Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Stepwise Debugging in Answer-Set Programming:

Theoretical Foundations and Practical Realisation

DISSERTATION

submitted in partial fulfilment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Jörg Pührer
Registration Number 0055983

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: a.o. Univ. Prof. Hans Tompits

The dissertation has been reviewed by:

(a.o. Univ. Prof. Hans Tompits) (Dr. Marina De Vos)

Wien, 1.11.2014

(Jörg Pührer)

Erklärung zur Verfassung der Arbeit

Jörg Pührer
Ploßstraße 50, D-04347 Leipzig

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit—einschließlich Tabellen, Karten und Abbildungen—die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

To my parents, Luise and Sepp.

Acknowledgements

First of all, I want to thank my dissertation advisor Hans Tompits for his continuous support and many years of collaboration. Hans taught me a lot about scientific writing, formal clarity, and academic activities in general. Moreover, he gave me the opportunity to work in science and the freedom to follow my own ideas in doing so. I am grateful for his personal effort in mentoring this thesis as well as for his strong commitment to all of our other projects which often led to open-ended discussions on the respective topic as well as on the best and the worst of all worlds.

I also want to thank Johannes Oetsch with whom it has been a pleasure to work with. He has been a great colleague with brilliant ideas and a good sense of humour. Moreover, I wish to thank all of my co-authors as well as my colleagues in Vienna and Leipzig.

My work on the topics of this thesis was partially supported by the Austrian Science Fund (FWF) under project P21698. Besides paying my salary, the project allowed me to visit and collaborate with renowned experts from the field such as Marina De Vos, Esra Erdem, Tomi Janhunen, Ilkka Niemelä, Francesco Ricca, and Torsten Schaub from which I have profited a lot.

Being creative is much easier if one can rely on a supportive personal environment. As this is the case for me I want to thank my family, my parents, Heidi, Johanna, Klaus, Vroni, Willi, as well as my girlfriend Tinka for being there.

Abstract

Answer-set programming (ASP) is a paradigm for declarative problem solving that is popular amongst researchers in artificial intelligence and knowledge representation. Yet it is rarely used by software engineers outside academia so far. Arguably, one obstacle preventing developers from using ASP is a lack of support tools for developing answer-set programs. One particular problem in the context of programming support is *debugging of answer-set programs*. Due to the fully declarative semantics of ASP, it can be quite tedious to detect an error in an answer-set program. In recent years, some approaches towards debugging in ASP were proposed to tackle this problem. These previous works are important contributions towards ASP development support, however current approaches come with limitations to their practical applicability. In particular, existing approaches do not cover important aspects of ASP solver languages and often the amount of information a user has to provide or is confronted with during debugging is high.

This thesis introduces the *stepping methodology for ASP*, which is a novel technique for debugging answer-set programs that is general enough to *deal with current ASP solver languages* and *intuitive and easy to use*. Our method is similar in spirit to a widespread and effective debugging strategy in imperative programming, where the idea is to gain insight into the behaviour of a program by executing statement by statement, following the program's control flow. In our technique, we allow for stepwise constructing interpretations by considering rules of an answer-set program at hand in a successive manner. A major difference to the imperative setting is that, due to its declarativity, ASP lacks any control flow. Instead, we allow the user to follow his or her intuition on which rule instances to become active. This way, one can focus on interesting parts of the debugging search space from the beginning. Bugs can then be detected quickly, whenever the stepping session reveals differences between the actual semantics of the answer-set program and the expectations of the user. We explain our approach using two example scenarios, discuss methodological aspects, and the embedding of stepping in the ASP development process.

In order to establish a solid formal basis for the stepping technique, we developed a framework of computations for answer-set programs. For fully supporting current solver languages we were faced with several challenges in doing so. For one, the languages of answer-set solvers differ from each other and from formal ASP languages in various ways. Thus, in order to develop a method that works for different solvers, we needed an abstract ASP language that is sufficiently general to capture solver languages. To this end, we make use of abstract constraints as an established abstraction for language constructs such as aggregates, weight constraints, or external atoms. However, there was no semantics available for arbitrary abstract-constraint programs with disjunctions being compatible with the semantics of all the ASP solvers we want to support, namely, `Clasp`, `DLV`, and `DLVHEX`. Therefore, we introduce such a semantics in this work that extends the well-known FLP-semantics and show several properties thereof including complexity results. Moreover, we extend the concept of unfounded sets to our targeted program class and present different characterisations of the new semantics that are relevant for the framework of computations. Another basic problem we address deals with the grounding step in which variables are removed from answer-set programs before solving. In formal ASP lan-

guages, the grounding of a program consists of all rules resulting from substitutions of variables by ground terms. In contrast, actual grounding tools apply many different types of simplifications and pre-evaluations for creating a variable-free program. In order to accommodate this fact, we use abstractions of the grounding step together with a very abstract notion of non-ground answer-set programs as the base language for the stepping methodology. This way, the technique can easily be applied to existing solver languages and it becomes robust to changes to these languages.

The stepping technique has been implemented in `SeaLion`, an integrated development environment for ASP that has been developed in the same context as this thesis, viz. in connection of a research project on methods and methodologies for developing answer-set programs. We present `SeaLion` and discuss how it can be used for stepping answer-set programs written in the `Gringo` or the `DLV` language.

Finally, we compare the concepts developed in this thesis with related approaches and discuss future work including interesting applications of our results beyond the scope of debugging.

Kurzfassung

Die Antwortmengenprogrammierung (engl. “answer-set programming” - ASP) ist ein Programmierparadigma für deklaratives Problemlösen, das sich im Bereich der künstlichen Intelligenz und der Wissensrepräsentation hoher Beliebtheit erfreut. Allerdings hat sie bislang außerhalb des Wissenschaftsbetriebs noch wenig Verbreitung gefunden. Ein Hindernis, welches die Verbreitung von ASP erschwert, ist die mangelnde Verfügbarkeit von Entwicklungswerkzeugen zum Erstellen von Antwortmengenprogrammen. Hier ist Debugging ein Kernbereich bezüglich der Unterstützung von ASP Entwicklern, das heißt, das Problem des Auffindens von Fehlern in Antwortmengenprogrammen, welches aufgrund der deklarativen Semantik von ASP sehr schwierig sein kann. Um diesem Problem zu begegnen wurden in den letzten Jahren einige Methoden für das Debuggen von Antwortmengenprogrammen vorgeschlagen. Während diese Arbeiten als wichtige Beiträge zur Programmierunterstützung für ASP anzusehen sind, weisen existierende Ansätze Einschränkungen bezüglich ihrer praktischen Anwendbarkeit auf. Insbesondere werden darin wichtige Teile des Sprachumfangs von modernen ASP Solvern nicht abgedeckt und oftmals ist die Menge an Informationen sehr hoch, die dem System bereitgestellt werden muss oder mit der die Benutzerin oder der Benutzer konfrontiert wird.

In dieser Dissertation wird die Stepping Methodologie für ASP eingeführt, eine neue Technik für das Debuggen von Antwortmengenprogrammen, die allgemein genug ist um auf verschiedene ASP Solver Sprachen angewandt zu werden und intuitiv und einfach zu benutzen ist. Unsere Methode ähnelt einer weitverbreiteten und erfolgreichen Strategie für das Debuggen von imperativen Programmiersprachen, die darauf abzielt Einsicht in das Verhalten eines Programmes zu erhalten indem, dem Kontrollfluß des Programmes folgend, einzelne Programmbefehle schrittweise ausgeführt werden. Bei unserer Technik erlauben wir schrittweise Interpretationen aufzubauen indem immer mehr Regeln eines Antwortmengenprogramms berücksichtigt werden. Ein zentraler Gegensatz zum imperativen Fall ist, dass es in ASP aufgrund des deklarativen Ansatzes keinen Kontrollfluss gibt. Stattdessen erlauben wir dem Benutzer oder der Benutzerin der eigenen Intuition zu folgen um zu entscheiden, welche Regel als nächstes betrachtet werden soll. Auf diese Weise ist es möglich sich von Anfang an auf interessante Bereiche des Debugging Suchraums zu fokussieren. Programmierfehler können so rasch gefunden werden, indem eine Stepping Sitzung Unterschiede zwischen der tatsächlichen Semantik eines Programms und der Intuition des Benutzers oder der Benutzerin aufzeigt. Neben einer Erklärung unseres Ansatzes am Beispiel zweier Problemstellungen, diskutieren wir methodologische Aspekte und die Einbettung von Stepping im ASP Entwicklungsprozess.

Um die Stepping Technik auf eine solide theoretische Grundlage zu stellen, haben wir ein formales Rahmenwerk für Berechnungen von Antwortmengenprogrammen entwickelt. Damit dieses aktuelle ASP Solver Sprachen tatsächlich vollständig unterstützt, mussten wir mehrere Herausforderungen bewältigen. Eine wesentliche Schwierigkeit hierbei ist, dass sich die Sprachen von ASP Solvern sowohl voneinander als auch von formalen ASP Sprachen unterscheiden. Um eine Methode zu entwickeln die für mehrere Solver funktioniert, benötigten wir daher eine abstrakte ASP Sprache die allgemein genug ist um verschiedene Solver Sprachen abzudecken. Zu diesem Zweck greifen wir auf Abstract Constraints zurück, einer etablierten Abstraktion von beliebten Sprachkonstrukten wie Aggregaten, Weight Constraints oder externen Atomen.

Allerdings war keine Semantik für Programme die sowohl Abstract Constraints als auch Disjunktionen in Regelköpfen unterstützen verfügbar, die mit den Semantiken der Solver die wir unterstützen wollen, `Clasp`, `DLV` und `DLVHEX`, kompatibel ist. Aus diesem Grund führen wir in dieser Arbeit eine solche Semantik ein, welche die bekannte FLP-Semantik erweitert, und zeigen einige ihrer Eigenschaften. Außerdem erweitern wir den Begriff einer nicht-fundierten Menge auf die von uns anvisierte Programmklasse und stellen verschiedene Charakterisierungen der neuen Semantik vor, die für unser Berechnungsmodell wichtig sind. Ein anderes grundlegendes Problem, dem wir uns widmen, steht im Zusammenhang mit dem Grundierungsschritt in dem Variablen in einem Antwortmengenprogramm eliminiert werden bevor seine Antwortmengen berechnet werden. In formalen ASP Sprachen besteht die Grundierung eines Programms aus allen Regeln die durch Substitution von Variablen durch grundierte Terme erzeugt werden können. Im Gegensatz dazu erstellen reale Grundierungstools eine optimierte Grundierung, verwenden dafür verschiedene Heuristiken und führen auch diverse Berechnungen aus, wie die Evaluierung von intepretierten Funktionssymbolen. Um diesem Umstand gerecht zu werden verwenden wir Abstraktionen des Grundierungsschrittes sowie einen sehr abstrakten Begriff von nicht-grundierten Programmen als Grundlage für die Stepping Methodologie. Das erlaubt es uns die Technik in einfacher Weise auf existierende Solver Sprachen anzuwenden und macht sie robust gegenüber Änderungen dieser Sprachen.

Die Stepping Technik wurde in `SeaLion` implementiert, einer integrierten Entwicklungsumgebung für ASP, die im selben Kontext wie diese Dissertation entstand, nämlich im Rahmen eines Forschungsprojektes über Methoden und Methodologien zur Enwicklung von Antwortmengenprogrammen. Wir stellen `SeaLion` vor und beschreiben wie Stepping darin für Antwortmengenprogramme in den Sprachen von `Gringo` und `DLV` verwendet werden kann.

Darüber hinaus vergleichen wir die Konzepte die in dieser Dissertation entwickelt wurden mit anderen Ansätzen aus der Literatur und geben einen Ausblick auf zukünftige Forschung, indem auch interessante Anwendungen unserer Resultate fernab des Debuggens besprochen werden.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 A Project on Methods and Methodologies for Developing Answer-Set Programs 2
1.3 On the Development of `SeaLion` and its Extensions . . . 3
1.4 Results and Structure of the Thesis . . . 3

2 State of the Art of Debugging Answer-Set Programs 7
2.1 What is Debugging (in the Context of ASP)? . . . 7
2.2 General Considerations on Debugging Answer-Set Programs . . . 8
2.3 Existing Approaches . . . 9
2.4 Shortcomings of Existing Approaches . . . 10

3 Background 13
3.1 A Brief History of ASP . . . 13
3.2 Basic Notions . . . 15
3.2.1 Alphabet . . . 15
3.2.2 Interpretations . . . 16
3.3 Syntax of Disjunctive Logic Programs . . . 16
3.3.1 A Note on Strong Negation . . . 17
3.3.2 Grounding . . . 17
3.4 Answer-Set Semantics . . . 19
3.4.1 Answer Sets vs. Stable Models . . . 19
3.4.2 Satisfaction and Models of LP-programs . . . 19
3.4.3 Gelfond-Lifschitz Reduct . . . 20
3.4.4 Answer Sets . . . 20
3.5 Extensions of Syntax and Semantics . . . 21
3.5.1 Weight Constraints, Cardinality Constraints, and Choice Atoms . . . 21
3.5.2 Aggregates . . . 22
3.5.3 External Atoms . . . 23
3.5.4 FLP-Semantics . . . 24
3.6 ASP (Solver) Languages . . . 25
3.6.1 General Remarks . . . 25
3.6.2 `Gringo` . . . 27
3.6.3 `DLV` . . . 33
3.6.4 `DLVHEX` . . . 35
3.7 Computational Complexity . . . 36
3.7.1 Complexity Classes . . . 36
3.7.2 Reductions, Hardness, and Completeness . . . 36

4 A Common Formal Basis for Different Solver Languages 37
4.1 Syntax of Abstract-Constraint Programs . . . 38

4.2 Satisfaction Relation . . . 39
4.3 Viewing ASP Constructs as Abstract Constraints . . . 39
4.4 Answer-Set Semantics . . . 40
4.4.1 FLP-Semantics for Elementary-Head C-Programs and a Simple Extension 41
4.4.2 Basic Definition . . . 41
4.5 Characterisations based on External Support, Unfounded Sets, and Unfounded-Freeness . . . 44
4.5.1 External Support . . . 44
4.5.2 Unfounded Sets . . . 45
4.5.3 Unfounded-Free Interpretations . . . 46
4.6 Complexity . . . 47
4.7 Comparison of our Semantics for Abstract-Constraint Programs to others . . . 47
4.7.1 Semantics in the Style of Faber, Pfeifer, and Leone . . . 48
4.7.2 Semantics in the Tradition of Simons et al. . . . 51
4.7.3 Solver Compatibility . . . 56

5 A Framework of Computations for Stepping 57
5.1 States . . . 57
5.2 Computations . . . 58
5.3 Properties . . . 60
5.4 Stable Computations . . . 64
5.5 Comparison to Computations by Liu et al. . . . 68

6 Computations for Non-Ground Programs 71
6.1 Gap between Theory and Practise: Non-Ground Programs in Solver Languages 71
6.2 An Abstraction of Non-Ground Programs . . . 74
6.3 Abstractions of Grounding . . . 74
6.3.1 Black-Box Grounding . . . 74
6.3.2 Conditional Grounding . . . 77

7 Stepping Answer-Set Programs 87
7.1 Example Problems . . . 87
7.1.1 Maze Generation Problem . . . 87
7.1.2 Fair Minesweeper . . . 89
7.2 General Idea . . . 91
7.3 Steps . . . 92
7.4 Jumps . . . 94
7.5 Methodology . . . 97
7.5.1 Stepping Cycle . . . 97
7.5.2 Program Analysis and Debugging Level Methodology . . . 97
7.5.3 Top-Level Methodology . . . 99
7.5.4 The Stepping Guide . . . 99
7.6 Use Cases . . . 101
7.7 General Guidelines for Development . . . 109
7.8 Comparison to other Debugging Approaches for ASP . . . 111

8 Stepping in the Integrated Development Environment `SeaLion` 115
8.1 Design, Architecture, and Availability of `SeaLion` . . . 115
8.2 IDE Features . . . 118
8.2.1 Solver Interaction . . . 118
8.2.2 LANA Support, Documentation Generation, and Model-Driven Engineering . . . 120

8.2.3 Visualisation and Visual Editing of Answer Sets 123
8.2.4 Debugging Features other than Stepping 128
8.3 Practical Stepping in `SeaLion` 128
8.3.1 Initiating Stepping 129
8.3.2 Stepping Perspective 130
8.4 Comparison of `SeaLion` to other IDEs for ASP 133

9 Summary and Conclusion **137**
9.1 Summary 137
9.2 Outlook 138

A Predefined Visualisation Predicates in `Kara` **141**

Bibliography **143**

1 Introduction

In this introductory chapter, we describe the context in which this thesis has been written. The next section gives motivation for the approach developed in this work. Section 1.2 discusses goals and achievement of the research project in whose context this thesis has evolved. The development process of the `SeaLion` system, in which the main contribution of this work—the stepping technique—has been implemented, is discussed in Section 1.3.The chapter closes with an outline of this thesis in which we highlight the most important results of each chapter.

1.1 Motivation

When Ada Lovelace wrote the first computer program for calculating the Bernoulli numbers on Babbage's conceptual analytical engine, she came up with a series of instructions that the machine would execute one after the other. That is, she designed an algorithm targeted at a certain machine model. In principle, this imperative form of programming—telling a computer what to do step-by-step to solve a problem—is still the prevalent form of programming today. Nevertheless, programming languages become more and more high-level which allows the programmer to abstract from the hardware and implement complex programs in a concise and clearly structured way. Going one step further in abstraction, programming languages have evolved in which it suffices that a programmer simply *describes* a problem in order to solve it. That is, following this principle, called *declarative programming*, it is no longer necessary to give the computer instructions on *how* to solve the problem.

One particular instance of declarative programming is the *answer-set programming* (ASP) paradigm. Here, the idea is to describe the problem in terms of a logic theory (typically in form of a so-called *logic program*) such that dedicated models of the theory are in one-to-one correspondence with the solutions of the problem. These models, called *answer sets*, can be automatically computed by a solver and from each answer set, the solution it represents can be read off. ASP is relatively young—it has been proposed as a programming paradigm in 1999 (see Section 3.1)—and has been applied for problems from many areas since then. Nevertheless, it has mainly been used by people from academia and has not become a mainstream programming approach yet. One possible explanation for that is that writing an answer-set program is quite different from what developers are used to. Another obstacle for a wider acceptance of ASP we identify, is a lack of support tools and methods for developing in ASP. Indeed, developers are used to tools, methods, and methodologies that ease the programming process, however many of these techniques cannot be applied to ASP in a straightforward way. In particular, ASP lacks debugging methods, i.e., techniques that help the programmer to identify and correct programming errors. In fact, errors in ASP can be quite hard to find, e.g., in the frequent scenario when a program unexpectedly has no answer sets. Then, the programmer has no indication where the problem could be. And, in general, often small changes in an answer-set program have major effects on the resulting answer sets.

So far, a few debugging techniques for ASP have been proposed in theoretical works and also some prototype debuggers have been implemented (for a more detailed discussion see Chapter 2). However, we identify two shortcomings of current approaches which limit their

potential for practical application. First, most existing techniques and tools only capture a basic ASP language fragment that does not include many language constructs that are available and frequently used in modern ASP solver languages. Second, usability aspects are often not considered in current approaches, in particular, the programmer is required to either provide a lot of data to a debugging system or he or she is confronted with a huge amount of information from the system.

The goal of this work is the development of a debugging technique for ASP that overcomes current limitations. We aim at a technique based on a solid formal basis that is general enough to deal with current ASP solver languages and intuitive and easy to use. Our target audience is not restricted to ASP experts but includes also "the programmer from the street", i.e., developers who are new to ASP but experienced with development tools in conventional programming paradigms. To properly address their needs, we want our approach to be conceptually close to debugging techniques for other programming languages and to be accessible from an integrated development environment similar to popular debugging tools.

We hope that the targeted debugging technique can contribute to the popularity of ASP in two ways. On the one hand, we want to offer new users a programming experience that they already feel familiar with. On the other hand, by allowing the programmer to get insight into the consequences of their programs, the new technique should allow her or him to understand the semantics of answer-set programs during debugging sessions in a hands-on fashion.

1.2 A Project on Methods and Methodologies for Developing Answer-Set Programs

This thesis is written in the context of the project "Methods and Methodologies for Developing Answer-Set Programs" (MMDASP), conducted at the Knowledge-based Systems Group of the Institute for Information Systems at the Vienna University of Technology and funded by the Austrian Science Fund (FWF). The project that lasted four years started in September 2009 and was led by Hans Tompits. Besides the principal investigator, Johannes Oetsch and the author of this work were members of the project team.

The aim of the project was to put forth a systematic study into development methods for ASP to address the need for tools, methods, and methodologies that ease the programming process that we discussed in Section 1.1. Before the project started there were only few preliminary works in this direction available. The focus of research in the project was on methodologies for systematic program development, program testing, and debugging. From the beginning, it was the goal to develop methods that respect the declarative nature of ASP and, in order to support a sufficient level of applicability, solutions were searched for that target not only the core language of ASP but also important extensions thereof that are commonly used and realised in various answer-set solvers. Furthermore, an important objective of the project was the implementation of an integrated development environment (IDE) for ASP that incorporates resulting methods and realises a convenient tool for developing answer-set programs.

The project team succeeded in reaching all the major project goals; most important with respect to this thesis is the work on debugging and the development of `SeaLion` (see Chapter 8) which is the first comprehensive integrated development environment that supports all major ASP language dialects. Project achievements that are not subject to this thesis include work on testing answer-set programs, including methods for systematic structure-based testing and random testing for ASP (Janhunen et al., 2010, 2011). Moreover, further testing methods that have been studied are mutation testing based on a dedicated mutation model for ASP and bounded-exhaustive testing based on a small-scope hypothesis for ASP (Oetsch et al., 2012a). Additionally, going beyond testing for ASP, ASP has been used to tackle challenging combinatorial testing problems related to testing event-driven software (Brain et al., 2012). Regarding

systematic program development, methods based on model-driven engineering and test-driven development have been considered for ASP (Oetsch et al., 2011a; Busoniu, 2013).

1.3 On the Development of SeaLion and its Extensions

The foundations of the SeaLion system (which is described in Chapter 8) were laid in the summer of 2009 when the first code to the repository was committed. Since then, the core components of SeaLion have grown to about 100 000 source lines of code. The author of this thesis is the lead developer of the system—but the overall implementation is the joint effort of several people. Indeed, a number of students were involved in SeaLion related projects and some of these implementations served as basis of their master's or bachelor's theses. In what follows, we give credit to them in chronological order of the contribution.

- Christian Kloimüllner implemented the Kara plugin for visualising and visual editing of answer-set programs. The plugin was the subject of his master's thesis (Kloimüllner, 2012) (which recently appeared as a paperback (Kloimüllner, 2013)) and a workshop paper (Kloimüllner et al., 2013).
- Michael Prischink has been working on a plugin that deals with testing of answer-set programs and assertions for ASP. One feature he implemented allows for computing random answer sets.
- Doğa Gizem Kısa implemented the ASPDOC documentation generator that has been integrated in SeaLion as well as the ASPUNIT testing tool while she was doing an internship at Vienna University of Technology. The tools are based on the LANA annotation language and were presented at conferences (De Vos et al., 2012a) and workshops (De Vos et al., 2012b).
- Based on the stepping framework developed in this thesis, Peter Skočovský implemented the stepping plugin of SeaLion. The plugin was one focus of a paper on SeaLion (Busoniu et al., 2013) and described in his master's thesis (Skočovský, 2014) in which he also gives a formalisation of the Gringo language.
- Paula-Andra Busoniu implemented a plugin for model-driven engineering in ASP which was subject of her master's thesis (Busoniu, 2013) and is also described in the latest paper on SeaLion (Busoniu et al., 2013).
- In her master's thesis (Frühstück, 2013), Melanie Frühstück describes an extension of a previous debugging approach for ASP (Oetsch et al., 2010a). She implemented the work in the Ouroboros plugin of SeaLion. The approach (Polleres et al., 2013) as well as the plugin (Frühstück et al., 2013) were described in conference papers.
- Peter Eder implemented an explanation feature for searching rules that derive given atoms. The plugin is described in his bachelor's thesis (Eder, 2013).
- Min Fang wrote her bachelor's thesis (Fang, 2013) about an approach for interpreting answer sets using controlled natural language and implemented a corresponding SeaLion plugin.

1.4 Results and Structure of the Thesis

We now give an overview of the structure of this work and point out the achievements of the individual chapters. After that we list publications that have emerged from the work on this thesis.

Chapter 1 provides initial motivation for our work and discusses the context under which this thesis has been written.

General considerations on debugging in the context of ASP are given in Chapter 2. We discuss previous debugging approaches for ASP, elaborate on why further work is needed, and outline important goals for a novel debugging technique.

Important background for the subsequent chapters is provided in Chapter 3. It starts with a historical account of ASP and introduces notation and the formalisms we use throughout this thesis. We formally introduce different classes of logic programs under the answer-set semantics in a uniform way. In doing so, we also give syntax and semantics for language extensions such as weight constraints, aggregates, or external atoms. We also highlight differences between formal ASP languages and the languages of several ASP solvers in which users write their programs in practise. Being aware of these differences, e.g., regarding the grounding of programs, is crucial for putting a development method such as a debugging technique into practise. Finally, we recall basic notions of complexity theory that we need.

Chapter 4 is concerned with an abstraction of ground answer-set programs that is based on abstract-constraint programs (Marek and Remmel, 2004; Marek and Truszczyński, 2004) and serves as a common formal basis for different solver languages. We recall abstract-constraint atoms and how they can be used to simulate popular ASP language constructs. Then, we introduce a novel answer-set semantics for disjunctive abstract-constraint programs that is a proper extension of the FLP-semantics (Faber et al., 2011). Based on the notion of unfounded sets, we provide different characterisations of this semantics, show different properties, and analyse its computational complexity. Finally, we study how our semantics is related to existing proposals and discuss why these were not suitable for our purposes. Besides being the underpinning for our stepping approach, the semantics can also be helpful for other purposes. For instance it can serve as a theoretical basis for extensions of solver languages, e.g., for adding choice rules to the language of the `DLV` solver. In particular, the characterisation in terms of unfounded sets can be seen as a practical step towards an implementation in `DLV` as unfounded sets are central elements of the evaluation strategy of this solver.

Chapter 5 introduces a framework of computations for the semantics of Chapter 4. Roughly, in a computation, an answer set is computed by stepwise considering more rules to be active. Thus, the framework allows breaking the semantics down to the level of individual rules which allows us to get very focused debugging information. After defining states and computations, we show several properties of the framework, most importantly soundness and completeness in the sense that the result of a successful computation is an answer set and that every answer set can be computed with a computation. Moreover, we study language fragments for which a simpler form of computation suffices. Finally, we compare our notion of computation with that of Liu et al. (2010). It turns out that our results solve an open problem stated in their work, that is, our framework demonstrates that the presence of disjunctions does not require a global minimality criterion on computations.

For applying the computational framework for real-world solver programs, we introduce abstractions of the grounding step in Chapter 6 in the form of grounding functions. These translate given non-ground solver programs—for which we make only very little assumptions on their syntax—into abstract-constraint programs that serve as abstractions of ground solver programs. We lift the framework of computations and its properties to abstract non-ground programs. Besides the use for stepping, our framework of computations can be seen as a calculus for ASP languages in the joint presence of disjunctions and aggregates. Moreover, the combination of the use of abstract-constraint programs and the abstractions of grounding could be beneficial for developing further development methods for ASP because techniques that work for our abstractions could then be applied to the solver languages easily. Moreover, a further application area of our framework besides debugging is the development of on-the-fly grounding answer-set solvers. Indeed, with little effort, the framework could be turned into an algorithm

for computing answer sets where variables are eliminated during solving.

In Chapter 7 we present the stepping technique for debugging answer-set programs based on the framework developed in Chapters 4 and 5. After discussing the general idea of the approach, we explain what we understand under *steps* and *jumps* as a means to progress in a computation. We discuss methodological aspects of stepping on different conceptual levels and discuss several use cases based on two example problems. Furthermore, we provide guidelines for stepping and discuss general recommendations for ASP development.

Chapter 8 deals with the integrated development environment `SeaLion` for ASP that comes with an implementation of the stepping technique. `SeaLion` has been developed in the realm of the MMDASP project (cf. Sections 1.2 and 1.3). General information on the implementation of `SeaLion` is provided and its architecture and availability are discussed. Furthermore, we describe important features of the environment, e.g., support for model-driven engineering and visualisation of answer sets. We then show how the stepping technique is realised in `SeaLion`. That is, we explain the user interface of the stepping plugin and how it can be used. Moreover, we compare `SeaLion` with related systems.

Finally, Chapter 9 concludes the thesis with a summary and an outlook on possible future research on the topics of this work.

Parts of this thesis have been presented in different publications. Our new semantics for abstract-constraint programs has been discussed in a paper at the 28th International Conference on Logic Programming (Oetsch et al., 2012b). Stepping, as introduced in this work is a generalisation of earlier versions of the methodology that have appeared in the proceedings of the 24th Workshop on (Constraint) Logic Programming (Oetsch et al., 2010b), the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (Oetsch et al., 2011c), and and a special collection dedicated to the 65th birthday of Vladimir Lifschitz (Oetsch et al., 2012c). There have been several publications related to `SeaLion` and `SeaLion` plugins. The IDE itself has been discussed in a Theory and Practice of Logic Programming article (Busoniu et al., 2013) presented at the 29th International Conference on Logic Programming and a paper at the 25th Workshop on Logic Programming (Oetsch et al., 2011b, 2013). Work related to `SeaLion` plugins have been presented at the 28th International Conference on Logic Programming (De Vos et al., 2012a), the 14th International Workshop on Non-Monotonic Reasoning (De Vos et al., 2012b), the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (Frühstück et al., 2013), and the 25th Workshop on Logic Programming (Kloimüllner et al., 2013).

2 State of the Art of Debugging Answer-Set Programs

In this chapter, we give a summary over previous efforts towards debugging support for ASP. We start with clarifying what we understand as debugging and then discuss the peculiarities of debugging answer-set programs. We proceed with descriptions of previous debugging approaches for ASP and with an explanation why further work is needed. A comparison of these techniques with the stepping method introduced in this thesis is given later in Chapter 7.

2.1 What is Debugging (in the Context of ASP)?

A definition of *debugging* is given in the ANSI/IEEE Standard Glossary of Software Engineering Terminology (ANSI/IEEE, 1983):

> "Debugging is the process of locating, analyzing, and correcting suspected faults."

Furthermore, a *fault* is defined as an

> "accidental condition that causes a program to fail its required function".

Thus, software debugging deals with finding and eliminating errors ("bugs") in computer programs. The nature of these errors is manifold and reaches from simple misspellings to conceptual programming errors. Software tools supporting debugging are referred to as *debuggers*.

