the following knowledge is inconsistent with $A_2$'s views: she believes that the paper needs more work but her mentor acknowledges it; she does not believe that the paper needs more work but there is further advice. This is reflected in the following extension of her knowledge base: $kb_2' = kb_2 \cup \{\, \leftarrow work, ack;\ \leftarrow advice, \text{not}\ work \,\}$.

This MCS turns out to be inconsistent. Indeed, one can easily verify that whatever the views of $A_1$ and the mentor, there is no consistent belief set for $A_2$ which is in equilibrium (i.e., compliant with applicable bridge rules).

We will use the following notation. Given an MCS $M$ and a set $R$ of bridge rules (compatible with $M$), by $M[R]$ we denote the MCS obtained from $M$ by replacing its set of bridge rules $br_M$ with $R$ (e.g., $M[br_M] = M$ and $M[\emptyset]$ is $M$ with no bridge rules). By $M \models \bot$ we denote that $M$ has no equilibrium, i.e., is inconsistent, 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.

## 3.1 Diagnoses and Explanations

In the following, we focus on two main possibilities for explaining inconsistency in MCSs considered in [22]: first, a consistency-based notion called *diagnosis*, which identifies a part of the bridge rules which need to be changed to restore consistency. Second, an entailment-based notion termed *inconsistency explanation*, which identifies a part of the bridge rules that is necessary to make the MCS inconsistent.

*Diagnoses.* Adding knowledge, as well as removing knowledge, can both cause and prevent inconsistency. Therfore, we consider pairs of sets of bridge rules, such that if we deactivate the rules in the first set, and add the rules in the second set in unconditional form, the MCS becomes consistent (i.e., admits an equilibrium).

**Definition 5.** *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.*

A more relevant set of diagnoses is obtained by preferring pointwise subset-minimal diagnoses. 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 $A_2 \subseteq B_2$. The set of all pointwise subset-minimal diagnoses of an MCS $M$ is denoted by $D^{\pm}_m(M)$.

*Example 5 (ctd.).* In our extended running example, the MCS $M'$ (cf. Example 4) which is inconsistent, $D^{\pm}_m(M) = \{(\{r_1\}, \emptyset), (\{r_2\}, \emptyset), (\{r_3\}, \emptyset), (\emptyset, \{r_3\})\}$. Accordingly, deactivating any of the 'new' (wrt. $M$ in Example 1) bridge rules $r_1$, $r_2$, or $r_3$, will result in a consistent MCS; likewise if $r_3$ is added in unconditional form.

Note that one could generalize Definition 5 to more fine-grained changes of rules, such that only some body atoms are removed instead of all. However, while this significantly increases the search space for diagnoses, there is little information gain: every diagnosis $(D_1, D_2)$ as above, together with a witnessing equilibrium $S$, can be refined to a generalized diagnosis $(D_1, D_2')$, where $D_2' \subseteq \{\alpha \leftarrow \beta \mid \alpha \leftarrow \beta, \gamma \in br_M\}$ contains for each $\alpha \leftarrow \beta, \gamma$ in $D_2$ some $\alpha \leftarrow \beta$ that is applicable in $S$. Conversely, every generalized diagnosis $(D_1, D_2')$, together with a witnessing equilibrium $S$, induces a diagnosis $(D_1, D_2)$ as above ($D_2$ contains all heads of rules in $D_2'$ that are applicable in $S$).

*Explanations.* In the spirit of abductive reasoning, an *inconsistency explanation* (subsequently also called *explanation*) is a pair of sets of bridge rules, such that their presence, respectively absence, necessarily entails an inconsistency in the given MCS.

**Definition 6.** *Given an MCS $M$, an* inconsistency explanation *of $M$ is a pair $(E_1, E_2)$ of sets $E_1, E_2 \subseteq br_M$ of bridge rules 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 $M[R_1 \cup \mathit{heads}(R_2)] \models \bot$. 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.*

The intuition about inconsistency explanations is as follows: bridge rules in $E_1$ cause an inconsistency in $M$ ($M[E_1] \models \bot$), and this inconsistency is relevant for $M$. By this we mean that adding some bridge rules from $br_M$ (the set of original bridge rules) to $M[E_1]$ never yields a consistent system. Note that a set of bridge rules may also create an inconsistency which is irrelevant for $M$ (if it does not occur given that more bridge rules are present). Similarly, bridge rules in $E_2$ govern the addition of bridge rules in unconditional form: $M[E_1]$ cannot be made consistent by adding bridge rule heads unconditionally, unless at least one bridge rule head from $E_2$ is used. In summary, bridge rules $E_1$ create a relevant inconsistency, and at least one head of a bridge rule in $E_2$ must be added unconditionally to repair that inconsistency.

*Example 6 (ctd.).* In our running example, we have just one minimal inconsistency explanation, namely $(\{r_1, r_2, r_3\}, \{r_3\})$. Thus, all three additional bridge rules *must* be present in the system to get inconsistency with author $A_2$'s views; if they are present, then it can only be resolved by telling the mentor unconditionally that they are done.

Again it is possible to consider more fine-grained modifications of rules (rather than $heads(R_2)$) in Definition 6, but this would not alter the notion of inconsistency explanation. Thus, in contrast to diagnoses, one cannot infer from an explanation whether the addition of a more fine-grained version of a rule in $E_2$ would yield consistency. However, this could be achieved considering explanations of a transformed MCS.

*Properties.* Intuitively, the two components of diagnoses and explanations represent dual aspects wrt. causation and prevention of inconsistency. For minimal diagnoses and explanations the definitions are closely related. Given a set $X$ of pairs $(A, B)$ of sets $A$ and $B$, let $\bigcup X$ denote the pair $(\bigcup\{A \mid (A, B) \in X\}, \bigcup\{B \mid (A, B) \in X\})$.

**Theorem 1.** *Given an inconsistent MCS $M$, $\bigcup D^{\pm}_{m}(M) = \bigcup E^{\pm}_{m}(M)$, i.e., the unions of all minimal diagnoses and all minimal inconsistency explanations coincide.*

This theorem strengthens the view that both notions capture exactly those parts of an MCS that are relevant for inconsistency: two different perspectives on inconsistency reveal exactly the same parts of an MCS as erroneous. In practice this allows to compute the set of all bridge rules which are relevant for restoring consistency (i.e., that appear in at least one diagnosis) in two ways, either by computing all minimal explanations, or by computing all minimal diagnoses. Moreover, this result also holds if the unconditional addition of bridge rule heads is not admitted.

Another important property concerns modularization. A syntactic criterion which allows to break up the computation of explanations for an MCS into computing explanations for parts of it is obtained by adapting the notion of *splitting sets* as introduced in the context of logic programming [41]. This can be exploited for computing minimal explanations more efficiently for certain classes of MCSs. Since an MCS may include contexts with arbitrary logics, a purely syntactical criterion can only be obtained resorting to beliefs occurring in bridge rules, hence splitting at the level of contexts.

Let $c(M)$ denote the set of contexts of an MCS $M$, and for a bridge rule $r$, let $h_c(r)$ be the context in its head and $b_c(r)$ the set of contexts referenced in its body. A set of contexts $U \subseteq c(M)$ is a *splitting set* of an MCS $M$, if $h_c(r) \in U$ implies $b_c(r) \subseteq U$, for every rule $r \in br_M$. The set $b_U \subseteq br_M$ of rules such that $r \in b_U$ iff $h_c(r) \in U$, is called the *bottom* relative to $U$.

Intuitively, if $U$ is a splitting set of $M$, then the consistency or inconsistency of contexts in $U$ does not depend on contexts in $c(M) \setminus U$. Thus if $M[b_U]$ is inconsistent, $M$ stays inconsistent.

**Proposition 2.** *Let $U$ be a splitting set of an MCS $M$. Then each (minimal) explanation of $M[b_U]$ is a (minimal) explanation of $M$, and each (minimal) diagnosis of $M[b_U]$ is a pointwise subset of a (minimal) diagnosis of $M$.*

Thus, if both $U$ and $U' = c(M) \setminus U$ are splitting sets of an MCS $M$, then $M$ can be partitioned into two parts where minimal explanations can be computed independently.

*Computation.* Regarding computation, a complexity analysis of recognizing diagnoses respectively explanations was provided in [22]. As diagnosis recognition has the same computational complexity as equilibrium existence, diagnoses can be computed using HEX-programs with external atoms (for oracle calls) in a similar way as outlined for the computation of equilibria in Section 2.3.

Note however, that for the purpose of recognizing diagnoses (and explanations), it suffices to check for consistency, i.e., for the existence of some equilibrium in a (modified) MCS. This consistency check can be done by limiting the equilibrium calculation to *output beliefs*, i.e., the beliefs used in bodies of bridge rules (rather than considering all beliefs in $\Sigma$).

Consider the program $P_D(M)$ for calculating diagnoses which is obtained from program $P(M)$ (cf. Section 2.3) restricted to output beliefs as follows: for each bridge rule $r \in br_M$, a guess is added:

$$norm(r) \vee d_1(r) \vee d_2(r). \tag{6}$$

Moreover, rules (4) are replaced by two corresponding HEX rules:

$$\begin{aligned} b_i(s) &\leftarrow \text{not } d_1(r), a_{c_1}(p_1), \ldots, a_{c_j}(p_j), \\ &\quad\;\; \text{not } a_{c_{j+1}}(p_{j+1}), \ldots, \text{not } a_{c_m}(p_m). & (7) \\ b_i(s) &\leftarrow d_2(r). & (8) \end{aligned}$$

Then, $P_D(M)$ can be used to compute diagnoses according to the following correspondence result.

**Proposition 3.** *For each diagnosis $(D_1, D_2) \in D^{\pm}(M)$ where $D_1 \cap D_2 = \emptyset$, at least one corresponding answer set of $P_p^D(M)$ exists. Each answer set $I$ of $P_p^D(M)$ corresponds to a unique diagnosis $D_I$, where $I \leftrightharpoons D_I = (\{r \mid d_1(r) \in I\}, \{r \mid d_2(r) \in I\})$.*

This approach has been followed in the implementation of the MCS-IE System for Explaining Inconsistency in Multi-Context Systems, cf. [5] and references therein.

### 3.2 Assessing Inconsistency

Large sets of diagnoses or explanations call for a further assessment, taking application specific criteria into account. For realizing such selection criteria on consistency restorations, preference-based approaches are a natural choice since they allow for a declarative specification. Two basic elements of preference-based selection can be found in the literature: filters, which discard non-preferred solutions that fail some preference condition, and qualitative comparison relations establishing preference orders to single out the most appealing solutions.

Selection-based preference on diagnoses, i.e. filters, allow a designer of an MCSs to apply sanity checks on diagnoses, thus they can be seen as hard constraints on them: diagnoses that fail to satisfy the conditions are filtered out and not considered for consistency restoration.

**Definition 7.** *Let $M$ be an MCS with bridge rules $br_M$. A* diagnosis filter *for $M$ is a function $f{:}2^{br_M} \times 2^{br_M} \to \{0,1\}$ and the set of* filtered diagnoses *is $D^{\pm}_{f}(M) = \{D \in D^{\pm}(M) \mid f(D) = 1\}$.*

Comparison-based preference on diagnoses provides a means to compare (minimal) diagnoses. In general, a *preference order* over diagnoses for an MCS $M$ is a transitive binary relation $\preceq$ on $2^{br}_{M} \times 2^{br}_{M}$; we say that $D$ is preferred to $D'$ iff $D \preceq D'$.

**Definition 8.** *Let $M$ be an inconsistent MCS. A diagnosis $D \in D^{\pm}(M)$ of $M$ is called pre-most preferred iff for all $D' \in 2^{br_M} \times 2^{br_M}$ with $D' \preceq D \wedge D \not\preceq D'$ it holds that $D' \notin D^{\pm}(M)$. A diagnosis $D \in D^{\pm}(M)$ is called* most preferred*, iff $D$ is subset-minimal among all pre-most preferred diagnoses.*

Given that MCSs are decentralized systems, users may want to express preferences on diagnoses solely based on a local set of bridge rules, assuming all other things equal. Such preferences can be formalized using conditional preference networks (CP-nets) [8], which are an extension of *ceteris paribus* orders ("all else being equal") and exhibit appealing features of locality and privacy.

For a further details on associating a CP-net with an MCS for inconsistency assessment, as well as on the realization of filters and comparison-based selection on diagnoses inside the MCS framework by using meta-reasoning on consistency restorations, i.e., utilizing a rewriting which yields a transformed system such that consistency restorations of the latter directly correspond to preferred consistency restorations of the original system (wrt. the given filter, preference order, or CP-net), the interested reader may consult [23].

## 4 Argumentation Context Systems

Argumentation context systems (ACS) [10] specialize multi-context systems in one respect, and are more general in another. First of all, in contrast to the MCS of [9], they are homogeneous in the sense that all reasoning components in an ACS are of the same type, namely Dung-style argumentation frameworks [19]. The latter are widely used as abstract models of argumentation. They are based on graphs whose nodes represent abstract arguments and edges describe attacks between arguments. Different semantics for argumentation frameworks have been defined; they specify *extensions*, i.e., subsets of the arguments which are considered jointly acceptable.

However, ACS go beyond MCS in two important aspects:

1. The influence of an ACS module $M_1$ on another module $M_2$ can be much stronger than in an MCS. $M_1$ may not only provide information for $M_2$ and thus augment the latter, it may directly affect $M_2$'s KB and reasoning mode: $M_1$ may invalidate arguments or attack relationships in $M_2$'s argumentation framework, and even determine the semantics to be used by $M_2$.
2. A major focus in ACS is on *inconsistency handling*. Modules are equipped with additional components called *mediators*. The main role of the mediator is to take care of inconsistencies in the information provided by connected modules. It collects the information coming in from connected modules and turns it into a consistent update specification for its module, using a pre-specified consistency handling method which may be based on preference information about other modules.

The first property listed above makes it possible to capture – in addition to peer-to-peer types of information exchange – hierarchical forms of argumentation as they are common in legal reasoning. Here a judge may declare certain arguments as invalid, for instance because they are based on evidence which was obtained illegally, or the type of trial may require a particular proof standard.

Technically, the additional functionality is achieved by explicitly representing possible updates to be performed on the underlying argumentation framework in a genuine description language. This language contains expressions which allow us to introduce - respectively delete - arguments, attack relations, preferences among arguments, values and value orderings, and also to fix a reasoning mode (sceptical vs. credulous) and an argumentation semantics (stable, preferred, grounded). Expressions of this language rather than formulas are then used as heads of the bridge rules. For more details on the language the reader is referred to [10].

Consistency handling is achieved by the introduction of mediators. Mediators are components which, intuitively, collect the update information for the underlying argumentation framework from connected modules and resolve potential conflicts within this information. As in MCS, the collection of the relevant information from other modules is done via bridge rules.

For the conflict resolution, mediators are equipped with a specific consistency handling method. The choice of different such methods allows a broad range of scenarios to be modeled, from strictly hierarchical ones (a judge decides) to more "democratic" forms of decision making (based on voting). In each case, the mediator takes the heads of its applicable bridge rules as input and returns one or several consistent sets of update statements, called (update) contexts. Each context then specifies how the argumentation framework is to be modified, and which semantics is to be applied.

Let $\mathcal{A}$ be an argumentation framework, $C$ an update context. We call a subset $E$ of $\mathcal{A}$'s arguments acceptable under $C$, if $E$ is an extension of the argumentation framework obtained from $\mathcal{A}$ by applying modifications specified by $C$ under the semantics fixed by $C$. We are now in a position to introduce the ACS framework more formally.

**Definition 9.** *An* (argumentation) module $\mathcal{M} = (\mathcal{A}, Med)$ *consists of an argument framework* $\mathcal{A}$ *and a mediator* $Med$ *for* $\mathcal{A}$ *based on some argumentation frameworks* $\mathcal{A}_1, \ldots, \mathcal{A}_k$[3].

**Definition 10.** *An* argumentation context system ($\mathcal{ACS}$) *is a sequence*

$$\mathcal{F} = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$$

*of modules* $\mathcal{M}_i = (\mathcal{A}_i, Med_i)$, $1 \leq i \leq n$, *such that each mediator* $Med_i$ *is based only on argumentation frameworks* $\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_k}$, *where* $i_j \in \{1, \ldots, n\}$ (self-containedness).

We next define the acceptable states for the framework, which correspond to the equilibria of MCS. Intuitively, such a state consists of an update context and a set of arguments for all modules $\mathcal{M}_i$. Again states have to satisfy a certain equilibrium condition: for each module, the chosen arguments must be an acceptable set of arguments for $\mathcal{A}_i$ under the respective update context, and this context (determined by $Med_i$) must be one of the contexts produced by the mediator's consistency method with respect to the argument sets chosen for the parent modules of $\mathcal{A}_i$ (we call such update contexts *acceptable* for $Med_i$ with respect to the state). More formally,

**Definition 11.** *Let* $\mathcal{F} = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$ *be an* $\mathcal{ACS}$. *A* state of $\mathcal{F}$ *is a function* $\mathcal{S}$ *that maps each* $\mathcal{M}_i = (\mathcal{A}_i, Med_i)$ *to a pair* $\mathcal{S}(\mathcal{M}_i) = (Acc_i, C_i)$, *where* $Acc_i$ *is a subset of the arguments of* $\mathcal{A}_i$ *and* $C_i$ *is an update context for* $\mathcal{A}_i$.

$\mathcal{S}$ *is* acceptable, *if (i) each* $Acc_i$ *is an acceptable* $C_i$*-extension for* $\mathcal{A}_i$, *and (ii) each* $C_i$ *is an acceptable context for* $Med_i$ *wrt.* $\mathcal{S}$.

We refer the reader to [10] for more details and illustrative examples.

## 5 Ongoing and Future Work

The work on MCS from the previous sections is, beyond improvements and further elaborations, complemented with several streams of ongoing and future work. In this section, we briefly consider some of them, where focus on generalizing MCS based on ACS.

### 5.1 Handling Incomplete Information

The approach to explain inconsistency in an MCS that we considered above is based on the assumption that the agent performing this tasks has an omniscient view of the system, i.e., has full information about each context, including the logic, the knowledge base, and the bridge rules. However, in real world scenarios, it is not expected that all this information is available, and some of the contexts are like "black boxes" in which the knowledge base and the logic respectively the precise semantics of the (in detail unknown) knowledge base is hidden. Such information hiding may be desired for various reasons, e.g. security, privacy or business motivated (intellectual property) etc. One may only have *partial knowledge* about the context, like the input-output behavior in particular cases, and some additional knowledge about properties of this behavior.

[3] This means the mediator's bridge rules only refer to $\mathcal{A}_1, \ldots, \mathcal{A}_k$ in their bodies.

In [21], an approach to deal with partial knowledge about MCS is developed using a two-level representation:

- the lower level aims to capture the known behavior of a context on various inputs and outputs in terms of a partially defined Boolean function, i.e., a Boolean function that leaves the value for some inputs open. Here, the complete semantics of a context $C$ is abstracted to a Boolean function $f_C(\boldsymbol{i}, \boldsymbol{o})$, where $\boldsymbol{i} = i_1, \ldots, i_n$ are the beliefs that occur in the bodies of the bridge rules of $C$, and $\boldsymbol{o} = o_1, \ldots, o_m$ are the beliefs from $C$ which occur in the bodies of any bridge rule in the MCS, and all $i_j$, $o_k$ are viewed as Boolean variables; intuitively, $\boldsymbol{i}$ and $\boldsymbol{o}$ are input and output beliefs of $C$, while all other beliefs in $C$ are hidden. The function takes the value 1 iff after adding the input beliefs according to $\boldsymbol{i}$ to the knowledge base, there exists some acceptable belief set for the augmented knowledge base in which the membership of the output beliefs is given by $\boldsymbol{o}$. The implementation of MCS described above uses essentially this abstraction.
- The higher level captures specific properties like consistency, monotonicity, functionality etc. which can be exploited to reason about context properties.

Then, using this representation, semantic approximations of diagnoses and explanations are developed, in terms of under- and over-approximations of actual diagnoses and explanation under complete information. Furthermore, strategies are discussed how to query contexts with a limited number of queries for some inputs to improve the approximation accuracy.

As it appears, dealing with incomplete information about contexts is not easy, and best methods to deal with it will require quite some efforts.

## 5.2 Aggregating Information

In real world scenarios, MCS may include some rather plain contexts whose knowledge bases are relational databases and the “logics” of their belief semantics will basically be given by querying these databases. An important aspect for databases is the possibility to aggregate data, for example for salary data, to find out the maximum, average, or sum of all salaries. In standard database query languages, this is supported via designated aggregation constructs; more recently, such aggregates have also been introduced in Answer Set Programming. Given that bridge rules allow to “import” data from a context, it would be desirable to perform aggregation of such data also in bridge rules, given that the access to data (and aggregation at the source context) might not be feasible.

To introduce useful aggregates in bridge rules requires in fact two extensions of the MCS formalism above: firstly, one has to move from a sentential setting of the context logics to a predicate logic setting, where we can talk about objects and their properties; e.g., $sal(1, joe, 20K)$ encoding the salary of an employee with internal ID 1 and name Joe. Then, atoms in bridge rules should be atoms in the predicate languages (permitting constants and variables). Secondly, we need aggregation constructs, and besides syntax also a semantics.

Current work at the KBS group of TU Vienna is developing such an extension, in which open bridge rules (with variables) have a grounding semantics over a (possibly virtual) domain shared by the contexts. For example, the import of the sum of salaries from a database relation $sal$ at some context $C_i$ could be accomplished by the following bridge rule:

$$total\_sal_i(X) \leftarrow X = \#sum\{S, I : (i : sal(I, N, S))\} \tag{9}$$

Here the atom in the body involves an aggregation expression according to the DLV convention, where each fact $sal(I, N, S)$ distinct on $S$ and $I$ (and as $I$ determines $S$, distinct on $I$) is considered and the values of $S$ are summed up. Aggregates of this form will be very useful to aggregate information from different sources, e.g., salary data in different databases, especially if this data is stored in heterogeneous formats.

Another interesting form of aggregation will be to collect information across different contexts, using also variables to access contexts. This will be particularly useful to gather a global picture in a MCS, which is needed for applications in which social choice based on group decisions (e.g., voting) plays a role. For example, an expression $\#count\{C : (C : vote(yes))\}$ could count all contexts $C$ which have the fact $vote(yes)$ in their belief set. Buccafurri et al.'s work on Social ASP [12,13] is in this direction, based on Answer Set Programs as logics at particular contexts. However, while support for aggregating information is available, this and a similar MCS formalism still lack a mechanism for decision making according to a protocol, which would need to be encoded from first principles.

## 5.3 Dynamic MCS

The use of variables for contexts is also important for a generalization of MCS that takes a dynamic configuration into account. So far an MCS $M$ is static in the sense that the contexts and the information interlinkage via bridge rules is fixed. One can imagine, however, that $M$ is not completely specified at the beginning, but has to be formed from a pool $P$ of contexts that may be connected with each other. For example, in a scenario different contexts may provide ontological information about diseases, but only one of the contexts should be included in the MCS. A natural problem is then, given a schematically described MCS $M$, to pick contexts from the pool for the generic contexts in the schema and concrete information links for the bridge rules such that a working MCS results. In a sense, this may be viewed as a configuration problem where the components are contexts and the constraints between the components are given by schematic bridge rules.

The KBS group at TU Vienna currently develops a framework for dynamic MCS of this kind, where besides variables for contexts also variable for beliefs in schematic bridge rules are introduced. The instantiation of the context proceeds by finding concrete contexts whose beliefs match with the beliefs $p$ that are mentioned in the bridge rules. To this end, a matchmaker is involved which determines the quality of match between $p$ and beliefs $p'$ in some context $C$; as perfect matches will not always exist, best matching beliefs may be considered, with a loss of accuracy. The task is then to find, given a starting set $S$ of schematic contexts from the pool $P$, a binding for each context in $S$ to contexts and beliefs in them, such that all variables are eliminated and an ordinary MCS results; this may involve the addition of further contexts. As in general, multiple solutions exist, an optimal one according to some criteria (e.g., the accuracy of information, and/or the number of contexts involved) may be desired. Given that finding optimal solutions is expensive, reasonable heuristics will be important (possibly also general qualitative preferences like in [16]). A formal framework and a respective algorithm configuration for dynamic configuration, which proceeds by neighborhood search, will be available in the near future. It will also move MCS closer to peer-to-peer systems, where also systems of peers are demand-driven formed ad hoc following a neighborhood selection. Furthermore, dynamic addition and removal of contexts to respectively from the pool (which may be large) can be accomplished.

## 5.4 Towards Mediator Based MCS

As we have seen, ACS go beyond MCS as they allow one context to modify another by not only adding, but also deleting information. Moreover, mediators in ACS provide inconsistency handling methods which make sure the required modifications make sense. On the other hand, ACS are not heterogeneous: they require a specific context logic, namely Dung argumentation frameworks. The natural question to ask, then, is: can we have the best of both worlds and generalize both ACS and MCS in such a way that heterogeneous contexts can be used, and yet contexts can be updated as in ACS?

In this section we give some initial answers to this question and introduce mediator based MCS (MMCS). Starting from the original MCS definitions, gradually more functionality is added to the contexts. Contrary to ACS where mediators and contexts are separated, we integrate the two within the contexts. A motivation for this is simplicity and uniformity of the overall approach, and closeness to the original MCS definitions.

**Step 1: Revising rather than augmenting KBs.** From an abstract point of view, a mediator can be seen as a function taking as input a set of - possibly conflicting - expressions specifying how the context's knowledge base is to be updated, and producing as output a revised knowledge base, respectively a collection of revised knowledge bases, each of them representing an acceptable update. If there is a conflict in the mediator's input, then the mediator is expected to resolve this conflict.

The first extension we thus need is a language for specifying relevant modifications. For this language we rely, in this first step, on operations studied in the area of belief revision (see [6] for a collection of recent papers), namely expansion (adding a formula), contraction (making a formula underivable) and revision (integrating a formula in a consistent way). We may use expressions of the form $add(F)$, $contract(F)$, $revise(F)$, respectively, to denote these operations, where $F$ is a formula of the respective context logic. The set of update expressions for logic $L$ will be denoted $UP_L$.

Expressions of this language replace formulas in the heads of a context's bridge rules. An MMCS bridge rule for logic $L$ thus is of the form

$$s \leftarrow (r_1 : p_1), \ldots, (r_j : p_j), \mathbf{not}\ (r_{j+1} : p_{j+1}), \ldots, \mathbf{not}\ (r_m : p_m) \qquad (10)$$

where $s \in UP_L$ and the body elements are as before. Intuitively, update expressions allow us to specify the required knowledge base modifications via applicable bridge rules. However, it may be necessary for the mediator to resolve conflicts within the set of update expressions generated by the bridge rules. For this reason, each context $C_i$ has an additional mediator component $med_i$, which associates with each pair $(U, kb_i) \in 2^{UP_{L_i}} \times KB_{L_i}$ a set of knowledge bases from $KB_{L_i}$ which are intuitively the different possible outcomes of the update (which in general might not be unique).

An MMCS is (as before) a structure $M = (C_1, \dots, C_n)$, yet now each $C_i$ is an MMCS context. What still needs to be defined is a new notion of equilibrium. Intuitively, an equilibrium is a belief state such that the belief set chosen for each context is a belief set of one of the KBs which result from applying the mediator to the heads of applicable bridge rules (first argument) and $kb_i$ (second argument).

More formally, this is captured as follows: Let $M = (C_1, \dots, C_n)$ be an MMCS, $S = (S_1, \dots, S_n)$ a belief state, that is each $S_j$ is a belief set of context $C_j$. $S$ is an equilibrium iff for each $i$, $1 \leq i \leq n$, we have

$$S_i \in ACC_i(kb_i^*) \text{ for some } kb_i^* \in med_i(up_i, kb_i)$$

where $up_i = \{s \in UP_{L_i} \mid s \text{ head of a bridge rule } r \in br_i, r \text{ applicable in } S\}$.

Clearly the original MCS are special cases of MMCS: given an MCS $M_1$ we can define an associated MMCS $M_2$ with the same equilibria. We just have to replace the heads of all bridge rules in $M_1$ with corresponding $add$ statements, generating in this way the bridge rules form $M_2$. The mediators are defined in such a way that they just pass on the additional information, that is all modifications are just augmentations.

**Step 2: Multiple semantics.** ACS allow the semantics of a context logic to be controlled explicitly. To introduce this useful feature in MMCS, we need a more general form of logics, allowing for a choice of semantics. We assume that for each logic $L$ an index set $I_L$ is given. The indices are used to refer to the different semantics.

We then can view a logic is a tuple $L = (KB_L, BS_L, \{Acc_L^i\}_{i \in I_L})$, where $KB_L$ and $BS_L$ are as before, and $I_L$ is an index set such that each index denotes a semantics.

Thus, the different semantics a logic may have are represented using a collection of $Acc$ functions. In addition, we need to foresee expressions in the update language $UP_L$ which allow a semantics to be fixed. This is simply achieved by introducing in $UP_L$ expressions of the form $sem(i)$ where $i \in I_L$. Finally, the mediator must produce a second output, namely one of the semantics:

$$med_i : 2^{UP_{L_i}} \times KB_{L_i} \rightarrow 2^{KB_{L_i}} \times I.$$

An equilibrium now must satisfy the additional condition that the belief set selected for each context must be a belief set under the chosen semantics. The equilibrium condition thus becomes:

$$S_i \in ACC_i^j(kb_i^*) \text{ for some } kb_i^* \in KB^*$$

where $med_i(up_i, kb_i) = (KB^*, j)$ and $up_i = \{s \in UP_{L_i} \mid s \text{ head of a bridge rule } r \in br_i, r \text{ applicable in } S\}$.

In ACS a distinction was made not only between semantics (grounded, preferred, stable) but also between reasoning modes (skeptical vs. credulous) for each argumentation context. For simplicity, whenever an arbitrary context semantics allows for a distinction between different reasoning modes, we subsume this under the semantics. For instance, credulous stable reasoning and skeptical stable reasoning for logic programs are considered as two different semantics.

**Step 3: Keeping track of sources of update information.** So far our mediators are still very abstract. We simply assumed that the mediator takes care of potential inconsistencies within the set of update expressions obtained through applicable bridge rules - without specifying how this is to be done. Similarly, we assumed that, once conflicts are resolved, the mediator produces a revised knowledge base (or a collection of such knowledge bases) - again without specifying how.

There are numerous methods for conflict handling, and a whole area of research, namely belief revision, is investigating how to revise belief sets, respectively belief bases, in the light of new information. We believe that different ways of instantiating mediators will be useful, in particular given the large variety of underlying logics multi-context systems encompass. Also, a pluralistic view regarding conflict resolution and revision methods fits well to the pluralistic view regarding context logics. For this reason we refrain from specifying particular methods at this point and prefer to keep the MMCS framework as abstract as it currently is.

However, we would at least like to point out a further extension of the framework that will be useful for a variety of conflict handling methods. Most of these methods will, in some way or the other, take information about the sources of update information (and potentially information about the reliability of sources) into account. The input of the mediator function so far does not contain this source information. In the context of MMCS it is natural to identify sources with the context(s) from which the update information stems (the contexts appearing in the bodies of bridge rules). Another natural option would be to take the particular bridge rule used to derive the information as its source. The mediator then could be based on preference information, stored in a local mediator knowledge base, about the context's bridge rules.

To realize this, the input of the mediator function would not be just a set of update expressions $e$, it would consist of pairs $(e, s)$ where $s$ is the source of expression $e$. How the modification is computed out of this information is then an issue which is internal to the mediator.

## 6 Related Work and Conclusion

Most closely related to MCS as considered above are the pioneering multi-context systems of the Trento School that we have already briefly discussed in the Introduction.

**Distributed ontologies.** Other work that has been carried out at the Trento group and which is related to MCS is work on distributed ontologies. Here, the scenario is that ontology parts, which reside in multiple knowledge bases that are formulated in a Description Logic, should be put together into a single global ontology, such that particular reasoning tasks across the knowledge base can be solved. Here in fact different views of the ontology information in the knowledge bases are possible: seeing them as parts of a dispersed or divided ontology (in database terms akin to the local-as-view scenario), or as relatively autonomous units which should be synthesized in a bottom up fashion (akin to global-as-view in databases). A recent survey and comparison of various approaches to distributed ontologies, is given in [38]. There, four main formalisms have been considered, viz. distributed description logics (DDL) [7], $\mathcal{E}$-connections [40], package-based description logics (P-DL) [4], and integrated DDL (IDDL) [46]. They are based on different principles and offer interlinkage with special bridge respectively integration axioms that allow to enforce semantic relationships between concepts and role across knowledge bases, or to important concepts and roles with their semantics relationships. However, there are some differences to MCS.

Firstly, distributed ontologies often assume a homogenous, or at least a similar, format of the knowledge bases which is in contrast to the heterogeneity of MCS. Secondly, distributed ontologies aim at a tight integration and an alignment of the signatures, but also of the domain of global models. To this end, different kinds of mappings between domain elements for the knowledge interlinkage are considered, and the linkage is at the level of concepts and roles, applied to individuals. In MCS, there is prima facie no domain of discourse and the linkage is at the sentential level; however, the work on aggregation mentioned in the previous section is heading towards such a more fine-grained setting. Thirdly, nonmonotonicity does not play a role in distributed ontologies.

**Combining rules and ontologies.** Besides distributed ontologies also the issue of combining rules and ontologies, which has been a hot topic in the recent years in the KR and Semantic Web research, is related to MCS (see [18] for a survey). In particular, [25] presented an approach in which a logic program $P$ can query a Description Logic knowledge base $L$ using special DL-query atoms in rule bodies. Prior to evaluating the query, $L$ may be augmented with assertions that copy the value of predicates in $P$ to assertions about concepts respectively roles in $L$. A nonmonotonic description logic (DL) program $KB$ then consists of a finite set $P$ of such generalized rules and $L$. We may see $KB$ as an MCS, with one context for $P$ and one for each query atom in $P$; the information flow between $P$ and $L$ can be emulated using bridge rules that are quite natural (but cumbersome to write). Furthermore, to capture the semantics of $KB$, one may need to resort to notions of grounded equilibria. While in this particular setting, MCS come close to a knowledge integration formalism, one has to keep in mind that plain MCS are not formalisms for knowledge integration per se, which adhere to a special integration semantics or perform alignment of knowledge. This would have to be defined either on top of an MCS, or encoded via the context logics. The mediator MCS discussed above are closer to a knowledge integration formalism, providing means to incorporate beliefs from other knowledge bases respectively contexts according to a particular incorporation semantics, which can emulate integration respective alignment to some extent. A detailed comparison of MCS to the many formalisms for combining rules and ontologies is beyond the scope of this article.

**Multi-agent systems.** Another area that naturally seems to be closely related to MCS are multi-agent systems, where agents share information over a communication infrastructure such that some global state emerges from the local states of the agents; see e.g. [44,45]. However, there is a salient difference between a multi-agent system and a multi-context system. In multi-agent systems, an important element even for information exchange is communication, which proceeds usually in multiple steps according to a protocol (see for example agent communication languages like KQML [28] or FIPA ACL [29]), which may foresee different options of how to deal with information requests and to integrate information into an agent's knowledge base. The communication primitives are designed according to the specific needs of agents taking into account fundamental aspects like autonomy, intentions and goals of an agent, etc. In particular, an agent may refuse information. In MCS, there are no such mechanisms for information exchange according to a protocol; rather, the semantics establishes a spontaneous "information equilibrium" via the links given by bridge rules, without any higher goals or intentions. Such goals and intentions, and a protocol, respectively policy, specifying how to deal with new information, would have to be mimicked within the knowledge base (assuming the underlying logic is capable of encoding the relevant aspects). We note that [12] presents "Communicating Answer Set Programs", whose semantics is closely related to minimal equilibria semantics of MCS and especially the semantics of Social ASP [12,13]. However, no mechanism for protocol support etc is incorporated there.

**Game theory.** We might disregard communication and other particular distinctive features of agents, and consider ensembles of "agents" from a more abstract perspective as the one in game theory; also here, we find salient differences, despite the possibility to view equilibria of an MCS as game-theoretic solutions.

Assume that an outcome (i.e., belief state) $S = (S_1, \ldots, S_n)$, has for $C_i$ reward 1 if $S_i \in \mathbf{ACC}_i(kb_i \cup \{head(r) \mid r \in app(br_i, S)\})$ and 0 otherwise. Then, it is easy to see that each equilibrium of $M$ is a *Nash equilibrium* of this game (indeed, each player has optimal reward); on the other hand, there might be Nash equilibria that do not correspond to any equilibrium. This may happen e.g. if no acceptable belief sets are possible. For instance, the MCS $M = (C_1, C_2)$, where $C_1$ and $C_2$ are isolated answer set programs $\{a \leftarrow \text{not } a\}$, has no equilibrium, but $S = (\emptyset, \emptyset)$ is a Nash-equilibrium of the game. Clearly, if $M$ has equilibria, then they coincide with the *Pareto-optimal* solutions of the game; under additional conditions (e.g., $\mathbf{ACC}_i(b_i \cup H_i) \neq \emptyset$ for each $H_i \subseteq \{h(r) \mid r \in br_i\}$) they coincide with the Nash equilibria.

However, these characterizations are technical in nature and employ games with trivial instances of the important aspects of an agent from a game-theoretic view: a non-trivial utility function of different outcomes and a (sophisticated) rational strategy of behavior. On the other hand, as MCS just serve for basic belief interlinkage without a deeper theory behind, only such a simple utility function and strategy may be expected.

**Fibring logics.** The combination of logical systems is a natural problem that has been considered abundantly, and many approaches have been developed. An influential one is Gabbay's *Fibring Logics* [30], which roughly aims at obtaining the logics $\mathcal{L}$ built over the combined languages of two logics $\mathcal{L}_1$ and $\mathcal{L}_2$ that conservatively extend each logic

$\mathcal{L}_i$ and are minimal in this respect. It involves two issues: characterizing the notion of a logical system (using e.g. algebraic structures and proof systems) and characterizing the methods how to combine logics in general. Here, syntactic and semantic based combinations can be conceived, where the latter raise more difficulties; for more discussion, see e.g. [14].

Fibring logics and similar approaches are only remotely related to MCS. The notion of "logic" in MCS is very abstract. Contrary to the respective component in Gabbay's approach, it does not require specific features of a logical system (such as e.g. a proof system or inference rules). This takes into account that, as often in realistic situations, the "logics" in use at a context may not be fully described logical systems. Hence, also "fibring" does not seem to make sense for arbitrary MCS (or would be trivial). Furthermore, bridge rules - an essential ingredient in MCS - do not have an apparent counterpart in the fibring construction.

**Distributed SAT and CSP.** From a computational point, evaluating a decentralized MCS as in Section 2.3 can be reduced to distributed SAT solving, given that the local semantics of each context (i.e., the acceptability function for belief sets) can be encoded to a SAT instance. In fact, this is the approach underlying the implementation of the DMCS system [15,1,2]. Overall, this means that the computation of a global equilibrium is reducible to solving a distributed SAT instance; algorithms like the family of MULTI-DB algorithms in [37] may be used for that. Similar as in Answer Set Programming, where transformations of ASP programs to SAT instances can take advantage of improvements on SAT solvers, reductions of decentralized MCS to distributed SAT can benefit from advances on distributed SAT solvers. In connection with this, it would be interesting to explore the usability of distributed CSP solving for some of the extensions of MCS mentioned above. In turn, algorithms for Distributed SAT/CSP may inspire algorithms for MCS that are based on specific contexts, e.g., ASP contexts.

## 6.1 Conclusion

In this paper, we have reviewed nonmonotonic multi-context systems (MCS) from [9] that emerged from previous work of the Trento School on multi-context systems, as a flexible and generic formalism that allows for the interlinkage of heterogeneous (possibly nonmonotonic) knowledge bases via possibly non-monotonic bridge rules. We have discussed recent and ongoing work on MCS, as well as variants and possible generalizations that are planned for the future.

It should be re-emphasized that the ground-breaking work of Michael Gelfond on the semantics of logic programs with negation, and in particular the way to define answer set semantics, has been fundamental for our work on MCS, like for many other formalisms and systems that we have developed. We are looking forward to face new frontiers and problems in knowledge representation where Michael's results can be fruitfully applied, and undoubtedly there will be plenty of them.

**Acknowledgement.** The authors are grateful for the constructive review comments.

## References

1. 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.) [39]
2. Bairakdar, S.E.-D., Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: The DMCS solver for distributed nonmonotonic multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) [39]
3. Balduccini, M., Gelfond, M.: Logic programs with consistency-restoring rules. In: International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series, pp. 9–18 (2003)
4. Bao, J., Honovar, V.: Extension to support collaborative ontology building. In: Poster & Demonstration Proceedings of the 3rd International Semantic Web Conference (ISWC 2004), page PID 37 (2004) (poster)
5. Bögl, M., Eiter, T., Fink, M., Schüller, P.: The MCS-IE system for explaining inconsistency in multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) [39]
6. Bonanno, G., Delgrande, J.P., Lang, J., Rott, H.: Special issue on formal models of belief change in rational agents. J. Applied Logic 7(4), 363 (2009)
7. Borgida, A., Serafini, L.: Distributed description logics: Directed domain correspondences in federated information sources. In: Meersman, R., Tari, Z. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519, pp. 36–53. Springer, Heidelberg (2002)
8. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. J. Artif. Intell. Res (JAIR) 21, 135–191 (2004)
9. Brewka, G., Eiter, T.: Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In: AAAI 2007, pp. 385–390. AAAI Press, Menlo Park (2007)
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. Brewka, G., Roelofsen, F., Serafini, L.: Contextual Default Reasoning. In: IJCAI 2007, pp. 268–273 (2007)
12. Buccafurri, F., Caminiti, G.: Logic programming with social features. TPLP 8(5-6), 643–690 (2008)
13. Buccafurri, F., Caminiti, G., Laurendi, R.: A logic language with stable model semantics for social reasoning. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 718–723. Springer, Heidelberg (2008)
14. Caleiro, C., Sernadas, A., Sernadas, C.: Fibring logics: Past, present and future. In: Artëmov, S.N., Barringer, H., d'Avila Garcez, A.S., Lamb, L.C., Woods, J. (eds.) We Will Show Them! (1), pp. 363–388. College Publications (2005)
15. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed nonmonotonic multi-context systems. In: Proceedings 12th International Conference on Principles of Knowledge Representation and Reasoning (KR 2010), Toronto, Canada, May 9-13 (2010)
16. Dell'Acqua, P., Pereira, L.M.: Preferring and updating in logic-based agents. In: Bartenstein, O., Geske, U., Hannebauer, M., Yoshie, O. (eds.) INAP 2001. LNCS (LNAI), vol. 2543, pp. 70–85. Springer, Heidelberg (2003)
17. Dix, J.: A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. Fundamenta Informaticae XXII(3), 257–288 (1995)
18. Drabent, W., Eiter, T., Ianni, G., Krennwallner, T., Lukasiewicz, T., Małuszyński, J.: Hybrid reasoning with rules and ontologies. In: Bry, F., Małuszyński, J. (eds.) Semantic Techniques for the Web. LNCS, vol. 5500, pp. 1–49. Springer, Heidelberg (2009)

19. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77(2), 321–358 (1995)
20. Eiter, T., Brewka, G., Dao-Tran, M., Fink, M., Ianni, G., Krennwallner, T.: Combining nonmonotonic knowledge bases with external sources. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 18–42. Springer, Heidelberg (2009)
21. Eiter, T., Fink, M., Schüller, P.: Approximations for explanations of inconsistency in partially known multi-context systems. In: Informal Proceedings Conference Thirty Years of Nonmonotonicty (NonMon30), Lexington, October 22-25 (2010)
22. Eiter, T., Fink, M., Schüller, P., Weinzierl, A.: Finding explanations of inconsistency in multi-context systems. In: Proceedings 12th International Conference on Principles of Knowledge Representation and Reasoning (KR 2010), Toronto, Canada, May 9-13 (2010)
23. Eiter, T., Fink, M., Weinzierl, A.: Preference-based inconsistency assessment in multi-context systems. In: Janhunen, T., Niemelä, I. (eds.) [39]
24. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 290–309. Springer, Heidelberg (1997)
25. 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); Preliminary version Tech.Rep. INFSYS RR-1843-07-04, Inst. Information Systems, TU Vienna (January 2007)
26. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: International Joint Conference on Artificial Intelligence, pp. 90–96 (2005)
27. 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)
28. Finin, T.W., Fritzson, R., McKay, D.P., McEntire, R.: Kqml as an agent communication language. In: CIKM, pp. 456–463. ACM, New York (1994)
29. Foundation for Intelligent Physical Agents (FIPA). Fipa2000 agent specification (2000), `http://www.fipa.org`
30. Gabbay, D. (ed.): Fibring Logics. Oxford University Press, Oxford (1999)
31. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation. Foundations of Artificial Intelligence, ch. 7, pp. 285–316. Elsevier, Amsterdam (2007)
32. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP 1988, pp. 1070–1080. MIT Press, Cambridge (1988)
33. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and deductive databases. New Generation Computing 9, 365–385 (1991)
34. Ghidini, C., Giunchiglia, F.: Local models semantics, or contextual reasoning=locality+compatibility. Artif. Intell. 127(2), 221–259 (2001)
35. Giunchiglia, F.: Contextual reasoning. Epistemologia XVI, 345–364 (1993)
36. Giunchiglia, F., Serafini, L.: Multilanguage hierarchical logics, or: How we can do without modal logics. Artificial Intelligence 65(1), 29–70 (1994)
37. Hirayama, K., Yokoo, M.: The distributed breakout algorithms. Artif. Intell. 161(1-2), 89–115 (2005)
38. Homola, M.: Semantic Investigations in Distributed Ontologies. PhD thesis, Comenius University, Bratislava, Slovakia (April 2010)
39. Janhunen, T., Niemelä, I. (eds.): JELIA 2010. LNCS, vol. 6341. Springer, Heidelberg (2010)
40. Kutz, O., Wolter, F., Zakharyaschev, M.: Connecting abstract description systems. In: Fensel, D., Giunchiglia, F., McGuinness, D.L., Williams, M.-A. (eds.) KR, pp. 215–226. Morgan Kaufmann, San Francisco (2002)

41. Lifschitz, V., Turner, H.: Splitting a logic program. In: International Conference on Logic Programming (ICLP), pp. 23–37 (1994)
42. McCarthy, J.: Generality in artificial intelligence. Commun. ACM 30(12), 1029–1035 (1987)
43. Roelofsen, F., Serafini, L.: Minimal and absent information in contexts. In: Proc. IJCAI 2005 (2005)
44. Weiss, G. (ed.): Multiagent Systems. A Modern Approach to Distributed Artifical Intelligence. MIT-Press, Cambridge (2000) (2nd print)
45. Wooldridge, M.: An Introduction to MultiAgent Systems. John Wiley and Sons, Chichester (2002); 2nd edn. (2009)
46. Zimmermann, A.: Integrated distributed description logics. In: Calvanese, D., Franconi, E., Haarslev, V., Lembo, D., Motik, B., Turhan, A.-Y., Tessaris, S. (eds.) Description Logics. CEUR Workshop Proceedings, vol. 250, CEUR-WS.org (2007)

# Perspectives on Logic-Based Approaches for Reasoning about Actions and Change

Agostino Dovier[1], Andrea Formisano[2], and Enrico Pontelli[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica
agostino.dovier@uniud.it
[2] Univ. di Perugia, Dip. di Matematica e Informatica
formis@dmi.unipg.it
[3] New Mexico State University, Dept. Computer Science
epontell@cs.nmsu.edu

**Abstract.** Action languages have gained popularity as a means for declaratively describing planning domains. This paper overviews two action languages, the Boolean language $\mathcal{B}$ and its multi-valued counterpart $\mathcal{B}^{MV}$. The paper analyzes some of the issues in using two alternative logic programming approaches (Answer Set Programming and Constraint Logic Programming over Finite Domains) for planning with $\mathcal{B}$ and $\mathcal{B}^{MV}$ specifications. In particular, the paper provides an experimental comparison between these alternative implementation approaches.

## 1 Introduction

As illustrated by Lifschitz [19], research on planning requires the resolution of two key problems: development of languages for the description of planning problems—using declarative and elaboration tolerant notations—and design of efficient and scalable planning algorithms.

Action languages [15] have gained popularity over the years as viable declarative notations for the description of planning domains. Since the original proposal of the languages $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ by Gelfond and Lifschitz [15], a variety of new action languages have appeared in the literature, with declarative features aimed at capturing important features of real-world planning domains, such as preferences [27], time and duration s[1], numerical reasoning [17], and beliefs [13].

In recent years, we have witnessed an increased interest in exploring ways of bridging the gap between the declarative problem encodings offered by action language and the development of effective implementations. In particular, an interesting line of work has been developed to study the relationships between action languages and *logic programming*. The link between these two paradigms is quite natural, considering the logical foundations underlying the semantics of most action languages. Furthermore, this direction of research is fueled by some very attractive properties of logic programming, such as:

- Research in logic programming has significantly enhanced the performance of modern logic programming inference engines; for example, answer set solvers are currently competitive with state-of-the-art SAT solvers (e.g., clasp in the 2009 SAT Competition—http://www.satcompetition.org).

- Logic programming implementations of action languages maintain the declarative nature of the original encoding, enabling, for example, to maintain a good level of elaboration tolerance in the executable encoding.
- The declarative nature of logic programming makes it feasible to envision the use of user-defined search strategies, expressed as logic programming theories. Furthermore, it facilitates the orthogonal introduction of domain knowledge, that can be used to guide the search for solutions during planning.

The advent of *Answer Set Programming (ASP)* [20, 22] has significantly impacted the area of logic programming encoding of action languages—the support for non-monotonic reasoning provided by ASP nicely matches the needs of action languages (e.g., facilitating the resolution of the frame problem [21]).

Over the last few years, we have embarked on a comparative investigation of the features of two of the most popular logic programming paradigms—*answer set programming* and *constraint logic programming over finite domains (CLP(FD))* [16]. Some preliminary results have been presented in [3, 4, 6]. Recently, this line of work has focused to the investigation of the respective strengths and weaknesses of ASP and CLP(FD) in dealing with planning problems and action languages. We have investigated the relative performances of the two paradigms on different classes of planning problems and on different types of action languages [5, 10, 7, 8].

In this paper, we continue this line of work with several contributions:

- We explore some modifications of the encodings in both ASP (Section 3) and CLP(FD) (Section 4), leading to significant improvements in performance;
- We make use of the state-of-the-art systems in ASP and CLP(FD)—in particular, ASP technology has made significant improvements since our previously published results (e.g., [10]);
- We expand the pool of benchmarks, including more challenging problems like the reverse folding problem and the tangram (Section 6);
- We emphasize the role of multi-valued fluents in gaining efficiency in planning and compactness of domain descriptions (Section 5).

## 2 The Action Language $\mathcal{B}$

In this section, we revisit the syntax and semantics of the action description language $\mathcal{B}$. The syntax and semantics presented in the following sections is a slight modification of the original definitions from the seminal paper of Gelfond and Lifschitz [15].

### 2.1 Syntax of $\mathcal{B}$

An action signature consists of a set $\mathcal{F}$ of *fluent* names, a set $\mathcal{A}$ of *action* names, and a set $\mathcal{V}$ of values for fluents in $\mathcal{F}$. In this section, we consider Boolean fluents, hence $\mathcal{V} = \{0, 1\}$ (or $\{\text{false}, \text{true}\}$). A *fluent literal* is either a fluent $f$ or its negation $\mathtt{neg}(f)$. Fluents and actions are concretely represented by *ground* atomic formulae $p(t_1, \ldots, t_m)$ from an underlying logic language $\mathcal{L}$—where $p$ is a predicate symbol and $t_1, \ldots, t_m$ are ground terms. We assume that the set of allowed terms for $\mathcal{L}$ is finite.

The language $\mathcal{B}$ allows us to specify an *(action) domain description* $\mathcal{D}$. The core components of a domain description are its *fluents*—properties used to describe the state of the world, that may dynamically change in response to execution of actions—and *actions*—denoting how an agent can affect the state of the world. Fluents and actions are introduced by assertions of the forms `fluent(`$f$`)` and `action(`$a$`)`. An action description $\mathcal{D}$ relates actions and fluents using axioms of the following types —where `[list-of-conditions]` denotes a list of fluent literals:

– `causes(`$a$`, `$\ell$`, [list-of-conditions])`: this axiom encodes a *dynamic causal law*, describing the effect (i.e., truth assignment to the fluent literal $\ell$) of the execution of the action $a$ in a state of the world that satisfies all the conditions in `[list-of-conditions]`;
– `caused([list-of-conditions], `$\ell$`)`: this axiom describes a *static causal law*—i.e., the fact that the fluent literal $\ell$ is true in any state satisfying all the given preconditions.

Moreover, preconditions can be imposed on the executability of actions by means of assertion of the forms:

– `executable(`$a$`, [list-of-conditions])`: this axiom asserts that, for the action $a$ to be executable, all the given conditions have to be satisfied in the current state of the world.

A *domain description* is a set of static causal laws, dynamic laws, and executability conditions. A specific *planning problem* $\langle \mathcal{D}, \mathcal{O} \rangle$ contains a domain description $\mathcal{D}$ along with a set $\mathcal{O}$ of *observations* describing the *initial state* and the *desired goal*, specified using the following types of statements:

– `initially(`$\ell$`)` asserts that the fluent literal $\ell$ is true in the initial state of the world;
– `goal(`$\ell$`)` asserts that the goal requires the fluent literal $\ell$ to be true in the final state of the world.

In the concrete specification of an action theory, we will allow a Prolog-like syntax to express in a more succinct manner the laws of the theory. For instance, to assert that in the initial state of the world all fluents are true, we can simply write the following rule:

```
initially(F) :- fluent(F).
```

instead of writing a fact `initially(`$f$`)` for each possible fluent $f$.

## 2.2 Semantics of $\mathcal{B}$

We will rely on sets of fluent literals to describe a state of the world. If $\ell$ is a fluent literal, and $S$ is a set of fluent literals, we say that $S \models \ell$ if and only if $\ell \in S$. A list of literals $L = [\ell_1, \ldots, \ell_m]$ denotes a conjunction of literals, hence $S \models L$ if and only if $S \models \ell_i$ for all $i \in \{1, \ldots, m\}$. We denote with $\neg S$ the set $\{f \in \mathcal{F} : \texttt{neg}(f) \in S\} \cup \{\texttt{neg}(f) : f \in S \cap \mathcal{F}\}$. We are interested in considering only sets of literals that satisfy certain properties:

– A set of fluent literals is *consistent* if there is no fluent $f$ s.t. $S \models f$ and $S \models \texttt{neg}(f)$.
– If $S \cup \neg S \supseteq \mathcal{F}$ then $S$ is *complete*.
– A set $S$ of literals is *closed* w.r.t. a set of static laws

$$\mathcal{SL} = \{\texttt{caused}(L_1, \ell_1), \ldots, \texttt{caused}(L_m, \ell_m)\}$$

if, for all $i \in \{1, \ldots, m\}$, it holds that $S \models L_i$ implies $S \models \ell_i$.

The set $\texttt{Clo}_{\mathcal{SL}}(S)$ is defined as the smallest set of literals containing $S$ and closed w.r.t. $\mathcal{SL}$. $\texttt{Clo}_{\mathcal{SL}}(S)$ is uniquely determined and not necessarily consistent. Whenever we are working with a domain description $\mathcal{D}$, we will also denote with $\texttt{Clo}_{\mathcal{D}}(S)$ the result of $\texttt{Clo}_{\mathcal{SL}}(S)$ where $\mathcal{SL}$ is the set of all static causal laws in $\mathcal{D}$.

Let $\mathcal{D}$ be an action description defined on the action signature $\langle \mathcal{V}, \mathcal{F}, \mathcal{A} \rangle$, composed of dynamic laws $\mathcal{DL}$, executability conditions $\mathcal{EL}$, and static causal laws $\mathcal{SL}$. The semantics of $\mathcal{D}$ is given in terms of a transition system $\langle \mathcal{S}, \nu, R \rangle$, consisting of a set $\mathcal{S}$ of states, a total interpretation function $\nu : \mathcal{S} \to \mathcal{F} \to \mathcal{V}$ (in this section $\mathcal{V} = \{0, 1\}$), and a transition relation $R \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$.

Given a transition system $\langle \mathcal{S}, \nu, R \rangle$ and a state $s \in \mathcal{S}$, let:

$$Lit(s) = \{f \in \mathcal{F} \,:\, \nu(s)(f) = 1\} \cup \{\texttt{neg}(f) \,:\, f \in \mathcal{F},\ \nu(s)(f) = 0\}.$$

Observe that $Lit(s)$ is consistent and complete.

Given the set of all the dynamic causal laws

$$\{\texttt{causes}(a, \ell_1, L_1), \ldots, \texttt{causes}(a, \ell_m, L_m)\}$$

for the action $a \in \mathcal{A}$ present in $\mathcal{D}$ and a state $s \in \mathcal{S}$, we define the *(direct) effects of $a$ in $s$* as follows:

$$E_{\mathcal{D}}(a, s) = \{\ell_i : 1 \leqslant i \leqslant m, Lit(s) \models L_i\}.$$

The action $a$ is said to be *executable* in a state $s$ w.r.t. $\mathcal{D}$ if it holds that

$$Lit(s) \models \bigvee_{i=1}^{h} C_i, \tag{1}$$

where $\texttt{executable}(a, C_1), \ldots, \texttt{executable}(a, C_h)$ for $h > 0$, are the executability axioms for the action $a$ in $\mathcal{D}$. Observe that multiple executability axioms for the same action $a$ are considered disjunctively. Hence, for each action $a$, at least one executable axiom must be present in the action description.

The transition system $\langle \mathcal{S}, \nu, R \rangle$ *described by* $\mathcal{D}$ is such that:

– $\mathcal{S}$ is the set of all states $s$ such that $Lit(s)$ is closed w.r.t. $\mathcal{SL}$;
– $R$ is the set of all triples $\langle s, a, s' \rangle$ such that $a$ is executable in $s$ and

$$Lit(s') = \texttt{Clo}_{\mathcal{D}}(E_{\mathcal{D}}(a, s) \cup (Lit(s) \cap Lit(s'))) \tag{2}$$

Let $\langle \mathcal{D}, \mathcal{O} \rangle$ be a planning problem instance, where $\{\ell \mid \texttt{initially}(\ell) \in \mathcal{O}\}$ is a consistent and complete set of fluent literals. A *trajectory* in $\langle \mathcal{S}, \nu, R \rangle$ is a sequence

$$\langle s_0, a_1, s_1, a_2, \cdots, a_{\mathsf{N}}, s_{\mathsf{N}} \rangle$$

such that $\langle s_i, a_{i+1}, s_{i+1} \rangle \in R$ for all $i \in \{0, \ldots, \mathsf{N} - 1\}$.

A sequence of actions $\langle a_1, \ldots, a_{\mathsf{N}} \rangle$ is a solution (a *plan*) to the planning problem $\langle \mathcal{D}, \mathcal{O} \rangle$ if there is a trajectory $\langle s_0, a_1, s_1, \ldots, a_{\mathsf{N}}, s_{\mathsf{N}} \rangle$ in $\langle \mathcal{S}, \nu, R \rangle$ such that:

– $Lit(s_0) \models r$ for each `initially`$(r) \in \mathcal{O}$, and
– $Lit(s_{\mathsf{N}}) \models \ell$ for each `goal`$(\ell) \in \mathcal{O}$.

The plans characterized in this definition are *sequential*—i.e., we disallow concurrent actions. Observe also that the desired plan length $\mathsf{N}$ is assumed to be given.

## 3 Answer Set Planning

The idea of using logic programming to address planning problems dates back to the origins of logic programming [28]. The idea of using extended logic programming and answer set programming can be traced back to the seminal works of Gelfond and Lifschitz [14] and Subrahmanian and Zaniolo [25]. The various encodings proposed in the literature tend to share similar ideas—fluents are represented by atoms of a logic program, with an additional parameter used to represent the state $s_i$ of a trajectory they refer to.

### 3.1 The General Encoding

Let us describe how a domain $\mathcal{D}$ and a problem instance $\langle \mathcal{D}, \mathcal{O} \rangle$ can be mapped to a logic program $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$; the intuition is that the mapping should guarantee that there is a one-to-one correspondence between plans of length $\mathsf{N}$ for $\langle \mathcal{D}, \mathcal{O} \rangle$ and answer sets of $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$. In the rest of this section, we illustrate the construction of $\Pi_{\mathcal{D}}$ as performed by a Prolog translator developed by the authors—and available at `www.dimi.uniud.it/CLPASP`. The structure of the translation follow the general lines delineated in [19, 23].

The initial components of $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$ are facts used to identify actions and fluents of the domain; for each $f \in \mathcal{F}$ and for each $a \in \mathcal{A}$ we assume that the facts

```
fluent(f).          action(a).
```

are present in $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$.

The encoding of the content of $\mathcal{O}$ is also immediate: for each `initially`$(\ell)$ and for each `goal`$(\ell')$ in $\mathcal{O}$ we assume the presence of analogous facts in $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$ —i.e., $\mathcal{O} \subseteq \Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$.

Auxiliary rules are introduced in order to provide the definition of some of the concepts used in the definition of the semantics of domain specifications; in particular, we introduce in $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$ rules aimed at defining the notions of literal and complement of a literal, as follows:

```
literal(F) :- fluent(F).          literal(neg(F)) :- fluent(F).
complement(F,neg(F)).             complement(neg(F),F).
```

The parameter $\mathsf{N}$ is used to denote the length of the desired trajectory; we introduce the facts `time`$(0..\mathsf{N})$ to identify the range of time points in the desired trajectory.

The states $s_i$ of a trajectory (for $i = 0, \dots, \mathsf{N}$) are described by the predicate `holds`; intuitively, $\nu(s_i)(f) = 0$ iff $\mathtt{holds}(\mathtt{neg}(f), i)$ is true and $\nu(s_i)(f) = 1$ iff $\mathtt{holds}(f, i)$ is true.

The various axioms lead to the following rules:

– The executability conditions for an action $a$ provide the definition of a predicate `possible`. Let us assume that $\mathtt{executable}(a, L_1), \dots, \mathtt{executable}(a, L_h)$ are all the executability axioms for $a$ in $\mathcal{D}$, and let us assume that for $j \in \{1, \dots, h\}$: $L_j = [\ell_1^j, \dots, \ell_{r_j}^j]$. Then the following rules are provided in $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$:

$$\begin{array}{l}\mathtt{possible}(a,T)\ \mathtt{:-}\ \mathtt{time}(T), \mathtt{holds}(\ell_1^1,T), \dots, \mathtt{holds}(\ell_{r_1}^1,T).\\ \qquad\qquad\qquad \cdots \\ \mathtt{possible}(a,T)\ \mathtt{:-}\ \mathtt{time}(T), \mathtt{holds}(\ell_1^h,T), \dots, \mathtt{holds}(\ell_{r_h}^h,T).\end{array}$$

– Each static causal law $\mathtt{caused}([\ell_1, \dots, \ell_r], \ell)$ leads to the introduction of a rule of the form

$$\mathtt{holds}(\ell,T)\mathtt{:-}\mathtt{time}(T), \mathtt{holds}(\ell_1,T), \dots, \mathtt{holds}(\ell_r,T).$$

– Each dynamic causal law $\mathtt{causes}(a, \ell, [\ell_1, \dots, \ell_r])$ in $\mathcal{D}$ introduces the following rule

$$\mathtt{holds}(\ell,T+1)\mathtt{:-}\mathtt{time}(T), \mathtt{occ}(a,T), \mathtt{holds}(\ell_1,T), \dots, \mathtt{holds}(\ell_r,T).$$

– The constraint that ensures consistency of each state constructed is

$$\mathtt{:-}\mathtt{time}(T), \mathtt{fluent}(F), \mathtt{holds}(F,T), \mathtt{holds}(\mathtt{neg}(F),T).$$

– The rule that provides the solution to the frame problem is

$$\begin{array}{ll}\mathtt{holds}(L,T+1)\ \mathtt{:-} & \mathtt{time}(T), \mathtt{literal}(L), \mathtt{holds}(L,T),\\ & \mathtt{complement}(L,L_1),\ \mathbf{not}\ \mathtt{holds}(L_1,T+1).\end{array}$$

The following additional rules are needed to model the instance $\mathcal{O}$:

– In order to model the initial state, we need the additional rule to generate the description of the state at time 0:

$$\mathtt{holds}(L,0)\mathtt{:-}\mathtt{initially}(L).$$

– In order to model the satisfaction of the goal, we introduce the constraint

$$\mathtt{:-}\mathtt{goal}(L),\ \mathbf{not}\,\mathtt{holds}(L,\mathsf{N}).$$

The following final rule is used to support the generation of a plan:

– The rules that generate the sequence of actions constituting the plan are:

$$\begin{array}{l}1\{\mathtt{occ}(A,T)\ :\ \mathtt{action}(A)\}1\mathtt{:-}\mathtt{time}(T), T < \mathsf{N}.\\ \mathtt{:-}\mathtt{action}(A), \mathtt{time}(T), \mathtt{occ}(A,T),\ \mathbf{not}\,\mathtt{possible}(A,T).\end{array}$$

**Proposition 1.** *Let us consider a planning problem instance $\langle \mathcal{D}, \mathcal{O} \rangle$ and the program $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$ constructed as discussed earlier. $\langle a_1, \dots, a_{\mathsf{N}} \rangle$ is a plan for $\langle \mathcal{D}, \mathcal{O} \rangle$ iff there is an answer set $M$ of $\Pi_{\mathcal{D}}(\mathsf{N}, \mathcal{O})$ such that $\{\mathtt{occ}(a_1, 0), \dots, \mathtt{occ}(a_{\mathsf{N}}, \mathsf{N}-1)\} \subseteq M$.*

### 3.2 An Optimized Encoding

If the action theory does not contain any static causal laws, then it becomes possible to simplify the translation to ASP. In particular, it becomes possible to avoid the creation of separate atoms for representing negative literals. At the semantic level, we can observe that, in absence of static causal laws, the formula (2) becomes

$$Lit(s') = (Lit(s) \setminus \neg(E_{\mathcal{D}}(a,s))) \cup E_{\mathcal{D}}(a,s)$$

Practically, this simplification leads to the following changes to the ASP encoding:

- In the encoding of the executability conditions, for each axiom

  $$\texttt{executable}(a, [p_1, \ldots, p_r, \texttt{neg}(q_1), \ldots, \texttt{neg}(q_s)]).$$

  we can generate the rule

  $$\begin{array}{l}\texttt{possible}(a,T)\texttt{:-}\ \texttt{time}(T), \texttt{holds}(p_1,T), \ldots, \texttt{holds}(p_r,T),\\ \qquad\qquad\qquad\qquad \textbf{not}\,\texttt{holds}(q_1,T), \ldots,\ \textbf{not}\,\texttt{holds}(q_s,T).\end{array}$$

- The encoding of the dynamic causal laws of the form $\texttt{causes}(a, f, L)$, for a fluent $f$, is as before, while each law of the form

  $$\texttt{causes}(a, \texttt{neg}(r), [p_1, \ldots, p_r,\ \textbf{not}\, q_1, \ldots,\ \textbf{not}\, q_s])$$

  in $\mathcal{D}$ introduces the following rules

  $$\begin{array}{l}\texttt{:-}\ \texttt{holds}(r, T+1), \texttt{time}(T), \texttt{occ}(a,T),\\ \quad \texttt{holds}(p_1,T), \ldots, \texttt{holds}(p_r,T),\\ \quad \textbf{not}\,\texttt{holds}(q_1,T), \ldots,\ \textbf{not}\,\texttt{holds}(q_s,T).\\ \texttt{non\_inertial}(r, T+1)\texttt{:-}\texttt{time}(T), \texttt{occ}(A,T).\end{array}$$

- Finally, the frame problem has a slightly different encoding: we exploit the above rules, defining $\texttt{non\_inertial}$, together with the rule:

  $$\begin{array}{l}\texttt{hold}(F, T+1)\texttt{:-}\ \texttt{time}(T), \texttt{fluent}(F), \texttt{hold}(F),\\ \qquad\qquad\qquad\qquad \textbf{not}\,\texttt{non\_inertial}(F, T+1).\end{array}$$

The main advantage of this encoding is to reduce the number of atoms and the size of the ground version of the ASP encoding. However, considering our experiments, this smaller grounding does not always guarantee better performance in the solving phase.

## 4 Planning Using CLP

In this section, we illustrate the main aspects of the encoding of the language $\mathcal{B}$ into constraint logic programming for the purpose of planning. Specifically, the target of the encoding is a constraint logic program over finite domains (CLP(FD)). The model presented here is an evolution of the pioneering work described in [10], with several modifications aimed at enhancing performance.

As for the ASP encoding, we are interested in computing plans with $\mathsf{N}$ action occurrences, relating a sequence of $\mathsf{N}+1$ states $s_0, \ldots, s_{\mathsf{N}}$. For each state $s_i$ and for each fluent $f$, we introduce a Boolean variable $F^i$ to describe the truth value of $f$ in $s_i$. The value of the literal $\texttt{neg}(F^i)$ is simply $1 - F^i$. A list of literals $\alpha = [p_1, \dots, p_k, \texttt{neg}(q_1), \dots, \texttt{neg}(q_h)]$ interpreted as a conjunction of literals in a state $i$ is described by a variable $\hat{\alpha}^i$ defined by the constraint:

$$\hat{\alpha}^i \equiv \left( \bigwedge_{j=1}^{k} P^i_j = 1 \wedge \bigwedge_{j=1}^{h} Q^i_j = 0 \right)$$

We will also introduce, for each action $a$, a Boolean variable $A^i$, representing whether the action is executed or not in the transition from $s_{i-1}$ to $s_i$.

Let us consider a state transition between $s_i$ to $s_{i+1}$; we develop constraints that relate the variables $F^{i+1}$, $F^i$, and $A^{i+1}$ for each fluent $f$ and for each action $A$. This is repeated for $i = 0, \dots, \mathsf{N} - 1$. Moreover, constraints regarding initial state and goal are added.

Let us consider a fluent $f$, and let

$$\begin{array}{rcl}
\texttt{causes}(a_{t_1}, f, \alpha_1) & \cdots & \texttt{causes}(a_{t_m}, f, \alpha_m) \\
\texttt{causes}(a_{z_1}, \texttt{neg}(f), \beta_1) & \cdots & \texttt{causes}(a_{z_p}, \texttt{neg}(f), \beta_p) \\
\texttt{caused}(\delta_1, f) & \cdots & \texttt{caused}(\delta_h, f) \\
\texttt{caused}(\gamma_1, \texttt{neg}(f)) & \cdots & \texttt{caused}(\gamma_k, \texttt{neg}(f))
\end{array}$$

be all of the dynamic and static laws that have $f$ or $\texttt{neg}(f)$ as their consequences. For each action $a_j$ let us assume that its executability conditions are the following:

$$\texttt{executable}(a_j, \eta_{r_1}) \quad \cdots \quad \texttt{executable}(a_j, \eta_{r_q})$$

Figure 1 describes the Boolean constraints that can be used in encoding the relations that determine the truth value of the fluent literal $F^{i+1}$. A fluent $f$ is true in state $i + 1$ (see rule (3)) if a dynamic rule or a static rule explicitly forces it to be true (captured by $\texttt{PosFired}_f$) or if it was true in state $i$ and no dynamic or static rule forces it to be false (expressed by $\texttt{NegFired}_f$). The constraint (4) forbids the execution of static/dynamic rules with contradictory consequences, thus ensuring the consistency of the states being created. The constraints (5) and (8) defines the conditions that make a fluent true or false in the following state, either as effect of an action execution (constraints (6) and (9)) or as result of static causal laws being triggered (constraints (7) and (10)). Two additional constraints on actions are also added. The constraint (11) states that at least one executability condition must be fulfilled in order for an action to occur. The constraint (12) states that exactly one action per transition is allowed.

As a technical consideration, differently from older versions of the solver (e.g. [5, 10]) conjunction and disjunction constraints are implemented using the built-in CLP(FD) predicate `minimum` and `maximum`, respectively. Moreover, constraints (3) and (4) regarding four variables, are dealt with the combinatorial constraint `table` (only six 4-tuples are candidate solutions). This allows us to restrict the search to the 6 solutions of the two combined constraints, instead of blindly exploring the 16 possible combinations of values at each state transition. Several other minor code optimizations have been implemented.

$$F^{i+1} = 1 \equiv \texttt{PosFired}_f^i \vee (\neg\texttt{NegFired}_f^i \wedge F^i = 1) \tag{3}$$

$$\neg\texttt{PosFired}_f^i \vee \neg\texttt{NegFired}_f^i \tag{4}$$

$$\texttt{PosFired}_f^i \equiv \texttt{PosDyn}_f^i \vee \texttt{PosStat}_f^{i+1} \tag{5}$$

$$\texttt{PosDyn}_f^i \equiv \bigvee_{j=1}^{m} (\hat{\alpha}_j^i \wedge A_{t_j}^{i+1} = 1) \tag{6}$$

$$\texttt{PosStat}_f^i \equiv \bigvee_{j=1}^{h} \hat{\delta}_j^i \tag{7}$$

$$\texttt{NegFired}_f^i \equiv \texttt{NegDyn}_f^i \vee \texttt{NegStat}_f^{i+1} \tag{8}$$

$$\texttt{NegDyn}_f^i \equiv \bigvee_{j=1}^{p} (\hat{\beta}_j^i \wedge A_{z_j}^{i+1} = 1) \tag{9}$$

$$\texttt{NegStat}_f^i \equiv \bigvee_{j=1}^{k} \hat{\gamma}_j^i \tag{10}$$

---

$$A_j^{i+1} = 1 \rightarrow \bigvee_{j=1}^{q} \hat{\eta}_{r_j}^i \tag{11}$$

$$\sum_{a_j \in \mathcal{A}} A_j^i = 1 \tag{12}$$

**Fig. 1.** Constraints for the fluent $f$ and for all the actions $a_j$ in the transition $(s_i, s_{i+1})$

## 5 From Boolean to Multi-valued

We have investigated several extensions of $\mathcal{B}$ (see, e.g., [10]). In this section, we summarize the extension which allows the use of multi-valued fluents in the description of a domain and references to values of fluents in past states. We refer to this extended version of the language as $\mathcal{B}^{MV}$.

The syntax of the action language is modified to allow the declaration of a domain for each fluent—the domain indicates the set of values that can be assigned to each fluent. The *domain declarations* have the form

$$\texttt{fluent}(f, \{v_1, \ldots, v_k\})$$

For the sake of simplicity, we restrict our attention to domains containing integer numbers. If the domain is an interval $[a \ldots b]$ of integer numbers, one is allowed to write simply: $\texttt{fluent}(f, a, b)$.

Fluents can be used in *Fluent Expressions (*FE*)*, which are defined inductively as follows:

$$\texttt{FE} ::= n \mid f^t \mid \texttt{FE} \oplus \texttt{FE} \mid \texttt{rei}(\texttt{FC})$$

where $n \in \mathbb{Z}$, $\oplus \in \{+, -, *, /, \texttt{mod}\}$, $t \in \mathbb{N}$ with $t \leq 0$, and $f \in \mathcal{F}$. The notation $f^0$ will be often written simply as $f$, and it refers to the value of $f$ in the current state; the notation $f^i$ denotes the value the fluent $f$ had in the $i^{th}$ preceding state. The expression $\texttt{rei}(C)$ denotes the reification of a the constraint $C$ (i.e., 1 if $C$ is entailed, 0 otherwise). FC are *Fluent Constraints* and they are defined as follows:

$$\texttt{FC} ::= \texttt{FE rel FE} \mid \neg\texttt{FC} \mid \texttt{FC} \wedge \texttt{FC} \mid \texttt{FC} \vee \texttt{FC}$$

where $\texttt{rel} \in \{=, \neq, \geq, \leq, >, <\}$. We will also refer to fluent constraints of the type FE rel FE as *primitive fluent constraints.*

The language $\mathcal{B}^{MV}$ allows one to specify an action domain description, which relates actions, states, and fluents using predicates of the following forms ($c$ denotes a primitive fluent constraint, while $C$ is a fluent constraint):

- Axioms of the form `executable`$(a, C)$, stating that the fluent constraint $C$ has to be entailed by the current state for the action `a` to be executable.
- Axioms of the form `causes`$(a, c, C)$ encode dynamic causal laws. When the action $a$ is executed, if the constraint $C$ is entailed by the current state, then state produced by the execution of the action is required to entail the primitive constraint $c$.
- Axioms of the form `caused`$(C, c)$ describe static causal laws. If the fluent constraint $C$ is satisfied in a state, then the constraint $c$ must also hold in such state.

For example, a dynamic causal law can have the form:

$$\begin{aligned} causes(pour(X,Y), contain(Y) = contain(Y)^{-1} + contain(X)^{-1},\\ [Y - contain(Y)^0 \geq contain(X)^0]). \end{aligned}$$

The description of the semantics of this modified version of the language is beyond the scope of this paper; it requires two major changes: *(1)* a state is now a function that assigns to each fluent a value drawn from the fluent's domain; *(2)* the truth of a fluent constraint is expressed with respect to a trajectory, in order to enable the resolution of the time references on the fluents. For example, a trajectory $\langle s_0, a_1, s_1, \cdots, a_k, s_k\rangle$ entails the constraint $f^0 = f^{-1} + f^{-2}$ if the value of $f$ in $s_k$ is equal to the sum of the value of $f$ in $s_{k-1}$ and the value of $f$ in $s_{k-2}$. The translation to CLP(FD) is also a relatively simple extension of what discussed earlier; the main changes are: *(1)* the variables $F^i$ are no longer Boolean variables, but they are finite domain variables, whose domain is derived from the domain declarations in the action language; *(2)* the constraints of Figure 1 need to map annotated fluents $f^t$ to corresponding variables $F^{i+t}$. The interested reader is referred to [10] for more details.

## 6 Experiments and Evaluation

We report here the results on experiments performed on a collection of domains used as benchmarks. Some of these domains (and instances) have been selected from problems presented in the last ASP competition (`dtai.cs.kuleuven.be/events/ASP-competition/`) and in some of the past International Planning Competitions (e.g., `ipc.informatik.uni-freiburg.de/`).

For problems modeled in $\mathcal{B}$ we used the following two approaches:

- **ASP:** We first translated the domain, given the desired plan length, using the translator described in Section 3. The result of the translation was processed by the gringo grounder [12] and the answer sets computed using the clasp [11] answer set solvers.[1]
- **CLPFD:** In this case we compile the solver `SICSplan` presented in Section 4 together with the domain and ask for the existence of a plan of a given length N.

[1] We used the combination of gringo and clasp since it provides the fastest ASP solver currently available. Other systems such as lparse+smodels/cmodels can be used as well.

In this case different search heuristics have been used. It is a relatively simple exercise to use different CLP(FD) systems—e.g., translations to B-Prolog have been investigated.

All the domains, the instances, the compiler, and the solvers are available at `www.dimi.uniud.it/CLPASP`.[2]

For the problems modeled in the $\mathcal{B}^{MV}$ language, we used the $CLP(FD)$ solver `SICSplanMV`. The $\mathcal{B}^{MV}$ domains have been also translated to the corresponding Boolean versions, where each multi-valued fluent $f$ with domain $\{a_1, \dots, a_k\}$ has been replaced by $k$ propositional fluents $f_1, \dots, f_k$, and the axioms changed accordingly. Let us make some observations about the drawbacks of this translation to the Boolean case. Let us consider, for instance, two fluents $f$ and $g$, each with the interval $1 \dots 100$ as domain; let us also assume that a dynamic causal law has the following effect:

$$f = f^{-1} + g^{-1}$$

This is a unique constraint on three variables in the $\mathcal{B}^{MV}$ encoding. In its propositional version, this constraint becomes:

$$\text{for } \{X, Y, Z\} \subseteq \{1, \dots, 100\} \text{ s.t. } Z = X + Y\text{: } f_X^{-1} \wedge g_Y^{-1} \rightarrow f_Z$$

This implies the use of 300 fluents and $4,950$ ground constraints:

$$\begin{array}{lll} f_1^{-1} \wedge g_1^{-1} \rightarrow f_2 & f_1^{-1} \wedge g_2^{-1} \rightarrow f_3 \quad \cdots & f_1^{-1} \wedge g_{99}^{-1} \rightarrow f_{100} \\ \quad\quad\vdots & & \\ f_{99}^{-1} \wedge g_1^{-1} \rightarrow f_{100} & & \end{array}$$

More in general, if the domain contains $k$ values and the constraint includes 3 fluents, the Boolean encoding will required $3k$ Boolean fluents and the ground version of the constraint will lead to $O(k^2)$ constraints.

An alternative encoding can be realized using a logarithmic encoding of numbers. In the example above, for each fluent we can introduce 7 Boolean fluents, say, $f_{b_6}, \dots, f_{b_0}$, each representing one bit of the binary encoding of the value of the fluent. Then, the various rules will have the form:

$$\begin{array}{lllllll} \texttt{neg}(f_{b_6}^{-1}) \wedge \texttt{neg}(f_{b_5}^{-1}) \wedge & f_{b_4}^{-1} \wedge & f_{b_3}^{-1} \wedge & f_{b_2}^{-1} \wedge & f_{b_1}^{-1} \wedge & f_{b_0}^{-1} & \wedge \\ \texttt{neg}(g_{b_6}^{-1}) \wedge \texttt{neg}(g_{b_5}^{-1}) \wedge & f_{g_4}^{-1} \wedge & f_{g_3}^{-1} \wedge & f_{g_2}^{-1} \wedge & f_{g_1}^{-1} \wedge & f_{g_0}^{-1} & \rightarrow \\ \texttt{neg}(f_{b_6}) \wedge \quad f_{b_5} \quad \wedge & f_{b_4} \wedge & f_{b_3} \wedge & f_{b_2} \wedge & f_{b_1} \wedge & \texttt{neg}(f_{b_0}) & \end{array}$$

This is the the rule for the sum $31 + 31 = 62$. In general, for domains with $k$ elements, we will need for each fluent $b = \lceil \log_2 k \rceil$ Boolean fluents, and the number of constraints becomes $O(2^b 2^b) = O(k^2)$, which leads to the same overall complexity of encoding.

## 6.1 Domains Used

We briefly describe here the domains used for testing the two approaches to handle planning domains.

[2] A slightly adapted version of the solver, tailored to be executed by B-Prolog, is available too.

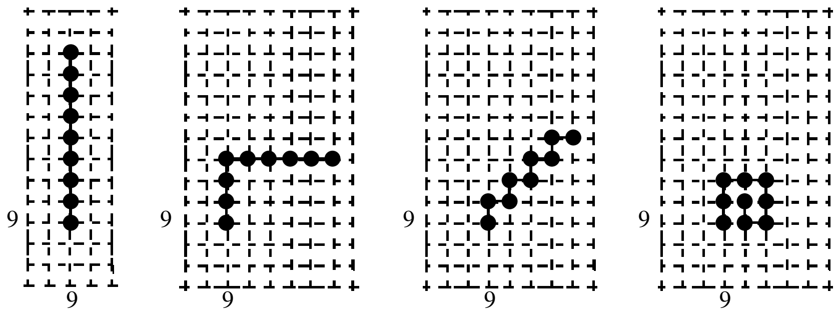


**Fig. 2.** Reverse folding problem. Four foldings with $k$=9: The initial (straight line) folding, the result of a clockwise pivot move on the 4th element, a zigzag folding, and a spiral folding.

From the 2009 Answer Set Competition, We encoded the *Peg Solitaire*, the classical *Sam Lloyd's 15 puzzle*, and the *Towers of Hanoi* problems, drawn from the Asparagus repository.[3] We also include the *Hydraulic planning* problem by Michael Gelfond, Ricardo Morales, and Yuanlin Zhang. This is a simplified version of the hydraulic system on a space shuttle, that is modeled with a directed graph, where nodes are labeled as tanks, jets, or junctions, and every link between two nodes is labeled by a valve. Tanks can be full or empty. Valves can be opened or closed. A node of G is *pressurized* if it is a full tank or if there exists a path from some full tank to this node such that all the valves on the edges of this path are open. The problem is to find a shortest sequential plan to pressurize a given node.

Instead of looking for a solution exploiting graph algorithms (as done, e.g., by the Potassco group in the ASP competition), we modeled the problem as a domain expressed in $\mathcal{B}$ and left the search to the solvers. We also developed a multi-valued numerical extension of this problem that points out the benefits of multi-valued modeling language.

We also encoded the following additional problems:

- The *trucks* domain from the IPC5 planning competition;
- A generalized version of the classical barrels problem; the generalization uses the parameters $2k/k+1/k-1$, for $k \in \mathbb{N}$: there are three barrels of capacity $2k/k+1/k-1$. At the beginning, the largest barrel is full of wine while the other two are empty. We wish to reach a state in which the two larger barrels contain the same amount of wine and the third is empty. The only permissible action is to pour wine from one barrel to another, until the latter is full or the former is empty.
- The *Gas Diffusion problem*, originally proposed in [10]. A building contains a number of rooms. Each room is connected to (some) other rooms via gates. Initially, all gates are closed and some of the rooms contain a quantity of gas—while the other rooms are empty. Each gate can be opened or closed—`open(x,y)` and `close(x,y)` are the only possible actions, provided that there is a gate between room `x` and room `y`. When a gate between two rooms is open, the gas contained in these rooms flows through the gate. The gas diffusion continues until the pressure

[3] `asparagus.cs.uni-potsdam.de/contest`

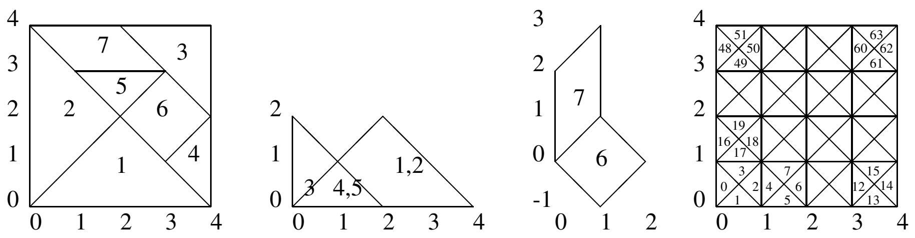


**Fig. 3.** Tangram solution, "base" position of the seven blocks, and space discretization using triangles

reaches an equilibrium. The only condition to be always satisfied is that a gate in a room can be opened only if all the other gates are closed. The goal is to move a desired quantity of gas to one specified room.

- A simplified version of the *reverse folding problem*. Given a string (e.g., representing a protein) composed of $k$ consecutive elements, we wish to place it on a 2D plane (e.g., a 2D grid). The only admissible angles are 0 (straight line), $-90°$ (left turn) and $+90°$ (right turn). Different elements must occupy different positions. We refer to each placement of the string as a *folding*. A *pivot move* is obtained by selecting an element $i \in \{2, \dots, k-1\}$ and turning clockwise or counter-clockwise the part of the string related to the elements $i+1, \dots, k$.

  The simplified reverse folding problem we propose is the following: given two foldings in a plane, such that points 1 and 2 are set in the positions $(k, k)$ and $(k, k+1)$ of the grid, we wish to find the sequence of pivot moves that transforms the first folding into the second. In our tests, we set the initial folding as a straight line, while the final foldings is set either as a sort of stair (zigzag) or as a spiral (see Figure 2).
- The *Tangram* puzzle: there are seven blocks of different forms (see Figure 3) and a form to be reconstructed (we just focus on the big square). The challenge for its representation is that the sizes of the blocks are related by the irrational number $\sqrt{2}$ and therefore cannot be easily discretized. We encoded it in $\mathcal{B}$ based on a discretization of the space in small triangles. Each move puts a block in a certain point and with a certain rotation—we allow 8 angles: $0°, 180°, \pm 45°, \pm 135°, \pm 90°$. Our implementation is inherently Boolean and does not benefit from multi-valued representations.

## 6.2 Experimental Results

We experimented with several instances for each of the domains described earlier. The $\mathcal{B}$ encodings have been translated into ASP, as described in Section 3, and solved using `gringo 2.0.5` and `clasp 1.3.3`. The CLP-based planners for $\mathcal{B}$ and $\mathcal{B}^{MV}$ (named $\mathcal{B}$-SICSplan and $\mathcal{B}^{MV}$-SICSplan, resp.) have been executed in SICStus Prolog version 4.1.1. All planners have been executed on an AMD Opteron 2.2GHz Linux machine.

An excerpt of the experimental results is reported in the Appendix. Tables 1–3 show the results for the $\mathcal{B}$-solvers while those regarding $\mathcal{B}^{MV}$-SICSplan are reported in Table 4. For the ASP solver, we separately report the time required for grounding

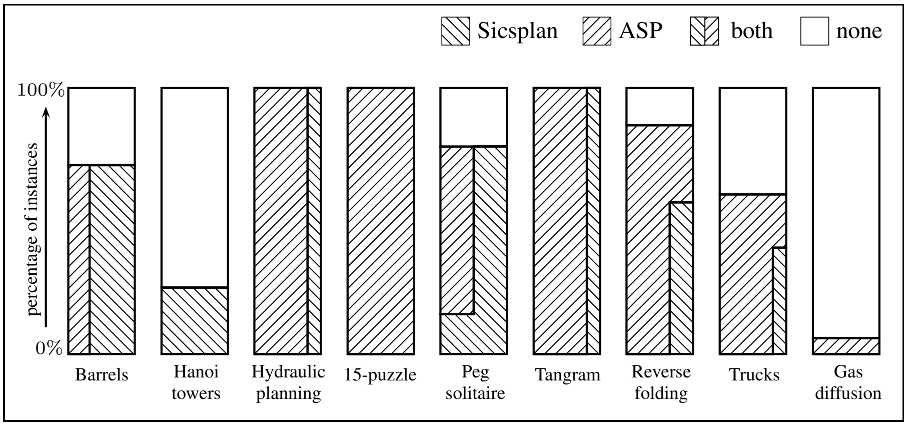


**Fig. 4.** Qualitative comparison of the ASP- and CLP-based solvers for $B$. Each bar shows the percentage of instances solved by $\mathcal{B}$-SICSplan, clasp, both solvers, or left unsolved within the fixed time limits. When part of the instances are solved by both solvers, the thickness of the bars reflects their relative efficiency (i.e., the larger, the faster).

the ASP program and the time for solving the instances. Similarly, for the CLP-based solvers, we separately report the time spent in imposing the constraint and the time to perform the search for solution (using labeling). The symbol "M" denotes that the solver ran out of memory (a bound of 2GB was imposed for each instance); "T" denotes that the problem could not be solved within a fixed period of time (a time bound of 60 minutes was imposed for the instances of the Towers of Hanoi domain, the bound was 30 minutes for all other domains).

The experiments confirmed that action languages such as $\mathcal{B}$, are suitable for expressing highly declarative specifications of planning domains. Moreover, the experiments indicate that this approach represents a viable alternative to solve even complex planning problems with reasonable efficiency. This has been made possible by recent improvements in the available implementations of non-monotonic and constraint logic programming. We believe that solvers for $\mathcal{B}$ are reaching a sufficiently mature stage of development to become, in the near future, competitive with state-of-the-art planners that exploit various kinds of problem-dependent heuristics in reasoning. However, as reported in [8], our approach already outperforms (in terms of efficiency) other logic programming approaches to reasoning with action and changes like GOLOG [18] and Flux [26] when they are used for planning.

The CLP-based implementations of $\mathcal{B}$ are competitive with the state-of-the-art ASP-solver clasp. Nevertheless, clasp remains a better choice whenever the length of the plan is large. We believe that further improvements in the strategies adopted in the labeling phase have to be designed in order to amend such weakness of $\mathcal{B}$-SICSplan.

Figure 4 visualizes a qualitative comparison of the results obtained by the two solvers for $\mathcal{B}$. Each bar shows the percentage of instances solved by $\mathcal{B}$-SICSplan, clasp, both solvers, or left unsolved within the fixed time limits. When part of the instances have been solved by both solvers, the thickness of the bars reflects their relative efficiency.

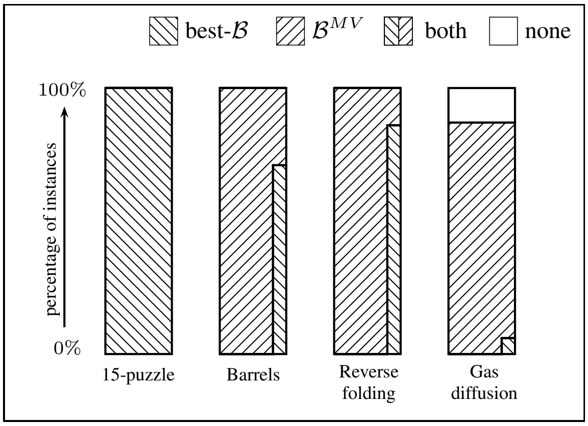


**Fig. 5.** Qualitative comparison between $\mathcal{B}^{MV}$-SICSplan and the best among the solvers for $\mathcal{B}$

E.g., for the Barrels domain, clasp and $\mathcal{B}$-SICSplan solved the same instances but the latter showed better performance. On the other hand, for the Trucks domain clasp was faster and solved more instances.

The results obtained by $\mathcal{B}^{MV}$-SICSplan (Table 4) confirm that the introduction of multi-valued fluents and constraints as first-class objects of the action language, allows us to develop more compact encodings—requiring a smaller number of fluents and actions. This translates in a faster constraint-solving phase and, consequently, in the ability of $\mathcal{B}^{MV}$-SICSplan to solve more instances that the solvers for $\mathcal{B}$. This is particularly evident for those domains where numerical fluents can be naturally introduced (c.f. also the summary of the analysis reported in Figure 5).

## 7 Current Directions and Conclusion

In this paper, we summarized the current results from an experimental study aimed to compare ASP and CLP(FD) in the encoding of action description languages. In particular, we emphasized some of the new features of the proposed encoding, such as the use of the `table` constraint to speed-up computation of state transitions, and the impact of the new answer set solvers (e.g., clasp) on the performance of ASP-based planners. The investigation relied on a new set of benchmarks, drawn from different sources and encoded using both Boolean and multi-valued action description languages.

The current directions of our investigation are pushing towards the development of new action description languages that can better meet the needs of real-world planning domains, while taking full advantage of the features of the underlying logic programming inference engines (e.g., the features offered by modern constraint logic programming systems). Some of the current directions being pursued are described next.

- *Multi-agency:* we are investigating extensions of $\mathcal{B}$ and $\mathcal{B}^{MV}$ to support the description of multi-agent domains, where agents can interact in different ways (e.g., cooperatively, competitively). Several features have been already investigated, including the creation of a core modeling framework and its encoding in CLP(FD)

[7, 8], the modeling of cooperative features like *negotiation* [24], and the use of reasoning about agents' knowledge and beliefs [13]. While the majority of the current approaches rely on a *centralized* perspective in the description of multi-agent systems, we have also launched an investigation of how logic programming (specifically CLP(FD) with blackboard-style mechanisms) can be used to provide a distributed implementation of a multi-agent action domain language [9].