According to a general belief, the first computer bug was a moth which caused a failure of the famous Mark-II computer in 1945. However, the term had already been used earlier for errors in technical devices, e.g., by Thomas Edison in 1878 (Hughes, 1989).

Brain and De Vos (2005) discussed the nature of bugs in ASP along the lines of a classification scheme for errors, tailored to suit classical imperative programming languages (Aho et al., 1986; Wertz, 1982). Herein, bugs are distinguished by the *level of specification* in which they occur in the program:

- *lexical* and *syntactic errors*: the program contains strings or sentences not occurring in the programming language;
- *semantic errors*: the program meets the syntactical requirements of the language but the assembly of its components does not make sense;
- *conceptual errors*: the program is correct but it does not serve the intended purpose.

The first category includes misspellings of keywords, identifiers or operators, unbalanced parenthesis in arithmetic expressions. Examples for semantic errors are "division by zero", "infinite loops", and "index-out-of-bound errors". They are typically recognised in the first place when the program is executed. Therefore, these types of errors are also referred to as *runtime errors*. Conceptual errors are often first recognised when the program is systematically tested or already in application (Ruzicka, 1990). Brain and De Vos pointed out that, due to the simple structure

of ASP languages, the scope for lexical and syntactic errors is rather small. Indeed, typos in identifiers for predicates or terms often lead to syntactically correct programs as ASP languages usually do not enforce prior declaration of identifiers. This may sometimes cause tedious bugs. The authors also claim that semantic errors do not exist in ASP, i.e., every syntactically correct program has a well-defined semantics. While one could argue that violations of, e.g., safety or stratification requirements amount to semantic errors, their argument remains true in essence: The majority of bugs in ASP are conceptual errors, i.e., mismatches between the actual and the intended semantics of a program. In terms of ASP, this means that the computed answer sets do not match our expectations. In conclusion, in this thesis, we are merely concerned with bugs that are parts of a computer program that cause the program's *actual semantics* to *differ from the semantics that is intended*.

2.2 General Considerations on Debugging Answer-Set Programs

Usually, when a new programming language is created, it is only a matter of time that a respective debugging system becomes available, following the approaches of similar tools for related languages. However, answer-set programming is not yet another programming language but a programming paradigm that significantly differs from other languages in some aspects that make it unclear how existing debugging approaches could be applied. In particular, the particularities of ASP that make debugging a challenge are *declarativity* and *non-determinism*.

An answer-set program can be seen as a declarative description of a problem, i.e., the order of the rules that constitute the program is irrelevant and the evaluation of a program does not follow a particular control flow. For this reason, typical debugging approaches of imperative programming that follow a program's execution cannot be applied in a straightforward way. One theoretical solution for that would be to follow the execution of an answer-set solver. A similar strategy is for example followed for debugging PROLOG programs, where the user follows the inference algorithm. However, it has been argued that such an approach would lead to several disadvantages (Brain and De Vos, 2005). For one, the debugging method would be solver specific and would require the user to understand the solver algorithm. While this might be feasible for toy programs and basic solving algorithms, it becomes useless for real-world programs and modern answer-set solvers that employ more complex algorithms, heuristics, or translations in other formalisms. Moreover, such an approach would, arguably, impose a procedural view on answer-set program, ruining the declarative flavour of ASP, i.e., an answer-set program would be seen as a configuration for a search algorithm rather than a declarative description. Finally, it would be difficult to focus on information the programmer is interested in when inspecting the execution of an answer-set solver. Nevertheless, tools for analysing runs of answer-set solvers exist (Calimeri et al., 2009; König and Schaub, 2013) and are useful for other purposes than debugging answer-set programs, such as debugging the solver itself or analysing the efficiency or bottlenecks of an ASP problem encoding.

ASP can be seen as a nondeterministic formalism in the sense that an answer-set program may have multiple answer sets. This necessarily has an influence on prospective debugging techniques. On the one hand it is not obvious how a debugging strategy could take multiple answer sets into consideration. On the other hand, for many sorts of bugs that arise in ASP, it would be beneficial to have debugging strategies that are local, i.e., can be seen with respect to an answer set or an answer-set candidate interpretation. In that case in turn, appropriate means for choosing such an interpretation are required.

An important concept in the light of these considerations is *declarative debugging* that was originally introduced as *algorithmic debugging* by Shapiro (Shapiro, 1982) in 1982. The basic idea is that a debugging system detects errors guided by information about intended properties of the program. This information has to be supplied by an oracle, typically the programmer.

Thus, the user has to supply declarative knowledge about the intended semantics of a program, but is not required to care about the computational behaviour of the system.

Declarative debugging was initially used for debugging PROLOG programs, but has been proposed as a general approach towards debugging and also been applied to other paradigms, such as functional (Naish, 1992) and imperative programming (Fritzson et al., 1991).

2.3 Existing Approaches

Next, we describe existing approaches for debugging answer-set programs. A discussion on their relation to the method proposed in this thesis is given later in Section 7.8.

The first work devoted to debugging of answer-set programs is a paper by Brain and De Vos (2005) in which they provide general considerations on the subject, such as the discussion of error classes in the context of ASP or implications of declarativity on debugging mentioned in the previous section. They also formulated important debugging questions in ASP, namely, why is a set of atoms subset of a specific answer set and why is a set of atoms not subset of any answer set. The authors provided pseudocode for two imperative ad-hoc algorithms for answering these questions for propositional normal answer-set programs. The algorithm addressing the first question returns answers in terms of active rules that derive atoms from the given set. The algorithm for explaining why a set of atoms is not subset of any answer set identifies different sorts of answers such as atoms with no deriving rules, inactive deriving rules, or supersets of the given set in which adding further literals would lead to some inconsistency.

The goal of the work by Pontelli et al. (2009) is to explain the truth values of literals with respect to a given actual answer set of a program. Explanations are provided in terms of *justifications* which are labelled graphs whose nodes are truth assignments of possibly default-negated ground atoms. The edges represent positive and negative support relations between these truth assignments such that every path ends in an assignment which is either assumed or known to hold. The authors have also introduced justifications for partial answer sets that emerge during the solving process (online justifications), being represented by three-valued interpretations.

Syrjänen (2006) aimed at finding explanations why some propositional program has no answer sets. His approach is based on finding minimal sets of constraints such that their removal yields consistency. Hereby, it is assumed that a program does not involve circular dependencies between literals through an odd number of negations which might also cause inconsistency. The author considers only a basic ASP language and hence does not take further sources of inconsistency into account, caused by program constructs of richer ASP languages, such as cardinality constraints.

Another early approach (Brain et al., 2007b; Pührer, 2007) is based on program rewritings using some additional control atoms, called *tags*, that allow, e.g., for switching individual rules on or off and for analysing the resulting answer sets. Debugging requests can be posed by adding further rules that can employ tags as well. That is, ASP is used itself for debugging answer-set programs. The translations needed were implemented in the command-line tool `Spock` (Brain et al., 2007a; Gebser et al., 2009b) which also incorporates the translations of another approach in which also ASP is used for debugging purposes (Gebser et al., 2008; Pührer, 2007). The technique is based on ASP meta-programming, i.e., a program over a meta-language is used to manipulate a program over an object language (in this case, both the meta-language and the object language are instances of ASP). It addresses the question why some interpretation is not an answer set of the given program. Answers are given in terms of a model-theoretic characterisation of answer sets due to Lee (2005): An interpretation I is not an answer set of a program P iff (i) some rule in P is not classically satisfied by I or (ii) I contains some loop of P that is unfounded by P with respect to I. Intuitively, Item (ii) states that some atoms in I are not justified by P in the sense that no rules in P can derive them without reference to I itself. Item (ii) captures also the case that some atoms are in I only because they are

derived by a set of rules in a circular way—like the Ouroboros, a dragon biting in its own tail. The approach has later been extended from propositional to disjunctive logic programs with constraints, integer arithmetic, comparison predicates, and strong negation (Oetsch et al., 2010a) and also to programs with cardinality constraints (Polleres et al., 2013). It has been implemented in the `Ouroboros` plugin of `SeaLion` (Frühstück et al., 2013).

Caballero et al. (2008) developed a declarative debugging approach for datalog using a classification of error explanations similar to that of the aforementioned meta-programming technique (Gebser et al., 2008; Oetsch et al., 2010a). Their approach is tailored towards query answering and the language is restricted to stratified datalog. However, the authors provide an implementation that is based on computing a graph that reflects the execution of a query.

Wittocx et al. (2009) show how a calculus can be used for debugging first-order theories with inductive definitions Denecker (2000); Denecker and Ternovska (2008) in the context of model expansion problems, i.e., problems of finding models of a given theory that expand some given interpretation. The idea is to trace the proof of inconsistency of such an unsatisfiable model expansion problem. The authors provide a system that allows for interactively exploring the proof tree.

Besides the mentioned approaches which rely on the semantical behaviour of programs, (Mikitiuk et al., 2007) use a translation from logic-program rules to natural language in order to detect program errors more easily. This seems to be a potentially useful feature for an IDE as well, especially for novice and non-expert ASP programmers.

Initial results of the work presented in this thesis have been published at international workshops and conferences. In particular, we reported on stepping for normal answer-set programs (Oetsch et al., 2010b, 2011c). Moreover, we devised a variant of our approach (Oetsch et al., 2012c) for debugging description logic programs, a formalism that combines logic programs under the answer-set semantics with description logics for semantic web reasoning.

2.4 Shortcomings of Existing Approaches

The different approaches discussed in the previous section are valuable contributions towards debugging of answer-set programs. In particular they have revealed interesting and highly relevant debugging questions, some of which are very specific to ASP. Nevertheless, their main common goal has not been reached yet, namely, having a debugging technique that allows for practical debugging of real-world answer-set programs. We identify two main reasons for that:

1. On the one hand, most of these approaches deal with an idealised mathematical ASP language, missing important features that are often used in actual solver languages. Most of the sketched approaches are applicable only to propositional programs whilst practical applications call for debugging methods for non-ground programs. Furthermore, typically, answer-set programmers make use of special language constructs, such as aggregates or choice rules that are not covered at all by current debugging strategies (except for the latest work by Polleres et al. (2013)).

2. On the other hand, only little attention has been paid to the usability and the human-computer interface of the proposed techniques. Some methods require that the user has some form for providing a considerable amount of information to the debugging system, while in others the contrary is true, i.e., the debugger overloads the user with too much output. In general, software developers will only adopt to debugging techniques that are easy to use and give a clear benefit over a manual search for bugs.

Therefore, we aimed for a debugging approach that

- is general enough to be applied to real-world answer-set programs written in (different) actual solver languages;

- is intuitive for persons familiar with debugging techniques from other paradigms;
- is based on a simple strategy;
- allows the user to provide information in a comfortable way;
- requires only a reasonable amount of user interaction;
- but respects the peculiarities of ASP as discussed in Section 2.2.

3 Background

Before proceeding to the technical contributions of this thesis, we give general background for the methods and techniques introduced in this work. First, we give a brief discussion on the history of ASP, elucidate its roots in different fields, and highlight important cornerstones until its identification as a programming paradigm. Moreover, basic notions are introduced that are used throughout the thesis, including the definition of syntax and semantics for a core fragment of answer-set programs. Furthermore, we present different extensions thereof, in particular we formally define weight constraints, aggregates, and external atoms, as examples of special literals whose truth depends on multiple atoms in an interpretation. Besides setting up basic concepts, a main goal of the chapter is to underline differences between formal ASP languages and ASP solver languages that lead to practical difficulties for developing flexible and extensible programming support techniques. Our strategy to overcome such problems is abstraction, using abstract-constraint programs to represent ground programs, introduced in Chapter 4, and an abstraction of non-ground programs and grounding, developed in Chapter 6.

3.1 A Brief History of ASP

The idea of using mathematical logic for programming has been extensively studied for the first time in the 1960s and early 1970. One of the first and most well-known logic programming languages is PROLOG that was developed in collaboration of Alain Colmerauer and Phillipe Roussel from the University of Aix-Marseille and Robert Kowalski at the University of Edinburgh. The language followed a philosophy expressed by Kowalski as

Algorithm = logic + control.

That is, to view an algorithm as controlled logical deduction, consisting of two components, a logic component that expresses the axioms that may be used in the computation and a control component that determines how deduction is applied to the axioms. In this respect, a PROLOG program is considered to be the logic component, while the control component is fixed and based on the inference rule of *selective linear definite clause resolution with negation as failure* (SLDNF). A PROLOG program consists of rules that describe relations over terms in a notation similar to that of first-order logic. PROLOG is query based, i.e., a PROLOG computation is an evaluation of a query that is provided by the user. This is performed in a top-down manner such that current subgoals are unified with rule consequences and rule antecedents may become new subgoals. Solutions to a query are given in the form of instantiations of variables contained in the query that lead to successful proof branches.

One criticism of PROLOG is that it is not fully declarative, e.g., the order of rules and the order of atoms within a rule body influences the semantics of the program. As a consequence, in order to understand the meaning of a PROLOG program, one must bear in mind how it would be interpreted by PROLOG's inference algorithm. The aim for a clear declarative semantics of logic programs has triggered a lot of research resulting in many different proposals for such a semantics. One central question in many of these works is how to handle negation. The problem has been addressed in the area of non-monotonic reasoning, among others, by the *program*

completion (Clark, 1978), *circumscription* (McCarthy, 1980), *default logic* (Reiter, 1980), and *autoepistemic logic* (Moore, 1985). All these formalisms had influence on the *stable-model semantics* for normal logic programs that was proposed by Gelfond and Lifschitz (1988). The reading of negation in this semantics can be seen as if it was in the scope of the epistemic operator of autoepistemic logic: the negation of a statement a holds if a is not known to hold. Alternatively, in the view of default logic, negation in a rule is treated as inverse justification, i.e., if all prerequisites are true (the atoms in the rule body that are not negated) and it is consistent with our beliefs that all atoms appearing negated in the rule body are false, then we are allowed to believe the conclusions of the rule. A further similarity of default logic and logic programming under the stable-model semantics is that both have a multiple solution semantics: default theories may have zero or more *extensions*, while logic programs may have zero or more *stable models* or *answer sets*, as they are called in Gelfond and Lifschitz' 1991 paper (Gelfond and Lifschitz, 1991). In this work, the authors extended their stable model semantics to extended disjunctive databases (cf. Section 3.3). Nevertheless, the emphasis at this time was still on query-oriented reasoning and the presence of multiple or absence of answer sets was considered unfavourable, e.g., in Gelfond and Lifschitz' paper, programs with a unique answer set are considered to be "well-behaved". A shift in thinking happened eight years later, when, what is now known as answer-set programming, had been recognised as a programming paradigm for the first time: Independently of each other, Niemelä (1999) and Marek and Truszczyński (1999) sketched a declarative programming paradigm based on logic programs under the stable-model semantics in which problems are encoded in a logic program such that the stable models of the program corresponds to the problem solutions. In both works the approach was perceived as a form of solving constraint-satisfaction problems. The term "answer-set programming" for the new paradigm is credited to Vladimir Lifschitz. ASP became important only due to the availability of answer-set solvers, i.e., systems that allow for computing the answer sets of a logic program. The first prominent answer-set solver was `Smodels` (Niemelä and Simons, 1996; Simons et al., 2002), soon followed by the `DLV` solver (Citrigno et al., 1997; Leone et al., 2006). Both reasoning systems were based on modified versions of the Davis-Putnam-Logemann-Loveland algorithm for solving the Boolean satisfiability problem (SAT). Generally, ASP systems have benefited a lot from advanced SAT solving techniques. There have been several proposals to translate answer-set programs to propositional formulas, e.g., Lin and Zhao showed how computing the answer sets of logic programs can be done by a transformation to propositional formulas, using Clark's completion and so-called loop formulas (Lin and Zhao, 2002, 2004). Other translations include that by Ben-Eliyahu and Dechter (1994) and Janhunen (2006). Consequently, many answer-set solvers have been developed that exploit SAT solvers in different ways, e.g., `ASSAT` (Lin and Zhao, 2004), `Cmodels` (Lierler, 2005), `sabe` and `pbmodels` (Liu and Truszczyński, 2006), `LP2SAT` (Janhunen, 2006), and `SUP` (Lierler, 2011). Moreover, modern SAT techniques have been introduced to native ASP solvers. The solver `Clasp` is based on advanced clause learning techniques (Gebser et al., 2007a) and has become competitive with SAT solvers even in their own discipline: Used as a SAT solver, `Clasp` has won tracks of the SAT Challenge 2012 (Balint et al., 2012), and the 2011 and 2009 SAT Competitions (Berre et al., 2009; Järvisalo et al., 2011). Unlike SAT, where a problem has to be compiled to a propositional formula, ASP comes with a rich yet simple modelling language that is human readable. The knowledge representation capabilities of ASP and its expressive power made the formalism an excellent host language for many applications in artificial intelligence such as argumentation (Egly et al., 2010), data integration (Leone et al., 2005), diagnosis (Eiter et al., 1999; Balduccini and Gelfond, 2003), learning (Sakama, 2001, 2005; Sakama and Inoue, 2009), planning (Lifschitz, 2002; Dix et al., 2003; Gebser et al., 2012), preferences (Schaub and Wang, 2001; Brewka, 2007; Brewka et al., 2008), probabilistic reasoning (Baral and Hunsaker, 2007; Baral et al., 2009), multi-agent systems (De Vos et al., 2006; Pontelli et al., 2012), multi-context systems (Brewka et al., 2011; Dix et al., 2012), natural language processing (Baral

et al., 2008; Lierler and Schüller, 2012), semantic web reasoning (Eiter et al., 2008; Simkus, 2009; Pührer et al., 2010), and theory update and revision (Osorio and Cuevas, 2007; Eiter and Wang, 2008; Delgrande, 2010).

But also many applications of ASP in other areas have been reported, including assisted living (Mileo et al., 2011), automatic music composition (Boenn et al., 2011), bio-informatics (Tran and Baral, 2004; Dworschak et al., 2008), configuration (Soininen and Niemelä, 1999; Syrjänen, 2000; Gebser et al., 2011b), decision support systems (Nogueira et al., 2001; Beierle et al., 2005), game theory (De Vos and Vermeir, 2002), hardware design (Erdem and Wong, 2004), model checking (Heljanko and Niemelä, 2003; Tang and Ternovska, 2007), phylogenetics (Erdem et al., 2006), robotics (Erdem et al., 2012), software testing (Brain et al., 2012), team building (Ricca et al., 2012), and verification of cryptographic protocols (Delgrande et al., 2009).

Most of these applications have been conducted by members of the scientific ASP community. We hope that software development methods for ASP will help to spread the ideas of the paradigm to a wider audience of interested developers.

3.2 Basic Notions

In this section we introduce an alphabet that we use as a common base for different logic-based formalisms. In doing so we sometimes adapt definitions and results from other work such that they fit the uniform notions and make corresponding notes whenever this affects claims or results. Furthermore, we define interpretations and truth values of ground atoms.

3.2.1 Alphabet

Throughout this work, we will assume a fixed implicit *first-order alphabet*, $\mathscr{A}$, which is a triple $\mathscr{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{F} \rangle$ of disjoint sets, where

- $\mathcal{P}$ is a set of *predicate symbols* of form p/n where p is the *name* and n is the *arity* of the predicate with $n \geq 0$,
- $\mathcal{V}$ is a set of *variables*, and
- $\mathcal{F}$ is a set of *function symbols* of form f/n where f is the *functor* and n is the *arity* of the function symbol with $n \geq 0$.

By convention, variables are denoted by symbol strings starting with capital letters, functors by strings starting with lower case letters, and predicate names by strings starting with a letter. Note that $p/1$ and $p/2$ are considered two different predicate symbols. We allow for multiple predicates with the same name because some ASP solver languages also do so.

Based on $\mathscr{A}$, we define basic terminology that is used throughout the thesis.

Definition 1. A *term* is a variable from $\mathcal{V}$ or a function, where a *function* is an expression $f(t_1, \ldots, t_n)$ such that $f/n \in \mathcal{F}$ and every t_i is a term for $1 \leq i \leq n$. An *atom* is an expression $p(t_1, \ldots, t_n)$ such that $p/n \in \mathcal{P}$ and every t_i is a term for $1 \leq i \leq n$. An expression is *ground* if it does not contain any variable. $\Diamond$

Functions of arity 0 are referred to as *constants*. The set of constants in $\mathcal{F}$ is denoted by $\mathcal{C}$. It is assumed that $\mathcal{C} \neq \emptyset$.

Example 1. The predicate of the atom

$$person(fatherOf(X))$$

is $person/1$ with name $person$ and arity 1. The atom is not ground as its only argument, the function $\mathit{fatherOf}(X)$, contains the variable X.

An example of a ground atom is

$$\mathit{friendOf}(\mathit{authorOf}(\mathit{frankenstein}), \mathit{byron}),$$

where $\mathit{frankenstein}$ and byron are constants. ■

Definition 2. Let $P \subseteq \mathcal{P}$ be a set of predicates and $F \subseteq \mathcal{F}$ a set of function symbols. Then, the *Herbrand base* with respect to P and F is the set B_P^F of all ground atoms $p(t_1, \ldots, t_n)$ such that $p/n \in P$ and each t_i for $1 \leq i \leq n$ contains only function symbols from F.

Furthermore, the set of all ground terms is the *Herbrand universe* $\mathrm{HU}_{\mathscr{A}}$ of $\mathscr{A}$ and $\mathrm{B}_{\mathrm{HU}_{\mathscr{A}}}^{\mathcal{P}}$, the set of all ground atoms, is called the *Herbrand base* of $\mathscr{A}$. ♢

We will sometimes make use of formalisms that were introduced for languages of *propositional logic*. For the aim of having a uniform language and better comparability, we identify propositional atoms with ground atoms. Consequently, the set of propositional atoms in such a foreign propositional formalism is given by $\mathrm{B}_{\mathrm{HU}_{\mathscr{A}}}^{\mathcal{P}}$ in the context of this thesis. Often, propositional atoms are identified with nullary predicates in first-order languages. However, as in our setting sometimes propositional techniques are applied on ground programs, viewing propositional variables as ground atoms seems more appropriate for our purposes.

3.2.2 Interpretations

Interpretations are structures that give semantics to expressions. As usual in answer-set programming, we use Herbrand interpretations, i.e., subsets of the Herbrand base of $\mathscr{A}$ that specify which ground atoms are considered true.

Definition 3. An *interpretation* is a set $I \subseteq \mathrm{B}_{\mathrm{HU}_{\mathscr{A}}}^{\mathcal{P}}$ of ground atoms. We say that a ground atom $p(t_1, \ldots, t_n)$ is *true* under interpretation I, symbolically $I \models p(t_1, \ldots, t_n)$, if $p(t_1, \ldots, t_n) \in I$, otherwise it is *false* under I. ♢

Note that we will use the symbol $\not\models$ to denote the negation of a relation denoted with the symbol $\models$ in different contexts.

For better readability we will sometimes make use of the following notation when the reader may interpret the intersection of two sets I and X of ground atoms as a projection from I to X.

Definition 4. For two sets I and X of ground atoms, $I|_X = I \cap X$ is the *projection* of I to X. ♢

3.3 Syntax of Disjunctive Logic Programs

In order to give a first glance at answer-set programming, we introduce disjunctive logic programs, which can be seen as a basic theoretical fragment of the ASP languages we cover in this thesis.

The central elements in a logic program are *rules* that express logical implications. A rule consists of two components, the rule *body*, representing the antecedent of the implication, and the rule *head*, that represents the consequent of the implication. Intuitively, the presence of a rule in a logic program guarantees that when its body is considered to be true, also its head must be true. Traditionally in logic programming, in contrast to typical notation of implications in logic, the head of a rule appears to the left of the rule body.

Definition 5. A *disjunctive logic program rule*, or LP-*rule* for short, is an expression of the form

$$a_1 \vee \ldots \vee a_k \leftarrow a_{k+1}, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\ , \qquad (3.1)$$

where every a_i for $1 \leq i \leq n$ is an atom. The operators $\vee$ and not denote *disjunction* and *default negation*, respectively, whereas commas represent *conjunctions*. ◊

Note that by Definition 1, an LP-rule is ground iff all atoms a_i are ground for $1 \leq i \leq n$.

We next formalise the different parts of an LP-rule, using set notation, and introduce some syntactic properties.

Definition 6. Let r be an LP-rule of form (3.1). The set

$$\mathrm{H}(r) = \{a_1, \ldots, a_k\}$$

is the *head* of r, while

$$\mathrm{B}(r) = \{a_{k+1}, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$$

is the *body* of r. We also differentiate between the *positive* and the *negative body* of r, given by

$$\mathrm{B}^+(r) = \{a_{k+1}, \ldots, a_m\}$$

and

$$\mathrm{B}^-(r) = \{a_{m+1}, \ldots, a_n\},$$

respectively. We call r *normal* if $k = 1$, *positive* if $\mathrm{B}^-(r) = \emptyset$, and *Horn* if it is both, normal and positive. Moreover, r is a constraint if $\mathrm{H}(r) = \emptyset$ and a *fact* if it is normal and $\mathrm{B}(r) = \emptyset$. ◊

Definition 7. A *disjunctive logic program*, or LP-*program* for short, is a set of LP-rules. An LP-program P is *ground*, *normal*, *positive*, or *Horn*, respectively, if every LP-rule in P is ground, normal, positive, or Horn, respectively. Finally, the *Herbrand universe* HU_P of P is the set of all ground functions containing only function-symbols appearing in P. ◊

Note that the Herbrand base of P can be defined as the Herbrand base with respect to PR and HU_P, where PR is the set of predicates appearing in P. As we define grounding using substitutions (see Definition 9) we do not make direct use of this notion.

3.3.1 A Note on Strong Negation

Gelfond and Lifschitz (1991) introduced their semantics for *extended disjunctive databases*, which correspond to LP-programs with *strong negation* (also called "classical negation" with some abuse of language), i.e., a second form of negation, besides default negation, that allows for expressing that some statement is explicitly known to be false. We do not formally introduce strong negation in this thesis, as this would complicate notation and does not affect the techniques we developed. However, we consider strong negation to be syntactic sugar of solver languages, give an informal explanation in Section 3.6.1, and cover it in our implementation.

3.3.2 Grounding

Gelfond and Lifschitz originally introduced the stable-model semantics for programs with variables based on the stable-model semantics for ground programs (Gelfond and Lifschitz, 1988, 1991). They defined answer sets of an LP-program P to be the answer sets of the so-called *grounding* of P. The idea is to replace every LP-rule r by a relevant set of ground rules, that are obtained from r by substituting variables by ground terms. LP-rules obtained in that way are called *ground instances* of r.

Although, direct definitions of answer sets on the first-order level have been introduced and became a popular research topic, the majority of answer-set solvers (notable exceptions are `ASPeRiX` (Lefèvre and Nicolas, 2009) and `OMiGA` (Dao-Tran et al., 2012)) follow the original theoretic approach for dealing with variables: they depend on a grounding step in which all variables are eliminated before solving. However, as we will further elaborate on in Section 6.1, the choice of ground instances that are considered relevant, i.e., are part of the grounding, differs among different grounding tools or strategies.

In the following we define a notion of grounding that can be considered to be ideal as it takes all ground instances with respect to $\mathscr{A}$ into account. To distinguish it from the outcome of more involved grounding techniques that aim at finite small groundings (as used in practise), we call it *naïve grounding*.

Definition 8. Let $V \subseteq \mathcal{V}$ be a set of variables and $F \subseteq \mathrm{HU}_{\mathscr{A}}$ be a set of ground terms. A V/F-substitution is a function $\theta : V \to F$. Let e be an expression. By $e\theta$ we denote the expression resulting from e by replacing each variable $X \in V$ appearing in e by $\theta(v)$. $\Diamond$

Definition 9. Let $F \subseteq \mathrm{HU}_{\mathscr{A}}$ be a set of ground terms. Let r be an LP-rule. The *grounding* of r with respect to F, denoted by $gr_F(r)$, is the set of all rules $r\theta$ where θ is some $\mathcal{V}/F$-substitution.

Let P be an LP-program. The *grounding* of P with respect to F is given by

$$gr_F(P) = \bigcup_{r \in P} gr_F(r).$$

The *naïve grounding* of P, $gr_{\mathsf{n}}(P)$, is the set $gr_{\mathrm{HU}_{\mathscr{A}}}(P)$. $\Diamond$

In the literature on ASP, including the seminal paper by Gelfond and Lifschitz (1988), it is often assumed that an alphabet is given implicitly through the expressions used in a logic program, and hence typically $gr_{\mathrm{HU}_P}(P)$, i.e., the grounding with respect to the Herbrand universe of P, is considered to be the (naïve) grounding of an LP-program P.

Note that also in the case of an explicit alphabet, logic programs are often required to be *safe*, i.e., variables appearing in the head or the negative body of an LP-rule are required to appear in positive body or r.[1] This restriction ensures that the answer sets of $gr_{\mathrm{HU}_P}(P)$ coincide with those of $gr_{\mathrm{HU}_{\mathscr{A}}}(P)$. This property is sometimes referred to as *domain independence*.

Example 2. In order to demonstrate the naïve grounding, we consider program P_{Ex2} for finding three colourings of a given graph:

$$\begin{aligned} P = \{&edge(a,b) \leftarrow, \\ &node(X) \leftarrow edge(X,Y), \\ &node(Y) \leftarrow edge(X,Y), \\ &col(X,red) \vee col(X,green) \vee col(X,blue) \leftarrow node(X), \\ &\leftarrow edge(X,Y), col(X,C), col(Y,C)\}. \end{aligned}$$

The first rule is a fact that encodes the graph for which we want to find a three colouring. In this case it is the graph with two nodes, labelled a and b, such that there is an edge from a to b. The next two rules identify nodes that have in- or outgoing edges. The disjunctive rule of P_{Ex2} guesses for each such node whether it is coloured in red, green, or blue. Finally, the constraint assures that adjacent nodes are of different colour.

Although the considered graph has only two nodes and one edge, the naïve grounding of P_{Ex2} is already huge, consisting of 166 rules in total. In fact, $gr_{\mathsf{n}}(P_{Ex2}) = P'_{Ex2} \cup P''_{Ex2}$, where P'_{Ex2} is given below and P''_{Ex2} consists of ground rules that have some of the constants

[1] The notion of safety has evolved from a similar notion, called *allowedness*, for deductive databases (Topor and Sonenberg, 1988; Sonenberg and Topor, 1988).

for denoting colors, red, $green$, and $blue$ in some predicate argument other than the second position in predicate $col/2$. For example, P''_{Ex2} contains the rule

$$node(a) \leftarrow edge(a, blue)$$

refering to the atom $edge(a, blue)$ that does not represent a meaningful statement about the problem domain. Due to their large number, we do not list the other rules of P''_{Ex2}.

$$\begin{aligned} P'_{Ex2} = \{ & edge(a,b) \leftarrow \\ & node(a) \leftarrow edge(a,a), \\ & node(a) \leftarrow edge(a,b), \\ & node(b) \leftarrow edge(b,a), \\ & node(b) \leftarrow edge(b,b), \\ & col(a, red) \vee col(a, green) \vee col(a, blue) \leftarrow node(a), \\ & col(b, red) \vee col(b, green) \vee col(b, blue) \leftarrow node(b), \\ & \leftarrow edge(a,a), col(a,a), col(a,a), \\ & \leftarrow edge(a,a), col(a,b), col(a,b), \\ & \leftarrow edge(b,a), col(b,a), col(a,a), \\ & \leftarrow edge(b,a), col(b,b), col(a,b), \\ & \leftarrow edge(b,b), col(b,a), col(b,a), \\ & \leftarrow edge(b,b), col(b,b), col(b,b)\} \end{aligned}$$

■

3.4 Answer-Set Semantics

In the section, we introduce the answer-set semantics of LP-programs following Gelfond and Lifschitz (1991). Later, we will switch to an alternative definition of answer sets by Faber et al. (2004, 2011) that became known as FLP-semantics and coincides with the original notion of answer-sets in many important classes of logic programs, including LP-programs. A discussion on why we build on the FLP-semantics (essentially for a high solver compatibility) is given later in Section 4.7.3. Prior to the definition of semantics, we next clarify how we handle a notional vagueness in this work.

3.4.1 Answer Sets vs. Stable Models

The terms “answer set” and “stable model” are often used interchangeably in the literature. As the answer-set programming paradigm can be used also with other semantics than the stable-model semantics one could see “answer set” as referring to a preferred model in the context of the ASP paradigm, whereas “stable model” refers to a concrete semantics, the stable-model semantics for logic programs (and its extensions, respectively).

Within this work we use the term “answer set” for preferred models of ASP languages that we introduce but we will also use the term “stable model” in the context of semantics introduced by others whenever it is used in their respective works.

3.4.2 Satisfaction and Models of LP-programs

Definition 10. Let r be a ground LP-rule and I an interpretation. I *satisfies* the body of r, symbolically $I \models \mathrm{B}(r)$, if

- $I \models a$ for all $a \in \mathrm{B}^+(r)$ and
- $I \not\models a$ for all $a \in \mathrm{B}^-(r)$.

I satisfies r, symbolically $I \models r$, if $I \models \mathrm{B}(r)$ implies that $I \models a$ for some atom $a \in \mathrm{H}(r)$. Whenever $I \models \mathrm{B}(r)$, we call r *active* under I.

I is a *model* of a ground LP-program P, denoted as $I \models P$, if $I \models r$ for each $r \in P$. Moreover, I is a *minimal model* of P, if

- $I \models P$ and
- there is no $I' \subset I$ such that $I' \models P$. $\Diamond$

3.4.3 Gelfond-Lifschitz Reduct

The stable-model semantics was first defined for normal LP-programs (Gelfond and Lifschitz, 1988), using a transformation, the *Gelfond-Lifschitz reduct*, that reduces a normal to a Horn LP-program for a given interpretation. It was later extended (Gelfond and Lifschitz, 1991) to reduce an LP-program with disjunctions to a positive LP-program as follows.

Definition 11 (Gelfond and Lifschitz, 1991). Let P be a ground LP-program and I an interpretation. Then, the *Gelfond-Lifschitz reduct* of P with respect to I, denoted by P^I_{GL} is obtained from P by deleting

1. each LP-rule r, where $\mathrm{B}^-(r) \cap I \neq \emptyset$ and
2. each default negated atom from the bodies of the remaining rules. $\Diamond$

3.4.4 Answer Sets

Definition 12 (Gelfond and Lifschitz, 1991). Let P be an LP-program. An interpretation I is an *answer set* of P if it is a minimal model of $gr_{\mathsf{n}}(P)^I_{\mathrm{GL}}$. $\Diamond$

Example 3. Consider program P_{Ex2} from Example 2. As the program does not involve default negation we have that $gr_{\mathsf{n}}(P_{Ex2})^I_{\mathrm{GL}} = gr_{\mathsf{n}}(P_{Ex2})$ for every interpretation I. Consequently, the answer sets of P_{Ex2} are given by the minimal models of $gr_{\mathsf{n}}(P_{Ex2})$:

$$\begin{aligned}
I_1 =&\{edge(a,b), node(a), node(b), col(a, blue), col(b, green)\},\\
I_2 =&\{edge(a,b), node(a), node(b), col(a, blue), col(b, red)\},\\
I_3 =&\{edge(a,b), node(a), node(b), col(a, green), col(b, blue)\},\\
I_4 =&\{edge(a,b), node(a), node(b), col(a, green), col(b, red)\},\\
I_5 =&\{edge(a,b), node(a), node(b), col(a, red), col(b, blue)\},\\
I_6 =&\{edge(a,b), node(a), node(b), col(a, red), col(b, green)\}.
\end{aligned}$$

Now, consider the program

$$\begin{aligned}
P_{Ex3} = \{&a \leftarrow not\ b,\\
&b \leftarrow not\ a\}
\end{aligned}$$

and interpretation $I_1 = \{a\}$. Then,

$$gr_{\mathsf{n}}(P_{Ex3})^{I_1}_{\mathrm{GL}} = \{a \leftarrow\}$$

and, as I_1 is a minimal model of $gr_{\mathsf{n}}(P_{Ex3})^{I_1}_{\mathrm{GL}}$, we have that I_1 is an answer set of P_{Ex3}.

For interpretation $I_2 = \{a, b\}$ we have $gr_{\mathsf{n}}(P_{Ex3})^{I_2}_{\mathrm{GL}} = \emptyset$. I_2 is not an answer set of P_{Ex3} as $\emptyset \subseteq I_2$ is a model of $gr_{\mathsf{n}}(P_{Ex3})^{I_2}_{\mathrm{GL}}$.

Finally, $I_3 = \emptyset$ is not an answer set of P_{Ex3} as it is not a model of

$$\begin{aligned}
gr_{\mathsf{n}}(P_{Ex3})^{I_3}_{\mathrm{GL}} = \{&a \leftarrow,\\
&b \leftarrow\}.
\end{aligned}$$ ■

3.5 Extensions of Syntax and Semantics

Much research has been carried out that is devoted to extensions of logic programs under the stable-model semantics. In what follows, we will describe the mathematical counterparts of popular features that are often used in the languages of answer-set solvers. These are in particular weight constraints, aggregates, and external atoms. They all amount to special *literals* that can appear in rules of logic programs. In this work, we use the term "literal" informally, for referring to an expression that may appear in the head or the body of a rule in a logic programming language and that can be either true or false under an interpretation in case it is ground. In this sense, atoms and default negated atoms are literals. In order to distinguish them from other literals we call them *standard literals*. Both aggregates and external atoms were introduced using the FLP-semantics that we build our work on. Therefore, we introduce program classes with these constructs and define their semantics in Section 3.5.4. Note that in this section, we do not consider syntactic restrictions for domain independence.

3.5.1 Weight Constraints, Cardinality Constraints, and Choice Atoms

We next introduce weight constraints and specialisations thereof, following Simons et al. (2002). Similar to aggregates, weight constraints are special literals whose truth depends on multiple atoms in an interpretation. Unlike aggregates in the approach of Faber et al. (2011) that we introduce in the following section, weight constraints may also appear in the head of rules in the work of Simons et al. Formally, we only define the ground variant of weight constraints as in the original paper, and discuss weight constraints with variables as used in the language of `Gringo` in Section 3.6.2.

Definition 13 (Simons et al., 2002). A *weight constraint* is an expression of form

$$l\,[a_1 = w_1, \dots, a_k = w_k, \mathit{not}\ a_{k+1} = w_{k+1}, \dots, \mathit{not}\ a_n = w_n]\,u\,, \tag{3.2}$$

where each a_i is a ground atom and each weight w_i is a real number, for $1 \leq i \leq n$. The lower bound l and the upper bound u are either a real number, ∞, or $-\infty$. $\Diamond$

Despite this definition, Simons, Niemelä, and Soininen (2002) effectively require weights to be non-negative, as in their semantics negative weights are eliminated in a pre-processing step that has been claimed to lead to unintuitive results in several works (Ferraris and Lifschitz, 2005; Ferraris, 2011).

Intuitively, for a weight constraint of form (3.2) to be true, the sum of weights w_i of those atoms a_i, $1 \leq i \leq k$, that are true and the weights of the atoms a_i, $k < i \leq n$, that are false must lie within the lower and the upper bound. More formally, the truth of weight constraints is defined as follows.

Definition 14 (Simons et al., 2002). A weight constraint of form (3.2) is *true* under interpretation I if

$$l \leq (\sum_{1 \leq i \leq k, a_i \in I} w_i + \sum_{k < i \leq n, a_i \notin I} w_i) \leq u\,,$$

otherwise it is *false* under I. $\Diamond$

A special form of a weight constraint is a *cardinality constraint* where all weights are 1 and all contained atoms are different ground atoms. Intuitively, this has the effect that if the cardinality constraint is true, its lower and upper bounds define how many of the contained atoms may be true in an answer set.

A further specialised form of a cardinality constraint is a *choice atom* that is of the form

$$0\,[a_1 = 1, \dots, a_k = 1]\,k\,,$$

hence upper bound and lower bound are fixed. Clearly, choice atoms are useless when they appear in rule bodies, as they are always true. However, they are often used in the head of a rule for non-deterministically guessing a subset of its domain $\{a_1, \ldots, a_k\}$.

Example 4. Instead of disjunction as in Example 2, the following rule uses a cardinality constraint to express that exactly one of the colours red, green, and blue shoud be assigned to the node a.

$$1\ [col(a, red) = 1, col(a, green) = 1, col(a, blue) = 1]\ 1 \leftarrow node(a). \qquad \blacksquare$$

3.5.2 Aggregates

We next define aggregates following Faber et al. (2004, 2011) with the slight difference that functions are restricted to constants in the original papers.

Definition 15 (Faber et al., 2011). A *symbolic set* is a pair $\langle \mathbf{V} : \mathrm{B} \rangle$, where $\mathbf{V}$ is a list of variables and B is a conjunction of atoms. A *ground set* is a set of pairs of the form $\langle \mathbf{T} : \mathrm{B} \rangle$, where $\mathbf{T}$ is a list of ground terms and $\mathrm{B} \subseteq \mathrm{B}^{\mathcal{P}}_{\mathrm{HU}_{\mathscr{A}}}$ is a set of ground atoms. A *set term* is either a symbolic set or a ground set.

An *aggregate function symbol* f represents a mapping ε_f from multisets of ground terms to ground terms. An *aggregate function* is of the form $f[\mathbf{ST}]$, where $\mathbf{ST}$ is a set term and f is an aggregate function symbol. An *aggregate atom* is an expression of form $f[\mathbf{ST}] \prec t$, where $f[\mathbf{ST}]$ is an aggregate function, t is a term called *guard*, and $\prec$ is a comparison operator on ground terms. ♢

Faber et al. allow aggregates to appear in rule bodies but not in rule heads.

Definition 16 (Faber et al., 2011). A *disjunctive logic programming rule with aggregates*, or AG-*rule*, is an expression of the form

$$a_1 \vee \ldots \vee a_k \leftarrow a_{k+1}, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, \tag{3.3}$$

where every a_i, for $1 \leq i \leq k$, is an atom and every a_j, for $k + 1 \leq j \leq n$, is either an atom or an aggregate atom. A variable in an AG-rule r is *local* with respect to r if it appears only in some aggregate function of r, otherwise it is global with respect to r.

A *disjunctive logic program with aggregates*, or AG-*program* for short, is a set of AG-rules.[2] ♢

Similarly as for atoms, the semantics of aggregates is only defined for aggregate atoms that are ground. The corresponding grounding step required is a bit more involved than the naïve grounding of LP-rules. In particular, it differentiates between *local* and global variables, i.e., the grounding of an aggregate-atom can only be determined with respect to an AG-rule.

Definition 17 (Faber et al., 2011). Let r be an AG-rule, G the set of global variables of r, $\mathbf{S}$ a symbolic set, L the set of local variables of r in $\mathbf{S}$, and $F \subseteq \mathrm{HU}_{\mathscr{A}}$ a set of ground terms. A G/F-substitution is a *global F-substitution* for r, whereas an L/F-substitution is a *local F-substitution* for $\mathbf{S}$ with respect to r.

For a symbolic set $\mathbf{S} = \langle \mathbf{V} : \mathrm{B} \rangle$ without global variables, the F-instantiation of $\mathbf{S}$ with respect to r is the ground set

$$\mathrm{INST}_{F,r}(\mathbf{S}) = \{\langle \mathbf{V} : \mathrm{B} \rangle\theta \mid \theta \text{ is a local } F\text{-substitution for } \mathbf{S} \text{ with respect to } r\}.$$

A *ground instance* of an AG-rule r with respect to F is obtained in two steps:

[2] In the paper of Faber et al., AG-programs are called $\mathbf{DLP}^{\mathcal{A}}$ programs.

1. a global F-substitution θ for r is first applied over r and
2. every symbolic set $\mathbf{S}$ in $r\theta$ is replaced by $\mathrm{INST}_{F,r}(\mathbf{S})$.

The *grounding* of r with respect to F is the set $gr_F(r)$ of all ground instances of r with respect to F .

Let P be an AG-program. The *grounding* of P with respect to F is given by

$$gr_F(P) = \bigcup_{r \in P} gr_F(r).$$

We call $gr_{\mathrm{HU}_{\mathscr{A}}}(P)$ the *naïve grounding* of P, denoted by $gr_{\mathsf{n}}(P)$. ◊

Definition 18 (Faber et al., 2011). Given an interpretation I and a ground set $\mathbf{G}$, $I_{[\mathbf{G}]}$ is the multiset

$$[t_1 \mid \langle t_1, \ldots, t_n : \mathrm{B} \rangle \in \mathbf{G}, \mathrm{B} \subseteq I] \,.$$

An aggregate atom $f[\mathbf{G}] \prec t$ appearing in the naïve grounding of an AG-rule is *true* under I if $\varepsilon_f(I_{[\mathbf{G}]})$ is defined and $\varepsilon_f(I_{[\mathbf{G}]}) \prec t$, otherwise it is *false* under I. ◊

Note that a ground aggregate atom could contain a symbolic set with an empty variable list. Like for aggregate atoms that are not ground, the semantics for these aggregate atoms is only given indirectly in terms of their grounded versions and is always true or always false, depending on the aggregate function. We only consider ground AG-rules and AG-programs that are obtained from the naïve grounding of some AG-program.

The Definitions 6 and 10 (on pages 17 and 19) for defining rule parts of LP-rules and the semantics of LP-rules and LP-programs carry over to AG-rules and AG-programs in the obvious way.

Example 5 (Faber et al. (2011)). The aggregate atom

$$max[\{\langle 2 : r(2), a(2,x) \rangle, \langle 2 : r(2), a(2,y) \rangle\}] > 1$$

could be the result of grounding the non-ground aggregate atom

$$max[\langle Z : r(Z), a(Z,V) \rangle] > Y.$$ ■

The answer-set semantics for AG-programs as defined by Faber et al. (2011) is introduced in Section 3.5.4. Further examples for aggregates, in the context of the language of DLV, are given in Section 3.6.3.

3.5.3 External Atoms

In order to integrate external sources of computation to answer-set programs, Eiter et al. (2005) introduced HEX programs which are logic programs augmented with so-called *external* and *higher-order atoms*. The latter are atoms which allow for variables at the predicate position. Indeed, the HEX formalism does not differentiate between constants and predicate symbols. Note that higher-order atoms can be simulated by ordinary atoms with fresh predicates and a further argument. We do not need higher-order atoms for the purposes of this thesis. However, we next introduce external atoms.

Definition 19. An *external atom* is an expression of the form

$$\#g[Y_1, \ldots, Y_n](X_1, \ldots, X_m), \tag{3.4}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms, called the input, respectively, the output list of the external atom, and $\#g$ is an *external predicate name* with associated lengths

n and m. Moreover, it is assumed that every such external predicate name has an associated $(n+m+1)$-ary Boolean function $f_{\#g}$ assigning each tuple $\langle I, y_1, \dots, y_n, x_1, \dots, x_m \rangle$ either 0 or 1, where I is an interpretation, and all y_i and x_j are ground terms for $1 \leq i \leq n$ and $1 \leq j \leq n$.

A ground external atom $\#g[y_1, \dots, y_n](x_1, \dots, x_m)$ is *true* under interpretation I, symbolically $I \models \#g[y_1, \dots, y_n](x_1, \dots, x_m)$, if, $f_{\#g}(I, y_1, \dots, y_n, x_1, \dots, x_m) = 1$, otherwise it is *false* under I. ♢

Intuitively, the function $f_{\#g}$ represents some source of computation that is external to an answer-set solver.

Definition 20 (Eiter et al., 2005). A *disjunctive logic programming rule with external atoms*, or EX-*rule*, is an expression of the form

$$a_1 \vee \ldots \vee a_k \leftarrow a_{k+1}, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n, \qquad (3.5)$$

where every a_i for $1 \leq i \leq k$ is an atom and every a_j for $k+1 \leq j \leq n$ is either an atom or an external atom.

A *disjunctive logic program with external atoms*, or EX-*program* for short, is a set of EX-rules. ♢

The grounding of EX-programs is similarly defined as the grounding of LP-programs, i.e., the naïve grounding for EX-programs is defined analogous to Definition 9 for LP-programs on page 18. Moreover, as for instantiated AG-programs, we assume the concepts introduced in Definitions 6 and 10 (on pages 17 and 19) to be available for EX-rules and EX-programs as well, using obvious extensions. Furthermore, also as in the case of AG-programs, an answer-set semantics for EX-programs is given in terms of the FLP-semantics that is introduced in the following section.

3.5.4 FLP-Semantics

The FLP-semantics has been introduced to provide an intuitive handling of recursive aggregates in answer-set programming (Faber et al., 2004, 2011). More specifically, it was defined for AG-programs as defined in Section 3.5.2. Similar to the original definition of answer-sets by Gelfond and Lifschitz, Faber et al. make use of a program reduct depending on a candidate interpretation I for determining whether I satisfies a stability criterion, i.e., is considered an answer set. However, the reduct of Faber, Pfeifer, and Leone differs in spirit from that of Gelfond and Lifschitz as it does not reduce the program to another syntactic class (cf. Definition 11 on page 20: the Gelfond-Lifschitz reduct of an LP-program is always positive). Instead, the so-called *FLP-reduct*, defined next, keeps the individual rules intact and just ignores all rules that are not active under the candidate interpretation.

Definition 21 (Faber et al., 2011; Eiter et al., 2005). Let I be an interpretation and P a ground AG-program, respectively a ground EX-program. The FLP-reduct of P with respect to I is given by

$$P^I = \{r \in P \mid r \text{ is active under } I\}$$ ♢

Based on this notion of a reduct, the FLP-semantics is defined as follows.

Definition 22 (Faber et al., 2011; Eiter et al., 2005). Let I be an interpretation and P an AG-program, respectively an EX-program. I is an *FLP-answer set* of P if I is a minimal model of $gr_{\mathsf{n}}(P)^I$. ♢

Notice that both AG-programs and EX-programs are proper syntactic extensions of LP-programs. It was shown that an interpretation is an answer set of an LP-program P according to Definition 12 (on page 20) if and only if it is an FLP-answer set of P (Faber et al., 2011).

The FLP-semantics is fully implemented for HEX programs (Eiter et al., 2005) in the solver `DLVHEX` (Eiter et al., 2006) that is discussed in Section 3.6.4.

3.6 ASP (Solver) Languages

The different origins of answer-set programming, discussed in the beginning of this chapter, as well as most efforts to extend and further develop the paradigm are of scientific nature. That is, up to now, the key driver of the field of ASP is the academic community rather than for example the demand by software developers or industry. This might change in the future, due to the availability of efficient answer-set solvers and, as discussed in Chapter 1, hopefully also due to contributions to software engineering in ASP such as this thesis. Nevertheless, because of the scientific roots of ASP and the lack of a single core driver there is no canonical ASP programming language. Instead, papers on ASP use different syntaxes and also different answer-set solvers feature a different input language.[3] As motivation for the abstract ASP language we use for our stepping framework, we want to highlight two aspects of answer-set programming Babylonia:

1. differences between solver dialects that are discussed in this section on the one hand but also

2. the gap between mathematical syntax and semantics of answer-set programs as used in research, opposed to the solver languages that are used in ASP applications, that we also deal with in the following but also in Section 6.1.

The next section deals with commonalities of solver dialects followed by discussions of the languages of the ASP systems `Gringo`, `DLV`, and `DLVHEX`.

3.6.1 General Remarks

Although there are differences between languages accepted by answer-set solvers they typically share a basic structure.[4] All the solver languages we consider have a syntax that is similar to that of many PROLOG dialects. Essentially, a typical program is a list of rules that are separated by dots (`.`), the implication operator is denoted by `:-`, and, similar as in our mathematical notation, commas (`,`) are used for conjunction in the body of a rule and default negation is denoted by `not`. Typically, the operator `:-` is dropped for facts. Moreover, single-line comments, i.e., text that is ignored by the solver until a line break, typically start with the `%` symbol. In the remainder of the thesis, we will sometimes present ASP source code, i.e., answer-set programs in the input language of some solver. In order to distinguish it from answer-set programs in mathematical notation, source code is displayed in `verbatim` font and, for longer listings, embedded in a dedicated bordered environment that also indicates the used solver language and a filename for referencing the answer-set program.

Example 6. The following listing shows a `Gringo` program with a single-line comment and two rules.

[3]Note, however, that there are endeavours towards a common core input language (Calimeri et al., 2012) in the realm of the ASP competitions.

[4]When we speak of solver languages, we usually refer to a human-readable input language, i.e., for solvers that require external grounding (cf. Section 3.6.2) we mean the input language of the grounder.

```
ex6.gr                                                    Gringo
% this is some example gringo source code
a :- not b.
b :- not a.
```

Note that this program would, e.g., also be a valid DLV or DLVHEX program as it contains only features that these languages share. ■

We will sometimes use the + operator as a means for referring to compositions of programs with explicit filename, e.g., `ex6.gr + ex.gr` refers to the program obtained from joining program `ex6.gr` with another program of filename `ex.gr`.

While the domain of discourse of our formal ASP language is determined by the Herbrand universe of $\mathscr{A}$, solver languages typically define a grammar for identifiers, a range of integers, and quoted strings that act as the constants of the language.

Moreover, some languages allow for uninterpreted function symbols of higher arity. Solvers (or grounders) usually define an order over all ground terms and allow for special comparison literals that are true when one term is greater than, less than, equal to, respectively, unequal to another term with respect to the order. For the integer range of the solver domain, the order coincides with the natural ordering of the integers. Moreover, integer arithmetics are available in many solver languages, either using special predicates or interpreted arithmetic functions, e.g., addition or multiplication. A handy feature available in all popular solver languages is the anonymous variable, denoted by the underscore (_). Every occurrence stands for a fresh variable that is not used anywhere else in the program.

As mentioned earlier, ASP solver languages feature strong negation, a second form of negation that allows for expressing that an atom is known to be false. An atom is negated by using the unary operator - (sometimes also denoted by ~).

Example 7. The following program is a variant of the bird example that is often used for illustrating default logic, expressing that by default birds are assumed to fly.

```
ex7.dlv                                                   DLV
bird(waldo).
bird(tux).
penguin(tux).
flies(X) :- bird(X), not -flies(X).
-flies(X) :- penguin(X).
```

The first three facts state that there are two birds, Waldo and Tux, where Tux is also known to be a penguin. The fourth rule expresses that if variable `X` stands for a bird and it is not known that `X` does not fly it is assumed that `X` flies. Hereby, the strong negated atom `-flies(X)` stands for the information that `X` is known not to fly; with additional default negation, the literal `not -flies(X)` means that `X` is not known not to fly. The final rule expresses that when `X` is a penguin then `X` is known not to fly. The program has one answer set:

```
{bird(waldo), bird(tux), penguin(tux), flies(waldo),
-flies(tux)}
```

Besides the information from the facts of the program, the answer set contains `flies(waldo)` and `-flies(tux)`, encoding that Waldo flies, but Tux, the penguin, does not. ■

As the example shows, strongly negated ground atoms can occur in interpretations and also answer sets. Indeed, strongly negated atoms can be seen and are often implemented like ordinary

atoms, with the only difference that a ground atom and its strong negation can never occur in the same answer set.

Example 8. The following program extends the one from Example 7 by a fact stating that Tux flies.

```
ex8.dlv                                              DLV
bird(waldo).
bird(tux).
penguin(tux).
flies(tux).
flies(X) :- bird(X), not -flies(X).
-flies(X) :- penguin(X).
```

The program has no answer set as the rules would enforce `flies(tux)` and `-flies(tux)` to be true. ■

Another commonality of solver languages is that they impose syntactic restrictions on programs which ideally lead to finite small groudings that are equivalent to the respective program and can be efficiently computed. Such restrictions include ω-restrictedness(Simons et al., 2002), λ-restrictedness (Gebser et al., 2007b), stratification(Apt et al., 1988; Van Gelder, 1989; Naqvi, 1986), and safety (cf. Section 3.3.2). Varying syntactic restrictions in different languages are one source of complication for developing common development support methods for multiple solvers. Another source are different types of language constructs that we discuss in the subsequent subsections, in particular non-standard literals and different types of statements, i.e., elements of a program that are not rules and used, e.g., for optimisation, filtering of answer sets, or constant assignments. In many cases, the two mentioned sources of complication add up, as, on the one hand, for each solver-specific language construct it must be clarified under what circumstances it fulfils a syntactic restriction, and, on the other hand, such constructs often require restrictions themselves, e.g., they could require that predicates they use have to be stratified.

3.6.2 `Gringo`

The grounding tool `Gringo` (Gebser et al., 2007b, 2011a) is developed as the grounding component for the `Clasp` solver (Gebser et al., 2007a) and further tools from the Potsdam Answer Set Solving Collection (Potassco) (Gebser et al., 2011c).

`Gringo` is the de-facto successor of the `Lparse` grounder written by Tommi Syrjänen at Helsinki University of Technology (which is now part of Aalto University) (Syrjänen, 2002). In current versions of `Gringo`, the input language, which is the language in which ASP developers write their programs, has become a lot richer than that of `Lparse`. The output language of `Gringo` however, is backward compatible to that of `Lparse`. It is a numerical interchange format that compactly represents the grounding and symbolic information, i.e., tables of names of predicates and functions, and is intended to be readable for machines rather than humans. It is sometimes referred to as the *`Smodels` format*, as it is designed as the input language of the `Smodels` solver (Simons et al., 2002). Many others solvers, e.g., `ASSAT` (Lin and Zhao, 2004), `Cmodels` (Lierler, 2005), `SUP` (Lierler, 2011), `GNT` (Janhunen et al., 2006), `sabe`, and `pbmodels` (Liu and Truszczyński, 2006), that depend on external grounding like `Smodels` and `Clasp` have also adopted this intermediate format as their input language. As a consequence, they can also be used in conjunction with `Gringo` as grounding component.

We consider the language of the `Gringo` version 3 series. Next, we describe some of its features.

Weight Constraints, Cardinality Constraints, and Choice Atoms

For one, `Gringo` supports weight constraints similar to those defined in Section 3.5.1.

Example 9. The following rule states that if the combined income of Waldo and Tux lies between 500 and 1000 they can be considered almost poor.

```
ex9.gr                                                    Gringo

income(waldo,300).
income(tux,200).
almostPoor :- 500 [income(waldo,300)=300,
                   income(tux,200)=200] 1000.
```

■

Example 10. The following program contains different rules with cardinality constraints in the head.

```
ex10a.gr                                                  Gringo

0{older(waldo,tux),older(tux,waldo)}1.
1{strongest(waldo),strongest(tux)}2.
0{seasick(waldo),seasick(tux)}2.
```

The first rule expresses that either Waldo is older than Tux, Tux is older than Waldo, or none of them is older than the other. The second rule states that either Waldo is the strongest or Tux is or they both are (in case they are of equal strength). The cardinality constraint in the third rule is a choice atom, guessing for Waldo and Tux whether they are seasick or not.

Note that bounds can be dropped if they do not impose any effective restrictions, hence the program can also be written as

```
ex10b.gr                                                  Gringo

{older(waldo,tux),older(tux,waldo)}1.
1{strongest(waldo),strongest(tux)}1.
{seasick(waldo),seasick(tux)}.
```

Moreover, notice that general weight constraints use square brackets (`[]`), whereas cardinality constraints are denoted with braces (`{}`), indicating a different behaviour when they contain an atoms more than once. That is, `1[a=1,a=1,b=1]1` and `1{a,a,b}1` are not equivalent. In particular, due to the multiset-semantics of weight constraints, `1[a=1,a=1,b=1]1` can never be true when `a` is true as in that case the overall sum is at least 2, violating the lower bound. The cardinality constraint `1{a,a,b}1` amounts to `1{a,b}1`. ■

Note that in the `Gringo` language, variables may appear in weight constraints.

Example 11. For example, the first rule of `ex10b.gr` could have been obtained[5] from the non-ground rule

```
0{older(X,Y),older(Y,X)}1 :- bird(X),bird(Y),X!=Y.
```

given the facts `bird(waldo)` and `bird(tux)`. ■

[5] Here, `Gringo` recognises the rule body to be satisfied and removes it in the grounding.

However, using variables alone is insufficient if we want to dynamically change the number of ground atoms contained in a cardinality constraint, e.g., a non-ground version of the second rule of `ex10b.gr` expressing that for all birds there is at least one who is the strongest. Likewise, we cannot write a rule that has a single rule as instantiation that can guess for every known bird whether it is seasick or not. For such cases, the language uses so-called *conditions* that we present next.

Conditions

The purpose of conditions is to represent a set of ground literals by a literal with variables and a conjunction of literals that determines variable substitutions. It can be seen as a local grounding of a literal within a rule. Conditions are mainly used for dynamically determining the atoms contained in weight and cardinality constraints, however they can also be used directly in rule heads or bodies. Conditions are added to a literal using the colon symbol (`:`).

Example 12. We next illustrate how a non-ground literal with conditions is expanded to an expression with multiple ground literals, depending on whether it occurs in the head or in the body of a rule.

```
ex12.gr                                                    Gringo

bird(tux).
bird(waldo).
birdClass(ratites).
birdClass(tinamous).
birdClass(neognathae).
happy(tux).
allBirdsAreHappy :- happy(X):bird(X).
ofClass(X,C):birdClass(C) :- bird(X).
```

Conditions in rule bodies represent conjunctions, e.g., the literal `happy(X):bird(X)` rule on line 7 is expanded to the conjunction `happy(tux),happy(waldo)` as `bird(X)` is true when `X` is substituted by `tux` or `waldo`. When conditions are used in the head of a rule they are expanded to a disjunction, hence, the rule

```
ofClass(X,C):birdClass(C) :- bird(X).
```

stands for the ground rules

```
ofClass(waldo,neognathae) | ofClass(waldo,tinamous) |
  ofClass(waldo,ratites) :- bird(waldo).
```

and

```
ofClass(tux,neognathae) | ofClass(tux,tinamous) |
  ofClass(tux,ratites) :- bird(tux).
```

Note that `Gringo` uses `|` to denote the disjunction operator and that not all solvers that are based on `Gringo` groundings support disjunctions. ■

The next examples illustrate the use of conditions in weight constraints.