- *Heuristics:* logic programming's ability to implement search strategies has not been properly employed to enhance efficiency of logic-based planning. CLP(FD)'s ability of dealing with search structures and with graphs is expected to provide very effective ways of implementing both well-known search heuristics used by the planning community (e.g., graph plan [2]) as well as new heuristics made possible by the declarative specification of planning domains provided by the action languages. The interaction between planning heuristics and search strategies explored by the constraint programming community is also an open area of investigation that we intend to explore.

*Acknowledgments.* The research has been partially supported by grants GNCS-INdAM: *Tecniche innovative per la programmazione con vincoli in applicazioni strategiche*; MUR-PRIN: *Innovative and multidisciplinary approaches for constraint and preference reasoning*; *Ricerca di base 2009–cod.2009.010.0336*; NSF grants IIS-0812267, CBET-0754525, and HRD-0420407.

## References

[1] Baral, C., Son, T., Tuan, L.-C.: A transition function based characterization of actions with delayed and continuous effects. In: Fensel, D., Giunchiglia, F., McGuinness, D.L., Williams, M.-A. (eds.) KR 2002: Principles of Knowledge Representation and Reasoning, pp. 291–302. Morgan Kaufmann, San Francisco (2002)

[2] Blum, A., Furst, M.: Fast planning through planning graph analysis. Artificial Intelligence 90, 281–300 (1997)

[3] Dovier, A., Formisano, A., Pontelli, E.: A comparison of CLP(FD) and ASP solutions to NP-complete problems. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 67–82. Springer, Heidelberg (2005)

[4] Dovier, A., Formisano, A., Pontelli, E.: An experimental comparison of constraint logic programming and answer set programming. In: Howe, A., Holt, R. (eds.) AAAI 2007: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, pp. 1622–1625. AAAI Press, Menlo Park (2007)

[5] Dovier, A., Formisano, A., Pontelli, E.: Multivalued action languages with constraints in CLP(FD). In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 255–270. Springer, Heidelberg (2007)

[6] Dovier, A., Formisano, A., Pontelli, E.: An empirical study of CLP and ASP solutions of combinatorial problems. Journal of Experimental & Theoretical Artificial Intelligence 21(2), 79–121 (2009)

[7] Dovier, A., Formisano, A., Pontelli, E.: Representing multi-agent planning in CLP. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 423–429. Springer, Heidelberg (2009)

[8] Dovier, A., Formisano, A., Pontelli, E.: An investigation of Multi-Agent Planning in CLP. Fundamenta Informaticae (2010) (to appear)

[9] Dovier, A., Formisano, A., Pontelli, E.: Autonomous agents coordination: Action description languages meet CLP(FD) and Linda. In: Proceedings of the 25th Italian Conference on Computational Logic. Workshop Proceedings, vol. 598, CEUR (2010)

[10] Dovier, A., Formisano, A., Pontelli, E.: Multivalued action languages with constraints in CLP(FD). Theory and Practice of Logic Programming 10(2), 167–235 (2010)

[11] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 260–265. Springer, Heidelberg (2007)

[12] Gebser, M., Schaub, T., Thiele, S.: GrinGo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)

[13] Gelfond, G., Baral, C., Pontelli, E., Tran, S.: Logic programmin for finding models in the logics of knowledge and its applications. Theory and Practice of Logic Programming 10(4-6), 675–690 (2010)

[14] Gelfond, M., Lifschitz, V.: Representing actions in extended logic programming. In: Joint International Conference and Symposium on Logic Programming, pp. 559–573. The MIT Press, Cambridge (1992)

[15] Gelfond, M., Lifschitz, V.: Action languages. Electronic Transactions on Artificial Intelligence 2, 193–210 (1998)

[16] Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming 19/20, 503–581 (1994)

[17] Lee, J., Lifschitz, V.: Describing additive fluents in action language C+. In: Gottlob, G., Walsh, T. (eds.) Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, IJCAI 2003, Acapulco, Mexico, August 9-15, pp. 1079–1084. Morgan Kaufmann, San Francisco (2003)

[18] Levesque, H., Pirri, F., Reiter, R.: GOLOG: a logic programming language for dynamic domains. Journal of Logic Programming 31(1-3), 59–83 (1997)

[19] Lifschitz, V.: Answer set planning. In: de Schreye, D. (ed.) Proc. of the 16th Intl. Conference on Logic Programming, pp. 23–37. MIT Press, Cambridge (1999)

[20] Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K., Marek, V., Truszczyński, M., Warren, D. (eds.) The Logic Programming Paradigm: A 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)

[21] McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press, Edinburgh (1969)

[22] Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4), 241–273 (1999)

[23] Son, T., Baral, C., McIlraith, S.A.: Planning with different forms of domain-dependent control knowledge - an answer set programming approach. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 226–239. Springer, Heidelberg (2001)

[24] Son, T., Pontelli, E., Sakama, C.: Logic programming for multiagent planning with negotiation. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 99–114. Springer, Heidelberg (2009)

[25] Subrahmanian, V.S., Zaniolo, C.: Relating stable models and ai planning domains. In: Sterling, L. (ed.) ICLP 1995: Proceedings of the Twelfth International Conference on Logic Programming, pp. 233–247. The MIT Press, Cambridge (1995)

[26] Thielscher, M.: Reasoning about actions with cHRs and finite domain constraints. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 70–84. Springer, Heidelberg (2002)

[27] Tran, S., Pontelli, E.: Planning with preferences using logic programming. Theory and Practice of Logic Programming 6(5), 559–607 (2006)

[28] Warren, D.: WARPLAN: A system for generating plans. Technical Report DCL Memo 76, University of Edinburgh (1974)

## A Appendix: Experimental Results

**Table 1.** An excerpt of the experimental results for the $\mathcal{B}$ encodings (timing in seconds)

| Instance | num.of fluents | num.of actions | plan length | gringo+clasp | $\mathcal{B}$-SICSplan |
|---|---|---|---|---|---|
| barrels-5-7-12 | 27 | 6 | 11 | 0.20+0.81 | 0.03+0.60+0.69 |
| barrels-7-9-16 | 35 | 6 | 15 | 0.40+8.47 | 0.05+1.75+2.98 |
| barrels-9-11-20 | 43 | 6 | 19 | 0.71+67.54 | 0.06+3.03+8.58 |
| barrels-11-13-24 | 51 | 6 | 23 | 1.17+183.32 | 0.55+0.55+21.05 |
| barrels-15-17-32 | 67 | 6 | 31 | 2.51+1871.43 | 0.17+14.25+79.16 |
| barrels-31-33-64 | 131 | 6 | 63 | T | M |
| barrels-63-65-128 | 259 | 6 | 127 | T | M |
| hanoi.16-50-6 | 42 | 33 | 34 | T | T |
| hanoi.16-54-6 | 42 | 33 | 38 | T | T |
| hanoi.16-56-6 | 42 | 33 | 40 | T | T |
| hanoi.20-50-7 | 52 | 42 | 30 | T | T |
| hanoi.20-55-7 | 52 | 42 | 35 | T | T |
| hanoi.21-53-6 | 42 | 33 | 32 | T | T |
| hanoi.27-7 | 52 | 42 | 27 | T | 0.67+3023.23 |
| hanoi.30-7 | 52 | 42 | 30 | T | 0.68+522.35 |
| hanoi.31-6 | 42 | 33 | 31 | T | 0.56+3062.04 |
| hanoi.32-6 | 42 | 33 | 32 | T | 1.32+3224.68 |
| hanoi.32-63-6 | 42 | 33 | 31 | T | 0.59+3421.79 |
| hanoi.32-7 | 52 | 42 | 32 | T | T |
| hanoi.33-6 | 42 | 33 | 33 | T | T |
| hanoi.34-6 | 42 | 33 | 34 | T | T |
| hanoi.35-6 | 42 | 33 | 35 | T | T |
| hanoi.36-6 | 42 | 33 | 36 | T | T |
| hanoi.36-7 | 52 | 42 | 36 | T | T |
| hanoi.37-6 | 42 | 33 | 37 | T | T |
| hanoi.37-7 | 52 | 42 | 37 | T | T |
| hanoi.38-6 | 42 | 33 | 38 | T | T |

**Table 2.** An excerpt of the experimental results for the $\mathcal{B}$ encodings (timing in seconds)

| Instance | num.of fluents | num.of actions | plan length | gringo+clasp | $\mathcal{B}$-SICSplan |
|---|---|---|---|---|---|
| hyd.cg21 | 24 | 13 | 3 | 0.01+0.01 | 0.01+0.01 |
| hyd.cg22 | 24 | 13 | 2 | 0.01+0.01 | 0.01+0.01 |
| hyd.cg23 | 24 | 13 | 2 | 0.01+0.01 | 0.01+0.01 |
| hyd.cg31 | 36 | 20 | 3 | 0.01+0.01 | 0.01+0.01 |
| hyd.cg32 | 36 | 20 | 3 | 0.01+0.01 | 0.01+0.01 |
| hyd.cg33 | 36 | 20 | 3 | 0.01+0.01 | 0.01+0.01 |
| hyd.cg61 | 104 | 64 | 16 | 0.07+0.07 | 0.34+0.06 |
| hyd.cg62 | 104 | 64 | 16 | 0.07+0.07 | 0.34+0.06 |
| hyd.cg63 | 104 | 64 | 16 | 0.07+0.07 | 0.31+0.06 |
| rev-fold-s-4-2 | 65 | 4 | 2 | 0.64+0.01 | 0.36+0.03 |
| rev-fold-s-9-4 | 325 | 14 | 4 | 82.70+11.63 | M |
| rev-fold-z-4-2 | 65 | 4 | 2 | 0.65+0.01 | 0.38+0.32 |
| rev-fold-z-5-3 | 101 | 6 | 3 | 2.62+0.01 | 2.47+0.36 |
| rev-fold-z-6-4 | 145 | 8 | 4 | 8.15+0.06 | 11.17+2.88 |
| rev-fold-z-8-6 | 257 | 12 | 6 | 49.00+19.88 | M |
| rev-fold-z-16-6 | 1025 | 28 | 6 | T | M |
| 15-puzzle-1 | 256 | 16 | 35 | 0.70+3.25 | T |
| 15-puzzle-2 | 256 | 16 | 35 | 0.69+0.49 | T |
| 15-puzzle-3 | 256 | 16 | 35 | 0.70+4.64 | T |
| 15-puzzle-4 | 256 | 16 | 35 | 0.70+8.26 | T |
| 15-puzzle-5 | 256 | 16 | 35 | 0.96+2.37 | T |
| 15-puzzle-6 | 256 | 16 | 36 | 0.72+5.63 | T |
| 15-puzzle-7 | 256 | 16 | 36 | 0.70+7.54 | T |
| 15-puzzle-8 | 256 | 16 | 36 | 0.70+1.18 | T |
| 15-puzzle-9 | 256 | 16 | 36 | 0.70+10.24 | T |
| 15-puzzle-10 | 256 | 16 | 40 | 0.78+8.08 | T |
| 15-puzzle-11 | 256 | 16 | 40 | 0.79+3.22 | T |
| 15-puzzle-12 | 256 | 16 | 40 | 0.80+0.99 | T |
| 15-puzzle-13 | 256 | 16 | 40 | 0.80+3.67 | T |
| 15-puzzle-14 | 256 | 16 | 40 | 0.76+0.96 | T |
| 15-puzzle-15 | 256 | 16 | 40 | 0.77+4.95 | T |
| tangram-1 | 135 | 578 | 7 | 1.10+0.07 | 20.49+0.52 |
| tangram-2 | 135 | 661 | 7 | 1.38+0.26 | 24.31+339.42 |
| tangram-3 | 135 | 744 | 7 | 1.56+1.55 | 29.08+273.72 |
| trucks-p01 | 99 | 324 | 13 | 0.31+0.51 | 0.03+1.26+9.06 |
| trucks-p02 | 128 | 420 | 17 | 0.49+14.82 | 0.03+2.37+635.67 |
| trucks-p03 | 205 | 939 | 20 | 1.24+1066.85 | T |
| trucks-p04 | 243 | 1116 | 23 | T | T |
| trucks-p05 | 347 | 2067 | 25 | T | T |

**Table 3.** An excerpt of the experimental results for the $\mathcal{B}$ encodings (timing in seconds)

| Instance | num.of fluents | num.of actions | plan length | gringo+clasp | $\mathcal{B}$-SICSplan |
|---|---|---|---|---|---|
| peg-asy-24 | 33 | 76 | 24 | 0.11+0.12 | 1.02+0.02 |
| peg-asy-25 | 33 | 76 | 25 | 0.12+2.33 | 1.02+0.02 |
| peg-asy-26 | 33 | 76 | 26 | 0.13+0.13 | 0.94+0.02 |
| peg-asy-27 | 33 | 76 | 27 | 0.13+1.87 | 0.95+0.03 |
| peg-asy-28 | 33 | 76 | 28 | 0.13+64.91 | 0.92+0.03 |
| peg-asy-29 | 33 | 76 | 29 | T | 0.81+3.37 |
| peg-asy-30 | 33 | 76 | 30 | T | 0.83+4.64 |
| peg-asy-31 | 33 | 76 | 31 | T | T |
| peg-asy-32 | 33 | 76 | 32 | T | T |
| peg-center-24 | 33 | 76 | 24 | 0.11+0.02 | 0.96+0.02 |
| peg-center-25 | 33 | 76 | 25 | 0.11+0.03 | 0.97+0.02 |
| peg-center-26 | 33 | 76 | 26 | 0.12+3.57 | 0.98+0.02 |
| peg-center-27 | 33 | 76 | 27 | 0.13+3.62 | 0.99+0.02 |
| peg-center-28 | 33 | 76 | 28 | 0.14+288.39 | 0.91+0.13 |
| peg-center-29 | 33 | 76 | 29 | 0.14+81.79 | 0.85+1.83 |
| peg-center-30 | 33 | 76 | 30 | T | T |
| peg-center-31 | 33 | 76 | 31 | T | 0.79+23.82 |
| peg-center-32 | 33 | 76 | 32 | T | T |
| peg-edge-24 | 33 | 76 | 24 | 0.11+0.11 | 1.02+0.02 |
| peg-edge-25 | 33 | 76 | 25 | 0.12+0.12 | 1.01+0.03 |
| peg-edge-26 | 33 | 76 | 26 | 0.13+0.21 | 1.09+0.03 |
| peg-edge-27 | 33 | 76 | 27 | 0.13+7,93 | 0.95+0.03 |
| peg-edge-28 | 33 | 76 | 28 | 0.13+181.88 | 0.98+0.95 |
| peg-edge-29 | 33 | 76 | 29 | 0.14+20.88 | 0.86+108.87 |
| peg-edge-30 | 33 | 76 | 30 | T | T |
| peg-edge-31 | 33 | 76 | 31 | T | 0.76+382.82 |
| peg-edge-32 | 33 | 76 | 32 | T | T |
| gas-i1a | 3433 | 24 | 9 | M | T |
| gas-i1b | 3433 | 24 | 9 | M | T |
| gas-i1c | 3433 | 24 | 9 | M | T |
| gas-i2a | 3433 | 24 | 13 | M | T |
| gas-i2b | 3433 | 24 | 13 | M | T |
| gas-i2c | 3433 | 24 | 13 | M | T |
| gas-i3a | 3433 | 24 | 15 | M | T |
| gas-i3b | 3433 | 24 | 15 | M | T |
| gas-i3c | 3433 | 24 | 15 | M | T |
| gas-i4a | 3433 | 24 | 9 | M | T |
| gas-i4b | 3433 | 24 | 9 | M | T |
| gas-i4c | 3433 | 24 | 9 | M | T |
| gas-i5e | 1123 | 24 | 11 | 45.18+257.38 | T |
| gas-i5f | 1123 | 24 | 11 | T | M |
| gas-i5g | 1123 | 24 | 11 | M | T |
| gas-i5h | 1123 | 24 | 11 | M | T |

**Table 4.** An excerpt of the experimental results for the $\mathcal{B}^{MV}$-encodings (timing in seconds)

| Instance | num.of fluents | num.of actions | plan length | $\mathcal{B}^{MV}$-SICSplan |
|---|---|---|---|---|
| rev-fold-s-4-2 | 9 | 4 | 2 | 0.07+0.01 |
| rev-fold-s-9-4 | 19 | 14 | 4 | 3.11+0.02 |
| rev-fold-z-4-2 | 9 | 4 | 2 | 0.06+0.01 |
| rev-fold-z-5-3 | 11 | 6 | 3 | 0.22+0.01 |
| rev-fold-z-6-4 | 13 | 8 | 4 | 0.55+0.04 |
| rev-fold-z-8-6 | 17 | 12 | 6 | 2.33+12.67 |
| rev-fold-z-16-6 | 33 | 28 | 6 | 48.37+24.94 |
| barrels-5-7-12 | 3 | 6 | 11 | 0.05+0.05 |
| barrels-7-9-16 | 3 | 6 | 15 | 0.06+0.10 |
| barrels-9-11-20 | 3 | 6 | 19 | 0.08+0.18 |
| barrels-11-13-24 | 3 | 6 | 23 | 0.12+0.27 |
| barrels-15-17-32 | 3 | 6 | 31 | 0.14+0.59 |
| barrels-31-33-64 | 3 | 6 | 63 | 0.45+3.19 |
| barrels-63-65-128 | 3 | 6 | 127 | 0.69+18.99 |
| gas-i1a | 23 | 24 | 9 | 0.14+166.21 |
| gas-i1b | 23 | 24 | 9 | 0.13+127.99 |
| gas-i1c | 23 | 24 | 9 | 0.13+273.04 |
| gas-i2a | 23 | 24 | 13 | T |
| gas-i2b | 23 | 24 | 13 | 0.16+679.95 |
| gas-i2c | 23 | 24 | 13 | 0.17+1654.44 |
| gas-i3a | 23 | 24 | 15 | T |
| gas-i3b | 23 | 24 | 15 | 0.25+0.94 |
| gas-i3c | 23 | 24 | 15 | 0.24+1.47 |
| gas-i4a | 23 | 24 | 9 | 0.15+163.43 |
| gas-i4b | 23 | 24 | 9 | 0.15+128.08 |
| gas-i4c | 23 | 24 | 9 | 0.14+270.96 |
| gas-i5e | 23 | 24 | 11 | 0.15+282.44 |
| gas-i5f | 23 | 24 | 11 | 0.15+17.92 |
| gas-i5g | 23 | 24 | 11 | 0.18+282.51 |
| gas-i5h | 23 | 24 | 11 | 0.14+284.53 |
| MV-hyd-5 | 21 | 28 | 5 | 0.30+0.01 |
| MV-hyd-9 | 20 | 40 | 9 | 0.97+2.89 |
| MV-hyd-12 | 21 | 41 | 12 | 1.46+102.02 |
| MV-hyd-14 | 30 | 45 | 14 | 2.91+7.24 |

# Refinement of History-Based Policies

Jorge Lobo[1], Jiefei Ma[2], Alessandra Russo[2], Emil Lupu[2]
Seraphin Calo[1], and Morris Sloman[2]

[1] IBM T.J. Watson Research Center,
New York, United States
[2] Department of Computing, Imperial College London,
United Kingdom

**Abstract.** We propose an efficient method to evaluate a large class of history-based policies written as logic programs. To achieve this, we dynamically compute, from a given policy set, a finite subset of the history required and sufficient to evaluate the policies. We maintain this history by monitoring rules and transform the policies into a non history-based form. We further formally prove that evaluating history-based policies can be reduced to an equivalent, but more efficient, evaluation of the non history-based policies together with the monitoring rules.

## 1 Introduction

The concept of policy is ubiquitous and it is frequently used to describe general rules that constrain the behavior of a system. For example, in computer systems we find policies that prescribe how often and what kind of data to back-up. In networks one can set policies to handle data traffic in different ways based on service agreements signed by the producers or consumers of the traffic. Good policy specification languages should be expressive yet succinct and amenable to efficient evaluation. Defining such languages is essentially a knowledge representation problem. Thus, logic programs [3,11,14] are frequently used as the basis for policy specification language. Logic programs are expressive, analysable and have well understood semantics.

It is often necessary to specify policies with historical conditions that refer to past system states or past policy decisions. For example, *"X is allowed to turn on the fan if the room temperature is above a threshold anytime within the past 5 minutes"*, or *"X is not allowed to enter room A if X was allowed to enter room B 30 minutes ago"*. We have developed a formal framework for the representation and analysis of authorization and obligation policies. Policies are represented in a limited but expressive logic progamming based language, which can capture complex dependencies amongst policy rules [2,6], including historical conditions.

For many policies, especially access control policies, the speed of evaluation is crucial, as it is performed upon every access request, thereby increasing the need of quick system response time. Policy evaluation can be interpreted as the process of answering a query posed to a small logic program representing the policy rules and the facts describing the current system state or the system history. However, storing a full history is impractical since the size of the history dominates both the size and time complexities of policy evalution; hence many existing policy enforcement systems (such as PONDER2 [15], CIM-SPL [1] and XACML [16]) limit their evaluations to single state conditions and leave the managment of the history to the application.

One goal in our framework is to let policy administrators write policies in high-level languages, possibly constrained natural language, and provide automatic or semiautomatic tools to refine high-level policies into policies that can be enforced by existing policy systems. The general problem of policy refinement is complex [5] but in this paper we would like to address the refinement of policies with historical conditions into policies that will be enforced by policy systems that do not support history-based evaluations. We use a technique originally proposed by Chomicki [4] for the management of temporal integrity constraints in relational databases and later on used by Gabaldon [7] and Gonzales *et al.* [10] to handle temporal conditions in reasoning about actions to do the refinement. In the same spirit of those works, we propose a *policy transformation method*, which transforms a set of history-based policies to an equivalent set of history-free policies that can be evaluated efficiently by existing systems. By syntactically analysing a policy set, we identify a *subset of the system history* sufficient to evaluate the conditions. Then, a set of monitoring rules is automatically generated to monitor this information (as current conditions), and record (and maintain) it in a store of *auxiliary facts*. Policies are then transformed into their history-free form where simple queries to the store replace the original historical conditions. Both the monitoring rules and the history-free policies are represented as logic programs which provide us with a formal framework for our proofs of complexity and correctness. Although the paper focuses on authorisation policies, the method presented can also be applied to other general policies (see [6]).

The paper is organised as follows. We first discuss related work and describe the policy language. We then present the generic transformation method and prove its correctness. Finally, we discuss the evaluation complexity, extensions of the method, review a reference implementation of the basic transformations and present some concluding remarks.

## 2 Related Work

Many studies describe policies using rule languages. See, for example, [3] and [13] for policy languages for the Web, and [9], [12] and [11] for languages based on logic programs. Although many aim to provide efficient implementations, none addresses the topic of temporal conditions.

Within artificial intelligence and database communities several approaches have been proposed for handling temporal constraints. The work in [7] proposes an extension of the Situation Calculus to encode Past Linear Temporal Logic (PLTL) connectives and provides a theorem to perform regression over the temporal operators. The work in [10] has shown how to extend the action language

$\mathcal{A}$ with PLTL connectives and use answer set computation for evaluating conditions. These studies as well as our earlier work in [6] are motivated by the need to reason about actions and their consequences and less by policy evaluation. Most related to our work is the approach proposed in [4], for efficient checking of temporal integrity constraints in DBMS. In this work, PLTL constraints are reduced to conditions depending only on the current and previous state, as historical information is re-computed inductively at each state and updated immediately. Our policy transformation method has been inspired by this work but differs in two ways. It deals with historical conditions expressed in first-order logic policy specification languages and, most importantly, it does not require immediate updates of the historical conditions - historical information can be updated with delays depending on the conditions, instead of at each state, so gaining in performance.

Another related topic is event correlation systems. Event correlation has been studied for long time (see e.g. [17]). Our approach can be seen as transforming temporal or historical conditions into event evaluations so that we can monitor events to update our working storage.

## 3 Policy Specification Language

*Basic language:* Craven et al.'s [6] policy language $\mathcal{L}$ is a many-sorted logic based language for the specification of security policies. We will use a subset of $\mathcal{L}$ to describe our approach. An authorisation policy in $\mathcal{L}$ is a rule of the form:

$$[permitted/denied](Sub, Tar, Act, T) \leftarrow H_1, \ldots, H_m, C_1, \ldots, C_n$$

where $Sub$, $Tar$ and $Act$ represent the *subject*, *target* and *action* regulated by the policy and are of predefined sorts $Subject$, $Target$ and $Action$ respectively – standard components of access control policy specifications (e.g. see [16]). $T$ is of the sort $Time$ – the non-negative real numbers and refers to the time when the policy is evaluated. Each $C_i$ is a *time constraint* – an expression of the form $T_1 \oplus T_2 \pm c$, where $T_1$ and $T_2$ are different time variables, $c$ is a non-negative real number and $\oplus$ is one of $\{=, <, \leq\}$. Each $H_i$ is a policy *condition*, and is either a positive literal (an atom) or a negative literal (an atom preceded with the *negation as failure* `not`) with one of the following predicates: $req/4$, $do/4$, $deny/4$, $permitted/4$, $denied/4$, with arguments $\langle Sub, Tar, Act, T\rangle$, $holds(F, T)$ and $happens(E, T)$. An instance $do(sub, tar, act, t)$ ($deny(sub, tar, act, t)$) means that subject $sub$ applied (was denied the application of) action $act$ on target $tar$ at time $t$. This $t$ must always be before the time $T$ when the policy is evaluated and refers to previous authorisation decisions (*do*: the request was permitted, *deny*: the request was denied). The instances of *permitted* and *denied* are self-explanatory. $holds(f, t)$ intuitively says that a property (or *fluent*) $f$ of the system holds at time $t$, and $happens(e, t)$ denotes an exogenous *event* $e$ occurring at time $t$ (these are events not regulated by the policies thus different from *do* and *deny*). An instance $req(sub, tar, act, t)$ denotes a special non-regulated exogenous event, a request by $sub$ to perform $act$ on $tar$ occurring at time $t$

(an authorization decision is always the result of a request). In addition, any time variable appearing in the time constraints must be the time argument of a condition or of the head of the rule. Conditions with the same time argument $T$ as in the head of the rule must not be *do* or *deny* predicates, and for any other time argument $T_i$ in a condition, it must be the case that $C_1, \ldots, C_n \models T_i \leq T$. This ensures that authorisations do not depend on "future" properties. We refer to conditions with the same time variable as that in the head of a policy rule as *current conditions*, and any other condition as *historical conditions.* The policy set is assumed to be locally stratified. The following are examples of policies:

*Example 1.* "If a file was declassified at least 3 days ago, then it is allowed to be deleted now."

$$\begin{aligned}&permitted(X, file(Y), delete, T) \leftarrow\\&\quad do(Z, file(Y), declassify, T_1), T_1 \leq T - 3days.\end{aligned}$$

*Example 2.* "If a node has already broadcasted one message within the past 2 seconds, then it is not allowed to broadcast any message now."

$$\begin{aligned}&denied(node(X), message(Y), broadcast, T) \leftarrow\\&\quad do(node(X), message(Z), broadcast, T_1),\\&\quad T - 2seconds \leq T_1, T_1 \leq T.\end{aligned}$$

*Durative historical conditions.* Historical conditions may require two time variables to express conditions with duration. These are temporal conditions, typical of PLTL, of the form: *a condition must at some point (or always) hold (or not hold) between two time points.* To express these notions, we need to extend the policy language with "duration" predicates that can appear as policy conditions among the $H_i$ literals and assume additionally seven pairs of domain independent axioms of the form:

$$\begin{aligned}&\mathcal{R}Between(\overrightarrow{X}, T_1, T) \leftarrow \mathcal{R}(\overrightarrow{X}, T_2), T_1 \leq T_2, T_2 \leq T.\\&un\mathcal{R}Between(\overrightarrow{X}, T_1, T) \leftarrow \texttt{not}\ \mathcal{R}(\overrightarrow{X}, T_2), T_1 \leq T_2, T_2 \leq T.\end{aligned}$$

where $\mathcal{R}$ is replaced with one of *do*, *deny*, *req*, *permitted*, *denied*, *holds* or *happens* with the appropriate arguments. Then we can express policies such as:

*Example 3.* "A server can start data transfer if the client has authenticated itself after the request was made."

$$\begin{aligned}&permitted(X, Y, transfer, T) \leftarrow\\&\quad do(Y, X, request, T_1), T_1 \leq T,\\&\quad doBetween(Y, X, auth, T_1, T).\end{aligned}$$

*Example 4.* "A client is allowed to connect to a server if the server has always been in subnet A since its start-up."

$$\begin{aligned}&permitted(X, Y, connect, T) \leftarrow\\&\quad do(admin, Y, startup, T_1), T_1 \leq T,\\&\quad \texttt{not}\ unholdsBetween(in(Y, subnetA), T_1, T).\end{aligned}$$

Note that Example 4 shows a historical condition equivalent to one using the *since* operator in PLTL.

### 3.1 Semantics

The evaluation of a policy depends on system states, where each state represents a system "snapshot" and transitions between states are marked by occurrences of a *req* or a *happens* event. We assume authorisation policies define *permitted* and *denied*, as, in practice, both are used by administrators and many policy languages. The system should execute *do*/*deny* according to the evaluation of domain-independent axioms, also defined in $\mathcal{L}$, that solve conflicts between *permitted* and *denied*. For example, axioms could look like:

$$\begin{aligned}
&do(S, Ta, A, T) \leftarrow \\
&\qquad req(S, Ta, A, T), permitted(S, Ta, A, T), \\
&\qquad \texttt{not}\ denied(S, Ta, A, T). \\
&deny(S, Ta, A, T) \leftarrow \\
&\qquad req(S, Ta, A, T), denied(S, Ta, A, T).
\end{aligned}$$

See [6] for more details on conflict resolution axioms expresible in $\mathcal{L}$.

We consider a function *clock* that maps a system state to a time value, i.e. $clock(S) = T$. Using it we formally define the notions of system state and *system trace*.

**Definition 1 (System State).** *A system state $S$, with associated time $t = clock(S)$, is a set of ground facts from $\mathcal{L}$ such that:*

- *$req(\overrightarrow{x}, t) \in S$ iff a request of $\overrightarrow{x}$ takes place at $t$;*
- *$[do/deny](\overrightarrow{x}, t) \in S$ iff the action of $\overrightarrow{x}$ is executed/denied at $t$;*
- *$happens(e, t) \in S$ iff an event $e$ occurs at $t$;*
- *$holds(f, t) \in S$ iff the system property $f$ holds at $t$*

**Definition 2 (System Trace).** *A system trace $\mathcal{T}$ is a (possibly infinite but countable) sequence of states $\langle S_0, S_1, \ldots \rangle$, where $S_0$ denotes the initial state. The* meaning *of $\mathcal{T}$, denoted as $\mathcal{M}(\mathcal{T})$, is the union of the facts of all the states the trace contains ($\bigcup_i S_i$).*

Intuitively, a trace is a history of the system execution. The laws governing how *holds* changes over time in a trace is not of interest here. In a typical policy evaluation system these values can be obtained from multiple sources (e.g. a system database or monitoring instruments such as a card or fingerprint reader). In our discussion we assume that they form a regular extensional database that is updated appropriately and can be accessed at any time. Let $T_i$ and $T_j$ be the time for states $S_i$ and $S_j$ in a system trace, then $T_i < T_j$ iff $S_i$ precedes $S_j$.

**Definition 3 (Policy Satisfaction).** *Let $\mathcal{P}$ be a set of locally stratified policies together with the domain independent axioms and $\mathcal{T}$ be a system trace, let $R$ be a ground instance of a rule in $\mathcal{P}$ and $r(\overrightarrow{x}, t)$ be the head of $r$. We say $R$* is supported *by $\mathcal{T}$ if and only if $\mathcal{P} \cup \mathcal{M}(\mathcal{T}) \models_{sm} r(\overrightarrow{x}, t')$ where $t'$ is the biggest time associated to a state in $\mathcal{T}$ such that $t' \leq t$, and $\models_{sm}$ is the logical entailment under the* Stable Model *semantics[8].*

This definition tells us, for example, how to build a monitoring system that detects policy violations by sending an alert as soon as a *do* or a *deny* that is not supported is detected in a trace. An auditing or policy compliance system can be built by checking whether all *do*'s and *deny*'s in the logs of traces are supported. Our interest, however, is to provide the system with an enforcement mechanism such that only traces that do not violate policies are generated. Such a mechanism must intersect every *req* in **real time** and only let *do*'s and *deny*'s that obey the policies be executed.

## 4 Policy Transformation

To evaluate a policy with historical conditions at time $t$, it is sufficient to record only the portion of the trace that is relevant to the specific historical conditions. The challenge is thus to identify *what to record*, *how to record it efficiently* and *when to record it or remove it once it is no longer needed.*

We associate the truth value of each historical condition instance with an *auxiliary fact*. Each fact is tagged with a *counter* and is maintained in a store (or database) by two operations: *assertion* which inserts the fact with its counter set to 1 if it was not in the store, or increments the fact's counter otherwise, and *retraction* which decrements the fact's counter if it is greater than 1, or removes the fact otherwise. Syntactically, each historical condition has two parts: the *condition part* (i.e. a predicate with a time variable) and the *constraint part* (i.e. the set of temporal constraints describing how the time variable relates to the evaluation time). The condition part can be evaluated at any state, but the changes of its value affect future evaluation of the whole historical condition only when the time span satisfies the constraint part. Therefore, by syntactically analysing a given policy, we can identify the conditions of interest, and generate a set of *monitoring rules*[1], which force the system to evaluate the conditions pro-actively at each state and to update the *auxiliary store* at the appropriate future state(s) depending on the evaluation results. The original history-based policy can then be transformed into a rule in which all the historical conditions are reduced to auxiliary store queries.

A policy may contain multiple historical conditions in the body, which can always be reduced to the combination of basic pair-wise temporal relations of the conditions and the head. There are four basic relations – the first three cover any arbitrary set of constraints (of the form $T_1 \oplus T_2 \pm c$) involving two time variables, and the last one is the durative condition *since*. In the rest of this section, we first describe how to handle these basic relations. Then, due to space limitations, we only sketch a general algorithm for transforming policies with multiple historical conditions for which the time constraints form a tree-like hierarchy.

### 4.1 Basic Historical Conditions

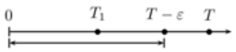
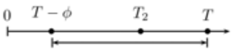
For simplicity, we use $p(\overrightarrow{X}, T)$ and $h(\overrightarrow{X}, T)$ as shorthands for a policy's head and condition respectively. When the context is clear, we may drop $\overrightarrow{X}$ and denote

[1] They can be implemented as Event-Condition-Action rules.

them as $p(T)$ and $h(T)$. We use $h_1$, $h_2$, etc. to denote different conditions. Hence, a policy with a single historical condition has the simplified form:

$$p(T) \leftarrow h(T'), c(T', T).$$

where $c(T', T)$ is a relation denoting the set of constraints over time variables $T'$ and $T$. Two basic relations are:

| | |
|---|---|
| **Case 1:** let $\varepsilon$ be a non-negative constant and $T$ be the current time, the condition must hold any time before $T - \varepsilon$ ago.  $p_1(T) \leftarrow h_1(T_1), T_1 \leq T - \varepsilon.(\varepsilon \geq 0)$ | **Case 2:** let $\phi$ be a positive constant and $T$ be the current time, the condition must hold any time within $T - \phi$ ago.  $p_2(T) \leftarrow h_2(T_2), T - \phi \leq T_2, T_2 \leq T.(\phi > 0)$ |

Examples 1 and 2 are instances of these two cases. In Case 1, as long as the condition holds from time $t$, the policy should be evaluated to true from $t + \varepsilon$ onwards, regardless of the truth value of the condition after $t$. However, in Case 2 if the condition holds from $t$ and ceases to hold at $t'$, the policy evaluates to true only from $t$ to $t' + \phi$. Thus, let $h'_1(T)$ and $h'_2(T)$ be the transformed current conditions (i.e. auxiliary store queries) replacing the historical conditions in Cases 1 and 2 respectively, and $\texttt{h}'_1$ and $\texttt{h}'_2$ be their associated auxiliary facts (i.e. whose presence in store implies the truth value of the transformed conditions), then the transformed policies and generated monitoring rules are:

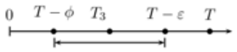
| | |
|---|---|
| $p_1(T) \leftarrow h'_1(T).$ | **on** $h_1$ changes value at $T$<br>**if** $h_1$ is true<br>**do** assert $\texttt{h}'_1$ at $T + \varepsilon$ |

| | |
|---|---|
| $p_2(T) \leftarrow h'_2(T).$ | **on** $h_2$ changes value at $T$<br>**if** $h_2$ becomes true<br>**do** assert $\texttt{h}'_2$ at $T$ |
| | **on** $h_2$ changes value at $T$<br>**if** $h_2$ becomes false<br>**do** retract $\texttt{h}'_2$ at $T + \phi$ |

A single negative historical condition can be handled similarly. However, the condition will be replaced with a positive query and the generated monitoring rules will monitor the negated value of the condition instead. For example, the policy $p(T) \leftarrow$ `not` $h(T'), T' \leq T - \varepsilon$ can be rewritten as $p(T) \leftarrow nh(T'), T' \leq T - \varepsilon$. The transformed policy will be $p(T) \leftarrow nh'(T)$ and the monitoring rule will be "**on** $h$ changes at $T$, **if** $h$ is *false*, **do** assert $\texttt{nh}'$ at $T + \varepsilon$".

Combining time relations in Cases 1 and 2 gives Case 3:

**Case 3:** let $\varepsilon$ and $\phi$ be positive constants such that $\phi \geq \varepsilon$ and $T$ be the current time, the condition must hold sometime within the past period $T - \phi$ and $T - \varepsilon$.



$$p_3(T) \leftarrow h_3(T_3), T - \phi \leq T_3, T_3 \leq T - \varepsilon.(\phi \geq \varepsilon > 0)$$

The transformation for Case 3 is the same as for Case 2 except that the first monitoring rule is replaced by the one generated for Case 1. One can think of $\phi$ as infinite for Case 1, thus, no need for the second rule. Note that policies with pattern $p(T) \leftarrow h(T'), T' = T - c$, are special cases of Case 3 where $\phi = \varepsilon = c$ and $c > 0$.

**Store Update Dependencies.** Without the auxiliary fact counters, interleaving *assert* and *retract* operations triggered by a pair of monitoring rules may have unintended effects. For example, in Case 3, if $\phi$ is sufficiently larger than $\varepsilon$, and $h_3$ becomes true, false and true again at $t$, $t + c_1$, $t + c_2$ respectively, for a $c_2 > c_1$ and $c_2 + \varepsilon < c_1 + \phi$, then, the assertion rule triggers twice (at $t$ and $t + c_2$) and the retraction rule would trigger between them (at $t + c_1$). Because of the delay between the store updates and the triggering of monitoring rules, the actual assertions occur at $t + \varepsilon$ and $t + c_2 + \varepsilon$, whilst the retraction occurs after them at $t + c_1 + \phi$. In theory, the retraction should be "cancelled" by the first "assertion" and hence the fact should remain after $t + c_1 + \phi$. Thus, counters are necessary for matching *assert/retract* pairs of store updates.

**Serial and Parallel Compositions.** Let $T$ and $T'$ be two time variables and let $T' \prec T$ denote a set of time constraints between $T'$ and $T$ that matches one of Cases 1–3. Then, conditions associated with $T'$ are historical w.r.t. those associated with $T$. If $h_1(\overrightarrow{X}_1, T_1)$ denotes a policy condition (or head), and $h_2(\overrightarrow{X}_2, T_2)$ and $h_3(\overrightarrow{X}_3, T_3)$ denote two historical conditions w.r.t. $h_1(\overrightarrow{X}_1, T_1)$, then, the two historical conditions can be combined either: in a *serial composition*, such that $T_3 \prec T_2 \wedge T_2 \prec T_1$, or in a *parallel composition*, such that $T_3 \prec T_1 \wedge T_2 \prec T_1$. For serial composition, we first consider $h_2(\overrightarrow{X}_2, T_2), h_3(\overrightarrow{X}_3, T_3), T_3 \prec T_2$ as a single condition *combined_h*$_2(\overrightarrow{X}_2, \overrightarrow{X}_3, T_2)$ and transform it w.r.t. $h1(\overrightarrow{X}_1, T1)$ according to $T_2 \prec T_1$. Then we transform $h_3(\overrightarrow{X}_3, T_3)$ w.r.t. $h_2(\overrightarrow{X}_2, T_2)$ according to $T_3 \prec T_2$. For parallel composition, we can transform $h_2(\overrightarrow{X}_2, T_2)$ and $h_3(\overrightarrow{X}_3, T_3)$ independently w.r.t. $h_1(\overrightarrow{X}_1, T_1)$ according to $T_2 \prec T_1$ and $T_3 \prec T_1$ respectively. For the serial transformation to work we need to limit any time variable to appear in the left hand side of a single time constraint (note that $T_j - \phi \leq T_i, T_i \leq T_j - \varepsilon$

is considered a single time constraint) other than $T_i \leq T$. With this restriction the relation $\prec$ forms a tree over the time variables as nodes rooted at $T$, the time variable occuring in the head of the rule.

**Undetermined Composition: $T'' - c \leq T'$.** Consider the following policy where $c > 0$,

$$p(T) \leftarrow h_1(T_1), h_2(T_2), T_1 \leq T, T_2 \leq T, T_1 - c \leq T_2.$$

Neither condition $h_1$ nor condition $h_2$ is historical to the other, but they are not independent to each other due to $T_1 - c \leq T_2$ (equivalently $T_1 \leq T_2 + c$). Nevertheless the policy can be decomposed into two rules:

$$\begin{aligned} &p(T) \leftarrow h_1(T_1), h_2(T_2), T_2 \leq T, T_1 \leq T_2. \\ &p(T) \leftarrow h_1(T_1), h_2(T_2), T_1 \leq T, T_1 - c \leq T_2, T_2 \leq T_1. \end{aligned}$$

and then transform them separately.

Let 1 be the smallest time interval in the implementation system, then $T_1 < T_2$ can be defined as $T_1 \leq T_2 - 1$ (this is defined by the granularity of the clock used). $>$ and $\geq$ can be defined by $<$ and $\leq$ respectively with the two sides swapped. Therefore, our approach presented so far can handle policies with up to two (non-durative) historical conditions with temporal constraints of the form $T_1 \oplus T_2 \pm c$.

## 4.2 Durative Historical Condition

If a durative predicate appears as a positive condition in a policy, it can always be replaced by its definition. For instance, the policy in Example 3 can be expanded as

$$\begin{aligned} &permitted(X, Y, transfer, T) \leftarrow \\ &\qquad do(Y, X, request, T_1), T_1 \leq T, do(Y, X, auth, T_2), \\ &\qquad T_1 \leq T_2, T_2 \leq T. \end{aligned}$$

The new policy has two historical conditions forming the serial composition, and hence can be transformed.

However, when a duration predicate appears as a negative condition in a policy (e.g. of Example 4), we cannot replace the predicate with its definition because of the presence of `not` in the condition. This results in the basic case of *since*:

**Case 4:** Condition $h2$ *since* condition $h1$.

$$p(T) \leftarrow h_1(T_1), T_1 \leq T, \texttt{not}\ un\_h_2\_Between(T_1, T).$$

Consider the following situations:

– if $h_1$ becomes true at $t$, and if $h_2$ is true, then $h_2$ `since` $h_1$ should be true at $t$. If $h_2$ remains true, then so does $h_2$ `since` $h_1$.

– if $h_2$ becomes true at $t$, and if $h_1$ is true, then $h_2$ `since` $h_1$ should be true at $t$ too. If $h_2$ remains true, then so does $h_2$ `since` $h_1$.
– if $h_1$ becomes false at $t$, it has no effect to the value of $h_2$ `since` $h_1$; if $h_2$ becomes false at $t$, it will change the truth value of $h_2$ `since` $h_1$ at $t$.

Thus, in order to evaluate $p(T)$, both $h_1$ and $h_2$ must be monitored, but only three monitoring rules are required. Let $h'(T)$ be the transformed current condition for $h_1(T_1), T_1 \leq T,$ `not` $not\_h_2\_Between(T_1, T)$, and `h'` be its auxiliary fact:

| | |
|---|---|
| $p(T) \leftarrow h'(T).$ | **on** $h_1$ changes at $T$<br>**if** $h_1$ is true, $h_2$ is true<br>**do** insert `h'` at $T$ |
| | **on** $h_2$ changes at $T$<br>**if** $h_2$ is true, $h_1$ is true<br>**do** insert `h'` at $T$ |
| | **on** $h_2$ changes at $T$<br>**if** $h_2$ is false<br>**do** retractall* `h'` at $T$ |

(* *retractall* removes `h'` regardless its counter)

The case for `not` $h_2\_Between/2$ can be handled similarly (i.e. the negated value of $h_2$ is used in the conditions of the monitoring rules).

## 4.3 Transformation of Multiple Conditions

We first use a flow chart (Figure 1) to illustrate the general transformation algorithm steps, and then use an example (Example 5) to walk though the steps.

The general algorithm has three main steps. (1) In the *pre-processing phase*, if the given policy contains undetermined composition conditions, then it is rewritten to an equivalent set of policies without such conditions. These new policies can be handled separately. (2) In the *compilation phase*, each policy is syntactically analysed, and a Directed Connected Graph (DCG) representing the policy conditions and their relations is *constructed*. Each node of the graph is a condition, and each directed arc is the temporal relation (i.e. matching one of Cases 1–4) between the two connecting conditions. Note that if the graph is not *connected*, at least one condition is neither a current nor a historical condition, and hence the policy is malformed. If the graph is not *acyclic*, then at least two conditions are historical to each other, and hence the policy will always be evaluated to false and can be removed. The algorithm is limited to policies whose DCGs are trees. (This limitation will be discussed in Section 6). Finally, the tree is *topologically sorted*, and a set of policies that do not contain serial composition historical conditions will be generated by recursively traversing the tree. This process will be illustrated in Example 5. (3) In the last *transformation phase*, policies generated from the tree will be transformed according to the basic cases. Thus, the final output of the algorithm will be a set of history-free policies with the corresponding monitoring rules.

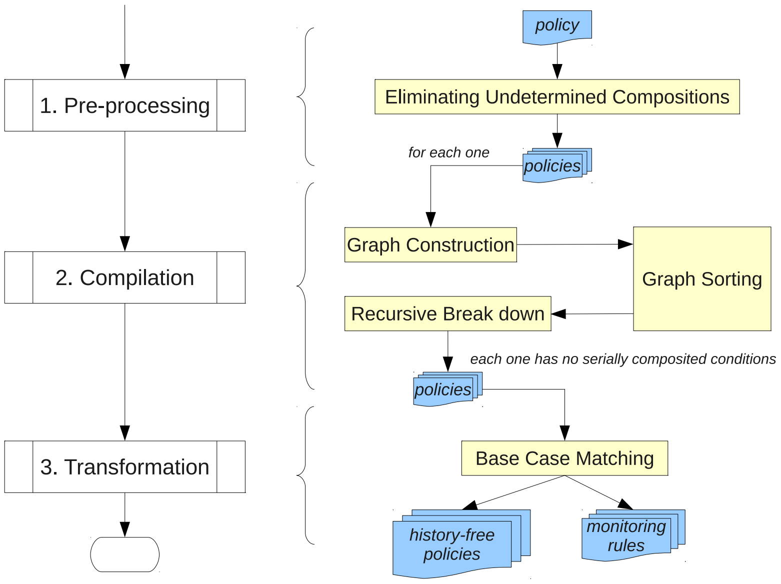


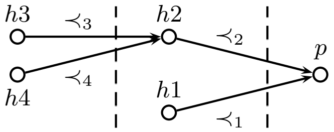
**Fig. 1.** General Transformation Steps

*Example 5* A policy with multiple historical conditions:

$$p(T) \leftarrow h_1(T_1), h_2(T_2), h_3(T_3), h_4(T_4),$$
$$T_1 \prec^1 T, T_2 \prec^2 T, T_3 \prec^3 T_2, T_4 \prec^4 T_2.$$

*[Transformation Steps:]*

1. Without loss of generality, we may assume there is no undetermined composition of the conditions. Thus, the policy's tree is constructed and sorted:



2. As we can see, the original policy has serial composition conditions (e.g. $T_3 \prec_3 T_2 \prec_2 T$ and $T_4 \prec_4 T_2 \prec_2 T$), and hence needs to be compiled. We do so by traversing the tree starting from the root node (i.e. $p$), and recursively generate new policies:
   (a) There are two arcs connecting $h_1$ and $h_2$ to $p$. The sub-tree starting from $h_2$ and the sub-tree containing only $h_1$ represent two parallel historical conditions to $p$ (one being composite and the other being single). The following policies are generated:

$$p(T) \leftarrow$$
$$h_1(T_1), T_1 \prec^1 T, comb\_h_2(T_2), T_2 \prec^2 T.$$
$$comb\_h_2(T) \leftarrow$$
$$h_2(T), h_4(T_4), T_4 \prec^4 T, h_3(T_3), T_3 \prec^3 T.$$

(b) Now none of the policies contains serial composition conditions. They can be transformed according to the basic Cases 1–4 and the parallel composition pattern, and the monitoring rules are generated. For example, the final history-free policies are:

$$\begin{aligned}&p(T) \leftarrow \\ &\qquad h'_1(T), comb\_h'_2(T). \\ &comb\_h_2(T) \leftarrow \\ &\qquad h_2(T), h'_4(T), h'_3(T).\end{aligned}$$

where $h'_1$, $comb\_h2'$, $h'_3$ and $h'_4$ are auxiliary store checking conditions.

## 5 Correctness of the Transformation

To prove the correctness theorem, we use the notion of a *synchronised system trace* as a mathematical tool for **simplying** the proof. A synchronised system trace synchronises an independent time line structure with a system trace. The synchronised trace is given by a system clock which generates pulses causing transitions between consecutive states. No property change (including the store) can occur between two consecutive pulses and thus the interval between two pulses is the smallest time unit (interval). A synchronised trace is defined as:

**Definition 4 (Synchronised System Trace).** *Let $\mathcal{T}$ be a system trace and $\omega$ the smallest time unit. A synchronised system trace $\mathcal{T}^{syn}$ of $\mathcal{T}$ is a sequence of states such that:*

1. *$\mathcal{T}^{syn}$ contains all the states of $\mathcal{T}$*
2. *for consecutive states $S_i$ and $S_{i+1}$ with associated time $T_i$ and $T_{i+1}$ in $\mathcal{T}$, $\mathcal{T}^{syn}$ contains extra $(T_{i+1} - T_i)/\omega - 1$ states, all of which are identical to $S_i$ except that their associated times are $T_i + \omega, T_i + 2\omega, \ldots, T_{i+1} - \omega$.*

Numbers appearing in time constraints without explicit units (e.g. *mins*, *seconds*) represent multiples of $\omega$ e.g., $T + 1$ is the smallest time after $T$. The meaning of a synchronised system trace extends the meaning of its corresponding system trace with all the ground facts for the extra states. In fact, they contain the same information, and one can be computed from the other.

To reason about the satisfaction of transformed policies, we also need to model the store and its update operations as logical rules. A monitoring rule of the form:

**on** $h_j(\overrightarrow{X}_j)$ changes value at $T$
**if** $h_1(\overrightarrow{X}_1) \wedge \cdots \wedge h_n(\overrightarrow{X}_n)$ becomes *true* $(1 \leq j \leq n)$
**do** assert $\mathtt{h}'(\overrightarrow{X})$ at $T + \varepsilon$

can be modelled as a logic formula

$$\begin{aligned}&assert(\mathtt{h}'(\overrightarrow{X}), T + \varepsilon) \leftarrow \\ &\qquad \mathtt{not}\ (h_1(\overrightarrow{X}_1, T-1) \wedge \cdots \wedge h_n(\overrightarrow{X}_n, T-1)) \wedge \\ &\qquad h_1(\overrightarrow{X}_1, T) \wedge \cdots \wedge h_n(\overrightarrow{X}_n, T).\end{aligned}$$

Note that when $T = 0$ (i.e. system start-up), $\texttt{not}\ (h_1(\overrightarrow{X}_1, -1) \wedge \cdots \wedge h_n(\overrightarrow{X}_n, -1))$ will be vacuously true. Hence, if $h_1(\overrightarrow{X}_1, 0) \wedge \cdots \wedge h_n(\overrightarrow{X}_n, 0)$ is true then so is $assert(\mathtt{h}'(\overrightarrow{X}), \varepsilon)$. This is important as rules will reflect the correct auxiliary store initialisation at system start-up.

A monitoring rule of the form:

| | |
|---|---|
| **when** | $h_j(\overrightarrow{X}_j)$ changes value at $T$ |
| **if** | $h_1(\overrightarrow{X}_1) \wedge \cdots \wedge h_n(\overrightarrow{X}_n)$ becomes *false* $(1 \leq j \leq n)$ |
| **then** | retract $\mathtt{h}'(\overrightarrow{X})$ at $T + \varepsilon$ |

can be modelled as a logic formula

$$\begin{aligned} &retract(\mathtt{h}'(\overrightarrow{X}), T + \varepsilon) \leftarrow \\ &\qquad h_1(\overrightarrow{X}_1, T-1) \wedge \cdots \wedge h_n(\overrightarrow{X}_n, T-1) \wedge \\ &\qquad \texttt{not}\ (h_1(\overrightarrow{X}_1, T) \wedge \cdots \wedge h_n(\overrightarrow{X}_n, T)). \end{aligned}$$

Finally, the effects of store update operations can be modelled with the following axioms:

$$\begin{aligned} &instore(\mathtt{h}'(\overrightarrow{X}), T) \leftarrow assert(\mathtt{h}'(\overrightarrow{X}), T1), T1 \leq T, \\ &\qquad \texttt{not}\ retracted(\mathtt{h}'(\overrightarrow{X}), T1, T). \\ &retracted(\mathtt{h}'(\overrightarrow{X}), T1, T) \leftarrow \\ &\qquad retract(\mathtt{h}'(\overrightarrow{X}), T2), T1 \leq T2 < T. \\ &h'(\overrightarrow{X}, T) \leftarrow instore(\mathtt{h}'(\overrightarrow{X}), T). \end{aligned}$$

where the last rule links the evaluation of a transformed condition $h'(\overrightarrow{X})$ and its corresponding auxiliary fact $\mathtt{h}'(\overrightarrow{X})$. Due to space limitations, we assume that no conflicting store updates may occur, to simplify the presentation of the proof sketch. In the full proof, the counter for each auxiliary fact in store is modelled as an extra argument of the *instore*/2 predicate, and the rules governing store update operations are adjusted accordingly. The main reasoning steps remain the same but the counter needs to be considered when reasoning about the effects of store updates.

**Theorem 1 (Correctness of Policy Transformation).** *Given a set of* locally stratified *history-based policies $\mathcal{P}$ plus the domain independent axioms and a system trace $\mathcal{T}$, let $\mathcal{P}'$ be the set of transformed non history-based policies, $\mathcal{T}^{syn}$ be the synchronised system trace, ECA be the set of generated monitoring rules and ST be effect axioms of auxiliary store updates. If $p(\overrightarrow{x}, t)$ is the head of a ground instance of a policy rule in $\mathcal{P}$ and in $\mathcal{P}'$, then*

$$\begin{aligned} \mathcal{P} \cup \mathcal{M}(\mathcal{T}) \models_{sm} p(\overrightarrow{x}, t) &\Leftrightarrow \mathcal{P} \cup \mathcal{M}(\mathcal{T}^{syn}) \models_{sm} p(\overrightarrow{x}, t) \\ &\Leftrightarrow \mathcal{P}' \cup ECA \cup ST \cup \mathcal{M}(\mathcal{T}^{syn}) \models_{sm} p(\overrightarrow{x}, t) \end{aligned}$$

*Proof Sketch:* we only outline the main steps here. The logic programs $\mathcal{P} \cup \mathcal{M}(\mathcal{T})$, $\mathcal{P} \cup \mathcal{M}(\mathcal{T}^{syn})$ and $\mathcal{P}' \cup ECA \cup ST \cup \mathcal{M}(\mathcal{T}^{syn})$ are all locally stratified, and thus have unique stable models. Let $M0$, $M1$ and $M2$ be their respective stable models. To show the first equivalence, we need to show that $p(\overrightarrow{X}, t)$ is in $M0$ iff it is in $M1$. This is easy as $M0$ is a subset of $M1$ ($\mathcal{T}^{syn}$ merely duplicates state information from $\mathcal{T}$), let $B$ be the body of the policy instance for $p(\overrightarrow{X}, t)$, then $M0 \models B$ iff $M1 \models B$. To show the second equivalence:

- We first consider the case where $p(\overrightarrow{x}, t)$ is the head of a policy with a single historical condition of Case 1, 2, 3 or 4, because the condition can be transformed directly with a set of monitoring rules generated. We also need to show that $p(\overrightarrow{x}, t)$ is in $M1$ iff it is in $M2$. If $p(\overrightarrow{x}, t)$ is in $M1$, then the historical condition must have been satisfied. By reasoning on the monitoring rules' conditions, we can show that the assertion rule must have triggered at some point $t' \leq t$ but since then the retraction rule has not. This further implies that the auxiliary fact must be in store at $t$, and hence $p(\overrightarrow{x}, t)$ is in $M2$ (e.g. by $ST$). Similarly, if $p(\overrightarrow{x}, t)$ is not in $M1$, then either the assertion rule has never triggered or the retraction rule has triggered after the assertion rule. Thus, the auxiliary fact must not be in store at $t$ and $p(\overrightarrow{x}, t)$ will not be in $M2$.
- We then consider the case where $p(\overrightarrow{x}, t)$ is the head of a policy with multiple historical conditions that are from Cases 1–4. This is proven by induction on its corresponding set of transformed policies with single historical condition.

Finally, we show that the theorem holds for policies with same head but different definitions (this is trivial). ■

Note again that the stable model containing $\mathcal{M}(\mathcal{T}^{syn})$ is used for proving correctness only. To evaluate a transformed policy at $t$ in practice, only the system state and the auxiliary store at $t$ are required.

## 6 Discussions

*Complexity of the Evaluation.* Given a policy with multiple historical conditions, let $m$ be the total number of historical conditions in the output policies of the compilation phase, let $C = size(Subject) \times size(Target) \times size(Action) + size(Fluent)$, then the maximum size of the encoded bounded history used for evaluation is equal to the total number of all the auxiliary facts relevant to the transformed policies, i.e. $C \times m$. Hence, given $n$ history-based policies, where the maximum number of (generated) historical conditions for each policy is $m$, the maximum auxiliary store size is $max\_space(Store) = C \times n \times m$, i.e. it is *polynomial in the number of policies and the number of historical conditions in the policies.* However, in practice this upper bound is hardly met for most common policies.

The evaluation of a history-free policy body or a history-free monitoring rule condition consists of the queries to the domain properties, say there are $k$, the queries to the auxiliary store, say there are $l$, and the trigger of evaluation of other (transformed or non-transformed) history-free policies, e.g. *permitted* and *denied*, say there are $j$. If $j = 0$, then let $C_{domain}$ and $C_{store}$ be the average time required for each domain property query and auxiliary store query respectively, the maximum time required for the evaluation will be $time(Evaluation) = C_{domain} \times k + C_{store} \times l$. If $j \neq 0$, it is easy to see the the logic program formed by the policies is equivalent to a Datalog program (since function symbols are never nested), and hence the evaluation time in this case is the same as that for a Datalog query.

*Dynamic Injection of Policies.* So far we assume that policies are given before the system starts, so that the necessary partial history can be encoded in the auxiliary store from the beginning. In practice, we should allow policies to be added at system runtime. But this may change the partial history required for policy evaluation and it will be too late to recover the information. To cope with such situation we can manage the system as follows: (1) we transform a policy when it is added; (2) we first activate only the monitoring rules, so that the system can start to accumulate the extra history; (3) when sufficient partial history is collected, the transformed policies can be activated. Only at this point, the new policy is considered to be *fully added* to the system and its correct evaluation can be guaranteed afterward.

*Tree-like Dependency Assumption for Multiple Historical Conditions.* Consider the following policy:

$$\begin{aligned} p(T) \leftarrow\ & h1(\overrightarrow{X}_1, T1), h2(\overrightarrow{X}_2, T2), \\ & T - \phi \leq T1, T1 \leq T, \\ & T - \phi \leq T2, T2 \leq T, \\ & T1 \leq T2. \end{aligned}$$

The temporal relationships between $T$, $T1$ and $T2$ are $T1 \prec T, T2 \prec T, T1 \prec' T2$, i.e. the DCG of the policy is not a tree but a triangle. If it were to be transformed using the algorithm presented in this paper, two history-free policies would be generated:

$$\begin{aligned} & p(T) \leftarrow h1'(\overrightarrow{X}_1, T), combined\_h2'(\overrightarrow{X}_2, \overrightarrow{X}_1, T). \\ & combined\_h2(\overrightarrow{X}_2, \overrightarrow{X}_1, T) \leftarrow h2(\overrightarrow{X}_2, T), h1''(\overrightarrow{X}_1, T). \end{aligned}$$

and three sets of monitoring rules would be generated for $h1'(T)$ (replacing $h1(T1), T1 \prec T$), $h1''(T)$ (replacing $h1(T1), T1 \prec' T2$), and $combined\_h2(T)$ (replacing $combined\_h2(T2), T2 \prec T$). Note that condition $h1$ would then be monitored independently by two sets of monitoring rules (i.e. for $h1'$ and $h2''$). Consider the following situation: let $h1$ hold at and only at time $t3$ and $t1$, and $h2$ hold at and only at time $t2$. If the first generated policy is evaluated at $t$, and the constraints $t3 < t - \phi, t - \phi < t2 < t, t - \phi < t1 < t, t2 < t1$ are satisfied, then the evaluation result will be *true*, which is incorrect. One possible way to address this problem is to extend $h1'$ and $combined\_h2'$ to carry the time of the $h1$ fact that is used to proved $h1'$ and $combined\_h2'$ to make sure that both predicates will use the same $h1$. Thus, this new time argument shared between $h1'$ to $combined\_h2'$ needs to be added to the vector of variables already passed from $h1'$ to $combined\_h2'$. The new rules generated would look like:

$$p(T) \leftarrow h1'(\overrightarrow{X}_1, T1, T), combined\_h2'(\overrightarrow{X}_2, \overrightarrow{X}_1, T1, T).$$
$$combined\_h2(\overrightarrow{X}_2, \overrightarrow{X}_1, T1, T) \leftarrow h2(\overrightarrow{X}_2, T), h1''(\overrightarrow{X}_1, T1, T).$$

With these extensions the counters tagged to the facts in the store will not work since copies of the same fact need to be tagged now with different time stamps. The ECA rules would need to be changed too. Changes to the *assert* are easy but the *retracts* would need to figure out what is the right fact to retract. This change may require the storing of a considerably larger amount of history but we still need to work out the details.

*Covering Conditions in PLTL.* To handle conditions expressed as a PLTL formula, we need to allow basic terms of the form $\mathcal{R}(\overrightarrow{X})$ as first argument of the predicate *holds*,[2] and recursively defined with FOL connectives `and`, `or`, `not`, and PLTL connectives such as `since` and `previously`. We also need the following domain independent axioms:

$$holds(F_1 \texttt{ and } F_2, T) \leftarrow holds(F_1, T), holds(F_2, T).$$
$$holds(F_1 \texttt{ or } F_2, T) \leftarrow holds(F_1, T).$$
$$holds(F_1 \texttt{ or } F_2, T) \leftarrow holds(F_2, T).$$

$$holds(\texttt{previously } F, T) \leftarrow holds(F, T_1), T_1 = T - 1.$$
$$holds(\texttt{not } F, T) \leftarrow \texttt{not } holds(F, T).$$
$$holds(F_1 \texttt{ since } F_2, T) \leftarrow holds(F_2, T_1), T_1 \leq T,$$
$$\texttt{not } unholdsBetween(F_1, T_1, T).$$
$$holds(\mathcal{R}(\overrightarrow{X}), T) \leftarrow \mathcal{R}(\overrightarrow{X}, T) \quad \textit{for each } \mathcal{R}$$
$$\textit{in } \{do, deny, req, permitted, denied, holds, happens\}.$$

With these axioms we are able to cover PLTL formulas, with $F_i$ at the literal level, in the condition of authorisations. To cover the full set of PLTL formulas, i.e. handling cases such as $holds(\texttt{not } ((F_1' \texttt{ or } F_2') \texttt{ since } F_3'), T)$, we need to extend the approach to perform recursive transformation of the generated monitoring rules.

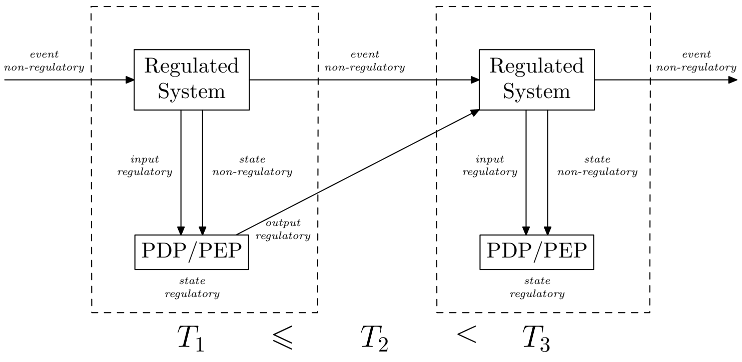
## 7 A Glance to Our Implementation

We have a reference implementation of our basic cases writen in Java for the Imperius policy language SPL (`http://incubator.apache.org/imperius/`). Below we give an overview of the implementation but first we describe briefly the operational semantics of the formal framework introduced in [6].

### 7.1 The Formal Framework

The operational semantics of the formal framework [6] can be illustrated in the following diagram:

[2] Note that these terms have no time argument.



Operational Model [6]

A system can be seen as a state machine:

– system actions or events will lead the system from one state to another;
– the system is assumed to be sequential (at the moment), i.e. no concurrent actions or events;
– at any state when a request for a system action is received by the Policy Enforcement Point (PEP), the relevant policies will be evaluated by the Policy Decision Point (PDP) according to the current or past state properties, and/or the past actions or events. Depending on the evaluation result, the request will be either *permitted* or *denied*. In the former case, the action will be performed by the system; whereas in the latter case the system will deny the request explicitly. In both cases, the system will move to a new state.
– a *trace* is a sequence of system states that obeys the regulation of the policies (specified by administrators).

The language described in Section 3 can be used to model authorisation policies for the system.

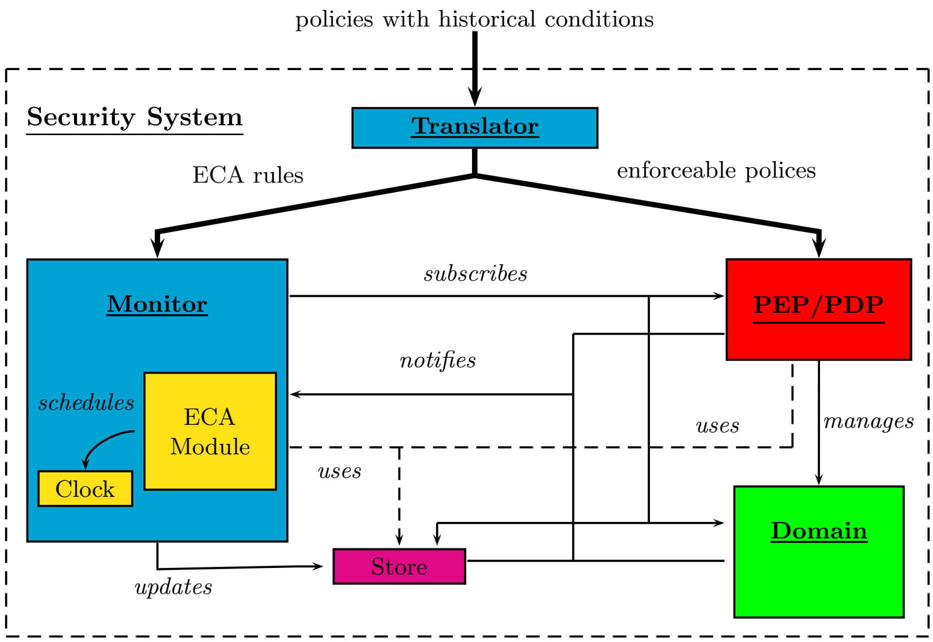
## 7.2 Design Overview

Below is a diagram showing the different components necessary for the implementation and the interactions between the components:

**Policy Management Module (the PEP/PDP):** it is the module that stores and enforces given policies. It implements the PEP/PDP of the framework, which intercepts the requests of actions from the domain (system resources), grants permissions to and performs the actions according to the policies.

**System Domain:** this is the set of all system resources whose operations/interactions are regulated by the security policies (i.e. managed by the PEP/PDP).

**Auxiliary Store:** it is used for storing the auxiliary facts generated by the *Monitor*.



**Translator:** it takes given policies with historical conditions, transforms them into low level enforceable policies (i.e. without historical conditions) for the PEP/PDP, and generates a set of ECA rules for the Monitor.

**Monitor:** it has two sub-components:

- the *ECA module* which stores and executes ECA rules from the translator;
- the *Clock* which can be used to schedule updates to the Storage.

When a new ECA rule is received by the Monitor, the Monitor will subscribe relevent *notifications* from the PEP/PDP, the Domain and the Store. A subscribed notification may be generated and sent to the Monitor when the PEP/PDP receives a request, or acts after a decision is made, or when a monitored resource property (including the Store) has changed. The notifications sent to the Monitor may trigger the activations of some ECA rules. Depending on the conditions and the resulting actions, the Monitor will assert/retract auxiliary facts to/from the Store. The Monitor Clock may help to schedule the actions at correct time. The Store is accessible by the PEP/PDP and the ECA module of the Monitor for policy evaluation.

## 8 Conclusion and Future Work

We have presented a transformation approach that allows history-based policies modelled as logic formulas to be implemented and evaluated efficiently in practice. By analysing the historical conditions of a set of policies, we automatically generate system monitoring rules that track changes at each state during execution and collect required information for future evaluation of historical conditions. The information collected is recorded as propositions/facts in a store. Historical conditions in the original policies are then replaced with "online" store queries, and policies can be evaluated efficiently at each state. We have given a proof sketch of the correctness theorem corresponding to this transformation. We gave an overview of our current implementation. The implementation only covers the base cases and paralallel temporal conditions. Our next step is to extend the implementation to cover sequential temporal conditions.

## Acknowlegements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

## References

1. Agrawal, D., Calo, S.B., Lee, K.-W., Lobo, J.: Issues in designing a policy language for distributed management of it infrastructures. In: Integrated Network Management, pp. 30–39 (2007)
2. Bandara, A.K.: A Formal Approach to Analysis and Refinement of Policies. PhD thesis, Imperial College London (2005)
3. Bonatti, P.A., Olmedilla, D.: Rule-based policy representation and reasoning for the semantic web. In: Reasoning Web, pp. 240–268 (2007)
4. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2), 149–186 (1995)
5. Craven, R., Lobo, J., Lupu, E., Russo, A., Sloman, M.: Decomposition techniques for policy refinement. In: International Conference on Network and Service Management (2010)
6. Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E., Bandara, A., Calo, S., Sloman, M.: Expressive policy analysis with enhanced system dynamicity. In: ASIACCS 2009 (2009)
7. Gabaldon, A.: Non-markovian control in the situation calculus. In: Eighteenth National Conference on Artificial Intelligence, pp. 519–524. American Association for Artificial Intelligence, Menlo Park (2002)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the Fifth International Conference on Logic Programming, pp. 1070–1080. The MIT Press, Cambridge (1988)
9. Gelfond, M., Lobo, J.: Authorization and obligation policies in dynamic systems. In: ICLP, pp. 22–36 (2008)
10. Gonzalez, G., Baral, C., Gelfond, M.: Alan: An action language for modelling non-markovian domains. Studia Logica 79(1), 115–134 (2005)

11. Gurevich, Y., Neeman, I.: Dkal: Distributed-knowledge authorization language. In: CSF 2008: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium, pp. 149–162. IEEE Computer Society, Washington, DC (2008)
12. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. ACM Trans. Database Syst. 26(2), 214–260 (2001)
13. Kolovski, V., Hendler, J., Parsia, B.: Analyzing web access control policies. In: WWW 2007: Proceedings of the 16th International Conference on World Wide Web, pp. 677–686. ACM, New York (2007)
14. Li, N., Mitchell, J.C., Winsborough, W.H.: Beyond proof-of-compliance: security analysis in trust management. J. ACM 52(3), 474–514 (2005)
15. Lupu, E., Dulay, N., Sloman, M., Sventek, J., Heeps, S., Strowes, S., Twidle, K., Keoh, S.-L., Schaeffer-Filho, A.: Amuse: autonomic management of ubiquitous e-health systems. Concurr. Comput.: Pract. Exper. 20(3), 277–295 (2008)
16. OASIS. OASIS eXtensible Access Control Markup Language (XACML) TC (2005)
17. Yemini, S.A., Kliger, S., Mozes, E., Yemini, Y., Ohsie, D.: High speed and robust event correlation. Communications Magazine 34(5), 82–90 (1996)

# Translating General Game Descriptions into an Action Language

Michael Thielscher

School of Computer Science and Engineering
The University of New South Wales
mit@cse.unsw.edu.au

**Abstract.** The game description language (GDL), which is the basis for the grand AI challenge of *general* game playing, can be viewed as yet another action language. However, due to its unique way of addressing the frame problem, GDL turned out to be surprisingly difficult to relate to any of the classical action formalisms. In this paper, we present the first complete embedding of GDL into an existing member, known as, $\mathcal{C}+$, of the family of action languages. Our provably correct translation paves the way for applying known results from reasoning about actions, including competitive implementations such as the Causal Calculator, to the new and ambitious challenge of general game playing.

## 1 Introduction

General game playing is concerned with the development of systems that understand the rules of previously unknown games and learn to play these games well without human intervention. Identified as one of the contemporary grand AI challenges, this endeavour requires to build intelligent agents that are capable of high-level reasoning and learning. An annual competition has been established in 2005 to foster research in this area [10]. This has lead to a number of successful approaches and systems [12,5,18,6].

Representing and reasoning about actions is a core technique in general game playing. A game description language (GDL) has been developed to formalise the rules of arbitrary $n$-player games ($n \geq 1$) in such a way that they can be automatically processed by a general game player [14]. The emphasis is on high-level, declarative descriptions. This allows successful players to reason about the rules of an unknown game in order to extract game-specific knowledge [19,23] and to automatically design evaluation functions [12,5,18].

The game description language shares principles with action formalisms, a central aspect of logic-based knowledge representation ever since the introduction of the classical Situation Calculus [15]. For instance, a game description must entail the conditions under which a move is legal. This corresponds to action preconditions. Furthermore, the rules of a game must include how the various moves change the game state. This corresponds to effect specifications.

Although GDL can thus be viewed as a special-purpose action language, formally relating GDL relates to any existing action formalism proved to be surprisingly difficult. The main reason seems to be that GDL is based on a rather unique solution to the fundamental frame problem [16] where positive frame axioms are combined with the principle of negation-by-failure [4] to encode negative effects. This makes it notoriously difficult to infer the explicit positive and negative effects of an individual move from its GDL description [21].

In this paper, we present the first formal result that links the general game description language to an existing action formalism. Specifically, we develop an embedding of GDL into a successor of the Action Description Language invented by Michael Gelfond and his colleague Vladimir Lifschitz [9]. The target action language we chose, known as $\mathcal{C}+$ [11], allows to circumvent the problem of identifying the positive and negative effects of individual moves. As the main result we formally prove that our embedding is correct and provides an easily automatable translation of full GDL into an action language.

Our result paves the way for applying known methods from reasoning about actions in the area of general game playing. For example, we can immediately deploy an existing implementation for $\mathcal{C}+$, the Causal Calculator [11,1], to do automated reasoning with, and about, game descriptions in GDL. In this way the relatively new field of General Game Playing can profit from many years of research in reasoning about actions. Conversely, this grand AI challenge may provide a new and interesting testbed for existing action formalisms.

The rest of the paper is organised as follows. In Section 2, we recapitulate the basic syntax and semantics of the general game description language, followed by a brief introduction to action languages. In Section 3, we present a complete embedding of this language into the action language $\mathcal{C}+$. In the section that follows, we prove that this translation is correct. We conclude in Section 5.

## 2 Preliminaries

### 2.1 Describing Games in GDL

The Game Description Language (GDL) has been developed to formalise the rules of any finite game with complete information in such a way that the description can be automatically processed by a general game player [10,14]. GDL is based on the standard syntax of logic programs, including negation. We assume familiarity with the basic notions of normal logic programs, as can be found in [3]. We adopt the Prolog convention according to which variables are denoted by uppercase letters and predicate and function symbols start with a lowercase letter. As a tailor-made specification language, GDL uses a few predefined predicate symbols shown in Table 1. The use of these keywords must obey the following syntactic restrictions [14].

**Definition 1.** *In a* valid *GDL game description,*

- `role` *only appears in facts;*
- `init` *and* `next` *only appear as head of clauses;*
- `init` *does not depend*[1] *on any of* `true`, `legal`, `does`, `next`, `terminal`, *or* `goal`*;*

[1] A predicate $p$ is said to *depend* on a predicate $q$ if $q$ occurs in the body of a clause for $p$, or if some predicate in a clause for $p$ depends on $q$.

**Table 1.** The pre-defined keywords of GDL

| | |
|---|---|
| `role(R)` | `R` is a player |
| `init(F)` | `F` holds in the initial position |
| `true(F)` | `F` holds in the current position |
| `legal(R,M)` | player `R` has legal move `M` |
| `does(R,M)` | player `R` does move `M` |
| `next(F)` | `F` holds in the next position |
| `terminal` | the current position is terminal |
| `goal(R,N)` | player `R` gets goal value `N` |

– `true` *and* `does` *only appear in the body of clauses; and*
– *neither of* `legal`*,* `terminal`*, or* `goal` *depends on* `does`*.*

**Example 1.** Figure 1 depicts a game description of standard Tic-Tac-Toe, where two players, respectively called *xplayer* and *oplayer*, take turn in marking the cells of a 3×3-board. (For the sake of simplicity, the straightforward definitions of termination and winning criteria are omitted.) Of particular interest from the viewpoint of action logics is the encoding of position updates. The first clause with head `next` specifies the direct positive effect of marking a cell. The second clause serves as an action-independent frame axiom for state feature *cell*. The third and fourth clause for `next` describe how the argument of *control* alters between consecutive moves. It is especially noteworthy that the clauses together entail the implicit negative effect that *control*(*xplayer*) will become false (after any joint move) unless *control*(*oplayer*) holds in the current position, and vice versa. This is a consequence of the negation-as-failure principle built into the semantics of GDL, which we will briefly recapitulate below. The reader should also note the clausal definition of the predicate *taken*. This clause acts as a state constraint, giving rise to the indirect effect that a cell will be taken after a player has marked it. ■

GDL imposes some general restrictions on a set of clauses as a whole. Specifically, it must be *stratified* [2], *allowed* [13], and comply with a recursion restriction that guarantees that the entire set of clauses is equivalent to a finite set of ground clauses (we refer to [14] for details). Stratified logic programs are known to admit a specific standard model [17], which coincides with its unique stable model (also called its answer set) [8,7].

Based on the concept of this standard model, a GDL description is understood as a state transition system as follows [20]. To begin with, any valid game description $G$ contains a finite set of function symbols, including constants, which implicitly determines a set of ground terms $\Sigma$. This set constitutes the symbol base $\Sigma$ in the formal semantics for $G$.

The players $R \subseteq \Sigma$ and the initial position of a game can be directly determined from the clauses for `role` and `init`, respectively. In order to determine

```
role(xplayer).
role(oplayer).

init(control(xplayer)).

legal(P,mark(M,N))   :- true(control(P)),
                        index(M), index(N), ¬taken(M,N)
legal(xplayer,noop) :- ¬true(control(xplayer))
legal(oplayer,noop) :- ¬true(control(oplayer))

next(cell(M,N,Z))       :- marking(M,N,Z)
next(cell(M,N,Z))       :- true(cell(M,N,Z))
next(control(oplayer)) :- true(control(xplayer))
next(control(xplayer)) :- true(control(oplayer))

marking(M,N,x) :- does(xplayer,mark(M,N))
marking(M,N,o) :- does(oplayer,mark(M,N))

taken(M,N) :- marker(Z), true(cell(M,N,Z))

index(1).
index(2).
index(3).
marker(x).
marker(o).

terminal :- ...
goal(xplayer,100) :- ...
```

**Fig. 1.** A GDL description of Tic-Tac-Toe. Game positions are encoded using two features: $\mathit{control}(P)$, where $P \in \{\mathit{xplayer}, \mathit{oplayer}\}$, and $\mathit{cell}(M,N,Z)$, where $M,N \in \{1,2,3\}$ and $Z \in \{x,o\}$ (the markers).

the legal moves, update, termination, and goal values (if any) for a given position, this position has to be encoded first, using the keyword `true`. To this end, for any *finite* subset $S = \{f_1, \ldots, f_n\} \subseteq \Sigma$ of ground terms, the following set of logic program facts encodes $S$ as the current position:

$$S^{\texttt{true}} \stackrel{\text{def}}{=} \{\texttt{true}(f_1)\texttt{.},\ \ldots,\ \texttt{true}(f_n)\texttt{.}\}$$

Furthermore, for any function $A : (\{r_1, \ldots, r_k\} \mapsto \Sigma)$ that assigns a move to each player $r_1, \ldots, r_k \in R$, the following set of facts encodes $A$ as a joint move:

$$A^{\texttt{does}} \stackrel{\text{def}}{=} \{\texttt{does}(r_1, A(r_1))\texttt{.},\ \ldots,\ \texttt{does}(r_k, A(r_k))\texttt{.}\} \tag{1}$$

Adding the unary clauses $S^{\texttt{true}} \cup A^{\texttt{does}}$ allows to infer the position that results from taking moves $A$ in state $S$. All this is summarised in the following definition.

**Definition 2.** *Let $G$ be a GDL specification whose signature determines the set of ground terms $\Sigma$. Let $2^{\Sigma}$ be the set of* finite *subsets of $\Sigma$. The* semantics *of $G$ is the state transition system $(R, S_0, T, l, u, g)$ where*[2]

- $R = \{r \in \Sigma : G \models \texttt{role}(r)\}$ *(the players);*
- $S_0 = \{f \in \Sigma : G \models \texttt{init}(f)\}$ *(the initial position);*
- $T = \{S \in 2^{\Sigma} : G \cup S^{\texttt{true}} \models \texttt{terminal}\}$ *(the terminal positions);*
- $l = \{(r, a, S) : G \cup S^{\texttt{true}} \models \texttt{legal}(r, a)\}$*, where $r \in R$, $a \in \Sigma$, and $S \in 2^{\Sigma}$ (the legality relation);*
- $u(A, S) = \{f \in \Sigma : G \cup S^{\texttt{true}} \cup A^{\texttt{does}} \models \texttt{next}(f)\}$*, for all $A : (R \mapsto \Sigma)$ and $S \in 2^{\Sigma}$ (the update function);*
- $g = \{(r, v, S) : G \cup S^{\texttt{true}} \models \texttt{goal}(r, v)\}$*, where $r \in R$, $v \in \mathbb{N}$, and $S \in 2^{\Sigma}$ (the goal relation).*

*For $S, S' \in 2^{\Sigma}$ we write $S \xrightarrow{A} S'$ if $A : (R \mapsto \Sigma)$ is such that $(r, A(r), S) \in l$ for each $r \in R$ and $S' = u(A, S)$ (and $S \notin T$). We call*

$$S_0 \xrightarrow{A_0} S_1 \xrightarrow{A_1} \ldots \xrightarrow{A_{n-1}} S_n$$

*a* development *(where $n \geq 0$).*

**Example 1. (Continued)** The clauses in Figure 1 obviously entail the initial state $S_0 = \{\mathit{control}(\mathit{xplayer})\}$. Suppose, therefore, $S_0^{\texttt{true}}$ is added to the program:

```
    true(control(xplayer)).
```

Since all instances of $\mathit{taken}(m, n)$ are false in the standard model of the program thus obtained, it follows that *xplayer* has nine legal moves, viz. $\mathit{mark}(m, n)$ for each pair of arguments $(m, n) \in \{1, 2, 3\} \times \{1, 2, 3\}$. The only derivable legal move for *oplayer* is the constant *noop*. In order to determine the outcome of a particular joint move, say $A = \{\mathit{xplayer} \mapsto \mathit{mark}(2, 2), \mathit{oplayer} \mapsto \mathit{noop}\}$, we have to further add $A^{\texttt{does}}$ to the program:

```
    does(xplayer,mark(2,2)).
    does(oplayer,noop).
```

The resulting position is determined as the derivable arguments of keyword `next`, which in this case are

$$\{\mathit{cell}(2, 2, x),\ \mathit{control}(\mathit{oplayer})\}$$ ■

Definition 2 provides the basis for interpreting a GDL description as an abstract $k$-player game as follows. In every position $S$, starting with $S_0$, each player $r$ chooses a move $a$ that satisfies $l(r, a, S)$. As a consequence the game state changes to $u(A, S)$, where $A$ is the joint move. The game ends if a position in $T$ is reached, and then the goal relation, $g$, determines the outcome. The restrictions in GDL ensure that entailment wrt. the standard model is decidable and that only finitely many instances of each predicate are entailed. This guarantees that the definition of the semantics is effective.

[2] Below, entailment $\models$ is via the aforementioned standard model for a stratified set of clauses.

## 2.2 Action Languages

Michael Gelfond and Vladimir Lifschitz in the early 1990s developed a basic language called $\mathcal{A}$ to describe action domains in a simple and intuitive way but with a precise, formal semantics [9]. The purpose of this language was to facilitate the assessment and comparison of existing AI formalisms and implementations for reasoning about actions. Over the years, $\mathcal{A}$ has been extended in various ways, and today there exists a whole family of action languages, including one called $\mathcal{C}+$ [11], along with several implementations.

Unlike GDL, all action languages use a sorted signature which distinguishes between *actions* and *fluents*. In $\mathcal{C}+$, there is the further distinction between *simple* fluents and *statically determined* ones. For example, a description of Tic-Tac-Toe as action domain may use the simple fluents $\mathit{control}(P)$ and $\mathit{cell}(M,N,Z)$ (for all instances $P \in \{\mathit{xplayer}, \mathit{oplayer}\}$, $M,N \in \{1,2,3\}$, and $Z \in \{x,o\}$); the statically determined fluent $\mathit{legal}(\mathit{xplayer}, \mathit{noop})$; and, with the same domain for $M,N,Z$, the actions $\mathit{marking}(M,N,Z)$ and $\mathit{does}(\mathit{xplayer}, \mathit{mark}(M,N))$.

Given a sorted domain signature, a *fluent formula* in the action language $\mathcal{C}+$ is a formula with only fluents as atoms, while an *action formula* is a formula with at least one action as atom. A general formula can have both actions and fluents as atoms. For instance, with the fluents and actions just introduced, $\neg \mathit{control}(\mathit{xplayer})$ is a fluent formula while any instance of the (reverse) implication $\mathit{marking}(M,N,Z) \subset \mathit{does}(\mathit{xplayer}, \mathit{mark}(M,N))$ is an action formula.

A domain description in $\mathcal{C}+$ is composed of so-called causal laws, of which there are three types.

1. A *static law* is of the form **caused** $F$ **if** $G$, where $F$ and $G$ are fluent formulas. Intuitively, it means that there is a cause for $F$ to be true in every state in which $G$ holds. An example is

$$\textbf{caused } \mathit{legal}(\mathit{xplayer}, \mathit{noop}) \textbf{ if } \neg \mathit{control}(\mathit{xplayer})$$

2. An *action dynamic law* is of the same form, **caused** $F$ **if** $G$, but with $F$ an action formula and $G$ a general formula. An example is

$$\textbf{caused } \mathit{marking}(M,N,Z) \subset \mathit{does}(\mathit{xplayer}, \mathit{mark}(M,N)) \textbf{ if } \top$$

   where symbol $\top$ denotes the unconditional truth.
3. A *fluent dynamic law* is of the form $G$ **causes** $F$, where $G$ is a general formula and $F$ is a fluent formula without statically determined fluents. Intuitively, it means that there is a cause for $F$ to be true in the next state if $G$ is true in the current one.[3] An example is

$$\mathit{marking}(M,N,Z) \textbf{ causes } \mathit{cell}(M,N,Z)$$

The formal semantics for causal laws will be introduced in Section 4 as part of the main correctness proof in this paper.

[3] Full $\mathcal{C}+$ uses a slightly more general definition of fluent dynamic laws, but the restricted version suffices as a target language for translating GDL.

## 3 Translating GDL into Action Language $\mathcal{C}+$

In this section we construct, step by step, a complete embedding of GDL into the action language $\mathcal{C}+$. For this we assume given an arbitrary game description $G$. By $grd(G)$ we denote the (finite) set of ground instances of the clauses in $G$.

Since action languages use a sorted signature, the first step of translating GDL into action language $\mathcal{C}+$ is to assign a unique sort to the various syntactic elements of a given game description. This is easily done as follows.

**Definition 3.** *Let $G$ be a GDL game description with roles $R$.*

1. *The* simple fluents *are the ground terms $f$ that occur as arguments in* `init`$(f)$*,* `true`$(f)$*, or* `next`$(f)$ *in some clause in $grd(G)$.*
2. *The* statically determined fluents *are*
   - *the instances of the three pre-defined predicates* `legal`*,* `terminal`*, and* `goal`*; and*
   - *the instances of the atoms other than the pre-defined GDL predicates (cf. Table 1) that occur in some clause in $grd(G)$ and do not depend on* `does`*.*
3. *The* actions *are*
   - *all terms of the form $does(r, a)$ such that $r \in R$ and "$a$" is a ground term with* `does`$(\_, a)$ *or* `legal`$(\_, a)$ *appearing in a clause in $grd(G)$; and*
   - *the instances of the atoms other than the pre-defined GDL predicates that occur as a head of some clause in $grd(G)$ and that depend on* `does`*.*

This assignment of sorts is straightforward with the exception of the domain-dependent predicates whose definition depends on the GDL keyword `does` (an example is the predicate $marking(M, N, Z)$ in Figure 1). These predicates need to be formally treated as actions for purely syntactic reasons, as will become clear in Section 4 when we prove the correctness of the translation.

**Example 1. (Continued)** Recall the Tic-Tac-Toe clauses in Figure 1. They determine the simple fluents[4]

$$control(xplayer),\ control(oplayer),\ cell(1,1,x),\ \ldots,\ cell(3,3,o)$$

along with the statically determined fluents

$$\begin{array}{l} legal(xplayer, mark(1,1)),\ \ldots,\ legal(oplayer, noop), \\ terminal,\ goal(xplayer, 100),\ \ldots, \\ taken(1,1),\ \ldots,\ taken(3,3),\ index(1),\ \ldots,\ marker(o) \end{array}$$

[4] For the sake of clarity, we only mention some of the instances of the fluents and actions in this domain; the actual sets are larger and include irrelevant instances such as $cell(xplayer, xplayer, xplayer)$. When deploying our translation in practice, such instances should be detected with the help of a pre-processor, which can be built using a method described in [18] to compute the domains of the predicates in a GDL description.

and the actions

$$\begin{array}{l} \mathit{does}(\mathit{xplayer}, \mathit{noop}),\ \ \mathit{does}(\mathit{oplayer}, \mathit{noop}),\ \ \mathit{does}(\mathit{xplayer}, \mathit{mark}(1,1)),\ \ldots \\ \mathit{marking}(1,1,x),\ \ldots,\ \ \mathit{marking}(3,3,o) \end{array} \qquad \blacksquare$$

Next we present the main result of this paper, which provides a fully automatic translation for any given GDL description into a set of causal laws in action language $\mathcal{C}+$.

**Definition 4.** *Consider a GDL description $G$ in which each occurrence of* `true`$(f)$ *has been substituted by $f$. The* translation *of $G$ into $\mathcal{C}+$ is obtained as follows.*

1. *For every clause $f$ `:-` $B$ in $\mathit{grd}(G)$ where $f$ is a statically determined fluent (or a user-defined action predicate, respectively) introduce the law*

$$\textbf{caused}\ \ (f \subset B^+)\ \ \textbf{if}\ \ B^- \tag{2}$$

   *where $B^+$ (respectively, $B^-$) is the conjunction of all atoms that occur positively (respectively, negatively) in $B$.*
2. *For every statically determined fluent $f$ (and every user-defined action predicate, respectively) add the law*

$$\textbf{caused}\ \ \neg f\ \ \textbf{if}\ \ \neg f \tag{3}$$

3. *For every clause* `next`$(f)$ `:-` $B$ *in $\mathit{grd}(G)$ introduce the fluent dynamic law*

$$B\ \ \textbf{causes}\ \ f \tag{4}$$

4. *For every simple fluent $f$ add the fluent dynamic law*

$$\bigwedge_j \neg B_j\ \ \textbf{causes}\ \ \neg f \tag{5}$$

   *where the conjunction ranges over the bodies (taken as conjunctions) of all clauses* `next`$(f)$ `:-` $B_j \in \mathit{grd}(G)$.[5]
5. *For every action $\mathit{does}(r,a)$ introduce the action dynamic law*

$$\textbf{caused}\ \ \mathit{does}(r,a)\ \ \textbf{if}\ \ \mathit{does}(r,a) \wedge \mathit{legal}(r,a) \tag{6}$$

6. *For every pair of actions $\mathit{does}(r,a)$ and $\mathit{does}(r,a')$ such that $a \neq a'$ add the fluent dynamic law*

$$\mathit{does}(r,a) \wedge \mathit{does}(r,a')\ \ \textbf{causes}\ \ \bot \tag{7}$$

7. *Add the fluent dynamic law*

$$\bigvee_r \bigwedge_a \neg \mathit{does}(r,a)\ \ \textbf{causes}\ \ \bot \tag{8}$$

[5] As usual, an empty body of a clause is equivalent to $\top$, and an empty conjunction is equivalent to $\bot$, which stands for the unconditional falsity.

This construction deserves some explanation. According to law (2), if there is a clause for a statically determined fluent or user-defined action $f$, then there is a cause for $f$ being implied by the positive atoms in the body of this clause if the negative atoms in the body hold.[6] Law (3) says that no such cause is needed for $f$ to be false (more precisely, $\neg f$ suffices as cause for itself); this corresponds to the negation-as-failure principle used in GDL when a user-defined predicate is false in a situation where none of the clauses for this predicate applies. Note that (2) and (3) are static laws if $f$ is a fluent, but action dynamic laws if $f$ is an action.

Laws (4) and (5) say that for a fluent $f$ to hold as a result of a move, there must be an applicable clause with head $\texttt{next}(f)$, otherwise there is a cause for this fluent to be false. This is analogous to the negation-as-failure principle in GDL for negative effects of moves.

Law (6) is the formal way to express that action occurrences do not require a cause as long as they are legal. This corresponds to the treatment of moves as exogenous when they are added to a GDL program via (1) as outlined in Section 2.1. Finally, laws (7) and (8) together say that each player has to choose exactly one move in each step.