Example 13. The program `ex9.gr` is the ground version of `ex13.gr`.

```
ex13.gr                                                  Gringo
income(waldo,300).
income(tux,200).
almostPoor :- 500 [income(N,I)=I:income(N,I)] 1000.
```

■

Example 14. The following program is a non-ground variant of `ex10b.gr` with additional facts for defining birds.

```
ex14.gr                                                  Gringo
bird(waldo).
bird(tux).
{older(X,Y),older(Y,X)}1 :- bird(X),bird(Y),X!=Y.
1{strongest(X):bird(X)}1.
{seasick(X):bird(X)}.
```

■

As literals with conditions must be expanded during the grounding phase, `Gringo` allows only for stratified predicates to occur in them, as those can be evaluated during grounding.

Pooling

Pooling of arguments are a means for compact representation of rules. Using semicolon (`;`) and double semicolon (`;;`) as operators, alternatives for terms appearing in arguments of functions and predicates can be given. Depending on where in a rule pooling occurs, a literal containing the pooled terms will be expanded to multiple literals within the same rule, or to multiple rules each containing a different version of the literal. In both cases the expanded literals cover all combinations of alternatives defined by pooling. The `;` operator allows for representing alternative terms for one argument, whereas `;;` is used to define alternative lists of arguments.

Example 15. The following program states that two birds are rivals when they are in love with the same bird.

```
ex15a.gr                                                 Gringo
inLoveWith(tux;waldo,tweety).
rivalOf(X,Y;;Y,X) :- inLoveWith(X;Y,Z),X!=Y.
```

Program `ex15a.gr` is a compact representation of the following program.

```
ex15b.gr                                                 Gringo
inLoveWith(tux,tweety).
inLoveWith(waldo,tweety).
rivalOf(X,Y) :- inLoveWith(X,Z),inLoveWith(Y,Z),X!=Y.
rivalOf(Y,X) :- inLoveWith(X,Z),inLoveWith(Y,Z),X!=Y.
```

As the two original rules contain pooling in their heads, each is expanded to two rules. The literal `inLoveWith(X;Y,Z)` is expanded to two literals as it is contained in a rule body. ■

Intervals

Another form of compact representation are interval terms which represent a set of integer constants from a given interval. Similar to pooled arguments, interval terms stand for alternative arguments, however, unlike pooling, intervals are expanded to different rules independent from the position of the interval term in the rule.

Example 16. The example program defines a 2x3 grid of cells that can be ok or faulty. If a cell is ok this is indicated with the predicate `cellOK/2`.

```
ex16a.gr                                                   Gringo

nrOfCols(2).
nrOfRows(3).
col(1..X) :- nrOfCols(X).
row(1..Y) :- nrOfRows(Y).
cellOK(1..2,1..2).
faultyCellExists :- not cellOK(1..X,1..Y), nrOfCols(X),
                                           nrOfRows(Y).
```

A ground version of `ex16a.gr` is given below.

```
ex16b.gr                                                   Gringo

nrOfCols(2).
nrOfRows(3).
col(1) :- nrOfCols(2).
col(2) :- nrOfCols(2).
row(1) :- nrOfRows(3).
row(2) :- nrOfRows(3).
row(3) :- nrOfRows(3).
cellOK(1,1).
cellOK(1,2).
cellOK(2,1).
cellOK(2,2).
faultyCellExists :- not cellOK(1,3), nrOfCols(2),
                                          nrOfRows(3).
faultyCellExists :- not cellOK(2,3), nrOfCols(2),
                                          nrOfRows(3).
```

Note that the program only contains relevant expanded rules. A naïve expansion would also contain rules like the following that are not active with respect to the unique answer set.

```
col(2) :- nrOfCols(3).
col(3) :- nrOfCols(3).
faultyCellExists :- not cellOK(1,1), nrOfCols(2), nrOfRows(3).
```

■

Lua Functions

`Gringo` allows for user-defined interpreted functions, i.e., the programmer can write an imperative piece of code which takes a list of ground terms as input and returns a ground term. The functions are written in the Lua programming language (Ierusalimschy, 2006), a lightweight interpreted scripting language. Besides computing a single ground term as output, embedded

Lua code can also be used to create multiple output values, interact with databases, or create new function-symbols that are not available in the original program's Herbrand universe.

Example 17. The following program demonstrates string concatenation using Lua in Gringo.

```
ex17.gr                                                   Gringo

#begin_lua

function concatName(a, b)
  return a .. " " .. b
end

#end_lua.

firstName(id1,"Ada").
surname(id1,"Lovelace").

firstName(id2,"George").
middleName(id2,"Gordon").
surname(id2,"Byron").

hasMiddleName(ID) :- middleName(ID,_).
fullName(@concatName(F,S)) :- firstName(ID,F),
                     surname(ID,S), not hasMiddleName(ID).
fullName(@concatName(@concatName(F,M),S)) :-
                        firstName(ID,F), middleName(ID,M),
                                                  surname(ID,S).
```

Lua functions are defined in a source block enclosed by the `#begin_lua` and `#end_lua` statements. Note that `..` is the Lua operator for string concatenation and the purpose of the Lua function `concatName(a, b)` is joining two strings with an additional blank space in between. The facts encode first name and surname for two persons, where for the second person also a middle name is given. The following rule defines the auxiliary predicate `hasMiddleName/1` indicating that a middle name is known for the respective person. The last two rules have references to the Lua function using the `@` key symbol. Depending on whether a middle name is given they use the function to concatenate either first name and surname or first name, middle name, and surname of each given person. The program has the single answer set:

```
{firstName(id1, "Ada"), surname(id1, "Lovelace"),
 firstName(id2, "George"), middleName(id2, "Gordon"),
 surname(id2, "Byron"), hasMiddleName(id2),
 fullName("Ada Lovelace"), fullName("George Gordon Byron")}
```

■

Gringo has several further language constructs such as optimisation statements or hide and show statements. For more information the interested reader may consult a comparison between the input languages of Lparse and Gringo (Gebser et al., 2009a) and the user guide of Gringo (Gebser et al., 2010).

3.6.3 DLV

The development of the ASP solver DLV has started as a joint endeavour of the University of Calabria and Vienna University of Technology (Citrigno et al., 1997; Leone et al., 2006). At present, DLV is further developed and maintained by DLVSYSTEM S.R.L., a spin-off company of the University of Calabria. The first version of DLV was available in 1997. DLV is proprietary but its license allows for academic and non-commercial educational use at no cost. While many other solvers depend on external grounding, DLV comes with its own grounding component (Faber et al., 2012). Its original kernel-language is that of function-free disjunctive logic programs, also called *disjunctive datalog*. Disjunction plays an important role in the DLV language, as it is the primary construct for guessing the solution space. This is reflected in the name of the tool, as the V in DLV stands for the disjunction operator.

DLV is in many aspects influenced by methods from deductive databases, e.g., its developers put a strong focus on query answering for which it exploits advanced techniques such as magic-set rewriting. Besides answer-set enumeration, DLV allows for conjunctive queries and offers two modes of query reasoning, *brave* reasoning and *cautious* reasoning. For ground queries, i.e., a conjunction of ground standard literals, DLV returns a boolean value. In particular, a ground query is true under brave reasoning if and only if there is at least one answer set under which the conjunction is true and true under cautious reasoning if and only if the conjunctions holds in every answer set of the program. For queries with variables, DLV computes variable substitutions such that the conjunction holds in one answer set in brave reasoning, respectively all answer sets in the cautious setting. In this thesis, we are not concerned with query answering, as our focus is the ASP paradigm, where output is given in terms of models rather than by answers to queries. Another database related feature of DLV is its ODBC interface that allows for reading and storing information in an external database. Moreover, an extension of DLV, called DLV^{DB}, allows for a tighter integration with external databases, allowing for exploiting optimisation techniques of database management systems (Terracina et al., 2008b,a).

Over the years, the syntax of DLV has continuously been extended. We consider the language of the December 21, 2011, release of DLV. We next describe some of its features.

Handling of Integers

The DLV language supports the use of non-negative integers. However, the use of integer arithmetics requires setting an upper integer limit to a concrete number. This can be done in two ways: either by a command-line argument or by the #maxint statement. The built-in predicate #int/1 holds for all integers in the interval between 0 and the integer limit that was set. It is often used in conjunction with arithmetic predicates to make variables safe.

Example 18. The following program demonstrates the use of integers in DLV.

```
ex18.dlv                                                    DLV
#maxint=5.
successorOf(X,Y) :- Y=X+1, #int(X).
```

The program computes successors of integers. The literal $Y = X + 1$ is an atom of a special built-in predicate for addition. As arithmetic predicates do not make their input arguments safe, we need the literal #int(X), stating that X is an integer, as further body literal. The single answer set of program ex18.dlv, as computed by DLV, is given by

```
{successorOf(0,1), successorOf(1,2), successorOf(2,3),
 successorOf(3,4), successorOf(4,5)}
```

Although 5 is within the defined integer range between 0 and 5, its successor is not computed as 6 is outside the range. As a consequence, setting a too small integer range is a potential source of unexpected program behaviour. ■

Unlike in the Gringo language, integer intervals in DLV can only be used in non-disjunctive facts where the atom is of a unary predicate. That is, DLV handles ranges using dedicated *integer range atoms*, whereas Gringo supports intervals in the form of terms that may occur in arbitrary positions (cf. Example 16).

Aggregate Predicates

Aggregates in DLV are similar to aggregates in AG-programs as defined in Section 3.5.2, i.e., it is checked whether the result of an aggregate function applied on a symbolic set is within the range of given guards. Additionally, DLV allows for another version of aggregates, where the result of an aggregate function applied on a symbolic set is assigned to a variable. The latter type of aggregate is always considered to be true. Moreover, as in AG-programs, aggregates in DLV are restricted to occur in rule bodies. DLV supports the aggregate functions #count, #sum, #times, #min, and #max.

Example 19. The following program is a DLV variant of ex13.gr.

```
ex19a.dlv                                                    DLV

income(waldo,300).
income(tux,200).
almostPoor :- 500 <= #sum{I:income(_,I)} <= 1000.
```

The aggregate function #sum is used to add up all values I that are used as second argument of a true atom of predicate income/2.

The following program illustrates assignment aggregates and counting with DLV aggregates.

```
ex19b.dlv                                                    DLV

bird(tux).
bird(tweety).
bird(roadrunner).
bird(waldo).
inLoveWith(tux,tweety).
inLoveWith(tux,roadrunner).
inLoveWith(woody,tweety).
inLoveWith(woody,roadrunner).
inLoveWith(waldo,tweety).

nrOfRivals(B,NR) :- #count{R:inLoveWith(B,L),
                        inLoveWith(R,L),B!=R}=NR,bird(B).
nrOfRivalRelationships(B,NR) :-
                              #count{R,L:inLoveWith(B,L),
                        inLoveWith(R,L),B!=R}=NR,bird(B).
```

The aggregate function #count returns the cardinality of the (grounded) symbolic set to which it is applied. The example illustrates the role of the list of variables in a symbolic set. The bodies of the defining rules of nrOfRivals/2 and nrOfRivalRelationships/2 only differ in the additional variable L in the symbolic set of the latter. The unique answer set (without the atoms from the facts in the program) is given by:

```
{nrOfRivals(roadrunner,0),
 nrOfRivals(tux,2),
 nrOfRivals(tweety,0),
 nrOfRivals(waldo,2),
 nrOfRivalRelationships(roadrunner,0),
 nrOfRivalRelationships(tux,3),
 nrOfRivalRelationships(tweety,0),
 nrOfRivalRelationships(waldo,2)}
```

While `tux` has only two rivals, `waldo` and `woody`, he has three rival relationships, as he shares the loving of both `tweety` and `roadrunner` with his rival `woody`. ■

3.6.4 DLVHEX

The ASP solver `DLVHEX` is developed at Vienna University of Technology and implements HEX programs (Eiter et al., 2005) (see Section 3.5.3). In the beginning of its development, `DLVHEX` used `DLV` as a backend. However, the current version (we consider version 2.3) allows for further backends with the combination of `Gringo` and `Clasp` as default option. `DLVHEX` has a modular architecture in which language features can be added by using plugins. The core language is similar to that of `DLV`. External atoms in the language of `DLVHEX` start with the ampersand symbol (`&`) followed by an identifier of the external atom, a list of input symbols within square brackets (`[]`), and a list of arguments within brackets (`()`). The semantics of every external atom used in a `DLVHEX` source file has to be determined by some `DLVHEX` plugin stored in a pre-defined directory in the filesystem for a successful evaluation. Plugins offered for the current version of the system include an aggregate plugin that realises aggregate functions, e.g., for counting, building sums, and a script plugin that can execute bash, python, and perl scripts. Moreover, there is a plugin for string manipulations (like concatenation, splitting, or substring checking), one for accessing the WordNet lexical database (Miller et al., 1990), and there is a plugin for explaining inconsistency in multi-context systems (Bögl et al., 2010).

Example 20. The following program is a `DLVHEX` program using external atoms of the string manipulations plugin.

```
ex20.hex                                              DLVHEX

date("2014-05-01").
year(Y)  :- &split[X,"-",0](Y), date(X).
month(M) :- &split[X,"-",1](M), date(X).
day(D)   :- &split[X,"-",2](D), date(X).
```

An external atom with id `split` is true if its argument is obtained from splitting the first element of its input list with the second element of the input list as separator and taking the $i+1$-th resulting substring where i is the value of the third input element. Consequently, `DLVHEX` returns the following answer set for program `ex20.hex`:

```
{day("01"), month("05"), year("2014"), date("2014-05-01")}
```
■

`DLVHEX` is the only solver we consider that allows for non-convex body literals as used in the next example (convexity will be discussed later in Section 4.2).

Example 21. The following program uses the external atom for counting from the aggregate plugin of `DLVHEX`.

```
ex21.hex                                              DLVHEX
p(a) :- not &count[p,mask](1).
p(a) :- p(b).
p(b) :- p(a).
```

The first input element of the external atom specifies the name of the predicate whose atoms shall be counted. The second element, the special constant `mask` determines that the first (and in this case only) argument of predicate `p/1` is unconstrained for counting, i.e., every `p/1` atom is counted. Finally, here, the argument is fixed with constant `1`. Hence, the literal `not &count[p,mask](1)` is true under interpretations that do not have exactly one `p/1` atom as one of its element. The final two rules ensure that `p(a)` and `p(b)` mutually support each other. The resulting single answer set is

```
{p(a), p(b)}.
```

This is a `DLVHEX` variant of the program we will use later in Section 4.7.2 to illustrate the difference of the FLP-semantics (as implemented in `DLVHEX`) to other semantics for non-convex literals. ■

3.7 Computational Complexity

We shortly recall the notions of complexity theory (Papadimitriou, 1994) that are used in this work.

3.7.1 Complexity Classes

A *complexity class* is a set of computational problems of a given *type* that can be solved using a given *machine model* and a given amount of *resources*. As machine models we will consider deterministic and nondeterministic Turing machines (Turing, 1936) and the resources of the complexity classes we encounter are given in terms of computation time. Moreover, we will consider *decision problems*, i.e., problems with a “yes” or “no” answer.

For example, P is the class of decisions problems decidable by a deterministic Turing machine in a polynomial number of time steps, i.e., polynomial in the size of the encoded problem instance. The class of problems decidable by a nondeterministic Turing machine in polynomial time is denoted by NP.

We also make use of *oracle computations*. An oracle for a complexity class C is a hypothetical procedure that solves problems from C in constant time. By P^C or NP^C, respectively, we denote the class of problems that can be solved by a deterministic Turing machine, or a nondeterministic Turing machine, respectively, that has access to an oracle for class C in polynomial time. In particular, we will need the class $\mathrm{NP}^{\mathrm{NP}}$, often denoted as Σ_2^{P}, containing all decision problems that can be solved in polynomial time by a non-deterministic Turing machine with access to an NP-oracle.

3.7.2 Reductions, Hardness, and Completeness

A *polynomial-time many-one reduction* is a transformation that translates every encoded problem instance of a certain problem A to an encoded problem instance of another fixed problem B such that solving the instance of B gives the solution for the instance of A and the transformation could be performed by a deterministic Turing machine in polynomial time.

We call a problem *hard* for some class only if all problems in the class can be reduced to an instance of the problem, using a polynomial-time many-one reduction.

A problem is *complete* for a complexity class, if it is hard for this class and element of it.

4 A Common Formal Basis for Different Solver Languages

Our goal is to develop a debugging method that works for real-world answer-set programs. However, actual solver languages differ from each other and are subject to change as the work on solvers progresses. For this reason, we want our techniques to be as robust as possible to language differences and hence develop them for an abstract ASP language that is general enough to capture different solver languages. One important choice for finding such a language is, which parts of the languages should be abstracted away. Observing how current answer-set solvers work, one can see that although their input languages differ tremendously, answer-set programs, once grounded, have a quite similar representation in different solvers. Therefore, we decided

- to utilise a common abstraction for grounded programs and
- not to abstract grounding away, but make the grounding step part of our abstraction.

The latter point, the abstraction of grounding, is introduced in Chapter 6. In the current chapter, we develop a common abstraction for grounded programs in ASP solver languages. To this end, we can rely on a formalism that was introduced to study different language extensions in a uniform manner: *Abstract-constraint programs* (Marek and Remmel, 2004; Marek and Truszczyński, 2004) are generalised logic programs providing abstractions of commonly used constructs like aggregates, weight constraints, and external atoms. In essence, abstract constraints are dedicated literals whose truth value depends on a set of propositional atoms.

Quite a few different semantics have been explored for abstract-constraint programs (Marek and Remmel, 2004; Marek and Truszczyński, 2004; Son et al., 2007; Shen et al., 2009; Marek et al., 2008; Liu and Truszczyński, 2006; Liu et al., 2010; Shen and You, 2007; Marek and Remmel, 2012), however none of the existing ones fitted all our requirements, namely that grounded `Clasp`, `DLV`, and `DLVHEX` programs can be captured, and that the definitions of the semantics are simple and direct, namely that they operate directly on program rules and that they do not depend on a translation to another formalism. In fact, most of the existing semantics do not cover disjunctions in rule heads that we need for compatibility with `DLV` and `DLVHEX` programs. An exception is that of Shen et al. (2009), whose semantics also handles disjunctions. However, their approach depends on an involved program transformation that introduces fresh atoms and changes rules which make it less attractive for our debugging purposes. Moreover, in the presence of non-convex literals (see Section 4.2), their semantics differs from that of `DLVHEX`, the only solver that currently implements such literals. The only other work on disjunctive abstract-constraint programs that we are aware of is a recent proposal for a semantics by Marek and Remmel (2012) which is in turn not able to express choice rules as discussed in Section 4.7.

The remainder of the chapter is organised as follows. We first introduce abstract-constraint programs and a corresponding satisfiability relation in Sections 4.1 and 4.2. Next, we show how weight constraints, aggregates, and external atoms can be represented as abstract constraints. Our new semantics for abstract-constraint programs is introduced in Section 4.4. It is based

on the FLP-reduct and conservatively extends the FLP-semantics for AG-programs. In the same section, we also discuss why a different, straightforward extension of the FLP-semantics to abstract-constraint programs is not suitable for our purposes. In Section 4.5, we provide three characterisations of our semantics in terms of *unfounded sets*, *unfounded-freeness*, and *external supports*, respectively, generalising the corresponding standard notions. We report on the computational complexity of the semantics in Section 4.6. A detailed analysis of the relation of the semantics introduced in this chapter with previous work is given in Section 4.7. Based on that, we underpin the decision to build on an extension of the FLP-semantics rather than other proposals (see Section 4.7.3) by showing that we gain compatibility with all the solver languages that were discussed in Section 3.6. Thus, we justify that our semantics can be used as a common abstraction of the semantics of these solver languages.

4.1 Syntax of Abstract-Constraint Programs

Instead of atoms, rule heads and bodies of abstract-constraint programs consist of more complex literals, so-called *abstract-constraint atoms*.

Definition 23 (Marek and Remmel, 2004; Marek and Truszczyński, 2004). An *abstract constraint*, *abstract-constraint atom*, or C-*atom*, is a pair $A = \langle D, C\rangle$, where $D \subseteq \mathrm{B}^{\mathcal{P}}_{\mathrm{HU}_{\mathscr{A}}}$ is a finite set called the *domain* of A, denoted by D_A, and $C \subseteq 2^D$ is a collection of sets of ground atoms, called the *satisfiers* of A, denoted by C_A. ♢

We can express atoms also as C-atoms. In particular, for a ground atom a, we identify the C-atom $\langle\{a\}, \{\{a\}\}\rangle$ with a. We call such C-atoms *elementary*.

As for LP-programs, we will also make use of default negation in abstract-constraint programs. An *abstract-constraint literal*, or C-*literal*, is a C-atom A or a default negated C-atom $not\ A$.

Unlike the original definition, we introduce abstract-constraint programs with disjunctive rule heads.

Definition 24. An *abstract-constraint rule*, or simply C-*rule*, is an expression of the form

$$A_1 \vee \cdots \vee A_k \leftarrow A_{k+1}, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\ , \tag{4.1}$$

where $0 \leq k \leq m \leq n$ and any A_i, for $1 \leq i \leq n$, is a C-atom. ♢

As we did for LP-rules, we similarly identify different parts of a C-rule and introduce some syntactic properties.

Definition 25. For a C-rule r of form (4.1),

$$\mathrm{B}(r) = \{A_{k+1}, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$$

is the *body* of r,

$$\mathrm{B}^+(r) = \{A_{k+1}, \ldots, A_m\}$$

is the *positive body* of r,

$$\mathrm{B}^-(r) = \{A_{m+1}, \ldots, A_n\}$$

is the *negative body* of r, and

$$\mathrm{H}(r) = \{A_1, \ldots, A_k\}$$

is the *head* of r. ♢

If $\mathrm{B}(r) = \emptyset$ and $\mathrm{H}(r) \neq \emptyset$, then r is a C-*fact*. For C-facts, we usually omit the symbol "←". A C-rule r of form (4.1) is *normal* if $k = 1$ and *positive* if $m = n$.

For defining the domain of a C-rule, we first define the domain of a default negated C-atom $not\ A$ as $D_{not\,A} = D_A$. Then, the domain D_S of a set S of C-literals is given by

$$D_S = \bigcup_{L \in S} D_L.$$

Finally, the domain of a C-rule r is

$$D_r = \bigcup_{X \in \mathrm{H}(r) \cup \mathrm{B}(r)} D_X.$$

Definition 26. An *abstract-constraint program*, or simply C-*program*, is a finite set of C-rules. A C-program is *normal*, respectively *positive*, if it contains only normal, respectively positive, C-rules. A C-program is *elementary* if it contains only elementary C-atoms. Furthermore, a C-program is *elementary-head* if only elementary C-atoms appear in rule heads. ◊

4.2 Satisfaction Relation

Definition 27. An interpretation I *satisfies* a C-atom $\langle D, C\rangle$, symbolically $I \models \langle D, C\rangle$, if $I|_D \in C$. Moreover, $I \models not\ \langle D, C\rangle$ iff $I \not\models \langle D, C\rangle$. ◊

Important criteria for distinguishing classes of C-atoms are concerned with their semantic behaviour with respect to growing (or shrinking) interpretations. In this respect, we identify several monotonicity properties in the following.

Definition 28. A C-literal L is *monotone* if, for all interpretations I and I', if $I \subseteq I'$ and $I \models L$, then also $I' \models L$. A C-literal L is *anti-monotone* if, for all interpretations I and I', if $I \subseteq I'$ and $I' \models L$, then also $I \models L$. Finally, a C-literal L is *convex* if, for all interpretations I, I', and I'', if $I \subseteq I' \subseteq I''$, $I \models L$, and $I'' \models L$, then also $I' \models L$. Moreover, a C-program P is monotone (respectively, anti-monotone, convex) if for all $r \in P$ all C-literals $L \in \mathrm{H}(r) \cup \mathrm{B}(r)$ are monotone (respectively, anti-monotone, convex). ◊

Notice that when a C-literal is monotone or anti-monotone it is also convex. Next, the notion of satisfaction is extended to C-rules and C-programs in the obvious way.

Definition 29. An interpretation I satisfies a set S of C-literals, symbolically $I \models S$, if $I \models L$ for all $L \in S$. For brevity, we will use the notation $I \models^{\exists} S$ to denote that $I \models L$ for some $L \in S$. Moreover, I satisfies a C-rule r, symbolically $I \models r$, if $I \models \mathrm{B}(r)$ implies $I \models^{\exists} \mathrm{H}(r)$. A C-rule r such that $I \models \mathrm{B}(r)$ is called *active under* I. As well, I satisfies a set P of C-rules, symbolically $I \models P$, if $I \models r$ for every $r \in P$. If $I \models P$, we say that I is a *model* of P. ◊

Analogous to Definition 21 on page 24, we define the FLP-reduct for C-programs.

Definition 30. Let I be an interpretation and P a C-program. The FLP-reduct of P with respect to I is given by

$$P^I = \{r \in P \mid r \text{ is active under } I\}.$$ ◊

4.3 Viewing ASP Constructs as Abstract Constraints

As we want to use abstract constraints as a uniform means to represent common constructs in ASP solver languages, we now show how the language constructs presented in Section 3.5,

namely weight constraints, aggregates, and external atoms, can be expressed by abstract-constraint atoms.

As first construct we consider weight constraints. A weight constraint

$$l\ [a_1 = w_1, \dots, a_k = w_k, not\ a_{k+1} = w_{k+1}, \dots, not\ a_n = w_n]\ u$$

corresponds to the C-atom $\langle D, C \rangle$, where $D = \{a_1, \dots, a_n\}$ and

$$C = \{X \subseteq D \mid l \leq (\sum_{1 \leq i \leq k, a_i \in X} w_i + \sum_{k < i \leq n, a_i \notin X} w_i) \leq u\} .$$

Thus, the domain of the C-atom consists of the atoms appearing in the corresponding weight constraint.

An aggregate atom $f[\mathbf{G}] \prec t$ appearing in the naïve grounding of an AG-rule can be expressed as the C-atom

$$\langle D, \{X \subseteq D \mid \varepsilon_f(X_{[\mathbf{G}]}) \prec t\} \rangle,$$

where

$$D = \bigcup_{\langle \mathbf{t}:\mathrm{B} \rangle \in \mathbf{G}} \mathrm{B}.$$

Similar as for weight constraints, the domain of the C-atom consists of the atoms that appear in the aggregate atom.

Example 22. Consider the aggregate atom $\#count[\mathbf{G}] = 1$, where

$$\mathbf{G} = \{\langle 2 : queen_2_1 \rangle, \langle 2 : queen_2_2 \rangle, \langle 2 : queen_2_3 \rangle, \langle 2 : queen_2_4 \rangle\},$$

stemming from an instantiation of an encoding of the n-queens problem with $n = 4$. Intuitively, the aggregate atom is true if only one queen is located on row 2 of a chessboard. The aggregate function symbol $\#count$ represents the mapping $\varepsilon_{\#count}$ which assigns to a multiset of ground terms its cardinality. Hence, under interpretation $I_1 = \{queen_2_3\}$, we have $I_{1[\mathbf{G}]} = [2]$, and therefore $\varepsilon_{\#count}(I_{1[\mathbf{G}]}) = 1$. Consequently, $\#count[\mathbf{G}] = 1$ is satisfied by I_1. For $I_2 = \{queen_2_3, queen_2_4\}$, we have $I_{2[\mathbf{G}]} = [2, 2]$, and thus $\varepsilon_{\#count}(I_{2[\mathbf{G}]}) = 2$. So, $\#count[\mathbf{G}] = 1$ is not satisfied by I_2. ■

A ground external atom $\#g[y_1, \dots, y_n](x_1, \dots, x_m)$ can be expressed as a C-atom

$$\langle D, \{X \subseteq D \mid f_{\#g}(X, y_1, \dots, y_n, x_1, \dots, x_m) = 1\} \rangle,$$

whose domain is given by

$$D = \{a \mid a \in \mathrm{B}^{\mathcal{P}}_{\mathrm{HU}_{\mathscr{A}}}, I \subseteq \mathrm{B}^{\mathcal{P}}_{\mathrm{HU}_{\mathscr{A}}}, f_{\#g}(I, y_1, \dots, y_n, x_1, \dots, x_m) \neq f_{\#g}(I \cup \{a\}, y_1, \dots, y_n, x_1, \dots, x_m)\}.$$

Here, unlike for weight constraints and aggregates, the domain of the C-atom is not already given in the ground external atom but has to be determined with respect to the Herbrand base of $\mathscr{A}$. The intuition, however is the same, it consists of the atoms of that may influence the truth value of the literal.

4.4 Answer-Set Semantics

Before presenting the definition of our semantics for full abstract-constraint programs, we first restate the FLP-semantics as defined for ground AG-programs and EX-programs in Section 3.5.4 in terms of C-programs. Both AG-programs and EX-programs only allow for atoms in their rule heads, i.e., the corresponding class of C-programs is that of elementary head C-programs. After this reformulation, we discuss the shortcomings of a straightforward extension of the definition to full abstract-constraint programs.

4.4.1 FLP-Semantics for Elementary-Head C-Programs and a Simple Extension

The *FLP-semantics* for elementary-head C-programs is defined as follows.

Definition 31. Let P be an elementary-head C-program. An interpretation I is an *FLP-answer set of* P if it is a minimal model of P^I. The set of all FLP-answer sets of P is denoted by $AS_{FLP}(P)$. $\Diamond$

Note that Definition 22 (on page 24) amounts to Definition 31 when ground AG-programs or EX-programs are viewed as C-programs, i.e., when aggregates and external atoms are expressed as abstract constraints as described in Section 4.3.

Complementing the reduct-based definition, Faber (2005) provided a definition of unfounded sets for AG-programs that we generalise to full abstract-constraint programs later on. Next, we provide the corresponding notion for the fragment of elementary-head C-programs. Note that whilst Faber considers strong negation and partial interpretations, we do not deal with these concepts in our formal framework (cf. Section 3.3.1).

Definition 32 (Faber, 2005). Let P be an elementary-head C-program and I an interpretation. A set X of atoms is *unfounded in* P *with respect to* I if, for each C-rule $r \in P$ with $\mathrm{H}(r) \cap X \neq \emptyset$, one of the following conditions hold:

- $I \not\models \mathrm{B}(r)$,
- $I \setminus X \not\models \mathrm{B}(r)$, or
- $I \models a$, for some $a \in \mathrm{H}(r) \setminus X$. $\Diamond$

Analogous to a result by Faber (2005) for AG-programs, a model I of a C-program P is an FLP-answer set of P iff $I \cap X = \emptyset$, for each unfounded set X for P with respect to I.

Now, let us call the *extended FLP-semantics* the one obtained from Definition 31 by allowing P to be a general abstract-constraint program.

Definition 33. Let P be a C-program. An interpretation I is an *eFLP-answer set of* P if it is a minimal model of P^I. $\Diamond$

This straightforward extension leads to undesired results, however, as we illustrate next.

As stated earlier, it is popular to use choice rules for realising guesses in applied ASP, i.e., rules with choice atoms in their heads. Consider the C-program consisting of the single C-fact

$$\langle \{a, b\}, \{\emptyset, \{a\}, \{b\}, \{a, b\}\} \rangle$$

which corresponds to the choice atom $\{a, b\}$. Here, the intended behaviour of a choice atom, viz. expressing a non-deterministic choice between the sets $\emptyset$, $\{a\}$, $\{b\}$, and $\{a, b\}$, can only be achieved if non-minimal answer sets are permitted. The extended FLP-semantics, however, allows only for the empty set as an eFLP-answer set of this C-program.

We are interested in a notion of an answer set that prevents minimisation between the different satisfiers of an abstract-constraint atom and thus allows for using choice atoms with their usual meaning, which will be introduced in the following.

4.4.2 Basic Definition

The notion of answer sets for abstract-constraint programs defined next provides the semantic foundation for the computation model we use for debugging.

Definition 34. Let P be a C-program and let I be an interpretation. Then, I is an *answer set* of P if

(i) $I \models P$ and

(ii) there is no $I' \subset I$ such that P, I, and I' obey the following relation:

($\star$) for every $r \in P^I$ with $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$.

The set of all answer sets of P is denoted by $AS(P)$. ◊

In order to discuss the intuition behind the definition and in order to compare it to the FLP-semantics, we reformulate Definitions 34 and 33 in the following characterisations:

Proposition 1. *Let P be a* C*-program and I an interpretation. Then, I is an eFLP-answer set of P iff*

(i) $I \models P^I$ *and*

(ii) *there is no interpretation $I' \subset I$ such that for every $r \in P^I$ with $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$.*

Moreover, I is an answer set of P iff

(i) $I \models P^I$ *and*

(ii') *there is no interpretation $I' \subset I$ such that for every $r \in P^I$ with $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$.*

Proof. The proposition for eFLP-answer sets is a slight reformulation of Definition 33. The property of answer sets holds because $I \models P$ iff $I \models P^I$, and if P, I, and I' satisfy Condition ($\star$) of Definition 34, then $I' \models P^I$ holds. □

The reformulation makes explicit that our notion of an answer set differs from that of Definition 33 only by the additional condition that $I'|_{D_A} = I|_{D_A}$. The purpose of this condition is to prevent minimisation within C-atoms. In order to illustrate its intuition, observe that the role of I' in Condition (ii) is to prevent I from being an answer set when all rules remain satisfied when switching from I to I'. The additional condition on I' in Condition (ii') ensures that, in this respect, it only counts that a rule remains satisfied when the atoms it derives are unchanged, i.e., they come from the same satisfier of the same C-atom as under I. This way, different satisfiers have equal opportunities for contributing to an answer set. As we detail in our comparison to related work in Section 4.7, every eFLP-answer set is an answer set and the two notions coincide for elementary-head C-programs.

Example 23. Consider C-program P_1 consisting of the single C-fact

$$\langle \{a,b\}, \{\{a\},\{b\},\{a,b\}\} \rangle$$

that realises a choice of at least one atom from $\{a,b\}$. The answer sets of P_1 are given by $\{a\}$, $\{b\}$, and $\{a,b\}$. However, under the extended FLP-semantics, we would lose the answer set $\{a,b\}$ as, e.g., $\{a\} \subseteq \{a,b\}$ and $\{a\} \models {P_1}^{\{a,b\}}$. ■

Opposed to the extended FLP-semantics for C-programs where such a choice cannot be expressed without introducing auxiliary atoms, we do not enforce subset-minimal answer sets.

The next example illustrates that there are, however, minimisation effects between different C-atoms in a disjunction.

Example 24. Consider the C-program

$$P_2 = \{\langle\{a,b\},\{\{a\},\{b\},\{a,b\}\}\rangle \vee \langle\{a,c\},\{\{a,c\}\}\rangle\}$$

that, as in the previous example, consists of a single (disjunctive) C-fact. This C-program again has answer sets $\{a\}$, $\{b\}$, and $\{a,b\}$. Here, the satisfier $\{a,c\}$ of the second disjunct is not an answer set as for interpretation $\{a\} \subset \{a,c\}$ and C-atom

$$A = \langle\{a,b\},\{\{a\},\{b\},\{a,b\}\}\rangle,$$

we have $\{a\} \models A$ and $\{a\}|_{D_A} = \{a,c\}|_{D_A}$, and thus P_2, $\{a,c\}$, and $\{a\}$ constitute an instance for which ($\star$) holds. ■

Note that minimisation as illustrated by this example only emerges between C-atoms with different domains. A disjunction of different C-atoms having the same domain has the same meaning as a single C-atom with the union of their satisfiers and vice versa, as shown by the following result.

Theorem 1. *Let $A_1 = \langle D, C_1\rangle$, $A_2 = \langle D, C_2\rangle$, and $A = \langle D, C_1 \cup C_2\rangle$ be C-atoms. Furthermore, let P be a C-program and consider C-rules r and r' such that*

(i) $\mathrm{B}(r) = \mathrm{B}(r')$ *and*

(ii) $\mathrm{H}(r) = H \cup \{A_1, A_2\}$ *and* $\mathrm{H}(r') = H \cup \{A\}$, *for some H.*

Then, $AS(P \cup \{r\}) = AS(P \cup \{r'\})$.

Proof. We first show that $AS(P \cup \{r\}) \subseteq AS(P \cup \{r'\})$ by contraposition. Consider some interpretation $I \notin AS(P \cup \{r'\})$. In view of the latter, we proceed by distinguishing two cases.

First, assume that $I \not\models P \cup \{r'\}$. It must hold that $I \not\models r''$ for some $r'' \in AS(P \cup \{r'\})$. If $r'' \in P \cup \{r\}$ we are done as then $I \notin AS(P \cup \{r\})$. Consider the other case that $r'' = r'$. Consequently, we have $I \not\models^{\exists} \mathrm{H}(r')$ and $I \models \mathrm{B}(r')$, and therefore also $I \models \mathrm{B}(r)$ as $\mathrm{B}(r) = \mathrm{B}(r')$. It must hold that $I \not\models A$ and $I \not\models^{\exists} H$. Therefore $I|_D \notin C_1 \cup C_2$ which leads to $I \not\models A_1$ and $I \not\models A_2$. As then $I \not\models^{\exists} \mathrm{H}(r)$ we get $I \notin AS(P \cup \{r\})$.

Now consider the case that there is some $I' \subset I$ such that ($\star$) is satisfied by $P \cup \{r'\}$, I, and I', i.e., for every $r'' \in (P \cup \{r'\})^I$ with $I' \models \mathrm{B}(r'')$, there is some $A' \in \mathrm{H}(r'')$ with $I' \models A'$ and $I'|_{D_{A'}} = I|_{D_{A'}}$. Consider some $r''' \in (P \cup \{r\})^I$ with $I' \models \mathrm{B}(r''')$. It suffices to show that there is some $A' \in \mathrm{H}(r''')$ such that $I' \models A'$ and $I'|_{D_{A'}} = I|_{D_{A'}}$ as then $I \notin AS(P \cup \{r\})$. If $r''' \in (P \cup \{r'\})^I$, this is clearly the case. In the remaining setting, it must hold that $r''' = r$. From $r \in (P \cup \{r\})^I$ and $I' \models \mathrm{B}(r)$, we get $r' \in (P \cup r')^I$ and $I' \models \mathrm{B}(r')$. By ($\star$), we obtain that there is some $A'' \in \mathrm{H}(r')$ with $I' \models A''$ and $I'|_{D_{A''}} = I|_{D_{A''}}$. If $A'' \in H$ we are done as then also $A'' \in \mathrm{H}(r)$. It must hold that $A'' = A$. From that we get that $I'|_D \in C_1 \cup C_2$ and $I'|_D = I|_D$. Then, $I' \models A_1$ or $I' \models A_2$. In either case we are done as $A_1 \in \mathrm{H}(r)$ and $A_2 \in \mathrm{H}(r)$.

We now show $AS(P \cup \{r'\}) \subseteq AS(P \cup \{r\})$ analogously. Consider some interpretation $I \notin AS(P \cup \{r\})$. As above, $I \notin AS(P \cup \{r\})$ yields two cases to consider.

Assume first that $I \not\models P \cup \{r\}$. It must hold that $I \not\models r''$ for some $r'' \in AS(P \cup \{r\})$. If $r'' \in P \cup \{r'\}$ we are done as then $I \notin AS(P \cup \{r'\})$.

Consider the other case that $r'' = r$. Consequently, we have $I \not\models^{\exists} \mathrm{H}(r)$ and $I \models \mathrm{B}(r)$, and therefore also $I \models \mathrm{B}(r')$. It must hold that $I \not\models^{\exists} H$, $I \not\models A_1$, and $I \not\models A_2$. As a consequence, $I|_D \notin C_1 \cup C_2$. Therefore, $I \not\models A$ and hence $I \not\models^{\exists} \mathrm{H}(r')$. As then $I \models r'$ we get $I \notin AS(P \cup \{r'\})$.

Now consider the case that there is some $I' \subset I$ such that ($\star$) is satisfied by $P \cup \{r\}$, I, and I', i.e., for every $r'' \in (P \cup \{r\})^I$ with $I' \models \mathrm{B}(r'')$, there is some $A' \in \mathrm{H}(r'')$ with $I' \models A'$ and

$I'|_{D_{A'}} = I|_{D_{A'}}$. Consider some $r''' \in (P \cup \{r'\})^I$ with $I' \models \mathrm{B}(r''')$. It suffices to show that there is some $A' \in \mathrm{H}(r''')$ such that $I' \models A'$ and $I'|_{D_{A'}} = I|_{D_{A'}}$ as then $I \notin AS(P \cup \{r'\})$. If $r''' \in (P \cup \{r\})^I$, this is clearly the case.

In the remaining setting, it must hold that $r''' = r'$. From $r' \in (P \cup \{r\})^I$ and $I' \models \mathrm{B}(r')$, we get $r \in (P \cup r)^I$ and $I' \models \mathrm{B}(r)$. By ($\star$), we obtain that there is some $A'' \in \mathrm{H}(r')$ with $I' \models A''$ and $I'|_{D_{A''}} = I|_{D_{A''}}$. If $A'' \in H$ we are done as then also $A'' \in \mathrm{H}(r)$. It must hold that $A'' = A_1$ or $A'' = A_2$. From that we get in either case that $I'|_D \in C_1 \cup C_2$ and $I'|_D = I|_D$. Then, we are done as $I' \models A$ and $A \in \mathrm{H}(r')$. □

4.5 Characterisations based on External Support, Unfounded Sets, and Unfounded-Freeness

In this section, we provide three characterisations of our semantics. The characterisations are not only interesting as such but are highly relevant to our framework of computations for stepping that we will introduce in Chapter 5, that is, external support and the complimentary notion of an unfounded set are integral parts of the computation model.

4.5.1 External Support

Often, answer sets are computed following a two-step strategy: First, a model of the program is built, and second, it is checked whether this model obeys a foundedness condition ensuring that it is an answer set. Intuitively, every set of atoms in an answer set must be "supported" by some active rule that derives one of the atoms. Here, it is important that the reason for this rule to be active does not depend on the atom it derives. Such rules are referred to as *external support* (Lee, 2005). In what follows, we extend this notion to our setting.

Definition 35. Let r be a C-rule, X a set of atoms, and I an interpretation. Then, r is an *external support for X with respect to I* if

(i) $I \models \mathrm{B}(r)$,

(ii) $I \setminus X \models \mathrm{B}(r)$,

(iii) there is some $A \in \mathrm{H}(r)$ with $X|_{D_A} \neq \emptyset$ and $I|_{D_A} \subseteq S$, for some $S \in C_A$, and

(iv) for all $A \in \mathrm{H}(r)$ with $I \models A$, $(X \cap I)|_{D_A} \neq \emptyset$ holds. ♢

Conditions (i) and (ii) ensures that r is active and, as we require support to be "external" of X, r must also be active when removing the atoms in X from the interpretation. In case I is a model, Items (iii) and (iv) jointly ensure that there is some C-atom A in the head of r that is satisfied by I and derives some atom of X. Example 25 below further illustrates the notion of external support.

We next show how answer sets can be characterised in terms of external supports.

Theorem 2. *Let P be a* C*-program and I an interpretation. Then, I is an answer set of P iff I is a model of P and for every X with $\emptyset \subset X \subseteq I$, there is some $r \in P$ such that r is an external support for X with respect to I.*

Proof. ($\Rightarrow$) We show the claim by contraposition. In case I is not a model of P we are done as then I is not an answer set of P. Assume $I \models P$ but there is some X with $\emptyset \subset X \subseteq I$ such that there is no $r \in P$ which is an external support for X with respect to I. Consider the interpretation $I' = I \setminus X$. Clearly, $I' \subset I$. It suffices to show that P, I, and I' satisfy Condition ($\star$) of Definition 34 as then $I \notin AS(P)$. Consider some $r' \in P^I$ such that $I' \models \mathrm{B}(r')$. As $I \models \mathrm{B}(r')$ and $I' \models \mathrm{B}(r')$, it follows that $I \setminus X \models \mathrm{B}(r')$. Moreover, from $I \models \mathrm{B}(r')$, $r' \in P$,

and $I \models P$, we also have that $I \models A$, for some $A \in \mathrm{H}(r')$. In the case that $X|_{D_A} = \emptyset$ we are done as then $I'|_{D_A} = I|_{D_A}$ by choice of I'. Assume that $X|_{D_A} \neq \emptyset$. Note that r' satisfies Condition (iii) of Definition 35 because $I|_{D_A} \in C_A$ follows from $I \models A$. As r' is not an external support for X with respect to I but satisfies Conditions (i), (ii), and (iii) for being one, Condition (iv) must not hold, i.e., there is some $A' \in \mathrm{H}(r')$ with $I \models A'$ and $X|_{D_{A'}} = (I \setminus I')|_{D_{A'}} = \emptyset$. From that we get $I'|_{D_{A'}} = I|_{D_{A'}}$ and hence also $I' \models A'$, i.e., P, I, and I' satisfy ($\star$).

($\Leftarrow$) Suppose $I \notin AS(P)$. We show that the right-hand side does not hold either. In case $I \not\models P$ we are done. Consider the case that $I \models P$ but there is some $I' \subset I$ such that P, I, and I' satisfy Condition ($\star$) of Definition 34. Consider $X = I \setminus I'$ and some $r \in P$. It remains to show that r is not an external support for X with respect to I. Assume that Conditions (i), (ii), and (iii) of Definition 35 hold for I, X, and r. From $I \models \mathrm{B}(r)$ we get $r \in P^I$. As $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$. Therefore, we have that $I \models A$ as well and $X|_{D_A} = \emptyset$. Since then $X \cap I|_{D_A} = \emptyset$, r does not satisfy Condition (iv) for being an external support for X with respect to I. $\square$

4.5.2 Unfounded Sets

To complete the picture, we express the absence of an external support in an interpretation by extending the concept of an *unfounded set* (Leone et al., 1997; Faber, 2005) to abstract-constraint programs (for the case of total interpretations). Defining unfounded sets in terms of external supports is motivated by the duality of these notions as discussed by Lee (Lee, 2005).

Definition 36. Let P be a C-program, X a set of atoms, and I an interpretation. Then, X is *unfounded in P with respect to I* if there is no C-rule $r \in P$ that is an external support for X with respect to I. $\Diamond$

Note that this is a conservative extension of Definition 32 (on page 41) for elementary-head C-programs.

Theorem 2 now immediately yields the following result:

Corollary 1. *Let P be a* C*-program and I an interpretation. Then, I is an answer set of P iff I is a model of P and there is no set X with $\emptyset \subset X \subseteq I$ that is unfounded in P with respect to I.*

Example 25. Consider C-program P_{Ex25}, consisting of the C-rules

$$\begin{aligned} r_1 :& \quad c \leftarrow not\ a, \\ r_2 :& \quad \langle \{a,b\}, \{\{a,b\}\} \rangle \leftarrow a, \qquad \text{and} \\ r_3 :& \quad b \vee \langle \{a,c\}, \{\{a,c\}\} \rangle \leftarrow , \end{aligned}$$

and model $I_1 = \{a,b\}$ of P_{Ex25}. The set $X_1 = \{a\}$ is unfounded in P_{Ex25} with respect to I_1 in view of the following facts:

- r_1 is no external support for X_1 with respect to I_1 because $I_1 \not\models \mathrm{B}(r_1)$ (but also because there is no $A \in \mathrm{H}(r_1)$ with $X_1|_{D_A} \neq \emptyset$ and $I_1|_{D_A} \subseteq S$, for some $S \in C_A$),
- r_2 is no external support for X_1 with respect to I_1 because $I_1 \setminus X_1 \not\models \mathrm{B}(r_2)$, and
- r_3 is no external support for X_1 with respect to I_1 because, for $b \in \mathrm{H}(r)$, we have $I_1 \models b$ and $(X_1 \cap I_1)|_{D_b} = \emptyset$.

In fact, X_1 is the only set X with $\emptyset \subset X \subseteq I_1$ that is unfounded in P_{Ex25} with respect to I_1 because r_3 is an external support for both $\{b\}$ and $\{a,b\}$ with respect to I_1. Nevertheless, as X_1 is unfounded in P_{Ex25} with respect to I_1, by Corollary 1, I_1 is not an answer set of P_{Ex25}.

Now let us consider $I_2 = \{b,c\}$. Then, the following holds:

- r_1 and r_3 are external supports for $\{b,c\}$ with respect to I_2,

- r_3 is an external support for $\{b\}$ with respect to I_2, and
- r_1 is an external support for $\{c\}$ with respect to I_2.

As I_2 is a model of P_{Ex25} and for every X with $\emptyset \subset X \subseteq I_2$ there is some external support with respect to I_2 in P_{Ex25}, I_2 is an answer set of P_{Ex25} by Theorem 2. ■

4.5.3 Unfounded-Free Interpretations

Faber (2005) also provides a characterisation of answer sets based on the *unfounded-freeness* property for the class of programs he considered. This concept can be lifted to the case of abstract-constraint programs under our semantics.

Definition 37. Let P be a C-program and I an interpretation. Then, I is *unfounded-free* in P if $I \cap X = \emptyset$ for each unfounded set X in P with respect to I. ◊

Opposed to Theorems 2 and Corollary 1, the definition of unfounded-freeness does not restrict the considered unfounded sets to subsets of the interpretation. Therefore, it is important to note that, due to the definition of external support, the part of an unfounded set contained in the interpretation is itself an unfounded set.

Theorem 3. *Let X be a set of atoms and I an interpretation. If a* C*-rule r is an external support for $I \cap X$ with respect to I, then r is an external support for X with respect to I.*

Proof. Assume r is an external support for $I \cap X$. Then, $I \models \mathrm{B}(r)$ and, for all $A \in \mathrm{H}(r)$ with $I \models A$, it holds that $(X \cap I)|_{D_A} \neq \emptyset$. Moreover, since $I \setminus (I \cap X) \models \mathrm{B}(r)$, also $I \setminus X \models \mathrm{B}(r)$. Furthermore, there is some $A' \in \mathrm{H}(r)$ with $I \cap X|_{D_{A'}} \neq \emptyset$ and $I|_{D_{A'}} \subseteq S$, for some $S \in C_{A'}$. From $I \cap X|_{D_{A'}} \neq \emptyset$ we get $X|_{D_{A'}} \neq \emptyset$. Consequently, r is an external support for X with respect to I. □

From Theorem 3 and the definition of unfounded sets, we immediately obtain the following corollary:

Corollary 2. *Let X be a set of atoms, P a* C*-program, and I an interpretation. If X is unfounded in P with respect to I, then $I \cap X$ is unfounded in P with respect to I.*

We conclude the section with the result that characterises answer sets in terms of unfounded-free models, generalising Corollary 3 of Faber (2005).

Theorem 4. *Let P be a* C*-program and I an interpretation. Then, I is an answer set of P iff I is a model of P and unfounded-free in P.*

Proof. ($\Rightarrow$) Suppose that $I \in AS(P)$. By Corollary 1, I is a model of P and it holds that (*) there is no set X with $\emptyset \subset X \subseteq I$ that is unfounded in P with respect to I. Assume that I is not unfounded-free in P. Then, there is some unfounded set X for P with respect to I such that $I \cap X \neq \emptyset$. Hence, by Corollary 2, $I \cap X$ is an unfounded set in P with respect to I, contradicting (*).
($\Leftarrow$) Assume that I is not an answer set of P. By Corollary 1, there must be some set X with $\emptyset \subset X \subseteq I$ that is unfounded in P with respect to I. Hence, as thus $I \cap X \neq \emptyset$, I is not unfounded-free in P. □

4.6 Complexity

Next, we present some basic complexity results regarding our notion of answer sets.

Theorem 5. *Deciding whether a* C*-program* P *has an answer set is* Σ_2^{P}*-hard.*

Proof. Note that for showing Σ_2^{P}-hardness for general C-programs it suffices to show that the problem is already Σ_2^{P}-hard for the restricted setting of elementary C-programs. The latter follows from the equivalence of our semantics and the FLP-semantics for elementary-head programs (shown in Theorem 8 on page 48), the equivalence of the FLP-semantics and the answer-set semantics by Gelfond and Lifschitz (Gelfond and Lifschitz, 1991) for elementary C-programs, as shown by Faber et al. (2011), and the Σ_2^{P}-hardness of answer-set existence for this language fragment (Eiter and Gottlob, 1995). □

Note that due to a result by Faber (2005), hardness already holds for normal C-programs without default negation and only elementary C-atoms in rule heads.

The next result establishes Σ_2^{P}-membership which also holds if the size of a C-atom is determined by the size of its domain only. The latter is important to note, since we see C-atoms only as a theoretical device representing other types of literals, as discussed earlier, whose representations are typically more compact than C-atoms.

Theorem 6. *Provided that* $I \models A$ *can be decided in time polynomial in* $|I| + |D_A|$ *for any interpretation* I *and every* C*-atom* A*, deciding whether a* C*-program* P *has an answer set is in* Σ_2^{P}*, even if the size of every* C*-atom* A *is determined by the size of* D_A *only.*

Proof. We can guess an interpretation I from the set of atoms appearing in P in polynomial time. For every C-rule $r \in P$, we check whether $I \models \mathrm{B}(r)$ implies $I \models^{\exists} \mathrm{H}(r)$ in polynomial time. It remains to show that we can check whether there is an interpretation $I' \subset I$ such that P, I, and I' obey ($\star$) using an NP-oracle. But this can be easily realised by first guessing some $I' \subset I$ and then checking ($\star$) for P, I, and I' in polynomial time. When the NP-check returns false, we give a positive answer, indicating that P has an answer set. □

4.7 Comparison of our Semantics for Abstract-Constraint Programs to others

In this section, we shed some light on commonalities and differences of the semantics for C-programs introduced in Chapter 4 with related proposals. The relation to other FLP-semantics and related approaches is discussed in Section 4.7.1. At first, it is shown that our semantics for general C-programs properly extends the FLP-semantics for elementary-head C-programs (corresponding to the language for which the FLP-semantics was introduced) and how answer sets of the straightforward extension of the FLP-semantics, the eFLP-semantics discussed in Section 4.4.1, relate to our notion of answer sets. Then, we analyse the semantics by Marek and Remmel (2012) for (restricted) C-programs and show that it amounts to the eFLP-semantics. We conclude the section with showing correspondence to an FLP-style semantics for propositional theories (Truszczyński, 2010) and discuss its implications on the relation to further semantics.

In Section 4.7.2, we compare our semantics with others that follow the tradition of Simons et al. (2002). At first, we discuss their semantic differences to FLP-like approaches. Then, we establish results that relate our notion of answer sets with stable models in the approach by Shen et al. (2009) and discuss relations to further works based on these results. Based on the relation to other formalisms, we justify in Section 4.7.3 why the semantics we chose for our framework is suitable for compatibility with current ASP solvers.

4.7.1 Semantics in the Style of Faber, Pfeifer, and Leone

The extended FLP-semantics for abstract-constraint programs, as discussed in Section 4.4.1, and our proposed semantics are interrelated as follows.

Theorem 7. *For any* C*-program P, each eFLP-answer set of P is an answer set of P.*

Proof. This is a direct consequence of the respective definitions. □

That the converse direction does not hold has already been shown in Example 23 on page 42. As intended, for the restricted setting of elementary-head programs that was considered by Faber et al. (2011), our semantics coincides with theirs (and the eFLP-semantics).

Theorem 8. *For an elementary-head* C*-program P, $AS(P) = AS_{FLP}(P)$.*

Proof. $AS_{FLP}(P) \subseteq AS(P)$ holds by Theorem 7. Assume now that $I \in AS(P)$ but $I \notin AS_{FLP}(P)$. From $I \in AS(P)$ it follows that $I \models P^I$. Hence, by Definition 31 (on page 41), there must be some $I' \subset I$ such that $I' \models P^I$. Furthermore, by Definition 34 (on page 41), there must be some $r \in P^I$ such that $I' \models \mathrm{B}(r)$ and

(†) for all $a \in \mathrm{H}(r)$ with $I' \models a$, $I'|_{D_a} \neq I|_{D_a}$ holds.

Since $r \in P^I$, we have $I' \models \mathrm{B}(r)$, and since $I' \models P^I$, it follows in turn that $I' \models^{\exists} \mathrm{H}(r)$. Thus, there is some $a' \in \mathrm{H}(r)$ with $I' \models a'$. Consequently, $I'|_{D_{a'}} = \{a'\}$. As $I' \subset I$ and $D_{a'} = \{a'\}$, we get $I|_{D_{a'}} = \{a'\}$, and hence $I'|_{D_{a'}} = I|_{D_{a'}}$. As this contradicts (†), $AS(P) = AS_{FLP}(P)$ must hold. □

Marek and Remmel (2012) defined a semantics for disjunctive positive abstract-constraint programs where C-atoms are required to be monotone if they occur in the head and monotone or anti-monotone if they appear in a rule body. The authors use a one-step provability operator for defining so-called *selector stable models* which are relative to a given *selector function*. Intuitively, this function determines for each rule statically which propositional atoms from the domains of the C-atoms in the head have to be true. Based on this notion, Marek and Remmel also define *minimal selector stable models* that are independent of any selector function and showed that they coincide with the following reduct-based semantics.

Definition 38 (Marek and Remmel, 2012). Let P be a positive C-program such that, for all $r \in P$, C-atoms in $\mathrm{H}(r)$ are monotone, C-atoms in $\mathrm{B}^+(r)$ are monotone or anti-monotone, and $\mathrm{B}^-(r) = \emptyset$. Furthermore, let I be an interpretation. The *MR-reduct*, P^I_{MR}, of P is obtained by

1. removing all $r \in P$ such that $I \not\models A$, for some anti-monotone C-atom $A \in \mathrm{B}^+(r)$, and
2. removing all anti-monotone C-atoms from the remaining rules in P.

An interpretation I is an *MR-stable model* of P if it is a minimal model of P^I_{MR}. ◇

It turns out that this semantics is equivalent to the straightforwardly extended FLP-semantics and is thus not suitable for expressing choice rules as discussed in Section 4.4.1.

Theorem 9. *Let P and I be as in Definition 38. An interpretation I is an MR-stable model of P iff it is an eFLP-answer set of P.*

Proof. (⇒) Assume that I is not an eFLP-answer set of P. Consider the case that I is not a model of P^I. There is some rule $r \in P$ with $I \models \mathrm{B}(r)$ and $I \not\models^{\exists} \mathrm{H}(r)$. The rule is not removed in the first step of building the MR-reduct of P. Let $r' \in P^I_{\mathrm{MR}}$ be the rule obtained from r in the second step. As $I \not\models r'$, we have that I is not an MR-stable model of P.

Now consider the case that there is some $I' \subset I$ that is a model of P^I and let $r \in P^I_{\mathrm{MR}}$ be a rule with $I' \models \mathrm{B}(r)$. We will show that $I' \models^{\exists} \mathrm{H}(r)$. By definition of P^I_{MR}, there must be some $r' \in P$ with $\mathrm{B}(r) \subseteq \mathrm{B}(r')$ and $\mathrm{H}(r) = \mathrm{H}(r')$. Since $\mathrm{B}(r)$ consists only of monotone C-atoms, it follows from $I' \models \mathrm{B}(r)$ that $I \models \mathrm{B}(r)$. It must hold that, for all $A \in \mathrm{B}(r') \setminus \mathrm{B}(r)$, A is an anti-monotone C-atom such that $I \models A$. Consequently, $I' \models \mathrm{B}(r')$ and $I \models \mathrm{B}(r')$. Hence, $r' \in P^I$ and as $I' \models P^I$ we have $I' \models^{\exists} \mathrm{H}(r)$. We showed that $I' \models P^I_{\mathrm{MR}}$, and consequently I is not an MR-stable model of P.
($\Leftarrow$) Assume that I is not an MR-stable model of P. Consider the case that I is not a model of P^I_{MR}. Then, there is some $r \in P^I_{\mathrm{MR}}$ such that $I \models \mathrm{B}(r)$ but $I \not\models^{\exists} \mathrm{H}(r)$. By Definition 38, there must be some $r' \in P$ with $\mathrm{H}(r) = \mathrm{H}(r')$, $\mathrm{B}(r) \subseteq \mathrm{B}(r')$, and $I \models A$, for every $A \in \mathrm{B}(r') \setminus \mathrm{B}(r)$. As $I \models \mathrm{B}(r)$, also $I \models \mathrm{B}(r')$ holds. But then $I \not\models P$, implying that I is not an eFLP-answer set of P. Now consider the case that there is some $I' \subset I$ that is a model of P^I_{MR} and consider some rule $r \in P^I$ with $I' \models \mathrm{B}(r)$. As $I \models \mathrm{B}(r)$, there is some $r' \in P^I_{\mathrm{MR}}$ with $\mathrm{B}(r') \subseteq \mathrm{B}(r)$ and $\mathrm{H}(r') = \mathrm{H}(r)$. By monotonicity of the C-atoms in $\mathrm{B}(r')$, we get from $I' \models \mathrm{B}(r)$ that $I' \models \mathrm{B}(r')$. As $I' \models P^I_{\mathrm{MR}}$, it must hold that $I' \models^{\exists} \mathrm{H}(r')$, and consequently $I' \models^{\exists} \mathrm{H}(r)$. We showed that $I' \models P^I$, and consequently I is not an eFLP-answer set of P. □

Another approach relevant for our discussion is the work by Truszczyński (2010) who introduced an FLP-style semantics for propositional theories. For comparing his work with ours, we consider propositional theories over atoms from $\mathrm{B}^{\mathcal{P}}_{\mathrm{HU}_{\mathscr{A}}}$ and the Boolean connectives $\bot$, $\wedge$, $\vee$, and $\rightarrow$. Moreover, we use the shorthands $\top = \bot \rightarrow \bot$ and $\neg f = f \rightarrow \bot$, for any formula f. Given an interpretation I and a formula f, the relation $I \models f$, specifying when f is true under I, is defined as usual. Also, following custom, we identify empty disjunctions with $\bot$ and empty conjunctions with $\top$.