It is easy to see that the translation is modular. If we assume that the number of players, fluents, and actions is small in comparison to the number of game rules, then the resulting action theory is linear in the size of the GDL description. As an example, the translation of our Tic-Tac-Toe game is shown in Figure 2.[7]

## 4 Correctness

### 4.1 Syntactic Correctness

We begin by showing that our translation always results in a syntactically correct $\mathcal{C}+$ domain theory.

**Proposition 1.** *Let $G$ be a valid GDL description. Given the signature of Definition 3, all laws constructed from $G$ by Definition 4 are syntactically correct.*

**Proof:**

– By Definition 1, neither of the pre-defined predicates `legal`, `terminal`, or `goal` depends on `does` in $G$. Also, no other statically determined fluent depends on `does` according to Definition 3. Hence, in case $f$ is statically determined fluent, (2) and (3) are syntactically correct static laws since $f$, $B$, $\neg f$, and $\bigwedge_j B_j$ are all fluent formulas. In case $f$ is an action formula, both (2) and (3) are syntactically correct action dynamic laws.

[6] For a subtle reason, which will be revealed in Section 4, the more straightforward translation of the corresponding GDL clause into **caused** $f$ **if** $B^+ \wedge B^-$ is, in general, incorrect.

[7] Again we refrain from depicting the specifications of termination and goal values, for the sake of simplicity. Also not shown are the generic executability laws according to items 5–7 of Definition 4.

$$\begin{array}{rll}
\mathit{marking}(M,N,Z) & \textbf{causes} & \mathit{cell}(M,N,Z) \\
\mathit{cell}(M,N,Z) & \textbf{causes} & \mathit{cell}(M,N,Z) \\
\neg\mathit{marking}(M,N,Z) \land \neg\mathit{cell}(M,N,Z) & \textbf{causes} & \neg\mathit{cell}(M,N,Z) \\
\mathit{control}(\mathit{xplayer}) & \textbf{causes} & \mathit{control}(\mathit{oplayer}) \\
\neg\mathit{control}(\mathit{xplayer}) & \textbf{causes} & \neg\mathit{control}(\mathit{oplayer}) \\
\mathit{control}(\mathit{oplayer}) & \textbf{causes} & \mathit{control}(\mathit{xplayer}) \\
\neg\mathit{control}(\mathit{oplayer}) & \textbf{causes} & \neg\mathit{control}(\mathit{xplayer})
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{caused } \mathit{legal}(P,\mathit{mark}(M,N)) \subset \mathit{control}(P) \land \mathit{index}(M) \land \mathit{index}(N)} \\
 & \textbf{if } \neg\mathit{taken}(M,N) \\
\textbf{caused } \mathit{legal}(\mathit{xplayer},\mathit{noop}) & \textbf{if } \neg\mathit{control}(\mathit{xplayer}) \\
\textbf{caused } \mathit{legal}(\mathit{oplayer},\mathit{noop}) & \textbf{if } \neg\mathit{control}(\mathit{oplayer}) \\
\textbf{caused } \neg\mathit{legal}(P,A) & \textbf{if } \neg\mathit{legal}(P,A) \\
 & \\
\multicolumn{2}{l}{\textbf{caused } \mathit{marking}(M,N,x) \subset \mathit{does}(\mathit{xplayer},\mathit{mark}(M,N))} \\
 & \textbf{if } \top \\
\multicolumn{2}{l}{\textbf{caused } \mathit{marking}(M,N,o) \subset \mathit{does}(\mathit{oplayer},\mathit{mark}(M,N))} \\
 & \textbf{if } \top \\
\textbf{caused } \neg\mathit{marking}(M,N,Z) & \textbf{if } \neg\mathit{marking}(M,N,Z) \\
 & \\
\multicolumn{2}{l}{\textbf{caused } \mathit{taken}(M,N) \subset \mathit{marker}(Z) \land \mathit{cell}(M,N,Z)} \\
 & \textbf{if } \top \\
\textbf{caused } \neg\mathit{taken}(M,N) & \textbf{if } \neg\mathit{taken}(M,N) \\
 & \\
\textbf{caused } \mathit{index}(1) & \textbf{if } \top \\
\ldots & \\
\textbf{caused } \mathit{marker}(o) & \textbf{if } \top
\end{array}$$

**Fig. 2.** Beginning with the fluent dynamic laws, these are the $\mathcal{C}+$ laws that result from translating the game rules for Tic-Tac-Toe in Section 2.1. Laws with variables represent the collection of their ground instances.

- By Definition 3, if $f$ occurs as argument in $\texttt{next}(f)$ in some clause then $f$ is a simple fluent. Hence, (4) and (5) are syntactically correct fluent dynamic laws since both $f$ and $\neg f$ are fluent formulas without statically determined fluents.
- By Definition 3, $\mathit{does}(r,a)$ is an action formula. Hence, (6) is a syntactically correct action dynamic law.
- Laws (7) and (8) are syntactically correct fluent dynamic laws since $\bot$ is a fluent formula without statically determined fluents. $\square$

## 4.2 From $\mathcal{C}+$ to Causal Theories

In order to prove that our translation from GDL results in a correct $\mathcal{C}+$ domain description, we need to recapitulate the precise meaning of the resulting set of causal laws. Following [11], the semantics is obtained in two steps. First, any $\mathcal{C}+$ domain $D$ along with any non-negative integer $m$, which indicates the number of time steps to be considered, defines a set $D_m$ of so-called *causal rules* as follows.

1. The signature of $D_m$ consists in the pairs $i\!:\!c$ for all $i \in \{0,\ldots,m\}$ and fluents $c$, and the pairs $i\!:\!c$ for all $i \in \{0,\ldots,m-1\}$ and actions $c$. The intuitive meaning of $i\!:\!c$ is that $c$ holds at time step $i$.
2. For every static law **caused** $F$ **if** $G$, $D_m$ contains the causal rule
$$i\!:\!F \Leftarrow i\!:\!G$$
for every $i \in \{0,\ldots,m\}$ .[8]
3. For every action dynamic law **caused** $F$ **if** $G$, $D_m$ contains the causal rule
$$i\!:\!F \Leftarrow i\!:\!G$$
for every $i \in \{0,\ldots,m-1\}$ .
4. For every fluent dynamic law $G$ **causes** $F$, $D_m$ contains the causal rule
$$i+1\!:\!F \Leftarrow i\!:\!G$$
for every $i \in \{0,\ldots,m-1\}$ .
5. For every simple fluent $f$, $D_m$ contains the two causal rules
$$\begin{aligned} 0\!:\!f &\Leftarrow \ \ 0\!:\!f \\ \neg 0\!:\!f &\Leftarrow \neg 0\!:\!f \end{aligned}$$

Intuitively, a causal rule $p \Leftarrow q$ means that there is a cause for $p$ if $q$ is true. The purpose of these rules is to allow the application of the principle of universal causation, according to which everything that holds must have a cause. This is made precise in the second step of defining the semantics for $\mathcal{C}+$. Let $I$ be an interpretation, that is, a set of atoms from the signature for $D_m$ as given above. The *reduct* $D_m^I$ is the set of all heads of causal rules in $D_m$ whose bodies are satisfied by $I$. A *model* of $D_m$ is an interpretation $I$ which is the unique model of $D_m^I$.

**Example 1. (Continued)** Consider the following small—and simplified—extract of the causal theory corresponding to the $\mathcal{C}+$ translation of Tic-Tac-Toe (cf. Figure 2):

$$\begin{aligned}
0\!:\!\mathit{legal}(\mathit{xplayer},\mathit{mark}(1,1)) \subset 0\!:\!\mathit{control}(\mathit{xplayer}) \\
&\Leftarrow \neg 0\!:\!\mathit{taken}(1,1) \\
0\!:\!\mathit{legal}(\mathit{xplayer},\mathit{noop}) &\Leftarrow \neg 0\!:\!\mathit{control}(\mathit{xplayer}) \\
\neg 0\!:\!\mathit{legal}(\mathit{xplayer},\mathit{mark}(1,1)) &\Leftarrow \neg 0\!:\!\mathit{legal}(\mathit{xplayer},\mathit{mark}(1,1)) \\
\neg 0\!:\!\mathit{legal}(\mathit{xplayer},\mathit{noop}) &\Leftarrow \neg 0\!:\!\mathit{legal}(\mathit{xplayer},\mathit{noop}) \\[1em]
0\!:\!\mathit{taken}(1,1) \subset 0\!:\!\mathit{cell}(1,1,x) &\Leftarrow \top \\
\neg 0\!:\!\mathit{taken}(1,1) &\Leftarrow \neg 0\!:\!\mathit{taken}(1,1) \\[1em]
0\!:\!\mathit{control}(\mathit{xplayer}) &\Leftarrow 0\!:\!\mathit{control}(\mathit{xplayer}) \\
\neg 0\!:\!\mathit{control}(\mathit{xplayer}) &\Leftarrow \neg 0\!:\!\mathit{control}(\mathit{xplayer}) \\
0\!:\!\mathit{cell}(1,1,x) &\Leftarrow 0\!:\!\mathit{cell}(1,1,x) \\
\neg 0\!:\!\mathit{cell}(1,1,x) &\Leftarrow \neg 0\!:\!\mathit{cell}(1,1,x)
\end{aligned}$$

[8] If $F$ is a formula, then $i:F$ denotes the result of placing "$i:$" in front of every action or fluent occurrence in $F$.

Let these rules be denoted by $D_0$. Consider, then, the interpretation

$$I = \{0{:}\mathit{control}(\mathit{xplayer}), 0{:}\mathit{legal}(\mathit{xplayer}, \mathit{mark}(1,1))\}$$

The reduct $D_0^I$ is

$$\begin{array}{l} 0{:}\mathit{legal}(\mathit{xplayer}, \mathit{mark}(1,1)) \subset 0{:}\mathit{control}(\mathit{xplayer}) \\ \neg 0{:}\mathit{legal}(\mathit{xplayer}, \mathit{noop}) \\ 0{:}\mathit{taken}(1,1) \subset 0{:}\mathit{cell}(1,1,x) \\ \neg 0{:}\mathit{taken}(1,1) \\ 0{:}\mathit{control}(\mathit{xplayer}) \\ \neg 0{:}\mathit{cell}(1,1,x) \end{array}$$

Obviously, $I$ is the unique model for this reduct, hence it is a model for $D_0$. The reader may verify that the following two are also models for this causal theory:

$$\begin{array}{ll} I' & = \{0{:}\mathit{legal}(\mathit{xplayer}, \mathit{noop})\} \\ I'' & = \{0{:}\mathit{cell}(1,1,x), 0{:}\mathit{taken}(1,1), 0{:}\mathit{legal}(\mathit{xplayer}, \mathit{noop})\} \end{array}$$

■

## 4.3 Game Developments and Causal Models Coincide

For the following we adopt from [11] a layered representation of interpretations for causal theories $D_m$. To this end, let $i{:}s_i$ denote the set of all fluent atoms of the form $i{:}f$ that are true in a given interpretations, and let $i{:}e_i$ denote the set of all action atoms of the form $i{:}a$ that are true. Any interpretation can then be represented in the form

$$(0{:}s_0) \cup (0{:}e_0) \cup (1{:}s_1) \cup (1{:}e_1) \cup \ldots \cup (m-1{:}e_{m-1}) \cup (m{:}s_m) \tag{9}$$

This enables us to define a formal correspondence between models of a causal theory and game developments.

**Definition 5.** *Consider a game with terminal positions $T$ and goal relation $g$. A game development $S_0 \overset{A_0}{\rightarrow} S_1 \overset{A_1}{\rightarrow} \ldots \overset{A_{n-1}}{\rightarrow} S_n$* coincides *with a model (9) if, and only if,*

1. *$m = n$;*
2. *$S_i$ and $s_i$ agree on all simple fluents, for all $i = 0, \ldots, n$;*
3. *$A_i$ and $e_i$ agree on all actions $\mathit{does}(r, a)$, for all $i = 0, \ldots, n-1$;*
4. *$S_i \notin T$ and $i{:}\mathit{terminal} \notin s_i$, for all $i = 0, \ldots, n-1$;*
5. *$S_n \in T$ iff $i{:}\mathit{terminal} \in s_n$; and*
6. *$(r, v, S_i) \in g$ iff $i{:}\mathit{goal}(r, v) \in s_i$, for all $i = 0, \ldots, n$.*

We are now prepared to state—and prove—our main result on the correctness of the mapping developed in Section 3 from GDL to $\mathcal{C}+$.

**Theorem 1.** *Consider a valid GDL game $G$ with initial state $S_0$. For a given non-negative number $n \geq 0$ let $D_n$ be the causal theory determined by the* *translation of $G$ into $\mathcal{C}+$ with horizon $n$. With the addition of causal rules for the initial game state,*

$$\begin{array}{rl} 0\!:\!f \Leftarrow & \textit{for all } f \in S_0 \\ \neg 0\!:\!f \Leftarrow & \textit{for all simple fluents } f \notin S_0 \end{array} \tag{10}$$

*every game development $S_0 \xrightarrow{A_0} S_1 \xrightarrow{A_1} \ldots \xrightarrow{A_{n-1}} S_n$ for $G$ coincides with some causal model for $D_n \cup$ (10) and vice versa.*

**Proof:** By induction on $n$.
For the base case $n = 0$, items 2, 5, and 6 of Definition 5 are the only relevant ones. In a lemma used in the proof for their Proposition 3, Giunchiglia et al. [11] show that the answer sets for a logic program are equivalent to the models of the causal theory that is obtained by

– identifying each clause $f$ :- $B$ with the rule $(f \subset B^+) \Leftarrow B^-$ and
– adding $\neg f \Leftarrow \neg f$ for every atom.

This observation can be directly applied to prove our base case: the construction in (10) ensures that the initial game state, $S_0$, agrees with the (unique) causal model for $D_0 \cup$ (10) on all simple fluents; furthermore, the construction of the static laws (2) and (3) from the respective clauses in $G$ ensures that the game model coincides with the causal model on all statically determined fluents, in particular *terminal* and *goal*.[9]

For the induction step, consider a game development $S_0 \xrightarrow{A_0} \ldots \xrightarrow{A_{n-1}} S_n$ and a causal model $(0 : s_0) \cup (0 : e_0) \cup \ldots \cup (n : s_n)$ for $D_n$ so that the two coincide and both $S_n$ and $s_n$ are non-terminal states. From the aforementioned lemma in [11] it follows that the game model and the causal model coincide on the interpretation of the statically determined fluent *legal*. With this, the construction of the dynamic laws (6), (7), and (8) ensures that for every joint legal action $A_n$ in state $S_n$ there is a model $(0\!:\!s_0) \cup (0\!:\!e_0) \cup \ldots \cup (n\!:\!s_n) \cup (n\!:\!e_n) \cup (n + 1 : s_{n+1})$ for $D_{n+1}$—and vice versa—so that $A_n$ and $e_n$ agree on all actions $\textit{does}(r, a)$. Moreover, the construction of the action dynamic laws (2) and (3) along with the fluent dynamic laws (4) and (5) ensure that the updated states $S_{n+1}$ and $s_{n+1}$ agree on all simple fluents. Items 5 and 6 of Definition 5 again follow from the aforementioned lemma in [11]. □

## 5 Conclusion

The game description language GDL has been developed to provide a basis for the new, grand AI challenge of general game playing [10]. Although GDL

[9] At this point it becomes clear why we have to map clauses $f$ :-$B$ onto causal laws **caused** $f \subset B^+$ **if** $B^-$ (cf. Footnote 6). The aforementioned observation in [11] would not hold if a clause $f$ :-$B$ were identified with the causal rule $f \Leftarrow B$. For example, the standard model for the program $\{p$ :- $p\}$ is $\{\}$, whereas the causal theory $\{p \Leftarrow p, \neg p \Leftarrow \neg p\}$ admits two causal models, viz. $\{\}$ and $\{p\}$.

can be viewed as a special-purpose action language, as yet it had not been formally related to any of the existing formalisms for reasoning about actions. In this paper, we have presented the first formal result in this regard, by giving a complete embedding of GDL into the action language $\mathcal{C}+$.

Our result paves the way for applying results from reasoning about actions to various aspects of general game playing, and conversely to use this AI challenge as an interesting and ambitious testbed for existing methods in reasoning about actions. Specifically, the embedding of GDL into $\mathcal{C}+$ allows to directly deploy an existing implementation, the Causal Calculator [11,1], to reason about game descriptions in GDL. Further interesting directions for future work are, first, to investigate the formal relation between GDL and other existing action formalisms, possibly with $\mathcal{C}+$ as intermediary language on the basis of our result; and second, to investigate how the extension of GDL to imperfect information games [22] can be related to an existing action language suitable for representing both incomplete knowledge and sensing actions.

*Acknowledgement.* The author is indebted to Marina De Vos for her valuable comments on an earlier version of this paper. The author is the recipient of an Australian Research Council Future Fellowship (project number FT 0991348).

## References

1. Akman, V., Erdogan, S., Lee, J., Lifschitz, V., Turner, H.: Representing the Zoo world and the Traffic world in the language of the Causal Calculator. Artificial Intelligence 153(1-2), 105–140 (2004)
2. Apt, K., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, ch. 2, pp. 89–148. Morgan Kaufmann, San Francisco (1987)
3. Apt, K., Bol, R.: Logic programming and negation: A survey. Journal of Logic Programming 19/20, 9–71 (1994)
4. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press, New York (1978)
5. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI Conference, pp. 1134–1139 (2007)
6. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the AAAI Conference, pp. 259–264 (2008)
7. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, pp. 285–316. Elsevier, Amsterdam (2008)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the International Joint Conference and Symposium on Logic Programming (IJCSLP), pp. 1070–1080 (1988)
9. Gelfond, M., Lifschitz, V.: Representing action and change by logic programs. Journal of Logic Programming 17, 301–321 (1993)
10. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
11. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence 153(1-2), 49–104 (2004)

12. Kuhlmann, G., Dresner, K., Stone, P.: Automatic heuristic construction in a complete general game player. In: Proceedings of the AAAI Conference, pp. 1457–1462 (2006)
13. Lloyd, J., Topor, R.: A basis for deductive database systems II. Journal of Logic Programming 3(1), 55–67 (1986)
14. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Computer Science Department, Stanford University (2006), games.stanford.edu
15. McCarthy, J.: Situations and Actions and Causal Laws. Stanford Artificial Intelligence Project, Memo 2, Stanford University (1963)
16. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence 4, 463–502 (1969)
17. Przymusinski, T.: On the declarative semantics of deductive databases and logic programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 193–216. Morgan Kaufmann, San Francisco (1988)
18. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the AAAI Conference, pp. 1191–1196 (2007)
19. Schiffel, S., Thielscher, M.: Automated theorem proving for general game playing. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 911–916 (2009)
20. Schiffel, S., Thielscher, M.: A multiagent semantics for the Game Description Language. In: Filipe, J., Fred, A., Sharp, B. (eds.) Agents and Artificial Intelligence: Proceedings of ICAART. CCIS, vol. 67, pp. 44–55. Springer, Heidelberg (2009)
21. Schiffel, S., Thielscher, M., Zhao, D.: Decomposition of multi-player games. In: Nicholson, A., Li, X. (eds.) AI 2009. LNCS, vol. 5866, pp. 475–484. Springer, Heidelberg (2009)
22. Thielscher, M.: A general game description language for incomplete information games. In: Proceedings of the AAAI Conference, pp. 994–999 (2010)
23. Thielscher, M., Voigt, S.: A temporal proof system for general game playing. In: Proceedings of the AAAI Conference, pp. 1000–1005 (2010)

# Revisiting Epistemic Specifications

Mirosław Truszczyński

Department of Computer Science
University of Kentucky
Lexington, KY 40506, USA
mirek@cs.uky.edu

*In honor of Michael Gelfond on his 65th birthday!*

**Abstract.** In 1991, Michael Gelfond introduced the language of epistemic specifications. The goal was to develop tools for modeling problems that require some form of meta-reasoning, that is, reasoning over multiple possible worlds. Despite their relevance to knowledge representation, epistemic specifications have received relatively little attention so far. In this paper, we revisit the formalism of epistemic specification. We offer a new definition of the formalism, propose several semantics (one of which, under syntactic restrictions we assume, turns out to be equivalent to the original semantics by Gelfond), derive some complexity results and, finally, show the effectiveness of the formalism for modeling problems requiring meta-reasoning considered recently by Faber and Woltran. All these results show that epistemic specifications deserve much more attention that has been afforded to them so far.

## 1 Introduction

Early 1990s were marked by several major developments in knowledge representation and nonmonotonic reasoning. One of the most important among them was the introduction of *disjunctive logic programs with classical negation* by Michael Gelfond and Vladimir Lifschitz [1]. The language of the formalism allowed for rules

$$H_1 \vee \ldots \vee H_k \leftarrow B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n,$$

where $H_i$ and $B_i$ are classical literals, that is, atoms and classical or *strong* negations ($\neg$) of atoms. In the paper, we will write "strong" rather than "classical" negation, as it reflects more accurately the role and the behavior of the operator. The *answer-set* semantics for programs consisting of such rules, introduced in the same paper, generalized the stable-model semantics of normal logic programs proposed a couple of years earlier also by Gelfond and Lifschitz [2]. The proposed extensions of the language of normal logic programs were motivated by knowledge representation considerations. With two negation operators it was straightforward to distinguish between $P$ being *false by default* (there is no justification for adopting $P$), and $P$ being *strongly false* (there is evidence for $\neg P$). The former would be written as $not\ P$ while the latter as $\neg P$. And with the disjunction in the head of rules one could model "indefinite" rules which, when applied, provide partial information only (one of the alternatives in the head holds, but no preference to any of them is given).

Soon after disjunctive logic programs with strong negation were introduced, Michael Gelfond proposed an additional important extension, this time with a modal operator [3]. He called the resulting formalism the language of *epistemic specifications*. The motivation came again from knowledge representation. The goal was to provide means for the "correct representation of incomplete information in the presence of multiple extensions" [3].

Surprisingly, despite their evident relevance to the theory of nonmonotonic reasoning as well as to the practice of knowledge representation, epistemic specifications have received relatively little attention so far. This state of affairs may soon change. Recent work by Faber and Woltran on *meta-reasoning* with answer-set programming [4, 5] shows the need for languages, in which one could express properties holding across all answer sets of a program, something Michael Gelfond foresaw already two decades ago.

Our goal in this paper is to revisit the formalism of epistemic specifications and show that they deserve a second look, in fact, a place in the forefront of knowledge representation research. We will establish a general semantic framework for the formalism, and identify in it the precise location of Gelfond's epistemic specifications. We will derive several complexity results. We will also show that the original idea of Gelfond to use a modal operator to model "what is known to a reasoner" has a broader scope of applicability. In particular, we will show that it can also be used in combination with the classical logic.

Complexity results presented in this paper provide an additional motivation to study epistemic specifications. Even though programs with strong negation often look "more natural" as they more directly align with the natural language description of knowledge specifications, the extension of the language of normal logic programs with the strong negation operator does not actually increase the expressive power of the formalism. This point was made already by Gelfond and Lifschitz, who observed that there is a simple and concise way to compile the strong negation away. On the other hand, the extension allowing the disjunction operator in the heads of rules is an essential one. As the complexity results show [6, 7], the class of problems that can be represented by means of disjunctive logic programs is strictly larger (assuming no collapse of the polynomial hierarchy) than the class of problems that can be modeled by normal logic programs. In the same vein, extension by the modal operator along the lines proposed by Gelfond is essential, too. It does lead to an additional jump in the complexity.

## 2 Epistemic Specifications

To motivate epistemic specifications, Gelfond discussed the following example. A certain college has these rules to determine the eligibility of a student for a scholarship:

1. Students with high GPA are eligible
2. Students from underrepresented groups and with fair GPA are eligible
3. Students with low GPA are not eligible
4. When these rules are insufficient to determine eligibility, the student should be interviewed by the scholarship committee.

Gelfond argued that there is no simple way to represent these rules as a disjunctive logic program with strong negation. There is no problem with the first three rules. They are modeled correctly by the following three logic program rules (in the language with both the default and strong negation operators):

1. $eligible(X) \leftarrow highGPA(X)$
2. $eligible(X) \leftarrow underrep(X), fairGPA(X)$
3. $\neg eligible(X) \leftarrow lowGPA(X).$

The problem is with the fourth rule, as it has a clear meta-reasoning flavor. It should apply when the possible worlds (answer sets) determined by the first three rules do not fully specify the status of eligibility of a student $a$: neither *all* of them contain $eligible(a)$ nor *all* of them contain $\neg eligible(a)$. An obvious attempt at a formalization:

4. $interview(X) \leftarrow not\ eligible(X), not\ \neg eligible(X)$

fails. It is just another rule to be added to the program. Thus, when the answer-set semantics is used, the rule is interpreted with respect to individual answer sets and not with respect to collections of answer-sets, as required for this application. For a concrete example, let us assume that all we know about a certain student named Mike is that Mike's GPA is fair or high. Clearly, we do not have enough information to determine Mike's eligibility and so we must interview Mike. But the program consisting of rules (1)-(4) and the statement

5. $fairGPA(mike) \vee highGPA(mike)$

about Mike's GPA, has two answer sets:

$$\{highGPA(mike), eligible(mike)\}$$
$$\{fairGPA(mike), interview(mike)\}.$$

Thus, the query $?interview(mike)$ has the answer "unknown." To address the problem, Gelfond proposed to extend the language with a modal operator $K$ and, speaking informally, interpret premises $K\varphi$ as "$\varphi$ is known to the program" (the original phrase used by Gelfond was "known to the reasoner"), that is, true in all answer-sets. With this language extension, the fourth rule can be encoded as

4′. $interview(X) \leftarrow not\ K\, eligible(X), not\ K\neg eligible(X)$

which, intuitively, stands for "*interview* if neither the eligibility nor the non-eligibility is known."

The way in which Gelfond [3] proposed to formalize this intuition is strikingly elegant. We will now discuss it. We start with the syntax of *epistemic specifications*. As elsewhere in the paper, we restrict attention to the propositional case. We assume a fixed infinite countable set $At$ of *atoms* and the corresponding language $\mathcal{L}$ of propositional logic. A *literal* is an atom, say $A$, or its *strong* negation $\neg A$. A *simple modal atom* is an expression $K\varphi$, where $\varphi \in \mathcal{L}$, and a *simple modal literal* is defined accordingly. An *epistemic premise* is an expression (conjunction)

$$E_1, \ldots, E_s, not\ E_{s+1}, \ldots, not\ E_t,$$

where every $E_i$, $1 \leq i \leq t$, is a simple modal literal. An *epistemic rule* is an expression of the form

$$L_1 \vee \ldots \vee L_k \leftarrow L_{k+1}, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n, E,$$

where every $L_i$, $1 \leq i \leq k$, is a literal, and $E$ is an epistemic premise. Collections of epistemic rules are *epistemic programs*. It is clear that (ground versions of) rules (1)-(5) and (4′) are examples of epistemic rules, with rule (4′) being an example of an epistemic rule that actually takes advantage of the extended syntax. Rules such as

$$a \vee \neg d \leftarrow b, not\ \neg c, \neg K(d \vee \neg c)$$
$$\neg a \leftarrow \neg c, not\ \neg K(\neg(a \wedge c) \rightarrow b)$$

are also examples of epistemic rules. We note that the language of epistemic programs is only a fragment of the language of epistemic specifications by Gelfond. However, it is still expressive enough to cover all examples discussed by Gelfond and, more generally, a broad range of practical applications, as natural-language formulations of domain knowledge typically assume a rule-based pattern.

We move on to the semantics, which is in terms of *world views*. The definition of a world view consists of several steps. First, let $W$ be a consistent set of literals from $\mathcal{L}$. We regard $W$ as a three-valued interpretation of $\mathcal{L}$ (we will also use the term *three-valued possible world*), assigning to each atom one of the three logical values **t**, **f** and **u**. The interpretation extends by recursion to all formulas in $\mathcal{L}$, according to the following truth tables

| $\neg$ | |
|---|---|
| **f** | **t** |
| **t** | **f** |
| **u** | **u** |

| $\vee$ | **t** | **u** | **f** |
|---|---|---|---|
| **t** | **t** | **t** | **t** |
| **u** | **t** | **u** | **u** |
| **f** | **t** | **u** | **f** |

| $\wedge$ | **t** | **u** | **f** |
|---|---|---|---|
| **t** | **t** | **u** | **f** |
| **u** | **u** | **u** | **f** |
| **f** | **f** | **f** | **f** |

| $\rightarrow$ | **t** | **u** | **f** |
|---|---|---|---|
| **t** | **t** | **u** | **f** |
| **u** | **t** | **u** | **u** |
| **f** | **t** | **t** | **t** |

**Fig. 1.** Truth tables for the 3-valued logic of Kleene

By a *three-valued possible-world structure* we mean a non-empty family of consistent sets of literals (three-valued possible worlds). Let $\mathcal{A}$ be a three-valued possible-world structure and let $W$ be a consistent set of literals. For every formula $\varphi \in \mathcal{L}$, we define

1. $\langle \mathcal{A}, W \rangle \models \varphi$, if $v_W(\varphi) = \mathbf{t}$
2. $\langle \mathcal{A}, W \rangle \models K\varphi$, if for every $V \in \mathcal{A}$, $v_V(\varphi) = \mathbf{t}$
3. $\langle \mathcal{A}, W \rangle \models \neg K\varphi$, if there is $V \in \mathcal{A}$ such that $v_V(\varphi) = \mathbf{f}$.

Next, for every literal or simple modal literal $L$, we define

4. $\langle \mathcal{A}, W \rangle \models not\ L$ if $\langle \mathcal{A}, W \rangle \not\models L$.

We note that neither $\langle \mathcal{A}, W \rangle \models K\varphi$ nor $\langle \mathcal{A}, W \rangle \models \neg K\varphi$ depend on $W$. Thus, we will often write $\mathcal{A} \models F$, when $F$ is a simple modal literal or its default negation.

In the next step, we introduce the notion of the *G-reduct* of an epistemic program.

**Definition 1.** *Let $P$ be an epistemic program, $\mathcal{A}$ a three-valued possible-world structure and $W$ a consistent set of literals. The* G-reduct *of $P$ with respect to $\langle \mathcal{A}, W\rangle$, in symbols $P^{\langle \mathcal{A},W\rangle}$, consists of the heads of all rules $r \in P$ such that $\langle \mathcal{A}, W\rangle \models \alpha$, for every conjunct $\alpha$ occurring in the body of $r$.*

Let $H$ be a set of disjunctions of literals from $\mathcal{L}$. A set $W$ of literals is *closed* with respect to $H$ if $W$ is consistent and contains at least one literal in common with every disjunction in $H$. We denote by $Min(H)$ the family of all minimal sets of literals that are closed with respect to $H$. With the notation $Min(H)$ in hand, we are finally ready to define the concept of a world view of an epistemic program $P$.

**Definition 2.** *A three-valued possible-world structure $\mathcal{A}$ is a* world view *of an epistemic program $P$ if $\mathcal{A} = \{W \mid W \in Min(P^{\langle \mathcal{A},W\rangle})\}$.*

*Remark 1.* The $G$-reduct of an epistemic program consists of disjunctions of literals. Thus, the concept of a world view is well defined.

*Remark 2.* We note that Gelfond considered also inconsistent sets of literals as minimal sets closed under disjunctions. However, the only such set he allowed consisted of *all* literals. Consequently, the difference between the Gelfond's semantics and the one we described above is that some programs have a world view in the Gelfond's approach that consists of a single set of all literals, while in our approach these programs do not have a world view. But in all other cases, the two semantics behave in the same way.

Let us consider the ground program, say $P$, corresponding to the scholarship eligibility example (rule (5), and rules (1)-(3) and (4′), grounded with respect to the Herbrand universe $\{mike\}$). The only rule involving simple modal literals is

$$interview(mike) \leftarrow not\ K\, eligible(mike), not\ K\neg eligible(mike).$$

Let $\mathcal{A}$ be a world view of $P$. Being a three-valued possible-world structure, $\mathcal{A} \neq \emptyset$. No matter what $W$ we consider, no minimal set closed with respect to $P^{\langle \mathcal{A},W\rangle}$ contains $lowGPA(mike)$ and, consequently, no minimal set closed with respect to $P^{\langle \mathcal{A},W\rangle}$ contains $\neg eligible(mike)$. It follows that $\mathcal{A} \not\models K\neg eligible(mike)$.

Let us assume that $\mathcal{A} \models K\, eligible(mike)$. Then, no reduct $P^{\langle \mathcal{A},W\rangle}$ contains $interview(mike)$. Let $W = \{fairGP(mike)\}$. It follows that $P^{\langle \mathcal{A},W\rangle}$ consists only of $fairGPA(mike) \vee highGPA(mike)$. Clearly, $W \in Min(P^{\langle \mathcal{A},W\rangle})$ and, consequently, $W \in \mathcal{A}$. Thus, $\mathcal{A} \not\models K\, eligible(mike)$, a contradiction.

It must be then that $\mathcal{A} \models not\ K\, eligible(mike)$ and $\mathcal{A} \models not\ K\neg eligible(mike)$. Let $W$ be an arbitrary consistent set of literals. Clearly, the reduct $P^{\langle \mathcal{A},W\rangle}$ contains $interview(mike)$ and $fairGPA(mike) \vee highGPA(mike)$. If $highGPA(mike) \in W$, the reduct also contains $eligible(mike)$. Thus, $W \in Min(P^{\langle \mathcal{A},W\rangle})$ if and only if

$W = \{fairGPA(mike), interview(mike)\}$, or
$W = \{highGPA(mike), eligible(mike), interview(mike)\}$.

It follows that if $\mathcal{A}$ is a world view for $P$ then it consists of these two possible worlds. Conversely, it is easy to check that a possible-world structure consisting of these two possible worlds is a world view for $P$. Thus, $interview(mike)$ holds in $\mathcal{A}$, and so our representation of the example as an epistemic program has the desired behavior.

## 3 Epistemic Specifications — A Broader Perspective

The discussion in the previous section demonstrates the usefulness of formalisms such as that of epistemic specifications for knowledge representation and reasoning. We will now present a simpler yet, in many respects, more general framework for epistemic specifications. The key to our approach is that we consider the semantics given by *two-valued* interpretations (sets of atoms), and standard *two-valued* possible-world structures (nonempty collections of two-valued interpretations). We also work within a rather standard version of the language of modal propositional logic and so, in particular, we allow only for one negation operator. Later in the paper we show that epistemic specifications by Gelfond can be encoded in a rather direct way in our formalism. Thus, the restrictions we impose are not essential even though, admittedly, not having two kinds of negation in the language in some cases may make the modeling task harder.

We start by making precise the syntax of the language we will be using. As we stated earlier, we assume a fixed infinite countable set of atoms $At$. The language we consider is determined by the set $At$, the modal operator $K$, and by the *boolean connectives* $\bot$ (0-place), and $\wedge$, $\vee$, and $\rightarrow$ (binary). The BNF expression

$$\varphi ::= \bot \mid A \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid K\varphi,$$

where $A \in At$, provides a concise definition of a formula. The parentheses are used only to disambiguate the order of binary connectives. Whenever possible, we omit them. We define the unary *negation* connective $\neg$ and the 0-place connective $\top$ as abbreviations:

$$\neg\varphi ::= \varphi \rightarrow \bot$$
$$\top ::= \neg\bot.$$

We call formulas $K\varphi$, where $\varphi \in \mathcal{L}_K$, *modal atoms* (simple modal atoms that we considered earlier and will consider below are special modal atoms with $K$-depth equal to 1). We denote this language by $\mathcal{L}_K$ and refer to subsets of $\mathcal{L}_K$ as *epistemic theories*. We denote the modal-free fragment of $\mathcal{L}_K$ by $\mathcal{L}$.

While we will eventually describe the semantics (in fact, several of them) for arbitrary epistemic theories, we start with an important special case. Due to close analogies between the concepts we define below and the corresponding ones defined earlier in the context of the formalism of Gelfond, we "reuse" the terms used there. Specifically, by an *epistemic premise* we mean a conjunction of simple modal literals. Similarly, by an *epistemic rule* we understand an expression of the form

$$E \wedge L_1 \wedge \ldots \wedge L_m \rightarrow A_1 \vee \ldots \vee A_n, \tag{1}$$

where $E$ is an epistemic premise, $L_i$'s are literals (in $\mathcal{L}$) and $A_i$'s are atoms. Finally, we call a collection of epistemic rules an *epistemic program*. It will always be clear from the context, in which sense these terms are to be understood.

We stress that $\neg$ is not a primary connective in the language but a derived one (it is a shorthand for some particular formulas involving the rule symbol). Even though under some semantics we propose below this negation operator has features of default negation, under some others it does not. Thus, we selected for it the standard negation symbol $\neg$ rather than the "loaded" $not$ .

A (two-valued) *possible-world structure* is any nonempty family $\mathcal{A}$ of subsets of $At$ (two-valued interpretations). In the remainder of the paper, when we use terms "interpretation" and "possible-world structure" without any additional modifiers, we always mean a two-valued interpretation and a two-valued possible-world structure.

Let $\mathcal{A}$ be a possible-world structure and $\varphi \in \mathcal{L}$. We recall that $\mathcal{A} \models K\varphi$ precisely when $W \models \varphi$, for every $W \in \mathcal{A}$, and $\mathcal{A} \models \neg K\varphi$, otherwise. We will now define the *epistemic reduct* of an epistemic program with respect to a possible-world structure.

**Definition 3.** *Let $P \subseteq \mathcal{L}_K$ be an epistemic program and let $\mathcal{A}$ be a possible-world structure. The* epistemic reduct *of $P$ with respect to $\mathcal{A}$, $P^{\mathcal{A}}$ in symbols, is the theory obtained from $P$ as follows: eliminate every rule with an epistemic premise $E$ such that $\mathcal{A} \not\models E$; drop the epistemic premise from every remaining rule.*

It is clear that $P^{\mathcal{A}} \subseteq \mathcal{L}$, and that it consists of rules of the form

$$L_1 \wedge \ldots \wedge L_m \rightarrow A_1 \vee \ldots \vee A_n, \tag{2}$$

where $L_i$'s are literals (in $\mathcal{L}$) and $A_i$'s are atoms.

Let $P$ be a collection of rules (2). Then, $P$ is a propositional theory. Thus, it can be interpreted by the standard propositional logic semantics. However, $P$ can also be regarded as a disjunctive logic program (if we write rules from right to left rather than from left to right). Consequently, $P$ can also be interpreted by the stable-model semantics [1, 2] and the supported-model semantics [8–11]. (For normal logic programs, the supported-model semantics was introduced by Apt et al. [8]. The notion was extended to disjunctive logic programs by Baral and Gelfond [9]. We refer to papers by Brass and Dix [10], Definition 2.4, and Inoue and Sakama [11], Section 5, for more details). We write $\mathcal{M}(P)$, $\mathcal{ST}(P)$ and $\mathcal{SP}(P)$ for the sets of models, stable models and supported models of $P$, respectively. An important observation is that *each* of these semantics gives rise to the corresponding notion of an epistemic extension.

**Definition 4.** *Let $P \subseteq \mathcal{L}_K$ be an epistemic program. A possible-world structure $\mathcal{A}$ is an* epistemic model *(respectively, an* epistemic stable model*, or an* epistemic supported model*) of $P$, if $\mathcal{A} = \mathcal{M}(P^{\mathcal{A}})$ (respectively, $\mathcal{A} = \mathcal{ST}(P^{\mathcal{A}})$ or $\mathcal{A} = \mathcal{SP}(P^{\mathcal{A}})$).*

It is clear that Definition 4 can easily be adjusted also to other semantics of propositional theories and programs. We briefly mention two such semantics in the last section of the paper.

We will now show that epistemic programs with the semantics of epistemic stable models can provide an adequate representation to the scholarship eligibility example for Mike. The available information can be represented by the following program $P(mike) \subseteq \mathcal{L}_K$:

1. $eligible(mike) \wedge neligible(mike) \rightarrow \bot$
2. $fairGPA(mike) \vee highGPA(mike)$

3. $highGPA(mike) \rightarrow eligible(mike)$
4. $underrep(mike) \wedge fairGPA(mike) \rightarrow eligible(mike)$
5. $lowGPA(mike) \rightarrow neligible(mike)$
6. $\neg K\, eligible(mike), \neg K\, neligible(mike) \rightarrow interview(mike)$.

We use the predicate *neligible* to model the strong negation of the predicate $eligible$ that appears in the representation in terms of epistemic programs by Gelfond (thus, in particular, the presence of the first clause, which precludes the facts $eligible(mike)$ and $neligible(mike)$ to be true together). This extension of the language and an extra rule in the representation is the price we pay for eliminating one negation operator.

Let $\mathcal{A}$ consist of the interpretations

$$W_1 = \{fairGPA(mike), interview(mike)\}$$
$$W_2 = \{highGPA(mike), eligible(mike), interview(mike)\}.$$

Then the reduct $[P(mike)]^{\mathcal{A}}$ consists of rules (1)-(5), which are unaffected by the reduct operation, and of the fact $interview(mike)$, resulting from rule (6) when the reduct operation is performed (as in logic programming, when a rule has the empty antecedent, we drop the implication symbol from the notation). One can check that $\mathcal{A} = \{W_1, W_2\} = \mathcal{ST}([P(mike)]^{\mathcal{A}})$. Thus, $\mathcal{A}$ is an epistemic stable model of $P$ (in fact, the only one). Clearly, $interview(mike)$ holds in the model (as we would expect it to), as it holds in each of its possible-worlds. We note that in this particular case, the semantics of epistemic supported models yields exactly the same solution.

## 4 Complexity

We will now study the complexity of reasoning with epistemic (stable, supported) models. We provide details for the case of epistemic stable models, and only present the results for the other two semantics, as the techniques to prove them are very similar to those we develop for the case of epistemic stable models.

First, we note that epistemic stable models of an epistemic program $P$ can be represented by partitions of the set of all modal atoms of $P$. This is important as *a priori* the size of possible-world structures one needs to consider as candidates for epistemic stable models may be exponential in the size of a program. Thus, to obtain good complexity bounds alternative polynomial-size representations of epistemic stable models are needed.

Let $P \subseteq \mathcal{L}_K$ be an epistemic program and $(\Phi, \Psi)$ be a partition of the set of modal atoms of $P$ (all these modal atoms are, in fact, simple). We write $P_{|\Phi,\Psi}$ for the program obtained from $P$ by eliminating every rule whose epistemic premise contains a conjunct $K\psi$, where $K\psi \in \Psi$, or a conjunct $\neg K\varphi$, where $K\varphi \in \Phi$ (these rules are "blocked" by $(\Phi, \Psi)$), and by eliminating the epistemic premise from every other rule of $P$.

**Proposition 1.** *Let $P \subseteq \mathcal{L}_K$ be an epistemic program. If a possible-world structure $\mathcal{A}$ is an epistemic stable model of $P$, then there is a partition $(\Phi, \Psi)$ of the set of modal atoms of $P$ such that*

*1.* $\mathcal{ST}(P_{|\Phi,\Psi}) \neq \emptyset$
*2. For every* $K\varphi \in \Phi$*,* $\varphi$ *holds in every stable model of* $P_{|\Phi,\Psi}$
*3. For every* $K\psi \in \Psi$*,* $\psi$ *does not hold in at least one stable model of* $P_{|\Phi,\Psi}$*.*

*Conversely, if there are such partitions,* $P$ *has epistemic stable models.*

It follows that epistemic stable models can be represented by partitions $(\Phi, \Psi)$ satisfying conditions (1)-(3) from the proposition above.

We observe that deciding whether a partition $(\Phi, \Psi)$ satisfies conditions (1)-(3) from Proposition 1, can be accomplished by polynomially many calls to an $\Sigma_2^P$-oracle and, if we restrict attention to non-disjunctive epistemic programs, by polynomially many calls to an $NP$-oracle.

*Remark 3.* If we adjust Proposition 1 by replacing the term "stable" with the term "supported," and replacing $\mathcal{ST}()$ with $\mathcal{SP}()$, we obtain a characterization of epistemic supported models. Similarly, omitting the term "stable," and replacing $\mathcal{ST}()$ with $\mathcal{M}()$ yields a characterization of epistemic models. In each case, one can decide whether a partition $(\Phi, \Psi)$ satisfies conditions (1)-(3) by polynomially many calls to an $NP$-oracle (this claim is evident for the case of epistemic models; for the case of epistemic supported models, it follows from the fact that supported models semantics does not get harder when we allow disjunctions in the heads or rules).

**Theorem 1.** *The problem to decide whether a non-disjunctive epistemic program has an epistemic stable model is* $\Sigma_2^P$*-complete.*

*Proof:* Our comments above imply that the problem is in the class $\Sigma_2^P$. Let $F = \exists Y \forall Z \Theta$, where $\Theta$ is a DNF formula. The problem to decide whether $F$ is true is $\Sigma_2^P$-complete. We will reduce it to the problem in question and, consequently, demonstrate its $\Sigma_2^P$-hardness. To this end, we construct an epistemic program $Q \subseteq \mathcal{L}_K$ by including into $Q$ the following clauses (atoms $w$, $y'$, $y \in Y$, and $z'$, $z \in Z$ are fresh):

1. $Ky \rightarrow y$ ; and $Ky' \rightarrow y'$, for every $y \in Y$
2. $y \wedge y' \rightarrow$ ; and $\neg y \wedge \neg y' \rightarrow$ , for every $y \in Y$
3. $\neg z' \rightarrow z$ ; and $\neg z \rightarrow z'$, for $z \in Z$
4. $\sigma(u_1) \wedge \ldots \wedge \sigma(u_k) \rightarrow w$ , where $u_1 \wedge \ldots \wedge u_k$ is a disjunct of $\Theta$, and $\sigma(\neg a) = a'$ and $\sigma(a) = a$, for every $a \in Y \cup Z$
5. $\neg Kw \rightarrow$ .

Let us assume that $\mathcal{A}$ is an epistemic stable model of $Q$. In particular, $\mathcal{A} \neq \emptyset$. It must be that $\mathcal{A} \models Kw$ (otherwise, $Q^{\mathcal{A}}$ has no stable models, that is, $\mathcal{A} = \emptyset$). Let us define $A = \{y \in Y \mid \mathcal{A} \models Ky\}$, and $B = \{y \in Y \mid \mathcal{A} \models Ky'\}$. It follows that $Q^{\mathcal{A}}$ consists of the following rules:

1. $y$, for $y \in A$, and $y'$, for $y \in B$
2. $y \wedge y' \rightarrow$ ; and $\neg y \wedge \neg y' \rightarrow$ , for every $y \in Y$
3. $\neg z' \rightarrow z$ ; and $\neg z \rightarrow z'$, for $z \in Z$
4. $\sigma(u_1) \wedge \ldots \wedge \sigma(u_k) \rightarrow w$ , where $u_1 \wedge \ldots \wedge u_k$ is a disjunct of $\Theta$, and $\sigma(\neg a) = a'$ and $\sigma(a) = a$, for every $a \in Y \cup Z$.

Since $\mathcal{A} = \mathcal{ST}(Q^{\mathcal{A}})$ and $\mathcal{A} \neq \emptyset$, $B = Y \setminus A$ (due to clauses of type (2)). It is clear that the program $Q^{\mathcal{A}}$ has stable models and that they are of the form $A \cup \{y' \mid y \in Y \setminus A\} \cup D \cup \{z' \mid z \in Z \setminus D\}$, if that set does not imply $w$ through a rule of type (4), or $A \cup \{y' \mid y \in Y \setminus A\} \cup D \cup \{z' \mid z \in Z \setminus D\} \cup \{w\}$, otherwise, where $D$ is any subset of $Z$. As $\mathcal{A} \models Kw$, there are no stable models of the first type. Thus, the family of stable models of $Q^{\mathcal{A}}$ consists of all sets $A \cup \{y' \mid y \in Y \setminus A\} \cup D \cup \{z' \mid z \in Z \setminus D\} \cup \{w\}$, where $D$ is an arbitrary subset of $Z$. It follows that for every $D \subseteq Z$, the set $A \cup \{y' \mid y \in Y \setminus A\} \cup D \cup \{z' \mid z \in Z \setminus D\}$ satisfies the body of at least one rule of type (4). By the construction, for every $D \subseteq Z$, the valuation of $Y \cup Z$ determined by $A$ and $D$ satisfies the corresponding disjunct in $\Theta$ and so, also $\Theta$. In other words, $\exists Y \forall Z \Theta$ is true.

Conversely, let $\exists Y \forall Z \Theta$ be true. Let $A$ be a subset of $Y$ such that $\Theta_{|Y/A}$ holds for every truth assignment of $Z$ (by $\Theta_{|Y/A}$, we mean the formula obtained by simplifying the formula $\Theta$ with respect to the truth assignment of $Y$ determined by $A$). Let $\mathcal{A}$ consist of all sets of the form $A \cup \{y' \mid y \in Y \setminus A\} \cup D \cup \{z' \mid z \in Z \setminus D\} \cup \{w\}$, where $D \subseteq Z$. It follows that $Q^{\mathcal{A}}$ consists of clauses (1)-(4) above, with $B = Y \setminus A$. Since $\forall Z \Theta_{|A/Y}$ holds, it follows that $\mathcal{A}$ is precisely the set of stable models of $Q^{\mathcal{A}}$. Thus, $\mathcal{A}$ is an epistemic stable model of $Q$. □

In the general case, the complexity goes one level up.

**Theorem 2.** *The problem to decide whether an epistemic program $P \subseteq \mathcal{L}_K$ has an epistemic stable model is $\Sigma_3^P$-complete.*

*Proof:* The membership follows from the earlier remarks. To prove the hardness part, we consider a QBF formula $F = \exists X \forall Y \exists Z \Theta$, where $\Theta$ is a 3-CNF formula. For each atom $x \in X$ ($y \in Y$ and $z \in Z$, respectively), we introduce a fresh atom $x'$ ($y'$ and $z'$, respectively). Finally, we introduce three additional fresh atoms, $w$, $f$ and $g$.

We now construct a disjunctive epistemic program $Q$ by including into it the following clauses:

1. $Kx \rightarrow x$; and $Kx' \rightarrow x'$, for every $x \in X$
2. $x \wedge x' \rightarrow$; and $\neg x \wedge \neg x' \rightarrow$, for every $x \in X$
3. $\neg g \rightarrow f$; and $\neg f \rightarrow g$
4. $f \rightarrow y \vee y'$; and $f \rightarrow z \vee z'$, for every $y \in Y$ and $z \in Z$
5. $f \wedge w \rightarrow z$; and $f \wedge w \rightarrow z'$, for every $z \in Z$
6. $f \wedge \sigma(u_1) \wedge \sigma(u_2) \wedge \sigma(u_3) \rightarrow w$, for every clause $C = u_1 \vee u_2 \vee u_3$ of $\Theta$, where $\sigma(a) = a'$ and $\sigma(\neg a) = a$, for every $a \in X \cup Y \cup Z$
7. $f \wedge \neg w \rightarrow w$
8. $\neg K \neg w \rightarrow$

Let us assume that $\exists X \forall Y \exists Z \Theta$ is true. Let $A \subseteq X$ describe the truth assignment on $X$ so that $\forall Y \exists Z \Theta_{X/A}$ holds (we define $\Theta_{X/A}$ as in the proof of the previous result). We will show that $Q$ has an epistemic stable model $\mathcal{A} = \{A \cup \{a' \mid a \in X \setminus A\} \cup \{g\}\}$. Clearly, $Kx$, $x \in A$, and $Kx'$, $x \in X \setminus A$, are true in $\mathcal{A}$. Also, $K\neg w$ is true in $\mathcal{A}$. All other modal atoms in $Q$ are false in $\mathcal{A}$. Thus, $Q^{\mathcal{A}}$ consists of rules $x$, for $x \in A$, $x'$, for $x \in X \setminus A$ and of rules (2)-(7) above. Let $M$ be a stable model of $Q^{\mathcal{A}}$ containing $f$. It follows that $w \in M$ and so, $Z \cup Z' \subseteq M$. Moreover, the Gelfond-Lifschitz reduct of $Q^{\mathcal{A}}$ with respect to $M$ consists of rules $x$, for $x \in A$, $x'$, for $x \in X \setminus A$, all $\neg$-free constraints of type (2), rule $f$, and rules (4)-(6) above, and $M$ is a minimal model of this program.

Let $B = Y \cap M$. By the minimality of $M$, $M = A \cup \{x' \mid x \in X \setminus A\} \cup B \cup \{y' \mid y \in Y \setminus B\} \cup Z \cup Z' \cup \{f, w\}$. Since $\forall Y \exists Z \Theta_{X/A}$ holds, $\exists Z \Theta_{X/A,Y/B}$ holds, too. Thus, let $D \subseteq Z$ be a subset of $Z$ such that $\Theta_{X/A,Y/B,Z/D}$ is true. It follows that $M' = A \cup \{x' \mid x \in X \setminus A\} \cup B \cup \{y' \mid y \in Y \setminus B\} \cup D \cup \{z' \mid z \in Z \setminus D\} \cup \{f\}$ is also a model of the Gelfond-Lifschitz reduct of $Q^{\mathcal{A}}$ with respect to $M$, contradicting the minimality of $M$.

Thus, if $M$ is an answer set of $Q^{\mathcal{A}}$, it must contain $g$. Consequently, it does not contain $f$ and so no rules of type (4)-(7) contribute to it. It follows that $M = A \cup \{a' \mid a \in X \setminus A\} \cup \{g\}$ and, as it indeed is an answer set of $Q^{\mathcal{A}}$, $\mathcal{A} = \mathcal{ST}(Q^{\mathcal{A}})$. Thus, $\mathcal{A}$ is a epistemic stable model, as claimed.

Conversely, let as assume that $Q$ has an epistemic stable model, say, $\mathcal{A}$. It must be that $\mathcal{A} \models K\neg w$ (otherwise, $Q^{\mathcal{A}}$ contains a contradiction and has no stable models). Let us define $A = \{x \in X \mid \mathcal{A} \models Kx\}$ and $B = \{x \in X \mid \mathcal{A} \models Kx'\}$. It follows that $Q^{\mathcal{A}}$ consists of the clauses:

1. $x$, for $x \in A$ and $x'$, for $x \in B$
2. $x \wedge x' \rightarrow$; and $\neg x \wedge \neg x' \rightarrow$, for every $x \in X$
3. $\neg g \rightarrow f$; and $\neg f \rightarrow g$
4. $f \rightarrow y \vee y'$; and $f \rightarrow z \vee z'$, for every $y \in Y$ and $z \in Z$
5. $f \wedge w \rightarrow z$; and $f \wedge w \rightarrow z'$, for every $z \in Z$
6. $f \wedge \sigma(u_1) \wedge \sigma(u_2) \wedge \sigma(u_3) \rightarrow w$, for every clause $C = u_1 \vee u_2 \vee u_3$ of $\Phi$, where $\sigma(a) = a'$ and $\sigma(\neg a) = a$, for every $a \in X \cup Y \cup Z$.
7. $f, \neg w \rightarrow w$

We have that $\mathcal{A}$ is precisely the set of stable models of this program. Since $\mathcal{A} \neq \emptyset$, $B = X \setminus A$. If $M$ is a stable model of $Q^{\mathcal{A}}$ and contains $f$, then it contains $w$. But then, as $M \in \mathcal{A}$, $\mathcal{A} \not\models K\neg w$, a contradiction. It follows that there is no stable model containing $f$. That is, the program consisting of the following rules has no stable model:

1. $x$, for $x \in A$ and $x'$, for $x \in X \setminus A$
2. $y \vee y'$; and $z \vee z'$, for every $y \in Y$ and $z \in Z$
3. $w \rightarrow z$; and $w \rightarrow z'$, for every $z \in Z$
4. $\sigma(u_1) \wedge \sigma(u_2) \wedge \sigma(u_3) \rightarrow w$, for every clause $C = u_1 \vee u_2 \vee u_3$ of $\Theta$, where $\sigma(a) = a'$ and $\sigma(\neg a) = a$, for every $a \in X \cup Y \cup Z$.
5. $\neg w \rightarrow w$

But then, the formula $\forall Y \exists Z \Theta_{|X/A}$ is true and, consequently, the formula $\exists X \forall Y \exists Z \Theta$ is true, too. □

For the other two epistemic semantics, Remark 1 implies that the problem of the existence of an epistemic model (epistemic supported model) is in the class $\Sigma_2^P$. The $\Sigma_2^P$-hardness of the problem can be proved by similar techniques as those we used for the case of epistemic stable models. Thus, we have the following result.

**Theorem 3.** *The problem to decide whether an epistemic program $P \subseteq \mathcal{L}_K$ has an epistemic model (epistemic supported model, respectively) is $\Sigma_2^P$-complete.*

## 5 Modeling with Epistemic Programs

We will now present several problems which illustrate the advantages offered by the language of epistemic programs we developed in the previous two sections. Whenever we use predicate programs, we understand that their semantics is that of the corresponding ground programs.

First, we consider two graph problems related to the existence of Hamiltonian cycles. Let $G$ be a directed graph. An edge in $G$ is *critical* if it belongs to every hamiltonian cycle in $G$. The following problems are of interest:

1. Given a directed graph $G$, find the set of all critical edges of $G$
2. Given a directed graph $G$, and integers $p$ and $k$, find a set $R$ of no more than $p$ new edges such that $G \cup R$ has no more than $k$ critical edges.

Let $HC(vtx, edge)$ be any standard ASP encoding of the Hamiltonian cycle problem, in which predicates $vtx$ and $edge$ represent $G$, and a predicate $hc$ represents edges of a candidate hamiltonian cycle. We assume the rules of $HC(vtx, edge)$ are written from left to right so that they can be regarded as elements of $\mathcal{L}$. Then, simply adding to $HC(vtx, edge)$ the rule:

$$Khc(X, Y) \rightarrow critical(X, Y)$$

yields a correct representation of the first problem. We write $HC_{cr}(vtx, edge)$ to denote this program. Also, for a directed graph $G = (V, E)$, we define

$$D = \{vtx(v) \mid v \in V\} \cup \{edge(v, w) \mid (v, w) \in E\}.$$

We have the following result.

**Theorem 4.** *Let $G = (V, E)$ be a directed graph. If $HC_{cr}(vtx, edge) \cup D$ has no epistemic stable models, then every edge in G is critical (trivially). Otherwise, the epistemic program $HC_{cr}(vtx, edge) \cup D$ has a unique epistemic stable model $\mathcal{A}$ and the set $\{(v, w) \mid \mathcal{A} \models critical(u, v)\}$ is the set of critical edges in G.*

*Proof (Sketch):* Let $H$ be the grounding of $HC_{cr}(vtx, edge) \cup D$. If $H$ has no epistemic stable models, it follows that the "non-epistemic" part $H'$ of $H$ has no stable models (as no atom of the form $critical(x, y)$ appears in it). As $H'$ encodes the existence of a hamiltonian cycle in $G$, it follows that $G$ has no Hamiltonian cycles. Thus, trivially, every edge of $G$ belongs to every Hamiltonian cycle of $G$ and so, every edge of $G$ is critical.

Thus, let us assume that $\mathcal{A}$ is an epistemic stable model of $H$. Also, let $S$ be the set of all stable models of $H'$ (they correspond to Hamiltonian cycles of $G$; each model contains, in particular, atoms of the form $hc(x, y)$, where $(x, y)$ ranges over the edges of the corresponding Hamiltonian cycle). The reduct $H^{\mathcal{A}}$ consists of $H'$ (non-epistemic part of $H$ is unaffected by the reduct operation) and of $C'$, a set of some facts of the form $critical(x, y)$. Thus, the stable models of the reduct are of the form $M \cup C'$, where $M \in S$. That is, $\mathcal{A} = \{M \cup C' \mid M \in S\}$. Let us denote by $C$ the set of the atoms $critical(x, y)$, where $(x, y)$ belongs to every hamiltonian cycle of $G$ (is critical). One can compute now that $H^{\mathcal{A}} = H' \cup C$. Since $\mathcal{A} = \mathcal{ST}(H^{\mathcal{A}})$, $\mathcal{A} = \{M \cup C \mid M \in S\}$.

Thus, $HC_{cr}(vtx, edge) \cup D$ has a unique epistemic stable model, as claimed. It also follows that the set $\{(v, w) \mid \mathcal{A} \models critical(u, v)\}$ is the set of critical edges in $G$. $\square$

To represent the second problem, we proceed as follows. First, we "select" new edges to be added to the graph and impose constraints that guarantee that all new edges are indeed new, and that no more than $p$ new edges are selected (we use here *lparse* syntax for brevity; the constraint can be encoded strictly in the language $\mathcal{L}_K$).

$$\begin{array}{l} vtx(X) \wedge vtx(Y) \rightarrow newEdge(X, Y) \\ newEdge(X, Y) \wedge edge(X, Y) \rightarrow \bot \\ (p+1)\{newEdge(X, Y) : vtx(X), vtx(Y)\} \rightarrow \bot \\ KnewEdge(X, Y) \wedge \neg newEdge(X, Y) \rightarrow \bot \\ \neg KnewEdge(X, Y) \wedge newEdge(X, Y) \rightarrow \bot. \end{array}$$

Next, we define the set of edges of the extended graph, using a predicate $edgeEG$:

$$\begin{array}{l} edge(X, Y) \rightarrow edgeEG(X, Y) \\ newEdge(X, Y) \rightarrow edgeEG(X, Y) \end{array}$$

Finally, we define critical edges and impose a constraint on their number (again, exploiting the *lparse* syntax for brevity sake):

$$\begin{array}{l} edgeEG(X, Y) \wedge Khc(X, Y) \rightarrow critical(X, Y) \\ (k+1)\{critical(X, Y) : edgeEG(X, Y)\} \rightarrow \bot. \end{array}$$

We define $Q$ to consist of all these rules together with all the rules of the program $HC(vtx, edgeEG)$. We now have the following theorem. The proof is similar to that above and so we omit it.

**Theorem 5.** *Let $G$ be a directed graph. There is an extension of $G$ with no more than $p$ new edges so that the resulting graph has no more than $k$ critical edges if and only if the program $Q \cup D$ has an epistemic stable model.*

For another example we consider the unique model problem: given a CNF formula $F$, the goal is to decide whether $F$ has a unique minimal model. The unique model problem was also considered by Faber and Woltran [4, 5]. We will show two encodings of the problem by means of epistemic programs. The first one uses the semantics of epistemic models and is especially direct. The other one uses the semantics of epistemic stable models.

Let $F$ be a propositional theory consisting of constraints $L_1 \wedge \ldots \wedge L_k \rightarrow \bot$, where $L_i$'s are literals. Any propositional theory can be rewritten into an equivalent theory of such form. We denote by $F^K$ the formula obtained from $F$ by replacing every atom $x$ with the modal atom $Kx$.

**Theorem 6.** *For every theory $F \subseteq \mathcal{L}$ consisting of constraints, $F$ has a least model if and only if the epistemic program $F \cup F^K$ has an epistemic model.*

*Proof:* Let us assume that $F$ has a least model. We define $\mathcal{A}$ to consist of all models of $F$, and we denote the least model of $F$ by $M$. We will show that $\mathcal{A}$ is an epistemic model of $F \cup F^K$. Clearly, for every $x \in M$, $\mathcal{A} \models Kx$. Similarly, for every $x \notin M$, $\mathcal{A} \models \neg Kx$. Thus, $[F^K]^{\mathcal{A}} = \emptyset$. Consequently, $[F \cup F^K]^{\mathcal{A}} = F$ and so, $\mathcal{A}$ is precisely the set of all models of $[F \cup F^K]^{\mathcal{A}}$. Thus, $\mathcal{A}$ is an epistemic model.

Conversely, let $\mathcal{A}$ be an epistemic model of $F \cup F^K$. It follows that $[F^K]^{\mathcal{A}} = \emptyset$ (otherwise, $[F \cup F^K]^{\mathcal{A}}$ contains $\bot$ and $\mathcal{A}$ would have to be empty, contradicting the definition of an epistemic model). Thus, $[F \cup F^K]^{\mathcal{A}} = F$ and consequently, $\mathcal{A}$ is the set of all models of $F$. Let $M = \{x \in At \mid \mathcal{A} \models Kx\}$ and let

$$a_1 \wedge \ldots \wedge a_m \wedge \neg b_1 \wedge \ldots \wedge \neg b_n \rightarrow \bot \tag{3}$$

be a rule in $F$. Then,

$$Ka_1 \wedge \ldots \wedge Ka_m \wedge \neg Kb_1 \wedge \ldots \wedge \neg Kb_n \rightarrow \bot$$

is a rule in $F^K$. As $[F^K]^{\mathcal{A}} = \emptyset$,

$$\mathcal{A} \not\models Ka_1 \wedge \ldots \wedge Ka_m \wedge \neg Kb_1 \wedge \ldots \wedge \neg Kb_n.$$

Thus, for some $i$, $1 \leq i \leq m$, $\mathcal{A} \not\models Ka_i$, or for some $j$, $1 \leq j \leq n$, $\mathcal{A} \models Kb_j$. In the first case, $a_i \notin M$, in the latter, $b_j \in M$. In either case, $M$ is a model of rule (3). It follows that $M$ is a model of $F$. Let $M'$ be a model of $F$. Then $M' \in \mathcal{A}$ and, by the definition of $M$, $M \subseteq M'$. Thus, $M$ is a least model of $F$. □

Next, we will encode the same problem as an epistemic program under the epistemic stable model semantics. The idea is quite similar. We only need to add rules to generate all candidate models.

**Theorem 7.** *For every theory $F \subseteq \mathcal{L}$ consisting of constraints, $F$ has a least model if and only if the epistemic program*

$$F \cup F^K \cup \{\neg x \rightarrow x' \mid x \in At\} \cup \{\neg x' \rightarrow x \mid x \in At\}$$

*has an epistemic stable model.*

We note that an even simpler encoding can be obtained if we use *lparse* choice rules. In this case, we can replace $\{\neg x \rightarrow x' \mid x \in At\} \cup \{\neg x' \rightarrow x \mid x \in At\}$ with $\{\{x\} \mid x \in At\}$.

## 6 Connection to Gelfond's Epistemic Programs

We will now return to the original formalism of epistemic specifications proposed by Gelfond [3] (under the restriction to epistemic programs we discussed here). We will show that it can be expressed in a rather direct way in terms of our epistemic programs in the two-valued setting and under the epistemic supported-model semantics.

The reduction we are about to describe is similar to the well-known one used to eliminate the "strong" negation from disjunctive logic programs with strong negation. In particular, it requires an extension to the language $\mathcal{L}$. Specifically, for every atom $x \in At$ we introduce a fresh atom $x'$ and we denote the extended language by $\mathcal{L}'$. The intended role of $x'$ is to represent in $\mathcal{L}'$ the literal $\neg x$ from $\mathcal{L}$. Building on this idea, we assign to each set $W$ of literals in $\mathcal{L}$ the set

$$W' = (W \cap At) \cup \{x' \mid \neg x \in W\}.$$

In this way, sets of literals from $\mathcal{L}$ (in particular, three-valued interpretations of $\mathcal{L}$) are represented as sets of atoms from $\mathcal{L}'$ (two-valued interpretations of $\mathcal{L}'$).

We now note that the truth and falsity of a formula form $\mathcal{L}$ under a three-valued interpretation can be expressed as the truth and falsity of certain formulas from $\mathcal{L}'$ in the two-valued setting. The following result is well known.

**Proposition 2.** *For every formula $\varphi \in \mathcal{L}$ there are formulas $\varphi^-, \varphi^+ \in \mathcal{L}'$ such that for every set of literals $W$ (in $\mathcal{L}$)*

1. $v_W(\varphi) = \mathbf{t}$ *if and only if* $u_{W'}(\varphi^+) = \mathbf{t}$
2. $v_W(\varphi) = \mathbf{f}$ *if and only if* $u_{W'}(\varphi^-) = \mathbf{f}$

*Moreover, the formulas $\varphi^-$ and $\varphi^+$ can be constructed in polynomial time with respect to the size of $\varphi$.*

*Proof:* This a folklore result. We provide a sketch of a proof for the completeness sake. We define $\varphi^+$ and $\varphi^-$ by recursively as follows:

1. $x^+ = x$ and $x^- = \neg x'$, if $x \in At$
2. $(\neg\varphi)^+ = \neg\varphi^-$ and $(\neg\varphi)^- = \neg\varphi^+$
3. $(\varphi \vee \psi)^+ = \varphi^+ \vee \psi^+$ and $(\varphi \vee \psi)^- = \varphi^- \vee \psi^-$; the case of the conjunction is dealt with analogously
4. $(\varphi \rightarrow \psi)^+ = \varphi^- \rightarrow \psi^+$ and $(\varphi \rightarrow \psi)^- = \varphi^+ \rightarrow \psi^-$.

One can check that formulas $\varphi^+$ and $\varphi^-$ defined in this way satisfy the assertion. □

We will now define the transformation $\sigma$ that allows us to eliminate strong negation. First, for a literal $L \in \mathcal{L}$, we now define

$$\sigma(L) = \begin{cases} x & \text{if } L = x \\ x' & \text{if } L = \neg x \end{cases}$$

Furthermore, if $E$ is a simple modal literal or its default negation, we define

$$\sigma(E) = \begin{cases} K\varphi^+ & \text{if } E = K\varphi \\ \neg K\varphi^- & \text{if } E = \neg K\varphi \\ \neg K\varphi^+ & \text{if } E = \mathit{not}\ K\varphi \\ K\varphi^- & \text{if } E = \mathit{not}\ \neg K\varphi \end{cases}$$

and for an epistemic premise $E = E_1, \ldots, E_t$ (where each $E_i$ is a simple modal literal or its default negation) we set

$$\sigma(E) = \sigma(E_1) \wedge \ldots \wedge \sigma(E_t).$$

Next, if $r$ is an epistemic rule

$$L_1 \vee \ldots \vee L_k \leftarrow F_1, \ldots, F_m, \mathit{not}\ F_{m+1}, \ldots, \mathit{not}\ F_n, E$$

we define

$$\sigma(r) = \sigma(E) \wedge \sigma(F_1) \wedge \ldots \wedge \sigma(F_m) \wedge \neg\sigma(F_{m+1}) \wedge \ldots \wedge \neg\sigma(F_n) \rightarrow \sigma(L_1) \vee \ldots \vee \sigma(L_k).$$

Finally, for an epistemic program $P$, we set

$$\sigma(P) = \{\sigma(r) \mid r \in P\}) \cup \{x \wedge x' \rightarrow \bot\}.$$

We note that $\sigma(P)$ is indeed an epistemic program in the language $\mathcal{L}_K$ (according to our definition of epistemic programs). The role of the rules $x \wedge x' \rightarrow \bot$ is to ensure that sets forming epistemic (stable, supported) models of $\sigma(P)$ correspond to consistent sets of literals (the only type of set of literals allowed in world views).

Given a three-valued possible structure $\mathcal{A}$, we define $\mathcal{A}' = \{W' \mid W \in \mathcal{A}\}$, and we regard $\mathcal{A}'$ as a two-valued possible-world structure. We now have the following theorem.

**Theorem 8.** *Let $P$ be an epistemic program according to Gelfond. Then a three-valued possible-world structure $\mathcal{A}$ is a world view of $P$ if and only if a two-valued possible-world structure $\mathcal{A}'$ is an epistemic supported model of $\sigma(P)$.*

*Proof (Sketch):* Let $P$ be an epistemic program according to Gelfond, $\mathcal{A}$ a possible-world structure and $W$ a set of literals. We first observe that the G-reduct $P^{\langle \mathcal{A}, W\rangle}$ can be described as the result of a certain two-step process. Namely, we define the *epistemic reduct* of $P$ with respect to $\mathcal{A}$ to be the disjunctive logic program $P^{\mathcal{A}}$ obtained from $P$ by removing every rule whose epistemic premise $E$ satisfies $\mathcal{A} \not\models E$, and by removing the epistemic premise from every other rule in $P$. This construction is the three-valued counterpart to the one we employ in our approach. It is clear that the epistemic reduct of $P$ with respect to $\mathcal{A}$, with some abuse of notation we will denote it by $P^{\mathcal{A}}$, is a disjunctive logic program with strong negation.

Let $Q$ be a disjunctive program with strong negation and $W$ a set of literals. By the *supp-reduct* of $Q$ with respect to $W$, $R^{sp}(Q, W)$, we mean the set of the heads of all rules whose bodies are satisfied by $W$ (which in the three-valued setting means that every literal in the body not in the scope of $not$ is in $W$, and every literal in the body in the scope of $not$ is not in $W$). A consistent set $W$ of literals is a supported answer set of $Q$ if $W \in Min(R^{sp}(Q, W))$ (this is a natural extension of the definition of a supported model [8, 9] to the case of disjunctive logic programs with strong negation; again, we do not regard inconsistent sets of literals as supported answer sets).

Clearly, $P^{\langle \mathcal{A}, W\rangle} = R^{sp}(P^{\mathcal{A}}, W)$. Thus, $\mathcal{A}$ is a world view of $P$ according to the definition by Gelfond if and only if $\mathcal{A}$ is a collection of all supported answer sets of $P^{\mathcal{A}}$.

We also note that by Proposition 2, if $E$ is an epistemic premise, then $\mathcal{A} \models E$ if and only if $\mathcal{A}' \models \sigma(E)$. It follows that $\sigma(P^{\mathcal{A}}) = \sigma(P)^{\mathcal{A}'}$. In other words, constructing the epistemic reduct of $P$ with respect to $\mathcal{A}$ and then translating the resulting disjunctive logic program with strong negation into the corresponding disjunctive logic program without strong negation yields the same result as first translating the epistemic program (in the Gelfond's system) into our language of epistemic programs and then computing the reduct with respect to $\mathcal{A}'$. We note that there is a one-to-one correspondence between supported answer sets of $P^{\mathcal{A}}$ and supported models of $\sigma(P^{\mathcal{A}})$ ($\sigma$, when restricted to programs consisting of rules without epistemic premises, is the standard transformation eliminating strong negation and preserving the stable and supported semantics). Consequently, there is a one-to-one correspondence between supported answer sets of

$P^{\mathcal{A}}$ and supported models of $\sigma(P)^{\mathcal{A}'}$ (cf. our observation above). Thus, $\mathcal{A}$ consists of supported answer sets of $P^{\mathcal{A}}$ if and only if $\mathcal{A}'$ consists of supported models of $\sigma(P)^{\mathcal{A}'}$. Consequently, $\mathcal{A}$ is a world view of $P$ if and only if $\mathcal{A}'$ is an epistemic supported model of $\sigma(P)$. □

## 7 Epistemic Models of Arbitrary Theories

So far, we defined the notions of epistemic models, epistemic stable models and epistemic supported models only for the case of epistemic programs. However, this restriction is not essential. We recall that the definition of these three epistemic semantics consists of two steps. The first step produces the reduct of an epistemic program $P$ with respect to a possible-world structure, say $\mathcal{A}$. This reduct happens to be (modulo a trivial syntactic transformation) a standard disjunctive logic program in the language $\mathcal{L}$ (no modal atoms anymore). If the set of models (respectively, stable models, supported models) of the reduct program coincides with $\mathcal{A}$, $\mathcal{A}$ is an epistemic model (respectively, epistemic stable or supported model) of $P$. However, the concepts of a model, stable model and supported model are defined for *arbitrary* theories in $\mathcal{L}$. This is obviously well known for the semantics of models. The stable-model semantics was extended to the full language $\mathcal{L}$ by Ferraris [12] and the supported-model semantics by Truszczynski [13]. Thus, there is no reason precluding the extension of the definition of the corresponding epistemic types of models to the general case. We start be generalizing the concept of the reduct.

**Definition 5.** *Let $T$ be an arbitrary theory in $\mathcal{L}_K$ and let $\mathcal{A}$ be a possible-world structure. The* epistemic reduct *of $T$ with respect to $\mathcal{A}$, $T^{\mathcal{A}}$ in symbols, is the theory obtained from $T$ by replacing each maximal modal atom $K\varphi$ with $\top$, if $\mathcal{A} \models K\varphi$, and with $\bot$, otherwise.*

We note that if $T$ is an epistemic program, this notion of the reduct does not coincide with the one we discussed before. Indeed, now no rule is dropped and no modal literals are dropped; rather modal atoms are replaced with $\top$ and $\bot$. However, the replacements are executed in such a way as to ensure the same behavior. Specifically, one can show that models, stable models and supported models of the two reducts coincide.

Next, we generalize the concepts of the three types of epistemic models.

**Definition 6.** *Let $T$ be an arbitrary theory in $\mathcal{L}_K$. A possible-world structure $\mathcal{A}$ is an* epistemic model *(respectively, an* epistemic stable model*, or an* epistemic supported model*) of $P$, if $\mathcal{A}$ is the set of models (respectively, stable models or supported models) of $\mathcal{M}(P^{\mathcal{A}})$.*

From the comments we made above, it follows that if $T$ is an epistemic program, this more general definition yields the came notions of epistemic models of the three types as the earlier one.

We note that even in the more general setting the complexity of reasoning with epistemic (stable, supported) models remains unchanged. Specifically, we have the following result.

**Theorem 9.** *The problem to decide whether an epistemic theory $T \subseteq \mathcal{L}_K$ has an epistemic stable model is $\Sigma_3^P$-complete. The problem to decide whether an epistemic theory $T \subseteq \mathcal{L}_K$ has an epistemic model (epistemic supported model, respectively) is $\Sigma_2^P$-complete.*

*Proof(Sketch):* The hardness part follows from our earlier results concerning epistemic programs. To prove membership, we modify Proposition 1, and show a polynomial time algorithm with a $\Sigma_2^P$ oracle (NP oracle for the last two problems) that decides, given a propositional theory $S$ and a modal formula $K\varphi$ (with $\varphi \in \mathcal{L}_K$ and not necessarily in $\mathcal{L}$) whether $\mathcal{ST}(S) \models K\varphi$ (respectively, $\mathcal{M}(S) \models K\varphi$, or $\mathcal{SP}(S) \models K\varphi$). $\square$

## 8 Discussion

In this paper, we proposed a two-valued formalism of epistemic theories — subsets of the language of modal propositional logic. We proposed a uniform way, in which semantics of propositional theories (the classical one as well as nonmonotonic ones: stable and supported) can be extended to the case of epistemic theories. We showed that the semantics of epistemic supported models is closely related to the original semantics of epistemic specifications proposed by Gelfond. Specifically we showed that the original formalism of Gelfond can be expressed in a straightforward way by means of epistemic programs in our sense under the semantics of epistemic supported models. Essentially all that is needed is to use fresh symbols $x'$ to represent strong negation $\neg x$, and use the negation operator of our formalism, $\varphi \rightarrow \bot$ or, in the shorthand, $\neg\varphi$, to model the default negation $not\ \varphi$.

We considered in more detail the three semantics mentioned above. However, other semantics may also yield interesting epistemic counterparts. In particular, it is clear that Definition 6 can be used also with the minimal model semantics or with the Faber-Leone-Pfeifer semantics [14]. Each semantics gives rise to an interesting epistemic formalism that warrants further studies.

In logic programming, eliminating strong negation does not result in any loss of the expressive power but, at least for the semantics of stable models, disjunctions cannot be compiled away in any concise way (unless the polynomial hierarchy collapses). In the setting of epistemic programs, the situation is similar. The strong negation can be compiled away. But the availability of disjunctions in the heads and the availability of epistemic premises in the bodies of rules are essential. Each of these factors separately brings the complexity one level up. Moreover, when used together under the semantics of epistemic stable models they bring the complexity two levels up. This points to the intrinsic importance of having in a knowledge representation language means to represent indefiniteness in terms of disjunctions, and what is known to a program (theory) — in terms of a modal operator $K$.

## Acknowledgments

This work was partially supported by the NSF grant IIS-0913459.

## References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
2. Gelfond, M., Lifschitz, V.: The stable semantics for logic programs. In: Proceedings of the 5th International Conference on Logic Programming (ICLP 1988), pp. 1070–1080. MIT Press, Cambridge (1988)
3. Gelfond, M.: Strong introspection. In: Proceedings of AAAI 1991, pp. 386–391 (1991)
4. Faber, W., Woltran, S.: Manifold answer-set programs for meta-reasoning. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 115–128. Springer, Heidelberg (2009)
5. Faber, W., Woltran, S.: Manifold answer-set programs and their applications. In: Balduccini, M., Son, T.C. (eds.) Gelfond Festschrift. LNCS (LNAI), vol. 6565, pp. 44–63. Springer, Heidelberg (2011)
6. Marek, W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM 38, 588–619 (1991)
7. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: propositional case. Annals of Mathematics and Artificial Intelligence 15, 289–323 (1995)
8. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 89–142. Morgan Kaufmann, San Francisco (1988)
9. Baral, C., Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming 19/20, 73–148 (1994)
10. Brass, S., Dix, J.: Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. Journal of Logic Programming 32(3), 207–228 (1997)
11. Inoue, K., Sakama, C.: Negation as failure in the head. Journal of Logic Programming 35, 39–78 (1998)
12. 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)
13. Truszczynski, M.: Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. Artificial Intelligence (2010) (in press), available through Science Direct at `http://dx.doi.org/10.1016/j.artint.2010.08.004`
14. 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)

# Answer Set; Programming?*

Pedro Cabalar

Dept. Computación,
University of Corunna, Spain
cabalar@udc.es

**Abstract.** Motivated by a discussion maintained by Michael Gelfond and other researchers, this short essay contains some thoughts and refections about the following question: should ASP be considered a programming language?

## 1 Introduction

During a break in the Commonsense[1] 2007 Symposium, sat around a table in some cafeteria inside the Stanford campus, an interesting and solid debate between Michael Gelfond and some well-known researcher (call him A.B.) in the area of Logic Programming and Nonmonotonic Reasoning was initiated. The discussion started when A.B. claimed at some point that Answer Set Programming (ASP) was a *programming* paradigm. Although at that moment, this seemed a quite obvious, redundant and harmless assertion, surprisingly, Gelfond's reaction was far from agreement. With his usual kind but firm, rational style, he proceeded to refute that argument, defending the idea that ASP was exclusively a logical knowledge representation language, *not* a programming language. This short essay contains some disconnected, personal thoughts and reflections motivated by that discussion.

## 2 Programming as Implementing Algorithms

Of course, the keypoint relies on what we consider to be a *programming language* or more specifically which is the task of *programming*. In its most frequent use, this term refers to *imperative* programming, that is, specifying sets of instructions to be executed by a computer in order to achieve some desired goal. This is opposed to *declarative* programming, which is frequently defined as specifying *what* is to be considered a solution to a given problem, rather than the steps describing *how* to achieve that solution. ASP would perfectly fit in this second definition, had not it been the case that declarative programming actually involves much more than this ideal goal of pure formal specification.

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

[1] Eighth International Symposium on Logical Formalizations of Commonsense Reasoning.

In practice, despite of the big differences between imperative and declarative programming languages, we can generally identify *programming* as *implementing algorithms.* We actually expect that a programming language allows us a way to execute algorithms on a computer. When a (declarative) programmer designs a program, either in Prolog or in some functional language, she must not only write a formal specification (using predicates, functions, equations, etc), but also be aware of how this specification is going to be processed by the language interpreter she is using, so that both things together become the algorithm implementation. This idea was clearly expressed by Kowalski's equation [7] "Algorithm = Logic + Control" or $A = L + C$, meaning that a given algorithm $A$ can be obtained as the sum of a logical description $L$ (our formal specification) plus a control strategy $C$ (top-down, bottom-up, SLD, etc). The same algorithm $A$ can be the result of different combinations, say $A = L_1 + C_1$ or $A = L_2 + C_2$. Efficiency issues and even termination will strongly depend on the control component $C$ we choose. If we use an interpreter like Prolog, where the control component is *fixed*, we will face the typical situation where the logic component $L$ must be adapted to the control strategy. A simple change in the ordering among clauses or literals in $L$ may make the program to terminate or not.

Now, back to our concern for ASP, despite of being an LP paradigm, it seems evident that the algorithmic interpretation for Predicate Logic [16] (which was in the very roots of the LP area) is not applicable here. In fact, in an ASP program, there is no such a thing as a "control strategy." In this sense, we reach a first curious, almost paradoxical, observation:

**Observation 1.** *ASP is a purely declarative language, in the sense that it exclusively involves formal specification, but cannot implement an algorithm, that is, cannot execute a program!*

Thus, ASP fits so well with the ideal of a declarative programming language (i.e., telling *what* and not *how*) that it does not allow programming at all.

## 3 Programming as Temporal Problem Solving

The previous observation saying that ASP cannot implement an algorithm may seem too strong. One may object that ASP has been extensively used for solving temporal problems in transition systems, including planning, that is, obtaining a sequence of actions to achieve a desired goal. This may look close to algorithm implementation. Let us illustrate the idea with a well-known example.

*Example 1 (Towers of Hanoi).* We have three vertical pegs $a, b, c$ standing on a horizontal base and a set of $n$ holed disks, numbered $1, \dots, n$, whose sizes are proportional to the disk numbers. Disks must always be stored on pegs, and a larger disk cannot rest on a smaller one. The HANOI problem consists in moving a tower formed with the $n$ disks in peg $a$ to peg $c$, by combination of individual movements that can carry the top disk of some peg on top of another peg. $\square$

Figure 1 shows an ASP program for solving HANOI in the language of `lparse`[2]. Constant `n` represents the number of disks and constant `pathlength` the number of transitions, so that, we must make iterative calls to the solver varying `pathlength=0,1,2,`... until a solution is found. For instance, for `n=2` the first solution is found with `pathlength=3` and consists in the sequence of actions `move(a,b,0) move(a,c,1) move(b,c,2)`; for `n=3`, the minimal solution is `pathlength=7` obtaining the sequence:

```
move(a,c,0) move(a,b,1) move(c,b,2) move(a,c,3) move(b,a,4)
move(b,c,5) move(a,c,6)
```

whereas for `n=3` with `pathlength=15` we get:

```
move(a,b,0) move(a,c,1) move(b,c,2) move(a,b,3) move(c,a,4)
move(c,b,5) move(a,b,6) move(a,c,7) move(b,c,8) move(b,a,9)
move(c,a,10) move(b,c,11) move(a,b,12) move(a,c,13) move(b,c,14)
```

Although the ASP program correctly solves the HANOI problem, it is far from being an algorithm. In particular, *we would expect that an algorithm told us how to proceed in the general case for an arbitrary number $n$ of disks.* This means finding some general process from which the above sequences of actions can be extracted once we fix the parameter $n$. For instance, in the sight of the three solved instances above, it is not trivial at all how to proceed for $n = 4$.

In the case of HANOI, such a general process to generate an arbitrary solution does exist. In fact, the HANOI problem is a typical example extensively used in programming courses to illustrate the idea of a recursive algorithm. We can divide the task of shifting $n$ disks from peg $X$ to peg $Y$, using $Aux$ as an auxiliary peg, into the general steps

1. Shift $n - 1$ disks from $X$ to $Aux$ using $Y$ as auxiliary peg;
2. Move the $n$-th disk from $X$ to $Y$;
3. Shift $n - 1$ disks from $Aux$ to $Y$ using $X$ as auxiliary peg.

This recursive algorithm is encoded in Prolog in Figure 2, where predicate `hanoi(N,Sol)` gets the number of disks `N` and returns the solution `Sol` as a list of movements to be performed.

Note now the huge methodological difference between both programs. On the one hand, the ASP program exclusively contains a formal description of HANOI but cannot tell us how to proceed in a general case, that is, cannot yield us a general pattern for the sequences of actions for an arbitrary $n$. On the other hand, the Prolog program (or any implementation[3] of the recursive algorithm) tells us the steps that must be performed in a compact (and in fact, much more efficient) way, but contains *no information at all* about the original problem. The recursive algorithm is just a "solution generator" or if preferred, a regular way of describing the structure of sequences of actions that constitute a solution.

[2] `http://www.tcs.hut.fi/Software/smodels/lparse.ps`

[3] We could perfectly use instead any programming language allowing recursive calls. In the same way, Prolog can also be used to implement a planner closer to the constraint-solving spirit of the ASP program.

```
%---- Types and typed variables
disk(1..n).                    peg(a;b;c).
transition(0..pathlength-1).  situation(0..pathlength).
location(Peg) :- peg(Peg).    location(Disk) :- disk(Disk).
#domain disk(X;Y).     #domain peg(P;P1;P2).  #domain transition(T).
#domain situation(I).  #domain location(L;L1).

%---- Inertial fluent: on(X,L,I) = disk X is on location L at time I
on(X,L,T+1) :- on(X,L,T), not otherloc(X,L,T+1).   % inertia
otherloc(X,L,I) :- on(X,L1,I), L1!=L.
:- on(X,L,I), on(X,L1,I), L!=L1.                    % on unique location

%---- Defined fluents
%   inpeg(L,P,I) = location L is in peg P at time I
%   top(P,X,I) = location L is the top of peg P. If empty, the top is P
inpeg(P,P,I).
inpeg(X,P,I) :- on(X,L,I), inpeg(L,P,I).
top(P,L,I) :- inpeg(L,P,I), not covered(L,I).
covered(L,I) :- on(X,L,I).

%---- State constraint: no disk X on a smaller one
:- on(X,Y,I), X>Y.

%---- Effect axiom
on(X,L,T+1) :- move(P1,P2,T), top(P1,X,T), top(P2,L,T).

%---- Executability constraint
:- move(P1,P2,T), top(P1,P1,T).     % the source peg cannot be empty

%---- Generating actions
movement(P1,P2) :- P1 != P2.        % valid movements
1 {move(A,B,T) : movement(A,B) } 1. % pick one at each transition T

%---- Initial situation
on(n,a,0).         on(X,X+1,0) :- X<n.

%---- Goal: at last situation, all disks in peg c
onewrong :- not inpeg(X,c,pathlength).
:- onewrong.
```

**Fig. 1.** An ASP program to solve the HANOI problem

This naturally leads to the following topic: *verification*. How can we guarantee that the actions recursively generated for some $n$ actually move our tower to the desired target without violating the problem constraints? Verifying the recursive algorithm is a crucial work, since it contains no information on the original puzzle. In the case of the ASP encoding, since it already constitutes a formal specification, verification is a much more subtle task. It would require providing

```
hanoi(N,Sol) :- shift(N,a,c,b,Sol).

shift(0,_,_,_,[]) :- !.
shift(N,X,Y,Aux,Sol) :- N1 is N-1,
                        shift(N1,X,Aux,Y,Pre),
                        append(Pre,[move(X,Y)|Post],Sol),
                        shift(N1,Aux,Y,X,Post).
```

**Fig. 2.** A Prolog program implementing a recursive algorithm to solve HANOI

a second, different enough, formal specification (perhaps using mathematical objects closer to the original problem) and proving afterwards that the answer sets we obtain are in one-to-one correspondence to the solutions of our second representation.

Together with verification, another typical issue in algorithm analysis is *complexity.* The recursive algorithm can be easily used to prove that a solution to HANOI requires $2^n - 1$ steps. Obtaining this complexity result from the ASP program (plus the iteration of `pathlength`) is far from trivial. Although the complexity of arbitrary ASP is well-known, a different open and interesting question is *how to use ASP for complexity analysis of a given encoded temporal problem.*

One final comment that stresses the difference between both programs is that, obviously, the recursive algorithm approach is not elaboration tolerant at all. A simple variation of HANOI that allowed a fourth peg would lead to shorter solutions (for instance, with $n = 4$ a solution is found in 9 steps). Note that the only change in the ASP representation would just require adding the fact `peg(d)`. It is easy to think about simple variations that would even make the Prolog program to become incorrect.

## 4 Programming as Implementing a Turing Machine

One of the usually desirable properties of programming languages is *Turing completeness*, that is, the capability of capturing any computation that can be performed by a Turing machine. It is well-known [15] that Prolog (in fact Horn clauses with functions) is Turing complete. Figure 3 shows one possible encoding of a generic Turing machine in Prolog. It just consists of five rules for predicate `tm(S,L,X,R)` which represents a machine configuration. The current state is represented by argument `S` and the tape is fragmented into three portions: `L` is the (reversed) list of symbols to the left of the machine head; `X` is the tape symbol currently pointed by the head; and `R` is the list of symbols to the right of the head. Constant `0` stands for the blank symbol. Predicate `nexttm` is just a recursive call to `tm` preceded by a display output of the new configuration.

In order to implement a particular machine, we just have to include facts for the predicates `t/5` (the transition table), and `final/1` (which states are final), assuming that no transition is given for a final state. As an example, the following facts would specify a 3 state, 2 symbol busy beaver:

```
tm(S, _, _, _)       :- final(S).
tm(S,[Y|L],X,  R  ) :- t(S,X,S1,X1,l), nexttm(S1,L,Y,[X1|R]).
tm(S,  L,  X,[Y|R]) :- t(S,X,S1,X1,r), nexttm(S1,[X1|L],Y,R).
tm(S,  L,  X,  [] ) :- t(S,X,S1,X1,r), nexttm(S1,[X1|L],0,[]).
tm(S,  [], X,  R  ) :- t(S,X,S1,X1,l), nexttm(S1,[],0,[X1|R]).

nexttm(S,L,X,R) :- write(tm(S,L,X,R)),nl,tm(S,L,X,R).
```

**Fig. 3.** A general implementation of a Turing machine in Prolog

```
t(a,0,b,1,r).  t(b,0,a,1,l).    t(c,0,b,1,l). t(a,1,c,1,l).
t(b,1,b,1,r).  t(c,1,halt,1,r). final(halt).
```

The execution of the busy beaver on an empty tape beginning with state '`a`' would correspond to the query call shown in Figure 4.

Back again to ASP, under its most accepted understanding as a logic programming paradigm [8,13], one of its characteristic features is the complexity class it can cover: NP-completeness for normal logic programs [9]; $\Sigma_2^P$-completeness for disjunctive programs [4]. Both results refer to existence of a stable model for a *propositional* program. The use of variables as abbreviations of all their possible ground instances to obtain a finite propositional program is possible thanks to another fundamental feature of the traditional ASP paradigm: forbidding function symbols. These complexity bounds point out that, although useful for solving many constraint-like problems, ASP is far from being a Turing-complete programming language.

However, this "traditional" picture needs to be revised in the sight of recent results obtained during the last years. First, in the theoretical field, the classical definition of stable models [6], which was only applicable to propositional programs, has been extended for covering any arbitrary first order theory, thanks to the definition of *First Order Equilibrium Logic* [14], a nonmonotonic formalism relying on a monotonic intermediate logic, or the equivalent *General Theory of Stable Models* [5], a syntactic construct very close to Circumscription [12]. Under these extensions, we can even remove the restriction to Herbrand models, so that assumptions like, for instance, domain closure or unique names are now optional. Thus, at least as a theoretical device, the new generalisations of ASP can now perfectly deal[4] with (Herbrand models) of the encoding of a Turing machine we presented in Figure 3.