Definition 39 (Truszczyński, 2010). Let f be a propositional formula and I an interpretation. The *T-reduct*, f^I, of f is defined inductively as follows:

1. if $f = \bot$, then $f^I = \bot$;
2. if $f = a$, where a is an atom, then $f^I = a$ if $I \models a$, and $f^I = \bot$ otherwise;
3. if $f = g \circ h$, where g and h are propositional formulas and $\circ \in \{\wedge, \vee\}$, then $f^I = g^I \circ h^I$ if $I \models g \circ h$, and $f^I = \bot$ otherwise;
4. if $f = g \rightarrow h$, where g and h are propositional formulas, then
   a) $f^I = g \rightarrow h^I$ if $I \models g$ and $I \models h$,
   b) $f^I = \top$ if $I \not\models g$, and
   c) $f^I = \bot$ otherwise.

For a propositional theory F, F^I is defined as $\{f^I \mid f \in F\}$. ◊

Definition 40 (Truszczyński, 2010). Let F be a propositional theory and I an interpretation. Then, I is a *T-stable model* of F iff I is a subset-minimal model of F^I. ◊

Note that any T-stable model of F is also a model of F. In order to compare T-stable models with our semantics, we use a standard translation of abstract-constraint programs to propositional theories. To this end, we use the following representation of abstract-constraint atoms in terms of DNF formulas.

Definition 41 (Shen et al., 2009). Let $A = \langle D, C \rangle$ be a C-atom. Then,

$$\varphi(A) = \bigvee_{X \in C} \left(\left(\bigwedge_{a \in X} a \right) \wedge \left(\bigwedge_{a \in D \setminus X} \neg a \right) \right). \quad ◊$$

We extend the translation $\varphi(\cdot)$ to rules and abstract-constraint programs as follows.

Definition 42. Let r be a C-rule of form (4.1). Then, $\varphi(r) = \varphi_B(r) \rightarrow \varphi_H(r)$, where

$$\varphi_H(r) = \varphi(A_1) \vee \cdots \vee \varphi(A_k) \text{ and}$$
$$\varphi_B(r) = \varphi(A_{k+1}) \wedge \cdots \wedge \varphi(A_m) \wedge \neg\varphi(A_{m+1}) \wedge \cdots \wedge \neg\varphi(A_n).$$

Finally, for a C-program P, we define the propositional theory

$$\varphi(P) = \{\varphi(r) \mid r \in P\}. \qquad \Diamond$$

Obviously, for a rule r and an interpretation I, $I \not\models^{\exists} \mathrm{H}(r)$ iff $I \models \varphi_H(r)$, and $I \models \mathrm{B}(r)$ iff $I \models \varphi_B(r)$. Moreover, for showing the relation between the semantics by Truszczyński and ours, we make use of the following properties.

Lemma 1 (Truszczyński, 2010, Proposition 1). *For any formula f and any set I of atoms, $I \models f$ iff $I \models f^I$. Moreover, for any theory F and any set I of atoms, $I \models F$ iff $I \models F^I$.*

Next, we show that our semantics resembles that of Truszczyński on the level of abstract-constraint programs.

Theorem 10. *Consider a* C*-program P and let I be an interpretation. Then, I is an answer set of P iff I is a T-stable model of $\varphi(P)$.*

Proof. ($\Rightarrow$) Suppose I is not a T-stable model of $\varphi(P)$. We will show that I is not an answer set of P. Consider the case that I is not a model of $\varphi(P)^I$. Then, there is some formula $f \in \varphi(P)^I$ such that $I \not\models f$. By Definition 42, there is some rule $r \in P$ with $f = \varphi(r)^I$. By Lemma 1, $I \models \varphi_B(r)$ and $I \not\models \varphi_H(r)$ and consequently $I \not\models r$. It follows that I is not an answer set of P. Now, consider the case that $I \models \varphi(P)^I$ but there is some $I' \subset I$ such that $I' \models \varphi(P)^I$. Consider some $r \in P^I$ such that $I' \models \mathrm{B}(r)$. As $I \models \mathrm{B}(r)$ it holds that $I \models \varphi_B(r)$. From $I \models \varphi(P)^I$ we get by Lemma 1 that $I \models \varphi(P)$. Consequently, since $\varphi(r) \in \varphi(P)$, from $I \models \varphi_B(r)$ follows $I \models \varphi_H(r)$. By Definition 39, $\varphi(r)^I = \varphi_B(r) \rightarrow \varphi_H(r)^I$. As $I' \models \varphi(P)^I$ and $\varphi(r)^I \in \varphi(P)^I$ we have $I' \models \varphi_B(r) \rightarrow \varphi_H(r)^I$. From $I' \models \mathrm{B}(r)$, we get $I' \models \varphi_B(r)$ and hence by modus ponens $I' \models \varphi_H(r)^I$. As $\varphi_H(r)^I$ is a disjunction, there must be at least one disjunct $\varphi(A)^I$ with $I' \models \varphi(A)^I$ and $A \in \mathrm{H}(r)$. As then $\varphi(A)^I \neq \bot$, we get by Definition 39 that $I \models \varphi(A)$. Therefore, also $I \models A$. By construction, $\varphi(A)$ is a disjunction with

$$s = (\bigwedge_{a \in I|_{D_A}} a) \wedge (\bigwedge_{a \in D_A \setminus I|_{D_A}} \neg a)$$

being the single disjunct that is satisfied by I. Hence, $\varphi(A)^I$ is a disjunction whose only disjunct different from $\bot$ is

$$s^I = (\bigwedge_{a \in I|_{D_A}} a) \wedge (\bigwedge_{a \in D_A \setminus I|_{D_A}} \top).$$

As $I' \models \varphi(A)^I$ it must hold that $I|_{D_A} \subseteq I'$. Therefore, it holds that $I|_{D_A} = I'|_{D_A}$ and, since $I \models A$, also $I' \models A$. By Definition 34 (on page 41), I is not an answer set of P.
($\Leftarrow$) Assume now that $I \notin AS(P)$. We will show that I is not a T-stable model of P. Consider the case that $I \not\models P^I$. Then, there is a rule $r \in P^I$ with $I \models \mathrm{B}(r)$ and $I \not\models^{\exists} \mathrm{H}(r)$. Hence, we get that $I \models \varphi_B(r)$ and $I \not\models \varphi_H(r)$. As $\varphi(r) \in \varphi(P)$, $I \not\models \varphi(P)$. By Lemma 1, $I \not\models \varphi(P)^I$. Hence, I is not a T-stable model of P. Consider the remaining case that $I \models P^I$ but there is some $I' \subset I$ with $I' \models P^I$ and for every $r \in P^I$ such that $I' \models \mathrm{B}(r)$, there is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$. Consider some formula $f \in \varphi(P)^I$. It remains to show that $I' \models f$. It must hold that $f = \varphi(r)^I$, for some $r \in P$. Hence, f is of the form $(\varphi_B(r) \rightarrow \varphi_H(r))^I$. By Definition 39, we can distinguish three cases. Note that it cannot hold

that $f = \bot$ as then $I \models \varphi_B(r)$ and $I \not\models \varphi_H(r)$ which would imply that $r \in P^I$ and $I \not\models r$, a contradiction to $I \models P^I$. In case $f = \top$ we are done as then $I' \models f$.

In the third case, we have $I \models \varphi_B(r)$, $I \models \varphi_H(r)$, and $f = \varphi_B(r) \to \varphi_H(r)^I$. In case $I' \not\models \varphi_B(r)$, we have reached our goal $I' \models f$. Assume $I' \models \varphi_B(r)$. Then, $I' \models \mathrm{B}(r)$. There is some $A \in \mathrm{H}(r)$ with $I' \models A$ and $I'|_{D_A} = I|_{D_A}$. Note that then also $I \models A$. The formula $\varphi(A)$ is a disjunct in $\varphi_H(r)$ with $I \models \varphi(A)$. It follows that $\varphi(A)^I$ is a disjunct in $\varphi_H(r)^I$. $\varphi(A)$ is a disjunction where, by construction, only the disjunct

$$s = (\bigwedge_{a \in I|_{D_A}} a) \wedge (\bigwedge_{a \in D_A \setminus I|_{D_A}} \neg a)$$

is satisfied by I. From that we know that

$$s^I = (\bigwedge_{a \in I|_{D_A}} a) \wedge (\bigwedge_{a \in D_A \setminus I|_{D_A}} \top)$$

is a disjunct in $\varphi(A)^I$. As $I' \models s^I$, we have $I' \models \varphi(A)^I$ and therefore $I' \models \varphi_H(r)^I$. We conclude that $I' \models f$. $\square$

As Bartholomew et al. (2011) showed that their semantics for first-order theories with aggregates extends that of Truszczyński, the same relation applies to our approach as well.

A main goal of the paper by Truszczyński is to study the differences between the semantics of Faber et al. and that of Ferraris (2011). It is worth mentioning that the same differences to the latter semantics hold for our semantics also. In particular, they differ in the treatment of default negated C-atoms. As an example, consider the C-program

$$P = \{a \leftarrow not\langle\{a\}, \{\emptyset\}\rangle\}$$

which has only $\emptyset$ as answer set under FLP-semantics and our semantics. Under Ferraris' semantics, $\varphi(P)$ has two stable models, $\emptyset$ and $\{a\}$. For further information on the relation between these families of semantics and on their intuitions, we refer to the work of Faber et al. (2011), Truszczyński (2010), and Lee and Meng (2009), who reduce AG-programs under the FLP-semantics to propositional formulas under the semantics of Ferraris.

4.7.2 Semantics in the Tradition of Simons et al.

Next, we discuss relations to semantics that follow the tradition of Simons et al. (2002) that laid the foundation for the `Smodels` solver.

A characteristic difference of these and FLP-semantics is the way how non-convex body literals may give support to atoms in an interpretation.

Consider the C-program consisting of the following C-rules:

$$\begin{aligned} &a \leftarrow \langle\{a, b\}, \{\emptyset, \{a, b\}\}\rangle, \\ &a \leftarrow b, \text{ and} \\ &b \leftarrow a. \end{aligned}$$

While $\{a, b\}$ is an answer set under FLP-style semantics, it is not considered stable in, e.g., the semantics discussed in the following.

Shen et al. (2009) defined a stable-model semantics for abstract-constraint programs involving disjunction—i.e., the language fragment they consider is the same as our setting. For showing the relation to our semantics, we need to introduce a range of auxiliary definitions and lemmas taken from their paper. Furthermore, note that Shen et al. distinguish propositional atoms and C-atoms, while we are treating the former as elementary C-atoms. However, also in their approach, atoms can be safely replaced by respective C-atoms. Moreover, they treat a

default negated C-atom $not\langle D, C\rangle$ as its complement $\langle D, 2^D \setminus C\rangle$. Thus, we assume that in any C-program only elementary C-atoms may be default negated in the remainder. Note that we can make this restriction without loss of generality, as a negated C-atom is satisfied by an interpretation iff the complement of the C-atom is. Thus, we can replace all negated c-atoms that are not elementary in a general C-program without changing the semantics of the program.

Next, we introduce required concepts that stem from Shen et al. (2009). Their work relies on a representation of sets of atoms in terms of power sets for which they use so-called *abstract prefixed power sets*.

Definition 43 (Shen et al., 2009, Definition 3.1). Let W and V be two sets of atoms. The *W-prefixed power set* of V, denoted by $W \uplus V$, is the collection

$$\{W \cup V' \mid V' \in 2^V\}.$$

For any set S of atoms, S is *covered* by $W \uplus V$ if $W \subseteq S$ and $S \subseteq W \cup V$. For any two abstract prefixed power sets $W \uplus V$ and $W' \uplus V'$, $W \uplus V$ is *included* in $W' \uplus V'$ if any set covered by $W \uplus V$ is covered by $W' \uplus V'$. ◊

We will need the auxiliary notions of *W-maximality* and *redundancy* of an abstract prefixed power set.

Definition 44 (Shen et al., 2009). Consider $W \in C_A$ and $V \subseteq D_A \setminus W$, for some C-atom A. $W \uplus V$ is *W-maximal* in A if all sets covered by $W \uplus V$ are in C_A and there is no V' with $V \subset V' \subseteq D_A \setminus W$ such that all sets covered by $W \uplus V'$ are in C_A.

Furthermore, in a collection containing two abstract prefixed power sets, $W \uplus V$ and $W' \uplus V'$, $W \uplus V$ is *redundant* in this collection if $W' \uplus V'$ includes $W \uplus V$. ◊

Shen et al. associate C-atoms with sets of abstract prefixed power sets as follows:

Definition 45 (Shen et al., 2009, part of Definition 3.2). Let A be a C-atom and $W \in C_A$. Then,

$$\{W \uplus V | W \uplus V \text{ is } W\text{-maximal in } A\}$$

is the *collection of abstract W-prefixed power sets of A*. ◊

We now have the means to define the representation of C-atoms as used by Shen et al.

Definition 46 (Shen et al., 2009, Definition 3.3). Let A be a C-atom. The *abstract representation* of A is a pair $\langle D_A, C^*_A\rangle$, where $C^*_A = \bigcup_{W \in C_A} C(W)$ such that $C(W)$ is the collection of abstract W-prefixed power sets of A, with all redundant abstract prefixed power sets removed. ◊

We also need the concepts of *abstract satisfiable sets* and *satisfiable sets* of a C-atom.

Definition 47 (Shen et al., 2009, Definition 4.1). Let A be a C-atom and I an interpretation. Then, $W \uplus V \in C^*_A$ is an *abstract satisfiable set of A with respect to I* if $W \uplus V$ covers $I|_{D_A}$. In this case, W is called a *satisfiable set of A with respect to $I|_{D_A}$*. ◊

Next, we give the definition of the transformation by Shen et al. that extends the Gelfond-Lifschitz reduct. The transformation introduces fresh atoms θ_A and β_A for every C-atom A. The set of these atoms is denoted by Γ.

Definition 48 (Shen et al., 2009, Definition 5.1). Given an abstract-constraint program P and an interpretation I, the *SYY-reduct of P with respect to I*, symbolically P^I_{SYY}, is obtained from P by performing the following operations:

1. Remove from P all C-rules whose bodies contain either a negated atom $not\ a$ such that $a \in I$ or a C-atom A such that $I \not\models A$.

2. Remove from the remaining rules all negative atoms.

3. Replace each C-atom A in the body of a C-rule with a fresh atom θ_A and introduce a new C-rule $\theta_A \leftarrow a_1, \ldots, a_m$ for each satisfiable set $\{a_1, \ldots, a_m\}$ of A with respect to $I|_{D_A}$.

4. Remove each C-atom A in the head of a C-rule if $I \not\models A$, and replace it with a fresh atom β_A and introduce a new C-rule $b \leftarrow \beta_A$, for each $b \in I|_{D_A}$, a new C-rule $\leftarrow c, \beta_A$, for each $c \in D_A \setminus I|_{D_A}$, and a new C-rule $\beta_A \leftarrow a_1, \ldots, a_m$, where $\{a_1, \ldots, a_m\} = I|_{D_A}$, if $I \models A$. ♢

Finally, we give the definition of stable models following Shen et al.

Definition 49 (Shen et al., 2009, Definition 5.2). Let P be a program and I an interpretation. Then, I is an *SYY-stable model of* P if $I = I' \setminus \Gamma$, where I' is a minimal model of P^I_{SYY}. ♢

Besides the definitions given above, we will reuse a couple of results of Shen et al. as lemmas for our upcoming proofs.

Lemma 2 (Shen et al., 2009, Theorem 4.5). *Let A be a* C*-atom and I an interpretation. Then, $I \models A$ iff, for some abstract satisfiable set $W \uplus V$ of A with respect to I, $I \models W$ and $I \models \{not\ a \mid a \in D_A \setminus (W \cup V)\}$.*

Lemma 3 (Shen et al., 2009, Theorem 4.6). *Let A be a* C*-atom and I an interpretation. If S is a satisfiable set of A with respect to $I|_{D_A}$, then, for every S' with $S \subseteq S' \subseteq I|_{D_A}$, $S' \in C_A$.*

Lemma 4 (Shen et al., 2009, Observation following Definition 3.3). *Let W and V be sets of atoms, and let A be a* C*-atom. If $\{W \cup \Delta \mid \Delta \in 2^V\} \subseteq C_A$, then there exist $W', V' \subseteq C_A$ such that $W' \subseteq W$, $W \cup V \subseteq W' \cup V'$, and $W' \uplus V' \in C^*_A$.*

Lemma 5 (Shen et al., 2009, Theorem 5.1). *Any SYY-stable model of an abstract-constraint program P is a model of P.*

Eventually, we can state the first theorem relating our semantics to SYY-stable models.

Theorem 11. *Let P be a* C*-program such that all* C*-literals appearing in a body of a rule in P are convex. If I is an answer set of P, then I is an SYY-stable model of P.*

Proof. Assume $I \in AS(P)$ and suppose that I is not an SYY-stable model of P. That is, there is no minimal model M' of P^I_{SYY} with $I = M' \setminus \Gamma$. Consider the interpretation

$$M = I \cup \{\theta_A \mid r \in P^I_{\mathrm{SYY}} \text{ such that } r = \theta_A \leftarrow l_1, \ldots, l_n \text{ and } I \models \mathrm{B}(r)\} \cup \{\beta_A \mid r \in P^I_{\mathrm{SYY}} \text{ such that } \beta_A \in \mathrm{H}(r) \text{ and } I \models A\}.$$

We derive a contradiction by showing that M is a minimal model of P^I_{SYY} with $I = M \setminus \Gamma$. Obviously, $I = M \setminus \Gamma$ holds. Consider the case that M is no model of P^I_{SYY}. Then, there is a C-rule $r \in P^I_{\mathrm{SYY}}$ with $M \not\models r$, i.e., $M \models \mathrm{B}(r)$ and $M \not\models A$, for all $A \in \mathrm{H}(r)$. We now consecutively consider r to be of one the different rule forms allowed in P^I_{SYY}. Consider the case that r is of the form

$$\beta_{A_1} \vee \cdots \vee \beta_{A_k} \leftarrow \theta_{B_1}, \ldots, \theta_{B_n}.$$

Then, by definition of P^I_{SYY}, there is some C-rule

$$r' = A_1 \vee \cdots \vee A_k \vee A_{k+1} \vee \cdots \vee A_m \leftarrow B_1, \ldots, B_n$$

in P from which r was constructed. Assume $I \not\models \mathrm{B}(r')$. Then, there is some $B_j \in \mathrm{B}(r')$ such that $I \not\models B_j$. Since $M \models \mathrm{B}(r)$, it follows that $M \models \theta_{B_j}$. From that, by Lemma 2 and Definition 47, there is some satisfiable set S of B_j with respect to $I|_{D_{B_j}}$ such that $I \models S$. By definition of P^I_{SYY}, there is some $r_2 \in P^I_{\mathrm{SYY}}$ where $\mathrm{H}(r_2) = \{\theta_{B_j}\}$ and $\mathrm{B}(r_2) = S$. By definition of a satisfiable set, we get that $S \uplus V$ covers $I|_{D_{B_j}}$, which implies in turn $S \subseteq I|_{D_{B_j}}$. From that and Lemma 3, it follows that $I|_{D_{B_j}} \in C_{B_j}$. We have a contradiction since then $I \models B_j$.

Next, consider the case that $I \models \mathrm{B}(r')$. Then, as I is an answer set of P, we also have $I \models A_h$, for some $h \in \mathrm{H}(r')$. By definition of P^I_{SYY}, there is some $r_3 \in P^I_{\mathrm{SYY}}$, where $\mathrm{H}(r_3) = \{\beta_{A_h}\}$ and $\mathrm{B}(r_3) = I|_{D_{A_h}}$. From $I \models A_h$ and $r_3 \in P^I_{\mathrm{SYY}}$, by construction of M, we get $\beta_{A_h} \in M$. As it must hold that $\beta_{A_h} \in \mathrm{H}(r)$, we arrive at a contradiction as then $M \models r$. The case that r is of the form $\theta_A \leftarrow l_1, \ldots, l_n$ is impossible by definition of M.

Now consider the case that r is of the form $l \leftarrow \beta_A$, where $l \in I|_{D_A}$. As then $l \in I$, and thus $l \in M$, we obtain $M \models l$. This is a contradiction to $M \not\models r$.

Consider that r is of the form $\leftarrow l, \beta_A$, where $l \in D_A \setminus I|_{D_A}$. We get $l \notin I$ and consequently $I \not\models l$. Hence, also $M \not\models l$, and therefore we arrive at a contradiction to $M \not\models r$.

Finally, consider the case that r is of the form $\beta_A \leftarrow I|_{D_A}$. Then, it follows that $\beta_A \notin M$. As $I \models \mathrm{B}(r)$, we again have $I \models A$, and therefore a contradiction, as then, by construction of M, we get $\beta_A \in M$. It follows that $M \models P^I_{\mathrm{SYY}}$.

By our initial assumption, it must hold that there is some $M' \subset M$ such that $M' \models P^I_{\mathrm{SYY}}$. Without loss of generality, assume M' is a minimal set with this property. First, consider the case that $M' \setminus \Gamma = I$. Let l be an atom from $M \setminus M'$. Then, either $l = \theta_A$ or $l = \beta_A$ for some C-atom A. In the former case, by minimality of M' and definition of P^I_{SYY}, there must be some C-rule $r = \theta_A \leftarrow A_1, \ldots, A_n$ in P^I_{SYY} such that $M' \models \mathrm{B}(r)$ as otherwise $M' \setminus \{\theta_A\}$ is an even smaller model of P^I_{SYY}. But then, $M' \not\models r$ holds, being a contradiction to $M' \models P^I_{\mathrm{SYY}}$. In the case that $l = \beta_A$, following the same line of argumentation, there must be some C-rule $r' = \beta_A \leftarrow I|_{D_A}$ in P^I_{SYY}. Since $I|_{D_A} \subseteq I$, we have $I \models \mathrm{B}(r')$. As I and M' only differ on Γ, we have that also $M' \models \mathrm{B}(r')$. This is in contradiction to $M' \models P^I_{\mathrm{SYY}}$ as then $M' \not\models r'$. It must hold that $M' \cap I \subset M' \cap I$. Consider the interpretation $I' = M' \cap I$. As I is an answer set of P, Condition ($\star$) of Definition 34 (on page 41) cannot hold for P, I, and I'. Hence, there is some $r \in P^I$ such that $I' \models \mathrm{B}(r)$ and, for all $A \in \mathrm{H}(r)$ with $I' \models A$, $I'|_{D_A} \neq I|_{D_A}$ holds. It follows that $I \models A$, for some $A \in \mathrm{H}(r)$. By construction of P^I_{SYY}, there must be some C-rule $r' \in P^I_{\mathrm{SYY}}$ of the form

$$r' = \beta_{A_1} \vee \cdots \vee \beta_{A_k} \leftarrow \theta_{B_1}, \ldots, \theta_{B_n}$$

that is constructed from r.

Next, we show that $M' \models \mathrm{B}(r')$. If $\mathrm{B}(r') = \emptyset$, this trivially holds. Otherwise, consider some $\theta_{B_j} \in \mathrm{B}(r')$. As $B_j \in \mathrm{B}(r)$ and $I \models \mathrm{B}(r)$, it follows that $I \models B_j$. Likewise, since $I' \models \mathrm{B}(r)$, also $I' \models B_j$ holds. Therefore, it follows that $I'|_{D_{B_j}} \in C_{B_j}$ and $I|_{D_{B_j}} \in C_{B_j}$. From the convexity of B_j, for all S where $I'|_{D_{B_j}} \subseteq S \subseteq I|_{D_{B_j}}$, it follows that $S \in C_{B_j}$. Hence,

$$\{I'|_{D_{B_j}} \cup \Delta \mid \Delta \in 2^{I|_{D_{B_j}} \setminus I'|_{D_{B_j}}}\} \subseteq C_{B_j}.$$

By Lemma 4, we get that there is some abstract prefixed power set $W \uplus V$ in $C^*_{B_j}$ such that $W \subseteq I'|_{D_{B_j}}$ and $I|_{D_{B_j}} \subseteq W \cup V$. Then, by Definition 43, $W \uplus V$ covers $I|_{D_{B_j}}$. From Definition 47, it follows that W is a satisfiable set of B_j with respect to $I|_{D_{B_j}}$. Therefore, by definition, P^I_{SYY} contains the C-rule $\theta_{B_j} \leftarrow W$. As $W \subseteq I'$ and $I' \subseteq M'$, we get that $M' \models W$. From $M' \models P^I_{\mathrm{SYY}}$, it follows that $\theta_{B_j} \in M'$. Hence, $M' \models \mathrm{B}(r')$ holds. There must be some $\beta_{A_h} \in \mathrm{H}(r')$ such that $M' \models \beta_{A_h}$. As $A_h \in \mathrm{H}(r)$, $r \in P^I$, and $I \in AS(P)$, we get that $I \models A_h$. By definition, P^I_{SYY} contains a C-rule $l \leftarrow \beta_{A_h}$, for every $l \in I|_{D_{A_h}}$. Therefore, since $M' \models P^I_{\mathrm{SYY}}$, we get $I|_{D_{A_h}} \subseteq M'$, and consequently $I|_{D_{A_h}} \subseteq I'$. Since

$I' \subseteq I$, $I|_{D_{A_h}} = I'|_{D_{A_h}}$ follows. Hence, $I \models A_h$ yields $I' \models A_h$. This is a contradiction that P, I, and I' do not satisfy Condition ($\star$) of Definition 34. Thus, I is an SYY-stable model of P. □

Regarding the converse direction, an even stronger result holds:

Theorem 12. *Let P be a* C*-program. If I is an SYY-stable model of P, then I is an answer set of P.*

Proof. Let I be an SYY-stable model of P and assume that I is not an answer set of P. From Lemma 5, we get that $I \models P$. Therefore, we also have that $I \models P^I$. There must be some $I' \subset I$ such that P, I, and I' obey Condition ($\star$) of Definition 34 (on page 41). Let M be the minimal model of P^I_{SYY} that causes I to be an SYY-stable model of P according to Definition 49. It holds that $I = M \setminus \Gamma$. Define

$$M' = M \cap (I' \cup \{\theta_A \mid I \models A \text{ and } I' \models A\} \cup \{\beta_A \mid I' \models A \text{ and } I'|_{D_A} = I|_{D_A}\}).$$

We have that $M' \subset M$ since $I' \subset I$, $I \subseteq M$, and $\Gamma \cap I = \emptyset$. By definition of an SYY-stable model, we have that $M' \not\models P^I_{\text{SYY}}$. Hence, there is a C-rule $r \in P^I_{\text{SYY}}$ such that $M' \not\models r$. We will now derive a contradiction as we show that such an r cannot exist.

Consider the case that r is of the form

$$\beta_{A_1} \vee \cdots \vee \beta_{A_k} \leftarrow \theta_{B_1}, \ldots, \theta_{B_n},$$

where r was constructed from the C-rule

$$r' = A_1 \vee \cdots \vee A_k \vee A_{k+1} \vee \cdots \vee A_m \leftarrow B_1, \ldots, B_n$$

contained in P. As $M' \models \mathrm{B}(r)$, by construction of M', we get that $I' \models \mathrm{B}(r')$ and $I \models \mathrm{B}(r')$. From the latter it follows that $r' \in P^I$. By Condition ($\star$) of Definition 34, there must be some $A_j \in \mathrm{H}(r')$ with $I' \models A_j$ and $I'|_{D_{A_j}} = I|_{D_{A_j}}$. Note that since $I \models A_j$, we have that $\beta_{A_j} \in \mathrm{H}(r)$. Moreover, by definition of P^I_{SYY}, the C-rule $\beta_{A_j} \leftarrow I|_{D_{A_j}}$ is contained in P^I_{SYY}. As $M \models I|_{D_{A_j}}$ and $M \models P^I_{\text{SYY}}$, it follows that $\beta_{A_j} \in M$. From that, together with the facts that $I' \models A_j$ and $I'|_{D_{A_j}} = I|_{D_{A_j}}$ both hold, we get, by the definition of M', that $\beta_{A_j} \in M'$, which is a contradiction to $M' \not\models r$.

Next, consider the case that r is of the form $\theta_A \leftarrow l_1, \ldots, l_n$. From $M' \models \mathrm{B}(r)$, it follows that $\mathrm{B}(r) \subseteq I'$ since $\mathrm{B}(r) \cap \Gamma = \emptyset$. Therefore, also $\mathrm{B}(r) \subseteq I$ holds. By definition of P^I_{SYY}, we have that $\mathrm{B}(r)$ is a satisfiable set of A with respect to $I|_{D_A}$. By Lemma 3, we get that

(†) for every S with $\mathrm{B}(r) \subseteq S \subseteq I|_{D_A}$, $S \in C_A$.

From $\mathrm{B}(r) \subseteq I'$ and $\mathrm{B}(r)$ being a satisfiable set of A with respect to $I|_{D_A}$, it follows $\mathrm{B}(r) \subseteq I|_{D_A}$, and we get in turn that $\mathrm{B}(r) \subseteq I'|_{D_A}$. Hence, by (†), we obtain $I'|_{D_A} \in C_A$. It follows that $I' \models A$. By definition of P^I_{SYY}, we also have that $I \models A$. Moreover, note that since $\mathrm{B}(r) \subseteq I$ and $I \subseteq M$, $M \models \mathrm{B}(r)$, and, consequently, as $M \models P^I_{\text{SYY}}$, we get that $\theta_A \in M$. Then, by construction of M', we have that $\theta_A \in M'$, being a contradiction to $M' \not\models r$.

Consider the case that r is of the form $l \leftarrow \beta_A$, where $I \models A$ and $l \in I|_{D_A}$. Since $M' \models \mathrm{B}(r)$, by construction of M', we get that $I'|_{D_A} = I|_{D_A}$. It follows that $l \in I'$, and hence also $l \in M'$, being a contradiction to $M' \not\models r$.

Now assume that r is of the form $\leftarrow l, \beta_A$, where $l \in D_A \setminus I|_{D_A}$. Hence, $l \in D_A$ and $l \notin I$. Since $M' \models \mathrm{B}(r)$, it follows that $l \in M'$. Therefore, by construction of M', and since $l \notin \Gamma$, we have that $l \in I'$, being a contradiction to $I' \subset I$.