But this capability is not limited to the theoretical field. A second important recent breakthrough has been the introduction of functions in ASP [2] and its implementation with solver `DLV-complex` [3]. When a program with functions, disjunction and negation satisfies a given property, so-called being *finitely-ground*, it has nice computational features: brave and cautious reasoning become decidable, and its answer sets are computable. An interesting result is that finitely-ground programs can encode any computable function. This was proven by *encoding a*

[4] A correctness proof of this program with respect to any of these two first order formalisms would be interesting.

```
?- nexttm(a,[],0,[]).
tm(a, [], 0, [])
tm(b, [1], 0, [])
tm(a, [], 1, [1])
tm(c, [], 0, [1, 1])
tm(b, [], 0, [1, 1, 1])
tm(a, [], 0, [1, 1, 1, 1])
tm(b, [1], 1, [1, 1, 1])
tm(b, [1, 1], 1, [1, 1])
tm(b, [1, 1, 1], 1, [1])
tm(b, [1, 1, 1, 1], 1, [])
tm(b, [1, 1, 1, 1, 1], 0, [])
tm(a, [1, 1, 1, 1], 1, [1])
tm(c, [1, 1, 1], 1, [1, 1])
tm(halt, [1, 1, 1, 1], 1, [1])
Yes
```

**Fig. 4.** A query and its corresponing display output for the 3 state, 2 symbol busy beaver machine

*Turing machine as an ASP program* so that a function computation stops in the machine iff its ASP encoding is finitely recursive (and the answer set will contain the execution steps). As it can be expected, checking whether a program is finitely recursive is undecidable. In fact, the encoding of Figure 3 is practically the same one[5] recently used in [1] to prove Turing-completeness of ASP with functions. In practical terms, this means that we can feed a program similar to the one in Figure 3 to `DLV-complex` and, as the machine halts, obtain one answer set containing all the facts for `tm` shown in Figure 4.

At the sight of this new scenario, we must reconsider our main question: should ASP be considered now a programming language? Capturing a Turing machine looks an undeniable proof. However, we can still find a subtle distinction between ASP and Prolog behaviours. The ASP encoding is being used to *represent* a Turing machine and its computations, whereas the Prolog encoding actually *executes* those computations. We can associate a time stamp to each of the ordered lines that the computer prints to solve the Prolog query: time is "real." Contrarily, the answer set would contain the same set of `tm` facts, but with *no associated ordering*. In fact, if we wanted to reconstruct the order among the steps followed by the machine, we would have to include one more parameter for an explicit representation of time. Thus, in ASP time is "virtual."

**Observation 2.** *Although ASP and Prolog can encode a Turing machine, ASP* represents *the machine computations using virtual, reified time, whereas Prolog* executes *the machine computations using real time.*

Under our point of view, this observation reaffirms the idea of seeing ASP as a formal specification language rather than a programming language. Note how

[5] The original encoding considered a version of Turing machine with a left-ended tape.

the ASP orientation could be used to *analyse* a real-time application or a reactive system, but not to *implement* it in a real scenario.

## 5 Programming as a Craft

Finally, one more meaning we may sometimes associate to the idea of programming to distinguish it from formal specification is that the former involves some kind of craft work whereas the latter is frequently seen as an accurate task and free from efficiency issues. In the particular case of Prolog programs, we all know that practical programming involves capabilities like a reasonable use of the cut predicate or an efficient list construction strategy. This usually means important sacrifices in program readability and declarativeness (the simple inclusion of a cut predicate may easily change the program semantics).

Unfortunately, efficiency considerations are also present in practical ASP (mostly related to reduce grounding) and, as happens with Prolog, they frequently introduce a sacrifice with respect to the program quality too. As ASP is a formal specification language, this sacrifice does not have to do with a lack of declarativeness – we can say that ASP programs are always declarative. The cost to pay may come as a lack of *elaboration tolerance* [10]. As defined by John McCarthy:

> *"A formalism is* elaboration tolerant *to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances."*

The quest for elaboration tolerance was present in ASP from its very beginning. Stable models and their use of default negation allowed establishing a fruitful connection between Logic Programming and Nonmonotonic Reasoning (NMR). We can even say that ASP constitutes nowadays one of the most successful tools for practical NMR. However, the fact that a language is elaboration tolerant (up to some degree) does not necessarily mean that any program built with that language is elaboration tolerant too. When we talk about elaboration tolerance of ASP, we mean that there exists a flexible way of representing a problem, not that *any* encoding of that problem in ASP will have the same degree of elaboration tolerance, or that it will have that property at all.

For instance, back to our HANOI problem, we will have few actions (and so, few choice points) if we consider, as we did, movements from one peg to another: for three pegs, this always means six possible actions. On the contrary, if we take, for instance, movements from a given disk to a given location (a peg base or another disk), the number of possible actions blows up as $n$ increases. The peg-to-peg representation is, however, less elaboration tolerant, as it is more focused on the given problem we try to solve rather than the physical possibilities of the scenario. As an example, assume now that some marked disks could be carried with all the disks they have above at a time. The disk-to-location encoding would still be valid, but the peg-to-peg representation no, as it implicitly assumes that we always take the top disk. Decisions like this arise in almost any ASP problem solving project.

To further illustrate this dilemma, think about the Second ASP Solver Competition[6], where efficiency plays a preeminent role, as expected. In this edition, the competition just defined the input and output format of the benchmark problems, and the final encoding was left to the competitors. This idea had the important advantage of opening the competition to solvers that accepted *different languages*, so they could compete altogether for a faster solution. The disadvantage, however, is obvious: we are measuring not only the performance of a given solver, but also the craft or experience of the ASP programmer to obtain a more "efficient representation" (usually, a less grounding-consuming one). Furthermore, if we look at the three ASP encodings available in the competition site for the HANOI problem we will find out that elaboration tolerance sacrifices have been considerable. None of the three solutions contain the default rule of *inertia* to avoid the frame problem [11], which has been the cornerstone of the NMR area of Reasoning about Actions and Change. In fact, the use of negation is quite limited (no defaults are really used) and two of the encodings contain an automaton-style description, which is probably one of the less elaboration tolerant ways of describing a problem (any slight variation usually means rebuilding the whole automaton).

Although we recognize the crucial importance of an efficiency competition for an active improvement of the available solvers (as happened in the SAT area), it is perhaps worth to consider a different track including benchmarks focused on an fixed, elaboration tolerant encoding of a given problem. After all, our final goal should proving that *efficient elaboration tolerance* is feasible. We can summarize this focusing by ending up with another quote by McCarthy, when talking about the use of chess as a drosophila for AI:

> *"Unfortunately, the competitive and commercial aspects of making computers play chess have taken precedence over using chess as a scientific domain. It is as if the geneticists after 1910 had organized fruit fly races and concentrated their efforts on breeding fruit flies that could win these races."*

## 6 Conclusions

After examining several aspects of the idea of programming, we claim that ASP is not a programming paradigm in a strict sense, but a formal specification language instead. Still, we find that the term Answer Set *programming* can be more vaguely used to refer to the craft (and perhaps in the future, to the methodologies) for developing an efficient and elaboration tolerant formal representation of a given problem.

*Acknowledgements.* Special thanks to Michael Gelfond for his always inspiring and enlightening discussions - hearing or reading him always means extracting

[6] `http://dtai.cs.kuleuven.be/events/ASP-competition/index.shtml`

new valuable ideas. Also thanks to Ramón P. Otero and Alessandro Provetti, who introduced me to Dr. Gelfond and brought my attention to his work some years ago.

## References

1. Alviano, M., Faber, W., Leone, N.: Disjunctive ASP with functions: Decidable queries and effective computation. Theory and Practice of Logic Programming 10(4-6), 497–512 (2010)
2. 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)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: An ASP System with Functions, Lists, and Sets. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 483–489. Springer, Heidelberg (2009)
4. Eiter, T., Gottlob, G.: Complexity results for disjunctive logic programming and application to nonmonotonic logics. In: Proceedings of the International Logic Programming Symposium (ILPS), pp. 266–278. MIT Press, Cambridge (1993)
5. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 372–379 (2007)
6. 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)
7. Kowalski, R.: Algorithm = logic + control. Communications of the ACM 22(7), 424–436 (1979)
8. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm, pp. 169–181. Springer, Heidelberg (1999)
9. Marek, W., Truszczyński, M.: Autoepistemic logic. Journal of the ACM 38, 588–619 (1991)
10. McCarthy, J.: Elaboration tolerance. In: Proc. of the 4th Symposium on Logical Formalizations of Commonsense Reasoning (Common Sense 1998), pp. 198–217, London, UK (1998), Updated version at `http://www-formal.stanford.edu/jmc/elaboration.ps`
11. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence Journal 4, 463–512 (1969)
12. McCarthy, J.: Circumscription - a form of non-monotonic reasoning. Artif. Intell. 13(1-2), 27–39 (1980)
13. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
14. 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)
15. Tärnlund, S.-A.: Horn clause computability. BIT 16(2), 215–226 (1977)
16. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM 23(4), 733–742 (1976)

# Michael Gelfond: Essay in Honour of His 65th Birthday

Stefania Costantini

Università degli Studi di L'Aquila, Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
Stefania.Costantini@univaq.it

I have been requested to prepare an essay "about the birthday boy" in honour of Michael Gelfond. Though gratified by the proposal and willing to do my best, I found it fairly difficult to present Michael Gelfond to his students and colleagues, and even more to the many scientists in the world that are aware of his important contributions to Computer Science. Then, I drew inspiration from Benjamin Franklin, who, in the *incipit* of his autobiography, motivates his writing it by saying: "I have ever had pleasure in obtaining little anecdotes of my ancestors" and assuming that the same might hold for his relatives and friends. Likewise, Michael's friends and, I hope, Michael himself may take some pleasure in my telling some anecdotes related to my long-termed acquaintance with Michael. Actually, I realized that I have quite a bit of stories to tell, and that the subtitle of this essay might be "Pleasures and inconveniences of traveling abroad".

I met Michael for the first time in 1990, at the Workshop on Logic Programming and Non-Monotonic Reasoning, held in conjunction to the NACLP Conference in Austin. Of that Conference I remember a very funny Halloween Parade in Austin downtown and a very intriguing keynote speech by Raymond Smullyan at the Social Dinner about three, four, $\ldots n$ Wise Men and their hats, where the Wise Men became so many as to make three Japanese colleagues who were sitting next to me fall fast asleep (to my astonishment, I must say). I noticed at the Workshop this impressive man with a peculiar beard, and I understood that he was the author of two papers that I had found particularly interesting, and that had strongly influenced the work I was presenting at the workshop. I was not yet aware of the seminal paper about the stable model semantics. However, we introduced ourselves and had the occasion to talk together. Since then we have always been in contact. I consider Michael to be my mentor, and not without reason, as he has constantly followed, supported and encouraged my work on stable model semantics and, later on, on answer set programming. He has always demonstrated a consideration of my work which has greatly contributed to my self-esteem, especially when sometimes it was about to drop low.

As soon as I became aware of the work by Michael Gelfond and Vladimir Lifschitz introducing the stable model semantics, I grew so interested in the matter as to make it one of my main research directions. In 1992, I happened to be the Chairperson of the Seventh Italian Conference GULP'92, to be held in Tremezzo, a lovely location along the Como lake, and invited Michael to give a talk there. Michael decided to arrive some days earlier, so as to visit me in Milan and then join me to Tremezzo. He claimed not to need help for lodging, as his travel agent had booked a hotel near the Linate Airport, which is close to Milan city center. I am afraid that the deal had been concluded on the phone (Internet was in its early days) where, due to the American pronunciation, ‘Linate’ had been taken for ‘Lainate’, another location around Milan, but a lot farther. After clarifying the misunderstanding (and, again on the phone, this was not so easy), I reached Lainate by car to fetch Michael. I found him in a very luxurious hotel related to an exclusive golf course, where the man at the reception looked at us with some condescendence, probably because we were underdressed for that context . . .. I believe however that Michael enjoyed the Conference in Tremezzo, that was held in a wonderful palace on the lake.

When in 1999 I attended the LPNMR Conference in El Paso, chaired by Michael, he made me the honour of an invitation to give a seminar about my work on stable semantics at his Department. I was also invited to visit him at home, so I got acquainted with his sweet and witty wife Lara.

In September 2003, Michael was invited to Messina (Sicily) by Alessandro Provetti (a former student of Michael’s who had become an Associate Professor there) to give a talk at the ASP 2003 Workshop. In the meanwhile in fact, the term ASP, for “Answer Set Programming”, had surfaced, and, to witness the growing success of the new paradigm, the ASP series of Workshops had been started, and Alessandro was at that time the Coordinator of an European Network of Excellence, the “Working Group on Answer Set Programming” (WASP). In Messina, we experienced a black-out that, originated by some fault in Switzerland, that left large part of Europe without electricity for many hours. There was an initial astonishment and confusion about what might be happening and everybody was worried about the family, as both telephones and the Internet were out of service. Luckily, we were soon reassured thanks to somebody who had a car radio. Then, as the University was equipped with generators, we could enjoy one day at the Workshop after having washed ourselves . . . with mineral water! In the afternoon however, the electricity came back and we were rewarded by a shower and a nice social trip to Milazzo, ending with a gorgeous dinner in the perfect tradition of Sicilian hospitality. After the workshop, Michael took the occasion and came to visit me in L’Aquila. Thus, luckily he could see that wonderful town (and my house) before the devastation brought by the earthquake of April 6, 2009. To his way to the airport to fly back home, Michael, Alessandro and myself found the time for a nice walk in Rome, a roundabout that Michael seemed to greatly appreciate.

In April 2005, while attending the Workshop on Nonmonotonic Reasoning, Answer Set Programming and Constraints at Schloss Dagstuhl (Germany) I met Michael on the door, just arriving by taxi, without luggage and barely wearing, in the not-too-mild German spring, his usual sleeveless shirt. It came out that not only he had had to divide his journey into two parts, as for some (maybe meteorological?) reason he had been forced to make a stop of one day somewhere, but they had been able to lose his baggage twice! Luckily I had noticed that at the Reception they sold sweaters with a multi-color drawing of the castle, so Michael could avoid freezing until his baggage was delivered the day after.

At ICLP 2008, in Udine (Italy) it kept raining a whole weak, and because of the wet pavements a bad accident occurred to Micheal’s wife Lara, who had to undergo an operation. Michael himself incurred some health problem more or less in the same period. Luckily, both recovered good health, and in 2010 I met them in Oxford, where Michael once again surprised the audience by introducing a novel unexpected extension to answer set programming. One thing that I find amazing with Michael is his ability to introduce profound concepts by means of very simple though inevitably convincing examples. This in my opinion is an ability that is partly innate and partly refined by himself in that quest for essentiality which is proper of the true scientist.

Then, Happy Birthday Michael, your 65 years are a milestone, I am sure, on a long way where you will continue to enlighten all of us with your intelligence, wit and profoundness.

# PLINI: A Probabilistic Logic Program Framework for Inconsistent News Information*

Massimiliano Albanese[1], Matthias Broecheler[1], John Grant[1,2], Maria Vanina Martinez[1], and V.S. Subrahmanian[1]

[1] University of Maryland, College Park, MD 20742, USA
{albanese,matthias,grant,mvm,vs}@umiacs.umd.edu
[2] Towson University, Towson, MD 21252
jgrant@towson.edu

**Abstract.** News sources are reliably unreliable. Different news sources may provide significantly differing reports about the same event. Often times, even the same news source may provide widely varying data over a period of time about the same event. Past work on inconsistency management and paraconsistent logics assume that we have "clean" definitions of inconsistency. However, when reasoning about events reported in the news, we need to deal with two unique problems: (i) are two events being reported on the same or are they different? and (ii) what does it mean for two event descriptions to be mutually inconsistent, given that these events are often described using linguistic terms that do not always have a uniquely accepted formal semantics? The answers to these two questions turn out to be closely interlinked. In this paper, we propose a probabilistic logic programming language called PLINI (Probabilistic Logic for Inconsistent News Information) within which users can write rules specifying what they mean by inconsistency in situation (ii) above. We show that PLINI rules can be learned automatically from training data using standard machine learning algorithms. PLINI is a variant of the well known generalized annotated program framework that accounts for similarity of numeric, temporal, and spatial terms occurring in news. We develop a syntax, model theoretic semantics, and fixpoint semantics for PLINI rules, and show how PLINI rules can be used to detect inconsistent news reports.

## 1 Introduction

Google alone tracks thousands news sites around the world on a continuous basis, collecting millions of news reports about a wide range of phenomena. While a large percentage of news reports are about different types of *events* (such as terrorist attacks, meetings of G-8 leaders, results of sporting events, to name a few), there are also other types of news reports such as editorials and style sections that may not always be linked to events, but to certain topics (which in turn may include events). For example, it is quite common to read editorials about a nuclear nonproliferation treaty or about a political candidate's attacks on his rival. Thus, even in news pieces that may not directly be about an event, there are often references to events.

* Some of the authors of this paper were funded in part by AFOSR grant FA95500610405 and ARO grant W911NF0910206.

In this paper, we study the problem of *identifying inconsistency* in news reports about events. The need to reason about inconsistency is due to the fact that different news sources generate their individual stories about an event which may differ from one another. We do not try to develop methods to resolve the inconsistency or perform paraconsistent reasoning in this paper. Existing methods for inconsistency resolution and paraconsistent logics [1,2,3,4,5,6,7,8] can be used on top of what we propose.

For instance, we may have a single event (a bombing in Ahmedabad, India in July 2008) that generates the following different news reports.

**(S1)** *An obscure Indian Islamic militant group is claiming responsibility for a bombing attack that killed at least 45 people in a western Indian city.*[1]

**(S2)** *Police believe an e-mail claiming responsibility for the bombing that killed 45 people Saturday was sent from that computer in a Mumbai suburb.*[2]

**(S3)** *MUMBAI – Police carried out a manhunt here Tuesday, believing that the serial blasts that rocked the western Indian city of Ahmedabad over the weekend, killing 42 people, were hatched in a Mumbai suburb.*[3]

Any reader who reads these reports will immediately realize that, despite the inconsistencies, they all refer to the same event. The inconsistencies in the above reports fall into the categories below.

1. **Linguo-Numerical Inconsistencies.** (S1) says *at least* 45 people were killed; (S2) says 45 people were killed; (S3) says 42 people were killed. (S1) and (S3) as well as (S2) and (S3) are inconsistent.
2. **Spatial Inconsistencies.** (S1) and (S3) are apparently (but not intuitively) inconsistent in terms of the geospatial location of the event. (S1) says the event occurred in a “western Indian city”, while (S3) says the event occurred in Ahmedabad. An automated computational system may flag this as an inconsistency if it does not recognize that Ahmedabad is in fact a western Indian city.
3. **Temporal Inconsistencies.** (S2) says the bombing occurred on Saturday, while (S3) says the bombing occurred over the weekend. When analyzing when the event occurred, we need to realize that the “Saturday” in (S2) refers to the past Saturday, while the “weekend” referred to in (S3) is the past weekend. Without this realization - and the realization that Saturday is typically a part of a weekend, a system may flag this as inconsistent.

In fact, when reasoning about events, many other kinds of inconsistencies or apparent inconsistencies can also occur. For example, a report that says an event occurred within 5 miles of College Park, MD and another report that says the event occurred in Southwest DC would (intuitively) be mutually inconsistent. When reasoning about inconsistency in reporting about news events, we need to recognize several factors.

– Are two news reports referring to the same event or not? The answer to this question determines whether integrity constraints (e.g. ones that say that if two violent events are the same, then the number of victims should be the same) are applicable or not?

[1] Canadian TV report on July 27, 2008.
[2] WBOC, based on an AP news report of July 28, 2008.
[3] The Wall Street Journal, based on an AP news report of July 30, 2008.

– Are the two event reports inconsistent or not? If the two events are deemed to be the same, then they should have "similar" attribute values. However, if the two events are considered to be different, then they may have dissimilar attribute values.
– A third problem, as mentioned above, is that inconsistency can arise in the linguistic terms used to describe news events. When should varying numbers, temporal references, and geospatial references be considered to be "close enough"? This plays an important role in determining whether news reports are inconsistent or not.

A problem arises because of circularity. The answer to the first question is based on whether the events in question have similar attribute values, while the answer to the second question says that equivalent events should have similar attribute values. The ability to distinguish whether two reports refer to the same event or not, and whether they are inconsistent or not, is key to the theory underlying PLINI. We start in Section 2 with an informal definition of what we mean by an event. In Section 3, we provide a formal syntax and semantics for PLINI-formulas that contain linguistically modified terms such as "about 5 miles from Ahmedabad", "over 50 people" and "the first weekend of May 2009." We briefly show how we can reason about linguistic modifications to numeric, temporal, and geospatial data. We discuss similarity functions in Section 4. Then, in Section 5, we provide a syntax for PLINI-programs. Section 6 provides a formal model theory and fixpoint semantics for PLINI-programs that is a variant of the semantics of generalized annotated programs [9]. The least fixpoint of the fixpoint operator associated with PLINI-programs allows us (with additional clustering algorithms) to infer that certain events should be considered identical, while other events should be considered different. This additional clustering algorithm is briefly described in Section 7. Finally, in Section 8, we describe our prototype implementation and experiments.

Figure 1 shows the architecture of our PLINI framework. We start with an information extraction program that extracts event information automatically from text sources. Our implementation uses T-REX [10], though other IE programs may be used as well. Information extracted from news sources is typically uncertain and may include information that is linguistically modified as for the sentences **(S1), (S2), (S3)** above. Once the information extractor has identified events and extracted properties of those events, we need to identify which events are similar (and this in turn requires determining which properties of events are similar). To achieve this, we assume the existence of similarity functions on various data types – we propose several such functions for certain data types that are common in processing news information. PLINI-programs may be automatically extracted from training data using standard machine learning algorithms and a training corpus. The rules in a PLINI-program allow us to determine the similarity between different events. Our PLINI-Cluster algorithm clusters events together based on the similarity determined by the rules. PLINI-Cluster runs very quickly and partitions the set of events into clusters. All events within the same cluster are deemed equivalent. Once this is done, we can determine whether a real inconsistency exists or not.

Our experiments are based on event data extracted by the T-REX [10] system. T-REX has been running continuously for over three years. It primarily extracts information on violent events worldwide from over 400 news sources located in 130 countries. Over 126 million articles have been processed to date by T-REX which has automatically

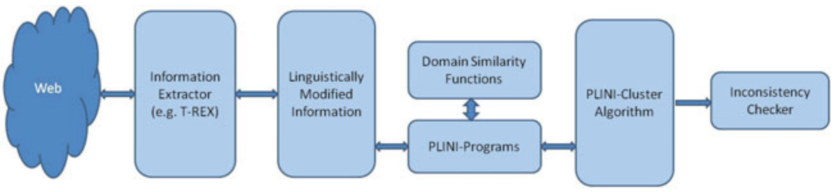

Web
Information Extractor (e.g. T-REX)
Linguistically Modified Information
Domain Similarity Functions
PLINI-Programs
PLINI-Cluster Algorithm
Inconsistency Checker

**Fig. 1.** Architecture of the PLINI-system

**Table 1.** Examples of event descriptions

| | |
|---|---|
| $e_{S1}$ | $(type, \text{“}bombing\ attack''), (perpetrator, \text{“}Indian\ Islamic\ Militant\ Group''),$ $(place, \text{“}western\ Indian\ city''), (number\_of\_victims, \text{“}at\ least\ 45'')$ |
| $e_{S2}$ | $(type, \text{“}bombing''), (date, \text{“}Saturday''),$ $(report\_time, 7/28/2008), (number\_of\_victims, 45)$ |
| $e_{S3}$ | $(type, \text{“}serial\ blast''), (number\_of\_victims, 42), (report\ time, 7/30/2008)$ $(place, \text{“}Ahmedabad''), (date, \text{“}over\ the\ weekend'')$ |

extracted a database of approximately 19 million property-value pairs related to violent events. We have conducted detailed experiments showing that the PLINI-architecture can identify inconsistencies with high precision and recall.

## 2 What Is an Event?

We assume that every event has three kinds of properties: a spatial property describing the region where the event occurred, a temporal property describing the period of time when the event occurred, and a set of event-specific properties describing various aspects of the event itself. The event specific properties vary from one type of event to another. Some examples of events are the following.

- **Terrorist act:** Here, the spatial property describes the region where the event occurred (e.g. Mumbai suburb), and various event-specific properties such as number_of_victims, number_injured, weapon, claimed_responsibility, arrested, etc.
- **Political meeting:** Here, the event specific properties here might include attendee, photo, agreement_reached, etc.
- **Natural disaster:** Here, the spatial properties may be somewhat different from those above. For instance, if we consider the 2004 tsunami in the Indian ocean, the region where the event occurred may be defined as a set of regions (e.g. Aceh, Sri Lanka, and so forth), while the time scales may also be different based on when the tsunami hit the affected regions. The event-specific attributes might include properties such as number_of_victims, number_injured, number_houses_destroyed, property_damage_value, and so forth.

An event can be represented as a set of $(property, value)$ pairs. Table 1 describes the events presented in Section 1.

## 3 PLINI Wffs: Syntax and Semantics

As shown in Section 1, news reports contain statements that contain numeric, spatial, and temporal indeterminacy. In this section, we introduce a multi-sorted logic syntax to capture such statements.

### 3.1 Syntax of Multi-sorted Wffs

Our definition of multi-sorted well formed formulas (mWFFs for short) builds upon well-known multi-sorted logics [11] and modifies them appropriately to handle the kinds of linguistic modifiers used in news articles as exemplified in sentences (S1), (S2) and (S3). In this section, we introduce the syntax of mWFFs.

Throughout this paper, we assume the existence of a set $\mathcal{S}$ of *sorts*. The set $\mathcal{S}$ includes sorts such as $Real$, $Time$, $TimeInterval$, $Date$, $NumericInterval$, $Point$, $Space$, and $ConnectedPlace$. Each sort $s$ has an associated set $dom(s)$ whose elements are called *constants* of sort $s$. For each sort $s \in \mathcal{S}$, we assume the existence of an infinite set $\mathcal{V}_s$ of variable symbols.

**Definition 1 (Term).** *A* term *$t$ of sort $s$ is any member of $dom(s) \cup \mathcal{V}_s$. A* ground *term is a constant.*

We assume the existence of a set $\mathcal{P}$ of predicate symbols. Each predicate symbol $p \in \mathcal{P}$ has an associated arity, $arity(p)$, and a *signature*. If a predicate symbol $p \in \mathcal{P}$ has arity $n$, then its signature is of the form $(s_1, \ldots, s_n)$ where each $s_i \in \mathcal{S}$ is a sort.

**Definition 2 (Atom).** *If $p \in \mathcal{P}$ is a predicate symbol with signature $(s_1, \ldots, s_n)$, and $t_1, \ldots, t_n$ are (resp. ground) terms of sorts $s_1, \ldots, s_n$ respectively, then $p(t_1, \ldots, t_n)$ is a (resp. ground) atom.*

**Definition 3 (mWFF).** *A* multi-sorted well formed formula (mWFF) *is defined as follows:*

- *Every atom is an mWFF (atomic mWFF).*
- *If $A$ and $B$ are mWFFs, then so are $A \wedge B$, $A \vee B$, and $\neg A$.*
- *If $s \in \mathcal{S}$, $X \in \mathcal{V}_s$, and $A$ is an mWFF, then $\forall_s X.A$ and $\exists_s X.A$ are also mWFFs.*

We are now ready to give a semantics for the syntactic objects introduced above. We start with the definition of denotation of various syntactic constructs.

**Definition 4 (Denotation).** *Suppose $s \in \mathcal{S}$ is a sort, and $c \in dom(s)$. Each sort $s$ has a* fixed *associated* denotation universe *$\mathcal{U}_s$. Each ground term $t$ of sort $s$ and each predicate symbol $p \in \mathcal{P}$ has a* denotation *$[\![t]\!]$ ($[\![p]\!]$ resp. ), defined as follows.*

- *$[\![c]\!]$ is an element of $\mathcal{U}_s$ for each $c \in dom(s)$.*
- *If $p \in \mathcal{P}$ is a predicate symbol with signature $(s_1, \ldots, s_n)$, then $[\![p]\!]$ is a subset of $\mathcal{U}_{s_1} \times \ldots \times \mathcal{U}_{s_n}$.*

This paper considers the sorts: $Real$, $Time$, $TimeInterval$, $Date$, $Point$, $Space$, and $ConnectedPlace$. We describe each of these sorts below.

**Real**. *Real* is a sort whose domain is the set $\mathbb{R}$ of real numbers. The denotation of symbols in $dom(Real)$ is:

- The denotation universe is $\mathcal{U}_{Real} = \mathbb{R}$.[4]
- For each symbol $r \in Real$, $[\![r]\!] = r \in \mathbb{R}$, i.e., real numbers denote themselves.

**Time**. Let us assume that *Time* is a sort having the set of symbols such as 2008, 08/2008, 08/01/2008, etc. as its domain.[5] The denotation of symbols in $dom(Time)$ can be defined as follows:

- The denotation universe is $\mathcal{U}_{Time} = \wp(\mathbb{Z})$ where $\mathbb{Z}$ is the set of non-negative integers and each $t \in \mathbb{Z}$ encodes a point in time, i.e. the number of time units elapsed since the origin of the time scale adopted by the user. As an example, $t \in \mathbb{Z}$ may encode the number of seconds elapsed since January $1^{st}$ 1970, 0:00:00 GMT.
- The denotation of each symbol $t' \in dom(Time)$ is an element of $\wp(\mathbb{Z})$, i.e. an unconstrained set of points in time.

**TimeInterval**. *TimeInterval* is a sort whose domain is the set of symbols of the form $(start, end)$ where $start, end \in \mathbb{Z}$. The denotation of symbols in $dom(TimeInterval)$ can be defined as follows:

- The denotation universe is $\mathcal{U}_{TimeInterval} = \{I \in \wp(\mathbb{Z}) \mid I \text{ is connected}\}$.
- The denotation of each symbol $(start, end) \in dom(TimeInterval)$ is defined in the obvious manner: $[\![(start, end)]\!] = [start, end)$ — note that this is a left-closed, right open interval.

**Date**. Let us assume that *Date* is a sort having the set of symbols of the form *month-day-year* as its domain and $dom(Date) \subset dom(TimeInterval)$. The denotation of symbols in $dom(Date)$ can be defined as follows:

- The denotation universe is $\mathcal{U}_{Date} = \{D \in \mathcal{U}_{TimeInterval} \mid sup(D) - inf(D) = \tau \wedge inf(D) \bmod \tau = 0\}$, where $\tau$ is the number of time units, in the selected time scale, contained in a day. For example, if the adopted time scale has a granularity of hours, then $\tau = 24$.

**Point**. *Point* is a sort whose domain is the set $\mathbb{R} \times \mathbb{R}$. The denotation of symbols in $dom(Point)$ can be defined as follows:

- The denotation universe is $\mathcal{U}_{Point} = \mathbb{R} \times \mathbb{R}$.
- For each symbol $p = (r_1, r_2) \in dom(Point)$, $[\![p]\!]$ is the point $p = (r_1, r_2) \in \mathbb{R}\times\mathbb{R}$.

[4] Though the domain and denotation universe of *Real* are identical, this is not the case for all sorts (the sorts *Space* and *ConnectedPlace* below are examples).

[5] Formally, we could define this set of symbols as follows. Every non-negative integer is a *year*. Every integer from 1 to 12 is a month. Every integer from 1 to 31 is a day. Every year is in $dom(Time)$. If $m$ is a month and $y$ is a year, then $m/y$ is in $dom(Time)$. If $d$ is a day, $m$ is a month, and $y$ is a year, then $d/m/y$ is in $dom(Time)$. The fact that 31/2/2009 is not a valid date can be handled by adding an additional "validity" predicate. We do not go into this as this is not the point of this paper.

**Space**. *Space* is a sort whose domain is an enumerated set of strings such as $Atlantic$ $Ocean$, $Great\ Lakes$, $WashingtonDC$, etc. The denotation of symbols in $dom(Space)$ can be defined as follows:

– The denotation universe is $\mathcal{U}_{Space} = \wp(\mathbb{R} \times \mathbb{R})$, where $\wp(\mathbb{R} \times \mathbb{R})$ is the power set of $\mathbb{R} \times \mathbb{R}$.
– For each symbol $a \in dom(Space)$, $[\![a]\!]$ is a member of $\wp(\mathbb{R} \times \mathbb{R})$, i.e. an unconstrained set of points in $\mathbb{R} \times \mathbb{R}$.

For instance, the denotation, $[\![Paris]\!]$, of *Paris*, is a set of points on the 2-dimensional Cartesian plane that corresponds to the region referred to as *Paris*. Another example of an element of sort *Space* is *United States*, whose denotation is the set of points that is the union of all points in the real plane corresponding to each of the regions that form the country (continental US, Alaska, Hawaii, etc.).

**Connected Place**. *ConnectedPlace*'s domain is the subset of *Space*'s domain that consists of connected regions. The denotation of symbols in $dom(ConnectedPlace)$ can be defined as follows:

– The denotation universe is $\mathcal{U}_{ConnectedPlace} = \{a \in \mathcal{U}_{Space} \mid a \text{ is connected}\}$.
– For each symbol $l \in dom(ConnectedPlace)$, $[\![l]\!]$ is a connected element $l$ of $\wp(\mathbb{R} \times \mathbb{R})$, i.e. a connected region in $\mathbb{R} \times \mathbb{R}$ that corresponds to $l$. Thus, $[\![Washington\ DC]\!]$ might be the set $\{(x, y) \mid 10 \leq x \leq 12 \wedge 36 \leq y \leq 40\}$ and $[\![Paris]\!]$ might be similarly defined.

Note that while *continental US* is an element of sort *ConnectedPlace*, *United States* is not because the US is not a connected region. ***Throughout the rest of this paper we assume an arbitrary but fixed denotation function $[\![]\!]$ for each constant and predicate symbol in our language.***

**Definition 5 (Assignment).** *An assignment $\sigma$ is a mapping, $\sigma : \cup_{s \in \mathcal{S}} \mathcal{V}_s \rightarrow \cup_{s \in \mathcal{S}} \mathcal{U}_s$ such that for every $X \in \mathcal{V}_s$, $\sigma(X) \in \mathcal{U}_s$.*

Thus $\sigma$ assigns an element of the proper sort for every variable. We write $\sigma[A]$ to denote the simultaneous replacement of each variable $X$ in $A$ by $\sigma(X)$.

**Definition 6 (Semantics of mWFFs).** *The evaluation of an mWFF under assignment $\sigma$ is defined as follows:*

1. *If $p$ is a predicate symbol of arity $n$ and signature $(s_1, \ldots, s_n)$, and $t_1, \ldots, t_n$ are terms of sort $s_1, \ldots, s_n$ respectively, then the atomic mWFF $\sigma[p(t_1, \ldots, t_n)]$ is true* iff $([\![\sigma(t_1)]\!], \ldots, [\![\sigma(t_n)]\!]) \in [\![p]\!]$.
2. *If $A$ is an mWFF, then $\sigma[\neg A]$ is true* iff *$\sigma[A]$ is not true.*
3. *If $A$ and $B$ are both mWFFs, then $\sigma[A \wedge B]$ is true iff $\sigma[A]$ is true and $\sigma[B]$ is true.*
4. *If $A$ and $B$ are both mWFFs, then $\sigma[A \vee B]$ is true iff $\sigma[A]$ is true or $\sigma[B]$ is true.*
5. *If $A$ is an mWFF and $X \in \mathcal{V}_s$, then $\sigma[\forall_s X.A]$ is true* iff *for each possible assignment $\tau$, identical to $\sigma$ except possibly for $X$, $\tau[A]$ is true.*
6. *If $A$ is an mWFF and $X \in \mathcal{V}_s$, then $\sigma[\exists X.A]$ is true* iff *there is an assignment $\tau$, identical to $\sigma$ except possibly for $X$, for which $\tau[A]$ is true.*

**Table 2.** Denotations for selected linguistically modified numeric predicates

| *Predicate Symbol* | *Signature* | *Denotation* | *Associated Region* |
|---|---|---|---|
| $almost$ | $(Real, Real, Real)$ | $\{(x, \epsilon, y) \mid x, \epsilon, y \in \mathbb{R}$ $(0 < [\![\epsilon]\!] \leq 1) \wedge ((1 - \epsilon) \times x \leq y < x)\}$ | The interval $[(1 - \epsilon) \times x, x)$ |
| $at\_least$ | $(Real, Real, Real)$ | $\{(x, \epsilon, y) \mid x, \epsilon, y \in \mathbb{R}$ $(0 \leq [\![\epsilon]\!] \leq 1) \wedge (x \leq y \leq (x + (x \times \epsilon))\}$ | The interval $[x, x + (\epsilon \times x)]$ |
| $around$ | $(Real, Real, Real)$ | $\{(x, \epsilon, y) \mid x, \epsilon, y \in \mathbb{R} \wedge (0 \leq [\![\epsilon]\!] \leq 1)$ $(x - (x \times \epsilon) \leq y \leq x + (x \times \epsilon))\}$ | The interval $[x - (\epsilon \times x), x + (\epsilon \times x)]$ |
| $most\_of$ | $(Real, Real, Real)$ | $\{(x, \epsilon, y) \mid x, \epsilon, y \in \mathbb{R} \wedge (0.0 < [\![\epsilon]\!] < 0.5)$ $(x \times (1 - \epsilon) \leq y < x)\}$ | The interval $[x - (x \times \epsilon), x)$ |
| $between$ | $(Real, Real, Real, Real)$ | $\{(x, y, \epsilon, z) \mid x, y, z, \epsilon \in \mathbb{R} \wedge (0 \leq [\![\epsilon]\!] \leq 1)$ $(x - (x \times \epsilon) \leq z \leq y + (y \times \epsilon))\}$ | The interval $[x - (x \times \epsilon), y + (y \times \epsilon)]$ |

**Table 3.** Denotations for selected linguistically modified spatial predicates

| *Predicate Symbol* | *Signature* | *Denotation* | *Associated Region* |
|---|---|---|---|
| ***Positional Indeterminacy*** | | | |
| $center$ | $(ConnectedPlace, Real, Point)$ | $\{(l, \delta, p) \mid l \in \mathcal{U}_{ConnectedPlace} \wedge \delta \in [0, 1]$ $\wedge p \in \mathcal{U}_{Point} \wedge d(p, Cent(l)) \leq \delta \cdot hside(l)\}$ | Circle centered at the center of the rectangle maximally contained in $l$[6], with radius equal to a fraction $\delta$ of half the length of the smaller side of the rectangle |
| $boundary$ | $(Space, Point)$ | $\{(a, p) \mid a \in \mathcal{U}_{Space} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\forall \epsilon > 0 : (\exists p_1 \in a, p_2 \notin a : d(p_1, p) < \epsilon$ $\wedge d(p_2, p) < \epsilon))\}$ | Points on the edge of $a$ |
| ***Distance Indeterminacy*** | | | |
| $distance$ | $(Space, Real, Point)$ | $\{(a, r, p) \mid a \in \mathcal{U}_{Space} \wedge r \in \mathbb{R} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\exists p_0 \in a : d(p_0, p) = r)\}$ | Points at a distance $r$ from a point in $a$ |
| $within$ | $(Space, Real, Point)$ | $\{(a, r, p) \mid a \in \mathcal{U}_{Space} \wedge r \in \mathbb{R} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\exists p_0 \in a : d(p_0, p) \leq r)\}$ | Points at a distance $r$ or less from a point in $a$ |
| ***Directional Indeterminacy*** | | | |
| $north$ | $(Space, Real, Space)$ | $\{(a, \theta, p) \mid a \in \mathcal{U}_{Space} \wedge \theta \in \mathbb{R} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\exists p_0 \in a : p \in NCone(\theta, p_0))\}$ | $NCone(\theta, p)$: $\ell_0$ upwards parallel to the $Y$-axis |
| $ne$ | $(Space, Real, Space)$ | $\{(a, \theta, p) \mid a \in \mathcal{U}_{Space} \wedge \theta \in \mathbb{R} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\exists p_0 \in a : p \in NECone(\theta, p_0))\}$ | $NECone(\theta, p)$: $\ell_0$ to the right with slope 1 |
| $nw$ | $(Space, Real, Space)$ | $\{(a, \theta, p) \mid a \in \mathcal{U}_{Space} \wedge \theta \in \mathbb{R} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\exists p_0 \in a : p \in NWCone(\theta, p_0))\}$ | $NWCone(\theta, p)$: $\ell_0$ to the left with slope $-1$ |
| $south$ | $(Space, Real, Space)$ | $\{(a, \theta, p) \mid a \in \mathcal{U}_{Space} \wedge \theta \in \mathbb{R} \wedge p \in \mathcal{U}_{Point}$ $\wedge (\exists p_0 \in a : p \in SCone(\theta, p_0))\}$ | $SCone(\theta, p)$: $\ell_0$ downwards parallel to the $Y$-axis |

*An mWFF $A$ is true iff $\sigma[A]$ is true for all assignments $\sigma$.*

The above definitions describe the syntax and semantics of mWFFs. It should be clear from the preceding examples that we can use the syntax of mWFFs to reason about numbers with attached linguistic modifiers (e.g. "around 25", "between 25 and 30". "at least 40"), about time with linguistic modifiers (e.g. "last month", "morning of June 1, 2009",) and spatial information with linguistic modifiers (e.g. "center of Washington DC", "southwest of Washington DC").

Table 2 shows denotations of some predicate symbols for linguistically modified numbers, while Tables 3 and 4 do the same for linguistically modified geospatial and temporal quantities, respectively.

*Example 1 (Semantics for linguistically modified numbers).* Consider the predicate symbol $most\_of$ in Table 2. Given $0 < \epsilon < 0.5$, we say that $most\_of(x, \epsilon, y)$ is true ($y$ is "most of" $x$) iff $x \times (1 - e) \leq y \leq x$. Thus, when $x = 4, e = 0.3, y = 3.1$, we see that $y$ lies between 2.8 and 4 and hence $most\_of(4, 0.3, 3.1)$ holds. However, if $e = 0.2$, then $most\_of(4, 0.2, 3.1)$ does not hold because $y$ must lie in the interval $[3.2, 4]$.

[6] We are assuming there is one such rectangle; otherwise a more complex method is used.

**Table 4.** Denotations for selected linguistically modified temporal predicates

| *Predicate Symbol* | *Signature* | *Denotation* | **Associated Region** |
|---|---|---|---|
| $morning$ | $(Date, Date)$ | $\{(d_1, d_2) \mid d_1, d_2 \in \mathcal{U}_{Date} \wedge GLB(d_1) \leq d_2 \leq (GLB(d_1) + LUB(d_1)/2\}$ | The entire first half of a day |
| $last\_month$ | $(Date, Date)$ | for $m = 1, \{((m, d_0, y), z) \mid (m, d_0, y), z \in \mathcal{U}_{Date} \wedge (\exists i)\ s.t.\ (12, i, y-1) \in Date \ \wedge\ z \in [\![(12, i, y-1)]\!]\}$ for $m \geq 2, \{((m, d_0, y), z) \mid (m, d_0, y) \in \mathcal{U}_{Date}, z \in \mathcal{U}_{Time} \wedge (\exists i)\ s.t.\ (m-1, i, y) \in Date \ \wedge\ z \in [\![(m-1, i, y)]\!]\}$ | The denotation of the month immediately preceding m |
| $around$ | $(Date, Real, TimeInterval)$ | $\{((m, d_0, y), k, (z_s, z_e)) \mid (m, d_0, y) \in \mathcal{U}_{Date} \wedge z_s, z_e \in \mathcal{U}_{Time} \ \wedge\ k \in Real \ \wedge\ z_s = inf((m_s, d_s, y_s) \wedge z_e = sup((m_e, d_e, y_e))\}$, where $(m_s, d_s, y_s)$ and $(m_e, d_e, y_e)$ refer to the days which are exactly $k$ days before and after $(m, d_0, y)$ | The time points which are within a few days of a given date |
| $shortly\_before$ | $(Date, Real, TimeInterval)$ | $\{((m, d_0, y), k, (z_s, z_e)) \mid (m, d_0, y) \in \mathcal{U}_{Date} \wedge z_s, z_e \in \mathcal{U}_{Time} \ \wedge\ k \in \mathcal{U}_{Real} \ \wedge\ z_s = inf((m_s, d_s, y_s)) \wedge z_e = inf((m, d_0, y))]\}$, where $(m_s, d_s, y_s)$ refers to the day which is exactly $k$ days before $(m, d_0, y)$ | The period shortly before a given date |
| $shortly\_after$ | $(Date, Real, TimeInterval)$ | $\{((m, d_0, y), k, (z_s, z_e)) \mid (m, d_0, y) \in \mathcal{U}_{Date} \wedge z_s, z_e \in \mathcal{U}_{Time} \ \wedge\ k \in \mathcal{U}_{Real} \ \wedge\ z_s = sup((m, d_0, y)) \wedge z_e = inf((m_e, d_e, y_e))]\}$, where $(m_e, d_e, y_e)$ refers to the day which is exactly $k$ days after $(m, d_0, y)$ | The period shortly after a given date |

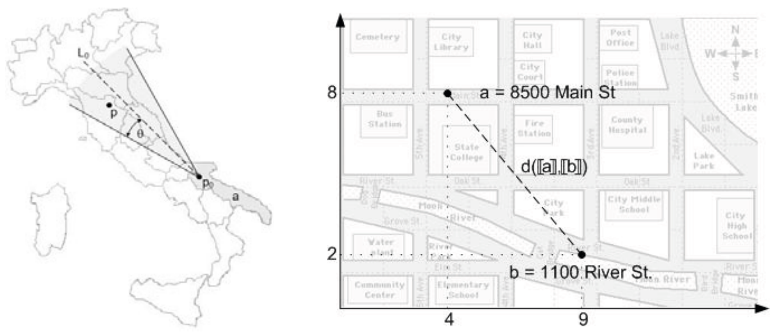


**Fig. 2.** Example of (a) point $p$ in the northwest of a region $a$; (b) application of $sim_1^P$ and $sim_2^P$

*Example 2 (Semantics for linguistically modified spatial concepts).* Consider the predicate symbol $boundary$ defined in Table 3 ($boundary$ is defined with respect to a set of points in a 2-dimensional space) and consider the rectangle $a'$ defined by the constraints $1 \leq x \leq 4$ and $1 \leq y \leq 5$. A point $p$ is on the boundary of $a$ iff for all $\epsilon > 0$, there is a point $p_1 \in a$ and a point $p_2 \notin a$ such that the distance between $p$ and each of $p_1, p_2$ is less than $\epsilon$. Using this definition, we see immediately that the point $(1, 1)$ is on the boundary of the rectangle $a'$, but $(1, 2)$ is not.

Now consider the predicate symbol $nw$ defining the northwest of a region (set of points). According to this definition, a point $p$ is to the northwest of a region $a$ w.r.t. cone-angle $\theta$ iff there exists a point $p_0$ in $a$ such that $p$ is in $NWCone(\theta, p_0)$. $NWCone(\theta, p_0)$[7] is defined to be the set of all points $p'$ obtained by (i) drawing a ray $L_0$ of slope $-1$ to the left of vertex $p_0$, (ii) drawing two rays with vertex $p_0$ at an angle of $\pm\theta$ from $L_0$ and (iii) looking at between the two rays in item (ii). Figure 2(a) shows this situation. Suppose $a$ is the shaded region and $\theta = 20$ (degrees). We see that $p$ is to the northwest of this region according to the definition in Table 3.

[7] The other cones referenced in Table 3 can be similarly defined.

## 4 Similarity Functions

We now propose similarity functions for many of the major sorts discussed in this paper. We do not claim that these are the only definitions – many definitions are possible, often based on application needs. We merely provide a few in order to illustrate that reasonable definitions of this kind exist.

We assume the existence of an arbitrary but fixed denotation function for each sort. Given a sort $s$, a similarity function is a function $sim^s : dom(s) \times dom(s) \rightarrow [0,1]$, which assigns a degree of similarity to each pair of elements in $dom(s)$. All similarity functions are required to satisfy two very basic axioms.

$$sim^s(a,a) = 1 \tag{1}$$

$$sim^s(a,b) = sim^s(b,a) \tag{2}$$

### 4.1 Sort *Point*

Consider the sort $Point$, with denotation universe $\mathcal{U}_{Point} = \mathbb{R} \times \mathbb{R}$. Given two terms $a$ and $b$ of sort $Point$, we can define the similarity between $a$ and $b$ in any of the following ways.

$$sim_1^P(a,b) = e^{-\alpha \cdot d([\![a]\!],[\![b]\!])} \tag{3}$$

where $d([\![a]\!],[\![b]\!])$ is the distance in $\mathbb{R} \times \mathbb{R}$ between the denotations of $a$ and $b$[8], and $\alpha$ is a factor that controls how fast the similarity decreases as the distance increases.

$$sim_2^P(a,b) = \frac{1}{1 + \alpha \cdot d([\![a]\!],[\![b]\!])} \tag{4}$$

where $d()$ and $\alpha$ have the same meaning as in Equation 3.

*Example 3.* Assuming that street addresses can be approximated as points, consider the points $a$ = "8500 Main St." and $b$ = "1100 River St." in Figure 2(b), with denotations $(4,8)$ and $(9,2)$ respectively. Assuming $\alpha = 0.3$, then $d([\![a]\!],[\![b]\!]) = 7.81$, $sim_1^P(a,b) = 0.096$, and $sim_2^P(a,b) = 0.299$.

### 4.2 Sort *ConnectedPlace*

Consider the sort $ConnectedPlace$, with denotation universe $\mathcal{U}_{ConnectedPlace} = \{a \in \mathcal{U}_{Space} \mid a \text{ is connected}\}$. Given two terms $a$ and $b$ of sort $ConnectedPlace$, the similarity between $a$ and $b$ can be defined in any of the following ways.

$$sim_1^{CP}(a,b) = e^{-\alpha \cdot d(c([\![a]\!]),c([\![b]\!]))} \tag{5}$$

where $c([\![a]\!]), c([\![b]\!])$ in $\mathbb{R}\times\mathbb{R}$ are the centers of $[\![a]\!]$ and $[\![b]\!]$ respectively, $d(c([\![a]\!]), c([\![b]\!]))$ is the distance between them, and $\alpha$ is a factor that controls how fast the similarity decreases as the distance between the centers of the two places increases. This similarity function works well when comparing geographic entities at the same level of granularity. When places can be approximated with points, it is equivalent to $sim_1^P(a,b)$.

$$sim_2^{CP}(a,b) = \frac{1}{1 + \alpha \cdot d(c([\![a]\!]),c([\![b]\!]))} \tag{6}$$

[8] If elements in $\mathcal{U}_{Point}$ are pairs of latitude, longitude coordinates, then $d()$ is the great-circle distance. We will assume that $d()$ is the Euclidean distance, unless otherwise specified.

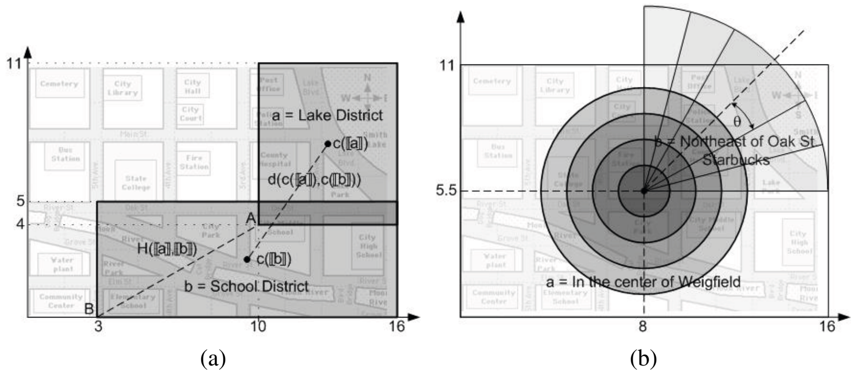


**Fig. 3.** Example of the application of similarity functions for sort *ConnectedPlace*

where $c()$, $d()$ and $\alpha$ have the same meaning as in Equation 5.

*Example 4.* Consider the two places $a =$ "Lake District" and $b =$ "School District" in Figure 3(a), and suppose their denotations are the two shaded rectangles in the figure. It is easy to observe that $c([\![a]\!]) = (13, 7.5)$, $c([\![b]\!]) = (9.5, 2.5)$, and $d(c([\![a]\!]), c([\![b]\!])) = 6.103$. Hence, for $\alpha = 0.3$, $sim_1^{CP}(a, b) = 0.160$ and $sim_2^{CP}(a, b) = 0.353$.

Other two similarity functions can be defined in terms of the areas of the two regions.

$$sim_3^{CP}(a, b) = \frac{A([\![a]\!] \cap [\![b]\!])}{A([\![a]\!] \cup [\![b]\!])} \tag{7}$$

where $A([\![t]\!])$ is a function that returns the area of $[\![t]\!]$. Intuitively, this function uses the amount of overlap between the denotations of $a$ and $b$ as their similarity.

$$sim_4^{CP}(a, b) = \frac{A([\![a]\!] \cap [\![b]\!])}{\max_{t \in \{a,b\}} A([\![t]\!])} \tag{8}$$

where $A([\![t]\!])$ has the same meaning as in Equation 7.

*Example 5.* Consider the two connected places $a =$ and $b =$ in Figure 3(a), and their respective denotations. The intersection of the two denotations is the darker shaded region, whereas their union is the whole shaded region. It is straightforward to see that $A([\![a]\!]) = 42$, $A([\![b]\!]) = 65$, $A([\![a]\!] \cap [\![b]\!]) = 6$, and $A([\![a]\!] \cup [\![b]\!]) = 101$. Thus, $sim_3^{CP}(a, b) = 0.059$ and $sim_4^{CP}(a, b) = 0.092$

In order to better illustrate the great expressive power of our framework, we now consider a more complex scenario, where the terms being compared are linguistically modified terms. We show how the similarity of such terms depends on the specific denotations assumed by the user for each predicate symbol.

*Example 6.* Consider the two linguistically modified terms of sort *ConnectedPlace* $a =$ "In the center of Weigfield" and $b =$ "Northeast of Oak St. Starbucks", where Weigfield is the fictional city depicted in Figure 3. Assuming the denotation of *center* and *ne* shown in Table 3, we now compute the similarity between $a$ and $b$ for different

**Table 5.** Value of $sim_3^{CP}(a,b)$ for different values of $\delta$ and $\theta$

| | $\delta = 0.2$ | $\delta = 0.4$ | $\delta = 0.6$ | $\delta = 0.8$ |
|---|---|---|---|---|
| $\theta = 15°$ | 0.0132 | 0.0276 | 0.0346 | 0.0380 |
| $\theta = 30°$ | 0.0157 | 0.0413 | 0.0593 | 0.0699 |
| $\theta = 45°$ | 0.0167 | 0.0494 | 0.0777 | 0.0970 |

values of $\delta$ and $\theta$. Figure 3(b) shows denotations of $a$ for values of $\delta$ of 0.2, 0.4, 0.6, and 0.8, and denotations of $b$ for values of $\theta$ of 15°, 30°, and 45°. In order to simplify similarity computation, we make the following assumptions (without loss of generality): (i) the term "Oak St. Starbucks" can be interpreted as a term of sort $Point$; (ii) the denotation of "Oak St. Starbucks" coincides with the geometrical center $(8, 5.5)$ of the bounding box of $[\![$"Weigfield"$]\!]$; (iii) the cones do not extend indefinitely, but rather within a fixed radius (8 units in this example) from their vertex. Table 5 reports the value of $sim_3^{CP}(a,b)$ for different values of $\delta$ and $\theta$. The highest similarity corresponds to the case where $\delta = 0.8$ and $\theta = 45°$, which maximizes the overlap between the two regions. Intuitively, this result tells us that a user with a very restrictive interpretation of $center$ and $ne$ (i.e., $\delta \ll 1$ and $\theta \ll 90°$ respectively) will consider $a$ and $b$ less similar than a user with a more relaxed interpretation of the same predicates.

Another similarity function can be defined in terms of the Hausdorff distance [12].

$$sim_5^{CP}(a,b) = e^{-\alpha \cdot H([\![a]\!],[\![b]\!])} \tag{9}$$

where $H(P,Q) = \max(h(P,Q), h(Q,P))$, with $P,Q \in \wp(\mathbb{R} \times \mathbb{R})$, is the Hausdorff distance, where $h(P,Q) = \max_{p \in P} \ min_{q \in Q} d(p,q)$ is the distance between the point $p \in P$ that is farthest from any point in $Q$ and the point $q \in Q$ that is closest to $p$. Intuitively, the Hausdorff distance is a measure of the mismatch between $P$ and $Q$; if the Hausdorff distance is $d$, then every point of $P$ is within distance $d$ of some point of $Q$ and vice versa.

*Example 7.* Consider again the two connected places $a =$ "Lake District" and $b =$ "School District" in Figure 3, and their respective denotations. In this example, the Hausdorff distance between $[\![a]\!]$ and $[\![b]\!]$ can be interpreted as the distance between the two points $A$ and $B$ shown in the figure. Therefore, $H([\![a]\!],[\![b]\!]) = 8.062$ and $sim_5^{CP}(a,b) = 0.089$ for $\alpha = 0.3$. Exchanging the roles of $[\![a]\!]$ and $[\![b]\!]$ would lead to a shorter value of the distance, whereas H() selects the maximum.

$$sim_6^{CP}(a,b) = e^{-\alpha \cdot d(c([\![a]\!]),c([\![b]\!]))} \cdot e^{-\beta \cdot (1 - o([\![a]\!],[\![b]\!]))} \tag{10}$$

where $c()$, $d()$ and $\alpha$ have the same meaning as in Equation 5, $o([\![a]\!],[\![b]\!]) = \frac{A([\![a]\!] \cap [\![b]\!])}{A([\![a]\!] \cup [\![b]\!])}$ is the amount of overlap between $[\![a]\!]$ and $[\![b]\!]$, and $\beta$ is a factor that controls how fast the similarity decreases as the amount of overlap between the two places decreases[9].

*Example 8.* Consider again the two connected places in Figure 3, and their respective denotations. In this example, $sim_6^{CP}(a,b) = 0.056$ for $\alpha = 0.3$ and $\beta = 0.5$.

[9] Alternatively, one could specify $o([\![a]\!],[\![b]\!]) = \frac{A([\![a]\!] \cap [\![b]\!])}{\max_{t \in \{a,b\}} A([\![t]\!])}$.

**Table 6.** Comparison between similarity functions over sort *ConnectedPlace*

| | $sim_1^{CP}$ | $sim_2^{CP}$ | $sim_3^{CP}$ | $sim_4^{CP}$ | $sim_5^{CP}$ | $sim_6^{CP}$ | $sim_{1'}^{CP}$ | $sim_{2'}^{CP}$ | $sim_{3'}^{CP}$ | $sim_{4'}^{CP}$ | $sim_{5'}^{CP}$ | $sim_{6'}^{CP}$ | $h$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (a) | 0.10 | 0.30 | 0.03 | 0.04 | 0.03 | 0.17 | 0.36 | 0.59 | 0.03 | 0.04 | 0.03 | 0.35 | 0.27 |
| (b) | 0.35 | 0.48 | 0.25 | 0.25 | 0.11 | 0.27 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.50 |
| (c) | 0.81 | 0.82 | 0.25 | 0.25 | 0.28 | 0.35 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.57 |
| (d) | 1.00 | 1.00 | 0.64 | 0.64 | 0.66 | 0.38 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.90 |
| (e) | 0.07 | 0.52 | 0.00 | 0.00 | 0.02 | 0.16 | 0.31 | 0.56 | 0.00 | 0.00 | 0.02 | 0.35 | 0.07 |
| (f) | 0.40 | 0.52 | 0.14 | 0.17 | 0.12 | 0.28 | 0.67 | 0.79 | 0.14 | 0.17 | 0.12 | 0.37 | 0.37 |
| (g) | 0.40 | 0.54 | 0.14 | 0.17 | 0.22 | 0.28 | 0.67 | 0.79 | 0.14 | 0.17 | 0.22 | 0.37 | 0.37 |
| (h) | 0.66 | 0.70 | 0.25 | 0.25 | 0.43 | 0.33 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.53 |
| (i) | 0.43 | 0.54 | 0.25 | 0.25 | 0.18 | 0.29 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.53 |
| (j) | 1.00 | 1.00 | 0.05 | 0.05 | 0.12 | 0.38 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.37 |
| (k) | 0.47 | 0.57 | 0.46 | 0.46 | 0.12 | 0.30 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.70 |

**Table 7.** Correlation between similarity functions and human judgment for $\alpha = 0.045$ and $\beta = 1$

| $sim_1^{CP}$ | $sim_2^{CP}$ | $sim_3^{CP}$ | $sim_4^{CP}$ | $sim_5^{CP}$ | $sim_6^{CP}$ | $sim_{1'}^{CP}$ | $sim_{2'}^{CP}$ | $sim_{3'}^{CP}$ | $sim_{4'}^{CP}$ | $sim_{5'}^{CP}$ | $sim_{6'}^{CP}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.74 | 0.73 | 0.96 | 0.95 | 0.76 | 0.74 | 0.79 | 0.79 | 0.75 | 0.75 | 0.75 | 0.74 |

The similarity function $sim_{1'}^{CP}$ below considers two places equivalent when their denotations are included into one another. We can define $sim_{2'}^{CP}, \ldots, sim_{6'}^{CP}$ in a similar way by modifying $sim_2^{CP}, \ldots, sim_6^{CP}$ analogously.

$$sim_{1'}^{CP}(a,b) = \begin{cases} 1 \text{ if } [\![a]\!] \subseteq [\![b]\!] \vee [\![b]\!] \subseteq [\![a]\!] \\ sim_1^{CP}(a,b) \text{ otherwise} \end{cases} \tag{11}$$

**Experimental evaluation of similarity functions.** In order to compare spatial similarity functions with human intuition, we asked three independent subjects to evaluate, on a 1 to 10 scale, the similarity of 11 pairs of connected regions, corresponding to rows (a), . . ., (k) in Table 6. Table 7 shows the correlation between each similarity function and average human-judged similarity, for the values of $\alpha$ and $\beta$ that maximize correlation ($\alpha = 0.045$ and $\beta = 1$). Average inter-human correlation was found to be 0.82 – therefore $sim_3^{CP}$ and $sim_4^{CP}$ exhibit a correlation with average human judged similarity stronger than average inter-human correlation. It is worth noting that the second ("primed") version of the 6 base similarity functions does not improve correlation, except for the case of $sim_1^{CP}$ and $sim_2^{CP}$, which are based solely on the distance between the centers, and thus lack the notion of overlap, which is implicitly used by humans as a measure of similarity. This notion is introduced in $sim_1^{CP}$ and $sim_2^{CP}$ by their augmented versions, therefore $sim_{1'}^{CP}$ and $sim_{2'}^{CP}$ perform slightly better.

## 4.3 Sort *Space*

Consider the sort *Space*, with denotation universe $\mathcal{U}_{Space} = \wp(\mathbb{R} \times \mathbb{R})$, where $\wp(\mathbb{R} \times \mathbb{R})$ is the power set of $\mathbb{R} \times \mathbb{R}$. Given a term $a$ of sort *Space*, let $P([\![a]\!])$ denote a subset of $\mathcal{U}_{ConnectedPlace}$ such that $\bigcup_{x \in P([\![a]\!])} x = [\![a]\!]$, elements in $P([\![a]\!])$ are pairwise disjoint and maximal, i.e. $\nexists y \in \mathcal{U}_{ConnectedPlace}, x_1, x_2 \in P([\![a]\!])\ s.t.\ y = x_1 \cup x_2$. Intuitively, $P([\![a]\!])$ is the set of the denotations of all the connected components of $a$. Given two terms $a$ and $b$ of sort *Space*, the distance between $a$ and $b$ may be defined in many ways – two are shown below.

$$d_c^S(a,b) = \operatorname*{avg}_{a_i \in P([\![a]\!]), b_i \in P([\![b]\!])} d(c(a_i), c(b_i)) \tag{12}$$

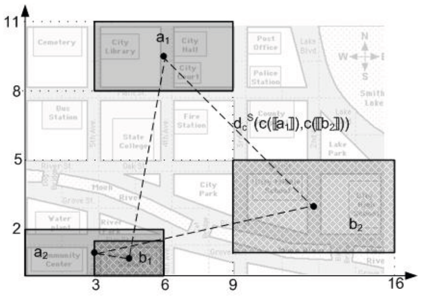


**Fig. 4.** Example of application of similarity functions for sort *Space*

where $c()$ and $d()$ have the same meaning as in Equation 5.

$$d_h^S(a,b) = \operatorname*{avg}_{a_i \in P(\llbracket a \rrbracket), b_i \in P(\llbracket b \rrbracket)} H(a_i, b_i) \tag{13}$$

where $H()$ is the Hausdorff distance.

Intuitively $d_c^S$ and $d_h^S$ measure the average distance between any two connected components of the two spaces being compared. Alternatively, the avg operator could be replaced by either min or max. As in the case of sort *ConnectedPlace*, a similarity function over sort *Space* can be defined in any of the following ways.

$$sim_1^S(a,b) = e^{-\alpha \cdot d_c^S(a,b)} \tag{14}$$

$$sim_2^S(a,b) = \frac{1}{1 + \alpha \cdot d_c^S(a,b)} \tag{15}$$

where $d_c^S$ is the distance defined by Equation 12 and $\alpha$ is a factor that controls how fast the similarity decreases as the distance increases.

*Example 9.* Consider the terms $a =$ "City buildings" and $b =$ "Schools" of sort *Space* in Figure 4 with denotations $\llbracket a \rrbracket = \{a_1, a_2\}$ and $\llbracket b \rrbracket = \{b_1, b_2\}$ respectively. By computing and averaging the distances between the centers of all pairs $a_i, b_j \in P(\llbracket a \rrbracket) \times P(\llbracket b \rrbracket)$ (see dashed lines in the figure), we obtain $d_c^S(a,b) = 7.325$ and $sim_1^S(a,b) = 0.111$, and $sim_2^S(a,b) = 0.313$ for $\alpha = 0.3$.

$$sim_3^S(a,b) = \frac{A(\llbracket a \rrbracket \cap \llbracket b \rrbracket)}{A(\llbracket a \rrbracket \cup \llbracket b \rrbracket)} \tag{16}$$

$$sim_4^S(a,b) = \frac{A(\llbracket a \rrbracket \cap \llbracket b \rrbracket)}{\max_{t \in \{a,b\}} A(\llbracket t \rrbracket)} \tag{17}$$

where $A(\llbracket t \rrbracket)$ is a function that returns the area of $\llbracket t \rrbracket$.

$$sim_5^S(a,b) = e^{-\alpha \cdot d_h^S(a,b)} \tag{18}$$

where $d_h^S$ is the distance defined by Equation 13 and $\alpha$ is a factor that controls how fast the similarity decreases as the distance increases.

$$sim_6^S(a,b) = e^{-\alpha \cdot d_c^S(a,b)} \cdot e^{-\beta \cdot (1 - o(\llbracket a \rrbracket, \llbracket b \rrbracket))} \tag{19}$$

where $d_c^S$ is the distance defined by Equation 12, $\alpha$ has the usual meaning, $o(\llbracket a \rrbracket, \llbracket b \rrbracket) = \frac{A(\llbracket a \rrbracket \cap \llbracket b \rrbracket)}{A(\llbracket a \rrbracket \cup \llbracket b \rrbracket)}$ is the amount of overlap between $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$, and $\beta$ is a factor that controls how fast the similarity decreases as the overlap between the two places decreases.

*Example 10.* Consider again the two terms of sort $Space$ in Figure 4. It is straightforward to see that $A(\llbracket a \rrbracket) = 30$, $A(\llbracket b \rrbracket) = 32.5$, $A(\llbracket a \rrbracket \cap \llbracket b \rrbracket) = 4.5$, and $A(\llbracket a \rrbracket \cup \llbracket b \rrbracket) = 58$. Therefore, $sim_3^S(a,b) = 0.078$, $sim_4^S(a,b) = 0.138$, and $sim_6^S(a,b) = 0.044$, for $\alpha = 0.3$ and $\beta = 1$.

## 4.4 Sort *TimeInterval*

Consider the sort $TimeInterval$, with denotation universe $\mathcal{U}_{TimeInterval} = \{I \in \wp(\mathbb{Z}) \mid I \text{ is connected}\}$[10]. Given two terms $a$ and $b$ of sort $TimeInterval$, the similarity between $a$ and $b$ can be defined in any of the following ways.

$$sim_1^{TI}(a,b) = e^{-\alpha \cdot |c(\llbracket a \rrbracket) - c(\llbracket b \rrbracket)|} \tag{20}$$

where, for each time interval $t \in dom(TimeInterval)$, $c(\llbracket t \rrbracket) = \text{avg}_{z \in \llbracket t \rrbracket}\, z$ is the center of $\llbracket t \rrbracket$, and $\alpha$ is a factor that controls how fast the similarity decreases as the distance between the centers of the two time intervals increases.

$$sim_2^{TI}(a,b) = \frac{1}{1 + \alpha \cdot |c(\llbracket a \rrbracket) - c(\llbracket b \rrbracket)|} \tag{21}$$

where $c()$ and $\alpha$ have the same meaning as in Equation 20.

*Example 11.* Consider the two terms of sort $TimeInterval$ $a =$ "around May 13, 2009" and $b =$ "shortly before May 16, 2009", and assume that the denotation of $around$ is a time interval extending $4$ days before and after the indeterminate date, and the denotation of $shortly\_before$ is the time interval extending 2 days before the indeterminate date. Then, $\llbracket a \rrbracket$ is the time interval $[05/9/09, 05/17/09]$ and $\llbracket b \rrbracket$ is the time interval $[05/14/09, 05/16/09]$. Assuming a time granularity of days, we have $c(\llbracket a \rrbracket) = 05/13/09$ and $c(\llbracket b \rrbracket) = 05/15/09$[11]. Therefore, assuming $\alpha = 0.3$, we conclude that $sim_1^{TI}(a,b) = 0.549$ and $sim_2^{TI}(a,b) = 0.625$.

$$sim_3^{TI}(a,b) = \frac{|\llbracket a \rrbracket \cap \llbracket b \rrbracket|}{|\llbracket a \rrbracket \cup \llbracket b \rrbracket|} \tag{22}$$

Intuitively, $sim_3^{TI}$ is the ratio of the number of time units in the intersection of the denotations of $a$ and $b$ to the number of time units in the union.

$$sim_4^{TI}(a,b) = \frac{|\llbracket a \rrbracket \cap \llbracket b \rrbracket|}{\max_{t \in \{a,b\}} |\llbracket t \rrbracket|} \tag{23}$$

$$sim_5^{TI}(a,b) = e^{-\alpha \cdot H(\llbracket a \rrbracket, \llbracket b \rrbracket)} \tag{24}$$

[10] Each $t \in \mathbb{Z}$ encodes a point in time, i.e. the number of time units elapsed since the origin of the time scale adopted by the user.

[11] Since we are assuming a time granularity of days, we are abusing notation and using 05/13/09 instead of the corresponding value $z \in \mathbb{Z}$.

where $H(P,Q) = \max(h(P,Q), h(Q,P))$, with $P, Q \in \wp(\mathbb{Z})$, is the Hausdorff distance.

$$sim_6^{TI}(a,b) = e^{-\alpha \cdot |c([\![a]\!]) - c([\![b]\!])|} \cdot e^{-\beta \cdot (1 - o([\![a]\!],[\![b]\!]))} \tag{25}$$

where $c()$ and $\alpha$ have the same meaning as in Equation 20, $o([\![a]\!],[\![b]\!]) = \frac{|[\![a]\!] \cap [\![b]\!]|}{|[\![a]\!] \cup [\![b]\!]|}$ is the amount of overlap between $[\![a]\!]$ and $[\![b]\!]$, and $\beta$ is a factor that controls how fast the similarity decreases as the amount of overlap between the two time intervals decreases.

*Example 12.* Consider again the two terms of sort *TimeInterval* of Example 11. We observe that $|[\![a]\!]| = 9$, $|[\![b]\!]| = 3$, $|[\![a]\!] \cap [\![b]\!]| = 3$, and $|[\![a]\!] \cup [\![b]\!]| = 9$. Therefore, $sim_3^{TI}(a,b) = 0.333$ and $sim_4^{TI}(a,b) = 0.333$. In addition, $H([\![a]\!],[\![b]\!]) = 5$, which implies $sim_5^{TI}(a,b) = 0.22$ and $sim_6^{TI}(a,b) = 0.469$, when $\alpha = 0.045$ and $\beta = 1$.

## 4.5 Sort *NumericInterval*

Consider the sort *NumericInterval*, with denotation universe $\mathcal{U}_{NumericInterval} = \{I \in \wp(\mathbb{N}) \mid I \text{ is connected}\}$[12]. As in the case of the sort *TimeInterval*, given two terms $a$ and $b$ of sort *NumericInterval*, the similarity between $a$ and $b$ can be defined in any of the following ways.

$$sim_1^{NI}(a,b) = e^{-\alpha \cdot |c([\![a]\!]) - c([\![b]\!])|} \tag{26}$$

where, for each numeric interval $t \in dom(NumericInterval)$, $c([\![t]\!]) = \text{avg}_{n \in [\![t]\!]}\, n$ is the center of $[\![t]\!]$, and $\alpha$ is a factor that controls how fast the similarity decreases as the distance between the centers of the two numeric intervals increases.

$$sim_2^{NI}(a,b) = \frac{1}{1 + \alpha \cdot |c([\![a]\!]) - c([\![b]\!])|} \tag{27}$$

where $c()$ and $\alpha$ have the same meaning as in Equation 26

*Example 13.* Consider the two terms of sort *NumericInterval* $a$ = "between 10 and 20" and $b$ = "at least 16", and assume that the denotation of *between* and *at_least* are those shown in Table 2, with $\epsilon = 0.1$ and $\epsilon = 0.5$ respectively. Then, $[\![a]\!]$ is the interval $[9, 22]$ and $[\![b]\!]$ is the interval $[16, 24]$. We have $c([\![a]\!]) = 16$ and $c([\![b]\!]) = 20$. Therefore, for $\alpha = 0.3$, $sim_1^{NI}(a,b) = 0.301$ and $sim_2^{NI}(a,b) = 0.455$.

$$sim_3^{NI}(a,b) = \frac{|[\![a]\!] \cap [\![b]\!]|}{|[\![a]\!] \cup [\![b]\!]|} \tag{28}$$

$$sim_4^{NI}(a,b) = \frac{|[\![a]\!] \cap [\![b]\!]|}{\max_{t \in \{a,b\}} |[\![t]\!]|} \tag{29}$$

$$sim_5^{NI}(a,b) = e^{-\alpha \cdot H([\![a]\!],[\![b]\!])} \tag{30}$$

where $H(P,Q)$ is the Hausdorff distance.

$$sim_6^{NI}(a,b) = e^{-\alpha \cdot |c([\![a]\!]) - c([\![b]\!])|} \cdot e^{-\beta \cdot (1 - o([\![a]\!],[\![b]\!]))} \tag{31}$$

[12] This seems to be a natural denotation for indeterminate expressions such as "between 3 and 6", "more than 3", etc. An exact quantity can be also represented as a singleton.

where $c()$ and $\alpha$ have the same meaning as in Equation 26, $o(\llbracket a \rrbracket, \llbracket b \rrbracket) = \frac{|\llbracket a \rrbracket \cap \llbracket b \rrbracket|}{|\llbracket a \rrbracket \cup \llbracket b \rrbracket|}$ is the amount of overlap between $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$, and $\beta$ controls how fast the similarity decreases as the amount of overlap between the two numeric intervals decreases.

*Example 14.* Consider again the two terms of sort $NumericInterval$ of Example 13. We observe that $|\llbracket a \rrbracket| = 14$, $|\llbracket b \rrbracket| = 9$, $|\llbracket a \rrbracket \cap \llbracket b \rrbracket| = 7$, and $|\llbracket a \rrbracket \cup \llbracket b \rrbracket| = 16$. Therefore, $sim_3^{NI}(a,b) = 0.438$ and $sim_4^{NI}(a,b) = 0.5$. Moreover, $H(\llbracket a \rrbracket, \llbracket b \rrbracket) = 7$, which implies $sim_5^{NI}(a,b) = 0.122$ and $sim_6^{NI}(a,b) = 0.447$, when $\alpha = 0.045$ and $\beta = 1$.

## 5 PLINI Probabilistic Logic Programs

In this section, we define the concept of a PLINI-rule and a PLINI-program. Informally speaking, a PLINI-rule states that when certain similarity-based conditions associated with two events $e_1, e_2$ are true, then the two events are equivalent with some probability. Thus, PLINI-rules can be used to determine when two event descriptions refer to the same event, and when two event descriptions refer to different events. PLINI-rules are variants of annotated logic programs [9] augmented with methods to handle similarity between events, as well as similarities between properties of events. Table 8 shows a small event database that was automatically extracted from news data by the T-REX system [10]. We see here that an event can be represented as a set of (property,value) pairs. Throughout this paper, we assume the existence of some set $\mathcal{E}$ of event names.

**Definition 7.** *An* event pair *over sort $s$ is a pair $(p, v)$ where $p$ is a property of sort $s$ and $v \in dom(s)$. An* event *is a pair $(e, EP)$ where $e \in \mathcal{E}$ is an event name and $EP$ is a finite set of event pairs such that each event pair $ep \in EP$ is over some sort $s \in \mathcal{S}$.*

We assume that a set $\mathcal{A}$ of properties is given and use the notation $eventname.property$ to refer to the property of an event. We start by defining event-terms.

**Definition 8 (Event-Term).** *Suppose $\mathcal{E}$ is a finite set of event names and $\mathcal{V}$ is a possibly infinite set of variable symbols. An* event-term *is any member of $\mathcal{E} \cup \mathcal{V}$.*

*Example 15.* Consider the event $e_{S3}$ presented in Section 2. Both $e_{S3}$ and $v$, where $v$ is a variable symbol, are event-terms.

We now define the concept of an equivalence atom. Intuitively, an equivalence atom says that two events (or properties of events) are equivalent.

**Definition 9 (Equivalence Atom).** *An* equivalence atom *is an expression of the form*

– *$e_i \equiv e_j$, where $e_i$ and $e_j$ are event-terms, or*
– *$e_i.a_k \equiv e_j.a_l$, where $e_i$, $e_j$ are event-terms, $a_k, a_l \in \mathcal{A}$, and $a_k, a_l$ are both of sort $s \in \mathcal{S}$, or*
– *$e_i.a_k \equiv b$, where $e_i$ is an event-term, $a_k \in \mathcal{A}$ is an attribute whose associated sort is $s \in \mathcal{S}$ and $b$ a ground term of sort $s$.*

**Table 8.** Example of Event Database extracted from news sources

| *Event name* | *Property* | *Value* |
|---|---|---|
| *Event1* | *date* | 02/28/2005 |
| | *location* | Hillah |
| | *number_of_victims* | 125 |
| | *weapon* | car bomb |
| *Event2* | *location* | Hilla , south of Baghdad |
| | *number_of_victims* | *at_Least* 114 |
| | *victim* | people |
| | *weapon* | massive car bomb |
| *Event3* | *killer* | twin suicide attack |
| | *location* | town of Hilla |
| | *number_of_victims* | *at_least* 90 |
| | *victim* | Shia pilgrims |
| *Event4* | *date* | 02/28/2005 |
| | *weapon* | suicide car bomb |
| | *location* | Hilla |
| | *number_of_victims* | 125 |
| *Event5* | *killer* | suicide car bomber |
| | *location* | Hillah |
| | *number_of_victims* | *at_least* 100 |
| *Event6* | *location* | Hillah |
| | *number_of_victims* | 125 |
| | *victim* | Iraqis |
| | *weapon* | suicide bomb |
| *Event7* | *weapon* | suicide bombs |
| | *location* | Hilla south of Baghdad |
| | *number_of_victims* | *at_least* 27 |
| *Event8* | *date* | 2005/02/28 |
| | *location* | Hilla |
| | *number_of_victims* | *between* 136 *and* 135 |
| | *victim* | people queuing to obtain medical identification cards |
| | *weapon* | suicide car bomb |
| *Event9* | *date* | 2005/03/28 |
| | *location* | Between Hillah and Karbala |
| | *number_of_victims* | *between* 6 *and* 7 |
| | *victim* | Shiite pilgrims |
| | *weapon* | Suicide car bomb |

*Example 16.* Let us return to the case of the events $e_{S1}, e_{S2}, e_{S3}$ from Table 1. Some example equivalence atoms include:

$$e_{S1} \equiv e_{S2}.$$
$$e_{S1}.place \equiv e_{S3}.place$$
$$e_{S3}.place \equiv Ahmedabad.$$

Note that two events need not be exactly identical in order for them to be considered equivalent. For instance, consider the events $e_{S1}, e_{S2}, e_{S3}$ given in Section 2. It is clear that we want these three events to be considered equivalent, even though their associated event pairs are somewhat different. In order to achieve this, we first need to state what it means for terms over various sorts to be equivalent. This is done via the notion of a PLINI-atom.

**Definition 10 (PLINI-atom).** *If $A$ is an equivalence atom and $\mu \in [0, 1]$, then $A : \mu$ is a PLINI-atom.*

The intuitive meaning of a PLINI-atom can be best illustrated via an example.

*Example 17.* The PLINI-atom $(e_1.weapon \equiv e_2.weapon) : 0.683$ says that the weapons associated with events $e_1$ and $e_2$ are similar with a degree of at least 0.683. Likewise, the PLINI-atom $(e_1.date \equiv e_2.date) : 0.575$ says that the dates associated with events $e_1$ and $e_2$ are similar with a degree of at least 0.575.

When providing a semantics for PLINI, we will use the notion of similarity function for sorts as defined in Section 4. There we gave specific examples of similarity functions for the numeric, spatial, and temporal domains. Our theory will be defined in terms of any arbitrary but fixed set of such similarity functions. The heart of our method for identifying inconsistency in news reports is the notion of PLINI-rules which intuitively specify when certain equivalence atoms are true.

**Definition 11 (PLINI-rule).** *Suppose $A$ is an equivalence atom, $A_1 : \mu_1, \ldots, A_n : \mu_n$ are PLINI-atoms, and $p \in [0, 1]$. Then*

$$A \xleftarrow{p} A_1 : \mu_1 \ \wedge \ldots \wedge \ A_n : \mu_n$$

*is a* PLINI-rule. *If $n = 0$ then the rule is called a* PLINI-fact. *$A$ is called the* head *of the rule, while $A_1 : \mu_1 \ \wedge \ldots \wedge \ A_n : \mu_n$ is called the* body. *A PLINI-rule is* ground *iff it contains no variables.*

**Definition 12 (PLINI-program).** *A* PLINI-program *is a finite set of PLINI-rules where no rule may appear more than once with different probabilities.*

Note that a PLINI-program is somewhat different in syntax than a probabilistic logic program [13] as no probability intervals are involved. However, it is a variant of a generalized annotated program due to [9]. In classical logic programming [14], there is a general assumption that logic programs are written by human (logic) programmers. However, in the case of PLINI-programs, this is not the case. PLINI-programs can be *inferred* automatically from training data. For instance, we learned rules (semi-automatically) to recognize when certain violent events were equivalent to other violent events in the event database generated by the information extraction program T-REX [10] mentioned earlier. To do this, we first collected a set of 110 events ("annotation corpus") extracted by T-REX from news events and then manually classified which of the resulting pairs of events from the annotation corpus were equivalent. We then used two classical machine learning programs called JRIP and J48 from the well known WEKA library[13] to learn PLINI-rules automatically from the data. Figure 5 shows some of the rules we learned automatically using JRIP.

We briefly explain the first two rules shown in Figure 5 that JRIP extracted automatically from the T-REX annotated corpus. The first rule says that when the similarity between the date field of events $e_1, e_2$ is at least 95.5997%, and when the similarity between the number of victims field of $e_1, e_2$ is 100%, and the similarity between their location fields is also 100%, then the probability that $e_1$ and $e_2$ are equivalent is 100%. The second rule says that when the dates of events $e_1, e_2$ are 100% similar, and the killer fields are at least 57.4707% similar, then the events are at least 75% similar.

[13] http://www.cs.waikato.ac.nz/ml/weka/

$$\begin{aligned}
e_1 \equiv e_2 \overset{1.0}{\longleftarrow}\ & e_1.\mathit{date} \equiv e_2.\mathit{date} : 0.955997 \ \wedge \\
& e_1.\mathit{number\_of\_victims} \equiv e_2.\mathit{number\_of\_victims} : 1 \ \wedge \\
& e_1.\mathit{location} \equiv e_2.\mathit{location} : 1. \\
e_1 \equiv e_2 \overset{0.75}{\longleftarrow}\ & e_1.\mathit{date} \equiv e_2.\mathit{date} : 1 \ \wedge \ e_1.\mathit{killer} \equiv e_2.\mathit{killer} : 0.574707. \\
e_1 \equiv e_2 \overset{0.5833}{\longleftarrow}\ & e_1.\mathit{date} \equiv e_2.\mathit{date} : 1 \ \wedge \\
& e_1.\mathit{weapon} \equiv e_2.\mathit{weapon} : 0.634663 \ \wedge \\
& e_1.\mathit{location} \equiv e_2.\mathit{location} : 1.
\end{aligned}$$

**Fig. 5.** Some automatically learned PLINI-rules from T-REX data using JRIP

We see from this example that PLINI-programs weave together notions of similarity from different domains (within the annotations of equivalence atoms in the rule body) and the notion of probability attached to a rule. We now recall the standard concept of a substitution.

**Definition 13 (Substitution).** *Suppose $R$ is a PLINI-rule. A substitution $\sigma = [X_1/e_1, \ldots, X_n/e_n]$ for $R$ is a finite set of pairs of terms where each $e_i$ is an event-term and $X_i \neq X_j$ when $i \neq j$.*

*A ground instance of $R$ under $\sigma$ is the result of simultaneously replacing all variables $X_i$ in $R$ with the event-term $e_i$ where $X_i/e_i \in \sigma$.*

## 6 Model Theory and Fixpoint Theory

In this section, we specify a formal model theory for PLINI-programs by leveraging the semantics of generalized annotated programs [9]. For each sort $s \in \mathcal{S}$, we assume the existence of a similarity function $sim_s : dom(s) \times dom(s) \to [0, 1]$. Intuitively, $sim_s(v_1, v_2)$ returns 0 if domain values $v_1, v_2$ are completely different and returns 1 if the two values are considered to be the same. We have already provided many possible definitions for similarity functions in Section 4. We first need to define the Herbrand Base.

**Definition 14 (Herbrand Base).** *$\mathcal{B}_{\mathcal{E}}$ is the set of* all ground equivalence atoms *that can be formed from the event-terms, attributes, and constant symbols associated with $\mathcal{E}$.*

Clearly, $\mathcal{B}_{\mathcal{E}}$ is finite. We now define the concept of an interpretation.

**Definition 15 (Interpretation).** *Any function $I : \mathcal{B}_{\mathcal{E}} \to [0, 1]$ is called an* interpretation.

Thus, an interpretation just assigns a number in $[0, 1]$ to each ground equivalence atom. We now define satisfaction of PLINI-rules by interpretations.

**Definition 16 (Satisfaction).** *Let $I$ be a interpretation, and let $A, A_1, \ldots, A_n \in \mathcal{B}_{\mathcal{E}}$. Then:*

- *$I \models A : \mu$ iff $I(A) \geq \mu$.*
- *$I \models A_1 : \mu_1 \ \wedge \ldots \wedge \ A_n : \mu_n$ iff $I \models A_i : \mu_i$ for all $1 \leq i \leq n$.*
- *$I \models A \overset{p}{\longleftarrow} A_1 : \mu_1 \ \wedge \ldots \wedge \ A_n : \mu_n$ iff either $I \not\models A_1 : \mu_1 \ \wedge \ldots \wedge \ A_n : \mu_n$ or $I(A) \geq p\}$.*

*$I$ satisfies a non-ground rule iff it satisfies all ground instances of the rule. $I$ satisfies a PLINI-program $\Pi$ iff it satisfies all PLINI-rules in $\Pi$.*

The first part of the above definition says that for $A : \mu$ to be true w.r.t. an interpretation $I$, we should just check that $I(A)$ is greater than or equal to $\mu$. Satisfaction of conjunctions is defined in the obvious way. Satisfaction of a ground PLINI-rule is defined in a more complex way. Either the body of the rule should be false with respect to $I$ or $I$ must assign a value at least $p$ to the head. As usual, $A : \mu$ is a *logical consequence* of $\Pi$ iff every interpretation that satisfies $\Pi$ also satisfies $A : \mu$.

**Definition 17.** *Suppose $\Pi$ is a PLINI-program and we have a fixed set of similarity functions $sim_s$ for each sort $s$. The* augmentation of $\Pi$ with similarity information *is the PLINI-program $\Pi^{sim} = \Pi \cup \{(x \equiv y) \overset{sim_s(x,y)}{\longleftarrow} \mid x, y$ are ground terms of sort $s$ in $\Pi\}$.*

*Throughout the rest of this paper, we only consider the augmented program $\Pi^{sim}$.* We are interested in characterizing the set of ground equivalence atoms that are logical consequence of $\Pi^{sim}$. Given a PLINI-program $\Pi$, we are now ready to associate with $\Pi$, a fixpoint operator $T_\Pi$ which maps interpretations to interpretations.

**Definition 18.** *$T_\Pi(I)(A) = A : \mathbf{sup}\{p|\ A \overset{p}{\longleftarrow} A_1 : \mu_1 \ \wedge \ldots \wedge \ A_n : \mu_n$ is a ground instance of a rule in $\Pi$ and for all $1 \leq i \leq n$, either $I(A_i) \geq \mu_i$ or $A_i$ has the form $x_i \equiv y_i$ and $sim_s(x_i, y_i) \geq \mu_i\}$.*

The above definition says that in order to find the truth value assigned to a ground atom $A$ by the interpretation $T_\Pi(I)$, we first need to look at all rules in $\Pi$ that have $A$ as the head of a ground instance of that rule. To check whether the body of such a rule is true w.r.t. $I$, we need to look at each ground equivalence atom $A_i : \mu_i$ where $A_i$ has the form $(x_i \equiv y_i)$. This atom is satisfied if either $I(x_i \equiv y_i)$ is greater than or equal to $\mu_i$ or if the similarity function for sort $s$ (of the type of $x_i, y_i$) assigns a value greater than or equal to $\mu_i$ to $(x_i, y_i)$. Note that the $T_\Pi$ operator operates on the $\Pi^{sim}$ program without explicitly adding equivalence atoms of the form $(x \equiv y) \overset{sim_s(x,y)}{\longleftarrow}$ to $\Pi$, thus ensuring a potentially large saving.

It is easy to see that the set of all interpretations forms a complete lattice under the following ordering: $I_1 \leq I_2$ iff for all ground equivalence atoms $A$, $I_1(A) \leq I_2(A)$. We can define the *powers* of $T_\Pi$ as follows.

$$\begin{aligned} T_\Pi \uparrow 0(A) &= A : 0 \text{ for all ground equivalence atoms A.} \\ T_\Pi \uparrow (j+1)(A) &= (T_\Pi(T_\Pi \uparrow j))(A). \\ T_\Pi \uparrow \omega(A) &= \bigcap\{T_\Pi \uparrow j(A) \mid j \geq 0\}. \end{aligned}$$

The result below follows directly from similar results for generalized annotated programs [9] and shows that the $T_\Pi$ operator has some nice properties.

**Proposition 1.** *Suppose $\Pi$ is a PLINI-program and $sim_s$ is a family of similarity functions for a given set of sorts. Then:*

1. *$T_\Pi$ is monotonic, i.e. $I_1 \leq I_2 \rightarrow T_\Pi(I_1) \leq T_\Pi(I_2)$.*
2. *$T_\Pi$ has a least fixpoint, denoted $lfp(T_\Pi)$ which coincides with $T_\Pi \uparrow \omega$.*
3. *$I$ satisfies $\Pi^{sim}$ iff $T_\Pi(I) = I$.*
4. *$A : \mu$ is a logical consequence of $\Pi^{sim}$ iff $lfp(T_\Pi)(A) \geq \mu$.*

## 7 Event Clustering Algorithm

Suppose $e_1, e_2$ are any two reported events. The least fixpoint of the $T_\Pi$ operator gives the probability that the two events are equivalent (i.e., they refer to the same real-world event). Alternatively, $lfp(T_\Pi)(e_1 \equiv e_2)$ can be interpreted as the similarity between $e_1$ and $e_2$, meaning that if the similarity between two events is high, they are likely to refer to the same real-world event. In other words, the least fixpoint of the $T_\Pi$ operator gives us some information on the pairwise similarity between events. However, we may have a situation where the similarity according to $lfp(T_\Pi)$ between events $e_1$ and $e_2$ is 0.9, between $e_2$, and $e_3$ is 0.7, but the similarity between $e_1$ and $e_3$ is 0.5. In general, given a finite set $\mathcal{E}$ of events, we would like to look at the results computed by $lfp(T_\Pi)$, and cluster the events into *buckets* of equivalent events. We then need to find a partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ of $\mathcal{E}$, such that similar events are assigned to the same partition and dissimilar events are assigned to different partitions. In this section, we define the PLINI-Cluster algorithm, which can find a sub-optimal solution – w.r.t. the score function defined below – in polynomial time.

**Definition 19 (Score of Event Partition).** *Let $\Pi$ be a PLINI-program and $\tau \in [0, 1]$ a threshold. Let $\mathcal{E}$ be a set of events and $\mathcal{P} = \{P_1, \ldots, P_k\}$ a partition of $\mathcal{E}$, i.e. $\bigcup_{i=1}^{k} P_i = \mathcal{E}$, and $(\forall i \neq j)\ P_i \cap P_j = \emptyset$. We define the score of partition $\mathcal{P}$ as*

$$S(\mathcal{P}) = S_i(\mathcal{P}) + S_e(\mathcal{P})$$

*where $S_i(\mathcal{P})$ is the* internal score *of partition $\mathcal{P}$ given by*

$$S_i(\mathcal{P}) = \sum_{P_j \in \mathcal{P}} \sum_{e_r, e_s \in P_j, e_r \neq e_s} (lfp(T_\Pi)(e_r \equiv e_s) - \tau)$$

*and $S_e(\mathcal{P})$ is the* external score *of partition $\mathcal{P}$ given by*

$$S_e(\mathcal{P}) = \sum_{e_r, e_s \in \mathcal{E}, e_r \in P_u, e_s \in P_v, u \neq v} (\tau - lfp(T_\Pi)(e_r \equiv e_s))$$

Intuitively, $S_i(\mathcal{P})$ measures the similarity between objects within a partition component. Two events $e_1, e_2$ in the same cluster contribute positively to the internal score if their similarity is above the threshold $\tau$. Instead, if their similarity is below the threshold the partition is penalized. Analogously, $S_e(\mathcal{P})$ measures the similarity across multiple partition components. The external score of a partition is higher when events in different clusters have lower similarity. Two events $e_1, e_2$ in different clusters contribute positively to the external score if their similarity is below the threshold $\tau$.

**Algorithm 1.** PLINI-Cluster($\Pi, \mathcal{E}, \tau$)

```
1: $\mathcal{P} \leftarrow \emptyset$
2: for all $e_i \in \mathcal{E}$ do
3:     $\mathcal{P} \leftarrow \mathcal{P} \cup \{\{e_i\}\}$
4: end for
5: repeat
6:     for all $P_i, P_j \in \mathcal{P}$ s.t. $i \neq j$ do
7:         // Compute the change in the score of the partition, if we were to merge clusters $P_i, P_j$
8:         $s_{i,j} \leftarrow S((\mathcal{P} \setminus \{P_i, P_j\}) \cup \{P_i \cup P_j\}) - S(\mathcal{P}) =$
                $= 2 \cdot \sum_{e_r \in P_i, e_s \in P_j} (lfp(T_\Pi)(e_r \equiv e_s) - \tau)$ // Note that $s_{i,j} = s_{j,i}$
9:     end for
10:    $s_{u,v} \leftarrow \max_{i,j \in [1,|\mathcal{P}|] \wedge i \neq j} \{s_{i,j}\}$
11:    if $s_{u,v} > 0$ then
12:        // Merge the two clusters that provide the highest increase of the score
13:        $\mathcal{P} \leftarrow (\mathcal{P} \setminus \{P_u, P_v\}) \cup \{P_u \cup P_v\}$
14:    end if
15: until $s_{u,v} \leq 0$
```

**Table 9.** Degrees of similarity as given by $lfp(T_\Pi)(e_i \equiv e_j)$

| | $e_1$ | $e_2$ | $e_3$ | $e_4$ |
|---|---|---|---|---|
| $e_1$ | 1 | 0.2 | 0.9 | 0.3 |
| $e_2$ | | 1 | 0 | 0.8 |
| $e_3$ | | | 1 | 0.6 |
| $e_4$ | | | | 1 |

Finding the partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ which maximizes the score $S$ defined above gives rise to a combinatorial optimization problem. Note that $k$ is a problem parameter and not known a priori, in contrast to common clustering problems in machine learning and other areas. Ozcan et al. [15] were the first to determine how to partition a set of entities under this intuition – they did so in the context of partitioning a set of activities an agent is supposed to perform so that the activities in each component of the partition could be executed by leveraging commonalities of tasks amongst those activities. They proved that the partitioning problem was NP-hard and provided efficient heuristics to solve the problem. Later, Bansal et al. [16] showed that finding the optimal partition with respect to the score $S$ is NP-complete. Demaine and Immorlica [17] presented an efficiently computable $O(\log n)$ approximation based on integer program relaxation and proved that the bound is optimal for such relaxations. We now present a simple, greedy, hill-climbing algorithm for this task (Algorithm 1). The algorithm starts by assigning each event to a different cluster. Then, at each iteration, it merges the two clusters that provide the highest increase of the score. It stops when no further increase of the score can be achieved by merging two clusters.

*Example 18.* Suppose we have 4 events $e_1, e_2, e_3, e_4$, and assume $\tau = 0.5$. Table 9 shows the degree of similarity for each pair of events according to some fixed PLINI-program $\Pi$, i.e. the values of $lfp(T_\Pi)(e_i \equiv e_j)$. The algorithm starts by initializing $P_1 = \{e_1\}$, $P_2 = \{e_2\}$, $P_3 = \{e_3\}$, $P_4 = \{e_4\}$. In the first iteration, $P_1$ and $P_3$ are merged, because this leads to the largest increase in $S(\mathcal{P})$, $s_{1,3} = 0.8$. In the second iteration, $P_2$ and $P_4$ are merged for the same reason, with $s_{2,4} = 0.6$. In the third iteration, the algorithm terminates because no score increase can be achieved, as merging the remaining two clusters would decrease the score by 1.8. Hence, the resulting partition is $\{\{e_1, e_3\}, \{e_2, e_4\}\}$.

The following result shows that the above algorithm finds the locally optimal partition w.r.t the cost function and that its worst case runtime complexity is $O(n^3)$.

**Proposition 2.** *Suppose $\Pi$ is a PLINI-program, $\mathcal{E}$ a set of events, and $\tau$ a threshold. Then PLINI-Cluster$(\Pi, \mathcal{E}, \tau)$ finds a locally optimal partition of $\mathcal{E}$ w.r.t. to the score function $S$, and its worst case complexity is $O(n^3)$ where $n$ is the total number of events.*

**Proof.** Let $n$ denote the total number of events. We assume that we have an oracle to look up or compute $lfp(T_\Pi)$ for any pair of events in constant time. During each iteration of the algorithm either two clusters are merged or the algorithm terminates. The algorithm starts with a partition of size $n$ and since a merger reduces the partition size by one, there can be at most $n$ iterations. Now we turn to the complexity of each iteration, namely the cost of computing the $s_{i,j}$'s. For any given iteration, we have at most $O(n)$ clusters of constant size and a constant number of clusters of size $O(n)$, since the total number of elements (i.e. events) is $n$. If $|P_i| \in O(1)$ and $|P_j| \in O(1)$, then the cost of computing $s_{i,j}$ is $O(1)$ as well. Similarly, if $|P_i| \in O(n)$ and $|P_j| \in O(1)$, this cost is $O(n)$ and if $|P_i| \in O(n)$ and $|P_j| \in O(n)$ the cost is $O(n^2)$. Since we compute $s_{i.j}$ for all pairs of clusters, the overall complexity is

$$O(n) \times O(n) \times O(1) + O(n) \times O(1) \times O(n) + O(1) \times O(1) \times O(n^2) = O(n^2)$$

where the first summand is the complexity for pairs of constant size partitions, the second summand for pairs of linear with constant size partitions, and the last summand for pairs of linear size partitions. Hence, the complexity of each iteration is $O(n^2)$ and therefore the overall runtime complexity of the event clustering algorithm is in $O(n^3)$.$\Box$

Note that due to the sparsity of event similarities in real world datasets, we can effectively prune a large number of partition comparisons. We can prune the search space for the optimal merger even further, by considering highly associated partitions first. These optimizations do not impact the worst case runtime complexity, but render our algorithm very efficient in practice.

## 8 Implementation and Experiments

Our experimental prototype PLINI system was implemented in approximately 5700 lines of Java code. In order to test the accuracy of PLINI, we developed a training data set and a separate evaluation data set. We randomly selected a set of 110 event descriptions from the millions automatically extracted from news sources by T-REX [10]. We then generated all the 5,995 possible pairs of events from this set and asked human reviewers to judge the equivalence of each such pair. The ground truth provided by the reviewers was used to learn PLINI-programs for different combinations of learning algorithms and similarity functions. Specifically, we considered 588 different combinations of similarity functions and learned the corresponding 588 PLINI-programs using both JRIP and J48. The evaluation data set was similarly created by selecting 240 event descriptions from those extracted by T-REX.

All experiments were run on a machine with multiple, multi-core Intel Xeon E5345 processors at 2.33GHz, 8GB of memory, running the Scientific Linux distribution of the GNU/Linux operating system. However, the current implementation has not been parallelized and uses only one processor and one core at a time.

PLINI-programs corresponding to each combination of algorithms and similarity functions were run on the entire set of 28,680 possible pairs of events in the test set. However, evaluation was conducted on subsets of pairs of a manageable size for human reviewers. Specifically, we selected 3 human evaluators and assigned each of them two subsets of pairs to evaluate. The first subset was common to all 3 reviewers and included 50 pairs that at least one program judged equivalent with confidence greater than 0.6 (i.e. $T_\Pi$ returned over 0.6 for these pairs) and 100 pairs that no program judged equivalent with probability greater than 0.6. The second subset was different for each reviewer, and included 150 pairs, selected in the same way as the first set. Thus, altogether we evaluated a total of 600 distinct pairs.

We then computed precision and recall as defined below. Suppose $\mathcal{E}_p$ is the set of event pairs being evaluated. We use $e_1 \equiv_h e_2$, to denote that events $e_1$ and $e_2$ were judged to be equivalent by a human reviewer. We use $P(e_1 \equiv e_2)$ to denote the probability assigned by the algorithm to the equivalence atom $e_1 \equiv e_2$. Given a threshold value $\tau \in [0,1]$, we define the following sets.

- $\mathcal{TP}_1^\tau = \{(e_1,e_2) \in \mathcal{E}_p | P(e_1 \equiv e_2) \geq \tau \wedge e_1 \equiv_h e_2\}$ is the set of pairs flagged as equivalent (probability greater than the threshold $\tau$) by the algorithm and actually judged equivalent by human reviewers;
- $\mathcal{TP}_0^\tau = \{(e_1,e_2) \in \mathcal{E}_p | P(e_1 \equiv e_2) < \tau \wedge e_1 \not\equiv_h e_2\}$ is the set of pairs flagged as not equivalent by the algorithm and actually judged not equivalent by human reviewers;
- $\mathcal{P}_1^\tau = \{(e_1,e_2) \in \mathcal{E}_p | P(e_1 \equiv e_2) \geq \tau\}$ is the set of pairs flagged as equivalent by the algorithm;
- $\mathcal{P}_0^\tau = \{(e_1,e_2) \in \mathcal{E}_p | P(e_1 \equiv e_2) < \tau\}$ is the set of pairs flagged as not equivalent by the algorithm;

Given a threshold value $\tau \in [0,1]$, we define precision, recall, and F-measure as follows.

$$P_1^\tau = \frac{|\mathcal{TP}_1^\tau|}{|\mathcal{P}_1^\tau|} \quad P_0^\tau = \frac{|\mathcal{TP}_0^\tau|}{|\mathcal{P}_0^\tau|} \quad P^\tau = \frac{|\mathcal{TP}_1^\tau| + |\mathcal{TP}_0^\tau|}{|\mathcal{E}_p|}$$

$$R_1^\tau = \frac{|\mathcal{TP}_1^\tau|}{|\{(e_1,e_2) \in \mathcal{E}_p | e_1 \equiv_h e_2\}|}[14] \quad F^\tau = \frac{2 \cdot P_1^\tau \cdot R_1^\tau}{P_1^\tau + R_1^\tau}$$

Note that all the quality measures defined above are parameterized by the threshold $\tau$.

**Accuracy results.** Tables 10(a) and 10(b) report the overall performance of the PLINI-programs derived using J48 and JRIP, respectively for different values of the threshold $\tau$. Performance measures are averaged over all the 588 combinations of similarity

[14] Given the nature of the problem, most pairs of event descriptions are not equivalent. Therefore, the best indicators of our system performance are recall/precision w.r.t. equivalent pairs.

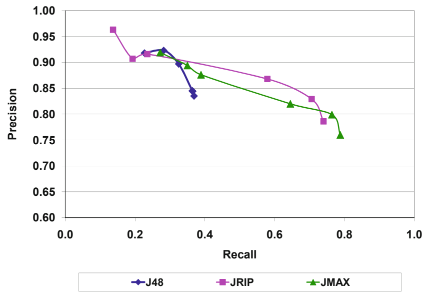
**Table 10.** Performance of (a) J48 and (b) JRIP for different values of $\tau$

| $\tau$ | $P_1^\tau$ | $P_0^\tau$ | $P^\tau$ | $R_1^\tau$ | $F^\tau$ |
|---|---|---|---|---|---|
| 0.50 | 83.5% | 88.6% | 88.2% | 36.9% | 0.512 |
| 0.60 | 84.4% | 88.5% | 88.2% | 36.5% | 0.510 |
| 0.70 | 84.5% | 88.5% | 88.2% | 36.4% | 0.509 |
| 0.80 | 89.7% | 87.9% | 88.1% | 32.5% | 0.477 |
| 0.90 | 92.3% | 87.3% | 87.6% | 28.2% | 0.432 |
| 0.95 | 91.8% | 86.5% | 86.7% | 22.7% | 0.364 |

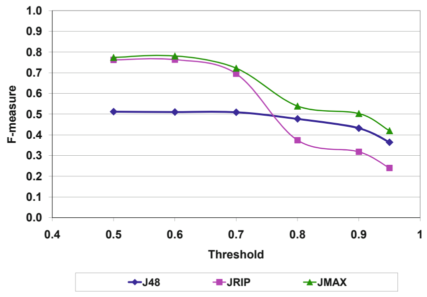
(a)

| $\tau$ | $P_1^\tau$ | $P_0^\tau$ | $P^\tau$ | $R_1^\tau$ | $F^\tau$ |
|---|---|---|---|---|---|
| 0.50 | 78.6% | 94.8% | 92.3% | 74.0% | 0.762 |
| 0.60 | 82.9% | 94.2% | 92.6% | 70.6% | 0.763 |
| 0.70 | 86.8% | 92.0% | 91.4% | 57.9% | 0.695 |
| 0.80 | 91.6% | 86.6% | 86.8% | 23.5% | 0.374 |
| 0.90 | 90.7% | 86.0% | 86.1% | 19.3% | 0.318 |
| 0.95 | 96.3% | 85.2% | 85.4% | 13.7% | 0.240 |

(b)



(a)



(b)

**Fig. 6.** (a) Recall/Precision and (b) F-measure for J48 and JRIP

metrics and over all the reviewers. Figure 6(a) shows precision/recall curves for both algorithms, while Figure 6(b) plots the F-measure vs. different values of $\tau$.

As expected, when $\tau$ increases $P_1^\tau$ increases and $R_1^\tau$ decreases. However, $R_1^\tau$ decreases more rapidly than the increase in $P_1^\tau$, causing $F^\tau$ to degrade. In general, rules extracted with JRIP outperform those extracted using J48, but it is also clear that the performance of JRIP in terms of F-measure degrades more rapidly than J48 due to a drastic drop in recall for higher thresholds. In fact, for thresholds above 0.75, J48-derived PLINI-programs surpass JRIP-derived PLINI-programs in terms of F-measure. Also note that when $\tau$ increases $P_0^\tau$ decreases. This is because higher thresholds cause a larger number of equivalent pairs to be classified as non equivalent, therefore the fraction of pairs flagged as not equivalent which are actually not equivalent decreases. However, precision is very high. When J48 and JRIP are used, the optimal value of the F-measure is achieved for $\tau = 0.6$, which corresponds to $84\%$ precision and $37\%$ recall for J48, and $83\%$ precision and $71\%$ recall for JRIP.

JMAX (which we defined) assigns to each equivalence atom the maximum of the probabilities assigned by JRIP-derived rules and J48-derived PLINI-rules. Figure 6(a) shows that JMAX produces significantly higher recall at the expense of a slight decrease in precision. The overall gain in performance is more evident in Figure 6(b), which shows that the F-measure for JMAX is higher than that for both J48 or JRIP for virtually all values of the threshold $\tau$. The optimal value of $F$ is achieved for $\tau = 0.6$, which corresponds to $80\%$ precision and $75\%$ recall.

**Table 11.** Average performance of JRIP for $\tau = 0.6$ when compared with different reviewers

| Algorithm | Reviewer | $P_1^\tau$ | $P_0^\tau$ | $P^\tau$ | $R_1^\tau$ | $F^\tau$ |
|---|---|---|---|---|---|---|
| J48 | 1 | 87.6% | 87.9% | 87.9% | 33.6% | 0.486 |
|  | 2 | 74.4% | 89.9% | 88.8% | 36.1% | 0.486 |
|  | 3 | 90.7% | 87.7% | 87.9% | 39.3% | 0.548 |
| JRIP | 1 | 84.6% | 93.9% | 92.6% | 68.8% | 0.759 |
|  | 2 | 80.2% | 95.9% | 93.7% | 76.1% | 0.781 |
|  | 3 | 83.9% | 92.9% | 91.6% | 67.8% | 0.750 |
| JMAX | 1 | 82.5% | 94.8% | 92.9% | 73.8% | 0.779 |
|  | 2 | 74.3% | 96.5% | 93.0% | 80.2% | 0.771 |
|  | 3 | 82.9% | 94.6% | 92.6% | 75.8% | 0.792 |

Table 11 reports the average performance of PLINI-programs derived using the 3 algorithms – J48, JRIP, JMAX when individually compared with the ground truth provided by each of the 3 reviewers enrolled for the evaluation. There is no significant difference between the reviewers and, in fact, they unanimously agreed in 138 out of the 150 common cases (92%).

We found that in general using both J48-based PLINI-rules and JRIP-based PLINI-rules (encompassed in our JMAX strategy) offers the best performance, while using only J48-derived PLINI-rules is the worst.

## 9 Related Work

The problem of event equivalence identification addressed in this work is closely related to a class of problems called *entity resolution* in machine learning [18]. Given a set of (potentially different types of) entities, entity resolution asks for a partition of this set such that entities are grouped together iff they are equivalent. In our problem, the entities of primary interest are events, but we also reason about the equivalence of actors, locations, and weapons which are entities of secondary interest.

Traditional machine learning approaches to entity resolution focus on pairwise entity equivalence determination using established techniques such as Bayesian networks [19], support vector machines [20], or logistic regression [21]. Hence, for each pair of entities a classifier is used to determine equivalence. In a post processing step, inconsistencies due to violations of transitivity are resolved.

Recent work has considered joint entity resolution in the case when entities are related [22]. Such approaches are termed *relational*, because they determine all event equivalences at once and take relationships between entities into account instead of making a series of independent decisions. Some proposals for relational entity resolution define a joint probability distribution over the space of entity equivalences and approximate the most likely configuration using sampling techniques, such as Gibbs sampling [23], or message passing algorithms, such as loopy belief propagation [24]. While these approaches are based on a probabilistic model they provide no convergence guarantee and allow little theoretical analysis. Other relational approaches are purely procedural in that they apply non-relational classifiers in an iterative fashion until some convergence criterion is met. While iterative methods are fast in practice, they are not amenable to theoretical analysis or convergence guarantees. All these approaches are feature driven and do not have a formal semantics.

## 10 Conclusion

The number of "formal" news sources on the Internet is mushrooming rapidly. Google News alone covers thousands of news sources from around the world. If one adds consumer generated content and informal news channels run by individual amateur newsmen and women who publish blogs about local or global items of interest, the number of news sources reaches staggering numbers. As shown in the Introduction, inconsistencies can occur for many reasons.

The goal of this paper is not to resolve these inconsistencies, but to identify when event data reported in news sources is inconsistent. When information extraction programs are used to automatically mine event data from news information, the resulting properties of the events extracted are often linguistically qualified. In this paper, we have studied three kinds of linguistic modifiers typically used when such programs are used – linguistic modifiers applied to numbers, spatial information, and temporal information. In each case, we have given a formal semantics to a number of linguistically modified terms.

In order to determine whether two events described in one or more news sources are the same, we need to be able to compare the attributes of these two events. This is done via similarity measures. Though similarity measures for numbers are readily available, no formal similarity mechanisms exist (to the best of our knowledge) for linguistically-modified numbers. The same situation occurs in the case of linguistically-modified temporal information and linguistically modified geospatial information. We provide formal definitions of similarity for many commonly used linguistically modified numeric, temporal, and spatial information.

We subsequently introduce PLINI-programs as a variant of the well known generalized annotated program (GAP) [9] framework. PLINI-programs can be learned automatically from a relatively small annotated corpus (as we showed) using standard machine learning algorithms like J48 and JRIP from the WEKA library. Using PLINI-programs, we showed that the least fixpoint of an operator associated with PLINI-programs tells us the degree of similarity between two events. Once such a least fixpoint has been computed, we present the PLINI-Cluster algorithm to cluster together sets of events that are similar, and sets of events that are dissimilar.

We have experimentally evaluated our PLINI-framework using many different similarity functions (for different sorts), many different threshold values, and three alternative ways of automatically deriving PLINI-programs from a small training corpus. Our experiments show that the PLINI-framework produced high precision and recall when compared with human users evaluating whether two reports talked about the same event or not.

There is much work to be done in the future. PLINI-programs do not include negation. A sort of stable models semantics [25] can be defined for PLINI-programs that include negation. However, the challenge will be to derive such programs automatically from a training corpus (standard machine learning algorithms do not do this) and to apply them efficiently as we can do with PLINI.

## References

1. Belnap, N.: A useful four valued logic. Modern Uses of Many Valued Logic, 8–37 (1977)
2. Benferhat, S., Dubois, D., Prade, H.: Some syntactic approaches to the handling of inconsistent knowledge bases: A comparative study part 1: The flat case. Studia Logica 58, 17–45 (1997)
3. Besnard, P., Schaub, T.: Signed systems for paraconsistent reasoning. Journal of Automated Reasoning 20(1-2), 191–213 (1998)
4. Blair, H.A., Subrahmanian, V.S.: Paraconsistent logic programming. Theoretical Computer Science 68(2), 135–154 (1989)
5. da Costa, N.: On the theory of inconsistent formal systems. Notre Dame Journal of Formal Logic 15(4), 497–510 (1974)
6. Fitting, M.: Bilattices and the semantics of logic programming. Journal of Logic Programming 11(2), 91–116 (1991)
7. Flesca, S., Furfaro, F., Parisi, F.: Consistent query answers on numerical databases under aggregate constraints. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 279–294. Springer, Heidelberg (2005)
8. Flesca, S., Furfaro, F., Parisi, F.: Preferred database repairs under aggregate constraints. In: Prade, H., Subrahmanian, V.S. (eds.) SUM 2007. LNCS (LNAI), vol. 4772, pp. 215–229. Springer, Heidelberg (2007)
9. Kifer, M., Subrahmanian, V.S.: Theory of generalized annotated logic programming and its applications. Journal of Logic Programming 12(3&4), 335–367 (1992)
10. Albanese, M., Subrahmanian, V.S.: T-REX: A domain-independent system for automated cultural information extraction. In: Proceedings of the First International Conference on Computational Cultural Dynamics, pp. 2–8. AAAI Press, Menlo Park (2007)
11. Cohn, A.G.: A many sorted logic with possibly empty sorts. In: Proceedings of the 11th International Conference on Automated Deduction, pp. 633–647 (1992)
12. Munkres, J.: Topology: A First Course. Prentice Hall, Englewood Cliffs (1974)
13. Ng, R., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation 101(2), 150–201 (1992)
14. Lloyd, J.: Foundations of Logic Programming. Springer, Heidelberg (1987)
15. Ozcan, F., Subrahmanian, V.S.: Partitioning activities for agents. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence, pp. 1218–1228 (2001)
16. Bansal, N., Blum, A., Chawla, S.: Correlation clustering. Machine Learning 56(1), 89–113 (2004)
17. Demaine, E.D., Immorlica, N.: Correlation clustering with partial information. In: Arora, S., Jansen, K., Rolim, J.D.P., Sahai, A. (eds.) RANDOM 2003 and APPROX 2003. LNCS, vol. 2764, pp. 71–80. Springer, Heidelberg (2003)
18. Bhattacharya, I., Getoor, L.: Collective entity resolution in relational data. ACM Transactions on Knowledge Discovery from Data (TKDD) 1(1) (2007)
19. Heckerman, D.: A tutorial on learning with bayesian networks. Proceedings of the NATO Advanced Study Institute on Learning in Graphical Models 89, 301–354 (1998)
20. Cristianini, N., Shawe-Taylor, J.: An introduction to support vector machines. Cambridge University Press, Cambridge (2000)

21. Ng, A.Y., Jordan, M.I.: On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. Advances in Neural Information Processing Systems 2, 841–848 (2002)
22. Getoor, L., Diehl, C.P.: Link mining: a survey. ACM SIGKDD Explorations Newsletter 7(2), 3–12 (2005)
23. Sen, P., Namata, G., Bilgic, M., Getoor, L., Gallagher, B., Eliassi-Rad, T.: Collective classification in network data. AI Magazine 29(3), 93 (2008)
24. Murphy, K., Weiss, Y., Jordan, M.I.: Loopy belief propagation for approximate inference: An empirical study. In: Foo, N.Y. (ed.) AI 1999. LNCS, vol. 1747, pp. 467–475. Springer, Heidelberg (1999)
25. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the 5th International Conference on Logic Programming, pp. 1070–1080 (1988)

# ASP as a Cognitive Modeling Tool: Short-Term Memory and Long-Term Memory

Marcello Balduccini[1] and Sara Girotto[2]

[1] Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com
[2] Department of Psychology
Texas Tech University
Lubbock, TX 79409 USA
sara.girotto@ttu.edu

**Abstract.** In this paper we continue our investigation on the viability of Answer Set Programming (ASP) as a tool for formalizing, and reasoning about, psychological models. In the field of psychology, a considerable amount of knowledge is still expressed using only natural language. This lack of a formalization complicates accurate studies, comparisons, and verification of theories. We believe that ASP, a knowledge representation formalism allowing for concise and simple representation of defaults, uncertainty, and evolving domains, can be used successfully for the formalization of psychological knowledge. In previous papers we have shown how ASP can be used to formalize a rather well-established model of Short-Term Memory, and how the resulting encoding can be applied to practical tasks, such as those from the area of human-computer interaction. In this paper we extend the model of Short-Term Memory and introduce the model of a substantial portion of Long-Term Memory, whose formalization is made particularly challenging by the ability to learn proper of this part of the brain. Furthermore, we compare our approach with various established techniques from the area of cognitive modeling.

## 1 Introduction

In this paper we continue our investigation on the viability of Answer Set Programming (ASP) [1,2,3] as a tool to formalize psychological knowledge and to reason about it.

ASP is a knowledge representation formalism allowing for concise and simple representation of defaults, uncertainty, and evolving domains, and has been demonstrated to be a useful paradigm for the formalization of knowledge of various kinds (e.g. intended actions [4] and negotiation [5] to name a few examples).

The importance of a precise formalization of scientific knowledge has been known for a long time (see e.g. Hilbert's philosophy of physics). Most notably, formalizing a body of knowledge in an area improves one's ability to (1) accurately study the properties and consequences of sets of statements, (2) compare competing sets of statements, and (3) design experiments aimed at confirming or refuting sets of statements. In the field of psychology, some of the theories about the mechanisms that govern the brain have been formalized using artificial neural networks and similar tools (e.g. [6]). That approach works well for theories that can be expressed in quantitative terms. However, theories of a more qualitative or logical nature, which by their own nature do not provide precise quantitative predictions, are not easy to formalize in this way.

In [7], we have presented an ASP-based formalization of the mechanism of Short-Term Memory (STM). Taking advantage of ASP's direct executability, i.e. the fact that the consequences of collections of ASP statements can be directly – and often efficiently – computed using computer programs, we have also applied our encoding of STM to practical tasks from the area of human-computer interaction.

However, as argued by various sources (e.g. [8]), to assess the viability of ASP for knowledge representation and reasoning, researchers need to build a number of theories, thus testing ASP's expressive power on a variety of domains. Therefore, in this paper we continue our investigation by using ASP to formalize a substantial part of the mechanism of Long-Term Memory (LTM), focusing in particular on a more complete account of the phenomenon of *chunking* (e.g. [9,10]), where by chunking we mean the brain's ability to recognize familiar patterns and store them efficiently (for example, it has been observed that it is normally difficult for people to remember the sequence CN NIB MMT VU SA, while most people have no problems remembering the sequence CNN IBM MTV USA, because each triplet refers to a familiar concept). Whereas chunking has been partially covered in [7], here we go far beyond the simplified representation of chunks adopted in our previous paper, and tackle the formalization of the LTM's ability to learn chunks upon repeated exposure, which is technically rather challenging. In this paper, we also compare our approach with various established techniques from the area of cognitive modeling.

As we already did in previous papers, it is worth stressing that we do not intend to claim that the psychological models we selected are the "correct" ones. On the contrary, any objections to the models that may come as a result of the analysis of our formalizations are a further demonstration of the benefits of formalizing psychological knowledge.

This paper is organized as follows. We start by providing a background on ASP and on the representation of dynamic domains. Next, we give an account of the mechanics of STM, LTM, and chunking, as it is commonly found in psychology literature. Then, we recap our ASP-based formalization of the mechanics of STM. The following section presents our formalization of the model of LTM and chunking. Finally, we compare with other approaches and conclude with a discussion on what we have achieved and on possible extensions.

## 2 Answer Set Programming and Dynamic Domains

Let us begin by giving some background on ASP. We define the syntax of the language precisely, but only give the informal semantics of the language in order to save space. We refer the reader to [1,11] for a specification of the formal semantics. Let $\Sigma$ be a propositional signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A (basic) literal is either an atom $a$ or its strong (also called classical or epistemic) negation $\neg a$. A *rule* is a statement of the form:

$$h_1 \ \vee \ \ldots \ \vee \ h_k \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

where $h_i$'s and $l_i$'s are ground 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 $\{l_{m+1}, \ldots, l_n\}$, must believe one of $h_i$'s. Symbol $\leftarrow$ can be omitted if no $l_i$'s are specified.

Often, rules of the form $h \leftarrow \text{not } h, l_1, \ldots, \text{not } l_n$, where $h$ is a fresh literal, are abbreviated into $\leftarrow l_1, \ldots, \text{not } l_n$, and called *constraints*. The intuitive meaning of a constraint is that $\{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ must not be satisfied. A rule containing variables is interpreted as the shorthand for the set of rules obtained by replacing the variables with all the possible ground terms.

A *program* is a pair $\langle \Sigma, \Pi \rangle$, where $\Sigma$ is a signature and $\Pi$ is a set of rules over $\Sigma$. We often denote programs just by the second element of the pair, and let the signature be defined implicitly. Finally, an *answer set* (or *model*) of a program $\Pi$ is one of the collections of its credulous consequences under the answer set semantics. Notice that the semantics of ASP is defined in such a way that programs may have multiple answer sets, intuitively corresponding to alternative views of the specification given by the program. In that respect, the semantics of default negation provides a simple way of encoding choices. For example, the set of rules $\{p \leftarrow \text{not } q. \ \ q \leftarrow \text{not } p.\}$ intuitively states that either $p$ or $q$ hold, and the corresponding program has two answer sets, $\{p\}$, $\{q\}$.

Because a convenient representation of alternatives is often important in the formalization of knowledge, the language of ASP has been extended with *constraint literals* [11], which are expressions of the form $m\{l_1, l_2, \ldots, l_k\}n$, where $m$, $n$ are arithmetic expressions and $l_i$'s are basic literals as defined above. A constraint literal is satisfied by a set $X$ of atoms whenever the number of literals from $\{l_1, \ldots, l_k\}$ that are satisfied by $X$ is between $m$ and $n$, inclusive. Using constraint literals, the choice between $p$ and $q$, under some set $\Gamma$ of conditions, can be compactly encoded by the rule $1\{p, q\}1 \leftarrow \Gamma$. A rule of this form is called *choice rule*. To further increase flexibility, a constraint literal $m\{l(a_1, b_1, \ldots), l(c_2, d_2, \ldots), \ldots, l(c_k, d_k, \ldots)\}n$ can also be specified using intentional notation, with an expression $m\{l(X, Y, \ldots) : dom_a(X) : dom_b(Y) : \ldots\}n$, where $X$ and $Y$ are variables, and relations $dom_a$ and $dom_b$ define the domain of the corresponding variables. We refer the reader to [11] for a more detailed definition of the syntax of constraint literals and of the corresponding extended rules. Readers who are familiar with ASP may have noticed that we do not impose restrictions on the $dom_i$ relations above. This is done to keep the presentation short. From the perspective of the implementation, when using ASP parsers that expect each $dom_i$ to be a domain predicate, one would have to use a slightly more elaborate encoding of knowledge.

Because of the dynamic nature of STM and LTM, for their formalization we use techniques from the area of reasoning about actions and change. The key elements of the representation techniques are presented next; we refer the readers to e.g. [12,13] for more details. *Fluents* are first-order ground terms, and intuitively denote the properties of interest of the domain (whose truth value typically depends on time). For example, an expression of the form $on(block_1, block_2)$ is a fluent, and may mean that $block_1$ is on top of $block_2$. A fluent literal is either a fluent $f$ or its negation ($\neg f$). Actions are also first-order ground terms. For example, $move(block_3, block_2)$ may mean that $block_3$ is moved on top of $block_2$. A set of fluent literals is *consistent* if, for every fluent $f$, $f$ and $\neg f$ do not both belong to the set. A set of fluent literals is *complete* if, for every fluent $f$, either $f$ or $\neg f$ belong to the set. The set of all the possible evolutions of a dynamic domain is represented by a *transition diagram*, i.e. a directed graph whose nodes – each labeled by a consistent set of fluent literals – correspond to the states of the domain in which the properties specified by the fluents are respectively true or false, and whose arcs – each labeled by a set of actions – correspond to the occurrence of state transitions due to the occurrence of the actions specified. When complete knowledge is available about a state, the corresponding set of fluent literals is also complete. When instead a set of fluent literals is not complete, that means that the knowledge about the corresponding state is incomplete (e.g. it is unknown whether $f$ or $\neg f$ holds). Incomplete or partial states are typically used to represent uncertainty about domains.

The size of transition diagrams grows exponentially with the increase of the number of fluents and actions; a direct representation is thus usually impractical. Instead, transition diagrams are encoded using an indirect representation, in the form of a domain description in an action language [12], or as a program in ASP (like in [14,15,16]). Here, we adopt the latter approach.

The encoding is based on the notion of a path in the transition diagram from a given initial state, corresponding to a particular possible evolution of the domain from that initial state. The steps in a path are identified by integers (with 0 denoting the initial state), and logical statements (often called *laws*) are used to encode, in general terms, the transitions from one step to the next. The fact that a fluent $f$ holds at a step $i$ in the evolution of the domain is represented by the expression $h(f, i)$, where relation $h$ stands for *holds*. If $\neg f$ is true, we write $\neg h(f, i)$. Occurrences of actions are represented by expressions of the form $o(a, i)$, saying that action $a$ occurs at step $i$ ($o$ stands for *occurs*). An *action description* is a collection of laws describing the evolution of the domain, and in particular the changes of state caused by the execution of actions. For example, the effect of flipping a switch on a connected bulb could be represented by the action description:

$$\begin{aligned}
h(on(bulb), S+1) \leftarrow\ & \neg h(on(bulb), S),\\
& h(connected(switch, bulb), S),\\
& o(flip(switch), S).\\
\neg h(on(bulb), S+1) \leftarrow\ & h(on(bulb), S),\\
& h(connected(switch, bulb), S),\\
& o(flip(switch), S).
\end{aligned}$$

Given an action description $AD$, a description of the initial state $\sigma_0$ (e.g. $\sigma_0 = \{h(f_1, 0), \neg h(f_2, 0), \ldots\}$), and a sequence of occurrences of actions $\alpha$ (e.g. $\alpha = \{o(a_1, 0), o(a_3, 0), o(a_4, 1), \ldots\}$), the corresponding path(s) in the transition diagram can be computed by finding the answer set(s) of $AD \cup \sigma_0 \cup \alpha$.

## 3 Short-Term Memory, Long-Term Memory, and Chunking

STM is “the memory storage system that allows for short-term retention of information before it is either transferred to long-term memory or forgotten” [10]. This view is based on the so called *three-stage model of memory* [17]: sensory inputs are first stored in Sensory Memory, which is very volatile and has large capacity; then, a portion of the inputs is processed – and possibly transformed into more rich representations – and moved to STM, which is less volatile than Sensory Memory, but of limited capacity. STM is also often viewed as a working memory, i.e. as a location where information is processed [18]. Finally, selected information is moved to LTM, which has larger capacity and longer retention periods.[1]

Beginning in the 1950s, several studies have been conducted to determine the capacity of STM. Miller [19] reported evidence showing that the capacity of STM in humans is of 7 pieces of information. Later studies have lowered the capacity limit of STM to about 4 pieces of information (e.g. [20]). Interestingly, the limit on the number of pieces of information that STM can hold does not affect directly the *amount* of information (in an information-theoretic sense) that STM can hold. In fact, STM appears to be capable to storing *references* to concepts that are stored in LTM. Although one such reference counts as a single piece of information toward the capacity limit of STM, the amount of information it conveys can be large. For example, it has been observed that it is normally difficult for people to remember the 12-letter sequence CN NIB MMT VU SA, while most people have no problems remembering the sequence CNN IBM MTV USA, because each triplet refers to a concept stored in LTM, and can thus be represented in STM by just 4 symbols [9]. The phenomenon of the detection and use of known patterns in STM is referred to as *chunking*. The patterns stored in LTM – such as CNN or USA – are called *chunks*.

Another limit of STM is that the information it contains is retained only for a short period of time, often set to about 30 seconds by researchers [10].[2] This limit can be extended by performing *maintenance rehearsal*, which consists in consciously repeating over and over the information that needs to be preserved. To increase the flexibility of our formalization, in our encoding we abstract from specific values for the limits of capacity and retention over time, and rather write our model in a parametric way. This makes it possible, among other things, to use our formalization to analyze the effects of different choices for these parameters, effectively allowing us to compare variants of the theory of STM.

As we mentioned earlier, LTM serves as a long-term storage of information. The literature describes several types of information that LTM has been found to store and arrange differently, ranging from information about statements which are known to be true (or false), to information about episodes of our life (see e.g. [10]). Researchers claim

[1] In fact, according to some researchers, LTM has unlimited capacity and retention periods. When information is forgotten, that is because of an indexing problem, and not because the information is actually removed from LTM.

[2] Notice however that the issue of a time limit on the information stored in STM is somewhat controversial – see e.g. [20,18]. For example, according to [18], decay is affected by variables such as the number of chunks that the user is trying to remember, and retrieval interference with similar chunks.

that, in spite of their being treated differently, the various types of information interact with each other. For example, information about episodes of our life may be used by the brain to derive knowledge about statements which are known to be true or false.

The information stored in LTM appears to serve two main purposes: (1) it can be recalled in order to perform inference by integrating it with information present in STM, and (2) it can be used to perform chunking. The conditions under which the memorization of information in LTM is triggered appear to be quite complex. Researchers have observed that repeated exposure to the same pieces of information causes them to be eventually moved from STM to LTM. Storage is more likely to occur when the exposures are spread over time. Maintenance rehearsal of information in STM can also be used to promote the transfer to LTM. Because of the amount of information stored in LTM, proper indexing plays an important role in being able to access the information at a later time. In order to improve the quality of the indexing for information that is being stored in LTM, one can replace maintenance rehearsal by *elaborative rehearsal*, which consists in consciously thinking about the information of interest and establishing links with knowledge that is already available.

## 4 A Formalization of Short-Term Memory

This section provides a brief overview of our formalization of STM. For a complete description, the reader is invited to refer to [7].

Using a common methodology in ASP-based knowledge representation, we start our formalization by condensing its natural-language description, as it is found in the literature, into a number of statements – still written in natural language, but precisely formulated. Next, the statements are encoded using ASP.

The statements describing STM are:

1. STM is a collection of symbols;
2. The size of STM is limited to $\omega$ elements;
3. Each symbol has an expiration counter associated with it, saying when the piece of information will be "forgotten";
4. New symbols can be added to STM. If a symbol is added to STM when $\omega$ elements are already in STM, the symbol that is closest to expiring is removed from STM ("forgotten"). In the case of multiple symbols equally close to expiring, one is selected arbitrarily;
5. When a symbol is added to STM its expiration counter is reset to a fixed value $\varepsilon$;
6. When maintenance rehearsal is performed, the expiration counters of all the symbols in STM are reset to a constant value $\varepsilon$;
7. At each step in the evolution of the domain, the expiration counter is decreased according to the duration of the actions performed;
8. When the expiration counter of a symbol in STM reaches zero, the symbol is removed from STM.
9. *Simplifying assumption:* only a single operation (where by operation we mean either addition of one symbol or maintenance rehearsal) can occur on STM at any given time.

We now show how the statements above are formalized in ASP. Fluent $in\_stm(s)$ says that symbol $s$ (where $s$ is a possibly compound term) is in STM; $expiration(s, e)$ says that symbol $s$ will expire (i.e. will be "forgotten") in $e$ units of time, unless the expiration counter is otherwise altered. Action $store(s)$ says that symbol $s$ is stored in STM; action $main\_rehearsal$ says that maintenance rehearsal is performed on the current contents of STM. Relation $symbol(s)$ says that $s$ is a symbol. Relation $stm\_max\_size(\omega)$ says that the size of STM is limited to $\omega$ elements; $stm\_expiration(\varepsilon)$ states that the symbols in STM expire after $\varepsilon$ units of time. Finally, in order to update the expiration counters based on the duration of the actions executed at each step, relation $dur(i, d)$ says that the overall duration of step $i$, based on the actions that took place, is $d$ units of time.

The direct effect of storing a symbol $s$ in STM is described by the following axioms.[3]

$$h(in\_stm(S), I + 1) \leftarrow \; symbol(S),\; step(I),\; o(store(S), I).$$
$$h(expiration(S, E), I + 1) \leftarrow symbol(S),\; stm\_expiration(E),\; o(store(S), I).$$

The above axioms say that the effect of the action is that $s$ becomes part of STM, and that its expiration counter is set to $\varepsilon$. The next axioms ensure that the size of STM does not exceed its limit when a new symbol is added:

$$\begin{array}{l}\neg h(in\_stm(S2), I + 1) \leftarrow \\ \qquad o(store(S1), I), \\ \qquad stm\_max\_size(MX),\; curr\_stm\_size(MX, I), \\ \qquad \text{not } some\_expiring(I),\; picked\_for\_deletion(S2, I). \\ 1\{\; picked\_for\_deletion(S2, I) \;:\; oldest\_in\_stm(S2, I)\;\}1 \leftarrow \\ \qquad o(store(S1), I), \\ \qquad stm\_max\_size(MX),\; curr\_stm\_size(MX, I), \\ \qquad \text{not } some\_expiring(I).\end{array}$$

The first axiom states that, if adding a symbol to STM would cause the STM size limit to be exceeded, then one symbol that is the closest to expiring will be removed from STM. Notice that the simplifying assumption listed among the natural language statements above guarantees that it is sufficient to remove one symbol from STM (lifting the assumption is not difficult, but would lengthen the presentation). The second axiom states that, if multiple symbols are equally close to expiring, then one is selected arbitrarily. From a practical perspective, this encoding allows to consider the evolutions to STM corresponding to all the possible selections of which symbol should be forgotten. The definition of auxiliary relation $oldest\_in\_stm$ can be found in [7]. The reader should however note that the two axioms above extend the encoding of [7] by allowing to deal with cases with symbols equally close to expiring (this situation may occur, for example, as the result of performing a maintenance rehearsal action, whose formalization is discussed later). It is worth pointing out that the second axiom can be

[3] To save space, we drop atoms formed by domain predicates after their first use and, in a few rules, use default negation directly instead of writing a separate rule for closed-world assumption. For example, if $p$ holds whenever $q$ is false and $q$ is assumed to be false unless it is known to be true, we might write $p \leftarrow \text{not } q$ instead of the more methodologically correct $\{p \leftarrow \neg q.\;\; \neg q \leftarrow \text{not } q.\}$.

easily modified to mimic the behavior formalized in [21], in which an arbitrary symbol (rather one closest to expiring) is forgotten.

The next axiom (together with auxiliary definitions) says that it is impossible for two STM-related actions to be executed at the same time.

$$\leftarrow o(A1,I), o(A2,I),\ A1 \neq A2,\ stm\_related(A1),\ stm\_related(A2).$$
$$stm\_related(store(S)) \leftarrow symbol(S).$$
$$stm\_related(maint\_rehearsal).$$

The direct effect of performing maintenance rehearsal is formalized by an axiom stating that, whenever maintenance rehearsal occurs, the expiration counters of all the symbols in STM are reset:

$$\begin{aligned} h(expiration(S,E), I+1) \leftarrow &\\ & stm\_expiration(E),\\ & h(in\_stm(S), I),\\ & o(maint\_rehearsal, I). \end{aligned}$$

The next group of axioms deals with the evolution of the contents of STM over time. Simple action theories often assume that fluents maintain their truth value *by inertia* unless they are forced to change by the occurrence of actions. In the case of fluent $expiration(s,e)$, however, the evolution over time is more complex (and such fluents are then called *non-inertial*). In fact, for every symbol $s$ in STM, $expiration(s,\varepsilon)$ holds at first, but then $expiration(s,\varepsilon)$ becomes false and $expiration(s,\varepsilon-\delta)$ becomes true, where $\delta$ is the duration of the latest step, and so on, as formalize by the axioms:

$$noninertial(expiration(S,E)) \leftarrow symbol(S),\ expiration\_value(E).$$

$$\begin{aligned} h(expiration(S,E-D), I+1) \leftarrow &\\ & expiration\_value(E),\\ & h(expiration(S,E), I),\\ & dur(I,D), E > D,\\ & \text{not } different\_expiration(S, E-D, I+1),\\ & \text{not } \neg h(in\_stm(S), I+1). \end{aligned}$$

$$\begin{aligned} different\_expiration(S,E1,I) \leftarrow &\\ & expiration\_value(E1),\\ & expiration\_value(E2),\\ & E2 \neq E1,\\ & h(expiration(S,E2), I). \end{aligned}$$

The inertia axiom, as well as axioms defining the evolution of fluent $in\_stm(s)$ depending on $expiration(s,e)$ and the auxiliary relations, can be found in [7].

## 5 A Formalization of Long-Term Memory and Chunking

In this section we describe our formalization of LTM. As mentioned earlier, we focus in particular on the learning of new chunks and on the phenomenon of chunking of the contents of STM by replacing them with references to suitable chunks from LTM.

In the discussion that follows, we distinguish between chunks, described in Section 3, and *proto-chunks*, which are used in the learning of new chunks. Like a chunk, a proto-chunk provides information about a pattern that has been observed, such as the symbols it consists of. However, differently from a chunk, a proto-chunk is also associated with information used to establish when it should be transferred to LTM and transformed into a chunk, such as the number of times it has been observed and the time elapsed from its first detection. Furthermore, proto-chunks cannot be used to perform chunking of the contents of STM.

In order to simplify the presentation, the ASP encoding shown here is focused on dealing with sequences of items, called *tokens*. (Extending the formalization to arbitrary data is not difficult.) A token is either an *atomic token*, such as a digit, or a *compound token*, which denotes a sequence of tokens. Given a sequence of tokens, the expression $seq(n, k)$, denotes the sub-sequence starting at the $n^{th}$ item and consisting of token $k$. If $k$ is atomic, then the sub-sequence is the token itself. If $k$ is compound, then the sub-sequence consists of the atomic elements of $k$. In the terminology introduced for the formalization of STM, $seq(n, k)$ is a symbol (see item (1) of Section 4).

As in the previous section, we begin our formalization by providing a natural-language description consisting of precisely formulated statements. To improve clarity, we organize the statements in various categories. Category *LTM model* contains the statements:

1. LTM includes a collection of chunks and proto-chunks;
2. A chunk is a token;[4]
3. Each chunk and proto-chunk is associated with a set of symbols, called elements;
4. A proto-chunk is associated with counters for the times it was detected and the amount of time since its first detection.

Category *chunking* consists of the statements:

5. A chunk is detected in STM if all the symbols that are associated with it are in STM;
6. When a chunk is detected in STM, the symbols associated with it are removed from STM and the corresponding chunk symbol is added to STM;
7. A symbol can be extracted from STM if it belongs to STM, or if it is an element of a chunk that can be extracted from STM;[5]
8. *Simplifying assumption:* chunks can only be detected when STM is not in use (i.e. no addition of maintenance rehearsal operations are being performed);
9. *Simplifying assumption:* at every step, at most one chunk can be detected.

Category *proto-chunk detection* contains the statements:

10. A proto-chunk is present in STM if all the symbols that are associated with it are in STM; a proto-chunk is detected in STM if it is present in STM, and the symbols associated with it have not been previously used for proto-chunk formation or detection;

[4] Note that a proto-chunk is not considered a token.

[5] This statement can of course be applied recursively.

11. When a proto-chunk is detected in STM, the counter for the number of times is incremented;
12. After a proto-chunk is detected, those particular occurrences of its elements in STM cannot be used for detection again except after performing maintenance rehearsal.
13. *Simplifying assumption:* presence of proto-chunks can only be verified when STM is not in use (see statement (8)), and no chunks are detected.

Category *proto-chunk formation* consists of the statements:

14. A new proto-chunk is formed by associating with it symbols from STM that have not yet been used for proto-chunk formation or detection;
15. A proto-chunk can only be formed if there are at least 2 suitable symbols in STM;
16. *Simplifying assumption:* when a proto-chunk is formed, it is associated with *all* the symbols from STM, which have not been previously used for proto-chunk formation or detection;
17. *Simplifying assumption:* proto-chunks can only be formed when STM is not in use (see statement (8)), no chunks are detected, and no proto-chunks are present.[6] Category *chunk learning* contains the statements:
18. A proto-chunk is replaced by a chunk after being detected at least $\tau$ times over a period of $\pi$ units of time;
19. When a proto-chunk is replaced by a chunk, the elements associated with the proto-chunk become associated with the chunk, and the proto-chunk is removed.

## 5.1 LTM Model

Category *LTM model* is formalized in ASP by defining suitable fluents and relations (the actual code is omitted to save space). In particular, fluent $in\_ltm(c)$ says that chunk $c$ is in LTM; $chunk\_element(c, s)$ says that $s$ is an element of $c$; $chunk\_len(c, l)$ says that the number of elements associated with $c$ is $l$; $is\_pchunk(c)$ says that $c$ is a proto-chunk (and is in LTM); $pchunk\_element(c, s)$ says that $s$ is an element of proto-chunk $c$; $pchunk\_len(c, l)$ says that the number of elements associated with $c$ is $l$; $times\_seen(c, n)$ says that proto-chunk $c$ was detected $n$ times; $age(c, a)$ says that the age of proto-chunk $c$ (i.e. the time since its first detection) is $a$; finally, fluent $considered(s)$ says that symbol $s$ in STM has already been used for proto-chunk formation or detection. Relation $min\_times\_promotion(\tau)$ says that a proto-chunk must be detected $\tau$ or more times before it can be transformed into a chunk; $min\_age\_promotion(\pi)$ says that $\pi$ units of time must have elapsed from a proto-chunk's first detection, before it can be transformed into a chunk.

## 5.2 Detection of Chunks

Category *chunking* of statements is formalized as follows. The detection of a chunk is encoded by auxiliary relation $detected(seq(p, c), i)$, intuitively stating that the symbols corresponding to chunk $c$ were detected, at step $i$, starting at position $p$ in the sequence of tokens stored in STM. The detection occurs, when STM is not in use as per simplifying assumption (8), by checking if there is any chunk whose components are all

[6] As a consequence, at every step, at most one proto-chunk can be formed.

in STM. If symbols corresponding to multiple chunks are available in STM, only one chunk is detected at every step, as per assumption (9). The choice of which chunk is detected is non-deterministic, and encoded using a choice rule, as follows:

$$
\begin{array}{l}
1\{\ \mathit{detected}(\mathit{seq}(P,C),I) \\
\qquad\qquad\qquad :\ \mathit{chunk}(C)\ :\ h(\mathit{in\_ltm}(C),I) \\
\qquad\qquad\qquad :\ \neg \mathit{chunk\_element\_missing}(\mathit{seq}(P,C),I)\ \}1 \leftarrow \\
\qquad \mathit{stm\_idle}(I), \\
\qquad \mathit{chunk\_detectable}(I). \\[4pt]
\mathit{chunk\_detectable}(I) \leftarrow \\
\qquad \mathit{stm\_idle}(I), \\
\qquad \mathit{chunk}(C),\ h(\mathit{in\_ltm}(C),I), \\
\qquad \mathit{position}(P), \\
\qquad \text{not}\ \mathit{chunk\_element\_missing}(\mathit{seq}(P,C),I). \\[4pt]
\neg \mathit{chunk\_element\_missing}(\mathit{seq}(P,C),I) \leftarrow \\
\qquad \text{not}\ \mathit{chunk\_element\_missing}(\mathit{seq}(P,C),I). \\[4pt]
\mathit{chunk\_element\_missing}(\mathit{seq}(P,C),I) \leftarrow \\
\qquad \mathit{chunk\_element\_instance}(P,C,I,S), \\
\qquad \neg h(\mathit{in\_stm}(S),I). \\[4pt]
\mathit{chunk\_element\_instance}(P1,C,I,\mathit{seq}(P1+P2-1,T)) \leftarrow \\
\qquad h(\mathit{chunk\_element}(C,\mathit{seq}(P2,T)),I). \\[4pt]
\neg \mathit{stm\_idle}(I) \leftarrow \\
\qquad o(A,I),\ \mathit{memory\_related}(A). \\[4pt]
\mathit{stm\_idle}(I) \leftarrow \\
\qquad \text{not}\ \neg \mathit{stm\_idle}(I).
\end{array}
$$

Relation $\mathit{detected}(\mathit{seq}(p,c),i)$, where $c$ is a chunk, says that sub-sequence $\mathit{seq}(p,c)$ was detected in STM at step $i$. When such a sub-sequence is detected in STM, we say that *chunk c was detected in STM*. Relation $\mathit{chunk\_element\_missing}(\mathit{seq}(p,c),i)$ says that, at step $i$, the sub-sequence $\mathit{seq}(p,c)$ is not contained in STM, because one or more elements of $c$ is missing. Relation $\mathit{chunk\_detectable}(i)$ says that there exists at least one chunk $c$ such that the sub-sequence $\mathit{seq}(p,c)$ is in STM for some position $p$. Relation $\mathit{chunk\_element\_instance}(p,c,i,s)$ says that, at step $i$, symbol $s$ is an item of the sub-sequence $\mathit{seq}(p,c)$.

It is interesting to note that the components of a detected chunk are allowed to be located anywhere in STM. However, *now that the model is formalized at this level of detail, one cannot help but wonder whether in reality the focus of the mechanism of chunking is on symbols that have been added more recently.* We were unable to find published studies regarding this issue.

The next three axioms state that, when a chunk is detected in STM, the symbols associated with it are replaced by the chunk symbol in STM[7], whose expiration counter is set to $\varepsilon$.

[7] Our simplifying assumption that at most one chunk can be detected at every step ensures that the number of items in STM does not increase as a result of the chunking process.

$$
\begin{aligned}
&\neg h(in\_stm(S), I+1) \leftarrow \\
&\qquad detected(seq(P,C), I), \\
&\qquad chunk\_element\_instance(P,C,I,S). \\
&h(in\_stm(seq(P,C)), I+1) \leftarrow \\
&\qquad detected(seq(P,C), I). \\
&h(expiration(seq(P,C),E), I+1) \leftarrow \\
&\qquad stm\_expiration(E), \\
&\qquad detected(seq(P,C), I).
\end{aligned}
$$

Finally, the fact that a symbol can be extracted from STM if it belongs to it, or if it is an element of a chunk that can be extracted from STM, is encoded by axioms:

$$
\begin{aligned}
&from\_stm(S,I) \leftarrow \\
&\qquad h(in\_stm(S), I). \\
&from\_stm(S,I) \leftarrow \\
&\qquad from\_stm(seq(P,C), I), \\
&\qquad chunk\_element\_instance(P,C,I,S).
\end{aligned}
$$

It is worth stressing that it is thanks to ASP's ability to encode recursive definitions that this notion can be formalized in such a compact and elegant way.

## 5.3 Detection of Proto-chunks

Next, we discuss the formalization of category *proto-chunk detection* of statements. The presence of a proto-chunk in STM is determined by the axioms:

$$
\begin{aligned}
&pchunk\_present(seq(P,C), I) \leftarrow \\
&\qquad stm\_idle(I), \\
&\qquad \text{not } chunk\_detectable(I), \\
&\qquad h(is\_pchunk(C), I), \\
&\qquad \neg pchunk\_element\_missing(seq(P,C), I). \\
&some\_pchunk\_present(I) \leftarrow \\
&\qquad pchunk\_present(seq(P,C), I).
\end{aligned}
$$

The first axiom informally states that, if proto-chunk $c$ is in LTM and the sub-sequence of all of its elements is found in STM starting from position $p$, then proto-chunk is present in STM, and starts at that position. Following simplifying assumption (13), the axiom is only applicable when STM is not in use, and no chunks are detected. The definition of relation $pchunk\_element\_missing(seq(p,c),i)$ is similar to that of relation $chunk\_element\_missing(seq(p,c),i)$, shown above. The axioms that follow are a rather straightforward encoding of statement (13), describing the conditions under which a proto-chunk is detected:

$$
\begin{aligned}
&pchunk\_detected(seq(P,C), I) \leftarrow \\
&\qquad pchunk\_present(seq(P,C), I), \\
&\qquad \text{not } \neg pchunk\_detected(seq(P,C), I). \\
&\neg pchunk\_detected(seq(P,C), I) \leftarrow \\
&\qquad pchunk\_present(seq(P,C), I), \\
&\qquad pchunk\_element\_instance(P,C,I,S), \\
&\qquad h(considered(S), I).
\end{aligned}
$$

Whenever a proto-chunk is detected, the counter for the number of times it was observed is incremented. Furthermore, the occurrences, in STM, of the symbols that it consists of are flagged so that they can no longer be used for proto-chunk detection or formation. The flag is removed when maintenance rehearsal is performed:

$$\begin{array}{l} h(times\_seen(C, N+1), I+1) \leftarrow \\ \qquad h(is\_pchunk(C), I), \\ \qquad pchunk\_detected(seq(P, C), I), \\ \qquad h(times\_seen(C, N), I). \end{array}$$

$$\begin{array}{l} h(considered(S), I+1) \leftarrow \\ \qquad pchunk\_detected(seq(P, C), I), \\ \qquad pchunk\_element\_instance(P, C, I, S), \\ \qquad h(in\_stm(S), I). \neg h(considered(S), I+1) \leftarrow \\ \qquad h(in\_stm(S), I), \\ \qquad o(maint\_rehearsal, I). \end{array}$$

## 5.4 Proto-chunk Formation

The statements in category *proto-chunk formation* are formalized as follows. First, we encode the conditions for the formation of a new proto-chunk listed in statements (14) and (15).

$$\begin{array}{l} new\_pchunk(I) \leftarrow \\ \qquad stm\_idle(I), \\ \qquad \text{not } chunk\_detectable(I), \\ \qquad \text{not } some\_pchunk\_present(I), \\ \qquad num\_avail\_sym\_in\_stm(N, I), \\ \qquad N > 1. \end{array}$$

$$\begin{array}{l} num\_avail\_sym\_in\_stm(N, I) \leftarrow \\ \qquad N\{\ avail\_sym(S, I) : symbol(S)\ \}N. \end{array}$$

$$\begin{array}{l} avail\_sym(S, I) \leftarrow \\ \qquad h(in\_stm(S), I), \\ \qquad \neg h(considered(S), I). \end{array}$$

In the axioms above, relation $new\_pchunk(i)$ intuitively states that a new proto-chunk can be formed at step $i$. Relation $num\_avail\_sym\_in\_stm(n, i)$ says that STM contains $n$ symbols that are available for chunk formation (that is, that have not been previously used for proto-chunk formation or detection). Whenever the conditions for chunk formation are met, the following axioms are used to create the new proto-chunk.

$$\begin{array}{l} h(is\_pchunk(p(I)), I+1) \leftarrow \\ \qquad new\_pchunk(I). \end{array}$$

$$\begin{array}{l} h(pchunk\_element(p(I), seq(P2 - P1 + 1, T)), I+1) \leftarrow \\ \qquad new\_pchunk(I), \\ \qquad lowest\_seq\_index(P1, I), \\ \qquad h(in\_stm(seq(P2, T)), I), \\ \qquad \neg h(considered(seq(P2, T)), I). \end{array}$$

$$\begin{array}{l} lowest\_seq\_index(P, I) \leftarrow \\ \qquad h(in\_stm(seq(P, T)), I), \\ \qquad \text{not } \neg lowest\_seq\_index(P, I). \end{array}$$

$$\begin{array}{l}\neg lowest\_seq\_index(P2, I) \leftarrow \\ \qquad P2 > P1, \\ \qquad h(in\_stm(seq(P1, T)), I).\end{array}$$

The first axiom causes fluent $is\_pchunk(c, i)$ to become true. Function term $p(\cdot)$ is used to assign a name to the new proto-chunk. The name is a function of the current step. Note that naming convention relies on the simplifying assumption that at most one proto-chunk is formed at each step. The second statement determines the elements of the new proto-chunk. The sequence of elements of a proto-chunk is indented start from position 1. However, because the contents of STM expire over time, all the symbols in STM may be associated with a position greater than 1. For this reason, in the axiom we offset the position of each symbol accordingly. The offset is determined by finding the smallest position of a symbol in STM. More precisely, the third axiom above says that $p$ is the smallest position that occurs in STM unless it is known that it is not. The last axiom states that $p$ is not the smallest position that occurs in STM if there is a symbol in STM whose position is smaller than $p$.

The next set of axioms resets the age and the counter for the times the proto-chunk was detected, and describes how the age of a proto-chunk increases from one step to the next according to the duration of the current step (see Section 6 regarding limitations of this representation, and ways to improve it). Because these fluents are *functional* (that is, each describes a function, with the value of the function encoded as a parameter of the function itself), we represent them as non-inertial fluents.[8]

$$\begin{array}{l}noninertial(times\_seen(C, N)). \\ h(times\_seen(p(I), 1), I + 1) \leftarrow new\_pchunk(I). \\ noninertial(age(C, A)). \\ h(age(p(I), 0), I + 1) \leftarrow new\_pchunk(I). \\ h(age(C, A + D), I + 1) \leftarrow \\ \qquad h(is\_pchunk(C), I),\ h(age(C, A), I), \\ \qquad dur(I, D),\ \text{not}\ \neg h(age(C, A + D), I + 1).\end{array}$$

## 5.5 Chunk Learning

Finally, we discuss the formalization of the statements in category *chunk learning*. The following axiom determines when the conditions from statement (18) are met:

$$\begin{array}{l}can\_be\_promoted(C, I) \leftarrow \\ \qquad min\_times\_promotion(Nmin), \\ \qquad min\_age\_promotion(Amin), \\ \qquad h(is\_pchunk(C), I), \\ \qquad h(times\_seen(C, N), I), \\ \qquad N \geq Nmin, \\ \qquad h(age(C, A), I), \\ \qquad A \geq Amin.\end{array}$$

[8] The non-inertial encoding appears to be more compact than the inertial encoding, in that it avoids having to write a rule explicitly stating that $f(n)$ becomes false whenever $f(n + 1)$ becomes true.

When the conditions are met, a suitable chunk is added to LTM:

$$\begin{array}{l} h(in\_ltm(C), I+1) \leftarrow \\ \quad can\_be\_promoted(C, I). \\ h(chunk\_element(C, S), I+1) \leftarrow \\ \quad h(is\_pchunk(C), I), \\ \quad can\_be\_promoted(C, I), \\ \quad h(pchunk\_element(C, S), I). \end{array}$$

The proto-chunk is removed from LTM with axioms such as the following (the other axioms are similar and omitted to save space):

$$\begin{array}{l} \neg h(is\_pchunk(C), I+1) \leftarrow \\ \quad can\_be\_promoted(C, I). \end{array}$$

## 5.6 Experiments

To demonstrate that our formalization captures the key features of the mechanisms of STM, LTM, and chunking, we subjected it to a series of psychological tests. Here we use variations of the *memory-span test*. In this type of test, a subject is presented with a sequence of digits, and is asked to reproduce the sequence [9]. By increasing the length of the sequence and by allowing or avoiding the occurrence of familiar sub-sequences of digits, one can verify the capacity limit of STM and the mechanics of LTM and chunking. Experiments using the basic type of memory-span test were described and used in [7]. Here we expand the tests by introducing a phase in which the subject learns chunks, which can be later used to memorize sequences the subject is presented with. This allows the testing of the detection and formation of proto-chunks, of the learning of chunks, and of the mechanism of chunking.

Because we are only concerned with correctly modeling the mechanisms of interest, we abstract from the way digits are actually read, and rather represent the acquisition of the sequence of digits directly as the occurrence of suitable $store(s)$ actions. Similarly, the final reproduction of the sequence is replaced by checking the contents of STM and LTM at the end of the experiment. As common in ASP, all computations are reduced to finding answer sets of suitable programs, and the results of the experiments are determined by observing the values of the relevant fluents in such answer sets.

From now on, we refer to the above formalization of STM, LTM, and chunking by $\Pi_{MEM}$. Boundary conditions that are shared by all the instances of the memory-span test are encoded by the set $\Pi_P$ of rules, shown below. The first two rules of $\Pi_P$ set the value of $\omega$ to a capacity of 4 symbols (in line with [20]) and the value of $\varepsilon$ to 30 time units. The next rule states that each step has a duration of 1 time unit. This set-up intuitively corresponds to a scenario in which STM has a 30 second time limit on the retention of information and the digits are presented at a rate of one per second. The next two rules set the value of $\tau$ to 2 detections and the value of $\pi$ to 3 units of time. The last set of rules define the atomic tokens for the experiments.

$$\begin{array}{l} stm\_max\_size(4). \\ stm\_expiration(30). \\ dur(I, 1). \end{array}$$

$$\begin{array}{l} min\_times\_promotion(2). \\ min\_age\_promotion(3). \\ token(0).\ token(1).\ \ldots token(9). \end{array}$$

The initial state of STM is such that no symbols are initially in STM. This is encoded by $\sigma_{STM}$:

$$\neg h(in\_stm(S), 0) \leftarrow symbol(S).$$

In the first instance, the subject is presented with the sequence $2, 4$. Human subjects are normally able to reproduce this sequence. Moreover, based on our model of LTM and chunking, we expect our formalization to form a proto-chunk for the sequence (intuitively, this simply means that LTM sets up the data structures needed to keep track of future occurrences of the sequence). The sequence of digits is encoded by set $SPAN_1$ of rules:

$$o(store(seq(1, 2)), 0).\ \ o(store(seq(2, 4)), 1).$$

To predict the behavior of the brain and determine which symbols will be in STM at the end of the experiment, we need to examine the path(s) in the transition diagram from the initial state, described by $\sigma_{STM}$, and under the occurrence of the actions in $SPAN_1$. As explained earlier in this paper, this can be accomplished by finding the answer set(s) of $\Pi_1 = \Pi_{MEM} \cup \Pi_P \cup \sigma_{STM} \cup SPAN_1$. It is not difficult to check that, at step 2, the state of STM is encoded by an answer set containing:

$$\begin{array}{l} h(in\_stm(seq(1, 2)), 2),\ h(in\_stm(seq(2, 4)), 2), \\ h(expiration(seq(1, 2), 29), 2),\ h(expiration(seq(2, 4), 30), 2), \end{array}$$

which shows that the sequence is remembered correctly (and far from being forgotten, as the expiration counters show). We also need to check that the proto-chunk has been correctly formed. For that, one can observe that the answer set contains:

$$\begin{array}{l} h(is\_pchunk(p(2)), 3), \\ h(pchunk\_element(p(2), seq(1, 2)), 3), \\ h(pchunk\_element(p(2), seq(2, 4)), 3), \\ h(times\_seen(p(2), 1), 3), \\ h(age(p(2), 0), 3). \end{array}$$

This shows that a proto-chunk named $p(2)$ has indeed been formed for the sequence $2, 4$. The counter for the number of times it has been detected is initially set to 1, and the age of the proto-chunk is set to 0.

Let us now consider another instance, in which the sequence of digits $2, 4$ is presented, and then, after a brief pause, maintenance rehearsal occurs. The corresponding actions are encoded by $SPAN_2 = SPAN_1 \cup \{o(maint\_rehearsal, 4)\}$. The pause is intended to provide enough time for the formation of the proto-chunk before maintenance rehearsal occurs. As explained earlier, maintenance rehearsal helps human subjects learn information. Because of the limit we have chosen for $\tau$ and $\pi$, we expect $SPAN_2$ to cause a chunk to be formed in LTM. It is not difficult to verify that our formalization exhibits the expected behavior. In fact, according to the answer set of program $\Pi_2 = \Pi_{MEM} \cup \Pi_P \cup \sigma_{STM} \cup SPAN_2$, the state of LTM at the end of the experiment is:

$$\begin{array}{l} h(in\_ltm(p(2)),7), \\ h(chunk\_element(p(2),seq(1,2)),7), \\ h(chunk\_element(p(2),seq(2,4)),7), \end{array}$$

which shows that a new chunk $p(2)$ describing the sequence $2,4$ has been added to LTM. As a further confirmation of the correctness of the formalization, it is not difficult to check that the answer set also contains:

$$h(in\_stm(seq(1,p(2))),8),$$

showing that chunking of the contents of STM occurred as soon as the new chunk became available.

In the next instance, we perform a thorough test of chunk learning and chunking by presenting the subject with a sequence that is, in itself, too long to fit in STM. We assume the subject's familiarity with the area code $806$, encoded by the set $\Gamma_1$ of axioms:

$$\begin{array}{l} h(in\_ltm(ac(lbb)),0). \\ h(chunk\_element(ac(lbb),seq(1,8)),0). \\ h(chunk\_element(ac(lbb),seq(2,0)),0). \\ h(chunk\_element(ac(lbb),seq(3,6)),0). \end{array}$$

Initially, the sequence $5,8,5$ is presented, and maintenance rehearsal is performed as before, in order to allow the subject to learn a new chunk. Then, the sequence $8,0,6-5,8,5$ is presented, where "$-$" represents a 1-second pause in the presentation of the digits. This sequence is, in principle, beyond the capacity of STM. However, if chunk learning and chunking are formalized correctly, then the sequence can be chunked using just two symbols, and thus easily fits in STM.[9] The sequence of digits is encoded by $SPAN_3$:

$$\begin{array}{l} o(store(seq(1,5)),0).\ o(store(seq(2,8)),1).\ o(store(seq(3,5)),2). \\ o(maint\_rehearsal,5). \\ o(store(seq(1,8)),7).\ o(store(seq(2,0)),8).\ o(store(seq(3,6)),9). \\ o(store(seq(4,5)),11).\ o(store(seq(5,8)),12).\ o(store(seq(6,5)),13). \end{array}$$

From a technical perspective, the experiment is made more challenging by the fact that $5,8,5$ is initially presented so that it starts at position $1$, but later it occurs as a sub-sequence of $8,0,6-5,8,5$, starting from position $4$. This allows to verify that chunk learning and chunking occur in a way that is position-independent. It is not difficult to check that at step $8$, the state of LTM predicted by our formalization includes:

$$\begin{array}{l} h(in\_ltm(p(3)),8), \\ h(chunk\_element(p(3),seq(1,5)),8), \\ h(chunk\_element(p(3),seq(2,8)),8), \\ h(chunk\_element(p(3),seq(3,5)),8), \end{array}$$

which shows that the chunk for $5,8,5$ was learned correctly. At the end of the experiment (we select step $15$ to allow sufficient time for the chunking of the final triplet to occur), the state of STM predicted by our formalization is:

[9] The 1-second pause is used to allow sufficient time for the detection of the first chunk. Subject studies have shown that chunk detection does not occur or occurs with difficulty when the stimuli are presented at too high a frequency.

$$h(in\_stm(seq(1, ac(lbb))), 15),$$
$$h(in\_stm(seq(4, p(3))), 15),$$
$$h(expiration(seq(1, ac(lbb)), 26), 15),$$
$$h(expiration(seq(4, p(3)), 30), 15).$$

As can be seen, the sub-sequence $8, 0, 6$ was chunked using the information encoded by $\Gamma_1$, while the sub-sequence $5, 8, 5$ was chunked using the learned chunk for $5, 8, 5$. Summing up, (1) a chunk for $5, 8, 5$ has been learned, and (2) the chunking of the two sub-sequences has occurred, allowing STM to effectively store a sequence of digits that is longer than $\omega$ symbols.

Although space restrictions prevent us from formalizing alternative theories of STM, LTM, and chunking, and from performing an analytical comparison, it is worth noting that even the single formalization presented here allows comparing (similar) variants of the theories corresponding to different values of parameters $\omega$, $\varepsilon$, $\tau$, and $\pi$. One could for example repeat the above experiments with different parameter values and compare the predicted behavior with actual subject behavior, thus confirming or refuting some of those variants.

## 6 Discussion and Related Work

In this paper we have continued our investigation on the viability of ASP as a tool for formalizing, and reasoning about, psychological models. Expanding the formalization from [7], we have axiomatized a substantial portion of LTM, with particular focus on the mechanisms of chunk learning and of chunking. The resulting formalization has been subjected to psychological experiments, in order to confirm its correctness and to show its power.

The formalization allows analysis and comparison of theories, and it allows one to predict the outcome of experiments, thus making it possible to design better experiments. Because of the direct executability of ASP, and the availability of ASP-based reasoning techniques, it is also possible to use the formalization in experiments involving e.g. planning and diagnosis (see e.g. [22,23,24]). Various reasons make the formalization of knowledge of this kind challenging. As we hope to have demonstrated, ASP allows one to tackle the challenges, thanks to its ability to deal with common-sense, defaults, uncertainty, non-deterministic choice, recursive definitions, and evolving domains.

*We believe it is difficult to find other languages that allow writing a formalization at the level of abstraction of the one shown here, and that are also directly executable.*

An interesting attempt in this direction was made by Cooper et al. in [21]. Their motivation was similar to ours, in that they wanted to define a language suitable for specifying cognitive models, and satisfying four key requirement: "(1) being syntactically clear and succinct; (2) being operationally well defined; (3) being executable; and (4) explicitly supporting the division between theory and implementation detail." The language proposed by Cooper et al., called Sceptic, is an extension of Prolog that introduces a forward chaining control structure consisting of rewrite rules and triggers. The rewrite rules are used to express the procedural control aspects of the program, thus keeping them separate from the formalization of the theory. An example of a Sceptic statement, used to describe memory decay in the Soar 4 architecture (see e.g. [25]), is:

$$\begin{array}{l} memory\_decay: \\ \qquad parameter(working\_memory\_decay, D), \\ \qquad wm\_match(WME), \\ \qquad wme\_is\_terminal(WME), \\ \qquad random(Z), \\ \qquad Z < D \\ \Rightarrow wm\_remove(WME). \end{array}$$

The intuition is that memory decay occurs with the probability specified by parameter $working\_memory\_decay$ (implemented using random number generation); when memory decay occurs, one terminal memory element is deleted by generating a $wm\_remove$ event. As can be seen from the structure of the Sceptic statement, the modeling approach adopted by [21] does not explicitly represent the evolution of the domain over time. Using the terminology adopted here, we would say that a particular path in the transition diagram is traversed during the execution of the program, but the path itself is not explicitly represented in the conclusions of the program. Because of that, it is difficult to analyze such path, and to compare alternative paths whenever multiple evolutions of the domain are possible (see e.g. the last memory-span test described in [7]). The approach is also likely to suffer from the known limitations inherited from Prolog – e.g. regarding the handling of defaults and uncertainty, especially when yielding multiple alternative conclusions – as well as the aspects of procedural flavor such as the importance of the order of rules within the program and of the elements of the body within a rule. One rather interesting claim made in [21] is that Sceptic allows to clearly distinguish between a psychological theory and the specification of the "implementation details", such as the definition of when certain conditions occur. To a large extent, this discussion seems to be motivated by the general lower level of abstraction of Prolog, compared to ASP, and thus does not apply to our approach. However, the discussion seems to also apply, at least in part, to the specification of boundary conditions, of which we gave a simple example earlier with the definition of the values of parameters such as $\tau$ and $\pi$. The intuition is that the specification of the theory and that of the boundary conditions should be kept clearly separated. We believe that this can be achieved by adopting one of the extensions of ASP that provide modules (see e.g. [26,27]).

In the area of cognitive modeling, quite popular is also the use of cognitive architectures, especially Soar [25] and ACT-R [28]. These are quite sophisticated architectures, not dissimilar from agent architectures, which often cover all the aspects of information processing in the brain, from input acquisition, to output generation. Their parametric and modular structure makes it possible to test alternative theories by replacing suitable modules or adjusting parameter values. In both Soar and ACT-R, the stress seems to be on the definition of a specific architecture, with assumptions being made on the way certain processes occur. Thus, formalization of theories that are not compatible with those assumptions seems difficult. On the other hand, our approach does not enforce any particular assumption, and allows greater freedom of formalization. Furthermore, in the area of cognitive architectures there appears to be no particular interest in, nor means for, the direct theoretical analysis of the properties of the architecture and their components. Whereas we have stressed that ASP allows for inspection of the formalization and comparison of alternative axiomatizations, in these architectures the main tool for analysis appears to be the simulation of the architecture itself, and the analysis of the experimental results (see e.g. [29]; [21] constitutes a remarkable exception).

As a concluding remark, let us stress that the formalization presented here could be made richer in various ways. Our formalization of fluents involving actual time, such as $age$, is rather simplified. A more sophisticated axiomatization can be obtained by adopting the techniques from e.g. [30]. This would also allow to model continuous decay. Several simplifying assumptions made earlier could be lifted by introducing additive fluents [31]. Furthermore, there may be benefits in describing the interaction between STM and LTM by using an agent architecture framework such as the one in [32].

## Acknowledgments

The authors to thank Esra Erdem for the useful comments and suggestions.

## References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
2. 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, Heidelberg (1999)
3. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
4. Baral, C., Gelfond, M.: Reasoning about Intended Actions. In: Proceedings of the 20th National Conference on Artificial Intelligence, pp. 689–694 (2005)
5. Son, T.C., Sakama, C.: Negotiation Using Logic Programming with Consistency Restoring Rules. In: 2009 International Joint Conferences on Artificial Intelligence, IJCAI (2009)
6. McCarley, J.S., Wickens, C.D., Gob, J., Horrey, W.J.: A Computational Model of Attention/Situation Awareness. In: Proceedings of the 46th Annual Meeting of the Human Factors and Ergonomics Society (2002)
7. 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)
8. Formalizing and Compiling Background Knowledge and its Applications to Knowledge Representation and Question Answering. In: AAAI 2006 Spring Symposium Series (2006)
9. Kassin, S.: Psychology in Modules. Prentice Hall, Englewood Cliffs (2006)
10. Nevid, J.S.: Psychology: Concepts and Applications, 2nd edn. Houghton Mifflin Company, Boston (2007)
11. Niemela, I., Simons, P.: Extending the Smodels System with Cardinality and Weight Constraints. In: Logic-Based Artificial Intelligence, pp. 491–521. Kluwer Academic Publishers, Dordrecht (2000)
12. Gelfond, M., Lifschitz, V.: Action Languages. Electronic Transactions on AI 3(16) (1998)
13. Gelfond, M.: Representing Knowledge in A-Prolog. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond. LNCS (LNAI), vol. 2408, pp. 413–451. Springer, Heidelberg (2002)

14. Balduccini, M., Gelfond, M., Nogueira, M.: A-Prolog as a tool for declarative programming. In: Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE 2000), pp. 63–72 (2000)
15. Delgrande, J.P., Grote, T., Hunter, A.: A general approach to the verification of cryptographic protocols using answer set programming. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 355–367. Springer, Heidelberg (2009)
16. Thielscher, M.: Answer Set Programming for Single-Player Games in General Game Playing. In: 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009), pp. 327–341 (September 2009)
17. Atkinson, R.C., Shiffrin, R.M.: The Control of Short-Term Memory. Scientific American 225, 82–90 (1971)
18. Card, S.K., Moran, T.P., Newell, A.: The Psychology of Human-Computer Interaction. L. Erlbaum Associates Inc., Mahwah (1983)
19. Miller, G.A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review 63, 81–97 (1956)
20. Cowan, N.: The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity. Behavioral and Brain Sciences 24, 87–185 (2000)
21. Cooper, R.P., Farringdon, J., Fox, J., Shallice, T.: A Systematic Methodology for Cognitive Modelling. Artificial Intelligence 85, 3–44 (1996)
22. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138, 39–54 (2002)
23. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Journal of Theory and Practice of Logic Programming (TPLP) 3(4-5), 425–461 (2003)
24. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence 47(1-2), 183–219 (2006)
25. Laird, J.E., Newell, A., Rosenbloom, P.S.: SOAR: An Architecture for General Intelligence. Artificial Intelligence 33, 1–64 (1987)
26. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)
27. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. Journal of Artificial Intelligence Research 35, 813–857 (2009)
28. Anderson, J.R., Bothell, D., Byrne, M.D., Douglass, S., Lebiere, C., Qin, Y.: An Integrated Theory of the Mind. Psychological Review 111(4), 1036–1060 (2004)
29. Halverson, T., Gunzelmann, G., Moore Jr., L.R., Dongen, H.V.: Modeling the Effects of Work Shift on Learning in Mental Orientation and Rotation Task. In: 10th International Conference on Cognitive Modeling (ICCM 2010) (August 2010)
30. Chintabathina, S., Gelfond, M., Watson, R.: Modeling Hybrid Domains Using Process Description Language. In: Proceedings of ASP 2005 – Answer Set Programming: Advances in Theory and Implementation, pp. 303–317 (2005)
31. Lee, J., Lifschitz, V.: Additive Fluents. In: Provetti, A., Son, T.C. (eds.) Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning. AAAI 2001 Spring Symposium Series (March 2001)
32. Balduccini, M., Gelfond, M.: The AAA Architecture: An Overview. In: AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents, AITA 2008 (March 2008)

# A Temporally Expressive Planner Based on Answer Set Programming with Constraints: Preliminary Design

Forrest Sheng Bao, Sandeep Chintabathina, A. Ricardo Morales, Nelson Rushton, Richard Watson, and Yuanlin Zhang

Texas Tech University, USA

**Abstract.** Recently, a new language $\mathcal{ACC}$ was proposed to integrate answer set programming (ASP) and constraint logic programming (CLP). In this paper, we show that $\mathcal{ACC}$ can be employed to build a temporally expressive planner for PDDL2.1. Compared with the existing planners, the new approach put less restrictions on the planning problems and is easy to extend with new features like PDDL axioms, thanks to the expressive power of $\mathcal{ACC}$. More interestingly, it can also leverage the inference engine for $\mathcal{ACC}$ which has the potential to exploit the best reasoning mechanisms developed in the ASP, SAT and CP communities.

## 1 Introduction

In a realistic setting, it is hard to have a clear separation of planning and scheduling problems. Planning problems ask what actions to take to achieve a goal while scheduling problems involve the decision of when to take the actions. The last two decades have seen significant efforts to investigate the planning and scheduling problems (e.g., IxTeT [1], INOVA [2], EUROPA [3] and ZENO [4]). In 2003, as an extension to the PDDL language, PDDL2.1 introduced durative actions to represent temporal information of a domain [5]. Several solvers, e.g., SGPlan [6] and CPT [7], had been developed for PDDL2.1. However, Cushing et al. [8] observed that those solvers are capable of handling only a subset of PDDL2.1, and they can not find plans for *temporally expressive problems*. A planning problem is temporally expressive if any plan of the problem requires concurrent actions.

To address the weakness of the existing solvers, Coles et al. [9] developed a new system Crikey for temporally expressive problems. Huang et al. [10] developed a SAT based approach, and Hu [11] proposed a constraint satisfaction problems based planner.

As an ad hoc system, it is hard for Crikey to fully exploit the the existing inference engines (e.g., SAT and constraint solvers). Another drawback of this approach is its extensibility to handle new features (e.g., PDDL axioms [12]). Huang et al's method needs the discretization of time. It may lead to large problem instances and therefore the planner may be very slow because of the huge search space. This approach cannot take advantage of effective constraint reasoning algorithms either. Hu's translation works only for planning problems without continuous durative actions or PDDL axioms.

Our approach is based on a general purpose language $\mathcal{ACC}$ we proposed recently. The language used in this paper is a revised version of the $\mathcal{ACC}$ in [13]. $\mathcal{ACC}$ integrates answer set programming (ASP) and constraint logic programming (CLP). It is well known that ASP provides elegant descriptions for planning problems while CLP can effectively represent and reason with (temporal) constraints. In this paper we will show how to employ $\mathcal{ACC}$to encode problems represented by PDDL2.1. Preliminary materials about PDDL2.1 and $\mathcal{ACC}$ are given in Sect. 2, the $\mathcal{ACC}$ encoding of PDDL2.1 planning problems is given in Sect. 3, and the paper is concluded in Sect. 4.

## 2 Preliminaries

In this section, we cover the basics of PDDL2.1 and $\mathcal{ACC}$ language.

### 2.1 PDDL2.1 Planning Instances

In this section, we will introduce a compact definition of PDDL2.1. It is slight different from standard PDDL2.1 syntax for convenience.

A *signature* is a tuple $\langle N, R, X\rangle$ where $N = \{a_1, \ldots, a_l\}$ is a set, each element of which is called an *action name*, $R = \{p_1, \ldots, p_m\}$ is a set, each element of which is called a *propositional letter*, and $X = \{x_1, \ldots, x_n\}$ is a set, each element of which is called a *variable.* $\forall x \in X, x \in \mathbb{R} \cup \{\perp\}$ where $\mathbb{R}$ is the set of real numbers and $\perp$ means undefined. A propositional letter or its negation is called a *literal.*

*Arithmetic expressions* are formed from variables, numbers and arithmetic operators. *Arithmetic constraints* are of the form $exp1 \circ exp2$ where $\circ \in \{=, \geq, \leq\}$ and $exp1$ and $exp2$ are numerical expressions. For example, $x_1 \cdot x_2 - x_3 \leq 5$ is an arithmetic constraint. Arithmetic constraints of the form $x = a$, where $x$ is a variable and $a$ a number, are called *simple constraints.*

A *formula* is defined as follows.

- A propositional letter is a *formula.*
- An arithmetic constraint is a *formula.*
- If $F$ and $G$ are formulas, $F \wedge G$ and $F \vee G$ are *formulas.*
- If $F$ is a formula, $\neg F$ is *formula.*

As an example, $p_1 \wedge (x_1 + x_2 \leq 3) \vee \neg p_2$ is a formula.

A *simple action* is a tuple $\langle name, \pi, \epsilon\rangle$:

1. $name \in N$.
2. $\pi$ is a formula. It is called the *preconditions* of the action.
3. $\epsilon$ is a set of conjunctions of literals and statements of the form $lv \diamond exp$ where $lv$ is a variable, $\diamond \in \{assign, increase, decrease, scaleup, scaledown\}$, and $exp$ is an arithmetic expression. Each conjunct is called an *effect* of the action. The statement of the form $lv \diamond exp$ is called an *assignment*

*statement.* It is called *direct assignment* if $\diamond = assign$, called *additive assignment* if $\diamond \in \{increase, decrease\}$, and called *scaling assignment* if $\diamond \in \{scaleup, scaledown\}$.

As an example, consider the action to pour water from one jug (jug1) to another jug (jug2). The precondition can be that jug1 is not empty. The effects can be that the amount of water in jug1 becomes 0 while the amount of water in jug2 increases by the amount of water in jug1. This action "pour" can be represented as

$$\langle pour, (amount\_jug1 > 0),\\ \{(amount\_jug2\ increase\ amount\_jug1) \ \wedge \ (amount\_jug1\ assign\ 0)\}\rangle.$$

Given an action $a$, we use $a.\pi$ and $a.\epsilon$ to denote the preconditions and effects of $a$. For example, $pour.\pi = amount\_jug1 > 0$ and

$$pour.\epsilon = \{amount\_jug2\ increase\ amount\_jug1 \wedge amount\_jug1\ assign\ 0\}$$

A *durative action* is a tuple $\langle name, c, \pi, \epsilon\rangle$:

1. $name \in N$,
2. $c = \langle c_s, c_e\rangle$, where $c_s$ and $c_e$ are formulas which may involve the special variable `?duration` that is not in $X$. $c$ is called *duration constraints.* $c_s$ and $c_e$ are called *start duration constraints* and *end duration constraints*, respectively.
3. $\pi = \langle \pi_s, \pi_o, \pi_e\rangle$, where $\pi_s, \pi_o, \pi_e$ are formulas. $\pi$ is called the *conditions.* $\pi_o$ is called *invariant* over the duration of the action.
4. $\epsilon = \langle \epsilon_s, \epsilon_c, \epsilon_e\rangle$, called *effects.*
   - $\epsilon_s$ (and $\epsilon_e$ respectively) is a set of conjunctions of literals and statements of the form $lv \diamond exp$ where $lv$ is a variable, $\diamond \in$ {*assign, increase, decrease, scaleup, scaledown*}, and $exp$ is an arithmetic expression.
   - $\epsilon_c$ is a conjunction of statements of the form $lv\ increase\ exp * \Delta t$ or $lv\ decrease\ exp * \Delta t$ where $\Delta t$ is understood as the time lapsing from the start of the action. $\epsilon_c$ is called *continuous effects.*

We use dot to access the elements of a durative action. For example, given a durative action $da$, $da.\epsilon_s$ refers to the effects $\epsilon_s$ of $da$.

Suppose in the "pour" action we discussed above, the water is poured into a jug at a constant speed. Then the "pour" action can be regarded as a durative action in which the duration is the amount of water in jug1 divided by the pouring speed. Then,

$$\begin{aligned} pour.c &= \langle ?\texttt{duration} = amount\_jug1/speed, True\rangle,\\ pour.\pi_s &= pour.\pi_o = pour.\pi_e = amount\_jug1 > 0,\\ pour.\epsilon_s &= \emptyset,\\ pour.\epsilon_e &= \{amount\_jug1\ assign\ 0\},\\ pour.\epsilon_c &= \{amount\_jug2\ increase\ speed * \Delta t \wedge amount\_jug1\ decrease\ speed * \Delta t\} \end{aligned}$$

where *speed* is the pouring speed.

If $\epsilon_c \neq \emptyset$, the action is called a *continuous action*. Otherwise, it is called a *discrete action*.

Intuitively, $\pi_s$ must hold before the start of the action, $\pi_o$ over the duration, and $\pi_e$ before the end. $\epsilon_s$ is the effect of the start of the action, $\epsilon_e$ the effect of the end of the action, and $\epsilon_c$ the continuous effect over the duration of the action.

A *planning instance* is a tuple $I = \langle N, R, X, Init, Goal, A\rangle$ where $A$ is a set of simple and durative actions, $Init$, called *initial state*, is a set of propositional letters and simple constraints; $Goal$, called *goal state*, is a formula.

## 2.2 Plans of PDDL2.1 Planning Instances

A *state* is a tuple $(t, s, \mathbf{b})$ where $t$ is a real number denoting the time, $s$ is a set of propositional letters, and $\mathbf{b}$ is a vector of values $\langle b_1, \ldots, b_n\rangle$, implying $x_1 = b_1, \cdots, x_n = b_n$, where $x_1, \cdots, x_n$ are variables in $X$ as defined in Sect. 2.1.

A formula $\alpha$ is *satisfied* by a state $(t, s, \mathbf{b})$ if it is true under the assignment $p = T(true)$ for all $p \in s$ and $p = F(false)$ for all $p \notin s$, and $X = \mathbf{b}$, i.e., $x_1 = b_1, \ldots, x_n = b_n$. As an example, the formula $p \wedge \neg q \vee (x_1 + x_2 \leq 5)$ is satisfied by the state $(5, \{p\}, \langle 2.0, 3.0\rangle)$ because it is true under the assignment that $p = T$, $q = F$, and $\langle x_1, x_2\rangle = \langle 2.0, 3.0\rangle$.

A *plan* is a set of timed pairs of either the form $(t, a[t'])$ *or* $(t, a)$ where $t$ is a real number for time, $a$ an action name and $t'$ a rational value specifying the duration when the action is durative.

A *happening sequence* $\langle t_i\rangle_{i=0\ldots k}$ of a plan $P$, denoted by $H(P)$, is an ordered sequence of the times in $P$. For any time instant $t$, the set of actions $E_t = \{a \mid (t, a) \in P \textit{ or } (t, a[t']) \in P \textit{ or } (t - t', a[t']) \in P\}$ of a plan $P$ is called a *happening* at time $t$.

The *induced plan* of a plan $P$, denoted by *simplify(P)*, is the union of

- $\{(t, a) \mid (t, a) \in P\}$
- $\{(t, a_s[t']), (t + t', a_e[t']) \mid (t, a[t']) \in P\}$
- $\{((t_i + t_{i+1})/2, a_{inv}) \mid (t, a[t']) \in P, t_i, t_{i+1} \in H(P),\ \textit{and}\ t \leq t_i < t + t'\}$

where $a_s$, $a_e$ and $a_{inv}$ are new simple actions that are defined in terms of the discrete durative action $a$, as follows

- $a_s$: $a_s.\pi = a.\pi_s \cup a.c_s$ and $a_s.\epsilon = a.\epsilon_s$.
- $a_e$: $a_e.\pi = a.\pi_e \cup a.c_e$ and $a_e.\epsilon = a.\epsilon_e$.
- $a_{inv}$: $a_{inv}.\pi = a.\pi_o$ and $a_{inv}.\epsilon = \emptyset$.

Intuitively, a discrete durative action $a$ is replaced (equivalently) by two simple actions $a_s$ and $a_e$ and a set of occurrences of simple action $a_{inv}$ to monitor whether the invariant holds from over the duration of $a$.

We need the following notations to define the continuous update function. First we introduce a function

$$signed(\diamond, exp) = \begin{cases} (exp) & \text{if } \diamond = increase \\ (-(exp)) & \text{if } \diamond = decrease \\ (exp) & \text{if } \diamond = scaleup \\ (1/(exp)) & \text{if } \diamond = scaledown \end{cases}$$

For example, let an assignment statement be

$$amount(t1)\ increase\ 5,$$

then we have

$$signed(increase, 5) = 5.$$

Given an additive assignment $lv \diamond exp$, its *derived function* is $f : (\mathbb{R} \cup \{\bot\})^n \rightarrow (\mathbb{R} \cup \{\bot\})^n$ such that

$$f(x_1, ..., x_{i-1}, lv, x_{i+1}, ..., x_n) = (x_1, ..., x_{i-1}, lv + signed(\diamond, exp), x_{i+1}, ..., x_n).$$

The *updating function* generated from a set of additive assignments is the composition of all derived functions from the assignments.

Let $C$ be a set of continuous effects for a planning instance $I$, and $St = (t, s, \mathbf{b})$ a state. The *continuous update function* defined by $C$ for state $St$ is the function $f : \mathbb{R} \rightarrow \mathbb{R}^n$, such that:

$$\frac{df}{dt} = g \tag{1}$$

and

$$f(0) = \mathbf{b} \tag{2}$$

where $g$ is the updating function generated from the additive assignments

$$\{(lv \diamond exp) \mid (lv \diamond exp * \Delta t) \in C\}$$

A continuous effect $Q$ of timed pairs $(t, a[d])$ is *active* in interval $(t_i, t_{i+1})$ if $(t_i, t_{i+1}) \subseteq [t, t + d]$. Given a planning instance $I$ and a plan $P$, define $Cts(P) = \{(C, t_i, t_{i+1}) \mid C$ is the set of active continuous effects over $(t_i, t_{i+1})$ and $t_i, t_{i+1} \in$ *H(simplify(P))* $\}$, i.e., $Cts(P)$ consists of the active continuous effects for each time interval $(t_i, t_{i+1})$.

Given a planning instance $I$ that includes continuous durative actions, and a plan $P$, let $\langle t_i \rangle_{i=1...k}$ be the happening sequences for *simplify(P)*, and $S_0$ be the initial state. The *trace* for $P$ is the sequence of states $\langle S_i \rangle_{i=0..k+1}$ defined as follows:

1. If there is no active continuous effect in $(t_i, t_{i+1})$, $S_{i+1}$ is the resulting state of applying the happening at $t_i$ in *simplify(P)* to $S_i$.
2. If there are active continuous effects in $(t_i, t_{i+1})$, let $T_i$ be the result of substituting $f(t_{i+1} - t_i)$ for the variables of $S_i$, where $f$ is the continuous update function defined by the active effects in $(t_i, t_{i+1})$. $S_{i+1}$ is the resulting state of applying the happening at $t_i$ in *simplify(P)* to the state $T_i$. If $f$ is undefined, $S_{i+1}$ is undefined.