Consider the final case that r is of the form $\beta_A \leftarrow I|_{D_A}$, where $I \models A$. From $M' \models \mathrm{B}(r)$ and $\mathrm{B}(r) \cap \Gamma = \emptyset$, we get $I|_{D_A} \subseteq I'$. Therefore, as $I' \subset I$, we have that $I'|_{D_A} = I|_{D_A}$. Notice that $M \models \mathrm{B}(r)$. Consequently, as $M \models P^I_{\mathrm{SYY}}$, we have that $\beta_A \in M$. By construction of M', we get $\beta_A \in M'$, being again a contradiction to $M' \not\models r$. It follows that I is an answer set of P. □

Due to known results from the literature (Shen et al., 2009; Liu et al., 2010; Son et al., 2007), Theorems 11 and 12 imply that our semantics is equivalent to a range of semantics proposed for more restricted classes of abstract-constraint programs including ones for *normal monotone abstract-constraint programs* (Marek and Truszczyński, 2004; Marek et al., 2008) and *normal convex abstract-constraint programs* (Liu and Truszczyński, 2006) that are based on a non-deterministic one-step provability operator.

Furthermore, there are semantics defined for normal abstract-constraint programs where every answer set in the respective approach is an answer set as defined in this thesis and where, if the considered C-programs are convex, also the converse holds, i.e., an answer set as defined in this thesis is also an answer set in the respective approach. In particular, these include

(i) the approach by Liu et al. (2010) based on a notion of computation,

(ii) the work of Son et al. (2007) that use the concept of conditional satisfaction of C-atoms for defining their semantics, and

(iii) the reduct-based semantics by Shen and You (2007).

Liu and Truszczyński (2006) showed that their semantics for normal convex C-programs resembles that of positive normal logic programs with weight constraints (Simons et al., 2002) with non-negative integer weights. This type of weight constraints can be represented by convex abstract-constraint atoms (cf. Section 4.3). Due to the relation of the semantics by Liu and Truszczyński and ours, it follows, in turn, that our semantics coincides with that of Simons et al. for that class of programs.

4.7.3 Solver Compatibility

The results of the last two subsections show that the semantics we have chosen as base for our framework of computations fits that of the solver languages of `Gringo`, `DLV`, and `DLVHEX` that were discussed in Section 3.6. We need an FLP-style treatment of non-convex literals for being compatible with `DLVHEX`, disjunctions to support `DLV` and `DLVHEX`, and we must allow for weight constraints in rule heads for compatibility with `Gringo`. As the `Gringo` language (like that of `DLV`) does not support non-convex literals so far, the mismatch between the semantics discussed in Section 4.7.1 to those of Section 4.7.2 is not relevant for the soundness of practical debugging as proposed in this work.

5 A Framework of Computations for Stepping

When an ASP developer detects that the answer sets of an answer-set program deviate from the expected semantics, the reason for this misfit can be given in terms of a definition or characterisations of the semantics. Typical definitions of answer sets are declarative and based on properties that consider whole programs and interpretations at once. Although formally elegant, when these definitions are used for explaining bugs, answers would then also be in terms of whole programs and interpretations and contain too much information to be of great value for locating the bug. Moreover, in practise, one has the problem to provide the entities that are considered to be given in the respective definition. While this is in most circumstances easy for the answer-set program, providing a candidate interpretation is often practically infeasible.

In this chapter, we are concerned with breaking the conceptual complexity of the definitions down to artefacts the programmer is familiar with—the rules the user has written, respectively their ground instances. To this end we introduce a framework of computations that captures the semantics we introduced in the previous chapter. In this computation model, an interpretation is built up step-by-step, by considering an increasing number of rule instances to be active. A *computation* in our framework is a sequence of *states* which are structures that keep information which rules and atoms have already been considered and what truth values were assigned to those atoms. Utilising the framework, only a rule and the atoms it contains have to be considered at once while building up an interpretation until an answer set is reached or a source for the unexpected behaviour becomes apparent.

In the next two sections we introduce states and computations. In Section 5.3, we will define and show some properties of computations that we need later on. Section 5.4 is concerned with the existence of a stable computation, that is a simpler form of computation that suffices for many popular classes of answer-set programs. In a related work, Liu et al. (2010) introduced a computation framework for normal C-programs. In order to clarify the relation between their notion of computation and ours, we give a comparison in Section 5.5.

Later, in Chapter 6, we develop abstractions for non-ground answer-set programs and the grounding step that allow us to capture the behaviour of practical grounding tools without formalising their concrete implementations. We then lift our framework of computations to this abstract non-ground setting.

5.1 States

Our framework is based on sequences of states, reassembling computations, in which an increasing number of ground rules are considered that build up a monotonically growing interpretation. Besides that interpretation, states also capture literals which cannot become true in subsequent steps and sets that currently lack external support in the state's interpretation.

Definition 50. A *state structure* S is a tuple $\langle P, I, I^-, \Upsilon \rangle$, where P is a set of C-rules, I is an interpretation, I^- a set of atoms such that I and I^- are disjoint, and Υ is a set of sets of atoms.

We call $D_S = I \cup I^-$ the *domain* of S and define $P_S = P$, $I_S = I$, $I^-{}_S = I^-$, and $\Upsilon_S = \Upsilon$.
A state structure $\langle P, I, I^-, \Upsilon \rangle$ is a *state* if

(i) $I \models \mathrm{B}(r)$ and $I \models^{\exists} \mathrm{H}(r)$ for every $r \in P$,

(ii) $D_r \subseteq D_S$ for every $r \in P$, and

(iii) $\Upsilon = \{X \subseteq I \mid X$ is unfounded in P with respect to $I\}$.

We call $\langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$ the *empty state*. $\diamond$

Intuitively, we use the first component P of a state to collect C-rules that the user has considered to be active and satisfied. The interpretation I collects atoms that have been considered true. Condition (i) ensures that P and I are compatible in the sense that every C-rule that is considered active and satisfied is active and satisfied with respect to I. Dual to I, the interpretation I^- collects atoms that the user has considered to be false. We require that all atoms appearing in a C-rule in P is either in I or in I^- which is expressed in Condition (ii). Finally, the set Υ keeps track of unfounded subsets of I, as stated in Condition (iii). Intuitively, as we will see later, when building a computation, the aim is to get rid of all unfounded sets (except for the empty set) in order to compute an answer set of a C-program. If a state does not contain such unfounded sets then we call it stable:

Definition 51. A state S is *stable* if $I_S \in AS(P_S)$. $\diamond$

The intuition is that when a state S is stable, no more C-rules need to be added to P_S to provide missing external support for the atoms in the current interpretation I_S. Note that a state S is stable exactly when $\Upsilon_S = \{\emptyset\}$. For example, the empty state is a stable state.

Example 26. Consider the C-rules

$$\begin{aligned} r_1 &: \langle \{a,b\}, \{\emptyset, \{a\}, \{b\}, \{a,b\}\} \rangle \leftarrow \mathit{not}\ a, \\ r_2 &: \qquad\qquad\qquad\qquad\qquad b \leftarrow a, \end{aligned}$$

and the state structures

$$\begin{aligned} S_1 &= \langle \{r_1\}, \emptyset, \{a,b\}, \{\emptyset\} \rangle, \\ S_2 &= \langle \{r_1\}, \{b\}, \{a\}, \{\emptyset\} \rangle, \\ S_3 &= \langle \{r_1\}, \{a,b\}, \emptyset, \{\emptyset\} \rangle, \\ S_4 &= \langle \{r_2\}, \{a,b\}, \emptyset, \{\emptyset\} \rangle, \\ S_5 &= \langle \{r_2\}, \{a,b\}, \emptyset, \{\{b\}, \{a,b\}\} \rangle. \end{aligned}$$

S_1 and S_2 are stable states. S_3 is not a state as $I_{S_3} \not\models \mathrm{B}(r_1)$. S_4 is not a state as the sets $\{b\}$ and $\{a,b\}$ are unfounded in P_{S_4} with respect to I_{S_4} but $\{b\} \notin \Upsilon_{S_4}$ and $\{a,b\} \notin \Upsilon_{S_4}$. S_5 is a state but not stable. ■

5.2 Computations

In what follows, we show how we can proceed forward in a computation, i.e., which states might follow a given state. This is expressed in the successor relation defined next.

Definition 52. For a state $S = \langle P, I, I^-, \Upsilon \rangle$ a state structure $S' = \langle P', I', I'^-, \Upsilon' \rangle$, S' is a *successor of* S if there is a C-rule $r \in P' \setminus P$ and sets $\Delta, \Delta^- \subseteq D_r$ such that

(i) $P' = P \cup \{r\}$,

(ii) $I' = I \cup \Delta$, $I'^- = I^- \cup \Delta^-$, and $D_S \cap (\Delta \cup \Delta^-) = \emptyset$,

(iii) $D_r \subseteq D_{S'}$,

(iv) $I \models \mathrm{B}(r)$,

(v) $I' \models \mathrm{B}(r)$ and $I' \models^{\exists} \mathrm{H}(r)$, and

(vi) $X' \in \Upsilon'$ iff $X' = X \cup \Delta'$, where $X \in \Upsilon$, $\Delta' \subseteq \Delta$, and r is not an external support for X' with respect to I'.

We denote r by $r_{new}(S, S')$. $\Diamond$

Condition (i) ensures that a successor state considers exactly one rule more to be active. Conditions (ii) and (iii) express that the interpretations I and I^- are extended by the so far unconsidered literals in Δ and Δ^- appearing in the new C-rule $r_{new}(S, S')$. Note that from S' being a state structure we get that Δ and Δ^- are distinct. A requirement for considering $r_{new}(S, S')$ as next C-rule is that it is active under the current interpretation I, expressed by Condition (iv). Moreover, $r_{new}(S, S')$ must be satisfied and still be active under the succeeding interpretation, as required by Condition (v). The final condition ensures that the unfounded sets of the successor are extensions of the previously unfounded sets that are not externally supported by the new rule.

Here, it is interesting that only extended previous unfounded sets can be unfounded sets in the extended C-program P' and that $r_{new}(S, S')$ is the only C-rule which could provide external support for them in P' with respect to the new interpretation I' as seen next.

Theorem 13. *Let S be a state and S' a successor of S, where $\Delta = I_{S'} \setminus I_S$. Moreover, let X' be a set of literals with $\emptyset \subset X' \subseteq I_{S'}$. Then, the following statements are equivalent:*

(*i*) *X' is unfounded in $P_{S'}$ with respect to $I_{S'}$.*

(*ii*) *$X' = \Delta' \cup X$, where $\Delta' \subseteq \Delta$, $X \in \Upsilon_S$, and $r_{new}(S, S')$ is not an external support for X' with respect to $I_{S'}$.*

Proof. ((i)⇒(ii)) It is obvious that $r_{new}(S, S')$ is not an external support for X' with respect to $I_{S'}$ as otherwise X' cannot be unfounded in $P_{S'}$ with respect to $I_{S'}$. It remains to be shown that $X' = \Delta' \cup X$ for some $\Delta' \subseteq \Delta$ and some $X \in \Upsilon_S$. Towards a contradiction, assume $X' \neq \Delta'' \cup X''$ for all $X'' \in \Upsilon_S$ and $\Delta'' \subseteq \Delta$. We define $X = X' \cap I_S$.

Consider the case that $X \in \Upsilon_S$. As $X' \setminus I_S \subseteq \Delta$, and $X' = (X' \setminus I_S) \cup X$, we have a contradiction to our assumption.

Therefore, it holds that $X \notin \Upsilon_S$. Hence, as $X \subseteq I_S$, by definition of a state, X is not unfounded in P_S with respect to I_S. Therefore, there is some external support $r \in P_S$ for X with respect to I_S.

In the following, we show that r is also an external support for X' with respect to $I_{S'}$. Since S' is a successor of S and S is a state, we get that I_S and $I_{S'}$ coincide on D_r. Consequently, from $I_S \models \mathrm{B}(r)$ we get that also $I_{S'} \models \mathrm{B}(r)$. Moreover, because of $I_S \setminus X \models \mathrm{B}(r)$ it is also true that $I_{S'} \setminus X' \models \mathrm{B}(r)$. Furthermore, we know that there is some $A \in \mathrm{H}(r)$ with $X|_{D_A} \neq \emptyset$ and $I_S|_{D_A} \subseteq C$, for some $C \in C_A$. As $X|_{D_A} = X'|_{D_A}$ and $I_S|_{D_A} = I_{S'}|_{D_A}$ we also have $X'|_{D_A} \neq \emptyset$ and $I_{S'}|_{D_A} \subseteq C$. Finally, note that for all $A \in \mathrm{H}(r)$ with $I_S \models A$, we have $(X \cap I_S)|_{D_A} \neq \emptyset$. Consider some $A \in \mathrm{H}(r)$ such that $I_{S'} \models A$. From the latter we get that $I_S \models A$ and therefore $(X \cap I_S)|_{D_A} \neq \emptyset$. As $X \cap I_S \subseteq X' \cap I_{S'}$, we also have $(X' \cap I_{S'})|_{D_A} \neq \emptyset$. Hence, r fulfils all conditions for being an external support for X' with respect to $I_{S'}$, which is a contradiction to X' being unfounded in $P_{S'}$ with respect to $I_{S'}$.

((ii)⇒(i)) Towards a contradiction, assume X' has some external support $r \in P_{S'}$ with respect to $I_{S'}$. From (ii) we know that $r \neq r_{new}(S, S')$ and $X' = \Delta' \cup X$ for some $\Delta' \subseteq \Delta$ and some $X \in \Upsilon_S$. As $r \neq r_{new}(S, S')$, we have that I_S and $I_{S'}$ coincide on D_r. Therefore, from $I_{S'} \models \mathrm{B}(r)$ and $I_{S'} \setminus X' \models \mathrm{B}(r)$, it follows that $I_S \models \mathrm{B}(r)$ and $I_S \setminus X' \models \mathrm{B}(r)$. Note that $X = X' \cap I_S$ and hence $I_S \setminus X \models \mathrm{B}(r)$. We know that there is some $A \in \mathrm{H}(r)$ with

$X'|_{D_A} \neq \emptyset$ and $I_{S'}|_{D_A} \subseteq C$, for some $C \in C_A$. As $X'|_{D_A} = X|_{D_A}$ we have $X|_{D_A} \neq \emptyset$. Moreover, as $I_{S'}|_{D_A} = I_S|_{D_A}$, it holds that $I_S|_{D_A} \subseteq C$. Finally, notice that for all $A \in \mathrm{H}(r)$ with $I_{S'} \models A$, we have $(X' \cap I_{S'})|_{D_A} \neq \emptyset$. Consider some $A \in \mathrm{H}(r)$ with $I_S \models A$. As $I_{S'}|_{D_A} = I_S|_{D_A}$, we also have $I_{S'} \models A$ and hence $(X' \cap I_{S'})|_{D_A} \neq \emptyset$. As $D_A \cap \Delta = \emptyset$, we have $(X' \cap I_{S'})|_{D_A} = (X \cap I_S)|_{D_A}$. Consequently, it holds that $(X \cap I_S)|_{D_A} \neq \emptyset$. We showed that r is an external support of X in P_S with respect to I_S. Therefore, we have a contradiction to $X \in \Upsilon_S$ because S is a state. □

The result shows that determining the unfounded sets in a computation after adding a further C-rule r can be done locally, i.e., only supersets of previously unfounded sets can be unfounded sets, and if such a superset has some external support then it is externally supported by r. The result also implies that the successor relation suffices to "step" from one state to another.

Corollary 3. *Let S be a state and S' a successor of S. Then, S' is a state.*

Proof. We show that the Conditions (i), (ii), and (iii) of Definition 50 hold for S'. Consider some rule $r \in P_{S'}$.

In case $r = r_{new}(S, S')$, $I_{S'} \models \mathrm{B}(r)$ and $I_{S'} \models^{\exists} \mathrm{H}(r)$ hold because of Item (v) of Definition 52 and $D_r \subseteq D_{S'}$ because of Item (iii) of the same definition.

Moreover, in case $r \neq r_{new}(S, S')$ we have $r \in P_S$. As S is a state we have $D_r \subseteq D_S$. Hence, since $D_S \subseteq D_{S'}$ also $D_r \subseteq D_{S'}$. Note that $I_{S'}|_{D_r} = I_S|_{D_r}$ because of Item (ii) of Definition 52. Therefore, as $I_S \models \mathrm{B}(r)$, $I_S \models^{\exists} \mathrm{H}(r)$ also $I_{S'} \models \mathrm{B}(r)$ and $I_{S'} \models^{\exists} \mathrm{H}(r)$. From these two cases we see that Conditions (i) and (ii) of Definition 50 hold for S'.

Finally, Condition (iii) follows from Item (vi) of Definition 52 and Theorem 13. □

Next, we define computations based on the notion of a state.

Definition 53. A *computation* is a sequence $C = S_0, \ldots, S_n$ of states such that S_{i+1} is a successor of S_i, for all $0 \leq i < n$. We call C *rooted* if S_0 is the empty state and *stable* if each S_i is stable, for $0 \leq i \leq n$. ♢

5.3 Properties

We next define when a computation has failed, gets stuck, or is complete. Intuitively, failure means that the computation reached a point where no answer set of the C-program can be reached. A computation is stuck when the last state activated rules deriving literals that are inconsistent with previously chosen active rules. It is considered complete when there are no more unconsidered active rules.

Definition 54. Let P be a C-program and $C = S_0, \ldots, S_n$ a computation such that $P_{S_n} \subseteq P$. Then, C is called a *computation for* P. Moreover,

- C has *failed for P at step i* if there is no answer set I of P such that $I_{S_i} \subseteq I$, ${I^-}_{S_i} \cap I = \emptyset$, and $P_{S_i} \subseteq P^I$;
- is *complete for P* if for every rule $r \in P^{I_{S_n}}$, we have $r \in P_{S_n}$;
- is *stuck in P* if it is not complete for P but there is no successor S_{n+1} of S_n such that $r_{new}(S_n, S_{n+1}) \in P$;
- *succeeded for P* if it is complete and S_n is stable. ♢

Example 27. Let P_{Ex27} be the C-program consisting of the C-rules

$$\begin{array}{ll} r_1: & a \leftarrow \langle\{a,b\},\{\emptyset,\{a,b\}\}\rangle, \\ r_2: & b \leftarrow a, \\ r_3: & a \leftarrow b, \\ r_4: \langle\{c\},\{\emptyset,\{c\}\}\rangle \leftarrow . & \\ r_5: & \leftarrow c. \end{array}$$

that has $\{a,b\}$ as its single answer set and consider the sequences

$$\begin{aligned} C_1 =& \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \\ & \langle\{r_4\},\{\},\{c\},\{\emptyset\}\rangle, \\ & \langle\{r_4,r_1\},\{a,b\},\{c\},\{\{a\},\{b\}\}\rangle, \\ C_2 =& \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \langle\{r_4\},\{\},\{c\},\{\emptyset\}\rangle, \langle\{r_4,r_1\},\{a,b\},\{c\},\{\{a\},\{b\}\}\rangle, \\ & \langle\{r_4,r_1,r_2\},\{a,b\},\{c\},\{\{a\}\}\rangle, \langle\{r_4,r_1,r_2,r_3\},\{a,b\},\{c\},\{\emptyset\}\rangle, \\ C_3 =& \langle\{r_4,r_1,r_2,r_3\},\{a,b\},\{c\},\{\emptyset\}\rangle, \\ C_4 =& \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \langle\{r_4\},\{c\},\emptyset,\{\emptyset\}\rangle, \\ C_5 =& \langle\{r_4,r_1,r_2,r_3\},\{a,b,c\},\emptyset,\{\emptyset\}\rangle, \\ C_6 =& \langle\{r_5\},\emptyset,\{c\},\{\emptyset\}\rangle, \quad \text{and} \\ C_7 =& \langle\emptyset,\emptyset,\emptyset,\{\emptyset\}\rangle, \langle\{r_4,r_1\},\{a,b\},\{c\},\{\{a\},\{b\}\}\rangle, \end{aligned}$$

C_1, C_2, C_3, C_4, and C_5 are computations for P_{Ex27}. The sequence C_6 is not a computation, as $\langle\{r_5\},\emptyset,\{c\},\{\emptyset\}\rangle$ is not a state. C_7 is not a computation, as the second state in C_7 is not a successor of the empty state. C_1, C_2, and C_4 are rooted. C_3, C_4, and C_5 are stable. C_2 and C_3 are complete and have succeeded for P_{Ex27}. C_1 is complete for $P_{Ex27} \setminus \{r_2,r_3\}$ but has failed for $P_{Ex27} \setminus \{r_2,r_3\}$ at step 0 because $P_{Ex27} \setminus \{r_2,r_3\}$ has no answer set. C_4 has failed for P_{Ex27} at step 1. C_5 has failed for P_{Ex27} at step 0 and is stuck in P_{Ex27}. ■

The following result guarantees the soundness of our framework of computations.

Theorem 14. *Let P be a* C*-program and $C = S_0,\ldots,S_n$ a computation that has succeeded for P. Then, I_{S_n} is an answer set of P.*

Proof. As C is complete for P we have $P^{I_{S_n}} \subseteq P_{S_n}$. Conversely, we have $P_{S_n} \subseteq P^{I_{S_n}}$ because for each $r \in P_{S_n}$ we have $r \in P$ and $I_{S_n} \models \mathrm{B}(r)$. By stability of S_n we get that $I_{S_n} \in AS(P_{S_n})$. As then $I_{S_n} \in AS(P^{I_{S_n}})$, the conjecture follows by definition. □

The computation model is also complete in the following sense:

Theorem 15. *Let S_0 be a state, P a* C*-program with $P_{S_0} \subseteq P$, and I an answer set of P with $I_{S_0} \subseteq I$ and $I \cap I^-{}_{S_0} = \emptyset$. Then, there is a computation $S_0,\ldots,S_n$ that has succeeded for P such that $P_{S_n} = P^I$ and $I_{S_n} = I$.*

Proof. The proof is by induction on the size of the set $P^I \setminus P_{S_0}$. Observe that from $I_{S_0} \subseteq I$, $I \cap I^-{}_{S_0} = \emptyset$, and $I_{S_0} \models \mathrm{B}(r)$ and $D_r \subseteq I_{S_0} \cup I^-{}_{S_0}$, for all $r \in P_{S_0}$, we get that $I \models \mathrm{B}(r)$ for all $r \in P_{S_0}$. Hence, as $P_{S_0} \subseteq P$, we have $P_{S_0} \subseteq P^I$.

Consider the base case that $|P^I \setminus P_{S_0}| = 0$. From $P_{S_0} \subseteq P^I$ we get $P_{S_0} = P^I$. Consider the sequence $C = \langle P_{S_0}, I_{S_0}, I^-{}_{S_0}, \Upsilon_{S_0}\rangle$. Towards a contradiction, assume $I_{S_0} \neq I$. As $I_{S_0} \subseteq I$ this means $I_{S_0} \subset I$. Hence, there is some $a \in I \setminus I_{S_0}$. As for all $r \in P_{S_0}$ it holds that $D_r \subseteq I_{S_0} \cup I^-{}_{S_0}$, and $I \cap I^-{}_{S_0} = \emptyset$, we get $a \notin D_{P_{S_0}}$. We have a contradiction to $I \in AS(P_{S_0})$ by Corollary 1 (on page 45), as $\{a\}$ is unfounded in P_{S_0} with respect to I. Consequently, $I_{S_0} = I$ must hold. As I_{S_0} is an answer set of P_{S_0} and S_0 is a state, we have that $\Upsilon_{S_0} = \{\emptyset\}$ by definition of state. It follows that C meets the criteria of the conjectured computation.

We proceed with the step case. As induction hypothesis, assume that the claim holds whenever $|P^I \setminus P_{S_0}| \leq i$ for an arbitrary but fixed $i \geq 0$. Consider some state S_0 and some $I \in AS(P_{S_0})$ for which the conditions in the premise hold such that $|P^I \setminus P_{S_0}| = i+1$. Towards a contradiction, assume there is no C-rule $r \in P^I \setminus P_{S_0}$ such that $I_{S_0} \models \mathrm{B}(r)$. Note that there is at least one C-rule $r' \in P^I \setminus P_{S_0}$ because $|P^I \setminus P_{S_0}| = i+1$. It cannot hold that $I = I_{S_0}$ since from $r' \in P^{I_{S_0}}$ follows $I_{S_0} \models \mathrm{B}(r')$. Consequently, we have $I_{S_0} \subset I$. Consider some $r'' \in P^I$ with $I_{S_0} \models \mathrm{B}(r'')$. By our assumption, we get that $r'' \in P_{S_0}$. It follows that $I_{S_0} \models r''$, and consequently there is some C-atom $A \in \mathrm{H}(r'')$ with $I_{S_0} \models A$. As $D_{r''} \subseteq D_{S_0}$, we have $D_A \subseteq I_{S_0} \cup {I^-}_{S_0}$. From that, since $I_{S_0} \subset I$ and $I \cap {I^-}_{S_0} = \emptyset$, we get $I|_{D_A} = I_{S_0}|_{D_A}$. Hence, we have a contradiction to I being an answer set of P by Definition 34 (on page 41).

So, there must be some C-rule $r \in P^I \setminus P_{S_0}$ such that $I_{S_0} \models \mathrm{B}(r)$. From $r \in P^I$ we get $I \models \mathrm{B}(r)$ and $I \models r$. Consider the state structure $S_1 = \langle P_1, I_1, {I_1}^-, \Upsilon_1 \rangle$, where $P_1 = P_{S_0} \cup \{r\}$, $I_1 = I_{S_0} \cup (I \cap D_r)$, ${I_1}^- = {I^-}_{S_0} \cup (D_r \setminus I)$, and

$$\Upsilon_1 = \{X \mid X = \Delta' \cup X', \text{ where } \Delta' \subseteq (I_1 \setminus I_{S_0}), X' \in \Upsilon_{S_0}, \text{ and } \\ r \text{ is not an external support of } X \text{ with respect to } I_1\}.$$

S_1 is a successor of state S_0, therefore S_1 is also a state by Corollary 3. As $P_1 \subseteq P$, $I_1 \subseteq I$, $I \cap {I_1}^- = \emptyset$, and $|P^I \setminus P_1| = i$, by the induction hypothesis we have that $S_1, \ldots, S_n$ is a computation, where S_n is a stable state, $P_{S_n} = P^I$, and $I_{S_n} = I$. Since S_1 is a successor of state S_0, we can conclude that also $S_0, S_1, \ldots, S_n$ is a computation. □

As the empty state, $\langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$, is trivially a state, we can make the completeness aspect of the previous result more apparent in the following corollary:

Corollary 4. *Let P be a* C*-program and $I \in AS(P)$. Then, there is a rooted computation $S_0, \ldots, S_n$ that has succeeded for P such that $P_{S_n} = P^I$ and $I_{S_n} = I$.*

Proof. The claim follows immediately from Theorem 15 in case $S_0 = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. □

Note, that there are states that do not result from rooted computations, e.g., the state

$$\langle \{a \leftarrow b\}, \{a, b\}, \emptyset, \{\emptyset, \{a, b\}, \{b\}\} \rangle$$

is not a successor of any other state. However, for stable states we can guarantee the existence of such a computation.

Corollary 5. *Let S be a stable state. Then, there is a rooted computation $S_0, \ldots, S_n$ with $S_n = S$.*

Proof. The result is a direct consequence of Corollary 5 and Definition 51 (on page 58). □

The next theorem lays the ground for the jumping technique we will introduce in Chapter 7. Its purpose is to allow for extending a computation by considering multiple rules of a program at once and using ASP solving itself for creating this extension.

Theorem 16. *Let P be a* C*-program, $C = S_0, \ldots, S_n$ a computation for P, P' a set of* C*-rules with $P' \subseteq P$, and I an answer set of $P_{S_n} \cup P'$ with $I_{S_n} \subseteq I$ and $I \cap {I^-}_{S_n} = \emptyset$. Then, there is a computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ for P, such that S_m is stable, $P_{S_m} = P_{S_n} \cup {P'}^I$ and $I_{S_m} = I$.*

Proof. By Theorem 15, as $P_{S_n} \subseteq P_{S_n} \cup P'$, $I_{S_n} \subseteq I$, and $I \cap {I^-}_{S_n} = \emptyset$, there is a computation $S_n, \ldots, S_m$ that has succeeded for $P_{S_n} \cup P'$ such that $P_{S_m} = (P_{S_n} \cup P')^I$ and $I_{S_m} = I$. Then, S_m is stable and, as ${P_{S_n}}^I = P_{S_n}$, we have $P_{S_m} = P_{S_n} \cup {P'}^I$. As $P_{S_m} \subseteq P$ we have that $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ is a computation for P. □

The following result illustrates that the direction one chooses for building up a certain interpretation, i.e., the order of the rules considered in a computation, is irrelevant in the sense that eventually the same state will be reached.

Proposition 2. *Let P be a* C*-program. Furthermore, let and $C = S_0, \ldots, S_n$ and $C' = S'_0, \ldots, S'_m$ be computations complete for P such that $S_0 = S'_0$. Then, $I_{S_n} = I_{S'_m}$ iff $S_n = S'_m$ and $n = m$.*

Proof. The "if" direction is trivial. Let $I = I_{S_n} = I_{S'_m}$. Towards a contradiction, assume $P_{S_n} \neq P_{S'_m}$. Without loss of generality we focus on the case that there is some $r \in P_{S_n}$ such that $r \notin P_{S'_m}$. Then, it holds that $I \models r$, $I \models \mathrm{B}(r)$, and $r \in P$. Consequently, $r \in P^I$. By completeness of C', we have $r \in P_{S'_m}$ which contradicts our assumption. Hence, we have $P_{S_n} = P_{S'_m}$.

By definition of a state, from $I_{S_n} = I_{S'_m}$ and $P_{S_n} = P_{S'_m}$, it follows that $\Upsilon_{S_n} = \Upsilon_{S'_m}$. Towards a contradiction, assume $I^-{}_{S_n} \neq I^-{}_{S'_m}$. Without loss of generality we focus on the case that there is some $a \in I^-{}_{S_n}$ such that $a \notin I^-{}_{S'_m}$. Consider the integer i where $0 < i \leq n$ such that $a \in I^-{}_{S_i}$ but $a \notin I^-{}_{S_{i-1}}$. Then, by definition of a successor, for $r = r_{new}(S_{i-1}, S_i)$, we have $a \in \Delta^-$ for some $\Delta^- \subseteq D_r$. As then $a \in D_r$ and, as $P_{S_n} = P_{S'_m}$, we have $r \in P_{S'_m}$, it must hold that $a \in D_{S'_m}$ by definition of a state structure. From $I \cap I^-{}_{S_n} = \emptyset$ we know that $a \notin I$. Therefore, since $a \in I \cup I^-{}_{S'_m}$, we get that $a \in I^-{}_{S'_m}$, being a contradiction to our assumption. As then $S_n = S'_m$, $P_{S_0} = P_{S'_0}$, and since in every step in a computation exactly one rule is added it must hold that $n = m$. □

For rooted computations, the domain of each state is determined by the atoms in the C-rules it contains.