Intuitively, here we define the *state transition* given a set of timed actions. In the case where continuous effects are involved, we apply continuous effects to $S_i$ to obtain $T_i$ by updating the corresponding variables in $S_i$, and then apply the actions *simplify(P)* to $T_i$ to obtain $S_{i+1}$.

Given a planning instance $I$ and a plan $P$, for each $(C, t_i, t_{i+1}) \in Cts(P)$, let $f_i$ be the continuous update function defined by $C$. $P$ is *invariant safe* if for each invariant constraint $Q$ of $(t, a[d])$, and for each $f_i$ such that $(t_i, t_{i+1}) \subseteq (t, t+d)$, and for all $x \in (t_i, t_{i+1})$, $Q'$ is satisfied by $S_i'$ where $Q'$ and $S_i'$ are the result of substituting $f_i(x - t_i)$ for $X$ in $Q$ and $X$ in $S_i$ respectively.

Given a planning instance $I$ and a plan $P$, $P$ is *executable* if the trace of $P$ is defined and the trace is invariant safe, and $P$ is *valid* if it is executable and the final state of the trace satisfies the goal in $I$.

A *PDDL2.1 problem* is to find a valid plan for a planning instance.

## 2.3 Syntax of $\mathcal{ACC}$

$\mathcal{ACC}$ is a typed language. Its programs are defined over a *sorted* signature $\Sigma$, consisting of *sorts*, and properly typed *predicate symbols*, *function symbols* and *variables*.

By a *sort*, we mean a non-empty countable collection of strings over some fixed alphabet. Strings of a sort $S$ are referred as *object constants* of $S$. Each variable takes on values of a unique sort.

A *term* of $\Sigma$ is either a constant, a variable, or an expression $f(t_1, \ldots, t_n)$ where $f$ is an $n$-ary function symbol, and $t_1, \ldots, t_n$ are terms of proper sorts.

An *atom* is of the form $p(t_1, \ldots, t_n)$ where $p$ is an $n$-ary predicate symbol, and $t_1, \ldots, t_n$ are terms of proper sorts. A *literal* is either an atom or its negation.

Sorts of $\mathcal{ACC}$ can be partitioned as *regular* and *constraint*. Intuitively, a sort is declared to be a constraint sort if it is a large (often numerical) set with primitive constraint relations, e.g., $\leq$.

For example, steps or times in a planning problem can be constraint sorts. In our examples above, *jug* is a regular sort.

A function $f : S_1 \times \cdots \times S_n \to S$, where $S_1, \ldots, S_n$ are regular sorts and $S$ is a constraint sort, is called a *bridge function*. We introduce a special predicate symbol *val* where $val(f(t), y)$ holds if $y$ is the value of function symbol $f$ for argument $t$. For simplicity, we write $f(t) = y$ instead of $val(f(t), y)$. The domain and range of a bridge function $f$ are denoted as $domain(f)$ and $range(f)$ respectively.

The partitioning of sorts induces a natural partition of predicates and literals of $\mathcal{ACC}$. *Regular predicates* denote relations among objects of regular sorts; *constraint predicates* denote primitive numerical relations on constraint sorts; predicate *val* is called the *bridge predicate*; all the other predicates are called *defined predicates.*

$\mathcal{ACC}$ uses *declarations* to describe signatures. A regular/constraint sort declaration consists of the keyword `#rsort`/`#csort` followed by a list of sort names, like:

$$\#rsort\ sort\_name_1, \ldots, sort\_name_n$$

or

$$\#csort\ sort\_name_1, \ldots, sort\_name_n$$

A *predicate declaration* is an expression of the form:

$$\#pred\_name(sort\_name_1, \ldots, sort\_name_n)$$

where $pred\_name$ is an $n$-ary predicate symbol and $sort\_name_1, \ldots, sort\_name_n$ is a list of sort names corresponding to the types of the arguments of $pred\_name$.

A *bridge function declaration* is an expression of the form:

$$\#func\_name(sort\_name_1, \ldots, sort\_name_n) : sort\_name$$

where $func\_name$ is an $n$-ary bridge function symbol, $sort\_name_1, \ldots,$ $sort\_name_n$ is a list of sort names corresponding to the types of the arguments of $func\_name$, and $sort\_name$ is the sort of $func\_name$.

For simplicity, we allow multiple predicate and bridge function declarations in the same line preceded by a single # symbol.

A *rule* in $\mathcal{ACC}$ is an expression of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$$

where $l$'s are literals of $\Sigma$.

A program $\Pi$ of $\mathcal{ACC}$ is a collection of rules and declarations over a signature $\Sigma$. Every predicate and sort used in a rule of $\Pi$ **must** appear in a declaration statement.

## 2.4 Semantics of $\mathcal{ACC}$

Terms, literals, and rules are called *ground* if they contain no variables and no symbols for arithmetic functions.

A *bridge assignment* is a mapping from every bridge function symbol $f$ and element $x \in domain(f)$ to a value $y \in range(f)$. Such an assignment will be represented by the set of atoms $f(x) = y$. A set $S$ of ground literals is an *answer set* of an $\mathcal{ACC}$ program $\Pi$ if there is a bridge assignment $V$ such that $S$ is an answer set of the ASP program $\Pi \cup V$.

An *extended literal* is an expressions of the form $l$ and $not\ l$ where $l$ is a literal. We identify every answer set $A$ of $\Pi$ with the *extended answer set*:

$$A \cup \{not\ p \mid p \text{ is an atom of } \Pi \text{ and } p \notin A\}$$

# 3 Translation of a PDDL Problem to an $\mathcal{ACC}$ Program

Given a planning instance $I = \langle N, R, X, Init, Goal, A\rangle$, we form an $\mathcal{ACC}$ program as follows.

### 3.1 Sorts

To represent the variables $\{x_1, x_2, \ldots, x_n\}$ from $X$, we need a regular sort $\mathcal{S}_v = \{1, \ldots, n\}$ which denotes the indexes of variables. To represent planning steps, we also need a regular sort $\mathcal{S}_s = \{0, 1, \ldots, l_s\}$. To represent action names from $N$, we need a regular sort $\mathcal{S}_a = \{a_1, \ldots, a_N\}$. We also need a constraint sort $\mathcal{R}$ for real numbers.

### 3.2 Predicates

The predicates of the $\mathcal{ACC}$ program are $\{p/1 \mid p \in R\}$. For every $p \in R$, $p(S)$ means the value of $p$ at step $S$. More will be added later.

### 3.3 Variables and Time

For variables $X$, we introduce a bridge function $m(I, S) : Var$, where $I$ is of sort $\mathcal{S}_v$, $S$ of sort $\mathcal{S}_s$, and $Var$ is a variable of the constraint sort $\mathcal{R}$.

For each step of $\mathcal{S}_s$, we introduce a bridge function $at(S) : T$, where $S$ is of $\mathcal{S}_s$ and $T$ is a non-negative real number. It means that step $S$ occurs at $T$.

For each step, we introduce a predicate $occurs(Name, S)$, where $Name$ is of sort $\mathcal{S}_a$ and $S$ is of sort $\mathcal{S}_s$, meaning that action $Name$ occurs at step $S$.

### 3.4 Formulas

Given a step $S$ of sort $\mathcal{S}_s$ and a formula $\alpha$, we define the following two operations on $\alpha$. $\alpha^T(S)$ is an atom and $\alpha^P(S)$ is a set of the rules to be added into $\mathcal{ACC}$.

- if $\alpha$ is a propositional letter $p$,
  $\alpha^T(S) \equiv p(S)$.
  $\alpha^P(S) = \emptyset$.
- if $\alpha$ is an arithmetic constraints $c$, let $\{x_1, ..., x_i\}$ be all the variables of $c$, without loss of generality.
  $\alpha^T(S) \equiv m(1, S) = X_1, ..., m(i, S) = X_i, c'$. where $c'$ is the result of substituting $X_j$ for $x_j$ (for all $j \in [1..i]$) in $c$.
  $\alpha^P(S) = \emptyset$.
- if $\alpha = \alpha_1 \wedge \alpha_2$, introduce a new predicate $p/1$.
  $\alpha^T(S) \equiv p(S)$.
  $\alpha^P(S) = \alpha_1^P(S) \cup \alpha_2^P(S) \cup \{p(S) \leftarrow \alpha_1^T(S), \alpha_2^T(S).\}$.
- if $\alpha = \alpha_1 \vee \alpha_2$, introduce a new predicate $p/1$.
  $\alpha^T(S) \equiv p(S)$
  $\alpha^P(S) = \alpha_1^P(S) \cup \{p(S) \leftarrow \alpha_1^T(S).\ \ p(S) \leftarrow \alpha_2^T(S).\}$
- if $\alpha = \neg\alpha_1$, introduce new predicates $p/1$.
  $\alpha^T(S) \equiv p(S)$.
  $\alpha^P(S) = \alpha_1^P(S) \cup \{p(S) \leftarrow \neg\alpha_1^T(S).\}$.

Example: Given $S$ and a formula $\alpha = \neg((p \vee q) \wedge (x_2 - x_1 > 3))$, $\alpha^T(S) = p''''(S)$, and $\alpha^P(S)$ is

$$\begin{aligned}
p'(S) &\leftarrow p(S).\\
p'(S) &\leftarrow q(S).\\
p'''(S) &\leftarrow p'(S), m(1,S) = X_1, m(2,S) = X_2, X_2 - X_1 > 3.\\
p''''(S) &\leftarrow \neg p'''(S).
\end{aligned}$$

## 3.5 Initial and Goal States

For any propositional letter $p$ in the initial state, we have the fact

$$p(0).$$

For any propositional letter $q$, we have the following closed world assumption

$$\neg q(0) \leftarrow \mathit{not}\ q(0).$$

For every simple constraint $x_i = c$ in the initial state, we have the fact

$$m(i,0) = c.$$

Let $\alpha$ be the goal formula, we need the $\mathcal{ACC}$ rules $\alpha^P(S)$ and the rules

$$\mathit{goal} \leftarrow \alpha^T(S).$$

$$\leftarrow \mathit{not}\ \mathit{goal}.$$

## 3.6 Simple Actions

As defined in Sect. 2.1, a *simple action* is a tuple $a = \langle \mathit{name}, \pi, \epsilon \rangle$.

For precondition $\pi$ of action *name*, we add $\pi^P(S)$ and this $\mathcal{ACC}$ rule

$$\leftarrow \mathit{occurs}(\mathit{name}, S), \mathit{not}\ \pi^T(S).$$

For each positive literal $p \in \epsilon$,

$$p(S+1) \leftarrow \mathit{occurs}(\mathit{name}, S).$$

For each negative literal $\neg p \in \epsilon$,

$$\neg p(S+1) \leftarrow \mathit{occurs}(\mathit{name}, S).$$

For every propositional letter $p \in R$, we have the inertia law

$$\begin{aligned}
p(S+1) &\leftarrow p(S), \mathit{not}\ \neg p(S+1).\\
\neg p(S+1) &\leftarrow \neg p(S), \mathit{not}\ p(S+1).
\end{aligned}$$

Let $x_i \diamond \mathit{exp}$, where $x_i \in X$ is a variable and $\diamond$ is an assignment operator, be an assignment in the effects of a simple action $a_j$. The assignment statements on variable $x_i$ will be encoded as follows.

1. If $\diamond$ is *assign*, then the rule encoding the effect is:

$$m(i, S+1) = exp' \leftarrow occurs(a_j, S), m(1, S) = X_1, \ldots, m(n, S) = X_n.$$

where $X_j$ $(1 \leq j \leq n)$ is the value of $x_j$ at step $S$, and $exp'$ results from substituting the variables in $exp$ by $X_i$'s.

2. If $\diamond$ is *scaleup* or *scaledown*, we need to first define the scaling contribution $(s\_contribution(i, a_j, S))$ of action $a_j$ to $x_i$ at step $S$.

$$\begin{aligned} s\_contribution(i, a_j, S) = signed(\diamond, exp') \leftarrow\ & occurs(a_j, S), \\ & m(1, S) = X_1, \\ & \vdots \\ & m(n, S) = X_n. \\ s\_contribution(i, a_j, S) = 1 \leftarrow\ & not\ occurs(a_j, S). \end{aligned}$$

where $exp'$ is defined as before.
For any action $a_k$ which does not have any scaling effect on $x_i$, we have

$$s\_contribution(i, a_k, S) = 1.$$

The value of $x_i$ at $S$ is the production of all contributions to it.

$$\begin{aligned} m(i, S+1) = Y \leftarrow\ & scaled(i, S), \\ & s\_contribution(a_1, S) = C_1, \\ & \vdots \\ & s\_contribution(a_N, S) = C_N, \\ & m(i, S) * \prod_{k=1}^{N} C_k = Y. \end{aligned}$$

3. If $\diamond$ is *increase* or *decrease*, we need to first define the additive contribution $(contribution(i, a_j, S))$ of action $a_j$ to $x_i$ at step $S$.

$$\begin{aligned} contribution(i, a_j, S) = signed(\diamond, exp') \leftarrow\ & occurs(a_j, S), \\ & m(1, S) = X_1, \\ & \vdots \\ & m(n, S) = X_n. \\ contribution(i, a_j, S) = 0 \leftarrow\ & not\ occurs(a_j, S). \end{aligned}$$

For any action $a_k$ where $x_i$ does not appear in an additive assignment, the contribution of this action to $x_i$ is 0 at any step.

$$contribution(i, a_k, S) = 0.$$

The value of $x_i$ at $S$ is the accumulation of all contributions to it.

$$\begin{aligned} m(i, S+1) = Y \leftarrow\ & added(i, S), \\ & contribution(a_1, S) = C_1, \\ & \quad\vdots \\ & contribution(a_N, S) = C_N, \\ & m(i, S) + \sum_{k=1}^{N} C_k = Y. \end{aligned}$$

Note that our treatment of the additive assignments is similar to the approach employed by Lee and Lifschitz (2003) in action language C+.

The auxiliary predicates used above are defined as follows: $assigned(i, S)$ holds if there is a direct assignment to $x_i$ at step $S$; $scaled(i, S)$ holds if there is a scaling assignment to $x_i$ at step $S$; and $added(i, S)$ holds if there is an additive assignment to $x_i$.

For every action $a$ whose effects contain a direct assignment to $x_i$, we have

$$assigned(i, S) \leftarrow occurs(a, S).$$

For any action $a$ that has an effect of additive assignment to $x_i$, we have

$$added(i, S) \leftarrow occurs(a, S).$$

For any action $a$ that has an effect of scaling assignment to $x_i$, we have

$$scaled(i, S) \leftarrow occurs(a, S).$$

Finally, the values of variables follow the law of inertia.

$$\begin{aligned} m(I, S+1) = X \leftarrow\ & m(I, S) = X, \\ & not\ assigned(I, S), \\ & not\ added(I, S), \\ & not\ scaled(I, S). \end{aligned}$$

Note that the PDDL2.1 restriction of no heterogeneous assignments in Sect. 3.9 ensures that at any step, we have at most one type of assignment statements.

## 3.7 Discrete Durative Actions

A durative action is a tuple $da = \langle name, c, \pi, \epsilon\rangle$ where $c = \langle c_s, c_e\rangle$, $\pi = \langle \pi_s, \pi_o, \pi_e\rangle$, and $\epsilon = \langle \epsilon_s, \epsilon_c, \epsilon_e\rangle$.

We introduce two new simple actions $da_s = \langle name_s, \pi_1, \epsilon_1\rangle$ and $da_e = \langle name_e, \pi_2, \epsilon_2\rangle$ indicating the start and the end of the durative action. Here, $name_s$ and $name_e$ are new names. The preconditions of $da_s$ and $da_e$ are from the corresponding preconditions and duration constraints of $da$: $\pi_1 = \pi_s \wedge c_s$ and $\pi_2 = \pi_e \wedge c_e$. So are the effects of $da_s$ and $da_e$: $\epsilon_1 = \epsilon_s$ and $\epsilon_2 = \epsilon_e$.

The simple actions can be translated into an $\mathcal{ACC}$ program as in the previous section.

Since here we try to use simple actions $name_s$ and $name_e$ to mimic the durative action $name$, we need the following constraints. There must be $name_e$ after each $name_s$. Similarly, there must be $name_s$ before each $name_e$.

$$\leftarrow occurs(name_s, S), not\ haveAnEnd(name_s, S).$$

$$haveAnEnd(name_s, S) \leftarrow occurs(name_e, S_1), S_1 > S.$$

Where $haveAnEnd(name_s, S)$ means that there is a $name_e$ occurred after $S$. Similarly,

$$\leftarrow occurs(name_e, S), not\ haveABegining(name_e, S).$$

$$haveABegining(name_e, S) \leftarrow occurs(name_s, S_1), S_1 < S.$$

The next constraint is that the occurrences of the same durative action do not overlap. We say that two occurrences of a simple action $name_s$ (and $name_e$ respectively) are *consecutive* ($consecutive(A, S_1, S_2)$, where $A$ is of the action sort, $S_1$ and $S_2$ the step sort $\mathcal{S}_s$) if there is no $name_e$ (and $name_s$ respectively) in between $S_1$ and $S_2$. For any durative action $name$, non-overlapping means that there should not be consecutive $name_s$ or $name_e$.

$$\leftarrow consecutive(name_s, S_1, S_2).$$

$$\leftarrow consecutive(name_e, S_1, S_2).$$

$$\begin{aligned} consecutive(name_s, S_1, S_2) \leftarrow\ & occurs(name_s, S_1), occurs(name_s, S_2), \\ & not\ existEnd(name_s, S_1, S_2). \end{aligned}$$

$$existEnd(name_s, S_1, S_2) \leftarrow occurs(name_e, S), S_1 < S, S < S_2.$$

$$\begin{aligned} consecutive(name_e, S_1, S_2) \leftarrow\ & occurs(name_e, S_1), occurs(name_e, S_2), \\ & not\ existStart(name_e, S_1, S_2). \end{aligned}$$

$$existStart(name_e, S_1, S_2) \leftarrow occurs(name_s, S), S_1 < S, S < S_2.$$

To translate $c_s$ (and similarly $c_e$) which may involve the special variable `?duration`, we introduce a new predicate $duration/3$ such that $duration(name, S_1, S_2)$ where $S_1$, $S_2 \in \mathcal{S}_s$ holds if the action $name$ starts at step $S_1$ and ends at step $S_2$. Its definition is given in Sect. 3.10.

Let $\alpha$ be an arithmetic constraint in $c_s$ of action $name$. Assume the variables of $\alpha$ be $x_1, ..., x_i$, and `?duration`. We define $\alpha^T(S) = p'(S)$ where $p'$ is a new predicate, and $\alpha^P(S)$ is

$$\begin{aligned} p'(S) \leftarrow\ & m(1, S) = X_1, ..., m(i, S) = X_i, duration(name, S_1, S_2), \\ & at(S_1) = T_1, at(S_2) = T_2, D = T_2 - T_1, c_s[\texttt{?duration} \to D]. \end{aligned}$$

$c_s$[?`duration` $\rightarrow D$] denotes the result of substituting $D$ for `?duration` in $c_s$ simultaneously.

Finally, to make sure that the invariant $\pi_o$ of action *name* holds during the duration of an action, we need the following rule.

$$\leftarrow not\ \pi_o^T(T_1), S_1 \leq T_1, T_1 \leq S_2, duration(name, S_1, S_2).$$

Note that when there are continuous effects, we need an alternative approach to the invariants as discussed in the next subsection.

## 3.8 Continuous Effect

For every variable $x_i$ that appears in the left hand side of a continuous assignment statement, we introduce a defined predicate $f(i, S, \Delta T) : X$, where $\Delta T$ is real number. It means that the value of $x_i$ at Step $S$ after the lapse of time $\Delta T$ is $X$. The value of $x_i$ at step $S$ is related to $f$ in the following way.

$$\begin{aligned} m(i, S+1) = X \leftarrow at(S) = T_1, at(S+1) = T_2, \\ \Delta T = T_2 - T_1, f(i, S, \Delta T) = X. \end{aligned}$$

For assignment statements of continuous effects, we introduce a predicate *delta(i, name, S, $\Delta T$)* to represent the impact of durative action *name* on $x_i$ at step $S$ after the lapse of time $\Delta T$. Recall that for continuous effect, we have only additive assignments. Let $x_i \diamond exp$ be an continuous effect of durative action *name*.

$$\begin{aligned} action\_on(name, S) &\leftarrow duration(name, S_1, S_2), \\ &\quad\ S_1 \leq S, S \leq S_2. \\ delta(i, name, S, \Delta T) = C &\leftarrow action\_on(name, S), \\ &\quad\ m(1, S) = X_1, \cdots, m(n, S) = X_n, \\ &\quad\ C = signed(\diamond, exp' * \Delta T). \\ delta(i, name, S, \Delta T) = 0 &\leftarrow not\ action\_on(name, S). \end{aligned}$$

For any action *name* that does not have any continuous effect on $x_i$, we have

$$delta(i, name, S, \Delta T) = 0.$$

The following rule accumulates all the contributions to the $x_i$.

$$\begin{aligned} f(i, S, \Delta T) = X \leftarrow\ & at(S+1) = T_1, at(S) = T_0, \\ & 0 \leq \Delta T, \Delta T \leq T_1 - T_0, \\ & X = m(i, S) + \sum_{Name \in DN} delta(i, Name, S, \Delta T). \end{aligned}$$

where $DN$ denotes the set of all continuous durative actions.

Finally we consider how to monitor the invariant. When the invariant does not contain any variables that may change continuously, the earlier translation works well. If it contains variables changing continuously, it is much more complex to represent the invariant by an $\mathcal{ACC}$ program. Here, for simplicity, we assume the invariant $\alpha$ consists of only an arithmetic constraint.

If $\alpha$ is false at certain moment of the duration, the invariant does not hold. Without loss of generality, let $x_1, ..., x_i$ be the variables of $\alpha$. Let *name* be the action involving $\alpha$. We have the following rules.

$$\begin{aligned} \mathit{violated}(\mathit{name}, S_s, S_e) \leftarrow\ & \mathit{duration}(\mathit{name}, S_s, S_e), \\ & \mathit{at}(S_s) = T_1, \mathit{at}(S_e) = T_2, \\ & S_s \leq S, S \leq S_e, T_1 \leq T, T \leq T_2. \\ & f(1, S, \Delta T) = X_1, \\ & \quad \vdots \\ & f(i, S, \Delta T) = X_i, \\ & \neg\alpha', \\ \leftarrow\ & \mathit{violated}(\mathit{name}, Ss, Se). \end{aligned}$$

where $\alpha'$ is the result of substituting $X_j$ for $x_j$ $(j \in [1..i])$ of $\alpha$.

### 3.9 Encoding PDDL2.1 Restrictions on Actions

PDDL2.1 imposes some restrictions on the occurrence of actions in a plan. We will discuss these restrictions and how to encode them into $\mathcal{ACC}$ rules.

Given an action $a$, $GPre_a$ denotes the set of propositional letters in the preconditions of $a$, $Add_a$ the set of positive literals in the effects of $a$, $Del_a$ the set of negative literals in the effects of $a$, $R_a$ the set of variables on the right hand side of assignment statements of $a$, and $L_a$ and $L'_a$ are defined as follows:

- $L_a = \{x \mid x \text{ appears as an lvalue in } a\}$, and
- $L'_a = \{x \mid x \text{ appears as an lvalue in an additive assignment effect in } a\}$.

Two actions $a$ and $b$ are *non-interfering*, if

$$\begin{gathered} GPre_a \cap (Add_b \cup Del_b) = GPre_b \cap (Add_a \cup Del_a) = \emptyset \\ Add_a \cap Del_b = Add_b \cap Del_a = \emptyset \\ L_a \cap R_b = R_a \cap L_b = \emptyset \\ L_a \cap L_b \subseteq L'_a \cap L'_b \end{gathered}$$

Two actions are *mutex* if they are not non-interfering.

Here are the restrictions on actions of a plan by PDDL2.1.

1. No heterogeneous assignments: at any time, a variable cannot appear as lvalue in more than one simple assignment effect, or in more than one different type of assignment effect.
2. Non-zero-separation: mutex actions much have different end (starting and stopping) points. make use of a value if one of the two is accessing the value to update it.

**No Heterogeneous Assignments.** For any two simple actions $a$ and $b$ such that $a.\epsilon$ and $b.\epsilon$ have different types of assignment on the same variable, we do not allow them to occur at the same moment, i.e.,

$$\leftarrow occurs(a, S), occurs(b, S).$$

For a simple action $a$ and a continuous durative action $b$ such that $a.\epsilon$ and $b.\epsilon_o$ have different types of assignment on the same variable, we do not allow overlapping of the occurrence of $a$ and $b$, i.e.,

$$\leftarrow occurs(a, S), duration(b, S_1, S_2), S_1 \leq S, S \leq S_2.$$

**Non-Zero-Separation.** For any two mutex actions $a$ and $b$, the non-zero-separation rule requires them to have different starting time and different ending time, namely,

$$\begin{aligned} \leftarrow\ & duration(a, S_a, E_a), \\ & duration(b, S_b, E_b), \\ & shareends(S_a, E_a, S_b, E_b). \end{aligned}$$

where

$$\begin{aligned} shareends(S_a, E_a, S_b, E_b) &\leftarrow\ at(S_a) = at(S_b). \\ shareends(S_a, E_a, S_b, E_b) &\leftarrow\ at(S_a) = at(E_b). \\ shareends(S_a, E_a, S_b, E_b) &\leftarrow\ at(E_b) = at(S_b). \\ shareends(S_a, E_a, S_b, E_b) &\leftarrow\ at(E_b) = at(E_b). \end{aligned}$$

### 3.10 Plan Generation

The search for a plan starts from length 1. The *length* of a plan is defined as the number of distinct starting and ending time points of durative actions and the occurring time points of simple actions. Assume we are looking for a plan with length $t$. We use two neighboring steps to represent every distinct starting, ending or occurring point. Hence, the total number of steps is $2t$ and thus $\mathcal{S}_s = \{0..2t-1\}$. Furthermore, we require that a simple action can only occur at even numbered steps. The real time mapping to the even step and its next step is the same, i.e.,

$$\leftarrow even(S), at(S) \neq at(S+1).$$

In the plan generation, for any simple action *name* and any even step $S$, *occurs(name, S)* either happens or not, i.e.,

$$occurs(name, S)\ or\ \neg occurs(name, S) \leftarrow even(S).$$

Note that we use *or* here, which can be translated into $\mathcal{ACC}$ rules in a standard way [15]. For any durative action *name*, it could occur at any even step and could occur as many times as necessary.

$$occurs(name_s, S) \text{ or } \neg occurs(name_s, S) \leftarrow even(S).$$

$$occurs(name_e, S) \text{ or } \neg occurs(name_e, S) \leftarrow even(S).$$

Note that to mimic action *name*, the occurrences of $name_s$ and $name_e$ must satisfy the constraints given in Sect. 3.7.

We now define $duration(name, S, E)$ which means durative action *name* occurs at step $S$ and ends at step $E$.

$$\begin{aligned} duration(name, S, E) \leftarrow\ & occurs(name_s, S), occurs(name_e, E), \\ & notexistSE(name_s, S, E). \end{aligned}$$

$$existSE(name_s, S, E) \leftarrow occurs(name_s, S_1), S < S_1, S_1 < E.$$
$$existSE(name_s, S, E) \leftarrow occurs(name_e, S_1), S < S_1, S_1 < E.$$

Finally, we require the time of steps follow the chronological order, and different steps, except those have to be mapped to the same time, be mapped to different time.

$$T_1 < T_2 \leftarrow even(S_1), even(S_2), at(S_1) = T_1, at(S_2) = T_2, S_1 < S_2.$$

$$T_1 < T_2 \leftarrow odd(S_1), odd(S_2), at(S_1) = T_1, at(S_2) = T_2, S_1 < S_2.$$

## 4 Conclusion

In this paper, we introduce a method to encode temporal planning problems represented in PDDL2.1 into $\mathcal{ACC}$. Thanks to the non monotonicity of $\mathcal{ACC}$ and its power in representing constraints, the encoding is rather natural and straightforward. The generality of $\mathcal{ACC}$ also makes it easy to incorporate new features like PDDL axioms. The planner is complete for the temporally expressive problems. In addition to the work mentioned in the introduction section, we also note that Dovier et al. [16] proposed a new action language which is able to represent constraints conveniently. In the future, besides proving the correctness of the proposed $\mathcal{ACC}$ based method, we will carry out a detailed comparison of the new method with the existing approaches and empirical evaluation of it.

## Acknowledgement

We would like to thank Michael Gelfond, whose earlier collaborative work with us on $\mathcal{ACC}$ and process has laid the foundation of the wrok reported here.

## References

1. Ghallab, M., Laruelle, H.: Representation and control in IxTeT, a temporal planner. In: Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS), pp. 61–67 (1994)
2. Tate, A.: Representing plans as a set of constraints. In: Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS), pp. 221–228 (1996)
3. Ai-Chang, M., Bresina, J., Farrell, K., Hsu, J., Jnsson, A., Kanefsky, B., McCurdy, M., Morris, P., Rajan, K., Vera, A., Yglesias, J., Charest, L., Maldague, P.: MAPGEN: Mixed-intiative activity planning for the mars exploratory rover mission. IEEE Intelligent Systems 19(1), 8–12 (2004)
4. Penberthy, J.S., Weld, D.S.: Temporal planning with continuous change. In: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI), vol. 2, pp. 1010–1015 (1994)
5. Fox, M., Long, D.: PDDL2.1: An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research (JAIR) 20, 61–124 (2003)
6. Chen, Y., Wah, B., Hsu, C.-W.: Temporal planning using subgoal partitioning and resolution in sgplan. Journal of Artificial Intelligence Research (JAIR) 26, 323–369 (2006)
7. Vidal, V., Geffner, H.: Branching and pruning: an optimal temporal POCL planner based on constraint programming. Artificial Intelligence 170(3), 298–335 (2006)
8. Cushing, W., Kambhampati, S., Weld, M., Weld, D.: When is temporal planning really temporal? In: Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI), pp. 1852–1859 (2007)
9. Coles, A., Fox, M., Long, D., Smith, A.: Planning with problems requiring temporal coordination. In: Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI), vol. 2, pp. 892–897 (2008)
10. Huang, R., Chen, Y., Zhang, W.: An optimal temporally expressive planner: Initial results and application to P2P network optimization. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS), pp. 178–185 (2009)
11. Hu, Y.: Temporally-expressive planning as constraint satisfaction problems. In: Proceedings of 17th International Conference on Automated Planning and Scheduling (ICAPS), pp. 192–199 (2007)
12. Thiébaux, S., Hoffmann, J., Nebel, B.: In defense of PDDL axioms. Artificial Intelligence 168(1), 38–69 (2005)
13. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence 53(1-4), 251–287 (2008)
14. Lee, J., Lifschitz, V.: Describing additive fluents in action language C+. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI), pp. 1079–1084 (2003)
15. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press, Cambridge (2003)
16. Dovier, A., Formisano, A., Pontelli, E.: Multivalued action languages with constraints in CLP(FD). In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 255–270. Springer, Heidelberg (2007)

# Applications of Answer Set Programming in Phylogenetic Systematics

Esra Erdem

Faculty of Engineering and Natural Sciences, Sabancı University
Orhanlı, Tuzla, İstanbul 34956, Turkey
esraerdem@sabanciuniv.edu

**Abstract.** We summarize some applications of Answer Set Programming (ASP) in phylogenetics systematics, focusing on the challenges, how they are handled using computational methods of ASP, the usefulness of the ASP-based methods/tools both from the point of view of phylogenetic systematics and from the point of view of ASP.

## 1 Introduction

Since the concept of a stable model was defined by Michael Gelfond and Vladimir Lifschitz [25], the declarative programming paradigm of Answer Set Programming (ASP) has emerged [32,37,30,31] and flourished with many efficient solvers (e.g., CLASP [22], CMODELS [29], DLV [13,28], SMODELS [35,37]) and applications. One active area for ASP has been computational biology and bioinformatics. Various computational problems have been studied in systems biology [38,39,40,23,24,36,10,26], haplotype inference [17,14,41], query answering and inconsistency checking over biomedical ontologies [1,20,21,19], and phylogenetic systematics [11,5,16,18,15,3,4,7,6] using computational methods of ASP. These applications have led to useful methods/tools not only for experts in these areas but also for the ASP community.

In this chapter, based on our earlier studies, we summarize some applications of ASP in phylogenetic systematics—the study of evolutionary relations between species based on their shared traits [27]. Represented diagrammatically, these relations can form a tree whose leaves represent the species, internal vertices represent their ancestors, and edges represent the genetic relationships between them. Such a tree is called a "phylogeny". We consider reconstruction of phylogenies as the first step of reconstructing the evolutionary history of a set of taxa (taxonomic units). The idea is then to reconstruct (phylogenetic) networks, which also explain the contacts (or borrowings) between taxonomic units, from the reconstructed phylogenies by adding some horizontal bidirectional edges. We have studied both steps of inferring evolutionary relations between species, using computational methods of ASP. We have illustrated the applicability and effectiveness of these ASP-based methods for the historical analysis of languages (e.g., Indo-European languages and Chinese dialects), the historical analysis of parasite-host systems (e.g., *Alcatenia* species), and the historical analysis of oak trees (e.g., *Quercus* species). While working on these real datasets in close collaboration with the experts, we have faced some challenges and introduced novel solutions to each one of them. The

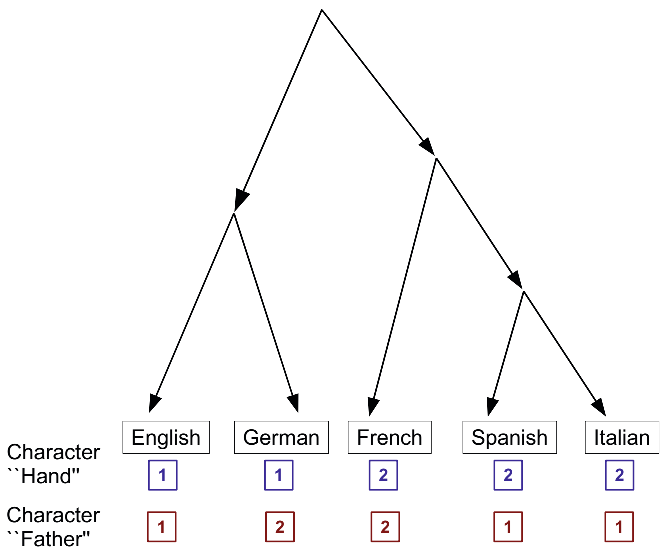


**Fig. 1.** A phylogeny reconstructed for English, German, French, Spanish, Italian with respect to a leaf-labeling function over two characters, "Hand" and "Father"

following sections summarize these challenges, and our solutions and their usefulness both from the point of view of phylogenetic systematics and from the point of view of ASP. We conclude with a discussion of further challenges in phylogenetic systematics that pose challenges for ASP.

## 2 Reconstructing Phylogenies

A *phylogeny* $(V, E, L, I, S, f)$ for a set of taxa is a finite rooted binary tree $\langle V, E \rangle$ along with two finite sets $I$ and $S$ and a function $f$ from $L \times I$ to $S$, where $L \subseteq V$ is the set of leaves of the tree. The set $L$ represents the given taxonomic units whereas the set $V \setminus L$ describes their ancestral units and the set $E$ describes the genetic relationships between them. The elements of $I$ are usually positive integers ("indices") that represent, intuitively, qualitative characters (i.e., shared traits), and elements of $S$ are possible states of these characters. The function $f$ "labels" every leaf $v$ by mapping every index $i$ to the state $f(v, i)$ of the corresponding character in that taxonomic unit. Figure 1 shows a phylogeny for five languages, $L = \{$English, German, French, Spanish, Italian$\}$, with two characters, $I = \{$"Hand", "Father"$\}$, that have two states, $S = \{1, 2\}$.

Our studies in phylogenetic systematics are based on the compatibility criterion [9], where the goal is to reconstruct a phylogeny with a large number of characters that are "compatible" with it. Intuitively, a character is compatible with a phylogeny/network if the taxonomic units that instantiate this character in the same way (i.e., that have the same character state) are connected via a tree in that phylogeny.

A character $i \in I$ is *compatible* with a phylogeny $(V, E, L, I, S, f)$ if there exists a function $g : V \times \{i\} \to S$ such that

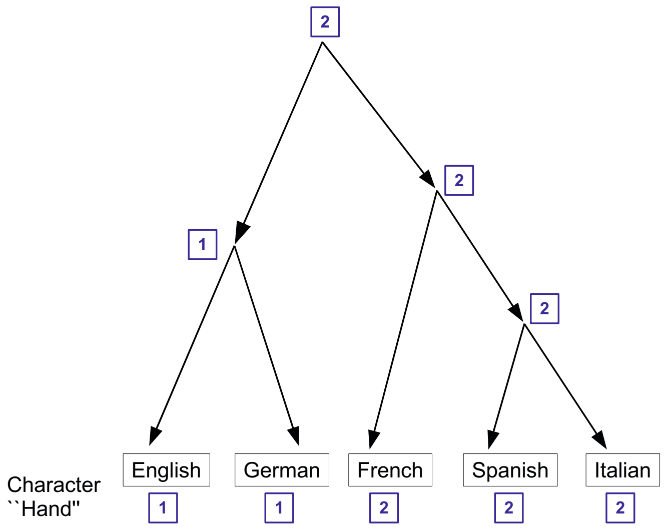


**Fig. 2.** An example for a compatible character ("Hand"): all vertices labeled by 1 (resp. 2) are connected via a rooted tree

- for every leaf $v$ of the phylogeny, $g(v, i) = f(v, i)$; and
- for every $s \in S$, if the set $V_{is} = \{x \in V : g(x, i) = s\}$ is nonempty then the digraph $\langle V, E\rangle$ has a subgraph with the set $V_{is}$ of vertices that is a rooted tree.

A character is *incompatible* with a phylogeny if it is not compatible with that phylogeny.

For example, in Figure 2, Character "Hand" is compatible with respect to the given phylogeny, since all vertices labeled with the same state are connected with a tree. In Figure 3, Character "Father" is incompatible, since there is no possible labeling of internal vertices such that all vertices labeled with the same state are connected with a tree.

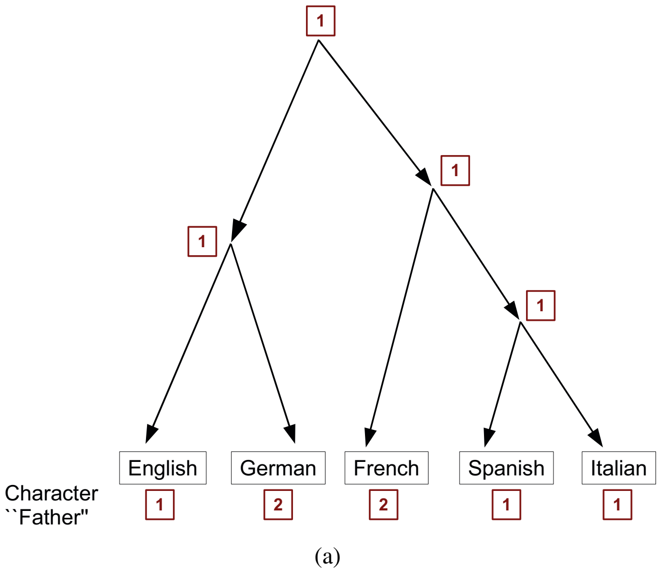
The computational problem we are interested in is the following problem:

> $n$-INCOMPATIBILITY PROBLEM Given three sets $L$, $I$, $S$ and a function $f$, and a non-negative integer $n$, decide the existence of a phylogeny $(V, E, L, I, S, f)$ with at most $n$ incompatible characters.

This problem is NP-hard [9,2,34,5].

In [4,3], we described this decision problem as an ASP program whose answer sets correspond to such phylogenies, and proved the correctness of this formalization. We observed that domain-specific geographical/temporal constraints could be added to the main program as constraints, to compute more plausible phylogenies.

Usually, instead of finding one phylogeny with the minimum number of incompatible characters, experts are interested in finding all phylogenies with a small number $n$ of characters. Finding these phylogenies, by first computing all the answer sets and then extracting distinct phylogenies from them may not be efficient since there are many



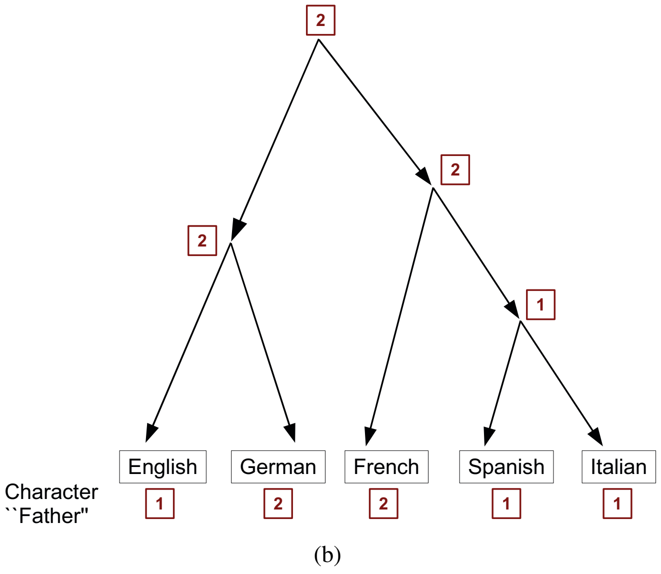


**Fig. 3.** An example for an incompatible character ("Father"): if you label the internal vertices by 1 as in (a) (resp. 2 as in (b)), the vertices labeled by 1 (resp. 2) are connected but the vertices labeled by 2 (resp. 1) are not connected

answer sets that correspond to the same phylogeny due to different possible labelings of internal vertices (e.g., the root of the phylogeny in Figure 2 can be labeled as 1 or 2, leading to two different labelings of vertices of the same tree). On the other hand, we can compute all phylogenies with $i = 0, ..., n$ incompatible characters as in [3,4] by iteratively calling an ASP solver to find a phylogeny different from the previously computed ones until no answer set is found. After that, the experts can identify the phylogenies that are plausible.

Although the ASP formulation of phylogeny reconstruction as in [4,3] is applicable to many real datasets, we observed in our studies that it is not applicable to real datasets with "ambiguous" labelings, i.e., where a leaf is labeled by a set of states rather than a unique state. Such ambiguities are mostly due to polymorphic characters. If a real dataset (such as Indo-European languages) has only a few such ambiguities, then every possible combination of these labelings are considered as a new character. However, if a dataset has many ambiguities (like Turkic languages), enumerating all possibilities will lead to too many new characters. In such cases, to deal with this challenge, we modified our ASP program by adding an explicit definition of a leaf-labeling function that picks exactly one state for each group, from the set of states that label the group, as described in [5].

## 3 Computing Similar/Diverse Phylogenies

For a given dataset, we usually compute all phylogenies with a large number of compatible characters, that also satisfy the given domain-specific constraints. However, sometimes there are too many such phylogenies, and then the experts have to analyze/compare them manually to pick the most plausible ones.

In our experiments with Indo-European languages [3,4], we observed that one way the experts analyze phylogenies is as follows: try to classify the phylogenies by finding few diverse phylogenies, and pick the most plausible one among them; afterwards, try to find phylogenies close to this plausible phylogeny. So if we can compute similar/diverse phylogenies, and phylogenies close/distant to a given set of phylogenies, then there is no need to compute all phylogenies in advance. With this motivation, we studied the following two main problems:

$n$ $k$-SIMILAR PHYLOGENIES (resp. $n$ $k$-DIVERSE PHYLOGENIES)
Given an ASP program that formulates reconstruction of phylogenies with at most $l$ incompatible characters, a distance measure $\Delta$ that maps a set of phylogenies to a nonnegative integer, and two nonnegative integers $n$ and $k$, decide whether a set $S$ of $n$ phylogenies with at most $l$ incompatible characters exists such that $\Delta(S) \leq k$ (resp. $\Delta(S) \geq k$).

$k$-CLOSE PHYLOGENY (resp. $k$-DISTANT PHYLOGENY)
Given an ASP program that formulates reconstruction of phylogenies with at most $l$ incompatible characters, a distance measure $\Delta$ that maps a set of phylogenies to a nonnegative integer, a set $S$ of phylogenies, and a nonnegative integer $k$, decide whether a phylogeny $s$ ($s \notin S$) with at most $l$ incompatible characters exists such that $\Delta(S \cup \{s\}) \leq k$ (resp. $\Delta(S \cup \{s\}) \geq k$).

We studied in [11,6] various related decision/optimization problems, analyzed their complexities, and computational methods for solving them offline/online. We also introduced novel distance functions for comparing phylogenies.

In particular, we investigated three sorts of online methods to compute $n$ $k$-similar or diverse phylogenies. In the first method, the idea is to reformulate the ASP program for phylogeny reconstruction to compute $n$-distinct phylogenies, to formulate the distance function as an ASP program, and then to extract $n$ $k$-similar (resp. $k$-diverse) phylogenies from an answer set for the union of these ASP programs.

The second method does not modify the ASP program for phylogeny reconstruction but formulates the distance function as an ASP program, so that a unique $k$-close (resp. $k$-distant) phylogeny can be extracted from an answer set for the union of these ASP programs and a previously computed phylogeny; then, by iteratively computing $k$-close (resp. $k$-distant) phylogenies one after other, it computes online a set of $n$ $k$-similar (or $k$-diverse) solutions.

The third method is different from the first two methods in that it does not modify the ASP program encoding phylogeny reconstruction, and it does not formulate the distance function as an ASP program. Instead it modifies the search algorithm of an ASP solver to compute all $n$ $k$-similar (or $k$-diverse) phylogenies at once with respect to the distance function implemented as a C++ program. For this method, we modified the search algorithm of CLASP [22], in the spirit of a branch-and-bound algorithm to compute similar/diverse solutions. CLASPperforms a DPLL-like [8,33] branch and bound search to find an answer set for a given ASP program: at each level, it "propagates" some literals to be included in the answer set, "selects" new literals to branch on, or "backtracks" to an earlier appropriate point in search while "learning conflicts" to avoid redundant search. We modified CLASP to obtain CLASP-NK as in Algorithm 1; the modifications are shown in color red. CLASP-NK can compute $n$ $k$-similar/diverse solutions. In our experiments for reconstructing similar/diverse solutions for Indo-European languages, we observed that this method is computationally more efficient than the other two methods in terms of time/space and that it allows for computing similar/diverse solutions in ASP when the distance function cannot be formulated in ASP (e.g., due to some numerical functions).

## 4 Computing Weighted Phylogenies

Another way to find more plausible phylogenies is to assign a weight to each phylogeny to characterize its plausibility, and to compute phylogenies whose weights are over some given threshold. For that we studied in [6,7] the following problem:

> AT LEAST (resp. AT MOST) $w$-WEIGHTED PHYLOGENY: Given an ASP program that formulates reconstruction of phylogenies with at most $l$ incompatible characters, a weight measure $\omega$ that maps a phylogeny to a nonnegative integer, and a nonnegative integer $w$, decide whether a phylogeny $S$ with at most $l$ incompatible characters exists such that $\omega(S) \geq w$ (resp. $\omega(S) \leq w$).

Motivated by our studies on computing similar/diverse solutions in ASP [11], we studied representation/search-based online methods for computing weighted solutions in ASP. We introduced novel weight functions that take into account domain-specific information such as expected groupings of taxonomic units. We also modified the search algorithm of CLASP to compute weighted solutions; this modified version is called CLASP-W.

**Algorithm 1.** The algorithm of CLASP-NK

```
Input: An ASP program $\Pi$, nonnegative integers $n$, and $k$
Output: A set $X$ of $n$ solutions that are $k$ similar ($n$ $k$-similar solutions)
  $A \leftarrow \emptyset$   // current assignment of literals
  $\bigtriangledown \leftarrow \emptyset$   // set of conflicts
  $X \leftarrow \emptyset$   // computed solutions
  while $|X| < n$ do
    $PartialSolution \leftarrow A$
      // compute a lower bound for the distance between any completion of a partial solution
    and previously computed solutions
    $LowerBound \leftarrow$ DISTANCE-ANALYZE$(X, PartialSolution)$
    PROPAGATION$(\Pi, A, \bigtriangledown)$
    if Conflict in propagation OR $LowerBound > k$ then
      RESOLVE-CONFLICT$(\Pi, A, \bigtriangledown)$   // learn and update the conflict set and do backtracking
    else
      if Current assignment does not yield an answer set then
        SELECT$(\Pi, A, \bigtriangledown)$   // select a literal to continue search
      else
        $X \leftarrow X \cup A$
        $A \leftarrow \emptyset$
      end if
    end if
  end while
  return X
```

## 5 Reconstructing Phylogenies for Very Large Datasets

For many real datasets, using the ASP program of [4,3] using an existing ASP solver is quite inefficient and even not possible. For a more efficient computation, we investigated methods to reduce the datasets and to improve the search.

In phylogeny reconstruction, one way to handle a large dataset is to reduce it to a smaller dataset by some preprocessing (e.g., by removing the noninformative parts of the dataset as in [3]). With such a preprocessing, for instance, out of 282 characters of Indo-European languages, 21 are found to be informative. However, such preprocessing may not be applicable to all datasets, like *Quercus* (oak trees), since every bit of the given information is essential in reconstructing a phylogeny.

In such cases, a phylogeny can be computed by a divide-and-conquer approach, if the experts provide some domain-specific information about how the species could be grouped; usually the experts provide such information. For instance, [3] applies this idea on Indo-European languages: first 8 language groups are identified over 24 languages by the historical linguist Don Ringe, then a plausible phylogeny is reconstructed for each group by "propagating up" the labelings of taxonomic units towards the root, and finally a main phylogeny is reconstructed over the roots of these phylogenies. However, this bottom-up approach may not be applicable to all datasets, like *Quercus*, where a group of taxonomic units may lead to many different phylogenies and cannot be characterized by a unique node. Note that if we pick one of these phylogenies and accordingly decide

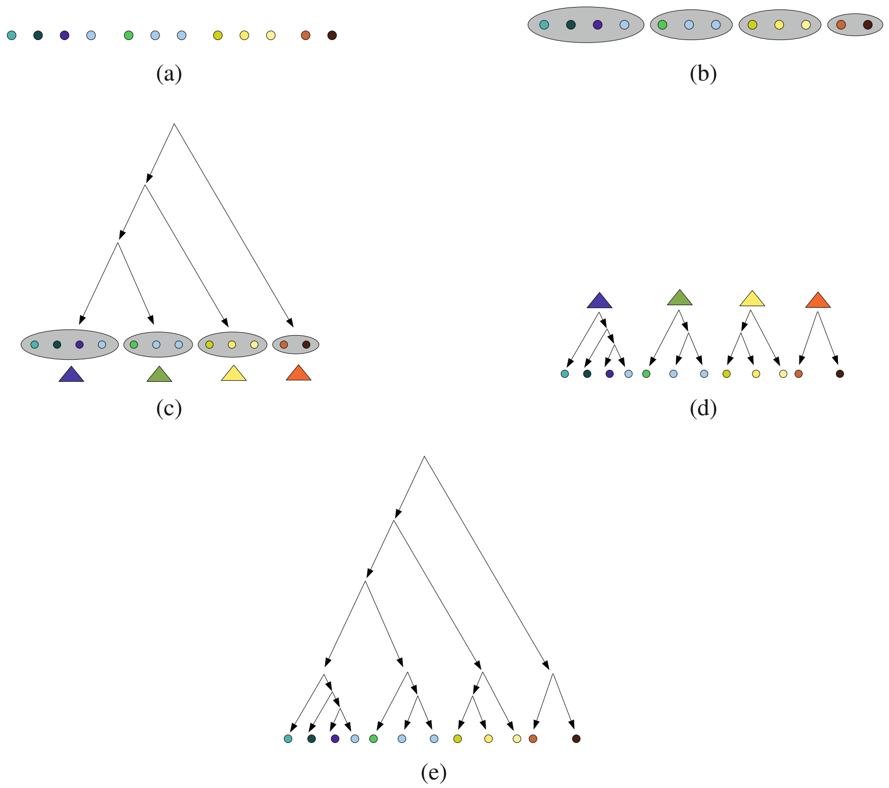


**Fig. 4.** (a) Given taxonomic units (b) and their groupings. (c) A main phylogeny for all groups, where the triangles denote a labeling for the groups. (d) Phylogenies reconstructed for each group. (e) A complete phylogeny obtained by combining the main phylogeny for all groups with the phylogenies for each group.

for a labeling for the groups, we may end up reconstructing a main phylogeny with more number of incompatible characters. However, since the main phylogeny characterizes deep evolution, experts expect less number of incompatible characters for such phylogenies.

To be able to automatically reconstruct phylogenies for large datasets, including *Quercus*, [5] introduces a different divide-and-conquer method for reconstructing large phylogenies with a top-down approach (Figure 4): first a main phylogeny is computed viewing each group as a unique node, and then phylogenies are computed for each group. This approach is applicable to datasets where the groupings are not as clean as in Indo-European. However, it still needs to address some challenges.

To be able to reconstruct a main phylogeny for the groups of taxonomic units, we need a labeling of these groups. How can we pick such a labeling considering the labelings of the taxonomic units? Recall that the ASP program of [4,3] for reconstruction of phylogenies assumes that a leaf-labeling function $f$ is given as an input. As described at the end of Section 2, this program can be modified to handle ambiguous data (as in

[5]) by adding an explicit definition of a leaf-labeling function that picks exactly one state for each group, from the set of states that label the group. By this way, labels of groups can be determined automatically while reconstructing the phylogeny.

The divide-and-conquer approach of [5] reconstructs two sorts of phylogenies: one phylogeny for each group of taxonomic units, and one main phylogeny for all groups. In each case, we may end up computing too many possible phylogenies for a set of taxa, with a large number of compatible characters. Instead, in each case, we can compute the most plausible phylogenies using domain-specific information provided by the experts. For instance, for the first kind of phylogenies, according to the compatibility criterion, we can group taxonomic units with more number of identical character states closer to each other. Once we define a weight measure (as in [6,7] introduced for Indo-European languages) to characterize this idea of plausibility, we can find a max-weighted phylogeny using methods of Section 4. For the reconstruction of the main phylogeny, we can utilize further domain-specific information about possible classification of groups so that the phylogenies that classify the groups with respect to the given domain-specific information have more weight. For that, we can define a weight measure that takes into account hierarchical groupings of taxonomic units (as in [7] introduced for *Quercus* species); and utilize our methods described in Section 4 for computing max-weighted phylogenies.

## 6 PHYLO-ASP

We implemented some of the methods/algorithms for reconstructing/analyzing/ comparing phylogenies summarized above, as part of a phylogenetic system, called PHYLO-ASP. This system has mainly four components.

PHYLORECONSTRUCT-ASP is used for reconstructing (weighted) phylogenies. Its overall structure is illustrated in Figure 5. To use this system, the expert provides a leaf-labeling function for a set of taxonomic units. Optionally, she can choose one of the four available weight measures, two domain-independent and two domain-dependent [5], or provide a new weight measure; and provide some domain-specific information about groupings of taxonomic units. Before reconstruction of phylogenies, the system performs some preprocessing as described in [3] and [5]. If some grouping information is provided then the system follows the divide-and-conquer algorithm of Section 5 by utilizing the methods for computing weighted phylogenies of Section 4. Otherwise, it computes phylogenies as described in Sections 2 and 4. According to the given option, the system can solve various optimization/decision problems.

PHYLORECONSTRUCTN-ASP is used for computing similar/diverse phylogenies, utilizing the methods described in Section 3. The expert needs to provide a leaf-labeling function for a set of taxonomic units. The system implements two distance measures, one domain-independent and one domain-dependent [11]; so the expert can choose one of them. Depending on the appropriate option, the system can solve various optimization/decision problems.

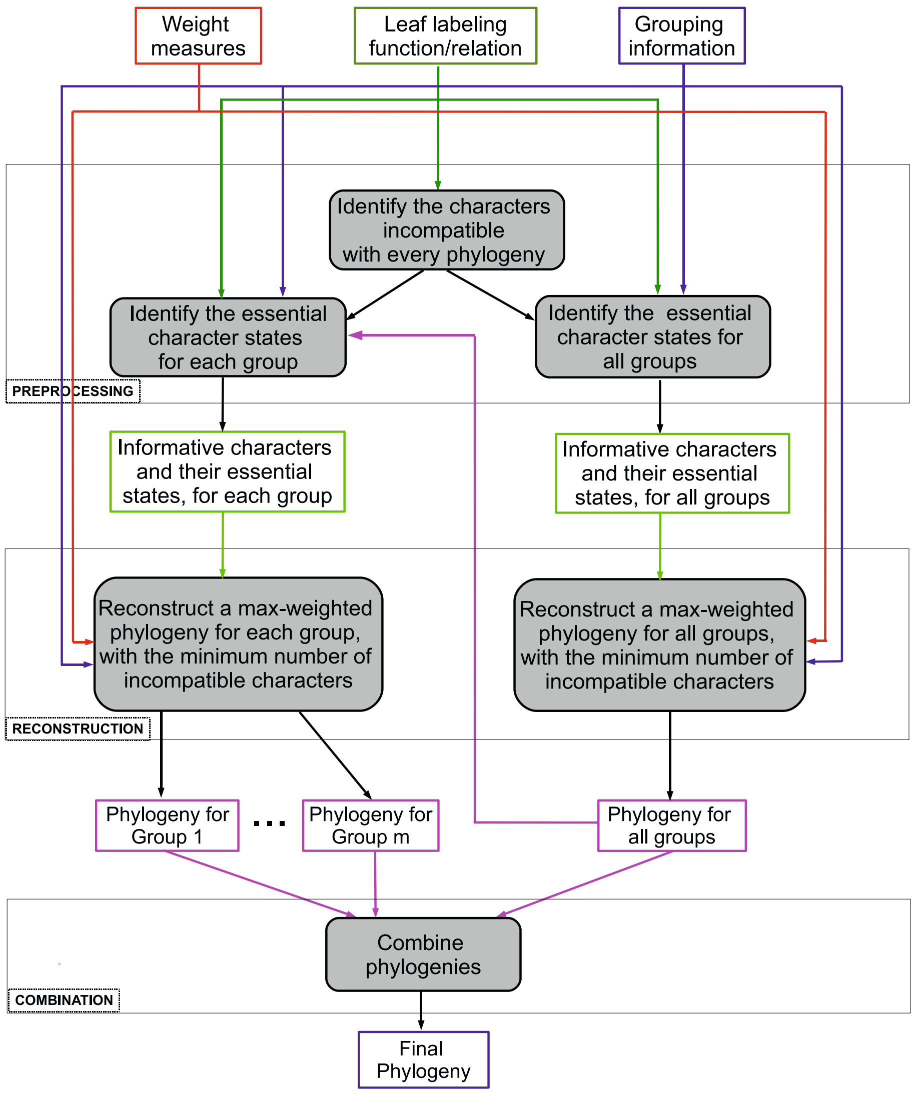

Weight measures
Leaf labeling function/relation
Grouping information
Identify the characters incompatible with every phylogeny
Identify the essential character states for each group
Identify the essential character states for all groups
PREPROCESSING
Informative characters and their essential states, for each group
Informative characters and their essential states, for all groups
Reconstruct a max-weighted phylogeny for each group, with the minimum number of incompatible characters
Reconstruct a max-weighted phylogeny for all groups, with the minimum number of incompatible characters
RECONSTRUCTION
Phylogeny for Group 1
...
Phylogeny for Group m
Phylogeny for all groups
Combine phylogenies
COMBINATION
Final Phylogeny

**Fig. 5.** The Overall System Architecture of PHYLORECONSTRUCT-ASP

PHYLOANALYZE-ASP is used for analyzing a given phylogeny or a given dataset, as described in [5]. For instance, we can find informative characters in a given dataset, or compatible characters with respect to a given phylogeny using this system.

PHYLOCOMPARE-ASP is used for comparing phylogenies offline, as described in [11]. For instance, we can find similar/diverse phylogenies among some given phylogenies. The system implements two distance measures, one domain-independent and one domain-dependent [11]; so the expert can choose one of them.

We applied PHYLO-ASP to reconstruct and analyze Chinese dialects, Indo-European languages, Turkic languages, *Alcataenia* species (a tapeworm genus), and *Quercus* species (oak trees). After reconstructing phylogenies for each taxa using PHYLO-ASP, we identified the phylogenies that are plausible. For instance, for the Chinese dialects and Indo-European languages, the plausibility of phylogenies is identified with respect to the available linguistics and archaeological evidence. For *Alcataenia*, the plausibility of the phylogeny we compute is dependent on the knowledge of host phylogeny (e.g., phylogeny of the seabird family *Alcidae*), chronology of the fossil record, and biogeographical evidence. On the other hand, we computed similar/diverse phylogenies for these datasets using PHYLO-ASP. All these experimental results are detailed in [5,6,7,16,11,3,4].

## 7 Reconstructing Temporal Networks

A phylogeny for a given set of taxonomic units characterizes the genetic evolutionary relations between them. If there are few characters left that are not compatible with the phylogeny, then the relations between the taxonomic units can be explained differently for these characters. For instance, contacts/borrowings may explain how these characters have such states. To characterize such relations, we transform the phylogeny into a network by adding a small number of horizontal edges that represent contact/borrowings between the taxonomic units.

For some datasets, like languages, experts can provide us explicit temporal information about extinct languages—by estimates of the dates when those languages could be spoken. Such temporal information can be integrated into the concept of a phylogeny and a network with a "date" assigned to each vertex, leading to the concepts of "temporal phylogenies" and "temporal networks" (Figure 6). Dates monotonically increase along every branch of the phylogeny, and the dates assigned to the ends of a lateral edge are equal to each other. With such an extended definition of a phylogeny/network, we can talk not only about the languages that are represented by the vertices, but also about the "intermediate" languages spoken by members of a linguistic community at various times.

A "temporal phylogeny" is a phylogeny along with a function $\tau$ from vertices of the phylogeny to real numbers denoting the times when these taxonomic units emerged (Figure 6(a)). A contact between two taxonomic units can be represented by a horizontal edge added to a pictorial representation of temporal phylogeny (Figure 6(b)). The two endpoints of the edge are simultaneous "events" in the histories of these communities. An event can be represented by a pair $v{\uparrow}t$, where $v$ is a vertex of the phylogeny and $t$ is a real number.

A finite set $C$ of contacts defines a *temporal (phylogenetic) network* $\langle V \cup V_C, E_C \rangle$—a digraph obtained from $T = \langle V, E \rangle$ by inserting the elements $v{\uparrow}t$ of the contacts from $C$ as intermediate vertices and then adding every contact in $C$ as a bidirectional edge. We say that a set $C$ of contacts is *simple* if the endpoints of all lateral edges are different from the vertices of $T$, and each lateral edge subdivides an edge of $T$ into exactly two edges.

About a simple set $C$ of contacts (and about the corresponding temporal network $\langle V \cup V_C, E_C \rangle$) we say that it is *perfect* if there exists a function $g : (V \cup V_C) \times I \to S$

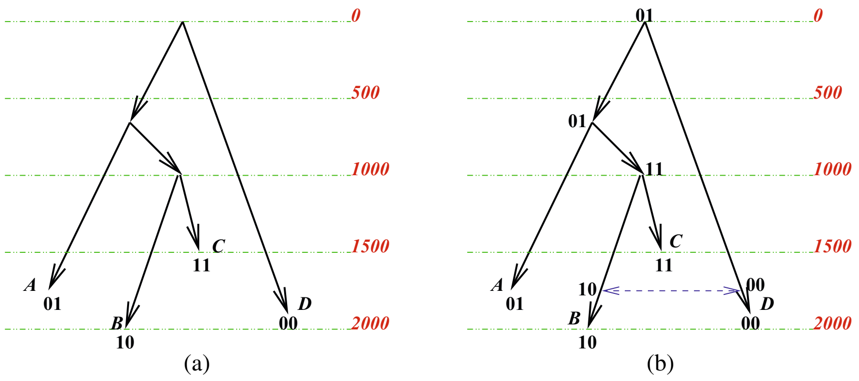


**Fig. 6.** A temporal phylogeny (a), and a perfect temporal network (b) with a lateral edge connecting $B{\uparrow}1750$ with $D{\uparrow}1750$

such that the function $g$ extends $f$ from leaves to all internal nodes of the temporal network, and that every state $s$ of every character $i$ could evolve from its original occurrence in some "root" (i.e., every character $i$ is compatible with the temporal network).

We are interested in the problem of turning a temporal phylogeny into a perfect temporal network by adding a small number of simple contacts. For instance, given the phylogeny in Figure 6(a), the single contact $\{B{\uparrow}1750, D{\uparrow}1750\}$ is a possible answer.

It is clear that the information included in a temporal phylogeny is not sufficient for determining the exact dates of the contacts that turn it into a perfect temporal network. To make this idea precise, let us select for each $v \in V \setminus \{R\}$ a new symbol $v{\uparrow}$, and define the *summary* of a simple set $C$ of contacts to be the result of replacing each element $v{\uparrow}t$ of every contact in $C$ with $v{\uparrow}$. Thus summaries consist of 2-element subsets of the set $V{\uparrow} = \{v{\uparrow} : v \in V \setminus \{R\}\}$. For instance, the summary of the set of contacts of Figure 6(b) is $\{\{B{\uparrow}, D{\uparrow}\}\}$. Then the computational problem we are interested in can be described as follows:

> $n$-INCREMENT TO PERFECT SIMPLE TEMPORAL NETWORK ($n$-IPSTN) Given a phylogeny $\langle V, E, I, S, f\rangle$, a function $v \mapsto (\tau_{min}(v), \tau_{max}(v))$ from the vertices of the phylogeny to open intervals, and a nonnegative integer $n$, decide the existence of a set of 2-element subsets of $V{\uparrow}$ that is the summary of a perfect simple set of $n$ contacts for a temporal phylogeny $\langle V, E, I, S, f, \tau\rangle$ such that, for all $v \in V$, $\tau_{min}(v) < \tau(v) < \tau_{max}(v)$.

This problem is NP-hard [5]. [16] describes the $n$-IPSTN problem as an ASP program whose answer sets correspond to such networks characterized by summaries of contacts.

While studying this problem we faced several difficulties. One challenge was to modify the mathematical model of a temporal network in such a way that we do not have to deal with (real) numbers; so we introduced the concept of summaries and related them to networks.

To ensure that the edges are horizontal, we need to solve a sort of time interval analysis problem; since it involves reasoning over (real) numbers, we investigated other methods that are better suited than ASP. Essentially, we solved $n$-IPSTN problems in

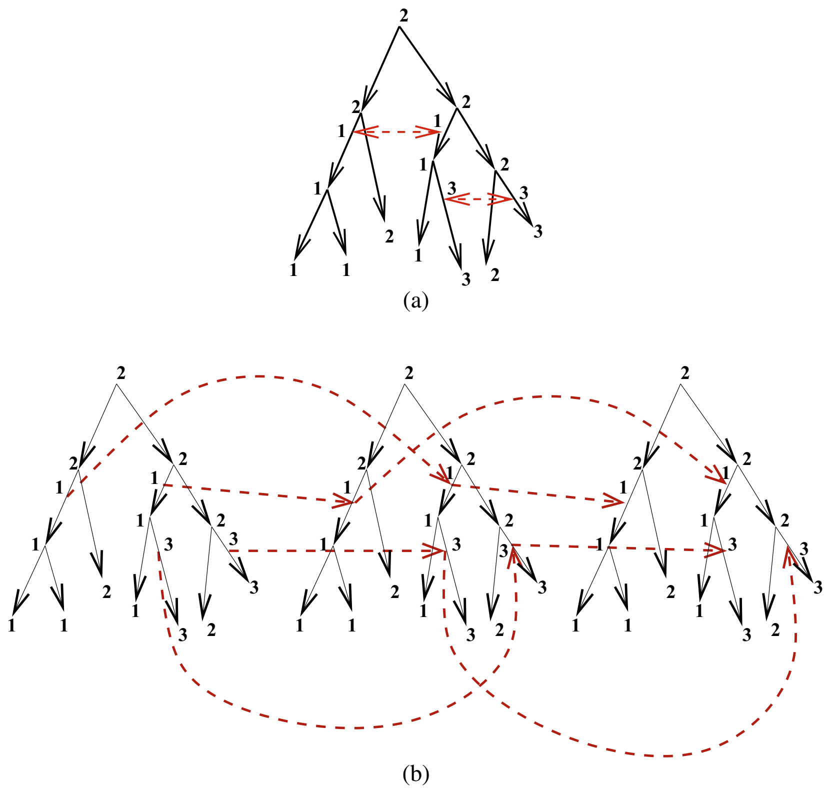


**Fig. 7.** A network (a) and its corresponding digraph (b)

two steps: use an ASP system (CMODELS) to compute summaries so that every character is compatible with the network, and then use a Constraint Logic Programming system (ECL$^i$PS$^e$) to check, for each summary, whether the corresponding contact occurs within the given time intervals.

Reconstructing all networks for real datasets, such as Indo-European languages, with this approach is still quite inefficient since the datasets are very large. To handle this challenge, we investigated in [15] the idea of using our ASP program on smaller problems, embedded in a divide-and-conquer algorithm: first reconstruct a set of minimal networks for each character, and then find a minimal hitting set using also ASP. We could compute solutions with this divide-and-conquer approach, that we could not compute earlier.

We also investigated the effect of reformulating the problem in ASP [16]. We observed that our ASP program is not tight due to bidirectional lateral edges in the network: Recall that we need to check the compatibility of every character with the network, and for that we need to check whether every vertex labeled by the same state are connected by a tree with root $r$; thus, for connectedness, we need to check the reachability of a vertex from another; therefore, with the usual recursive definition of reachability, our ASP program is not tight. We also observed that our ASP program that describes a network with $n$ bidirectional edges can be turned into a tight program: consider instead $n + 1$ copies of the network connected by $n$ additional edges (as in Figure 7), define the reachability of a vertex in one network from a vertex in another network, and check the compatibility of a character with respect to this directed acyclic graph. We observed that the tight version of the program led to more efficient computations.

We implemented a phylogenetic system, called PHYLONET-ASP [5], for reconstructing (temporal) phylogenetic networks based on the tight formulation of the $n$-IPSTN problem, and by considering a simpler version of the interval analysis problem so that an ASP solver can be used to solve it. In our experiments with Indo-European languages, Chinese dialects and *Quercus* species, after identifying a plausible phylogeny for each taxa, we transformed the phylogeny into phylogenetic networks using PHYLONET-ASP [5,15,18].

## 8 Discussion

All of the ASP-based methods/tools briefly described above present novelties for phylogenetic systematics: new mathematical models for reconstructing phylogenies/networks based on the compatibility criterion (e.g., concept of a temporal phylogeny/network), new software for reconstructing phylogenies/networks (PHYLORECONSTRUCT-ASP and PHYLONET-ASP), new domain-specific/independent distance/weight measures for phylogenies, new software for comparing and analyzing phylogenies (PHYLOCOMPARE-ASP and PHYLOANALYZE-ASP) and computing similar/diverse/ weighted phylogenies (PHYLORECONSTRUCTN-ASP). Due to the compatibility criterion, our ASP-based methods are applicable to datasets (mostly based on morphological characters) with no (or very rare) backmutation; they are not applicable to genomic datasets. Therefore, the usefulness of these approaches have been shown on various morphological datasets, such as Indo-European languages, Chinese dialects, Turkic languages, *Alcatenia* species, and *Quercus* species: various plausible phylogenies/networks are computed that conformed with the available evidences; some of these results were different from the ones computed earlier.

All these applications have been useful in understanding the strengths of ASP (e.g., easy representation of recursive definitions, such as reachability of a vertex from another vertex in a directed cyclic graph, easy representation of cardinality constraints, elaboration-tolerant representation of domain-specific information in terms of constraints) and the weaknesses of ASP. In particular, understanding the latter has led to novel methods/methodologies for solving computational problems as part of real world applications.

For instance, for many large datasets, it is not possible to reconstruct phylogenies/ networks using a straightforward representation of the problem with an existing ASP solver. To handle this challenge, we have investigated methods to reduce the datasets and to make the search more efficient. First, we have developed some preprocessing methods to find the informative part of the datasets. When we noticed that preprocessing is not sufficient to reconstruct phylogenies for some datasets, we have developed a divide-and-conquer algorithm that computes a phylogeny in pieces and combines them in the end. Similarly, we have developed a divide-and-conquer algorithm to reconstruct networks from phylogenies, making use of hitting sets.

When we noticed that some numerical computations (e.g., some weight/distance measures) are not supported by the ASP solvers, we have modified the ASP solver CLASP to compute a weighted solution where the weight function is implemented as a C++ program. The modified version of CLASP, called CLASP-W, is general enough to compute weighted solutions for other computational problems such as planning. Another novelty of CLASP-W is that it builds a weighted solution incrementally in the spirit of a branch-and-bound algorithm; by this way, redundant search is avoided. Similarly, we have modified CLASP to compute similar/diverse solutions with respect to a given distance function; the modified version is called CLASP-NK.

To handle many of the challenges, we have used computational methods of ASP in connection with other computational methods. For instance, for reconstructing temporal networks, we have used an ASP solver (CMODELS) for reconstructing a network and a Constraint Programming system (ECLIPSE) for time interval analysis. Also, in some applications, we have embedded ASP solvers as part of larger implementations (cf. the overall structure of PHYLORECONSTRUCT-ASP in Figure 5). One interesting idea in connection with these ideas of using ASP with other paradigms and embedded in more complex algorithms is to investigate the other way around: how other computational methods can be embedded in ASP. The latest studies on programs with external predicates [12] and their implementations (e.g., DLVHEX) may be helpful for extending the ASP solvers, like CLASP, whose input language is the same as LPARSE's.

On the other hand, we have investigated how the straightforward ASP formulations of the problems can be modified to obtain better computational efficiency in terms of time and space. For instance, for network reconstruction, we have turned the non-tight ASP program that includes the usual recursive definition of reachability to a tight ASP program. In our studies for computing weighted phylogenies, we have observed that using aggregates in the definitions of weight functions, compared to their explicit definitions, leads to better computational performance. One useful direction to investigate would be the effect of reformulation in ASP for various computational problems.

## Acknowledgments

Our work on the use of ASP in phylogenetic systematics has involved, at various stages, close collaborations with Yasin Bakis, Balkiz Ozturk Basaran, Dan Brooks, Duygu Cakmak, Thomas Eiter, Halit Erdogan, Selim Erdogan, Michael Fink, Vladimir Lifschitz, Luay Nakhleh, James Minett, Mehmet Olmez, Don Ringe, Ugur Sezerman, Feng Wang. Agostino Dovier has provided useful suggestions on an earlier version of this paper.

## References

1. Bodenreider, O., Çoban, Z.H., Doğanay, M.C., Erdem, E., Koşucu, H.: A preliminary report on answering complex queries related to drug discovery using answer set programming. In: Proc. of ALPSWS (2008)
2. Bodlaender, H.L., Fellows, M.R., Warnow, T.J.: Two strikes against perfect phylogeny. In: Proc. of 19th International Colloquidum on Automata Languages and Programming, pp. 273–283. Springer, Heidelberg (1992)

3. Brooks, D.R., Erdem, E., Erdoğan, S.T., Minett, J.W., Ringe, D.: Inferring phylogenetic trees using answer set programming. Journal of Automated Reasoning 39(4), 471–511 (2007)
4. Brooks, D.R., Erdem, E., Minett, J.W., Ringe, D.: Character-based cladistics and answer set programming. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 37–51. Springer, Heidelberg (2005)
5. Cakmak, D.: Reconstructing weighted phylogenetic trees and weighted phylogenetic networks using answer set programming, M.S. Thesis, Sabancı University (2010)
6. Cakmak, D., Erdem, E., Erdogan, H.: Computing weighted solutions in answer set programming. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 416–422. Springer, Heidelberg (2009)
7. Cakmak, D., Erdogan, H., Erdem, E.: Computing weighted solutions in ASP: Representation-based method vs. search-based method. In: Proc. of RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (2010)
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5, 394–397 (1962)
9. Day, W.H.E., Sankoff, D.: Computational complexity of inferring phylogenies by compatibility. Systematic Zoology 35(2), 224–229 (1986)
10. Dworschak, S., Grell, S., Nikiforova, V.J., Schaub, T., Selbig, J.: Modeling biological networks by action languages via answer set programming. Constraints 13(1-2), 21–65 (2008)
11. Eiter, T., Erdem, E., Erdogan, H., Fink, M.: Finding similar or diverse solutions in answer set programming. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 342–356. Springer, Heidelberg (2009)
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, pp. 90–96 (2005)
13. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: A deductive system for non-monotonic reasoning. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 364–375. Springer, Heidelberg (1997)
14. Erdem, E., Erdem, O., Ture, F.: HAPLO-ASP: Haplotype inference using answer set programming. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 573–578. Springer, Heidelberg (2009)
15. Erdem, E., Lifschitz, V., Nakhleh, L., Ringe, D.: Reconstructing the evolutionary history of indo-european languages using answer set programming. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 160–176. Springer, Heidelberg (2003)
16. Erdem, E., Lifschitz, V., Ringe, D.: Temporal phylogenetic networks and logic programming. Theory and Practice of Logic Programming 6(5), 539–558 (2006)
17. Erdem, E., Ture, F.: Efficient haplotype inference with answer set programming. In: Proc. of AAAI, pp. 436–441 (2008)
18. Erdem, E., Wang, F.: Reconstructing the evolutionary history of Chinese dialects (2006) (accepted for presentation at the 39th International Conference on Sino-Tibetan Languages and Linguistics, ICSTLL 2006)
19. Erdem, E., Yeniterzi, R.: Transforming controlled natural language biomedical queries into answer set programs. In: Proc. of BioNLP, pp. 117–124 (2009)
20. Erdogan, H., Bodenreider, O., Erdem, E.: Finding semantic inconsistencies in UMLS using answer set programming. In: Proc. of AAAI (2010)
21. Erdogan, H., Erdem, E., Bodenreider, O.: Exploiting umls semantics for checking semantic consistency among umls concepts. In: Proc. of MedInfo (2010)
22. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proc. of IJCAI, pp. 386–392 (2007)

23. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., Veber, P.: Repair and prediction (under inconsistency) in large biological networks with answer set programming. In: Proc. of KR (2010)
24. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 130–144. Springer, Heidelberg (2008)
25. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Logic Programming: Proceedings of the Fifth International Conference and Symposium, pp. 1070–1080 (1988)
26. Grell, S., Schaub, T., Selbig, J.: Modelling biological networks by action languages via answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 285–299. Springer, Heidelberg (2006)
27. Hennig, W.: Phylogenetic Systematics. University of Illinois Press, Urbana (1966); translated from: Davis, D. D., Zangerl, R.: Grundzuege einer Theorie der phylogenetischen Systematik (1950)
28. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)
29. 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 (2004)
30. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138, 39–54 (2002)
31. Lifschitz, V.: What is answer set programming? In: Proc. of AAAI (2008)
32. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1999)
33. Marques-Silva, J., Sakallah, K.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers 5, 506–521 (1999)
34. Nakhleh, L.: Phylogenetic Networks. Ph.D. thesis, The university of Texas at Austin (2004)
35. Niemelä, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal lp. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 421–430. Springer, Heidelberg (1997)
36. Schaub, T., Thiele, S.: Metabolic network expansion with answer set programming. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 312–326. Springer, Heidelberg (2009)
37. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138, 181–234 (2002)
38. Tari, L., Anwar, S., Liang, S., Hakenberg, J., Baral, C.: Synthesis of pharmacokinetic pathways through knowledge acquisition and automated reasoning. In: Proc. of PSB, pp. 465–476 (2010)
39. Tran, N., Baral, C.: Reasoning about triggered actions in ansprolog and its application to molecular interactions in cells. In: Proc. of KR, pp. 554–564 (2004)
40. Tran, N., Baral, C.: Hypothesizing about signaling networks. J. Applied Logic 7(3), 253–274 (2009)
41. Ture, F., Erdem, E.: Efficient haplotype inference with answer set programming. In: Proc. of AAAI, pp. 1834–1835 (2008)

# ASP at Work: Spin-off and Applications of the DLV System[*]

Giovanni Grasso, Nicola Leone, Marco Manna, and Francesco Ricca

Department of Mathematics, University of Calabria, Italy
{grasso,leone,manna,ricca}@mat.unical.it

**Abstract.** Answer Set Programming (ASP) is a declarative language for knowledge representation and reasoning. After its proposal, in a seminal paper by Michael Gelfond and Vladimir Lifschitz dated in 1988, ASP has been the subject of a broad theoretical research-work, ranging from linguistic extensions and semantic properties to evaluation algorithm and optimization techniques. Later on, the availability of a number of efficient systems made ASP a powerful tool for the fast development of knowledge-based applications. In this paper, we report on the ongoing effort aimed at the industrial exploitation of DLV– one of the most popular ASP systems – in the area of Knowledge Management. Two spin-off companies of University of Calabria are involved in such an exploitation: DLVSYSTEM Srl and Exeura Srl. They have specialized DLV into some Knowledge Management products for Text Classification, Information Extraction, and Ontology Representation and Reasoning, which have allowed to develop a number of successful real-world applications.

**Keywords:** ASP, DLV System, Knowledge Management, Applications.

## 1 Introduction

Answer Set Programming (ASP) is a powerful logic programming language having its roots in the seminal papers [38,39] by Michael Gelfond and Vladimir Lifschitz. ASP is a purely declarative and expressive language that can represent, in its general form allowing for disjunction in rule heads [50] and nonmonotonic negation in rule bodies, *every* problem in the complexity classes $\Sigma_2^P$ and $\Pi_2^P$ (under brave and cautious reasoning, respectively) [27]. The comparatively-high expressive power of ASP comes with the capability of providing, in a compact-yet-elegant way, ready-to-execute formal specifications of complex problems. Indeed, the ASP encoding of a large variety of problems is very concise, simple, and elegant [5,26,36,23].

As an example, consider the well-known NP-complete problem 3-COLORING: given an undirected graph $G = (V, E)$, assign each vertex one of three colors -say, red, green, or blue- such that adjacent vertices always have distinct colors. 3-COLORING can be encoded in ASP as follows:

[*] This work was partially supported by the Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA project of TopClass Srl.

```
vertex(v).      ∀ v ∈ V
edge(u,v).      ∀ (u,v) ∈ E
col(X,red) v col(X,green) v col(X,blue) :- vertex(X).
:- edge(X,Y), col(X,C), col(Y,C).
```

The first two lines introduce suitable facts, representing the input graph $G$, the third line states that each vertex needs to have some color.[1] The last line acts as an integrity constraint since it disallows situations in which two connected vertices are associated with the same color.

After the formal definition of the semantics of ASP, in more than twenty years the core language has been extensively studied and extended by: constructs for dealing with optimization problems [13,16,15,17,14,19,46,62]; probabilities [7]; special atoms handling aggregate functions [32,22,30,42,54]; function symbols [12,10,11,21,1,29,64,65]; complex terms (allowing the explicit usage of lists and sets) [21]; consistency-restoring rules [3]; ontology handling constructs [28,56,55, 57,24]; nested expressions [48,18,53], and many others.

Unfortunately, the high expressiveness of ASP comes at the price of a high computational cost in the worst case, which makes the implementation of efficient systems a difficult task. Nevertheless, starting from the second half of the 1990ies, and even more in the latest years, a number of efficient ASP systems have been released [46,61,62,67,49,2,35,44,47,25]. The availability of efficient systems gave rise to the exploitation of ASP in real-world applications. For instance, one of the first significant applications of ASP was the development of a decision support system for the Space Shuttle flight controllers (the USA-Advisor) [4]. The system was employed by the NASA for maneuvering the Space Shuttle while it is in orbit. Afterwards, many other ASP applications followed, developed for solving complex knowledge-based problems arising in Artificial Intelligence [6,8,34,33,36,52] as well as in Information Integration [45] and in other areas of Knowledge Management (KM) [5,9,41]. These applications have confirmed the viability of the ASP exploitation for advanced knowledge-based tasks, and stimulated further research in this field and some interest also in industry. Currently, the research group of the University of Calabria, which developed the popular ASP system DLV [46], has been trying the industrial exploitation of DLV in the area of Knowledge Management. Two spin-off companies of University of Calabria are involved in such an exploitation: DLVSYSTEM Srl and Exeura Srl.

In this paper, we describe some market-oriented specialization of DLV into Knowledge Management products for Text Classification, Information Extraction (IE), and Ontology Representation and Reasoning. Moreover, we overview some successful applications of these products, and we briefly report also on some further applications exploiting DLV directly. These applications fall in many different domains, including Team Building in a Seaport [40], E-Tourism [43], and E-Government.

[1] The semantics of ASP ensures that solutions are minimal w.r.t. set inclusion, thus each vertex will be associated to precisely one color in each solution.

## 2 The Language of DLV

In this section, we describe the language of the DLV system by examples, providing the intuitive meaning of the main constructs. For further details and the formal definition, we refer to [46,21,31]. We first introduce the basic language, which is based on the founding work by Gelfond and Lifschitz [39] and then we illustrate a number of extensions including aggregates [31], weak constraints [20], complex terms [21], queries, and database interoperability constructs [66].

### 2.1 Basic Language

The main construct in the DLV language is a rule, an expression of the form:

$$Head \ :- \ Body.$$

where *Body* is a conjunction of literals and *Head* is a disjunction of atoms. Informally, a rule can be read as follows: "if *Body* is true, then *Head* is true". A rule without a body is called a *fact*, since it models an unconditional truth (for simplicity :– is omitted); whereas a rule with an empty head, called *strong constraint*, is used to model a condition that must be false in any possible solution. A set of rules is called *program*. The semantics of a program is given by its *answer sets* [39]. A program can be used to model a problem to be solved: the problem's solutions correspond to the answer sets of the program (which are computed by DLV). Therefore, a program may have no answer set (if the problem has no solution), one (if the problem has a unique solution) or several (if the problem has more than one possible solutions).

As an example, consider the problem of automatically creating an assessment test from a given database of questions where each question is identified by a unique string, covers a particular topic, and requires an estimated time to be answered. The input data about questions can be represented by means of a set of facts of type $question(q, topic, time)$; in addition, facts of the form $relatedTopic(topic)$ specify the topics related to the subject of the test. For instance, consider the case in which only four questions are given, represented by facts: `question(q1,computerscience,8). question(q3,math,15). question(q2,computerscience,15). question(q4,math,25).`

Moreover, suppose that computer science is the only topic to be covered by the test, therefore `relatedTopic(computerscience)` is also part of the input facts. The program consisting only of these facts has one answer set $A_1$ containing exactly the five facts.

Assessment creation amounts to selecting a set of questions from the database, according to a given specification. To single out questions related to the subject of the test, one can write the rule:

```
relatedQuestion(Q) :- question(Q,Topic,Time), relatedTopic(Topic).
```

that can be read: "$Q$ is a question related to the test if $Q$ has a topic related to some of the subjects that have to be assessed". Adding this rule to the input facts reported earlier yields one answer set:

· $A_2 = A_1 \cup \{$`relatedQuestion(q1), relatedQuestion(q2)`$\}$

For determining all the possible subsets of related questions the following disjunctive rule can be used:

```
inTest(Q) v discard(Q) :- relatedQuestion(Q).
```

Intuitively, this rule can be read as: "if $Q$ identifies a related question, then either $Q$ is taken in the test or $Q$ is discarded." The rule has the effect of associating each possible choice of related questions with an answer set of the program. Indeed, the answer sets of the program consisting of the above rules and the input facts are: $A_3 = A_2 \cup \{$`discard(q1), discard(q2)`$\}$, $A_4 = A_2 \cup \{$`inTest(q1), discard(q2)`$\}$, $A_5 = A_2 \cup \{$`discard(q1), inTest(q2)`$\}$, $A_6 = A_2 \cup \{$`inTest(q1), inTest(q2)`$\}$, corresponding to the four possible choices of questions $\{\}, \{\mathtt{q1}\}, \{\mathtt{q2}\}, \{\mathtt{q1}, \mathtt{q2}\}$. Note that, the answer sets are minimal with respect to subset inclusion. Indeed, for each question $Q$ there is no answer set in which both `inTest(Q)` and `discard(Q)` appear.

At this point, some strong constraints can be used to single out some solutions respecting a number of specification requirements. For instance, suppose we are interested in tests containing only questions requiring less than 10 minutes to be answered. The following constraint models this requirement:

```
:- inTest(Q), question(Q,Topic,Time), Time < 10.
```

The new program obtained by adding this constraint has only two answer sets: $A_3$ and $A_4$.

## 2.2 Aggregate Functions

More involved properties requiring operations on sets of values can be expressed by aggregates, a DLV construct similar to aggregation in the SQL language. DLV supports five aggregate functions, namely `#sum`, `#count`, `#times`, `#max`, and `#min`.

In our running example, we might want to restrict the included questions to be solvable in an estimated time of less than 60 minutes. This can be achieved by the following strong constraint:

```
:- not #sum{Time,Q: inTest(Q), question(Q,Topic,Time)} < 60.
```

The aggregate sums up the estimated solution times of all questions in the test, and the constraint eliminates all scenarios in which this sum is not less than 60.

## 2.3 Optimization Constructs

The DLV language also allows for specifying optimization problems (i.e., problems where some goal function must be minimized or maximized). This can be achieved by using *weak constraints*. From a syntactic point of view, a weak constraint is like a strong one where the implication symbol `:-` is replaced by `:~`. Contrary to strong constraints, weak constraints allow for expressing conditions that *should* be satisfied, but not necessarily have to be.

The informal meaning of a weak constraint :∼ $B$ is "try to falsify $B$", or "$B$ should preferably be false". Additionally, a weight and a priority level for the weak constraint may be specified enclosed in square brackets (by means of positive integers or variables). The answer sets minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level and, among them, those which minimize the sum of weights of the violated weak constraints in the next lower level, and so on.

As an example, if we want to prefer quick-to-answer questions in tests, the following weak constraint represents this desideratum.

```
:∼ inTest(Q), question(Q,Topic,Time). [Time:1]
```

Intuitively, each question in the test increases the total weight of the solution by its estimated solution time. Thus, solutions where the total weight is minimal are preferred.

### 2.4 Complex Terms

The DLV language allows for the use of complex terms. In particular, it supports function symbols, lists, and sets. Prolog-like syntax is allowed for both function symbols and lists, while sets are explicitly represented by listing the elements in brackets.

As an example, we enrich the question database for allowing two types of questions, open and multiple choice. Input questions are now represented by facts like the following:

```
question(q1, math, open(text), 10).
question(q2, physics, multiplechoice(text,{c1,c2,c3},{w1,w2,w3}), 3).
```

where function symbols `open` and `multiplechoice` are used for representing the two different types of questions. In particular, `open` is a unary function whose only parameter represents the text of the question, while `multiplechoice` has three parameters, the text of the question, a set containing correct answers and another set of wrong answers.

The use of sets allows for modeling multi-valued attributes, while function symbols can be used for modeling "semi-structured" information.

Handling complex terms is facilitated by a number of built-in predicates. For instance, the following rule uses the `#member` built-in for selecting correct answers given by a student in the test:

```
correctAnswer(Stud,QID,Ans) :- inTest(QID), answer(Stud,QID,Ans),
   question(QID,To,multiplechoice(Tx,Cs,Ws),Ti), #member(Ans,Cs).
```

### 2.5 Queries

The DLV language offers the possibility to express conjunctive queries. From a syntactic point of view, a query in DLV is a conjunction of literals followed by a question mark. Since a DLV program may have more than one answer set, there are two different reasoning modes, brave and cautious, to compute a query answer. In the brave (resp. cautious) mode, a query answer is true if the corresponding conjunction is true in some (resp. all) answer sets.

For instance, the answers to the following simple query are the questions having as topic `computerscience` that are contained in some (resp. all) answer sets of the program when brave (resp. cautious) reasoning is used.

```
inTest(Q), question(Q,computerscience,T)?
```

### 2.6 Database Interoperability

The DLV system supports interoperability with databases by means of the `#import`/`#export` commands for importing and exporting relations from/to a DBMS. The `#import` command reads tuples from a specified table of a relational database and stores them as facts with a predicate name provided by the user. In our example, questions can be retrieved from a database by specifying in the program the following directive:

```
#import(questionDB,"user","passwd","SELECT * FROM question",question).
```

where `questionDB` is the name of the database, `"user"` and `"passwd"` are the data for the user authentication, `"SELECT * FROM question"` is an SQL query that constructs the table that will be imported, and `question` is the predicate name which will be used for constructing the new facts. Similarly, command `#export` allows for exporting the extension of a predicate in an answer set to a database.

## 3 Spin-off Companies

In this section, we give a brief overview of the two spin-off companies (Dlvsystem Srl and Exeura Srl) that are engaged in the industrial exploitation of ASP in the area of Knowledge Management.

### 3.1 Dlvsystem Srl

The Dlvsystem[2] Srl is a research spin-off of the University of Calabria. It was founded in 2005 by professors and researchers from University of Calabria, TU Wien, and University of Oxford, with the aim of developing and commercially-distributing the DLV [46] system.

The founders of Dlvsystem are all the key participants of the DLV project. Among them we recall Prof. T. Eiter (TU Wien), Prof. W. Faber, (University of Calabria), Prof. G. Gottlob (Oxford University), Prof. N. Leone (University of Calabria), and Dr. G. Pfeifer (Novell). Dlvsystem currently employs 4 permanent workers, and many consultants. The staff members have an excellent technical profile and backgrounds, with postdoctoral degrees in Informatics and an in-depth knowledge on ASP.

[2] See `http://www.dlvsystem.com`

In the last few years, DLVSYSTEM Srl has been actively involved in several projects on both research and applications. Currently, its main project concerns the engineering of DLV and its extension for information integration and for the semantic web. It is supported by the Regione Calabria and EU under POR Calabria FESR 2007-2013 within the PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) programme. Moreover, DLVSYSTEM currently provides consultancy on ASP and DLV usage in a number of knowledge based applications, including the team-building application and the e-tourism application described in Section 5.1 and 5.2, respectively.

### 3.2 Exeura Srl

Exeura[3] Srl is a spin-off company founded by a group of professors from the University of Calabria. Exeura has the aim of employing advanced technologies, stemming from research results, for developing industrial products. The targets of the company are both software houses interested in enhancing their platforms by enriching them with intelligent and adaptive capabilities, and companies asking for specific knowledge-based applications.

Exeura enrolls about 30 highly-skilled employees which develop innovative Knowledge Management technologies and advanced IT services.

Among the products we recall *Rialto*, a data mining suite, OntoDLV [55,56], an advanced platform for ontology management and reasoning, HiLeX [59,58] a text extraction system and OLEX [60] a document classification system. The last three products are based on the ASP system DLV, and have been already successfully applied in several contexts (more details in the next sections).

In the last few years, Exeura completed several research projects and developed solutions for data mining and knowledge management systems.

## 4 DLV-Based Systems for KM

In this section, we overview the three main KM industrial products that are strongly based on DLV: OntoDLV [55,56], OLEX [60], HiLeX [59,58]. A number of real-world applications based on these systems are described in Section 5.

### 4.1 OntoDLV

The basic ASP language is not well-suited for ontology specifications, since it does not directly support features like classes, taxonomies, individuals, etc. These observations motivated the development of OntoDLV [55,56], which is a system for ontologies specification and reasoning based on DLV. Indeed, by using OntoDLV, domain experts can create, modify, store, navigate, and query ontologies; and, at the same time, application developers can easily develop their own knowledge-based applications. OntoDLV implements a powerful logic-based ontology representation language, called OntoDLP, which is an extension

[3] See `http://www.exeura.com`

of (disjunctive) ASP with all the main ontology constructs including classes, inheritance, relations, and axioms. Importantly, OntoDLV supports a powerful interoperability mechanism with OWL [63], allowing the user to retrieve information from external OWL ontologies and to exploit this data in OntoDLP ontologies and queries.

In OntoDLP, a *class* can be thought of as a collection of individuals who belong together because they share some features. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifiers (*oid*'s) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute has a name and a type, which is, in truth, a class. This allows for the specification of *complex objects* (objects made of other objects). Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*). Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type). OntoDLP relations are strongly typed while in ASP relations are represented by means of simple flat predicates. Importantly, OntoDLP supports two kind of classes and relations:

1. (base) classes and (base) relations, that correspond to basic facts (that can be stored in a database); *and*
2. *collection* classes and *intensional* relations, that correspond to facts that can be inferred by logic programs.

In particular, *collection classes* are mainly intended for object reclassification (i.e., for repeatedly classifying individuals of an ontology). For instance, the following statement declares a class modeling questions, which has two attributes, namely: `topic` of type `Topic`, and `time` of type `Time`.

```
class Question(topic:Topic, time:Time).
```

As in ASP, logic programs are sets of logic rules and constraints. However, OntoDLP extends the definition of logic atoms by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). This way, the OntoDLP rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of object-oriented systems. In addition, logic programs are organized in *reasoning modules*, to take advantage of the benefits of modular programming. For example, with the following program we single out questions having the same topic:

```
module (SameTopic) {
   sameTopic(Q1,Q2) :- Q1:Question(topic:T), Q2:Question(topic:T). }
```

Note that, attributes are typed (and the system ensures that only well-typed rules are accepted as input), and the presence of attribute names allows for more compact and readable rule specifications, where irrelevant attributes can be omitted.

The core of OntoDLV is a rewriting procedure [56] that translates ontologies and reasoning modules to equivalent ASP programs run on the DLV system.

Importantly, if the rewritten program is stratified and non-disjunctive [39,37,51] (and the input ontology resides in relational databases), then the evaluation is carried out directly in mass memory by exploiting a specialized version of the same system, called DLV$^{DB}$ [66]. Note that, since class and relation specifications are rewritten into stratified and non-disjunctive programs, queries on ontologies can always be evaluated by exploiting a DBMS. This makes the evaluation process very efficient, and allows the knowledge engineer to formulate queries in a language more expressive than SQL. Clearly, more complex reasoning tasks are dealt with by exploiting the standard DLV system instead.

## 4.2 OLEX

The OntoLog Enterprise Categorizer System (OLEX) [60] is a corporate classification system supporting the entire content classification life-cycle, including document storage and organization, pre-processing and classification. OLEX exploits a reasoning-based approach to text classification which synergically combines: ($a$) pre-processing technologies for a symbolic representation of texts; and ($b$) a categorization rule language. Logic rules, indeed, provide a natural and powerful way to encode how document contents may relate to document categories. More in detail, the main task of OLEX is text categorization, which consists on assigning documents (containing natural language texts) to predefined categories on the basis of their content. OLEX exploits a method for the automatic construction of rule-based text classifiers, which are sets of *classification rules* of the form: "if at least one of the terms $T_1, \ldots, T_n$ occurs in document $d$, and none of the terms $T_{n+1}, \ldots, T_r$ $(r > n > 0)$ occurs in $d$, then classify $d$ under category *cat*", where each $T_i = t_1^i \wedge \ldots \wedge t_k^i$ $(k > 0)$ is a conjunction of occurrences of terms. A classification rule of this kind is easily encoded in ASP as:

```
positiveForCategory(cat,Doc) :- term(Doc,t$_1^1$),...,term(Doc,t$_k^1$).
...
positiveForCategory(cat,Doc) :- term(Doc,t$_1^n$),...,term(Doc,t$_k^n$).

negativeForCategory(cat,Doc) :- term(Doc,t$_1^{n+1}$),...,term(Doc,t$_k^{n+1}$)
...
negativeForCategory(cat,Doc) :- term(Doc,t$_1^r$),...,term(Doc,t$_k^r$)

classify(cat,Doc) :- positiveForCategory(cat,Doc),
                     not negativeForCategory(cat,Doc).
```

where facts of the form $term(d, t)$ associate terms $t$ occurring in document $d$. Clearly, the system has to pre-process input documents in order to produce a logic representation of their contents (i.e., *term* facts). Essentially, the OLEX pre-processor performs the following tasks: Pre-Analysis and Linguistic Analysis. The former consists of document normalization, structural analysis and tokenization; whereas the latter includes lexical analysis, which determines the Part of Speech (PoS) of each token, reduction (elimination of the stop words), and frequency analysis. The obtained facts are fed into the DLV system together with the classification rules for computing an association between the processed document and categories.

## 4.3 HiLeX

HiLeX [59,58] is an advanced system for ontology-based information extraction from semi-structured (document repositories, digital libraries, Web sites, ...) and unstructured documents (mainly, free texts written in natural language), that has been (already) exploited in many relevant real-world applications. In practice, the HiLeX system implements a semantic approach to the Information Extraction problem based on a new-generation semantic conceptual model by exploiting:

- · Ontologies as knowledge representation formalism;
- · A general document representation model able to unify different document formats (html, pdf, doc, ...);
- · The definition of a formal attribute grammar able to describe, by means of declarative rules (a sort of "semantic enhanced regular expressions"), objects/classes w.r.t. a given ontology.

HiLeX is based on OntoDLP for describing ontologies, since this language perfectly fits the definition of semantic extraction rules.

Regarding the unified document representation, the idea is that a document (unstructured or semi-structured) can be seen as a suitable arrangement of objects in a two-dimensional space. Each object has its own semantics, is characterized by some attributes, and is located in a two-dimensional area of the document, called *portion*. Each portion "contains" one or more objects and an object can be recognized in different portions.

The language of HiLeX is founded on the concept of *ontology descriptor*. A "descriptor" looks like a production rule in a formal attribute grammar, where syntactic items are replaced by ontology elements, and where extensions for managing two-dimensional objects are added. Each descriptor allows us to describe:

- · An ontology object in order to recognize it in a document; *or*
- · How to "generate" a new object that may be added in the original ontology.

In particular, an object may have more than one descriptor, allowing one to recognize the same kind of information when it is presented in different ways; but also, a descriptor might generate more than one object.

In the following, we show some sentences (and their intuitive semantics) hightailing some of the HiLeX features for IE tasks. Suppose we have a semi-structured/unstructured document $d$ and the following OntoDLP ontology:

```
class Token (value:string).
relation hasLemma (tk:token, lm:string, tag:string).
relation synonymous(lemma1:string, lemma2:string).
   synonymous(L1,L2) :- synonymous(L2,L1).
   synonymous(lemma1:"street", lemma2:"road").
   ...
class StockMarket(name:string).
   sm01:StockMarket(name:"NASDAQ").
   sm02:StockMarket(name:"Dow Jones").
   ...
class Evaluation(sm:StockMarket, value:integer).
```

First of all, without writing any descriptor, we could “select” (in *d*) any *token* of length three (only *whitespace* is filtered) by writing the rule:

```
<T:Token(value:V), #hasLenght(V,3)>?
```

that behaves like a query run on a document rather than on a database (`#hasLenght` is a built-in function). Moreover, if we enable the Natural Language Processing (NLP) Tool, the HiLeX system automatically and transparently populates (from *d*) relation `hasLemma`. Suppose that we want to “select” the *tokens* (in *d*) whose lemmas are synonymous of *street* we might use the following rule:

```
<T:Token(), hasLemma(T,L,"noun"), synonymous(L,"street")>?
```

As mentioned before, HiLeX allows to define descriptors for existing objects. For instance, the following rules “define” how to “recognize” *Stock Markets*:

```
<sm01> -> <T:token(V), #tolower(V,"nasdaq")>

<sm02> -> <T1:token(V1), #tolower(V1,"dowjones")> |
   <T2:token(V2), #tolower(V2,"dow")> <T3:token(V3), #tolower(V3,"jones")>
```

where the above two descriptors can be, respectively, read as follows: “recognize object `sm01` in *d* whenever there is a token whose value is `nasdaq`” and “recognize object `sm02` (in *d*) if there is either a token whose value is `dowjones` or there are two consecutive tokens whose values are `dow` and `joens`”. Moreover, HiLeX allows for generating new ontology objects by combining already recognized objects:

```
<Evaluation(SM,N)> -> <SM:StockMarket()> <N:Number()>
```

Here, we create a new evaluation instance whenever an instance of class `Number` follows an instance of class `StockMarket` in the document.

## 5 Applications

In this section, we report a brief description of a number of applications and commercial products employing DLV.

### 5.1 Team-Building in the Gioia-Tauro Seaport

In this section, we give a short overview of a system [40], based on DLV, that has been employed in the port authority of Gioia Tauro for solving a problem about *Intelligent resource allocation* and *Constraint Satisfaction*: the automatic generation of the teams of employees in the seaport. In detail, the seaport of Gioia Tauro[4] is the largest transshipment terminal of the Mediterranean Sea. Several ships of different size moor in the port every day, transported vehicles are handled, warehoused, if necessary technically processed and then delivered to their final destination. The goal is to serve them as soon as possible. Data regarding the mooring ships (arrival/departure date, number and kind of vehicles, etc.), is available at least one day in advance; and, suitable teams of employees have to be arranged for the purpose. Teams are subject to many conditions. Some constraints are imposed by the contract (e.g., an employee cannot work more than 36 hours per week, etc.), some other by the required skills. Importantly, heavy/dangerous roles have to be turned over, and a fair distribution of the workload has to be guaranteed. Once the information regarding mooring ships is received, the management easily produces a meta-plan specifying the number of employees required for each skill; but a more difficult task is to assign the available employees to shifts and roles (each employee might cover several roles according to his/her skills) in such a way that the above-mentioned constrains can be satisfied every day. The impossibility of allocating teams to incoming ships might cause delays and/or violations of the contract with shipping companies, with consequent pecuniary sanctions for the company serving the seaport. To cope with this crucial problem DLV has been exploited for developing a team builder. A simplified version of the kernel part of the employed ASP program is reported in Figure 1. The inputs are modeled as follows:

· the employees and their skills by predicate *skill(employee, skill)*;
· a meta-plan specification by predicate *metaPlan(shift, skill, neededEmployees, duration)*;
· weekly statistics specifying, for each employee, both the number of worked hours per skill and the last allocation date by predicate *wstat(employee, skill, hours, lastTime)*;
· absent employees by predicate *absent(employee)*;
· employees excluded due to a management decision by predicate *manuallyExcluded(employee).*

Following the guess&check programming methodology [46], the disjunctive rule $r$ (see Figure 1) generates the search space by guessing the assignment of a number of available employees to the shift in the appropriate roles. Absent or manually excluded employees, together with employees exceeding the maximum number of weekly working hours are automatically discarded. Then, admissible solutions are selected by means of constraints: $c_1$ discards assignments with a wrong number of employees for some skill; $c_2$ avoids that an employee covers two

[4] See `http://www.portodigioiatauro.it/index.php`

```
r    assign(E,Sh,Sk) v nAssign(E,Sh,Sk) :- skill(E,Sk),
     metaPlan(Sh,Sk,_, D), not absent(E), not manuallyExcluded(E),
     workedHours(E,Wh), Wh + D ≤ 36.
c1   :- metaPlan(Sh,Sk,EmpNum,_), #count{E : assign(E,Sh,Sk)} ≠ EmpNum.
c2   :- assign(E,Sh,Sk1), assign(E,Sh,Sk2), Sk1 ≠ Sk2.
c3   :- wstats(E1,Sk,_,LastTime1), wstats(E2,Sk,_,LastTime2),
        LastTime1 > LastTime2, assign(E1,Sh,Sk), not assign(E2,Sh,Sk).
c4   :- workedHours(E1,Wh1), workedHours(E2,Wh2), threshold(Tr),
        Wh1 + Tr < Wh2, assign(E1,Sh,Sk), not assign(E2,Sh,Sk).
r'   :- workedHours(E,Wh) :- skill(E,_), #count{H,E : wstats(E,_,H,_)} = Wh.
```

**Fig. 1.** Team Builder Encoding

roles in the same shift; $c_3$ implements the turnover of roles; and $c_4$ guarantees a fair distribution of the workload. Finally, rule $r'$ computes the total number of worked hours per employee. Note that, only the kernel part of the employed logic program is reported here (in a simplified form), and many other constraints were developed, tuned and tested.

The complete system features a Graphical User Interface (GUI) developed in Java, and either builds new teams or completes the allocation automatically when the roles of some key employees are fixed manually. Computed teams can be also modified manually, and the system is able to verify whether the manually-modified team still satisfies the constraints. In case of errors, causes are outlined and suggestions for fixing a problem are proposed. E.g., if no plan can be generated, then the system suggests the user to relax some constraints. In this application, the pure declarative nature of the language allowed for refining and tuning both problem specifications and encodings together while interacting with the stakeholders of the seaport. It is worth noting that, the possibility of modifying (by editing text files) in a few minutes a complex reasoning task (e.g., by adding new constraints), and testing it "on-site" together with the customer has been a great advantage of the approach. The system, developed by Exeura Srl, has been adopted by the company ICO BLG operating automobile logistics in the seaport of Gioia Tauro.

## 5.2 E-Tourism: The IDUM System

IDUM [43] is an e-tourism system developed in the context of the project "IDUM: Internet Diventa Umana" funded by the administration of the Calabria Region. The IDUM system helps both employees and customers of a travel agency in finding the best possible travel solution in a short time. It can be seen as a "mediator" system finding the best match between the offers of the tour operators and the requests of the turists. IDUM, like other existing portals, has been equipped with a proper (web-based) user interface; but, behind the user interface there is an "intelligent" core that exploits knowledge representation and reasoning technologies based on ASP. In IDUM, the information regarding the touristic offers provided by tour operators is received by the system as a set of e-mails. Each e-mail might contain plain text and/or a set of leaflets, usually

```
class Place (description:string).
class TransportMean (description:string).
class TripKind (description:string).
class Customer (firstName:string, lastName:string, birthDate:Date,
   status:string, childNumber:positive integer, job:Job).
class TouristicOffer (start:Place, destination:Place, kind:TripKind,
   means:TransportMean, cost:positive integer, fromDay:Date,
   toDay:Date, maxDuration:positive integer, deadline:Date, uri:string).
relation PlaceOffer (place:Place, kind:TripKind).
relation SuggestedPeriod (place:Place, period:positive integer).
relation BadPeriod (place:Place, period:positive integer).
intentional relation Contains (pl1:place, pl2:place) {
   Contains(P1,P2) :- Contains(P1,P3), Contains(P3,P2).
   Contains('Europe', 'Italy').  ... }
```

**Fig. 2.** Some entities from the Tourism Ontology

distributed as pdf or image files which store the details of the offer (e.g., place, accommodation, price etc.). Leaflets are devised to be human-readable, might mix text and images, and usually do not have the same layout. E-mails (and their content) are automatically processed by using the HiLeX system, and the extracted data about touristic offers is used to populate an OntoDLP ontology that models the domain of discourse: the *"Tourism Ontology"*. The resulting ontology is then analyzed by exploiting a set of reasoning modules combining the extracted data with the knowledge regarding places (geographical information) and users (user preferences). The system mimics the typical deductions made by a travel agency employee for selecting the most appropriate answers to the user needs. In the following, we briefly describe the Tourism Ontology and the implementation of the ASP-based features of IDUM.

**The Tourism Ontology.** The Tourism Ontology has been specified by analyzing the nature of the input in cooperation with the staff of a real touristic agency. This way, the key entities that describe the process of organizing and selling a complete holiday package could be modeled. In particular, the Tourism Ontology models all the required information, such as geographic information, kind of holiday, transportation means, etc. In Figure 2, we report some entities constituting the Ontology. In detail, class `Customer` allows us to model the personal information of each customer. The kind of trip is represented by using a class `TripKind`. Examples of `TripKind` instances are: `safari`, `sea_holiday`, etc. In the same way, e.g., `airplane`, `train`, etc., are instances of class `TransportMean`. Geographical information is modeled by means of class `Place`, which has been populated by exploiting Geonames[5], one of the largest publicly-available geographical databases. Moreover, each place is associated with a kind of trip by means of the relation `PlaceOffer` (e.g., Kenya offers safari, Sicily offers both sea and sightseeing). Importantly, the natural *part-of* hierarchy of places is easily

[5] See `www.geonames.org`

modeled by using the intensional relation `Contains`. The full hierarchy is computed by evaluating a rule (which, basically, encodes the transitive closure).

The mere geographic information is, then, enriched by other information that is usually exploited by travel agency employees for selecting a travel destination. For instance, one might suggest avoiding sea holidays in winter; whereas, one should be recommended a visit to Sicily in summer. This was encoded by means of the relations `SuggestedPeriod` and `BadPeriod`. Finally, the `TouristicOffer` class contains an instance for each available holiday package. The instances of this class are added either automatically, by exploiting the HiLeX system or manually by the personnel of the agency. This has been obtained by encoding several ontology descriptors (actually, we provided several descriptors for each kind of file received by the agency). As an example, the following descriptor:

```
<TouristicOffer(destination:D,fromDay:FD,toDay:TD)> -> <D:Place("Sicily")>
   <T1:Token()>* <FD:Date()> <T2:Token()>* <TD:Date()>.
```

was used to extract instances of class `TouristicOffer` (from leaflets) regarding trips to Sicily. Moreover, it also extracts the period in which this trip is offered.

**Personalized Trip Search.** This feature has been conceived to simplify the task of selecting the holiday packages that best fit the customer needs. In a typical scenario, when a customer enters the travel agency, what has to be clearly understood (for properly selecting a holiday package fitting the customer needs) is summarized in the following four words: *where*, *when*, *how*, and *budget*. However, the customer does not directly specify all this information, for example, he can ask for a sea holiday in January but he does not specify a precise place, or he can ask for a kind of trip that is unfeasible in a given period. In IDUM, current needs are specified by filling an appropriate search form, where some of the key information has to be provided (i.e., where and/or when and/or available money and/or how). Note that, the Tourism Ontology models the knowledge of the travel agent. Moreover, the extraction process continuously populates the ontology with new touristic offers. Thus, the system, by running a specifically devised reasoning module, combines the specified information with the one available in the ontology, and shows the holiday packages that best fit the customer needs. For example, suppose that a customer specifies the kind of holiday and the period, then the following (simplified) module creates a selection of holiday packages:

```
module(kindAndPeriod) {
 %detect possible and suggested places
 possiblePlace(P) :- askFor(tripKind:K), PlaceOffer(place:P, kind:K).

 suggestPlace(P) :- possiblePlace(P), askFor(period:D),
                    SuggestedPeriod(place:P1, period:D),
                    not BadPeriod(place:P1, period:D).

 %select possible packages
 possibleOffer(O) :- O:TouristicOffer(destination:P), possiblePlace(P). }
```

The first two rules select: possible places (i.e., the ones that offer the kind of holiday in input); and places to be suggested (because they offer the required kind of holiday in the specified period). Finally, the remaining rule searches in the available holiday packages the ones which offer an holiday that matches the original input (possible offer). The example above reports one of the several reasoning modules that have been devised for implementing the intelligent search.

### 5.3 Text Categorization Applications

In this section, we give a brief overview of some applications developed by Exeura Srl, which are based on OLEX and applied in different domains.

**e-Government.** In this field, the objective was to classify legal acts and decrees issued by public authorities. The system employs an ontology based on TE.SE.O (Thesaurus of the Senato della Repubblica Italiana), on an archive that contains a complete listing of words arranged in groups of synonyms and related concepts regarding juridical terminology employed by the Italian Parliament, and on a set of categories identified by analyzing a corpus of 16,000 documents of the public administration. The system, validated with the help of the administrative employees of the *Regione Calabria*, performed very well. In particular, it obtained an f-measure of 92% and a mean precision of 96% in real-world documents.

**e-Medicine.** Here, the goal was to develop a system able to automatically classify case histories and documents containing clinical diagnoses. In particular, it was developed for conducting epidemiological analysis, by the ULSS n.8 (which is, a local authority for health services) of the area of Asolo, in the Italian region Veneto. Basically, available case histories are classified by the system in order to help the analysts of the ULSS while browsing and searching documents regarding specific pathologies, supplied services, or patients living in a given place etc. The application exploits an ontology of clinical case histories based on both the MESH (Medical Subject Headings) ontology and ICD9-CM a system employed by the Italian Ministry of the Heath for handling data regarding medical services (e.g., X-Rays analysis, plaster casts, etc.). The system has been deployed and is currently employed by the personnel of the ULSS of Asolo.

### 5.4 Other Applications

In this Section, we report on a number of further real-world applications still exploiting DLV.

The European Commission funded a project on Information Integration, which produced a sophisticated and efficient data integration system, called INFOMIX, which uses DLV at its computational core [45], and DLV was successfully employed to develop a real-life integration system for the information system of the University of Rome "La Sapienza". The DLV system has been used for Census Data Repair [33], where the main objective was to identify and eventually repair errors in census data. DLV has been employed at CERN, the European Laboratory for Particle Physics, for an advanced deductive database application that involves complex knowledge manipulation on large-sized databases. The Polish company Rodan Systems S.A. has exploited DLV in a tool (used by the Polish Securities and Exchange Commission) for the detection of price manipulations and unauthorized use of confidential information. The Italian region Calabria is experimenting a reasoning service exploiting DLV for automatic itinerary search. The system builds itineraries from a given place to another in the region. In the area of self-healing Web Services, DLV was exploited for implementing the computation of minimum cardinality diagnoses [34].

## 6 Conclusion

In this paper we have reported on the industrial exploitation of the ASP system DLV. In particular, we have presented two spin-off companies of the University of Calabria: DLVSYSTEM Srl and Exeura Srl which have as vision the commercialization of DLV and the development of DLV-based software products. Although the industrial exploitation of DLV has started very recently, there are already a number of systems having DLV as kernel component. The most valuable ones, for team-building, e-tourism, and text categorization, are also described in detail in this paper.

## References

1. Alviano, M., Faber, W., Leone, N.: Disjunctive ASP with Functions: Decidable Queries and Effective Computation. In: TPLP, 26th Int'l. Conference on Logic Programming (ICLP 2010) Special Issue, vol. 10(4-6), pp. 497–512 (2010)
2. 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`
3. Balduccini, M., Gelfond, M.: Logic Programs with Consistency-Restoring Rules. In: Proceedings of AAAI 2003 Spring Symposium Series (2003)
4. Balduccini, M., Gelfond, M., Watson, R., Nogeira, 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)
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving.CUP (2003)
6. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Logic-Based Artificial Intelligence, pp. 257–279 (2000)
7. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. TPLP 9(1), 57–144 (2009)
8. 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)
9. 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 (1996)
10. Baselice, S., Bonatti, P.A., Criscuolo, G.: On Finitely Recursive Programs. TPLP (TPLP) 9(2), 213–238 (2009)

11. Bonatti, P.A.: Reasoning with infinite stable models II: Disjunctive programs. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 333–346. Springer, Heidelberg (2002)
12. Bonatti, P.A.: Reasoning with infinite stable models. Artificial Intelligence 156(1), 75–111 (2004)
13. Brewka, G.: Logic Programming with Ordered Disjunction. In: NMR 2002, pp. 67–76 (April 2002)
14. Brewka, G.: Answer Sets: From Constraint Programming Towards Qualitative Optimization. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 34–46. Springer, Heidelberg (2003)
15. Brewka, G., Benferhat, S., Le Berre, D.: Qualitative Choice Logic. In: Proceedings of (KR 2002), pp. 158–169 (April 2002)
16. Brewka, G., Niemelä, I., Syrjänen, T.: Implementing ordered disjunction using answer set solvers for normal programs. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 444–455. Springer, Heidelberg (2002)
17. Brewka, G., Niemelä, I., Truszczyński, M.: Answer Set Optimization. In: IJCAI 2003, Acapulco, Mexico, pp. 867–872 (August 2003)
18. Bria, A., Faber, W., Leone, N.: Normal form nested programs. FI 96(3), 271–295 (2009)
19. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE 12(5), 845–860 (2000)
20. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE TKDE 12(5), 845–860 (2000)
21. 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)
22. 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)
23. 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)
24. Drabent, W., Eiter, T., Ianni, G., Krennwallner, T., Lukasiewicz, T., Małuszyński, J.: Hybrid Reasoning with Rules and Ontologies. In: Bry, F., Małuszyński, J. (eds.) Semantic Techniques for the Web. LNCS, vol. 5500, pp. 1–49. Springer, Heidelberg (2009)
25. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: KR 2008, Sydney, Australia, 2008, pp. 422–432. AAAI Press, Menlo Park (2008)
26. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System. In: Logic-Based Artificial Intelligence, pp. 79–103 (2000)
27. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (1997)
28. 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, nburgh, UK (August 2005)
29. Elkabani, I., Pontelli, E., Son, T.C.: Smodels$^A$ - A System for Computing Answer Sets of Logic Programs with Aggregates. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 427–431. Springer, Heidelberg (2005)

30. 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)
31. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. In: AI (2010) (accepted for publication)
32. 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)
33. Franconi, E., Palma, A.L., Leone, N., Perri, S., Scarcello, F.: 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)
34. Friedrich, G., Ivanchenko, V.: Diagnosis from first principles for workflow executions. Technical report. Alpen Adria University, Applied Informatics, Klagenfurt, Austria (2008), `http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf`
35. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007, pp. 386–392 (January 2007)
36. 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)
37. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective. AI 138(1–2), 3–38 (2002)
38. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: ICLP/SLP 1988, pp. 1070–1080. MIT Press, Cambridge (1988)
39. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
40. Grasso, G., Iiritano, S., Leone, N., Lio, V., Ricca, F., Scalise, F.: An ASP-Based System for Team-Building in the Gioia-Tauro Seaport. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 40–42. Springer, Heidelberg (2010)
41. 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)
42. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. JACM 48(4), 880–907 (2001)
43. Ielpa, S.M., Iiritano, S., Leone, N., Ricca, F.: An ASP-Based System for e-Tourism. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 368–381. Springer, Heidelberg (2009)
44. 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)
45. 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, 2005, pp. 915–917. ACM Press, New York (2005)
46. 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)
47. 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)

48. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. AMAI 25(3–4), 369–389 (1999)
49. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, AAAI Press, Menlo Park (2002)
50. Minker, J.: On Indefinite Data Bases and the Closed World Assumption. In: Loveland, D.W. (ed.) CADE 1982. LNCS, vol. 138, pp. 292–308. Springer, Heidelberg (1982)
51. Minker, J.: Overview of Disjunctive Logic Programming. AMAI 12, 1–24 (1994)
52. 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)
53. Pearce, D., Sarsakov, V., Schaub, T., Tompits, H., Woltran, S.: A Polynomial Translation of Logic Programs with Nested Expressions into Disjunctive Logic Programs: Preliminary Report. In: NMR 2002 (2002)
54. Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. TPLP 7(3), 301–353 (2007)
55. 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 19(4), 643–670 (2009)
56. Ricca, F., Leone, N.: Disjunctive Logic Programming with types and objects: The DLV$^+$ System. Journal of Applied Logics 5(3), 545–573 (2007)
57. Rosati, R.: On Combining Description Logic Ontologies and Nonrecursive Datalog Rules. In: Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems, Berlin, Heidelberg, pp. 13–27 (2008)
58. Ruffolo, M., Leone, N., Manna, M., Saccà, D., Zavatto, A.: Exploiting ASP for Semantic Information Extraction. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK, pp. 248–262 (2005)
59. Ruffolo, M., Manna, M.: HiLeX: A System for Semantic Information Extraction from Web Documents. In: ICEIS (Selected Papers). Lecture Notes in Business Information Processing, vol. 3, pp. 194–209 (2008)
60. Rullo, P., Policicchio, V.L., Cumbo, C., Iiritano, S.: Olex: Effective Rule Learning for Text Categorization. IEEE TKDE 21, 1118–1132 (2009)
61. Simons, P.: Smodels Homepage (1996), `http://www.tcs.hut.fi/Software/smodels/`
62. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI 138, 181–234 (2002)
63. Smith, M.K., Welty, C., McGuinness, D.L.: OWL web ontology language guide. W3C Candidate Recommendation (2003), `http://www.w3.org/TR/owl-guide/`
64. Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. TPLP 7, 355–375 (2007)
65. Syrjänen, T.: Omega-Restricted Logic Programs. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 267–279. Springer, Heidelberg (2001)
66. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP 8, 129–165 (2008)
67. Zhao, Y.: ASSAT homepage, since (2002), `http://assat.cs.ust.hk/`

# Combining Answer Set Programming and Prolog: The $\mathbb{ASP}-\mathbb{PROLOG}$ System

Enrico Pontelli, Tran Cao Son, and Ngoc-Hieu Nguyen

Department of Computer Science
New Mexico State University
{epontell,tson,nhieu}@cs.nmsu.edu

**Abstract.** This paper presents a framework for combining Answer Set Programming and Prolog, as developed in the $\mathbb{ASP}-\mathbb{PROLOG}$ system. The paper focuses on the development of a model-theoretical account for $\mathbb{ASP}-\mathbb{PROLOG}$. It also illustrates the use of $\mathbb{ASP}-\mathbb{PROLOG}$ in several practical applications in knowledge representation and reasoning, common-sense reasoning, and planning.

## 1 Introduction

*Answer Set Programming (ASP)* [Niemelä, 1999, Marek and Truszczyński, 1999] arises from the research in logic programming and has become a viable declarative programming paradigm. ASP has been applied in several important areas, such as diagnosis and planning for space shuttle missions [Balduccini et al., 2001], multi-agent reasoning [Baral et al., 2010], natural language processing [Baral et al., 2008], bioinformatics [Gebser et al., 2010], and LTL model checking [Heljanko and Niemelä, 2003]. The continued improvements of answer set solvers, with respect to the representation language and the implementation techniques, enable ASP to be competitive with other state-of-the-art technologies in several domains (e.g., planning [Tu et al., 2010]).

The growing availability of data, e.g., the online deployment of portals and domain specific data repositories and resources, has brought new challenges to Computer Science. One of these challenges is the integration of several knowledge bases, possibly under different semantics, for use as a unique information source. This need has inspired the development of *Multi-Context Systems (MCS)* [Brewka and Eiter, 2007], of which the $\mathbb{ASP}-\mathbb{PROLOG}$ system [El-Khatib et al., 2005] has been one of the first representatives. An implementation of MCS is described in [Dao-Tran et al., 2010]. The key difference between these two systems lies in the fact that the implementation of MCS through the DMCS algorithm [Dao-Tran et al., 2010] focuses on computing the partial equilibria (or the semantics) of MCSs, while the $\mathbb{ASP}-\mathbb{PROLOG}$ system provides an environment for query answering with multiple knowledge bases. In other words, the former represents a bottom-up evaluation of MCSs while $\mathbb{ASP}-\mathbb{PROLOG}$ provides a top-down one. To the best of our knowledge, there exists no formal comparison between these two systems. One of the main difficulties in this endeavor is the lack of a model-theoretical account for $\mathbb{ASP}-\mathbb{PROLOG}$ system.

In this paper, we address the aforementioned issue and present a model-theoretical semantics for the $\mathbb{ASP}-\mathbb{PROLOG}$ system. We will also present a revised operational semantics for $\mathbb{ASP}-\mathbb{PROLOG}$, which builds on the new model-theoretical semantics. We illustrate the flexibility of $\mathbb{ASP}-\mathbb{PROLOG}$ in several applications, mostly drawn from the field of Knowledge Representation and Reasoning.

We will begin with the description of the syntax of $\mathbb{ASP}-\mathbb{PROLOG}$ (Section 2). In Section 3, we present the semantics of $\mathbb{ASP}-\mathbb{PROLOG}$ and a brief description of the $\mathbb{ASP}-\mathbb{PROLOG}$ system (Section 4). We illustrate the use of $\mathbb{ASP}-\mathbb{PROLOG}$ in various applications (Section 5) and conclude in Section 6.

## 2 $\mathbb{ASP}-\mathbb{PROLOG}$: The Syntax

### 2.1 Describing a Knowledge Base

The syntax of $\mathbb{ASP}-\mathbb{PROLOG}$ builds on a signature $\langle \mathcal{F}, \Pi, \mathcal{V} \rangle$, where $\mathcal{F}$ is a countable collection of *function* symbols, $\Pi$ is a countable collection of *predicate* symbols, and $\mathcal{V}$ is a collection of *variables*. We make the following initial assumptions:

- There is an arity function $ar : \mathcal{F} \cup \Pi \rightarrow \mathbb{N}$.
- The set $\Pi$ is partitioned as $\Pi = \Pi_u \cup \Pi_i$; we refer to $p \in \Pi_u$ as a *user-defined* predicate, while $q \in \Pi_i$ is referred to as an *interface* predicate. We assume, for this presentation, that $\{\texttt{assert}, \texttt{retract}, \texttt{model}, \texttt{clause}\} \subseteq \Pi_i$.
- The two symbols $\leftarrow$ and $\wedge$ belong to $\mathcal{F}$, both with arity 2, as well as the symbol *not* with arity 1.
- There is subset $\mathcal{F}_u \subseteq \mathcal{F}$ and a bijection $\zeta : \Pi_u \rightarrow \mathcal{F}_u$.

We assume that the two symbols $\bot, \top$ are present in $\Pi$, both with arity 0; these are used to denote, respectively, *false* and *true*.

The notions of term and atom are defined as usual. A *term* is either a variable from $\mathcal{V}$ or an entity of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms, $f \in \mathcal{F}$ and $ar(f) = n$. An atom is an entity of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms, $p \in \Pi$ and $ar(p) = n$. An atom is a $i$-atom if $p \in \Pi_i$, otherwise it is an $u$-atom. Given an atom $A$, we denote with $pred(A)$ the predicate used in the construction of $A$. We denote with $\mathcal{H}(\mathcal{F})$ the Herbrand universe of $\mathcal{F}$ (i.e., the set of all ground terms built using terms of $\mathcal{F}$) and with $\mathcal{P}(\Pi, \mathcal{F})$ the Herbrand base of $\mathcal{F}$ and $\Pi$ (i.e., all the ground atoms built using predicates of $\Pi$ and terms of $\mathcal{H}(\mathcal{F})$). With a slight abuse of notation, we use the notation $\mathcal{H}(\mathcal{F}, \mathcal{V})$ to denote all the possible terms built with functions symbols from $\mathcal{F}$ and variables from $\mathcal{V}$.

A *rule* is an entity of the form

$$A \,\texttt{:-}\, B_1, \ldots, B_n, \textit{not}\ C_1, \ldots, \textit{not}\ C_m \tag{1}$$

where $A$, $B_1, \ldots, B_n$, and $C_1, \ldots, C_m$ are atoms, and $A$ is an $u$-atom. If $A = \bot$, then we will write the rule simply as

$$\texttt{:-}\, B_1, \ldots, B_n, \textit{not}\ C_1, \ldots, \textit{not}\ C_m$$

Given a rule $r$ of the form (1), we denote with $head(r)$ the atom $A$ and with $body(r)$ the set $\{B_1, \ldots, B_n, \textit{not}\ C_1, \ldots, \textit{not}\ C_m\}$. A *knowledge base* is a collection of rules.

A *substitution* $\sigma$ is a function $\sigma : \mathcal{V} \rightarrow \mathcal{H}(\mathcal{F}, \mathcal{V})$; the substitution is *grounding* if, for each $v \in \mathcal{V}$, we have that $\sigma(v) \in \mathcal{H}(\mathcal{F})$. Given a knowledge base $P$, we denote with $ground(P) = \{\sigma(r) \mid r \in P, \sigma \textit{ is a grounding substitution}\}$.
With a slight abuse of notation, we also:

- Use the function symbols $\leftarrow$ and $\wedge$ in infix notation, i.e., we write $s \leftarrow t$ and $s \wedge t$ instead of $\leftarrow(s,t)$ and $\wedge(s,t)$;
- Use the simplified notation $t_1 \wedge t_2 \wedge \cdots \wedge t_n$ as a syntactic sugar for $t_1 \wedge (t_2 \wedge (\cdots \wedge (t_{n-1} \wedge t_n)))$;
- Extend the function $\zeta$ to a function over $u$-atoms and rules:
  - if $p(t_1, \ldots, t_n)$ is a $u$-atom, then $\zeta(p(t_1, \ldots, t_n)) = \zeta(p)(t_1, \ldots, t_n)$.
  - if the rule $r$ is of the form $A \,\text{:-}\, B_1, \ldots, B_n, \textit{not } C_1, \ldots, \textit{not } C_m$, then $\zeta(r) = \zeta(A) \leftarrow \zeta(B_1) \wedge \cdots \wedge \zeta(B_n) \wedge \textit{not } (\zeta(C_1)) \wedge \cdots \wedge \textit{not } (\zeta(C_m))$.

Given a knowledge base $P$, we denote

$$\textit{def}(P) = \{p \mid p \in \Pi_u, r \in P, pred(head(r)) = p\}$$

## 2.2 Describing a Collection of Knowledge Bases

A *module* is a knowledge base that has the capabilities of interacting with other knowledge bases. In order to make the interaction possible, we need to introduce, first of all, some extensions to the syntax of a knowledge base.

A *qualified atom* is a formula of the form $t : A$ where $A$ is an atom and $t$ is a term. We modify the notion of rule (1) by allowing $B_1, \ldots, B_n$ and $C_1, \ldots, C_m$ to be either atoms or qualified atoms (note that $A$ is still required to be an atom)—we refer to these as *qualified rules*. A qualified knowledge base is a knowledge base composed of qualified rules. We also use the term *literal* to denote either an atom $A$ or the formula $not\ A$ and the term *qualified literal* to denote either a qualified atom $t : A$ or a formula $not\ (t : A)$. A *program literal* is either a literal or a qualified literal. We denote with $\mathcal{PL}(\Pi, \mathcal{F}, \mathcal{V})$ the set of all program literals, and with $\mathcal{PL}(\Pi, \mathcal{F})$ the set of all ground program literals.

Intuitively, the term $t$ in the annotation is used to represent the scope in which the atom will be evaluated—and the scope will be a set of interpretations identified by the term $t$. We denote with $used(\alpha)$ the annotation of $\alpha$, i.e., $used(A) = \emptyset$ for an atom $A$ and $used(t : A) = \{t\}$ for a qualified atom $t : A$. This notion is extended to the case of rules; if $r$ is a rule of the form (1), then $used(r) = \bigcup_{i=1}^{n} used(B_i) \cup \bigcup_{j=1}^{m} used(C_j)$; and for a program $P$, $used(P) = \bigcup_{r \in ground(P)} used(r)$.

A *module* is a tuple $M = \langle name, type, P, I, O \rangle$ where

- $name$ is a ground term, indicating the name of the module (denoted by $name(M)$);
- $type$ is either `prolog` or `asp` (denoted by $type(M)$);
- $P$ is a qualified knowledge base (denoted by $pgm(M)$);
- $I$ is a set of ground terms—referred to as the *import set* (denoted by $import(M)$); we require $I$ to satisfy the following conditions: $used(P) \subseteq I$ and $name \notin I$; and
- $O \subseteq \textit{def}(P)$—referred to as the *export set* (denoted by $export(M)$).

Intuitively, each module has a name, a type (which defines the semantics to be used to interpret the module), a collection of rules defining the "meaning" of the module, an import set—containing the names of other modules from which we import knowledge—and an export set—containing the predicates defined in the module and visible to the outside world.

A *program* is a finite collection $\{M_1, \ldots, M_k\}$ of modules, with the following constraints

- For each $i = 1, \ldots, k$, if $t \in import(M_i)$, then there exists $M_j$ such that $name(M_j) = t$;
- For each $t : A$ occurring in a rule of $ground(M_i)$, if $name(M_j) = t$ then $pred(A) \in export(M_j)$.

For the sake of simplicity, we often denote the module $M$ with $M_{name(M)}$. Additionally, given a program $P$, we denote with $name(P) = \{name(M) \mid M \in P\}$. We also extend the function $\zeta$ to operate with qualified atoms and qualified rules in the obvious manner, starting with the assumption that $\zeta(t : A) = t : \zeta(A)$.

### 2.3 Additional Considerations

The interface predicates in $\Pi_i$ are meant to provide built-in mechanisms to enable meaningful interaction between modules. We distinguish two types of interface predicates:

- *Querying:* these are predicates that are used to access the semantics of other modules;
- *Updating:* these are predicates used to modify the content of other modules.

In the rest of the discussion, we classify the predicates `model` and `clause` as static, while `assert` and `retract` are classified as dynamic.

Whenever a module contains occurrences of dynamic predicates, we refer to it as a *dynamic module*, otherwise we refer to it as a *static module*. A program is static if it contains exclusively static modules, otherwise it is considered to be dynamic.

The distinction is important, as it affects the way the semantics of programs is developed. A static program can be modeled model-theoretically, as a collection of models. On the other hand, in presence of dynamic predicates, we wish to model the dynamic evolution of the modules—i.e., we intend to model the semantics of the program $P$ as an evolving sequence of programs $P_1, P_2, \ldots$ each with its own semantics.

## 3 $\mathbb{ASP}-\mathbb{PROLOG}$: The Semantics

### 3.1 Preliminary Definitions

Let us consider a program $P$. The import/export declarations present in the various modules establish a dependence relation among modules. We say that $M_i$ depends on $M_j$ (denoted $M_j \prec M_i$) if $name(M_j) \in import(M_i)$. The dependence relation can be also seen in the form of a *dependence graph* where the set of nodes is

$N = \{name(M) \mid M \in P\}$ and the set of edges is $E = \{(name(M_i), name(M_j)) \mid M_i, M_j \in P, M_j \prec M_i\}$. We denote with $graph(P)$ the graph $(N, E)$ generated by the modules in the program $P$. We also denote with $\prec^*$ the transitive closure of $\prec$.

For the first part of our discussion, we impose some restrictions on the admissible structures for $graph(P)$:

- $graph(P)$ is acyclic;
- There exists a topological sort $n_1, n_2, \ldots, n_k$ of $graph(P)$ such that
  - ◦ If $M_{n_i} \prec M_{n_j}$ then $i < j$;
  - ◦ For each $1 \leq i < k$ we have that $M_{n_i} \prec^* M_{n_k}$.
- predicates from $\Pi_i$ are used exclusively in qualified atoms.

We refer to the module $M_{n_k}$ as the entry module, and we denote it with *entry*$(P)$. For the sake of simplicity we also consider only modules that export all their predicates.

In the rest of the discussion, we rely on the fact that each module should be interpreted according to a possibly different semantics. We also assume that the semantics of each module can be described by a collection of models, in turn described by sets of atoms. Given a knowledge base $P$ (without qualifications) of type $\kappa$, we denote with $\mathcal{SEM}^{\kappa}(P) \subseteq 2^{\mathcal{P}(\Pi,\mathcal{F})}$ the $\kappa$-semantics of $P$, described as sets of atoms. For example, if $\kappa$ is `prolog`, then $\mathcal{SEM}^{\texttt{prolog}}(P)$ includes a single element, which corresponds to the least Herbrand model of $P$,[1] while if $\kappa$ is `asp`, then $\mathcal{SEM}^{\texttt{asp}}(P)$ contains the answer sets of $P$ [Gelfond and Lifschitz, 1988]. Whenever we refer to a module $M$, we often use the notation $\mathcal{SEM}(M)$ to simply denote $\mathcal{SEM}^{type(M)}(pgm(M))$.

## 3.2 Model-Theoretic Semantics

In order to develop a model-theoretic semantics for $\mathbb{ASP}-\mathbb{PROLOG}$, we will restrict our attention to static programs. The intuition behind the construction of the semantics of a program $P$ is as follows:

- ◦ The semantics is described in terms of assignments of a collection of models to each module of the program;
- ◦ The construction follows the topological sort of $graph(P)$—i.e., if $n_1, n_2, \ldots, n_k$ is the sorted list of modules, then the semantics assigned to $n_i$ will build on the semantics assigned to the modules $n_1, \ldots, n_{i-1}$. Intuitively, the semantics of $n_i$ will be constructed by constructing a "reduced" version of $M_{n_i}$, where each qualified atom $t : A$ is interpreted according to the semantics assigned to the module $M_t$ (and accordingly removed from the module), and then using $\mathcal{SEM}$ to determine the semantics of the reduced version of the module.

The starting point of the construction is the introduction of a *model naming function*—i.e., a function which maps ground terms to interpretations. The need for model naming comes from the need to interpret qualified atoms of the form $t : A$, where $t$ is treated as the name of a model of a different module. Formally, a model naming function is a function $\tau : \mathcal{H}(\mathcal{F}) \rightarrow 2^{\mathcal{P}(\Pi,\mathcal{F})}$ which maps ground terms to possible interpretations.

[1] We assume Prolog modules to contain definite programs and admit a least Herbrand model.

Let us consider a program $P$ with module names $n_1, n_2, \ldots, n_k$ (already topologically sorted according to $graph(P)$). An $\mathbb{ASP}-\mathbb{PROLOG}$-*interpretation* of $P$ is a pair $I = (\tau, \pi)$ where $\tau$ is a model-naming function and $\pi$ is a function $\pi : \mathit{name}(P) \rightarrow 2^{2^{\mathcal{P}(\Pi,\mathcal{F})}}$. Intuitively, for each module name $n_j$, $\pi(n_j)$ is a set of interpretations to be used with the module $M_{n_j}$.

Given an $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I = (\tau, \pi)$, we define the notion of entailment of qualified atoms w.r.t. $I$ and the module named $n_i$ in $P$ (denoted by $\models^P_{n_i}$) as follows:

- If $t : A$ is a ground qualified atom and $t \in \mathit{import}(M_{n_i})$ then $I \models^P_{n_i} t : A$ iff $\forall X \in \pi(t)$ we have that $A \in X$ (i.e., $A$ is skeptically entailed by all the models of the module named $t$).
- If $t : A$ is a ground qualified atom and $t \in \mathit{import}(M_{n_i})$, then $I \models^P_{n_i} \mathit{not}\, t : A$ iff $\exists X \in \pi(t)$ such that $A \notin X$.
- If $t : A$ is a ground qualified atom and $t \notin \mathit{import}(M_{n_i})$, then $I \models^P_{n_i} t : A$ iff $\exists s \in \mathit{import}(M_{n_i})$ such that $\tau(t) \in \pi(s)$ and $A \in \tau(t)$—i.e., $t$ is the name of an interpretation which entails $A$ and it is one of the models of one of the imported modules (i.e., $A$ is credulously entailed by one of the imported modules).
- If $t : A$ is a ground qualified atom and $t \notin \mathit{import}(M_{n_i})$, then $I \models^P_{n_i} \mathit{not}\, t : A$ iff either $\forall s \in \mathit{import}(M_{n_i})$ we have that $\tau(t) \notin \pi(s)$ or $\exists s \in \mathit{import}(M_{n_i})$ such that $\tau(t) \in \pi(s)$ and $A \notin \tau(t)$.

The interface predicates `model` and `clause` are interpreted as follows:

- Given a ground atom of the form $t : \texttt{model}(s)$, we have that $I \models^P_{n_i} t : \texttt{model}(s)$ iff $t \in \mathit{import}(M_{n_i})$ and $\tau(s) \in \pi(t)$—i.e., $s$ is a name associated to an interpretation that is a valid model of the imported module $s$.
- Given a ground atom of the form $t : \texttt{clause}(r)$, we have that $I \models^P_{n_i} t : \texttt{clause}(r)$ iff $t \in \mathit{import}(M_{n_i})$ and $\zeta^{-1}(r) \in \mathit{pgm}(M_t)$.

Given a program $P$, a module name $n_i$ and an $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I$, we define a notion of reduct used to eliminate from a module all qualified entities—and using the interpretation $I$ to determine whether the qualified items are true or false. Formally:

- If $A$ is a ground atom, then $A^I_{n_i} = A$ and $(\mathit{not}\ A)^I_{n_i} = \mathit{not}\ A$—i.e., unqualified items are left unchanged in the reduction process;
- $(t : A)^I_{n_i} = \mathit{true}$ (resp. $(t : A)^I_{n_i} = \mathit{false}$) iff $I \models^P_{n_i} t : A$ (resp. $I \not\models^P_{n_i} t : A$);
- $(\mathit{not}\ (t : A))^I_{n_i} = \mathit{true}$ (resp. $(\mathit{not}\ (t : A))^I_{n_i} = \mathit{false}$) iff $I \models^P_{n_i} \mathit{not}\ (t : A)$ (resp. $I \not\models^P_{n_i} \mathit{not}\ (t : A)$).

Given a generic qualified rule $r \in \mathit{ground}(P)$ of the form $A \,\text{:-}\, B_1, \ldots, B_k$, a program $P$, a module name $n_i$ in $P$, and an $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I$, we denote with $r^I_{n_i}$ the $\mathbb{ASP}-\mathbb{PROLOG}$-reduct of $r$ w.r.t. $I$ and $n_i$, defined as

$$r^I_{n_i} = \begin{cases} \emptyset & \textit{if there exists } 1 \leq j \leq k \textit{ such that } (B_j)^I_{n_i} = \mathit{false} \\ \{A \,\text{:-}\, (B_1)^I_{n_i}, \ldots, (B_k)^I_{n_i}\} & \textit{otherwise} \end{cases}$$

The entailment relation and the topological sort of the modules in $graph(P)$ allow us to introduce the notion of $\mathbb{ASP}-\mathbb{PROLOG}$-reduct of a program. Given a program $P = \{M_{n_1}, \ldots, M_{n_k}\}$ (where $n_1, n_2, \ldots, n_k$ is the topological sort of $graph(P)$) and an $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I$, the $\mathbb{ASP}-\mathbb{PROLOG}$-reduct of $ground(P)$ w.r.t. $I$, denoted $ground(P)^I$, is the program $\{M^I_{n_1}, M^I_{n_2}, \ldots, M^I_{n_k}\}$, where $M^I_{n_i}$ is a module identical to $M_{n_i}$ except for the following:

$$pgm(M^I_{n_i}) = \bigcup_{r \in ground(pgm(M_{n_i}))} r^I_{n_i}.$$

We can now define the intended meaning of a program $P$: an $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I = (\tau, \pi)$ is an intended meaning for $P$ if for each module $n$ in $P$ we have that $\mathcal{SEM}^{type(M_n)}(ground(pgm(M^I_n))) = \pi(n)$.

**Proposition 1.** *Given a $\mathbb{ASP}-\mathbb{PROLOG}$ program $P$ and a model naming function $\tau$, there is a unique function $\pi$ such that $I = (\tau, \pi)$ is an intended meaning for $P$.*

**Proposition 2.** *Let $P$ be a program, $\tau$ a model naming function, and $I = (\tau, \pi)$ the corresponding intended meaning for $P$. If $M \in P$ is a module such that for each other module $M' \in P$ we have that $M' \not\prec M$, then $\mathcal{SEM}^{type(M)}(ground(pgm(M))) = \pi(name(M))$.*

*Example 1.* Let us consider a two-module program; the first module is of type `asp`, it is named $m_1$, and it contains the rules

$$p :- not\, q \qquad q :- not\, p$$

The second module, named $m_2$, is of type `prolog`, and it includes the rules

$$r :- m_1 : p \qquad s :- t_1 : model(m_1), t_1 : q \qquad s :- t_2 : model(m_1), t_2 : q$$

Let us consider a model naming function $\tau$ where

$$\tau(t_1) = \{p\} \qquad \tau(t_2) = \{q\} \qquad \tau(t_3) = \{s\}$$

It is easy to see that any $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I$ which is a model of the program should have $\pi(m_1) = \{\{p\}, \{q\}\}$. This guides the reduct of $m_2$ to become

$$s :-$$

which can be satisfied by requiring $\pi(m_2) = \{\{s\}\}$.

## 3.3 Semantics for Dynamic Programs

In presence of dynamic programs, we propose a goal-oriented operational semantics—since the dynamic changes provided by predicates like `assert` and `retract` are sensitive to the order of execution. For the sake of simplicity, in the rest of the discussion we will concentrate exclusively on programs composed of two types of modules—`prolog` and `asp`. We assume that the natural semantics $\mathcal{SEM}$ for `prolog` is the least Herbrand model, while for `asp` $\mathcal{SEM}$ corresponds to the answer set semantics. Furthermore, we assume that dynamic interface predicates are used exclusively in qualified atoms in `prolog` programs, and for each program $P$ we have that $head(P)$ is a `prolog` module. Since the modifications of the program are based on adding or removing rules from programs, they will not affect the structure of $graph(P)$. Thus, we will work with different versions of a program with respect to a fixed $graph$ structure and with the same topological sort $n_1, n_2, \dots, n_k$ of its modules. Formally:

- Two modules $M$ and $M'$ are variants (denoted $M \asymp M'$) if *name*$(M) =$ *name*$(M')$, *type*$(M) =$ *type*$(M')$, *import*$(M) =$ *import*$(M')$, *export*$(M) =$ *export*$(M')$.
- Two programs $P_1 = \{M_1^1, \dots, M_k^1\}$ and $P_2 = \{M_1^2, \dots, M_k^2\}$ are variants ($P_1 \asymp P_2$) if $M_1^1 \asymp M_1^2, \dots, M_k^1 \asymp M_k^2$.

Given a program $P$, we refer to $\mathcal{PR}(P) = \{P' \mid P' \asymp P\}$ to denote the set of all possible variants of $P$. For each $P' \in \mathcal{PR}(P)$, we denote with $P'(n_i)$ the module named $n_i$ in $P'$.

Given two $\mathbb{ASP}-\mathbb{PROLOG}$ programs $P_1 \asymp P_2$, we say that $P_1$ and $P_2$ are $i$-compatible ($i \leq k$) if $P_1(n_j) = P_2(n_j)$ for $j \geq i$.

In presence of dynamic operations, we need to associate the elements of an interpretation with the modified program that supports such element. A *dynamic atom* is a tuple $(P_1, A, P_2)$ where $A$ is a ground atom and $P_1, P_2$ are $\mathbb{ASP}-\mathbb{PROLOG}$ programs such that $P_1, P_2 \in \mathcal{PR}(P)$. Intuitively, a dynamic atom $(P_1, A, P_2)$ indicates that the truth of $A$ in $P_1$ is possible only if the program $P_1$ is transformed into the program $P_2$ as result of the `asserts` and `retracts` involved in the rules related to $A$; alternatively, one could say that $A$ is true in $P_1$, but its truth requires transforming $P_1$ to $P_2$.

A dynamic interpretation is a collection of dynamic atoms. The notion of interpretation in $\mathbb{ASP}-\mathbb{PROLOG}$ can be modified to become a dynamic $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $(\tau, \pi)$, where $\tau$ is a model naming function, while $\pi(n)$ will correspond to a set of dynamic interpretations for the module $n$.

For the sake of simplicity the successive discussion assumes that $P$ is a ground program. We will successively discuss the lifting to the non-ground case.

**Dynamic Natural Semantics.** Let us start by modifying the notion of standard semantics of a knowledge base $M$ (without qualifications) to the case of dynamic interpretations. The intuition is to transform the programs so that they deal with dynamic atoms and are interpreted using dynamic interpretations.

Let us start with some preliminary notation. If $r$ is a rule of the form

$$A \,\text{:-}\, B_1, \dots, B_h, not\ B_{n+1}, \dots, not\ B_m$$

(with $m \geq 1$) in $P_1(name(M))$ then $\hat{r}(P_1, m)$ contains all the rules of the form

$$(P_1, A, P_{m+1}) \,\text{:-}\, (P_{i_1}, B_1, P_{i_1+1}), (P_{i_2}, B_2, P_{i_2+1}), \dots, not\ \ (P_{i_m}, B_m, P_{i_m+1})$$

where $\langle i_1, \dots, i_m \rangle$ is a permutation of $\langle 1, \dots, m \rangle$, $P_1, \dots, P_{m+1} \in \mathcal{PR}(P)$, and for each $1 \leq j \leq m$, if $B_j$ is not a qualified atom, then $P_{i_j} = P_{i_j+1}$. The intuition is that the order in which the elements in the body of a rule are coinsidered is meaningful, thus we generate rules that can account for any order (the permutation $\langle i_1, \dots, i_m \rangle$). The connection between the different variants of the program allows to capture the successive changes to the program that may be required in asserting the truth of each element in the body of the program.

If $r$ is a fact $A$ in $P_1(name(M))$ (i.e., a rule with an empty body), then $\hat{r}(P_1, m) = \{(P_1, A, P_1)\}$.

We also introduce the notation $\widehat{r}(m)$ to contain the union of all the $\widehat{r}(P_1, m)$ for each $P_1$ such that $P_1(m)$ contains the rule $r$.

The semantics in absence of qualifications and imports for a module $M$ (named $m$) is as follows: for each $P_1 \in \mathcal{PR}(P)$, if $S \in \mathcal{SEM}(P_1(m))$ then

$$\{(P_1, a, P_1) \mid a \in S\} \in \mathcal{SEM}_d(m)$$

and no other models are considered.

**Dynamic Reduct and Dynamic Semantics.** In order to expand the definitions of Section 3.2, we now need to extend the notion of entailment to the case of dynamic interpretations and dynamic programs. In the following discussion, $n$ denotes the name of a module in the program $P$, and by $P(n)$ we denote the module $M \in P$ such that *name*$(M) = n$. The entailment is equivalent, for the most part, to the case of non-dynamic programs, except for the use of dynamic atoms. The novelty is in the ability to process `assert` and `retract` statements. Given $I = (\tau, \pi)$ a dynamic $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation, and a program $P$, the entailment of qualified ground atoms is defined as follows (in all the following cases, $P_1, P_2 \in \mathcal{PR}(P)$):

- If $t : A$ is a ground qualified atom and $t \in$ *import*$(P(n))$ then $I \models_n (P_1, t : A, P_2)$ iff for each $X \in \pi(t)$ we have that $(P_1, A, P_2) \in X$.
- If $t : A$ is a ground qualified atom and $t \in$ *import*$(P(n))$ then $I \models_n$ *not* $(P_1, t : A, P_2)$ iff there exists $X \in \pi(t)$ such that $(P_1, A, P_2) \notin X$.
- If $t : A$ is a ground qualified atom and $t \notin$ *import*$(P(n))$ then $I \models_n (P_1, t : A, P_2)$ iff $\exists s \in$ *import*$(P(n))$ such that $\tau(t) \in \pi(s)$ and $(P_1, A, P_2) \in \tau(t)$.
- If $t : A$ is a ground qualified atom and $t \notin$ *import*$(P(n))$ then $I \models_n$ *not* $(P_1, t : A, P_2)$ iff either $\forall s \in$ *import*$(P(n))$ we have that $\tau(t) \notin \pi(s)$ or there exists $s \in$ *import*$(P(n))$ such that $\tau(t) \in \pi(s)$ but $(P_1, A, P_2) \notin \tau(t)$.
- If $t : \texttt{model}(s)$ is a ground qualified atom, then $I \models_n (P_1, t : \texttt{model}(s), P_2)$ iff $t \in$ *import*$(P(n))$, $\tau(s) \in \pi(t)$, and $P_1 = P_2$.
- If $t : \texttt{clause}(r)$ is a ground qualified atom, then $I \models_n (P_1, t : \texttt{clause}(r), P_2)$ iff $t \in$ *import*$(P(n))$, $\zeta^{-1}(r) \in$ *pgm*$(P_1(n))$, and $P_1 = P_2$.
- If $t : \texttt{assert}(r)$ is a ground qualified atom, then $I \models_n (P_1, t : \texttt{assert}(r), P_2)$ iff $t \in$ *import*$(P(n))$, $M'$ is a module such that $M' \asymp P_1(t)$,
*pgm*$(M') =$ *pgm*$(P_1(t)) \cup \{\zeta^{-1}(r)\}$, and $P_2 = (P_1 \setminus \{P_1(t)\}) \cup \{M'\}$. Intuitively, $P_2$ should be a program obtained by introducing the rule $r$ in the module $t$ of program $P_1$.
- If $t : \texttt{retract}(r)$ is a ground qualified atom, then $I \models_n (P_1, t : \texttt{retract}(r), P_2)$ iff $t \in$ *import*$(P(n))$, $M'$ is a module such that $M' \asymp P_1(t)$,
*pgm*$(M') =$ *pgm*$(P_1(t)) \setminus \{\zeta^{-1}(r)\}$, and $P_2 = (P_1 \setminus \{P_1(t)\}) \cup \{M'\}$. Intuitively, $P_2$ should be a program obtained by removing the rule $r$ from the module $t$ of program $P_1$.

Given a ground program $P = \{M_{n_1}, \ldots, M_{n_k}\}$ and a dynamic $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I$, it is now possible to determine the $\mathbb{ASP}-\mathbb{PROLOG}$-reduct of $P$ as the program $\{(\widehat{M_{n_1}})^I, \ldots, (\widehat{M_{n_k}})^I\}$, and proceed in defining the notion of intended meaning in the same way as done for the static case. In particular,

- For each dyanmic literal or qualified literal $\ell$, we define $\ell^I_n$ as follows:
  - For each ground dynamic atom $\ell \equiv (P_1, A, P_2)$, $\ell^I_n = \ell$;
  - For each ground dynamic literal $\ell \equiv \mathit{not}\ (P_1, A, P_2)$, $\ell^I_n = \ell$;
  - For each ground qualified dynamic atom $\ell \equiv (P_1, t : A, P_2)$, we have that $\ell^I_n =$ *true* (resp. $\ell^I_n =$ *false*) iff $I \models_n \ell$ (resp. $I \not\models_n \ell$);
  - For each ground qualified literal $\ell = \mathit{not}(P_1, t : A, P_2)$, we have that $\ell^I_n =$ *true* (resp. $\ell^I_n =$ *false*) iff $I \models_n \ell$ (resp. $I \not\models \ell$).
- Let $r$ be a rule in $P_1(n)$ and let us consider an element $\rho$ of the form $A\,\text{:-}\,B_1, \ldots, B_k$ in $\hat{r}(P_1, n)$; $\rho^I_n$ is the rule

$$\rho^I_n = \begin{cases} \emptyset & \textit{there is a qualified literal } B \textit{ in body}(r) \textit{ such that } B^I_n = \textit{false} \\ \{A\,\text{:-}\,(B_1)^I_n, \ldots, (B_k)^I_n\} & \textit{otherwise} \end{cases}$$

- for a module $\widehat{M_m}$ containing imports, and given a dynamic interpretation $I$, we denote with $\widehat{M_m}^I$ the module such that $\widehat{M_m}^I \asymp M_m$ and

$$\mathit{pgm}(\widehat{M_m}^I) = \bigcup_{\substack{r \in \mathit{pgm}(P_1(m)) \\ \rho \in \hat{r}(P_1, n)}} \rho^I_m$$

- Finally, we can define a dynamic $\mathbb{ASP}-\mathbb{PROLOG}$-interpretation $I = (\tau, \pi)$ to be an intended meaning of $P$ if for each module $n$ in $P$ we have that $\mathcal{SEM}(\widehat{M_n}^I) = \pi(n)$.

*Example 2.* Let us consider a program $P$ composed of a module $t_1$ containing the rules $p\,\text{:-}\,q, r$ and $q\,\text{:-}$ and the module $t_2$ containing the rule $s\,\text{:-}\,t_1 : \texttt{assert}(r), t_1 : q$.

Let us consider the variant $P_1$ of the program where the only change is the addition of the fact $r\,\text{:-}$ to the module $t_1$, then we have that $\widehat{M_{t_1}}$ will include only the following two rules with head atoms of the type $(P_1, \cdot, \cdot)$:

$$(P_1, p, P_1)\,\text{:-}\,(P_1, q, P_1), (P_1, r, P_1) \qquad (P_1, q, P_1)\,\text{:-} \qquad (P_1, r, P_1)\,\text{:-}$$

Thus, the dynamic semantics will ensure that each dynamic model of this module will include the dynamic atom $(P_1, r, P_1)$. The dynamic reduct of the rules of the module $t_2$ for $P$ include the fact $(P, s, P_1)\,\text{:-}$. Thus, the dynamic models for the module $t_2$ will include $(P, s, P_1)$.

**Operational Semantics for Dynamic Programs.** The top-down nature of the computation in the case of dynamic programs can be captured through the notion of derivation. This provides us also with a mechanism to lift the previous definitions to the non-ground case. Let us start with some preliminary definitions. Let us refer to a *goal* as a sequence of program literals. A *computation rule*

$$\rho : \mathcal{PL}(\Pi, \mathcal{F}, \mathcal{V})^* \rightarrow \mathcal{PL}(\Pi, \mathcal{F}, \mathcal{V})^* \times \mathcal{PL}(\Pi, \mathcal{F}, \mathcal{V}) \times \mathcal{PL}(\Pi, \mathcal{F}, \mathcal{V})^*$$

is a function that, given a sequence of program literals, it selects a particular program literal, splitting the sequence in two parts, the part preceding the selected literal and the part following it.

A *computation state* is a tuple $\sigma = \langle \alpha, \theta, P \rangle$ where $\alpha$ is a goal (referred to a *goal part* and denoted by *goal*$(\sigma)$), $\theta$ is a substitution for the variables $\mathcal{V}$ (denoted by *subs*$(\sigma)$) and $P$ is an $\mathbb{ASP}-\mathbb{PROLOG}$ program (denoted by *program*$(\sigma)$).

Given a goal $\alpha$ and an $\mathbb{ASP}-\mathbb{PROLOG}$ program $P$, the *initial computation state* w.r.t. $\alpha$ and $P$ is the computation state $\sigma_0(G) = \langle \alpha, \epsilon, P \rangle$.[2]

We introduce a resolution relation between computation states, denoted by $\leadsto_\rho^{n,I}$, where $\rho$ is a computation rule, $I$ is an intended meaning of the program, and $n$ is the name of a `prolog` module. We will also denote with $\overset{*}{\leadsto}_\rho^{n,I}$ the reflexive and transitive closure of the relation $\leadsto_\rho^{n,I}$. We have that $\langle \alpha, \theta, P_1 \rangle \leadsto_\rho^{n,I} \langle \alpha', \theta', P' \rangle$ if $\rho(\alpha) = (\alpha_1, \gamma, \alpha_2)$ and:

- If $\gamma$ is an atom $A$ and $r$ is a new variant of a rule in *pgm*$(P(n))$ such that $head(r)$ unifies with $A\theta$ then[3]
  - $\alpha' = \alpha_1 \cdot body(r) \cdot \alpha_2$;
  - $\theta' = \theta \circ \mu$ where $\mu$ is the most general unifier of $A\theta$ and $head(r)$;
  - $P' = P_1$.
- If $\gamma$ is a qualified atom $t : A$, there exists a substitution $\psi$ such that $t(\theta \circ \psi) \in$ *import*$(P(n))$, *type*$(P(t)) =$ `prolog`, and $\langle A, \theta \circ \psi, P_1 \rangle \overset{*}{\leadsto}_\rho^{t(\theta\circ\psi),I} (\Box, \sigma, P')$ then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \sigma$
- If $\gamma$ is a qualified atom $t : A$, there exists a substitution $\psi$ such that $t(\theta \circ \psi) \in$ *import*$(P(n))$, *type*$(P(t)) =$ `asp`, $A(\theta \circ \psi)$ is ground and there is a program $P' \in \mathcal{PR}(P)$ such that $(P_1, A(\theta \circ \psi), P') \in X$ for each $X \in \pi(t(\theta \circ \psi))$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
- If $\gamma$ is a qualified atom $t : A$, there is a substitution $\psi$ such that $t(\theta \circ \psi) \not\in$ *import*$(P(n))$, there is $s \in$ *import*$(P(n))$ such that $\tau(t(\theta \circ \psi)) \in \pi(s)$, and there is a program $P' \in \mathcal{PR}(P)$ such that $(P_1, A(\theta \circ \psi), P') \in \tau(t(\theta \circ \psi))$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
- If $\gamma$ is of the form *not* $t : A$, there is a substitution $\psi$ such that $t(\theta \circ \psi) \in$ *import*$(P(n))$, $X \in \pi(t(\theta \circ \psi))$, for all ground instances $A(\theta \circ \psi \circ \sigma)$ and for all $P' \in \mathcal{PR}(P)$ we have that $(P_1, A(\theta \circ \psi \circ \sigma), P') \not\in X$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
  - $P' = P_1$
- If $\gamma$ is of the form *not* $t : A$, there is a substitution $\psi$ such that $t(\theta \circ \psi) \not\in$ *import*$(P(n))$, there is $s \in$ *import*$(P(n))$ such that $\tau(t(\theta \circ \psi)) \in \pi(s)$, and for each ground instance $A(\theta \circ \psi \circ \sigma)$ and for all $P' \in \mathcal{PR}(P)$ we have that $(P_1, A(\theta \circ \psi \circ \sigma), P') \not\in \tau(t(\theta \circ \psi))$, then

[2] We denote with $\epsilon$ the identity substitution, i.e., the substitution such that for each $X \in \mathcal{V}$ we have that $\epsilon(X) = X$.

[3] $\cdot$ represents concatenation of lists and $\circ$ represents function composition.

  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
  - $P' = P_1$
- If $\gamma$ is of the form $t$ : `model`$(s)$, there is a substitution $\psi$ such that $t(\theta \circ \psi) \in$ *import*$(P(n))$, and $\tau(s(\theta \circ \psi)) \in \pi(t(\theta \circ \psi))$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
  - $P' = P_1$
- If $\gamma$ is of the form $t$ : `clause`$(r)$, there is a substitution $\psi$ such that $t(\theta \circ \psi) \in$ *import*$(P(n))$, $\zeta^{-1}(r(\theta \circ \psi)) \in$ *pgm*$(P_1(t(\theta \circ \psi)))$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
  - $P' = P_1$
- If $\gamma$ is of the form $t$ : `assert`$(r)$, there is a substitution $\psi$ such that $\zeta^{-1}(r(\theta \circ \psi))$ is a well-formed rule, $t(\theta \circ \psi) \in$ *import*$(P(n))$, $M'$ is a module such that $M' \asymp P_1(t(\theta \circ \psi))$ and *pgm*$(M') =$ *pgm*$(P_1(t(\theta \circ \psi))) \cup \{\zeta^{-1}(r(\theta \circ \psi))\}$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
  - $P' = (P_1 \setminus P_1(t(\theta \circ \psi))) \cup \{M'\}$
- If $\gamma$ is of the form $t$ : `retract`$(r)$, there is a substitution $\psi$ such that $t(\theta \circ \psi) \in$ *import*$(P(n))$, $\zeta^{-1}(r(\theta \circ \psi)) \in$ *pgm*$(P_1(t(\theta \circ \psi)))$, $M'$ is a module such that $M' \asymp P_1(t(\theta \circ \psi))$ and *pgm*$(M') =$ *pgm*$(P_1(t(\theta \circ \psi))) \setminus \{\zeta^{-1}(r(\theta \circ \psi))\}$, then
  - $\alpha' = \alpha_1 \cdot \alpha_2$;
  - $\theta' = \theta \circ \psi$
  - $P' = (P_1 \setminus P_1(t(\theta \circ \psi))) \cup \{M'\}$

The notion of computed answer can be given as follows: for a goal $\alpha$, a computed answer substitution $\theta$ is such that there exists a derivation $\langle \alpha, \epsilon, P \rangle \rightsquigarrow_{\rho}^{n_k *} \langle \Box, \theta, P' \rangle$.

## 4 $\mathbb{ASP}-\mathbb{PROLOG}$: An Overview of the Implementation

A first implementation of $\mathbb{ASP}-\mathbb{PROLOG}$ was developed in 2004; it built on the CIAO Prolog system and on a modified version of the `lparse`/`smodels` systems. A new implementation is currently in progress, which advances the state-of-the-art w.r.t. the original implementation in the following directions:

- The new implementation builds on the new semantics presented in this paper;
- The implementation presented in [El-Khatib et al., 2004, El-Khatib et al., 2005] builds on a relatively less popular Prolog system and requires modified versions of ASP solvers (`lparse`, `smodels`); this prevents the evolution of the system and the use of state-of-the-art ASP solvers. The new implementation makes use of ASP solvers as black boxes, enabling the use of the most efficient solvers available. Additionally, the new implementation builds on a more standard and popular Prolog system, i.e., SICStus Prolog.

A new implementation is currently under development. The new implementation aims at more precisely implementing the formal semantics illustrated in the previous section, which provides the foundation of the ASP−PROLOG system. The current prototype provides only a minimal set of predicates aimed at providing the basic interaction between Prolog modules and ASP modules and among ASP modules. It has the following predicates:

– `use_asp(ASPModule, PModule, Parameters)`: a Prolog module, named `PModule`, is created as an interface storing the results of the computation of answer sets of the ASP program `ASPModule` with the parameters specified in `Parameters`. The new module contains the atoms entailed by all the answer sets of `ASPModule`, atoms of the form `model/1`, and has sub-modules which encode the answer sets of `ASPModule`. Two special cases:
  - `use_asp(ASPModule,PModule)`: this has the same effect as `use_asp(ASPModule,PModule,[])`.
  - `use_asp(ASPModule)`: same as `use_asp(ASPModule,ASPModule)`.

  This predicate provide the analogous of the import declaration discussed in the formal syntax; the implementation extends it by allowing its use as a *dynamic predicate*, i.e., allowing the dynamic import of ASP modules at any point during the execution.[4]
– `import(PModule,PredsIn, ASPModule, PredsOut, NewPModule, Parameters)`: a Prolog module named `NewPModule` is created from the ASP program `ASPModule` by (*i*) importing from the Prolog module `PModule` the atoms specified in `PredsIn`, a list of pairs of the form $(f(\boldsymbol{X}), g(\boldsymbol{Y}))$ into `ASPModule` where the set of free variables in $\boldsymbol{Y}$ is a subset of the set of free variables in $\boldsymbol{X}$; (*ii*) computing the answer sets of the new program with the parameters `Parameters`; and (*iii*) exporting the results according to `PredsOut` to the new Prolog module `NewPModule`.
– `assertnb(ASPModule, Progs)` and `retractnb(ASPModule, Progs)`: (*i*) assert and retract, respectively, the clauses specified in `Progs` to the `ASPModule` and (*ii*) create new modules similar to `use_module(ASPModule)`. The "nb" in the names of these predicates indicate that their effects are "non-backtrackable", i.e., they persist as Prolog backtracks over their executions.

## 5 ASP−PROLOG: An Illustration

In this section, we present a set of sample applications exhibiting the main features of ASP−PROLOG.

### 5.1 Reasoning with Answer Set Semantics

The authors of [Gelfond and Leone, 2002] stipulate two different modes of reasoning with the answer set semantics: the skeptical and the credulous reasoning mode. An

[4] It is a simple, though tedious, exercise to extend the semantics presented earlier to cover this dynamic import behavior.

atom $a$ is skeptically entailed by a program $P$ if $a$ belongs to every answer set of $P$. It is credulously entailed if it belongs to at least one answer set of $P$. Both reasoning modes can be used in $\mathbb{ASP}-\mathbb{PROLOG}$, as shown in the next example.

*Example 3.* Suppose that we have the ASP program `t1.lp` which contains the following clauses:

$$\{p \leftarrow \ not\, q. \qquad q \leftarrow \ not\, p. \qquad c \leftarrow p. \qquad c \leftarrow q.\}$$

After the execution of `?- use_asp(t1).` in $\mathbb{ASP}-\mathbb{PROLOG}$, the following queries and answers can be obtained:

| Query | Answer | Reason |
|---|---|---|
| `?- t1:p.` | `No` | $p$ is not true in all answer sets of `t1.lp` |
| `?- t1:c.` | `Yes` | $c$ is true in all answer sets of `t1.lp` |
| `?- t1:model(Q).` | `Q = t11` | |
| | `Q = t12` | if ask for the second answer |
| `?- t1:model(Q),Q:p.` | `Q = t11` or | since $p$ is contained in one answer set |
| | `Q = t12` | |

Observe that the query `?- m1:q, m2:model(Q), Q:p` represents a combination of the two reasoning modes: skeptical reasoning about $q$ in module `m1` and credulous reasoning about $p$ in module `m2`. Such ability might become important when one wants to combine several knowledge bases whose semantics might be different (e.g., as discussed in [Brewka and Eiter, 2007]). The terms $t11$ and $t12$ are automatically generated names for the answer sets of `t1`.

## 5.2 Commonsense Reasoning

Logic programs with answer set semantics have been used extensively in knowledge representation and reasoning (see, e.g., [Baral and Gelfond, 1994] for a survey). The following example illustrates the use of $\mathbb{ASP}-\mathbb{PROLOG}$ in this area.

*Example 4.* Suppose that we have the ASP program `bird.lp`

$$bird(tweety). \qquad bird(sam). \qquad bird(fred). \qquad bird(X) \leftarrow penguin(X).$$
$$fly(X) \leftarrow bird(X),\ not\, ab\_f(X). \qquad ab\_f(X) \leftarrow penguin(X).$$

After the execution of `?- use_asp(bird)` in $\mathbb{ASP}-\mathbb{PROLOG}$, the following queries and answers can be obtained:

`?- bird:bird(X).` As expected, the system will respond with `X = tweety`, `X = sam`, and `X = fred`.
`?- bird:fly(X).` Again, we will receive three answers, as none of the birds is declared as a penguin in the program.
`?- bird:penguin(X).` This will result in the answer `No`, as expected.

The execution of `?- assertnb(bird, [penguin(tweety)]).` results in the following changes of the three queries as follows:

`?- bird:bird(X).` Answer: `X = tweety`, `X = sam`, and `X = fred`.
`?- bird:fly(X).` Answer: `X = sam`, and `X = fred`.
`?- bird:penguin(X).` Answer: `X = tweety`.

The execution of
`?- assertnb(bird, [injured(fred), ab_f(X):- injured(X)]).`
results in the following change of the second query:

`?- bird:fly(X).` Answer: `X = sam`.

*Example 5.* Consider an instance of the well-known knapsack problem with four objects 1, 2, 3, and 4 whose values and weights are given by the pairs (10, 2.1), (15, 1.2), (6, 0.4), and (10, 1.1), respectively. Furthermore, the weight limit is 1.5. Computing a solution for this problem cannot be done in current answer set solvers as real-valued arithmetic is not supported.[5] The problem can be solved in ASP−PROLOG using an ASP-encoding, called `knapsack.lp`,

$$object(1). \quad object(2). \quad object(3). \quad object(4). \quad \{selected(X) : object(X)\}.$$

and a prolog program that obtains the answer sets of `knapsack` and finds the best answer. The code of this program, called `ek.pl`, is given next:

```
:- use_asp(knapsack).
description(1, 10, 2.1).    description(2, 15, 1.2).
description(3, 6, 0.4).     description(4, 10, 1.1).
compute(Bound) :-
     findall(Q, knapsack:model(Q), L), evaluation(L, E, Bound),
     best(E, X), print_solution(X).
evaluation([], [], _).
evaluation([H|T], Result, Bound):-
     evaluation(T, X, Bound),
     findall(O, H:selected(O), LO),
     value_weight_of(LO,V,W),
     (W =< Bound -> append([(H, V, W)], X, Result);
          append([], X, Result)).
value_weight_of([], 0, 0).
value_weight_of([H|T], TotalV, TotalW):-
     description(H, V, W),
     value_weight_of(T, VR, WR),
     TotalV is V + VR, TotalW is W + WR.
best([(H,V,W)|[]], (H,V,W)).
best([(H,V,W)|T], Result):-
     best(T, (H1,V1,W1)),
     (V1 =< V -> Result=(H,V,W); Result=(H1,V1,W1)).
print_solution((X,V,W)):-
     findall(O, X:selected(O), LO),
     write('The selected objects are: '), write(LO), nl,
     write('Total weight: '), write(W), nl,
     write('Total value: '), write(V), nl.
```

[5] Scaling the values by 10 will help in this case. However, identifying a scalar for each instance is rather cumbersome.

Compiling `ek.pl`, we get:

```
| ?- compute(1.5).
The selected objects are: [3,4]
Total weight: 1.5
Total value: 16
```

## 5.3 Planning

In this subsection, we will discuss different applications of $\mathbb{ASP}-\mathbb{PROLOG}$ in planning. Given a planning problem $P = \langle D, I, G\rangle$ where $D$ is an action theory in the language $\mathcal{B}$ [Gelfond and Lifschitz, 1998], $I$ is a set of fluents representing the initial state, and $G$ is a set of fluent literals representing the goal, answer set planning [Lifschitz, 2002] solves it by translating $P$ into an ASP program, $\Pi(P)$, whose answer sets correspond one-to-one to solutions of $P$. Solutions of $P$ are then computed using an answer set solver. This method requires the user to specify the length of the plans that need to be computed. For instance, the command

$$\texttt{cclingo} -\texttt{c length=n} \;\; \Pi(P)$$

will compute a solution of length `n` of $P$, if it exists. When the user does not have a good guess on the length of the plan, he/she can compute a shortest plan by (*i*) setting `n` to 1 and (*ii*) incrementing `n` by 1 until a solution is found. This process can be expressed in $\mathbb{ASP}-\mathbb{PROLOG}$ using the following clauses:

```
plan(Name):-  compute_plan(Name, 0).

compute_plan(Name, I) :-
  Paras = [c(length=I)],
  use_asp(Name,Name,Paras),
  ( Name:model(_) ->
         format("Shortest plan length = ~q ~n",[I]),
            print_plan(Name);
         I1 is I+1,compute_plan(Name,I1)).
```

where `print_plan(Name)` collects the action occurrences in an answer set of `Name` and prints it out.[6]

One of the advantages of answer set planning is its simplicity and its generality. This has been shown in the extensive literature on answer set planning: planning with domain knowledge [Son et al., 2006], durative actions [Son et al., 2004], incomplete information [Tu et al., 2006], etc. To tackle a new type of planning domains, all it is needed is an encoding in ASP. This, however, does not allow one to exploit the different heuristics specialized for planning. Most heuristic search-based planners, on the other hand, were developed mainly for a specific type of planning domains. $\mathbb{ASP}-\mathbb{PROLOG}$ provides an ideal environment to exploit the best of the two worlds. We will next describe a heuristic-based planner in $\mathbb{ASP}-\mathbb{PROLOG}$. We will assume that the planning domain is given by a Prolog programs defining the following predicates:

[6] `Iclingo`, a solver developed by the Potsdam's group, has the ability to compute answer sets of a program incrementally and can be used for this purpose as well. However, there are some syntactically differences between `Iclingo` and `clingo`.

– $action(A)$: $A$ is an action;
– $fluent(F)$: $F$ is a fluent;
– $executable(A, Conditions)$: $A$ is executable when *Conditions* is meet;
– $causes(A, Effects, Conditions)$: *Effects* will be true after the execution of $A$ if *Conditions* are true;
– $caused(L, Conditions)$: $L$ is true whenever *Conditions* is true.

where $L$ is a fluent literal (a fluent or its negation) and *Conditions* and *Effects* are lists of fluent literals. The following is an example of the block world domain with three blocks $a$, $b$, and $c$.

```
% initial 3-blocks
blk(a).        blk(b).       blk(c).

fluent(on(X,Y)):- blk(X), blk(Y), X \== Y.
fluent(ontable(X)):- blk(X).
fluent(clear(X)):- blk(X).

action(move(X,Y,Z)):-
  blk(X), blk(Y), blk(Z), X \== Y, X \== Z, Y \==Z.
action(movetotable(X,Y)):- blk(X), blk(Y), X \== Y.
action(movefromtable(X,Y)):- blk(X), blk(Y), X \== Y.

causes(move(X,Y,Z),[on(X,Y),
  neg(clear(Y)),clear(Z),neg(on(X,Z))],[]):-  action(move(X,Y,Z)).
causes(movetotable(X,Y), [ontable(X),
  neg(on(X,Y)),clear(Y)],[]):- action(movetotable(X,Y)).
causes(movefromtable(X,Y),[on(X,Y),
  neg(clear(Y)),neg(ontable(X))],[]):- action(movefromtable(X,Y)).

executable(move(X,Y,Z), [clear(X),clear(Y),on(X,Z)]):-
  action(move(X,Y,Z)).
executable(movetotable(X,Y), [on(X,Y), clear(X)]):-
  action(movetotable(X,Y)).
executable(movefromtable(X,Y), [ontable(X), clear(X), clear(Y)]):-
  action(movefromtable(X,Y)).
```

Translating a domain representation from its Prolog encoding to its ASP-encoding is simple and can be done in Prolog as well. For example, we can use the translation from http://www.cs.nmsu.edu/~tson/ASPlan/Knowledge/. We assume that the initial state is given by a set of clauses defining the predicate `initially(L)`. With this, a generic heuristic search based planner can be implemented in ASP−PROLOG by the following code:

```
search(Name, [(State,CPlan,Value)|Rest], Goal):-
  checkGoal(State, Goal) ->
   ((write('Plan found: '), nl, pretty_print(CPlan));
    (expand(Name, State, CPlan, Rest, Goal, Queue),
     search(Name, Queue, Goal))).
```

where

- $(State, CPlan, Value)$ represents a node in the search tree, where $State$ is the module name storing the content of the state, $CPlan$ is the plan used to reach $State$ (from the initial state), and $Value$ is the heuristic value of $State$;
- $Queue$ is a priority queue (an ordered list) of nodes, ranked by the heuristic value of the nodes;
- `checkGoal(State, Goal)`: true iff $Goal$ is satisfied by $State$;
- `expand(Name, CurrentQueue, Goal, Queue)`: this predicate is true iff $Queue$ is the list of all nodes that need to be explored after expansion of the first element of $CurrentQueue$.

The key clauses in this program are those defining the expansion step `expand(Name, CurrentQueue, Goal, Queue)` and computing heuristic of a given state. The `expand` predicate is defined by:

```
expand(Name, [(State, CPlan, _)|Rest], Goal, NewQueue):-
  import(State,[(holds(X,1),holds(X,0))],Name,
                             [holds/2,occ/2],NewState,[]),
  findall((Q, X), (NewState:model(Q), Q:occ(X,0)), Models),
  update_heuristics(CPlan, Models, Goal, NextStates),
  update_list(Rest, NextStates, NewQueue).
```

The first step in `expand` creates an ASP module containing the domain representation (specified by `Name`), the initial state (by the module named `State`), with the list of import predicates in the second parameter and the list of export atoms with their arities. The successor states are updated with their corresponding heuristic and then combined with the rest of the $CurrentQueue$ to create the $NewQueue$.

We experimented with the GraphPlan distance heuristic [Hoffmann and Nebel, 2001] which can be easily implemented in Prolog. We observe that the planner performs much better than the Prolog breadth-first planner, in the sense that the number of nodes which are expanded is smaller. It remains to be seen whether this method of planning can be competitive to state-of-the-art planners. We plan an extensive experimental evaluation of this method of planning in the near future.

## 5.4 Multi-agent Planning

The previous sections present different uses of $\mathbb{ASP}-\mathbb{PROLOG}$ with one single ASP program. In this section, we show how an algorithm for computing a coordination plan between multiple agents can be implemented in $\mathbb{ASP}-\mathbb{PROLOG}$. We will assume that each agent $A$ is associated to a planning problem encoded as an ASP program, called $\Pi(A)$, similar to the program described in the previous subsection, with the following additional information:

- ◦ $\Pi(A)$ contains atoms of the form $agent(X)$, denoting that $X$ is one of the agents who can help $A$ (or ask for help from $A$);
- ◦ actions in $\Pi(A)$ are divided into two groups: the first group is specific to $A$ and the second group has the form $get\_this(F, X)$ ($A$ establishes $F$ for $X$) or $give\_me(F, X)$ ($A$ asks for help from $X$ to establish $F$);
- ◦ $\Pi(A)$ contains the initial condition as well as the goal of $A$.

For simplicity, we assume that the integrity constraints among the worlds of agents are encoded in $\Pi(A)$ as well. Detailed description of this problem can be found in [Son et al., 2009]. Given a collection of planning problems $(a_i, plan_i)$ for $i = 1, \ldots, n$, where $a_i$ is the agent and $plan_i$ is $a_i$'s planning problem, we can compute a coordination plan incrementally by:

(i) computing a possible solution, as a sequence of elements of the form $(a_j, m_j)$ where $1 \leq j \leq k$ and $m_j$ is an answer set of $plan_j$, for the first $k$ agents; and
(ii) expanding the solution by computing an answer set of $plan_{k+1}$ that is compatible with the current solution.

The computation of a step is realized by the following Prolog code.

```
compute_step(S, [], _, S):- S = [_|_].
compute_step([S|L], [(A,ProbA)|Agents], Len, Plans):-
  compatible(S, A, ProbA, Len, Models),
  Models = [_|_],
  update(S,Models,NS),
  compute_step(NS, Agents, Len, Plans).
compute_step([S|L], [(A,ProbA)|Agents], Len, Plans):-
  compute_step(L, [(A,ProbA)|Agents], Len, Plans).

compatible(S, A, ProbA, Len, Models):-
  findall(request(give_me(F,B), T),
            (member((B,MB),S), MB:occ(get_this(F,A),T)), Get),
  remove_dups(Get, ToGet),
  findall(request(get_this(F,B), T),
            (member((B,MB),S), MB:occ(give_me(F,A),T)), Give),
  remove_dups(Give, ToGive),
  append(ToGet, ToGive, Occs),
  findall(considered(Ag), member((Ag,MAg), S), Considered),
  request_okay(Occs),
  ... {create constraints on ProbA} ...
  ... {prepare the parameters Paras} ...
  use_asp(ProbA, ProbA, Paras),
  findall((A,Q), ProbA:model(Q), RModels),
  remove_dups(RModels, Models).
```

`compute_step` has four parameters: the list of possible solutions for the agents that have been considered, the list of agents without plans and their problems, the plan length, and the final solution (output). It tries to find a compatible answer set of the next agent (`compatible/5`) to update the list of possible solutions. It either gets some solutions (when the set of agents without plans is empty) or fails (when the set of possible solutions becomes empty while there are some agents without plans).

The basic steps in `compatible/5` include (*i*) collecting all exchanges generated by the previous agents and involving the agent in question; (*ii*) verifying that these exchanges are eligible (`request_okay`); (*iii*) creating the constraints on the answer set of the agent being considered; (*iv*) computing a possible plan for the agent.

We experimented with a problem involving three agents originally described in [Son et al., 2009] and observed that the program quickly gets to a solution even though the size of the set of answer sets for each individual agent is large (128, 9746, and 456, respectively). As with heuristic planning, an investigation on the scalability and performance of this method for multi-agent planning will be our focus in the near future.

## 6 Conclusion and Future Work

In this paper, we presented a system, called $\mathbb{ASP}-\mathbb{PROLOG}$, for combining ASP and Prolog. We developed a model-theoretical semantics for static programs and an operational semantics for dynamic programs. We briefly discussed a preliminary prototype of $\mathbb{ASP}-\mathbb{PROLOG}$ and showed how several practical applications can be encoded in $\mathbb{ASP}-\mathbb{PROLOG}$. One of our main goals in the near future is to complete the development of a full fledge $\mathbb{ASP}-\mathbb{PROLOG}$ system and employ it in real-world applications. We will also investigate the relationship between $\mathbb{ASP}-\mathbb{PROLOG}$ and MCSs.

## Acknowledgments

We are indebted to Michael Gelfond for his pioneering work in answer set semantics of logic programs and the many fruitful discussions on the use of answer set programming in practical applications and the need for tools which simplify the development of answer set programming applications. The important ideas underlying this paper, including the model-theoretical account based on reduction, the use of logic programming in knowledge representation and reasoning (KRR), and the development of tools for KRR's applications, have been greatly influenced by his research. The authors would also like to acknowledge the partial support from the NSF grants HRD-0420407 and IIS-0812267.

## References

[Balduccini et al., 2001] 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)

[Baral et al., 2008] Baral, C., Dzifcak, J., Son, T.C.: Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions. In: AAAI, pp. 818–823. AAAI Press, Menlo Park (2008)

[Baral et al., 2010] Baral, C., Gelfond, G., Pontelli, E., Son, T.C.: Logic programming for finding models in the logics of knowledge and its applications: A case study. TPLP 10(4-6), 675–690 (2010)

[Baral and Gelfond, 1994] Baral, C., Gelfond, M.: Logic programming and knowledge representation. Journal of Logic Programming 19/20, 73–148 (1994)

[Brewka and Eiter, 2007] Brewka, G., Eiter, T.: Equilibria in Heterogeneous Nonmonotonic Multi-Context Systems. In: AAAI, pp. 385–390 (2007)

[Dao-Tran et al., 2010] Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed nonmonotonic multi-context systems. In: KRR. AAAI Press, Menlo Park (2010)

[El-Khatib et al., 2004] Elkhatib, O., Pontelli, E., Son, T.C.: ASP-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)

[El-Khatib et al., 2005] El-Khatib, O., Pontelli, E., Son, T.C.: Integrating an Answer Set Solver into Prolog: ASP-PROLOG. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 399–404. Springer, Heidelberg (2005)

[Gebser et al., 2010] Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. TPLP 10(4-6) (2010)

[Gelfond and Leone, 2002] Gelfond, M., Leone, N.: Logic programming and knowledge representation – the A-Prolog perspective. Artificial Intelligence 138(1-2), 3–38 (2002)

[Gelfond and Lifschitz, 1988] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP, pp. 1070–1080 (1988)

[Gelfond and Lifschitz, 1998] Gelfond, M., Lifschitz, V.: Action languages. ETAI 3(6) (1998)

[Heljanko and Niemelä, 2003] Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming 3(4,5), 519–550 (2003)

[Hoffmann and Nebel, 2001] Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research 14, 253–302 (2001)

[Lifschitz, 2002] Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138(1-2), 39–54 (2002)

[Marek and Truszczyński, 1999] Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-year Perspective, pp. 375–398 (1999)

[Niemelä, 1999] Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3,4), 241–273 (1999)

[Pontelli et al., 2009] Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. TPLP 9(1), 1–56 (2009)

[Son et al., 2004] Son, T.C., Tuan, L.C., Baral, C.: Adding Time and Intervals to Procedural and Hierarchical Control Specifications. In: AAAI, pp. 92–97. AAAI Press, Menlo Park (2004)

[Son et al., 2006] Son, T.C., Baral, C., Tran, N., McIlraith, S.: Domain-dependent knowledge in answer set planning. ACM Trans. Comput. Logic 7(4), 613–657 (2006)

[Son et al., 2009] Son, T.C., Pontelli, E., Sakama, C.: Logic programming for multiagent planning with negotiation. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 99–114. Springer, Heidelberg (2009)

[Son and Sakama, 2009] Son, T.C., Sakama, C.: Reasoning and planning with cooperative actions for multiagents using answer set programming. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 208–227. Springer, Heidelberg (2010)

[Tu et al., 2006] Tu, P.H., Son, T.C., Baral, C.: Reasoning and Planning with Sensing Actions, Incomplete Information, and Static Causal Laws using Logic Programming. Theory and Practice of Logic Programming 7, 1–74 (2006)

[Tu et al., 2010] Tu, P.H., Son, T.C., Gelfond, M., Morales, R.: Approximation of action theories and its application to conformant planning. Artificial Intelligence Journal (2010) (to appear)

# On the Practical Side of Answer Set Programming

Tommi Syrjänen

Variantum Oy
Tekniikantie 12, 02150 Espoo, Finland
tommi.s.syrjanen@iki.fi

**Abstract.** We examine issues that arise from creating practical tools that combine answer set programming (ASP) with programs created using traditional programming languages. A tool is mostly written in a traditional language and it calls an ASP solver as an oracle to solve some difficult subproblem that is best represented using ASP. We give a brief introduction on a generate-and-test based methodology for creating ASP programs and on how to optimize them for efficiency. We examine methods for computing answer sets incrementally based on user choices and show how we can guide the user in making the choices and how to give diagnostic information in the case that the user's choices are inconsistent. We use the kakuro puzzles as a practical example.

## 1 Introduction

If we want to use Answer Set Programming (ASP) to solve practical problems, it is not enough to create an ASP encoding for the problem. A usable tool has to have a convenient way to enter the problem instances into the system and it needs to present the solutions in a way that is easy to understand.

Most ASP semantics are not Turing-complete.[1] This is both a strength and a weakness at the same time. It is a weakness because we cannot implement arbitrary algorithms. On the other hand, when we restrict the computational power of a language, we can design it so that we can have elegant and efficient programs for some set of interesting problems. In particular, many **NP**-complete problems have simple and natural encodings as ASP programs.

In particular, the declarative ASP semantics are not suitable for input handling and output processing so a real-life tool has to embed an ASP program in a framework that is implemented using a traditional programming language. The traditional program uses an ASP solver as an oracle to solve a problem that is difficult to handle with traditional programming languages.

There are several other considerations in creating ASP programs. One is efficiency. In an ideal world all correct encodings would be suitable for practical use. In practice it is common that two problem encodings that are equivalent in the sense that they have the same answer sets have vastly different runtime

[1] A language is Turing-complete if it is possible to simulate the computations of an arbitrary Turing machine using a program written with it.

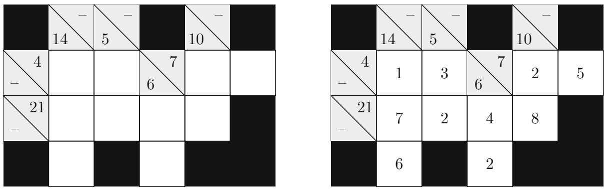


**Fig. 1.** A cross-sum/kakuro puzzle and a solution

performances. Computing answer sets is a difficult operation because we are usually working on **NP**-hard problems. In difficult problem instances a solver can make a poor guess early in the computation and then spend a lot of time examining a part of the search space that does not contain any answer sets at all [5].

Another consideration is that the user may not necessarily want to compute a complete solution at one time. For example, in the configuration problem a user wants to find a configuration that satisfies all her requirements and there may be any number of them, including none if the requirements are inconsistent. We could require the user to enter all the requirements before starting the solver but a more usable system allows interactive configuration. The user enters her choices one at a time and the system checks what remaining choices are possible after each selection and presents only those options to her and gives an explanation why the other options are impossible.

In the remaining paper we will examine issues that arise when we want to create a program to solve kakuro puzzles. We will first give a straightforward encoding that directly follows the problem definition and then create an optimized encoding that allows us to solve larger puzzles. Finally, we examine how to create a complete puzzle game where the player solves the puzzles manually but she can ask the ASP oracle for hints when stuck. We will use Smodels [9] for our programs but the principles extend also to other ASP systems.

### 1.1 Kakuro or Cross Sum Puzzles

In a *kakuro* or *cross sum* puzzle we have to find a way to place a number from the interval 1–9 to each square of a grid so that the rows and columns sum to given totals. There is an additional restriction that each line may contain each number at most once.

Figure 1 shows an example puzzle and one of its solutions. This is a poor quality example in that it has more than one solution as a well-formed kakuro puzzle should have a unique answer.

## 2 Principles of ASP Encodings

Herb Sutter's observation that "it is far, far easier to make a correct program fast than it is to make a fast program correct" [11] holds also in ASP. The first step in creating an ASP program is to create a straightforward encoding of the problem domain and optimize it only if it is not efficient enough for real use.

## 2.1 Generate and Test Method

Usually the simplest approach is the *generate and test method* [8,2]. In the ASP context it means that we divide our problem into two parts:

1. a *generator* that creates all possible solution candidates; and
2. a *tester* that checks whether a particular candidate is, indeed, a valid solution.

The generator contains the rules that create choices over atoms. These choices should be made as clear as possible. How exactly we implement them depends on the ASP semantics that we are using.

In the original stable model semantics for normal logic programs [3] choices are made using even negative recursion: $a$ is true if $b$ is false and $b$ is true if $a$ is false. It is possible to create a program whose answer sets are defined by complex arrangements of negative recursion where each atom depends on most other atoms of the program. The problem with this approach is that the resulting program is difficult to debug. The tiniest mistake in the program code results in an unsatisfiable program and it is not easy to find the exact place of the error.

Keeping the generator part separate from the tester makes debugging easier as there are fewer interactions between predicates to consider. When we use normal logic programs with classical negation [4] a good pattern for creating choices is the pair of rules:

$$\begin{aligned} \mathit{choose}(X) &\leftarrow \mathit{option}(X), \text{not } \neg\mathit{choose}(X) \qquad (1)\\ \neg\mathit{choose}(X) &\leftarrow \mathit{option}(X), \text{not } \mathit{choose}(X)\ . \qquad (2) \end{aligned}$$

In its simplest form a tester may be a single constraint that prunes out invalid answer set candidates:

$$\leftarrow \mathit{choose}(X), \mathit{invalid\text{-}choice}(X)\ . \qquad (3)$$

Conceptually it is often useful to identify *auxiliary predicates* as the third part of the program. These predicates are used during computation but they themselves are not really part of the solution. For example, if we want to solve the Hamiltonian cycle problem we need to compute the transitive closure of the edges of the cycle to ensure that it visits every node but the closure itself is not interesting to the user.

These auxiliary predicates should not introduce new choices to the program. Instead, their extensions should follow directly from predicates defined by the generator or from facts given as input to the program. This is particularly important when we need to preserve the number of solutions—if an auxiliary predicate can add a new choice to the program, then there may be many answer sets that all correspond to the same solution.

In this paper we use Smodels *choice rules* for encoding the generators. A choice rule has the form:

$$\{\ \mathit{choose}(X) : \mathit{option}(X)\ \}\ . \qquad (4)$$

Intuitively, a choice rule states that if the body of the rule is satisfied, then any number of the atoms in its head may be true. Here (4) states that we can take any set of atoms $\mathit{choose}(X)$ where $\mathit{option}(X)$ is true.

If we are using an ASP system based on disjunctive logic programming [2], we can express the same condition with the rule:

$$\mathit{choose}(X) \mid \neg \mathit{choose}(X) \leftarrow \mathit{option}(X) \ . \tag{5}$$

## 2.2 Uniform Encodings

A problem encoding is *uniform* [10,2,6] if we can use it to solve all instances of the problem: we have a non-ground program that encodes the constraints of the problem domain and each problem instance is defined as a set of ground facts.

The main advantage of using uniform encodings is *modularity*. We can often create a program by partitioning it into subproblems and then combining their encodings. This is much easier to do when the encodings are uniform and then we can also replace a component by another logically equivalent one.

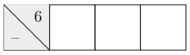
## 2.3 On Optimization

Each atom that occurs in a ground ASP program potentially doubles the size of the search space. Fortunately the situation is not as bad as that in most practical cases since such an exponential growth would prevent us from using ASP on any real-life problems. Computationally hard instances of **NP**-complete problems are rare [1,7] but even relatively easy instances may be hard to solve in practice and the difficulty goes up with the size of the program. Thus, one of the most important goals in optimizing encodings is to try to minimize the number of atoms that occur in a program.

At this point there are no comprehensive guides for optimizing answer set programs. It is likely that there is no single approach that always would lead to the most efficient encoding. For example, it is possible that sometimes adding a new atom may allow us to define new constraints to prune out the search space so the larger program is more efficient. It can also happen that adding a new constraint actually increases the running time because processing it takes some time and it removes so small portions of the search space that it could have been explored faster. Also, the solvers use complex heuristics to decide the order that they explore the search space and a small change in a program may have a large effect on how well the heuristics behaves for the problem.

Our practical experience has been that using the following procedure leads to reasonably efficient encodings:

1. Create a working encoding in the most straightforward way as possible.
2. Define new auxiliary predicates that allow tighter definitions for rules in the generator in the sense that we get less candidate answer sets.
3. Add new constraints to prune out parts of search space that can be guaranteed to not contain valid solutions.



$$\begin{array}{l} \textit{line}(l) \\ \textit{line-data}(l, 6, 3) \\ \textit{in-line}(l, x, y) \\ \textit{in-line}(l, x+1, y) \\ \textit{in-line}(l, x+2, y) \end{array}$$

**Fig. 2.** Kakuro input encoding

## 3 Basic Encoding

Creating a uniform ASP encoding happens in two steps: we have to decide what facts we use to represent the instances and we need to write the actual rules.

### 3.1 Representing the Puzzle Instances

From the viewpoint of the user the most convenient way to define kakuro puzzles is to have a graphical tool that allows her to draw a grid on the screen and set the line totals. We need to take this representation and turn it into a set of facts.

The lines are the key concept of kakuro puzzles so we structure the encoding around them. We need to know what squares belong to which lines and the totals for each line. We give an unique identifier for each horizontal and vertical line and use the predicates that are shown in Fig 2 to connect the squares with the totals. An atom $\textit{line}(l)$ denotes that $l$ is an identifier for a line and $\textit{in-line}(l, x, y)$ denotes that the square $(x, y)$ occurs in a line with the identifier $l$. An atom $\textit{line-data}(l, n, t)$ denotes that the line with the identifier $l$ sums to the total $t$ and is $n$ squares long.[2]

The tool that translates the user-generated puzzle description into facts is responsible for generating the line identifiers and identifying which squares belong to which lines. Both are straightforward things to do with traditional programming languages.

### 3.2 Rules of the Game

As the first step we define two auxiliaries that we use in defining the rules:

$$\textit{square}(X, Y) \leftarrow \textit{in-line}(L, X, Y) \tag{6}$$

$$\textit{number}(1 \mathrel{..} 9)\ . \tag{7}$$

For our generator we need a rule that selects a number for every square of the puzzle:

$$1\ \{\textit{has-number}(N, X, Y) : \textit{number}(N)\}\ 1 \leftarrow \textit{square}(X, Y)\ . \tag{8}$$

[2] The basic encoding does not use the line length data but the optimized one does. It is included already here because then the same preprocessor can be used for both encodings.

The first rule of the tester ensures that no number occurs twice on one line:

$$\leftarrow 2\ \{\mathit{has\text{-}number}(N, X, Y) : \mathit{in\text{-}line}(L, X, Y)\}, \mathit{number}(N), \mathit{line}(L). \tag{9}$$

A total of a line may not be too large:

$$\leftarrow T + 1\ [\mathit{has\text{-}number}(N, X, Y) : \mathit{number}(N) : \mathit{in\text{-}line}(L, X, Y) = N], \\ \mathit{line}(L). \tag{10}$$

This is a weight constraint rule where we assign the weight $n$ for each atom $\mathit{has\text{-}number}(n, x, y)$ and then asserts that it is an error if the total weight of the atoms selected to be true is too large for the rule.

Finally, the total of a rule may not be too small:

$$\leftarrow [\mathit{has\text{-}number}(N, X, Y) : \mathit{number}(N) : \mathit{in\text{-}line}(L, X, Y) = N]\ T - 1, \\ \mathit{line}(L). \tag{11}$$

## 4 Optimized Encoding

The basic encoding creates a great number of spurious choices since it has rules for placing all digits in all squares, even though in most cases we can immediately rule out some of them. Consider the line in Fig 2. As the only combination of three digits whose sum is six is $1 + 2 + 3 = 6$, we know for certain that the line has to contain these digits in some order and no other choices are possible.

We can make the encoding more efficient by using a two-phase selection where we choose a digit combination for an entire line and then place the digits of a combination into the squares of the line in some order.

We first compute the combinations that may possibly occur on a given line, and then select one of them. We have to make a decision of what we mean with a possible combination. A literalist approach would be to define that a combination is possible if it occurs in a correct solution of the puzzle. This would be useless in practice since we would have to solve the puzzle to know what solutions it has. Instead, we want an approximation that allows us to prune out most incorrect combinations without losing correct solutions while still being easy to compute.

A simple heuristics is that we start with the set of combinations of the correct length and total, and then look at the intersections of the lines. If a combination for one line contains only digits that may not occur in the other line, we remove that combination as impossible. Similarly, we remove a combination if it contains some digit that cannot be placed in any square of the line. We can implement this heuristics using a stratified set of rules so we can find the possible combinations with a straightforward least model computation.

*Example 1.* Consider the leftmost vertical line from Fig 3. Since the only two-digit combination that sums to four is $\{1, 3\}$, we know that no other digits are possible for the two squares. The rightmost vertical line has three plausible combinations: $\{1, 7\}$, $\{2, 6\}$, and $\{3, 5\}$. Next, we examine the horizontal line

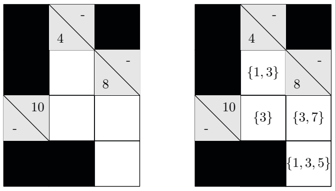


**Fig. 3.** A puzzle and a computed set of possible numbers

that has four plausible combinations: $\{1, 9\}$, $\{2, 8\}$, $\{3, 7\}$, and $\{4, 6\}$. As its left square has either 1 or 3, we can immediately rule out $\{2, 8\}$ and $\{4, 6\}$. As the right square may not contain a digit higher than 7, we can reject also $\{1, 9\}$ since there is no square where we could place 9. Finally, we can reject the combination $\{2, 6\}$ from the right vertical line since neither of the numbers can occur in the horizontal line.

### 4.1 Representing the Combinations

Each non-zero digit may occur zero or one times in a line so there are $2^9 - 1 = 511$ different possible number combinations.[3] As there are so few combinations, we can compute them in advance and store them as ground facts. An atom *combination*(*id*, *len*, *sum*) denotes that the combination with the identifier *id* has the length *len* and the total *sum* and an atom *in-combination*(*id*, *d*) denotes that the digit *d* occurs in the combination *id*.

### 4.2 Selecting Combinations and Numbers

We use predicates $p\text{-}combination(C, L)$ and $p\text{-}number(N, X, Y)$ to denote the possible combinations and numbers. An atom $p\text{-}combination(c, l)$ is true if the combination $c$ can occur in the line $l$ and an atom $p\text{-}number(n, x, y)$ is true if the number $n$ may occur in the square $(x, y)$.

We choose exactly one combination for each line out of the possible options:

$$1\ \{has\text{-}combination(C, L) : p\text{-}combination(C, L)\}\ 1 \leftarrow line(L)\ . \tag{12}$$

Each square has exactly one possible number.

$$1\ \{has\text{-}number(N, X, Y) : p\text{-}number(N, X, Y)\}\ 1 \leftarrow square(X, Y)\ . \tag{13}$$

Note that (13) does not depend on the selected combination directly. This allows us to have only one ground rule that can be used to derive an atom which is a micro-optimization that may in some cases help to guide the solver heuristics to

[3] High-quality kakuro puzzles do not contain lines of length one so they have only 502 combinations. However, a real kakuro solver should be able to handle also the poor-quality puzzles.

the right direction. We then need to add a constraint to ensure that the chosen numbers belong to the chosen combination:

$$\begin{aligned}\leftarrow\ & \mathit{has\text{-}number}(N,X,Y), \mathit{has\text{-}combination}(C,L),\\ & \text{not } \mathit{in\text{-}combination}(C,N), \mathit{p\text{-}number}(N,X,Y),\\ & \mathit{p\text{-}combination}(C,L), \mathit{in\text{-}line}(L,X,Y)\ .\end{aligned} \tag{14}$$

We also need a variant of (9) to say that no number occurs twice in one line:

$$\begin{aligned}\leftarrow\ & 2\ \{\mathit{has\text{-}number}(N,X,Y) : \mathit{in\text{-}line}(L,X,Y)\},\\ & \mathit{p\text{-}combination}(C,L), \mathit{in\text{-}combination}(C,N)\ .\end{aligned} \tag{15}$$

## 4.3 Possible Combinations

We compute the possible combinations by starting with an assumption that combinations that match the length and the total are plausible and then by removing all combinations that we can prove to be impossible.

$$\mathit{pl\text{-}combination}(C,L) \leftarrow \mathit{line\text{-}data}(L,N,T), \mathit{combination}(C,N,T)\ . \tag{16}$$

A number is plausible if it occurs in a plausible combination of a line:

$$\begin{aligned}\mathit{pl\text{-}number}(N,L,X,Y) \leftarrow\ & \mathit{pl\text{-}combination}(C,L), \mathit{in\text{-}line}(L,X,Y),\\ & \mathit{in\text{-}combination}(C,N)\ .\end{aligned} \tag{17}$$

All numbers that are not plausible for both intersecting lines have to be removed:

$$\begin{aligned}\mathit{remove}(N,X,Y) \leftarrow\ & \mathit{in\text{-}line}(L,X,Y), \mathit{number}(N),\\ & \text{not } \mathit{pl\text{-}number}(N,L,X,Y)\ .\end{aligned} \tag{18}$$

A combination is impossible for a line if some number in it has been removed from every square of the line:

$$\begin{aligned}\mathit{impossible}(C,L) \leftarrow\ & \mathit{remove}(N,X,Y) : \mathit{in\text{-}combination}(C,N),\\ & \mathit{pl\text{-}combination}(C,L), \mathit{in\text{-}line}(L,X,Y)\ .\end{aligned} \tag{19}$$

Here we use a Smodels conditional literal for universal quantification inside the rule body. During instantiation the conditional literal $\mathit{remove}(N,X,Y)$ : $\mathit{in\text{-}combination}(C,N)$ is replaced by the conjunction of atoms $\{\mathit{remove}(n,X,Y) \mid \mathit{in\text{-}combination}(c,n) \text{ is true}\}$ and the values of $X$ and $Y$ are bound by the atom $\mathit{in\text{-}line}(L,X,Y)$.

A combination is also impossible if all its numbers are removed from a square:

$$\begin{aligned}\mathit{impossible}(C,L) \leftarrow\ & \mathit{remove}(N,X,Y) : \mathit{in\text{-}combination}(C,N),\\ & \mathit{pl\text{-}combination}(C,L), \mathit{in\text{-}line}(L,X,Y)\ .\end{aligned} \tag{20}$$

**Table 1.** Comparison of encodings

| Encoding | Puzzle: Fig. 1 | | | Puzzle: Fig. 4 | | |
|---|---|---|---|---|---|---|
| | Atoms | Rules | Choices | Atoms | Rules | Choices |
| Basic | 252 | 275 | 2 | 512 | 513 | 487196 |
| Optimized | 183 | 565 | 3 | 414 | 675 | 0 |

When we have marked a combination impossible, we may have to remove its numbers from the set of possible numbers. We need to keep track of which combinations have a number and remove it after all of them are ruled out:

$$\begin{aligned} \mathit{provides}(N, C, L, X, Y) &\leftarrow \mathit{pl\text{-}number}(N, L, X, Y), \\ &\quad \mathit{pl\text{-}combination}(C, L), \\ &\quad \mathit{in\text{-}combination}(C, N) \ . \end{aligned} \tag{21}$$

$$\begin{aligned} \mathit{remove}(N, X, Y) &\leftarrow \mathit{impossible}(C, L) : \mathit{provides}(N, C, L, X, Y), \\ &\quad \mathit{in\text{-}line}(L, X, Y), \mathit{number}(N) \ . \end{aligned} \tag{22}$$

Finally, all plausible combinations that are not impossible are possible and all plausible numbers that are not removed are possible:

$$\mathit{p\text{-}combination}(C, L) \leftarrow \mathit{pl\text{-}combination}(C, L), \text{not } \mathit{impossible}(C, L) \ . \tag{23}$$

$$\mathit{p\text{-}number}(N, X, Y) \leftarrow \mathit{pl\text{-}number}(N, L, X, Y), \text{not } \mathit{remove}(N, X, Y) \ . \tag{24}$$

### 4.4 Comparison

Table 1 shows how the two encodings behave when we use `smodels-2.34` to solve the kakuro puzzles that are shown in Fig 1 and Fig 4. The table shows the numbers of ground atoms and rules in the instantiation of the programs and also the number of choices that the solver had to make while searching for the solution.

In both cases the optimized encoding has much fewer atoms but it has more rules. In the case of the smaller example the solver found an answer as easily using both—the one extra choice for the optimized encoding comes for selecting a combination for a line. The larger example shows the importance of good encodings. With the basic encoding the solver had to explore almost half a million choices before finally finding the unique correct answer but with the optimized encoding it found the answer without having to make a single guess.

## 5 Interactive Use

The two encodings both allow us to completely solve kakuro puzzles. Unfortunately, this is not what the average user wants to do. Instead, most people who are interested in kakuro puzzles want to solve them themselves. In this section we examine how we can create a kakuro solver that gives as much support to the user that she wants and no more. The goal is to have a program where the user can solve puzzles using her usual methods but where she can ask a hint from the computer when stuck. The issues that we meet occur also in other problem domains, such as in product configuration.

## 5.1 Use Cases

The first thing to do is to identify the possible user groups and what they would want to do with a tool that can solve puzzles. The two main groups that would be interested in such a tool are:

1. people who solve puzzles for fun; and
2. people who create new puzzle instances.

We will concentrate mostly on the first group. Creating high-quality puzzles in an automatic or semi-automatic fashion is a complex problem that cannot be addressed adequately in this work.

Next, we have to identify what the puzzle solvers would want to do with a kakuro solver. Roughly speaking the things can be divided into two categories:

1. *hints* that tell the user how to proceed towards the solution; and
2. *explanations* that tell the user why some choices are impossible or necessary.

There should be different levels of hints so that the user will not get too much information if that is not desired. People solving puzzles typically look at local information trying to find places where it is possible to make deductions about the contents of a single line based on the lines that intersect with it and the hints should guide that process. The possible hints that we examine are:

1. identifying which squares can be filled based on the existing partial solution;
2. identifying the set of numbers that might go into a particular square; and
3. identifying the places where the user made an error.

The two first cases are specific to kakuro solving but the third case occurs in every problem domain where we compute answer sets based on the user input. For example, if the user requirements for a configuration are impossible to satisfy, we have to have a way to find out the incompatible requirements.

Explanations are closely related to the problem of inconsistent user input. If there are no answer sets that would satisfy the user's requirements, it is not enough to just tell her that the choices are inconsistent but there should also be a concrete explanation that tells how to resolve the problem. An advantage of ASP is that it is often possible to compute the explanation on the fly.

## 5.2 Representing User Choices

We use the predicate symbol *user-choice*/4 to represent the numbers that the user has placed on the grid. An atom *user-choice*$(d, x, y, i)$ denotes that the $i$th move of the user was to place the digit $d$ into the square $(x, y)$. We use a new predicate symbol instead of defining *has-number*/3 directly because it allows more flexibility in computing hints and explanations. We store the order of the moves because we may want to answer questions like: "What was the first mistake that the user made."

## 5.3 Identifying User Errors

In many problem domains we want to detect inconsistent user input as early as possible. An interactive system should have a satisfiability check after each input choice if the problem instance is simple enough that the check can be executed fast enough to not annoy the user. As kakuro is a game we do not do this and instead make the consistency check only if the user requests it.

There are two possible approaches that we can take for finding the errors:

1. we can find the first error that the user made and invalidate all choices made after that; or
2. we can compute a minimal set of inputs that cause an inconsistency.

If a puzzle instance has only one solution, we can compare the correct answer to the user choices to find the errors. This is not enough if we want our tool to be able to handle arbitrary puzzles that may have multiple answers. There may also be many different minimal sets that cause a contradiction. For practical purposes it is enough that the tool reports one such set to the user.

Trying to find a minimal set of inputs that cause an inconsistency in a Smodels program is computationally in the second level of the polynomial hierarchy[4] [12]. The expressive power of the full language would allow us to write a program to solve it directly but the program would be unintuitive and inefficient. Instead, it is more efficient to use an approach where we use a series of ASP queries to find the answer.

The idea is to start with the assumption that all user choices can be met and then progressively relax this condition by canceling choices until the resulting program is satisfiable.

The general rule is that if the user puts a number somewhere, it has to be there unless the choice is explicitly canceled:

$$\leftarrow \mathit{user\text{-}choice}(N, X, Y, I), \mathrm{not}\ \mathit{has\text{-}number}(N, X, Y), \mathrm{not}\ \mathit{cancel}(I)\ . \tag{25}$$

When we want to find a minimal set of errors, we state that at most $n$ choices may be canceled:

$$\{\mathit{cancel}(I) : \mathit{choice}(I)\}\ n\ . \tag{26}$$

The numerical constant $n$ is set by the main program that calls the ASP solver as an oracle. Setting $n$ to zero allows us to check whether the user choices are satisfiable and if not, we can increase it by one at a time until an answer set exists.[5] The atoms $\mathit{cancel}/1$ that are true in the answer set denote the errors that the user made.

We can find the first error using the rule:

$$\mathit{cancel}(I) \leftarrow \mathit{choice}(I), I > n\ . \tag{27}$$

[4] This holds when there are no function symbols in the program and the number of variables that may occur in a rule is limited.

[5] We could also use a Smodels minimize statement to get the same effect. However, minimize statements are often less efficient in practice than using multiple queries with hard limits.

where $n$ is again a numerical constant set by the main program. In this case we can use binary search to find the point where the program gets unsatisfiable. Rules (26) and (27) should not be used in the same query.

## 5.4 Identifying Solvable Squares

The rules that we use to compute the possible combinations and numbers already give us a way to identify the squares that can be filled at the beginning of the puzzle: they are those where there is exactly one true atom $\mathit{p\text{-}number}(N, X, Y)$. Now we extend this idea to take into account the choices that the user has made during the play. We introduce new predicate symbols so that the new rules do not interfere with the original computation of possible combinations and numbers.

If the user has placed a number into a square, it occurs in all lines that go through the square:

$$\mathit{occurs}(N, L) \leftarrow \mathit{user\text{-}choice}(N, X, Y, I), \mathit{in\text{-}line}(L, X, Y) \ . \tag{28}$$

A choice may have ruled out some possible combinations:

$$\begin{aligned}\mathit{ruled\text{-}out}(C, L) \leftarrow\ & \mathit{p\text{-}combination}(C, L), \mathit{occurs}(N, L), \\ & \text{not } \mathit{in\text{-}combination}(C, N) \ .\end{aligned} \tag{29}$$

A digit can occur in a line if it occurs in a combination that is not ruled out and if it is not anywhere else in the line:

$$\begin{aligned}\mathit{can\text{-}occur}(N, L) \leftarrow\ & \mathit{p\text{-}combination}(C, L), \mathit{in\text{-}combination}(C, N), \\ & \text{not } \mathit{ruled\text{-}out}(C, L), \text{not } \mathit{occurs}(N, L) \ .\end{aligned} \tag{30}$$

A number is a possible choice for a square if it can occur in all lines that go through the square:

$$\begin{aligned}\mathit{p\text{-}choice}(N, X, Y) \leftarrow\ & \mathit{can\text{-}occur}(N, L) : \mathit{in\text{-}line}(L, X, Y), \\ & \mathit{number}(N), \mathit{square}(X, Y) \ .\end{aligned} \tag{31}$$

Finally, we identify the squares that contain exactly one option:

$$\mathit{m\text{-}choice}(X, Y) \leftarrow \mathit{p\text{-}choice}(N_1, X, Y), \mathit{p\text{-}choice}(N_2, X, Y), N_1 < N_2 \ . \tag{32}$$

$$\mathit{only\text{-}choice}(X, Y) \leftarrow \mathit{square}(X, Y), \text{not } \mathit{m\text{-}choice}(X, Y) \ . \tag{33}$$

The squares for which $\mathit{only\text{-}choice}(X, Y)$ is true can be filled at this point. We can tell the user either one or all of these squares. Figure 4 shows an example where we can fill three squares after the first move. The predicate *p-choice* tells us which numbers are possible for which squares and we can tell those also the user if she desires it.

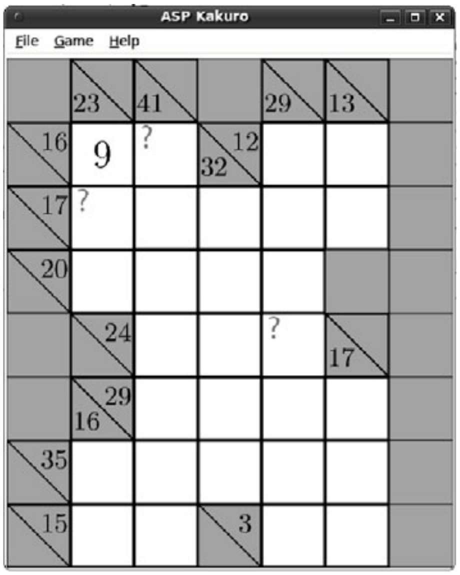


**Fig. 4.** An example of hint computation. Squares marked with '?' can be immediately deduced

## 5.5 Explanations

High-quality explanations are particularly useful for ASP programs that are intended to be used by the general public. We cannot expect an ordinary non-expert user to understand the program and its answer sets—the syntax and semantics of logic programs are often difficult to even programmers who are not familiar with them. An explanation abstracts away the rules of the program and connects the user choices to the results. The driver program has to have some pretty-printing facility that takes out the explanation computed by the ASP engine and presents it to the user in some format that is easy to understand.

In the kakuro case we examine the case where the user asks for the reason why a specific square cannot contain a specific number.[6] Note that this question has a reasonable answer only if the previous choices constrain the choices enough. The user queries are given with the predicate $\textit{user-explain}(N, X, Y)$.

When checking why a number is not in a square, we first assert that it is not possible. This prevents us from trying to compute an non-existing explanation.

$$\leftarrow \textit{user-explain}(N, X, Y), \text{not } \textit{remove}(N, X, Y) \ . \tag{34}$$

It often happens that it is not enough to examine only one square when constructing an explanation so we define a new predicate for marking the interesting squares and initialize it with the user query:

$$\textit{explain}(N, X, Y) \leftarrow \textit{user-explain}(N, X, Y) \ . \tag{35}$$

[6] The case of finding out why a particular number has to be in a square can be solved using rules based on Section 5.4.

The simplest reason why a number may not be in a square is that it has been placed on a different square of the same line:

$$\begin{aligned} \mathit{already\text{-}in}(N, X_1, Y_1, X_2, Y_2) \leftarrow\ & \mathit{explain}(N, X_1, Y_1), \\ & \mathit{user\text{-}choice}(N, X_2, Y_2, I), \\ & \mathit{in\text{-}line}(L, X_1, Y_1), \mathit{in\text{-}line}(L, X_2, Y_2), \\ & ((X_1 \neq X_2) \vee (Y_1 \neq Y_2)) \ . \end{aligned} \tag{36}$$

The second possibility is that the number does not exist in any plausible combination for one of the lines that goes through the square. In practice, this happens only if the total of the line is small.

$$\begin{aligned} \mathit{no\text{-}combination}(N, L) \leftarrow\ & \mathit{explain}(N, X, Y), \mathit{in\text{-}line}(L, X, Y), \\ & \text{not } \mathit{pl\text{-}number}(N, L, X, Y) \ . \end{aligned} \tag{37}$$

In other cases we have to deduce why the number has been removed. For this we need to identify why all combinations containing it have been ruled out.

$$\begin{aligned} \mathit{explain\text{-}impossible}(C, L) \leftarrow\ & \mathit{pl\text{-}combination}(C, L), \mathit{explain}(N, X, Y), \\ & \mathit{in\text{-}combination}(C, N) \ . \end{aligned} \tag{38}$$

A combination is impossible if the user has placed a number on the line that is not in it:

$$\begin{aligned} \mathit{incompatible\text{-}choice}(C, L, N, X, Y) \leftarrow\ & \mathit{explain\text{-}impossible}(C, L), \\ & \mathit{pl\text{-}combination}(C, L), \\ & \text{not } \mathit{in\text{-}combination}(C, N), \\ & \mathit{user\text{-}choice}(N, X, Y, I), \\ & \mathit{in\text{-}line}(L, X, Y) \ . \end{aligned} \tag{39}$$

If we find out that there is an existing incompatible choice, we do not need a further explanation for the impossible combination.

$$\begin{aligned} \mathit{explained}(C, L) \leftarrow\ & \mathit{incompatible\text{-}choice}(C, L, N, X, Y), \\ & \mathit{plausible\text{-}combination}(C, L), \\ & \mathit{in\text{-}combination}(C, N), \mathit{in\text{-}line}(L, X, Y) \ . \end{aligned} \tag{40}$$

A combination is also impossible if one of its numbers cannot be placed in any square. In these cases we need an explanation for the removed number.

$$\begin{aligned} \mathit{removed\text{-}number}(C, N, L) \leftarrow\ & \mathit{explain\text{-}impossible}(C, L), \mathit{pl\text{-}combination}(C, L), \\ & \mathit{remove}(N, X, Y) : \mathit{in\text{-}line}(L, X, Y), \\ & \mathit{in\text{-}combination}(C, N), \text{not } \mathit{explained}(C, L) \ . \end{aligned} \tag{41}$$

$$\begin{aligned} \mathit{explain}(N, X, Y) \leftarrow\ & \mathit{removed\text{-}number}(C, N, L), \mathit{in\text{-}line}(L, X, Y), \\ & \mathit{in\text{-}combination}(C, N), \\ & \mathit{pl\text{-}combination}(C, L) \ . \end{aligned} \tag{42}$$

A combination is also impossible if no number from it can fit in some square of the line:

$$\begin{aligned} \textit{empty-square}(C, L, X, Y) \leftarrow\ & \textit{explain-impossible}(C, L), \textit{pl-combination}(C, L), \\ & \textit{remove}(N, X, Y) : \textit{in-combination}(C, N), \qquad (43) \\ & \textit{in-line}(L, X, Y), \text{not } \textit{explained}(C, L)\ . \\ \textit{explain}(N, X, Y) \leftarrow\ & \textit{empty-square}(C, L, X, Y), \textit{pl-combination}(C, L) \\ & \textit{in-line}(L, X, Y), \textit{in-combination}(C, N)\ . \qquad (44) \end{aligned}$$

Giving the user an answer set as an explanation does not help much. Instead, it has to be translated into terms that she understands. This means extracting the explanatory atoms from the answer set and processing them into a human-readable format. The intuitive meanings of the explanatory atoms are:

– $\textit{already-in}(N, X_1, Y_1, X_2, Y_2)$: the number $N$ may not be placed at $(X_1, Y_1)$ because it is already present at $(X_2, Y_2)$.
– $\textit{no-combination}(N, L)$: there is no combination at all for line $L$ that contains $N$.
– $\textit{incompatible-choice}(C, L, N, X, Y)$: the combination $C$ cannot be placed on the line $L$ because it does not contain the number $N$ that is already placed on the square $(X, Y)$ that belongs to $L$.
– $\textit{removed-number}(C, N, L)$: the combination $C$ may not be placed on the line $L$ since the number $N$ from it cannot be placed in any square of it.
– $\textit{empty-square}(C, L, X, Y)$: the combination $C$ may not be placed on the line $L$ since no number from it can be placed on the square $(X, Y)$.

### 5.6 Puzzles with Multiple Solutions

The low-quality puzzles that have more than one solution have the problem that there will be situations where the user cannot proceed with pure deduction since there is more than one valid option for every remaining square. In these cases we can use the ASP program to force the puzzle to have a unique solution.

We do this by first computing one solution and then reveal enough numbers from it to guarantee an unique solution for the remaining squares. One way to do this is to select squares one at a time by random and checking whether the puzzle still has more than one solution after the number is fixed in place.

### 5.7 Considerations on Puzzle Generation

Even though automatic creation of puzzles is too large a subject to examine a detail, some general observations can be made. The first thing to note is that a puzzle creator can easily use the kakuro solver to check whether a puzzle has a unique answer. If there are more than one, she can tweak the grid layout and check again.

The second observation is that the solver can be used for estimating the difficulty of a problem instance. We can count the number of squares that can be deduced at the start and perhaps at some other points of game play and then compute a suitable metric based on that figure.

## 6 Conclusions

The aim of this paper was to give a brief introduction to issues that have to be considered when writing an ASP program intended for the general public. The output of ASP solvers is cryptic to the uninitiated. We cannot expect that an average user can decode an answer set or find out why the solver gives only 'False' as its output. Instead, we have to build a supporting framework with traditional programming languages that creates the input for the solver and then translates the answer to terms that the user understands.

In particular, a tool should be able to handle cases where there is no answer set. When the user gives an inconsistent input, we should be able to find out where the error lies and explain its causes to her. It is often possible to use an ASP program for the error-finding. The technique from Section 5.3 where we allow a set of user choices to be canceled is applicable to many problem domains for that purpose.

The more the tool uses ASP, the more important it is that the problem encoding is efficient enough. A delay of one second may be too long in an interactive program. Long waits are inevitable if the problem is very difficult but many practical problem instances are small enough that we can call the solver without causing perceptible slowdown for the main program. This allows us to use ASP facilities to compute extra information on the problem instance and to present it to the user to guide her operations.

Answer set programming is a powerful tool for analyzing and solving difficult problems. It is already possible to embed an ASP solver into an application but a lot of work has to be done before we get to the point where an average computer programmer who is not an expert on logic can use ASP to solve practical problems in a robust way.

## References

1. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the Really Hard Problems Are. In: 12th International Joint Conference on Artificial Intelligence, pp. 331–337. Morgan Kaufmann, San Francisco (1991)
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. 22(3), 364–418 (1997)
3. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: 5th International Conference on Logic Programming, pp. 1070–1080. The MIT Press, Cambridge (1988)
4. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: 7th International Conference on Logic Programming, pp. 579–597. The MIT Press, Cambridge (1990)
5. Gomes, C.P., Selman, B., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reasoning 24, 67–100 (2000)
6. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K., Marek, V., Truszczyński, M., Warren, D. (eds.) The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer, Heidelberg (1998)

7. Mitchell, D.G., Selman, B., Levesque, H.J.: Hard and easy distributions for SAT problems. In: Rosenbloom, P., Szolovits, P. (eds.) 10th National Conference on Artificial Intelligence, pp. 459–465. AAAI Press, Menlo Park (1992)
8. Newell, A., Simon, H.A.: Computer science as empirical inquiry: symbols and search. Commun. ACM 19(3), 113–126 (1976)
9. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A system for answer set programming. In: 8th International Workshop on Non-Monotonic Reasoning (2000)
10. Schlipf, J.S.: The expressive powers of logic programming semantics. J. Comput. Syst. Sci. 51(1), 64–86 (1995)
11. Sutter, H., Alexandrescu, A.: C++ Coding Standards. Addison-Wesley, Reading (2005)
12. Syrjänen, T.: Logic Programs and Cardinality Constraints: Theory and Practice. Doctoral dissertation, TKK Dissertations in Information and Computer Science TKK-ICS-D12, Helsinki University of Technology, Faculty of Information and Natural Sciences, Department of Information and Computer Science, Espoo, Finland (2009)

# ASTREA: Answer Sets for a Trusted Reasoning Environment for Agents

Richard Watson[1] and Marina De Vos[2]

[1] Department of Computer Science
Texas Tech University
Lubbock, TX 79409, USA
richard.watson@ttu.edu

[2] Department of Computer Science
University of Bath
Bath BA2 7AY, UK
mdv@cs.bath.ac.uk

**Abstract.** In recent years, numerous papers have shown the power and flexibility of answer set programming (ASP) in the modeling of intelligent agents. This is not surprising since ASP was developed specifically for non-monotonic reasoning - including common-sense reasoning required in agent modeling. When dealing with multiple agents exchanging information, a common problem is dealing with conflicting information. As with humans, our intelligent agents may trust information from some agents more and than from others. In this paper, we present ASTREA, a methodology and framework for modeling multi-agent systems with trust. Starting from agents written in standard *AnsProlog* , we model the agent's knowledge, beliefs, reasoning capabilities and trust in other agents together with a conflict resolution strategy in CR-Prolog. The system is then able to advise the agent what information to take into account and what to discard.

## 1 Introduction

The world is full of situations where entities interact with each other and exchange information. Whether the interactions are between humans, machines, or some combination thereof, interaction is often necessary in order for each to achieve their goals. To model such situations it is therefore important to be able to represent and reason about the various agents and their interaction. One of the difficulties in modeling such agents is dealing with conflicting information. Conflicts may arise for a number of reasons. In extreme cases one agent may be intentionally lying to another agent, but it is more often the case that the agent is simply mistaken in its beliefs. An agent, modeling the behavior of a human, should reason based on assumptions just as a human does. Such assumptions may be wrong and, when passed to another agent, may disagree with information held by that agent or received from other agents. Conflicts may even arise when modeling electrical or mechanical systems. Such systems often rely on sensors to observe the world. If a sensor malfunctions, it may lead to a false belief about the true state of the world.

It is often difficult, if not impossible, to resolve conflicting information in a way that guarantees that the agent's beliefs are correct with respect to what is true in the world. This is a normal state of affairs when considering common-sense reasoning where the primary concern is that the agent's reasoning is rational, even if it turns out later that the agent was wrong. As an example, suppose a person asks two other people to answer a mathematics' problem. One of the persons asked is a first year math student and the other is a professor of mathematics. If the answers are different and the person has no other reason to disbelieve the professor, they should trust the professor over the student. This does not imply the professor was correct, but it is the rational choice under those circumstances.

There are three major topics one must consider when modeling agents: 1) the choice of language, 2) the methodology used for modeling, and 3) the framework under which the agents operate. In this paper we discuss our approach to agent modeling, ASTREA, which stands for "Answer Sets for a Trusted Reasoning Environment for Agents". As stated in the name, for the language we use Answer Set Programming (ASP). For conflict resolution, due to information arriving from different sources, we propose the use of CR-Prolog. Consistency restoring rules are used to indicate whether information supplied by a particular agent should be disbelieved in case of a conflict. Different conflict resolution strategies, based on the trust agents have in each other, can be encoded in CR-Prolog to provide that the most appropriate consistency restoring rules be used.

The rest of the paper is organized as follow. In Section 2 we provide a short introduction to multi-agent systems, answer set programming, and CR-Prolog. In Section 3, we introduce our ASTREA model. After highlighting the motivation behind our methodology, we propose the formal model of an ASTREA agent in Section 3.2. The mapping to CR-Prolog of the agent's knowledge base is detailed in Section 3.2. The different trust-based conflict resolution strategies are discussed in Section 3.3. and comments on future work.

## 2 Preliminaries

### 2.1 Multi-Agent Systems

The key contributors to a Multi-Agent System (MAS) are, naturally, agents. Despite the vast number of research papers published in this area, no general consensus exists about a definition of an agent. In this paper we take the definition from [33] which is an adaption of the one given in [34]:

> An *agent* is a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives.

A MAS contains a number of agents, each with their own objectives and goals, that affect the environment and collaborate to achieve their individual goals. A detailed discussion on MAS and its history can be found in [33]

Conceptually agents operate in an *observe-deliberate-act* loop [10,7]. In general the loop has the form:

1. observe and integrate observations, gather information (also referred to as beliefs)
2. consider the options available (also referred desires)
3. select goal (also referred to as intentions)
4. plan
5. execute

When implementing these agents one often finds that each phase is implemented by different components which share common knowledge [1,26].

This paper is concerned with the first step of the loop where the agent makes the observations and reflects upon the beliefs it holds about the state of the world. Among these observations are the things that the agent has been told by other agents. The agent must then decide what can consistently be believed from among those things it was told. This is, of course, not the easiest of tasks. Not only does one have to worry about the agent being told directly conflicting information, but information may also conflict indirectly.

Take, for example, an agent reasoning about cars and getting additional information from two other agents. Suppose the agent asks for information about a particular car and is told by one of his sources that the car has 4 doors and told by the other that the car is a Mazda Miata. While this information is not directly contradictory, if the agent knows that a Maita is a 2-door sports car then it knows that one of the two must be mistaken.

## 2.2 Answer Set Programming

To model the knowledge and beliefs of the individual agents we have opted for a form of logic programming, called Answer Set Programming (ASP) [23] developed by Michael Gelfond and his colleague Vladimir Lifschitz. Here we only present a short flavor of the language *AnsProlog*, and the interested reader is referred to [8] for in-depth coverage. *AnsProlog* and its extensions have demonstrated [7,9,13,18,24,12,16] that they are a good and useful tool in the domain of multi-agent systems.

*AnsProlog* is a knowledge representation language that allows the programmer to describe a problem and the requirements on the solutions in an intuitive way, rather than the algorithm to find the solutions to the problem. The basic components of the language are *atoms*; elements that can be assigned a truth value. An atom, $a$, can be negated using *classical negation* so creating the *literal* $\neg a$. An *inconsistency* occurs when both $a$ and $\neg a$ are true. A literal $l$ can be negated using *negation as failure* so creating the *extended literal*, *not* $l$. We say that *not* $l$ is true if we cannot find evidence supporting the truth of $l$. If $l$ is true then *not* $l$ is false and vice versa. Atoms, literals, and extended literals are used to create rules of the general form: $l$ :- $B$, *not* $C$., where $l$ is an literal and $B$ and $C$ are sets of literals. Intuitively, this means *if all elements of $B$ are known/true and no element of $C$ is known/true, then $l$ must be known/true.* We refer to $l$ as the head and $B \cup$ *not* $C$ as the body of the rule. Rules with empty body are called *facts. A* program in *AnsProlog* is a finite set of rules.

The semantics of *AnsProlog* is defined in terms of *answer sets*, i.e. consistent assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion. A program has zero or more answer sets, each corresponding to a solution.

When used as a knowledge representation and programming language, *AnsProlog* is enhanced to contain constraints (e.g. :- $b$, *not* $c$.), cardinality constraints [28] (*e.g.* $n[a_1, \ldots, a_k, \textit{not } b_1, \ldots \textit{not } b_l]m$) and weight constraints $L \leq \{a_1 = w_{a_1}, \ldots, a_k = w_{a_k}, \neg b_1 = w_{b_1}, \ldots, b_l = w_{b_l}\} \leq U$. The first type are rules with an empty head, stating that an answer set cannot meet the conditions given in the body. Cardinality constraints are a short hand notation a non-deterministic choice; for the constraint to hold a number between $n$ and $m$ of literals in the construct need to be contained in an answer set. Weight constraints are similar to cardinality constraints except that each literal is now given a weight. It is the addition of the weight of all the literals that are true which is taking into account. These additions are syntactic sugar and can be removed with linear, modular transformations (see [8]). Variables and predicated rules are also used and are handled, at the theoretical level and in most implementations, by instantiation (referred to as *grounding*).

Most ASP systems are composed of two processes: removing the variables from the program by instantiation with a *grounder*; and computing answer sets of the propositional program with an *answer set solver*. Common grounders are LPARSE [27] and GRINGO [22] while SMODELS [27], CLASP [21] and DLV [20] are frequently used solvers.

## 2.3 CR-Prolog

Programs do not always return answer sets. While this is exactly the behavior one would expect for certain problems (e.g. if the problem has no solution); in other situations having an answer is essential (e.g. a decision is required). A program fails to produce an answer set when the program is inherently contradictory. Removing or adding selected rules could resolve this contradiction; resulting in the program returning answer sets. While a learning system could learn the rules to add in certain system, it is more than often the designer's responsibility to specify these rules.

CR-Prolog [5,4] is a knowledge representation language which extends traditional answer set programming with consistency-restoring rules (cr-rules for short). These rules can be added to the program to automatically resolve the contradictions. The use of cr-rules allows for modeling situations that are unlikely, unusual, or avoided when possible. CR-Prolog has been successfully applied in areas like planning and diagnostic reasoning [6].

A CR-Prolog program[1] consists, apart from the normal *AnsProlog* rules, of consistency restoring rules of the form: $r : l \mathrel{+\!\!-} B, \mathit{not}\ C$ where $r$ is a label for the rule, $l$ is a literal, and both $B$ and $C$ are sets of literals. These rules have a similar meaning as normal *AnsProlog* rules except that they are only added when inconsistencies occur in the standard program. Given the nature of cr-rules, we of course want to add as few[2] of them as possible.

When there are different sets of cr-rules which could be added to resolve an inconsistency – i.e. various answer sets can be obtained – it is possible to add a preference to indicate which rules should be used. This can be done using atoms of the form $\mathit{prefer}(r1, r2)$ where $r1$ and $r2$ are the labels of cr-rules. When this atom is true, it indicates that no solutions using $r2$ should be considered unless no solution with $r1$ can be found. Adding this preference to the program also rules out solutions in which both $r1$ and $r2$ are used.

CRMODELS is the associated answer set solver for CR-Prolog programs. It is constructed on top of LPARSE and SMODELS [27]. The solver starts with checking if the normal part of the cr-program is consistent by calling SMODELS. If inconsistencies occur, CRMODELS iterative adds CR-rules on the basis of the preference relations until answer sets are found.

We will be using CR-Prolog to resolve inconsistencies that occur when beliefs from various agents are merged. Based on the trust an agent has in the other agents, it will be more likely to believe information from one agent than an other.

## 3 ASTREA

### 3.1 Motivation

As discussed in the previous section, an agent's knowledge and reasoning capabilities are modeled as a logic program under the Answer Set Semantics. The Answer Set Programming approaches used to model the knowledge of agents have been quite effective. In such approaches the agent's knowledge is modeled as rules in logic programming. The answer sets of the resulting program correspond to the possible sets of beliefs of a rational agent. This works well in a single agent environment, however, when there are multiple agents passing information problems can arise. The problem lies in the absence of any differentiation in how strongly the agent holds each of the beliefs. For illustration, consider the following well-known example. The agent is aware of the existence of a bird, Tweety. The agent also knows that, while there are exceptions, birds can normally fly. If this is the only information the agent has, the agent would rationally believe that Tweety could fly. Suppose the agent is then told by another agent that Tweety cannot fly. This contradicts the agent's beliefs. The question is how to resolve the contradiction. As a human reasoner, it is easy to see that either the agent was missing the knowledge that Tweety was an exception to the rule about

[1] In this paper we will restrict ourselves to non-disjunctive programs (i.e. rules only have one head atom).

[2] With respect to set theoretic inclusion, not cardinality.

flying or the second agent was wrong about Tweety not being able to fly. The contradiction would also be resolved if the agent was wrong about Tweety being a bird. This third possibility is not one that would be normally considered as it may be assumed that Tweety being a bird is not in question. The real question is one of which of the agent's beliefs is the agent sure of, and which are subject to doubt. The formalism presented in this paper will address the agent's confidence in its facts.

Contradiction between what an agent believes and what is it told by others is due to the fact that agents almost always have to reason with incomplete, and sometimes unreliable, information. Reasoning with such information leads to uncertainty. Uncertainty about an agent's knowledge can arise in several different ways. As mentioned before, one possibility is that the agent may be unsure of some of its own "facts". For example, in the situation above, if the statement was "The agent is aware of an animal, named Tweety, which the agent thinks is a bird" then there would be cause to be less than positive about Tweety actually being a bird. It is often the case that agents, human or otherwise, receive their information from sensors which are not infallible. Hence one often hears statements which begin with phrases such as "I thought I just saw...". A second reason for uncertainty is because the agent was using a defeasible rule and has incomplete information. In the situation above, if the agent has no reason to believe the bird is an exception to the rule about flying, the agent believes that Tweety can fly. The agent knows however that it is making an assumption that may be wrong. If on the other hand the agent knew that Tweety was not an exception, then it would be more certain. A third reason that can lead to uncertainty is that the agent may be reasoning using premises that are uncertain. It the agent above was reasoning about a cage for Tweety, they may reason that if a bird can fly then its cage needs to have a roof. As the agent is not certain that the bird can fly, they cannot be certain that the cage needs a roof. Finally, an agent may have uncertainty about a belief because it is something they were told by another agent.

## 3.2 ASTREA Agents

**Formalization.** An ASTREA framework consists of a number of agents $\mathcal{A}$. Each agent is a 6-tuple of the form $a = \langle id_a, \Pi_a, C_a, I_a, trust_a, \rho \rangle$. Here, the first element, $id_a$, is a unique name for the agent. We denote the set of all agent names as $A_{id}$. The second element, $\Pi_a$, is the agents knowledge and beliefs written as a standard *AnsProlog* program. The set of literals, $C_a$, are the facts within $\Pi_a$ that the agent is certain of. To help establish communication and make it possible to receive information we specify the agents from which agent $a$ can receive information. This is denoted as $I_a$ with $I_a \in 2^{A_{id} \setminus id_a}$. With the possibility of conflicting information, agents also contain a trust relation to specify their confidence in themselves and the agents they communicate with regarding the correctness of the beliefs they hold. The trust function is defined as follows: $trust_a : I_a \cup \{id_a\} \rightarrow \mathbb{N}$. The higher the number the more trusted the agent is.

In this paper we assume that agents assign single trust values to communicating agents but this could easily be extended to a function where trust is also assigned on the basis of topic.

Depending on the type of reasoning that is needed by the agent, the trust relation can be used in a variety of ways to inform the agent's reasoning. In this paper we look at three different trust strategies plus the situation in which the trust relationship is ignored. In the later case, we obtain all possible ways of obtaining a consistent set of beliefs. When the trust relationship is taken into account, we can take the views of the most trusted agent. In case of conflicts at the highest level, we obtain several possibilities and it will be up to the agent to decided upon one. Another option is to give each agent a vote and count the votes for each side of the contradiction and follow the majority. A similar strategy is to use a weighted voted mechanism on the trust level. More details will be given in the section where we model each strategy. Each agent within the ASTREA framework can have its own preference strategy. Its choice is denoted by $\rho$ with $\rho \in \{none, trusted, majority, weighted\text{-}majority\}$. The set of choices can easily be expanded as more trust relations are formalized.

We assume that the information the agent receives from other agents is in response to a request, $r$, that each request from a given agent has a unique id number, and that such information consists of a set of ground literals. We further assume that the literals in such a response come from a set of common literals. Formally, for an agent $a$, the set of all responses received by $a$ will be denoted by $R_a$. Elements of $R_a$ are 3-tuples of the form $r = \langle r_i, a_j, s\rangle$ where $r_i$ is the unique request id that is being responded to, $a_j$ is the name of the agent returning the response, and $s$ is the set of ground literals which make up the response.

In order to resolve inconsistencies, ASTREA agents consider what they have been told by other agents. In doing so the whole set of literals in a response will either be believed or rejected as a group. The reasoning behind rejecting all elements of a response as a group is that, since the set is in response to a request, one can assume that the literals in the set are likely related and therefore if one is rejected the rest are also suspect. Identifying conditions under which one might only partially reject a response are left for future research.

Before discussing the implementation of ASTREA agents, we will first give an example of a possible program, $\Pi$, that an agent may have and a response the agent may receive concerning the domain in question. The example is a precisely stated and expanded version of the Tweety bird problem mentioned earlier in the paper. This example will be used throughout the remainder of the paper to illustrate the use of ASTREA.

*Example 1 (A traditional method of modeling an agents knowledge in ASP).* The agent knows that birds can normally fly, but that there are exceptions. One such exception is penguins. The agent knows that penguins cannot fly. Another exception is wounded birds. Depending on the nature of the injury, a wounded bird may be able to fly or it may not. However, knowing a bird is wounded is enough reason for the agent to withhold judgment whether it can fly or not. The agent has knowledge of two birds, Tweety and Sally. The agent also believes that

Sally is wounded. There is a third bird, Sam, that the agent is currently unaware of. There are other agents who know about Sam, some of whom believe Sam is a penguin.

Using the traditional methods, this agent's knowledge can be represented by the following program $\Pi$:

$$\begin{array}{l} \mathit{fly}(X) \text{ :- } \mathit{bird}(X), \mathit{not}\ \mathit{ab}(X). \\ \mathit{bird}(X) \text{ :- } \mathit{penguin}(X). \\ \mathit{ab}(X) \text{ :- } \mathit{penguin}(X). \\ \neg fly(X) \text{ :- } \mathit{penguin}(X). \\ \mathit{ab}(X) \text{ :- } \mathit{wounded}(X). \\ \mathit{bird}(\mathit{tweety}). \\ \mathit{bird}(\mathit{sally}). \\ \mathit{wounded}(\mathit{sally}). \end{array}$$

Note that another standard method of representing the first rule above is:

$$\mathit{fly}(X) \text{ :- } \mathit{bird}(X), \mathit{not}\ \mathit{ab}(X), \mathit{not}\ \neg fly(X).$$

In this case the third rule, which states that penguins are abnormal, is not needed. Writing the first rule in this way is useful in the single agent case. If the agent knows a bird cannot fly, but does not have any knowledge about the bird being abnormal, the resulting program would be still be consistent with the alternate rule, but not with the one originally given. In a multi-agent situation however, the encoding we gave is preferable. With the information given in the story above, the agent would believe Tweety can fly. If told by another agent that Tweety could not fly, the addition would be consistent with the alternate rule. This is not the behavior we want - the agent should not automatically believe such an assertion without the other agent giving a reason to believe the bird was abnormal.

Given the story, a possible request the agent may make is to ask other agents for the information they have about birds. An example of a possibly response is

$$\langle r_i, a_j, \{\neg \mathit{fly}(\mathit{tweety}), \neg \mathit{fly}(\mathit{sam}), \mathit{bird}(\mathit{sam}), \mathit{penguin}(\mathit{sam})\}\rangle$$

with $r_i$ the request id and $a_j$ the name of the agent that responded. In this case, the information about Sam is new. It does not contradict anything the agent currently believes so the agent has no reason to disbelieve it. However, the information about Tweety contradicts the agent's belief that Tweety can fly. It is contradictions such as this that ASTREA is meant to resolve.

**Modeling Agent's Beliefs.** In the example above the domain was modeled in a standard way, as is usual in a single agent environment. In this work, however, we need to be able to resolve conflicts between multiple agents. There are several ways we could approach this problem. We could create either: a new semantics; a new methodology for modeling the domain; or a translation from models created using the traditional approach into new models which meets our needs. In this paper we opt for the third solution and present a translation. This will allow users to program in a formalism they are familiar with and to use existing off-the-shelf tools.

Before we present the translation we will start by highlighting the assumptions we make. First, for this paper we assume a single time instant and only the rules of the agent which concern the agents knowledge. That is, we do not consider rules regarding the actions of the agent. This assumption is not critical, however it does simplify the discussion in the paper. We also assume that the predicate *disbelieved* of arity 2 does not to exist in $\Pi$. If such a predicate did exist we could simply use a different, unused predicate in the translation. Again, this assumption just simplifies the discussion. Furthermore, we assume the agents themselves are consistent. In other words, the program representing the agent's knowledge using traditional methods has at least one answer set. This assumption is quite natural since we assume rational agents and it is irrational for an agent to have inconsistent beliefs.

With respect to the translation, we assume that, for the sake of grounding, the set of all constants used by the agents are known to each agent. This will simplify our discussion of the translation. In actual use one would need to modify the translated program if new constants were discovered in communications with other agents. Due to space consideration, rules which are not changed by the translation will be left ungrounded.

**Definition 1 (Translation $\Pi_a^T$)**
*Let $a = \langle id_a, \Pi_a, C_a, I_a, trust_a, \rho \rangle$ be an agent. Using $\Pi_a$ (grounded) and $C_a$ we create the translated program $\Pi_a^T$ as follows:*

*For each rule with an empty body (fact) in $\Pi$*

$$l.$$

*we add one the following to $\Pi_a^T$:*
*if $l \in C_a$ then add*

$$l. \tag{1}$$

*otherwise, two rules are added*

$$l \text{ :- } not \;\; disbelieved(x, id_a). \tag{2}$$

$$r(x, id_a) : disbelieved(x, id_a) \;+\!\!-\; . \tag{3}$$

*where $x$ is a unique identification number for each new rule.*
*For each rule in $\Pi$ with a non-empty body*

$$l_0 \text{ :- } l_1, \ldots, l_m, not \; l_{m+1}, \ldots, not \; l_n.$$

*(including rules with empty heads) add the rule to $\Pi_a^T$ and for each $m+1 \leq i \leq n$, add the following two rules to $\Pi_a^T$:*

$$l_i \;\text{:-}\; \mathit{disbelieved}(x, id_a). \tag{4}$$

$$r(x, id_a) : \mathit{disbelieved}(x, id_a) \;\text{+-}\; . \tag{5}$$

*where, as before, $x$ is a unique identification number for each new rule.*

In addition to translating the agents own knowledge, the responses received from other agents must also be translated. The translation of responses, $R_a^T$, is formed as follows:

**Definition 2 (Translation $R_a^T$)**
*Let $a$ be an agent and $R_a$ be the set of responses received by that agent from its requests. For each $r = \langle r_i, a_j, s\rangle \in R_a$ with $s = \{l_1, \ldots, l_n\}$ the rule*

$$r(r_i, a_j) : \mathit{disbelieved}(r_i, a_j) \;\text{+-}\; . \tag{6}$$

*is added to $R_a^T$ and for each*
*$1 \leq k \leq n$ the following rule is also added:*

$$l_k \;\text{:-}\; not \;\; \mathit{disbelieved}(r_i, a_j). \tag{7}$$

As in the case of the non-translated program, answer sets of the translated program correspond to possible sets of beliefs of the agent. It is possible that information from other agents in response to requests leads to conflicts. In this case CR-Prolog will automatically try to restore consistency by firing cr-rules.

Each cr-rule fired will add a *disbelieved* predicate to the answer set. If a *disbelieved* predicate added is one from a rule of type 2 then it indicates the agent chose to disbelieve an uncertain fact in the agent's original information. If it is from a rule of type 4, it indicates the agent chose to believe a literal occurring under default negation may be true, hence blocking that default rule. Finally, if the predicate is of the form $\mathit{disbelieved}(r, a)$, where $r$ is a request number and $a$ is the name of another agent, then the agent chose to disbelieve the entire response from agent $a$ to request $r$. By firing some subsets of the cr-rules, consistency will be restored. Notice that, since we assume the agent itself is consistent, in the worst case disbelieving all responses from other agents will necessarily result in consistency. There will often be more than one minimal subset of cr-rules by which consistency can be restored. The resulting answer sets are the possible beliefs of the agent. This will be illustrated by the following example.

*Example 2 (Translation and restoring consistency).* Recall the Tweety bird story from Example 1. The agent's knowledge was represented by the following program, $\Pi$:

$$\begin{aligned}
&\mathit{fly}(X) \;\text{:-}\; \mathit{bird}(X), not \;\; ab(X).\\
&\mathit{bird}(X) \;\text{:-}\; \mathit{penguin}(X).\\
&ab(X) \;\text{:-}\; \mathit{penguin}(X).
\end{aligned}$$

$$\begin{array}{l} \neg \mathit{fly}(X) \text{ :- } \mathit{penguin}(X). \\ \mathit{ab}(X) \text{ :- } \mathit{wounded}(X). \\ \mathit{bird}(\mathit{tweety}). \\ \mathit{bird}(\mathit{sally}). \\ \mathit{wounded}(\mathit{sally}). \end{array}$$

For this example, assume that the current agent is named $\mathit{agent}_0$ and the unique ids used in the rules start from 1. Furthermore, assume that $C_0 = \{\mathit{bird}(\mathit{tweety}), \mathit{bird}(\mathit{sally}), \mathit{wounded}(\mathit{sally})\}$. Using our translation, the resulting translated program, $\Pi_0^T$ is:

$$\begin{array}{l} \mathit{fly}(X) \text{ :- } \mathit{bird}(X), \mathit{not}\ \mathit{ab}(X). \\ \mathit{ab}(\mathit{tweety}) \text{ :- } \mathit{disbelieved}(1, \mathit{agent}_0). \\ r(1, \mathit{agent}_0) : \mathit{disbelieved}(1, \mathit{agent}_0). \\ \mathit{ab}(\mathit{sally}) \text{ :- } \mathit{disbelieved}(2, \mathit{agent}_0). \\ r(2, \mathit{agent}_0) : \mathit{disbelieved}(2, \mathit{agent}_0). \\ \mathit{ab}(\mathit{sam}) \text{ :- } \mathit{disbelieved}(3, \mathit{agent}_0). \\ r(3, \mathit{agent}_0) : \mathit{disbelieved}(3, \mathit{agent}_0). \\ \mathit{bird}(X) \text{ :- } \mathit{penguin}(X). \\ \mathit{ab}(X) \text{ :- } \mathit{penguin}(X). \\ \neg \mathit{fly}(X) \text{ :- } \mathit{penguin}(X). \\ \mathit{ab}(X) \text{ :- } \mathit{wounded}(X). \\ \mathit{bird}(\mathit{tweety}). \\ \mathit{bird}(\mathit{sally}). \\ \mathit{wounded}(\mathit{sally}). \end{array}$$

As mentioned, for space rules were kept unground when possible.

If then, in response to a request with id 4, the agent receives information from another agent, $\mathit{agent}_1$, stating that they believe that Tweety cannot fly but Sally can (i.e. $R_0 = \{\langle 4, \mathit{agent}_1, \{\neg \mathit{fly}(\mathit{tweety}), \mathit{fly}(\mathit{sally})\}\rangle\}$), then $R_0^T$ is:

$$\begin{array}{l} \neg \mathit{fly}(\mathit{tweety}) \text{ :- } \mathit{not}\ \mathit{disbelieved}(4, \mathit{agent}_1). \\ \mathit{fly}(\mathit{sally}) \text{ :- } \mathit{not}\ \mathit{disbelieved}(4, \mathit{agent}_1). \\ r(4, \mathit{agent}_1) : \mathit{disbelieved}(4, \mathit{agent}_1) \text{ +- }. \end{array}$$

If not for the cr-rules, the resulting program, $\Pi_0^T \cup R_0^T$ would entail both $\mathit{fly}(\mathit{tweety})$ and $\neg \mathit{fly}(\mathit{tweety})$ and hence it would be inconsistent. However, using CR-Prolog, one of the two cr-rules will fire, adding either $\mathit{disbelieved}(1, \mathit{agent}_0)$ or $\mathit{disbelieved}(4, \mathit{agent}_1)$. Firing the first cr-rule and adding $\mathit{disbelieved}(1, \mathit{agent}_0)$ corresponds to the agent deciding that they may be wrong about Tweety not being abnormal with respect to flying. It is then safe for the agent to believe what they were told by the other agent; that Tweety could not fly and Sally could.

If the second cr-rule was fired instead, it would correspond to the case when the agent chooses to disbelieve what they had been told by the other agent. As a result, the agent would go on believing that Tweety could fly and would hold no belief one way or the other as to Sally's ability to fly. Even though the information about Sally does not conflict with the beliefs of the agent, the whole response is rejected. Notice that if a response from a different agent had stated that Sam was a penguin and could not fly, then the agent would believe the response in either case as it would not cause any contradictions.

Without adding any additional code, the answer sets returned will correspond to all possible minimal ways of resolving the contradiction. There is no preference given between them. This is the behavior the agent will have if the agent's choice of preference strategy, $\rho$, equals *none*.

In general, if $\rho = none$ the agent computes its possible sets of beliefs by using CR-Prolog to find the answer sets of $\Pi_a^T \cup R_a^T$. Each answer set is a possible set of beliefs of the agent. In the subsections that follow the other trust relations will be shown.

## 3.3 Building on Trust

Recall from Section 3.2 that each agent has its own trust function. Two of the four trust relations given below require that the values of this trust function be encoded. If $a = \langle id_a, \Pi_a, C_a, I_a, trust_a, \rho\rangle$ is an agent where $id_a = agent_0$ and $I_a = \{agent_1, \ldots, agent_n\}$ then we define $T_a$ as the program consisting of all facts of the form $trust(agent_i, t)$. where $0 \leq i \leq n$ and $trust_a(agent_i) = t$. Recall that the agent will quantify the trust it has in its own ability.

**Distrust Lowest Trust Levels First.** The first preference relation presented here is one in which the agent prefers to disbelieve information from agents it trusts less when they are in conflict with information from agents it trusts more. This corresponds to the agent having trust strategy $\rho = trusted$. This strategy is encoded using the following program, denoted by $Tr$:

$$\begin{aligned}
prefer(r(N1, A1), r(N2, A2)) \;\; &\text{:-} \;\; trust(A1, T1), trust(A2, T2), T1 < T2,\\
&\qquad disbelieved(N1, A1), not\ \ disbelieved(N2, A2).\\
prefer(r(N1, A1), r(N2, A2)) \;\; &\text{:-} \;\; trust(A1, T1), trust(A2, T2), T1 < T2,\\
&\qquad not\ \ disbelieved(N1, A1), disbelieved(N2, A2).
\end{aligned}$$

In order to compute beliefs of an agent using this strategy, CR-Prolog is used on the program $\Pi_a^T \cup R_a^T \cup T_a \cup Tr$.

*Example 3 (Trust Strategy "Trusted").* Consider an agent, $agent_0$, $\Pi_0^T$ as in example 2, and $T_0$ as follows:

$$\begin{aligned}
&trust(agent_0, 2).\\
&trust(agent_1, 3).\\
&trust(agent_2, 1).
\end{aligned}$$

Notice this means that the agent trusts $agent_1$ more then they trust themselves, but $agent_2$ least of all. Suppose, in response to a request, the agent had $R_0^T =$

$$\begin{aligned}
&\neg \mathit{fly}(\mathit{tweety}) \text{ :- } \mathit{not}\ \mathit{disbelieved}(4, \mathit{agent}_1).\\
&r(4, \mathit{agent}_1) : \mathit{disbelieved}(4, \mathit{agent}_1) \text{ +- }.\\
&\mathit{fly}(\mathit{tweety}) \text{ :- } \mathit{not}\ \mathit{disbelieved}(4, \mathit{agent}_2).\\
&r(4, \mathit{agent}_2) : \mathit{disbelieved}(4, \mathit{agent}_2) \text{ +- }.
\end{aligned}$$

In other words, in their responses $agent_1$ said Tweety cannot fly but $agent_2$ says Tweety can.

When the resulting program, $\Pi_0^T \cup R_0^T \cup T_0 \cup Tr$, is run using CR-Prolog there is only one answer set. The answer set contains $\neg \mathit{fly}(\mathit{tweety})$ and both $\mathit{disbelieved}(1, \mathit{agent}_0)$ and $\mathit{disbelieved}(4, \mathit{agent}_2)$. This is the desired result as $agent_1$ said Tweety could not fly and $agent_1$ is the most trusted agent.

If there is a conflict between two equally trusted agents and there is no agent with a higher trust level that resolves the same conflict then there will be multiple answer sets. Note that, if there are multiple requests, an agent may be believed on some of its responses and disbelieved on others.

**Trust the Majority.** Another trust relation an agent might use is to choose to trust the majority ($\rho = \mathit{majority}$). In this case there is no need to specify specific trust levels. In case of a conflict the agent minimizes the number of responses it needs to disbelieve, hence believing the majority. As it is very difficult to tell which *disbelieved* statements within $\Pi_a^T$ correspond to a given response, if anything is disbelieved within $\Pi_a^T$ then the agent counts itself as one response disbelieved. In order to implement this relation, the following program, $Mv$ is used:

$$\begin{aligned}
\mathit{agentvote} &\text{ :- } \mathit{disbelieved}(N, \mathit{id}_a).\\
\mathit{count}(C+V) &\text{ :- } C\{\mathit{disbelieved}(J,K) : \mathit{request}(J) : \mathit{agent}(K)\}C,\\
&\qquad V\{\mathit{agentvote}\}V.\\
\mathit{ac}(C) : \mathit{countallowed}(C) &\text{ +- }.\\
\mathit{prefer}(\mathit{ac}(C1), \mathit{ac}(C2)) &\text{ :- } C1 < C2.\\
\neg \mathit{count}(C) &\text{ :- } \mathit{not}\ \mathit{countallowed}(C).
\end{aligned}$$

Notice that here we also need predicates $\mathit{request}(J)$ and $\mathit{agent}(K)$ for each request id, $J$, and agent name, $K$, other than $id_a$. The program works by forcing an inconsistency unless the number of responses disbelieved (plus one if the agent has to disbelieve anything in its own knowledge and beliefs) is not equal to the allowed count. The allowed count is controlled by a cr-rule with a preference for answer sets with lower counts. As a result the agent gets the answer set(s) with the minimum count. As the trust function is not used here, the beliefs are computed by passing $\Pi_0^T \cup R_0^T \cup Mv$ to CR-Prolog.

*Example 4 (Trust Strategy "Majority").* Consider an agent, $agent_0$, with $\Pi_0^T$ as in Example 2, and $R_0^T$ as in example 3. In this case, since both the agent itself and $agent_2$ believe the Tweety can fly and only $agent_1$ believes that Tweety cannot, there is only one answer set. In that answer set Tweety can fly, as that was the majority.

It is interesting to note, however, that if $agent_1$'s response had been that Tweety cannot fly and furthermore Tweety is a penguin, then there would have been two answer sets. One, as before, in which Tweety can fly, and one in which Tweety is a penguin and hence cannot fly. This is due to the fact that $agent_1$'s beliefs were only in conflict with the agents beliefs in the first case because $agent_1$ said that Tweety could not fly but did not give any reason for $agent_0$ to believe Tweety was abnormal. By adding that Tweety was a penguin, $agent_1$'s response is no longer in conflict with the beliefs of $agent_0$. This is because, if $agent_0$ accepts what $agent_1$ replied, it assumes it was simply missing the information that Tweety was a penguin and therefore an exception to the rule about flying.

**A Weighted Majority Wins.** The third trust relation is also based on a majority, however this time, agents with higher trust levels carry more weight than those with lower trust levels when deciding the majority. It can be viewed as a voting system where some agents may get more votes than others. This is the strategy used when $\rho = \mathit{weighted\text{-}majority}$. For this, the program, $Wm$ is:

$$\begin{array}{rcl}
vote(R,A,V) & :- & disbelieved(R,A), trust(A,V). \\
\#weight\ vote(J,K,L) & = & L. \\
agentvote & :- & disbelieved(N, id_a). \\
\#weight\ trust(id_a,T) & = & T. \\
\#weight\ agentvote & = & trust(id_a,T). \\
count(C+V) & :- & C\{vote(J,K,L) : request(J) : \\
 & & agent(K) : numvotes(L)\}C, \\
 & & V\{agentvote\}V. \\
ac(C) : countallowed(C) & +- & . \\
prefer(ac(C1), ac(C2)) & :- & C1 < C2. \\
\neg count(C) & :- & not\ countallowed(C).
\end{array}$$

The code is similar to the code for the previous relation with a few changes. First, and most importantly, the trust level returned by the agent's trust function for each agent is the number of votes that agent receives. Next, the predicate $numvotes(L)$ must exist for each value $L$ in the range of the agent's trust function. Finally, weight rules from SMODELS are used to assign weights to each vote. The program works similarly to that in the previous example except here total number of votes is minimized rather than simply the number of responses disbelieved. As we use the trust function in this relation, the CR-Prolog program used is $\Pi_0^T \cup R_0^T \cup T_a \cup Wm$.

*Example 5 (Trust Strategy "Weighted-Majority").* Consider an agent, $agent_0$, with $\Pi_0^T$ as in example 2, and both $R_a^T$ and $T_a$ as in example 3. The agent and $agent_2$ both agree that Tweety can fly. In order to accept this belief, $agent_1$ would have to be disbelieved. Since $agent_1$ has 3 votes, that is 3 votes against this answer. $agent_1$ says Tweety cannot fly. Accepting that response would require that both the agent itself and $agent_2$ be disbelieved. Together they also have 3 votes. There is a tie so there are two answer sets, one in which the agent believes Tweety can fly, one in which the agent believes Tweety cannot.

If, as in the previous example, $agent_1$ had added that Tweety was a penguin then in the case where Tweety cannot fly, the agent is not in conflict so there is only one vote against. Therefore there is only one answer set, the one in which Tweety cannot fly.

## 4 Related Work

The ASP community has a vast body of research in the area of representing either preferences, order, priority, or trust in answer set programs. Some researchers allow preferences expressed on the level of atoms, within the program [11], or on the Herbrand base [29], while others define them through rules [14,17]. In some systems the preferences are used to select the most appropriate solution after the models are computed [31], while others use them for computing their solutions [19]. It is beyond the scope of this paper to give a detailed overview of all those systems. In CR-Prolog, preference is based on rules. Because of our transformation of the initial *AnsProlog* program, the rules on which the preferences are defined are facts. So one could argue that due to construction we have obtained an ordering on atoms. Each of these atoms is a *disbelieved* atom indicating that all information received from a certain agent about a certain request is ignored. The preferences used in CR-Prolog are tightly linked with its consistency restoring mechanism. While we would have been able to encode the preference in other formalisms using different transformations, we believe that the CR-Prolog's philosophy, believe everything unless you have no other option, fitted in the most natural way.

One of the ways of dealing with contradiction between information supplied from agents that we presented used a voting mechanism. To our knowledge, [25] is the only other paper to describe voting mechanisms in answer set programming. In that paper voters provide a partial preference relation over the set of candidates and three voting mechanism are provided: Borda, plurality, Condorcet. In our paper, voting only takes place when a contradiction takes place and votes are distributed on either side of the contradiction. It is either a straight majority vote or the preference relation is used to assign weighted votes.

Multi-agent systems is a very active research domain, but most of research focuses on agents' actions. While there is research on modeling agents' beliefs [32], very little research has been taken place so far in updating the beliefs of the agents.

[15] proposes an agent architecture based on intelligent logic agents. Abduction is used to model agent reasoning. The underlying language is LAILA, a logic-based language which allows one to express intra-agent reasoning and inter-agent coordination coordinating logic based agents. No preferences between sources can be expressed. Using CR-Prolog together with our *disbelieved* atoms mimics up to a certain point abductive behavior, as we try to deduce who is or could be responsible for the contradiction in the belief set.

In [30], the authors use program composition to model agent cooperation and information sharing. We believe that is not a viable way in large multi-agent systems due to the size of the program. Furthermore, one could imagine agents being reluctant to share all the information (including possible private information) with others.

In the Minerva architecture [26], the authors build their agents out of sub-agents that work on a common knowledge base written as a MDLP (Multi-Dimensional Logic Programs) which is an extension of Dynamic Logic Programming. Our agents do not work with a common knowledge base; each agent decides what she wants to keep private or make available. Minerva does not restrict itself to modeling the beliefs of agents, but allows for full BDI-agents that can plan towards a certain goal. It would be interesting to see if this can also be incorporated in our system.

## 5 Conclusions and Future Work

In this paper we proposed a new methodology for modeling and reasoning about multi-agents system in which the agents exchange information that is possibly contradictory. The information processing part of the agents uses answer set programming to describe the current knowledge, beliefs and capabilities. When new data is added as a result of a series of requests, the program is translated into CR-Prolog program that resolves possible inconsistencies. Our framework currently offers four different strategies to select the most appropriate conflict resolution recommendations. Two of these are based on the trust agent's place in themselves and communicating agents when it comes to supplying correct information.

One area for future work concerns knowledge versus belief. With new and possibly contradicting information coming from other agents and an ever-changing environment, it is important that agents make a distinction between knowledge, pieces of information that cannot be refuted, and beliefs, information that can be changed over time. This was dealt with to some extent in the formalism presented in that the agent can differentiate between facts it is certain of and those it is not. But what about information passed in a response? A responding agent could separate their response into two parts; the part it knows to be true and the part that it only believes. If we assume that if an agent says it "knows" a literal then the literal must be true in the real world, then the receiving agent could always accept the known part of the response even if it rejected the believed part. The next question that arises is "how does an agent tell the difference between what it knows and what it believes?" Answering these questions and incorporating the answers into ASTREA could result in a more robust system. We have some preliminary work on these questions. Our current approach involves a larger, more complex translation of the agent's knowledge. It is not presented here due to space constraints.

A somewhat related topic is the question of when can an agent accept part of a response while rejecting the rest. When the parts in question are beliefs this is a much more difficult question. In our examples in this paper, we were concerned with birds and their ability to fly. As a human reasoner we can see that a response which contained information about two different birds could be split. The question is how to incorporate that reasoning ability in the agents. One approach may be to limit requests to try to prevent the responses from containing information that was not closely related. That approach, however, seems to be overly restrictive. For us partial acceptance is an open question.

Another open question concerns incorporating knowledge from responses into the agent's facts. At present new information gained from other agents as a result of a request for information is not added to the agent's knowledge base for future use. Neither is the program updated when the agent finds out that some of its beliefs are false. CR-Prolog guarantees that if disbelieved information is removed, a consistent program remains. However, difficulties arise when CR-Prolog returns with several equally preferred ways of restoring consistency. In the future, we plan to look at different update strategies. It is, of course, theoretically possible to continue to store responses and reason with them as a separate component. In human reasoning however, there comes a time in which the agent moves beliefs from "things they were told" into "things they believe" and no longer try to remember where they heard them. Deciding when and how to do this in ASTREA agents is an interesting question for future consideration. In [2], the authors present dynamic logic programs to capture a knowledge base that changes of over time. EvoLP [3] is an extension of this language designed for multi-agent system. While both deal with inconsistencies, neither consider trust to determine the update. It will be interesting to see if both we can combine updates with trust.

At present agents only communicate facts. One could envisage situations in which agents want to communicate rules as well. Using CR-Prolog, it should be easy to incorporate this. As mentioned earlier, preference relations could be expanded to specify preferences not only on the basis of the agent but also on the topic.

In this paper CR-Prolog was only used as a tool. The research did, however, inspire several ideas about potential modifications to CR-Prolog. While the current version of CR-Prolog can use CLASP as its solver, CR-Prolog itself is tightly coupled to LPARSE. As a result, some new constructs available in CLASP, such as aggregates, cannot be used. Such aggregates would have been useful in encoding some of the trust relations presented in this paper. A new version of CR-Prolog, which was either based on GRINGO or which allowed the easy addition of new language feature, is a possible area for future research.

Another direction for further research on CR-Prolog concerns preferences. One of the trust relations presented in this paper was one in which, in case of conflict, you prefer to disbelieve agents with lower trust levels over those with higher trust levels. An obvious way to model this would be to have the rule

$$prefer(r(N1, A1), r(N2, A2)) \text{ :- } trust(A1, T1), trust(A2, T2), T1 < T2.$$

where $A1$ and $A2$ are agents identifiers, $N1$ and $N2$ are either unique agent rule identifiers or unique request identifiers, and $T1$ and $T2$ are trust levels. Unfortunately, this would not give the desired results. In the current version of CR-Prolog, given two cr-rules, $r1$ and $r2$, if rule $r1$ is preferred to rule $r2$ then if there is a model which uses $r1$ but not $r2$ it is preferred to a model which uses $r2$ but not $r1$. However, it also disallows models which use both $r1$ and $r2$. Suppose there are three agents, $a1, a2$, and $a3$ with $a1$ being the least trusted and $a3$ being most trusted. For the desired trust relation, disbelieving $a1$ is preferred to disbelieving $a2$, but it may be necessary to disbelieve both if they are both in conflict with $a3$. The ability to use different preference relations within CR-Prolog would enhance its abilities and would allow for more elegant problem solutions in some cases.

## References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The socs computational logic approach to the specification and verification of agent societies. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 314–339. Springer, Heidelberg (2005)
2. Alferes, J.J., Banti, F., Brogi, A., Leite, J.A.: Semantics for dynamic logic programming: A principle-based approach. In: Lifschitz, V., Niemel, I. (eds.) LPNMR 2003. LNCS, vol. 2923, pp. 8–20. Springer, Heidelberg (2003)
3. Alferes, J., Brogi, A., Leite, J., Pereira, L.: Logic programming for evolving agents. In: Klusch, M., Zhang, S.-W., Ossowski, S., Laamanen, H. (eds.) CIA 2003. LNCS (LNAI), vol. 2782, pp. 281–297. Springer, Heidelberg (2003)
4. Balduccini, M.: Answer set based design of highly autonomous, rational agents. Phd thesis, Texas Tech University (December 2005)
5. Balduccini, M., Gelfond, M.: Logic programs with consistency-restoring rules. In: AAAI Spring 2003 Symposium, pp. 9–18 (2003)
6. Balduccini, M., Gelfond, M.: The aaa architecture: An overview. In: AAAI 2008 Spring Symposium on Architectures for Intelligent Theory-Based Agents (2008)
7. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Minker, J. (ed.) Logic Based Artificial Intelligence, pp. 257–279. Kluwer Academic Publishers, Dordrecht (2000)
8. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge Press, Cambridge (2003)
9. Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Logic-based artificial intelligence, pp. 257–279. Kluwer Academic Publishers, Dordrecht (2000)
10. Bratman, M.E.: What is intention? In: Cohen, P.R., Morgan, J.L., Pollack, M.E. (eds.) Intentions in Communication, pp. 15–32. MIT Press, Cambridge (1990)

11. Brewka, G., Niemelä, I., Truszczynski, M.: Answer set programming. In: International Joint Conference on Artificial Intelligence (JCAI 2003). Morgan Kaufmann, San Francisco (2003)
12. Buccafurri, F., Caminiti, G.: A social semantics for multi-agent systems. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 317–329. Springer, Heidelberg (2005)
13. Buccafurri, F., Gottlob, G.: Multiagent compromises, joint fixpoints, and stable models. In: Kowalski, R.A. (ed.) Computational Logic: Logic Programming and Beyond. LNCS (LNAI), vol. 2407, pp. 561–585. Springer, Heidelberg (2002)
14. Buccafurri, F., Leone, N., Rullo, P.: Disjunctive ordered logic: Semantics and expressiveness. In: Cohn, A.G., Schubert, L.K., Shapiro, S.C. (eds.) Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning, June 1998, pp. 418–431. Morgan Kaufmann, Trento (1998)
15. Ciampolini, A., Lamma, E., Mello, P., Toni, F., Torroni, P.: Cooperation and competition in alias: A logic framework for agents that negotiate. Annals of Mathematics and Artificial Intelligence 37, 65–91 (2003), doi:10.1023/A:1020259411066
16. Cliffe, O., De Vos, M., Padget, J.: Specifying and analysing agent-based social institutions using answer set programming. In: Boissier, O., Padget, J., Dignum, V., Lindemann, G., Matson, E., Ossowski, S., Sichman, J.S., Vázquez-Salceda, J. (eds.) ANIREM 2005 and OOOP 2005. LNCS (LNAI), vol. 3913, pp. 99–113. Springer, Heidelberg (2006)
17. De Vos, M., Cliffe, O., Watson, R., Crick, T., Padget, J., Needham, J., Brain, M.: T-LAIMA: Answer Set Programming for Modelling Agents with Trust. In: European Workshop on Multi-Agent Systems (EUMAS 2005), pp. 126–136 (December 2005)
18. De Vos, M., Vermeir, D.: Extending Answer Sets for Logic Programming Agents. Annals of Mathematics and Artifical Intelligence 42(1-3), 103–139 (2004); Special Issue on Computational Logic in Multi-Agent Systems
19. Delgrande, J., Schaub, T., Tompits, H.: Logic programs with compiled preferences. In: Horn, W. (ed.) European Conference on Artficial Intelligence, pp. 392–398. IOS Press, Amsterdam (2000)
20. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The KR system dlv: Progress report, comparisons and benchmarks. In: Cohn, A.G., Schubert, L., Shapiro, S.C. (eds.) KR1998: Principles of Knowledge Representation and Reasoning, pp. 406–417. Morgan Kaufmann, San Francisco (1998)
21. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: M. Veloso, editor, Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 386–392. AAAI Press/The MIT Press (2007), http://www.ijcai.org/papers07/contents.php.
22. Gebser, M., Schaub, T., Thiele, S.: GrinGo: A New Grounder for Answer Set Programming. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
23. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3-4), 365–386 (1991)
24. Gelfond, M., Morales, R.: Encoding conformant planning in a-prolog. In: DRT 2004 (2004)
25. Konczak, K.: Voting theory in answer set programming. In: Fink, M., Tompits, H., Woltran, S. (eds.) Proceedings of the Twentieth Workshop on Logic Programmin (WLP 2006). Number INFSYS RR-1843-06-02 in Technical Report Series, pp. 45–53. Technische Universität Wien (2006)

26. Leite, J.A., Alferes, J.J., Pereira, L.M.: Minerva - a dynamic logic programming agent architecture. In: Intelligent Agents VIII. LNCS (LNAI), vol. 2002, pp. 141–157. Springer, Heidelberg (2002)
27. Niemelä, I., Simons, P.: Smodels: An implementation of the stable model and well-founded semantics for normal LP. In: Dix, J., Furbach, U., Nerode, A. (eds.) LPNMR 1997. LNCS (LNAI), vol. 1265, pp. 420–429. Springer, Berlin (1997)
28. Niemelä, I., Simons, P.: Extending the smodels system with cardinality and weight constraints. In: Logic-Based Artificial Intelligence, pp. 491–521. Kluwer Academic Publishers, Dordrecht (2000)
29. Sakama, C., Inoue, K.: Representing Priorities in Logic Programs. In: Maher, M. (ed.) Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, September 2–6, pp. 82–96. MIT Press, Cambridge (1996)
30. Sakama, C., Inoue, K.: Coordination between Logical Agents. In: ao Leite, J., Torroni, P. (eds.) Pre=Proceedings of CLIMI V: Computation logic in multi-agent systems, Lisbon, Portugal, September29–30, pp. 96–113 (2004)
31. Van Nieuwenborgh, D., Vermeir, D.: Preferred answer sets for ordered logic programs. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 432–443. Springer, Heidelberg (2002)
32. Witteveen, C., Brewka, G.: Skeptical reason maintenance and belief revision. Artificial Intelligence 61(1), 1–36 (1993)
33. Wooldridge, M.: An introduction to multiagent systems. Wiley, Chichester (2002) ISBN: 0 47149691X
34. Wooldridge, M., Jennings, N.R.: Intelligent agents: theory and practice. The Knowledge Engineering Review 10(02), 115–152 (1995)

# Tea Times with Gelfond

Veena S. Mellarkod

The MathWorks Inc.,
3 Apple Hill Dr., Natick, MA, USA
veena.mellarkod@mathworks.com

## *С Днем Рождения, Dr.Gelfond, Я желаю тебе долгих лет жизни*

I must thank Marcello Balduccini and Tran Cao Son for giving me this opportunity to revisit and recount my education with Prof. Michael Gelfond. Not everyone is fortunate to have a dedicated teacher at any phase of life. In this regard, I consider myself to be among the luckiest few to have learned from someone who cares so much about imparting knowledge as part of their students' education. It has been a couple of years since I have left the Knowledge Representation lab of Texas Tech, and yet it seems not too long ago that I was learning *about* Prof. Gelfond. It certainly was well before I started learning from him. The ten best years of my life were spent immersed in learning from THE TEACHER among teachers, Prof. Michael Gelfond. Writing in celebration of his 65th birthday is an honor and a fitting tribute to the times well spent.

## My Guru

*Maatha, pitha, Guru, Daivam*: Mother, father, teacher, and God; this is precisely the order of importance and honor one must confer, according to Vedic philosophy. Giving parents the highest regards can perhaps be rationalized because the individual owes his or her life to them. Giving *guru* the next place is rationalized because it is the *guru* that guides the individual towards the path of enlightenment. Growing up in India, I was taught to practice this ideal throughout my life. One can readily follow and offer respects to one's parents. What about *guru*? Vedic times dictated that an individual is sent to study with a *guru* for an extended time – often years – to be considered a learned one. Where does one get an opportunity or time to study with and learn from a single *guru* these days? Naturally, I wondered whether the concept of *guru* is still relevant for our times and, if it were, when I would come across a *guru* worthy of complete devotion. I consider it my *good karma* for I did find a modern-day *guru* – one who taught me everything that I know today about intelligent computing and algorithm development – over many cups of tea and home-made cakes, no less. Finding a *guru* is a treasure in itself. But finding a *guru patni* (teacher's wife) who cares for the students as well as the *guru* must truly be a blessing. Our celebration of Michael and Lara on this momentous occasion is no less than the celebration of teachers everywhere who serve as role models in their students' lives.

## Learning from Dr. Gelfond

My association with Dr. Gelfond goes back to the turn of the century in 1999. I was a naïve graduate student freshly admitted to the computer science department at University of Texas at El Paso. Dr. Gelfond taught one of the courses in my first semester, Algorithms and Data Structures. Despite a lack of background in computer science – my undergraduate training was in chemical engineering – Dr. Gelfond's articulation of the technical concepts clarified my numerous doubts about the subject. Little did I realize then that this would be a continuous learning experience in the years ahead! Dr. Gelfond would also engage in lively discussions during a seminar series organized on behalf of Knowledge Representation lab at The University of Texas at El Paso. It was his course on the *History of Logic Programming* and teaching style that inspired me to pursue a challenging topic for my master's thesis instead of finishing up the coursework to get a degree. Rather than limiting the discussion to technical developments, Dr. Gelfond took us to the times and circumstances faced by those great minds that put forward the many advances in logic programming. His desire and enthusiasm to instill a thirst for knowledge in his students is contagious enough to drive a chemical engineer towards pursuing a PhD in computer science.

## Tea Times at Tech

Prof. Michael Gelfond moved to Texas Tech University soon after and I, along with few other students, followed him to TTU with a single-minded interest in learning from him. The Knowledge Representation lab took roots once again at Texas Tech and lively debates ensued over a diverse array of topics such as Artificial Intelligence (including Spielberg's movie of the same name), logic programming, philosophers of yore, global cultures, origin of religions, and existence of God or lack thereof. Dr. Gelfond would often walk over to the lab with a non-descript green cup filled with tea and ask in general, "what's up?"; the lab members would arrange themselves in a circle and chat over cups of their preferred beverage, usually coffee, tea, or water. The participants of these debates would often join in or leave as they please but the dynamic discussion progressed regardless. Our interaction with Dr. Gelfond was full of such intellectual discussions in the lab, which I would call *tea times with Gelfond.*

## A True Advisor

Beyond being an exceptional teacher and outstanding researcher, Dr. Gelfond is truly an advisor and a friend for life. A kind human being, he taught me many qualities one should imbibe in one's personality. He taught me the necessity of discipline and patience, the importance of being kind and rational, the art of listening and the virtues of a good human being. I will forever be grateful for his guidance in making me the person I am today. The fable of eka Lavya from Indian epic Mahabharata celebrates *guru* drOnachArya for famously inspiring students to learn his art. As many of his current and former students agree, Dr. Michael Gelfond is worth the devotion of many such eka Lavyas. Thank you, Dr. Gelfond, for all the time you put into shaping us to be the individuals we are today. Hope you will continue to inspire many more students to come.

# Author Index