Proposition 3. *Let $C = S_0, \ldots, S_n$ be a rooted computation. Then, $I_{S_i} = I_{S_n}|_{D_{P_{S_i}}}$ and $I^-{}_{S_i} = I^-{}_{S_n}|_{D_{P_{S_i}}}$, for all $0 \leq i \leq n$.*

Proof. The proof is by contradiction. Let j be the smallest index with $0 \leq j \leq n$ such that $I_{S_j} \neq I_{S_n}|_{D_{P_{S_j}}}$ or $I^-{}_{S_j} \neq I^-{}_{S_n}|_{D_{P_{S_j}}}$. Note that $0 < j$ as $I_{S_0} = I^-{}_{S_0} = D_{P_{S_0}} = \emptyset$.

As S_j is a successor of S_{j-1}, we have $I_{S_j} = I_{S_{j-1}} \cup \Delta$ and $I^-{}_{S_j} = I^-{}_{S_{j-1}} \cup \Delta^-$, where $\Delta, \Delta^- \subseteq D_{r_{new}(S_{j-1}, S_j)}$, $D_{S_{j-1}} \cap (\Delta \cup \Delta^-) = \emptyset$, and $D_{r_{new}(S_{j-1}, S_j)} \subseteq I_{S_j} \cup I^-{}_{S_j}$. As we have $I_{S_{j-1}} = I_{S_n}|_{D_{P_{S_{j-1}}}}$ and $I^-{}_{S_{j-1}} = I^-{}_{S_n}|_{D_{P_{S_{j-1}}}}$, it holds that

$$I_{S_{j-1}} \cup I_{S_n}|_{D_\delta} = I_{S_n}|_{D_{P_{S_{j-1}}}} \cup I_{S_n}|_{D_\delta} = I_{S_n}|_{D_{P_{S_j}}}$$

and

$$I^-{}_{S_{j-1}} \cup I^-{}_{S_n}|_{D_\delta} = I^-{}_{S_n}|_{D_{P_{S_{j-1}}}} \cup I^-{}_{S_n}|_{D_\delta} = I^-{}_{S_n}|_{D_{P_{S_j}}},$$

where $D_\delta = D_{P_{S_j}} \setminus D_{P_{S_{j-1}}}$.

For establishing the contradiction, it suffices to show that

$$I_{S_n}|_{D_\delta} = \Delta$$

and

$$I^-{}_{S_n}|_{D_\delta} = \Delta^-.$$

Consider some $a \in \Delta$. Then, $a \in D_\delta$ because $a \in D_{r_{new}(S_{j-1}, S_j)}$, $D_{S_{j-1}} \cap (\Delta \cup \Delta^-) = \emptyset$, and $D_{P_{S_{j-1}}} \subseteq D_{S_{j-1}}$. Moreover, $a \in I_{S_j}$ implies $a \in I_{S_n}$ and therefore $\Delta \subseteq I_{S_n}|_{D_\delta}$. Now, consider some $b \in I_{S_n}|_{D_\delta}$. As $D_{r_{new}(S_{j-1}, S_j)} \subseteq I_{S_j} \cup I^-{}_{S_j}$, we have $b \in I_{S_j} \cup I^-{}_{S_j}$. Consider the case that $b \in I^-{}_{S_j}$. Then, also $b \in I^-{}_{S_n}$ which is a contradiction to $b \in I_{S_n}$ as S_n is a state structure. Hence, $b \in I_{S_j} = I_{S_{j-1}} \cup \Delta$. First, assume $b \in I_{S_{j-1}}$. This leads to a contradiction as then $b \in D_{P_{S_{j-1}}}$ since $I_{S_{j-1}} = I_{S_n}|_{D_{P_{S_{j-1}}}}$.

It follows that $b \in \Delta$ and therefore $\Delta = I_{S_n}|_{D_\delta}$. One can show that $\Delta^- = I^-{}_{S_n}|_{D_\delta}$ analogously. □

5.4 Stable Computations

In this section we are concerned with the existence of stable computations, i.e., computations that do not involve unfounded sets. We single out an important class of C-programs for which one can solely rely on this type of computation and also give examples of C-programs that do not allow for succeeding stable computations.

Intuitively, the Σ_2^P-hardness of our semantics, as shown in Section 4.6, demands for unstable computations in the general case. This becomes obvious when considering that for a given C-program, one could guess a candidate sequence C for a stable computation in polynomial time. Then, a polynomial number of checks whether each state is a successor of the previous one in the sequence suffices to establish whether C is a computation. Following Definition 52 on page 58, these checks can be done in polynomial time when we are allowed to omit Condition (vi) for unfounded sets. Hence, answer-set existence for the class of C-programs for which every answer set can be built up with stable computations is in NP.

Naturally, it is interesting whether there are syntactic classes of C-programs for which we can rely on stable computations only. It turns out that many syntactically simple C-programs already require the use of unfounded sets.

Example 28. Consider C-program P_{Ex28} consisting of the C-rules

$$\begin{aligned} r_1 &: a \leftarrow b \qquad \text{and} \\ r_2 &: b \leftarrow \langle \{a\}, \{\emptyset, \{a\}\}\rangle. \end{aligned}$$

We have that $\{a, b\}$ is the only answer set of P_{Ex28} and

$$\begin{aligned} C = &\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle, \\ &\langle \{r_2\}, \{a, b\}, \emptyset, \{\emptyset, \{a\}\}\rangle, \\ &\langle \{r_2, r_1\}, \{a, b\}, \emptyset, \{\emptyset\}\rangle \end{aligned}$$

is the only computation that succeeds for P_{Ex28}. ■

As Example 28 shows, unstable computations are already required for a C-program without disjunction and a single monotone C-atom. Hence, also the use of weaker restrictions, like convexity of C-atoms or some notion of head-cycle freeness (Ben-Eliyahu and Dechter, 1994), is not sufficient.

One can observe, that the C-program from the example has cyclic positive dependencies between atoms a and b. Hence, we next explore whether such dependencies influence the need for computations that are not stable. To this end, we introduce notions of *positive dependency* in a C-program.

Definition 55. Let S be a set of C-literals. Then, the *positive normal form* of S is given by

$$S^+ = \{A \mid A \in S, A \text{ is a C-atom}\} \cup \{\bar{A} \mid \mathit{not}\ A \in S\},$$

where $\bar{A} = \langle D_A, 2^{D_A} \setminus C_{D_A}\rangle$ is the *complement* of A. Furthermore, the *set of positive atom occurences* in S is given by $\mathit{posOcc}(S) = \bigcup_{A \in S^+} C_A$.

Let P be a C-program. The *positive dependency graph* of P is the directed graph

$$G(P) = \langle D_P, \{\langle a, b\rangle \mid r \in P, a \in \mathit{posOcc}(\mathrm{H}(r)), b \in \mathit{posOcc}(\mathrm{B}(r))\}\rangle. \qquad \Diamond$$

We next introduce the notion of *absolute tightness* for describing C-programs without cyclic positive dependencies after recalling basic notions of graph theory. For a (directed) graph $G = \langle V, \prec\rangle$, the *reachability relation* of G is the transitive closure of $\prec$. Let $\prec'$ be the reachability relation of G. Then, G is *acyclic* if there is no $v \in V$ such that $v \prec' v$.

Definition 56. Let P be a C-program. P is *absolutely tight* if $G(P)$ is acyclic. $\Diamond$

One could assume that absolute tightness paired with convexity or monotonicity is sufficient to guarantee stable computations because absolute tightness forbids positive dependencies among disjuncts and the absence of such dependencies lowers the complexity of LP-programs (Ben-Eliyahu and Dechter, 1994). However, as the following example illustrates, this is not the case for C-programs.

Example 29. Consider C-program P_{Ex29} consisting of the C-rules

$$r_1 : a \vee \langle \{a, b\}, \{\{a\}, \{a, b\}\}\rangle \leftarrow \qquad \text{and}$$
$$r_2 :\ b \vee \langle \{a, b\}, \{\{b\}, \{a, b\}\}\rangle \leftarrow .$$

We have that $\{a, b\}$ is the only answer set of P_{Ex29} and

$$\begin{aligned} C_1 = &\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle, \\ &\langle \{r_1\}, \{a, b\}, \emptyset, \{\emptyset, \{b\}\}\rangle, \\ &\langle \{r_1, r_2\}, \{a, b\}, \emptyset, \{\emptyset\}\rangle \qquad \text{and} \end{aligned}$$

$$\begin{aligned} C_2 = &\langle \emptyset, \emptyset, \emptyset, \{\emptyset\}\rangle, \\ &\langle \{r_2\}, \{a, b\}, \emptyset, \{\emptyset, \{a\}\}\rangle, \\ &\langle \{r_1, r_2\}, \{a, b\}, \emptyset, \{\emptyset\}\rangle \end{aligned}$$

are the only computations that succeed for P_{Ex29}. Clearly, P_{Ex29} is monotone and, as the rule bodies of r_1 and r_2 are empty, absolutely tight but C_1 and C_2 are not stable. ■

Nevertheless, we can assure the existence of stable computations for answer sets of normal C-programs that are absolutely tight and convex. This is good news, as this class corresponds to a large subset of typical answer-set programs written for solvers like `Clasp` or `Smodels` that do not rely on disjunction as their guessing device.

For establishing our results we make use of the following notion which reflects positive dependency on the rule level.

Definition 57. The *positive rule dependency graph* of P is given by

$$G_{\mathfrak{R}}(P) = \langle P, \{\langle r_1, r_2\rangle \mid r_1, r_2 \in P, posOcc(\mathrm{B}(r_1)) \cap posOcc(\mathrm{H}(r_2)) \neq \emptyset\}\rangle. \qquad \Diamond$$

We can relate the two notions of dependency graph as follows.

Lemma 6. *Let P be a C-program. $G_{\mathfrak{R}}(P)$ is acyclic iff $G(P)$ is acyclic.*

Proof. Let $\prec_{\mathfrak{D}}$ denote the edge relation of $G(P)$ and $\prec_{\mathfrak{R}}$ that of $G_{\mathfrak{R}}(P)$.

($\Rightarrow$) Assume $G(P)$ is not acyclic. There must be some path $a_1, \dots, a_n$ of atoms a_i such that for $1 \leq i < n$, we have $a_i \in D_P$, $a_i \prec_{\mathfrak{D}} a_{i+1}$, and $a_1 = a_n$. Hence, by the definition of $G(P)$, there must be a sequence $r_1, \dots, r_{n-1}$ such that for each $1 \leq i \leq n-1$, $r_i \in P$, $a_i \in posOcc(\mathrm{H}(r_i))$, and $a_{i+1} \in posOcc(\mathrm{B}(r_i))$. Therefore, for each $1 \leq i < n-1$, we have $r_{i+1} \prec_{\mathfrak{R}} r_i$. Note that $a_1 \in posOcc(\mathrm{H}(r_1))$ and $a_1 \in posOcc(\mathrm{B}(r_{n-1}))$. Consequently, we have $r_{n-1} \prec_{\mathfrak{R}} r_1$ and thus $r_1, r_{n-1}, \dots, r_1$ forms a cycle in $G_{\mathfrak{R}}(P)$. It follows that $G_{\mathfrak{R}}(P)$ is not acyclic.

($\Leftarrow$) Assume now that $G_{\mathfrak{R}}(P)$ is not acyclic. There must be some path $r_1, \dots, r_n$ of C-rules r_i such that for $1 \leq i < n$ we have $r_i \in P$, $r_1 = r_n$, and $r_i \prec_{\mathfrak{R}} r_{i+1}$. Hence, by the definition of $G_{\mathfrak{R}}(P)$, there must be a sequence $a_1, \dots, a_{n-1}$ such that for each $1 \leq i \leq n-1$, $a_i \in posOcc(\mathrm{H}(r_{i+1}))$, and $a_i \in posOcc(\mathrm{B}(r_i))$. Therefore, for each $1 \leq i < n-1$ we have $a_{i+1} \prec_{\mathfrak{D}} a_i$. Note that $a_{n-1} \in posOcc(\mathrm{H}(r_1))$ and $a_1 \in posOcc(\mathrm{B}(r_1))$. Consequently, we have $a_{n-1} \prec_{\mathfrak{D}} a_1$ and thus $a_1, a_{n-1}, \dots, a_1$ forms a cycle in $G(P)$. We have that $G_{\mathfrak{R}}(P)$ is not acyclic. □

Lemma 7. *Let P be an absolutely tight* C*-program. There is a strict total order $\prec$ on P that extends the reachability relation of $G_{\mathfrak{R}}(P)$.*

Proof. By Definition 56, $G(P)$ is acyclic. Hence, by Lemma 6, $G_{\mathfrak{R}}(P)$ is also acyclic. The conjecture holds, since every directed acyclic tree has a topological ordering. □

We now have the means to show the following result, guaranteeing the existence of stable computations.

Theorem 17. *Let* $\mathrm{C} = S_0, \ldots, S_n$ *be a computation such that S_0 and S_n are stable and* $P_\Delta = P_{S_n} \setminus P_{S_0}$ *is a normal, convex, and absolutely tight* C*-program. Then, there is a stable computation* $\mathrm{C}' = S'_0, \ldots, S'_n$ *such that* $S_0 = S'_0$ *and* $S_n = S'_n$*.*

Proof. Let $\prec$ be the strict total order extending the reachability relation of $G_{\mathfrak{R}}(P_\Delta)$ that is guaranteed to exist by Lemma 7. Let $r(\cdot) : \{1, \ldots, n\} \mapsto P_\Delta$ denote the one-to-one mapping from the integer interval $\{1, \ldots, n\}$ to the C-rules from P_Δ such that for all i, j in the range of $r(\cdot)$, we have that $i < j$ implies $r(j) \prec r(i)$. Consider the sequence $\mathrm{C}' = S'_0, \ldots, S'_n$, where $S'_0 = S_0$, and for all $0 \leq i < n$,

$$P'_{i+1} = P'_i \cup \{r(i+1)\},$$

$$I_{S'_{i+1}} = I_{S'_i} \cup (I_{S_n} \cap D_{r(i+1)}),$$

$$I^-{}_{S'_{i+1}} = I^-{}_{S'_i} \cup (I^-{}_{S_n} \cap D_{r(i+1)}), \quad \text{and}$$

$$\Upsilon_{S'_{i+1}} = \{\emptyset\}.$$

Notice that $S'_n = S_n$ and

$$I_{S'_{i+1}}|_{D_{P_{S'_i}}} = I_{S'_i}|_{D_{P_{S'_i}}},$$

for all $0 \leq i < n$. We show that C' is a computation by induction on the length of a subsequence of C'.

As base case consider the sequence $\mathrm{C}'' = S'_0$. As $S'_0 = S_0$ and S_0 is a state, C'' is a computation. For the induction hypothesis, assume that for some arbitrary but fixed i with $0 \leq i < n$, the sequence $S'_0, \ldots, S'_i$ is a computation.

In the induction step it remains to be shown that S'_{i+1} is a successor of S'_i. Clearly, S'_{i+1} is a state structure, and by definition of C', since C is a computation and

$$I_{S'_{i+1}}|_{D_{P_{S_{i+1}}}} = I_{S_n}|_{D_{P_{S_{i+1}}}},$$

Conditions (i), (ii), (iii), and (v) of Definition 52 (on page 58) for being a successor of S'_i are fulfilled by S'_{i+1}. Let Δ denote $I_{S'_{i+1}} \setminus I_{S'_i}$.

Next we show that Condition (iv) holds, i.e., $I_{S'_i} \models \mathrm{B}(r(i+1))$. Note that since Condition (v) holds, we have $I_{S'_{i+1}} \models \mathrm{B}(r(i+1))$ and hence (iv) holds in the case $\Delta = \emptyset$. Towards a contradiction assume $\Delta \neq \emptyset$ and $I_{S'_i} \not\models \mathrm{B}(r(i+1))$. We define $\Delta_{B^+} = \Delta \cap posOcc(\mathrm{B}(r(i+1)))$.

First, consider the case that $\Delta_{B^+} = \emptyset$. As $I_{S'_i} \not\models \mathrm{B}(r(i+1))$, there must be some C-literal $L \in \mathrm{B}(r(i+1))$ such that $I_{S'_i} \not\models L$. We know that $I_{S'_{i+1}} \models L$. Consequently, $I_{S'_i}|_{D_L} \subset I_{S'_{i+1}}|_{D_L}$ and therefore $\Delta|_{D_L} \neq \emptyset$. Moreover, from $I_{S'_{i+1}} \models L$ we have

$$I_{S'_{i+1}}|_{D_L} \subseteq posOcc(\mathrm{B}(r(i+1))).$$

It follows that

$$\Delta|_{D_L} \cap posOcc(\mathrm{B}(r(i+1))) \neq \emptyset,$$

indicating a contradiction to $\Delta_{B^+} = \emptyset$. It holds that $\Delta_{B^+} \neq \emptyset$. Note that $X \subseteq I_{S_n}$. From that, since S_n is a state, there must be some C-rule $r_{\Delta_{B^+}} \in P_{S_n}$ such that $r_{\Delta_{B^+}}$ is an external support for Δ_{B^+} with respect to I_{S_n}. It cannot be the case that $r \in P_{S_0}$, since $\Delta_{B^+} \cap I_{S'_i} = \emptyset$, therefore, $r_{\Delta_{B^+}} \in P_\Delta$. As $r_{\Delta_{B^+}}$ is an external support for Δ_{B^+} with respect to I_{S_n}, for $\{A\} = \mathrm{H}(r_{\Delta_{B^+}})$, we have $I_{S_n} \models A$ and $\Delta_{B^+}|_{D_A} \neq \emptyset$.

Consider the case that $r_{\Delta_{B^+}} = r(i+1)$. From that we get that $posOcc(\mathrm{H}(r(i+1))) \cap \Delta_{B^+} \neq \emptyset$. This, in turn, implies

$$posOcc(\mathrm{H}(r(i+1))) \cap posOcc(\mathrm{B}(r(i+1))) \neq \emptyset$$

which is a contradiction to $G_{\mathfrak{R}}(P_\Delta)$ being acyclic. Note that the latter is guaranteed by absolute tightness of P_Δ and Lemma 6.

Consider the case that $r(i+1) \prec r_{\Delta_{B^+}}$. Then, by definition of C$'$ we have that $r_{\Delta_{B^+}} \in P_{S'_i}$. Hence, from $\Delta_{B^+}|_{D_A} \neq \emptyset$ follows

$$\Delta_{B^+}|_{D_{P_{S'_i}}} \neq \emptyset$$

and thus

$$I_{S'_{i+1}} \setminus I_{S'_i}|_{D_{P_{S'_i}}} \neq \emptyset.$$

The latter is a contradiction to

$$I_{S'_{i+1}}|_{D_{P_{S'_i}}} = I_{S'_i}|_{D_{P_{S'_i}}}.$$

Consider the remaining case that $r_{\Delta_{B^+}} \prec r(i+1)$. As $\Delta_{B^+}|_{D_A} \neq \emptyset$, $\Delta_{B^+} \subseteq I_{S_n}$, and $I_{S_n}|_{D_A} \in C_A$, we have that

$$posOcc(\mathrm{H}(r_{\Delta_{B^+}})) \cap \Delta_{B^+} \neq \emptyset.$$

Therefore, we have $posOcc(\mathrm{H}(r_{\Delta_{B^+}})) \cap posOcc(\mathrm{B}(r(i+1))) \neq \emptyset$. This implies $r(i+1) \prec r_{\Delta_{B^+}}$, being a contradiction to $\prec$ being a strict order as we also have $r_{\Delta_{B^+}} \prec r(i+1)$. We showed that Condition (iv) of Definition 52 (on page 58) for being a successor of S'_i holds for S'_{i+1}.

Towards a contradiction assume Condition (vi) does not hold. Hence, it must hold that there is some $\Delta' \subseteq \Delta$ such that $\Delta' \neq \emptyset$ and $r(i+1)$ is not an external support for Δ' with respect to $I_{S'_{i+1}}$. We have $I_{S'_{i+1}} \models \mathrm{B}(r(i+1))$ and since we already know that $I_{S'_i} \models \mathrm{B}(r(i+1))$, also $I_{S'_{i+1}} \setminus \Delta' \models \mathrm{B}(r(i+1))$ holds by convexity of P_Δ. Moreover, as $I_{S'_{i+1}} \models r(i+1)$, it must hold that $I_{S'_{i+1}} \models A$ for $\mathrm{H}(r(i+1)) = \{A\}$. Consequently, for $r(i+1)$ not to be an external support for Δ' with respect to $I_{S'_{i+1}}$, we have $\Delta'|_{D_A} = \emptyset$. As then $\Delta'|_{D_{\mathrm{H}(r(i+1))}} = \emptyset$ but $\Delta'|_{D_{r(i+1)}} \neq \emptyset$ it must hold that $\Delta'|_{D_{\mathrm{B}(r(i+1))}} \neq \emptyset$. Consider $\Delta'' = \Delta' \cap posOcc(\mathrm{B}(r(i+1)))$ and assume that $\Delta'' \neq \emptyset$. Then, as $\Delta'' \subseteq I_{S_n}$, there must be some C-rule $r_{\Delta''}$ that is an external support for Δ'' with respect to I_{S_n}. Hence, $posOcc(\mathrm{H}(r_{\Delta''})) \cap \Delta'' \neq \emptyset$ and therefore $posOcc(\mathrm{H}(r_{\Delta''})) \cap posOcc(\mathrm{B}(r(i+1))) \neq \emptyset$. It follows that $r(i+1) \prec r_{\Delta''}$. From that we get $r_{\Delta''} \in P_{S'_i}$. This is a contradiction as we know that $posOcc(\mathrm{H}(r_{\Delta''})) \cap \Delta'' \neq \emptyset$, $posOcc(\mathrm{H}(r_{\Delta''})) \cap \Delta'' \subseteq I_{S'_i}$, and $\Delta'' \subseteq I_{S'_{i+1}} \setminus I_{S'_i}$. Consequently, $\Delta' \cap posOcc(\mathrm{B}(r(i+1))) = \emptyset$ must hold. From $\Delta'|_{D_{\mathrm{B}(r(i+1))}} \neq \emptyset$ we get that there is some $L \in \mathrm{B}(r(i+1))$ with $\Delta'|_{D_L} \neq \emptyset$. As $I_{S'_{i+1}} \models L$, we have that $I_{S'_{i+1}}|_{D_L} \in C$ in the case L is a C-atom $L = \langle D_L, C \rangle$, and $I_{S'_{i+1}}|_{D_L} \in 2^D_L \setminus C$ in the case L is a default negated C-atom $L = not\ \langle D_L, C \rangle$. In both cases, as $\Delta' \subseteq I_{S'_{i+1}}$ and $\Delta'|_{D_L} \neq \emptyset$, we get a contradiction to $\Delta' \cap posOcc(\mathrm{B}(r(i+1))) = \emptyset$. □

As a direct consequence of Theorem 17 and Corollary 4 (on page 62), we get an improved completeness result for normal convex C-programs that are absolutely tight, i.e., we can find a computation that consists of stable states only.

Corollary 6. *Let P be a normal* C*-program that is convex and absolutely tight, and consider some $I \in AS(P)$. Then, there is a rooted stable computation $S_0, \dots, S_n$ such that $P_{S_n} = P^I$ and $I_{S_n} = I$.*

Proof. From $I \in AS(P)$, we get by Corollary 4 that there is a rooted computation $S_0, \dots, S_n$ such that $P_{S_n} = P^I$ and $I_{S_n} = I$. Note that S_0 is the empty state. S_0 and S_n are stable according to Definition 51 on page 58. From Theorem 17 we can conclude the existence of another computation $C' = S'_0, \dots, S'_n$ such that $S_0 = S'_0$ and $S_n = S'_n$ that is stable. Clearly, C' is also rooted. □

5.5 Comparison to Computations by Liu et al.

As mentioned in Section 4.7.2, Liu et al. (2010) also use a notion of computation to characterise their semantics for normal C-programs. These computations are sequences of evolving interpretations. Unlike the three-valued ones used for online justifications (Pontelli et al., 2009) (cf. Section 7.8), these carry only information about atoms considered true. Thus, conceptionally, they correspond to sequences $I_{S_0}, I_{S_1}, \dots$ where $S_0, S_1, \dots$ is a computation in our sense. The authors formulate principles for characterising different variants of computations. We will highlight differences and commonalities between the approaches along the lines of some of these properties.

One main structural difference between their and our notion of computation is the granularity of steps: In the approach by Liu et al. it might be the case that multiple rules must be considered at once, as required by their *revision property* (R') while in our case computation proceeds rule instance by rule instance. The purpose of property (R') is to assure that every successive interpretation must be supported by the rules active with respect to the previous interpretation. But it also requires that every active rule in the overall program is satisfied after each step, whereas we allow rule instances that were not considered yet in the computation to be unsatisfied. For the purpose of debugging, rule-based computation granularity seems favourable as rules are our primary source code artifacts. Moreover, ignoring parts of the program that were not considered yet in a computation is essential in the stepping method, as this breaks down the amount of information that has to be considered by the user at once and allows for getting stuck and thereby detect discrepancies between his or her understanding of the program and its actual semantics.

Our computations (when translated as above) meet the *persistence principle* (P') of Liu et al. that ensures that a successor's interpretation is always a superset of the current one.

Their *convergence principle* (C'), requiring that a computation stabilises to a supported model, is not met by our computations, as we do not enforce support in general. However, when a computation has succeeded (cf. Definition 54 on page 60), it meets this property.

A further difference is that Liu et al. do not allow for non-stable computations as required by the founded persistence of reasons principle (FPr). This explains why the semantics they characterise treats non-convex atoms, e.g., like in the example in the introduction to Section 4.7.2, not in the FLP-way. Besides that, the use of non-stable computations allow us to handle disjunction. Interestingly, Liu et al. mention the support for disjunction in computations as an open challenging problem and suspect the necessity of a global minimality requirement on computations for this purpose. Our framework demonstrates that we can do without such a condition: As shown in Theorem 13 on page 59, unfounded sets in our semantics can be computed incrementally "on-the-fly" by considering only the rule instance added in a step as potential new external support.

Finally, the *principle of persistence of reasons* (Pr') suggests that the "reason" for the truth value of an atom must not change in the course of a computation. Liu et al. identify such reasons by sets of rules that keep providing support in an ongoing computation. We have a similar

principle of persistence of reasons that is however stricter as theirs as it is operates on the atom level rather than the rule level: Once a rule instance is considered part of a computation in our sense, the truth value of the atoms in the rule's domain is frozen, i.e., it cannot be changed or forgotten in subsequent steps. Persistence of reasons is also reflected in our definition of answer sets: The requirement $I'|_{D_A} = I|_{D_A}$ in Condition ($\star$) of Definition 34 (on page 41) that the stability of interpretation I is only spoiled by I' if the reason for $I' \models A$ is the same satisfier of C-atom A as for $I \models A$.

6 Computations for Non-Ground Programs

As discussed in the introduction of Chapter 4, we use C-programs as abstraction of real-world ground answer-set programs as produced by ASP grounding tools. In this chapter, we target the remaining abstraction step for covering practical ASP languages, i.e., we present an abstraction of non-ground solver languages, and abstractions of the grounding step that turn abstract non-ground programs into C-programs. There are several reasons for keeping non-ground programs abstract for our purposes. For one, we want to support multiple solver languages that differ in various aspects, as discussed in Section 3.6, that we do not want to formalise. Examples for such differing features are different types of special literals such as aggregates or weight constraints, the handling of arithmetics, built-in functions and predicates, the use of interpreted functions, and different syntactic restrictions. Finding a single non-ground language that is not abstract and captures multiple solver languages and their semantics seems hard to accomplish and not desirable as it would have to respect a multitude of aspects inherited from the different languages. Clearly, a complicated language would impede the development of a debugging methodology. Moreover, even if such a unified non-ground language were available, the question how to handle grounding remains: using a form of naïve grounding would result in many irrelevant rules (cf. Example 2 on page 18), finite grounding may be impossible, and the transformations realised in grounders often cannot be described by a variable replacement. We next elaborate on these aspects in Section 6.1 that deals with the gap between formal answer-set programs and real world answer-set programs in solver languages and their respective groundings. We overcome these discrepancies by also making grounding abstract. That is, grounding can be seen as a black box which can be realised by actual grounding tools. Hence, we can reuse existing software and exploit the grounders' capabilities to focus on relevant information. We introduce an abstraction of non-ground programs in Section 6.2. Then, we will discuss two different abstractions of groundings. The first one, discussed in Section 6.3.1, is very general, makes minimal assumptions on the grounding procedure, and treats it as a black box. It leads to trivial generalisations of our framework of computations to the non-ground case and captures current solver languages. The other abstraction of grounding, introduced in Section 6.3.2, gives more insight into the grounding process, and formalises the idea that the grounding is influenced by a partial evaluation of the program as necessary for features like `Gringo` conditions. We also lift the results for computations for C-programs with respect to this form of grounding and discuss to which extent existing solver languages are compatible with it. The application of our computation model to stepping, respectively debugging, is discussed later in Chapter 7.

6.1 Gap between Theory and Practise: Non-Ground Programs in Solver Languages

ASP languages have been proposed as formal languages before the first answer-set solvers became available (cf. Section 3.1 on the history of ASP). While providing a clear mathematical formulation, these languages may naturally tolerate structures, like huge or infinite programs,

respectively groundings, or infinite interpretations, that cannot directly be handled by implementations.

A particular bottleneck in ASP solving is the grounding step. Modern grounding tools try to reduce the number of rules that remain in the computed grounding as much as possible while at the same time they already partially evaluate the answer-set program at hand. Thus, the ground program they produce differ from the naïve grounding as introduced in Section 3.3.2. Furthermore, the resulting grounding is often not just a subset of the naïve grounding, but contains rules that cannot be obtained from the non-ground program by rule-wise variable substitutions. For example, literals in rule bodies that are already known to be true during grounding are by default ommitted.

Example 30. Reconsider program `ex7.dlv` of Example 7 on page 26.

```
ex7.dlv                                                    DLV

bird(waldo).
bird(tux).
penguin(tux).
flies(X) :- bird(X), not -flies(X).
-flies(X) :- penguin(X).
```

The grounding step of `DLV` translates `ex7.dlv` to the ground program `ex30.dlv`.

```
ex30.dlv                                                   DLV

bird(waldo).
bird(tux).
penguin(tux).
flies(waldo).
-flies(tux).
```

By pre-evaluation, the instances of the two non-ground rules were reduced to facts. ■

A further example aspect of practical grounding that goes beyond variable substitution is caused by language features of solver languages that can be considered syntactic sugar and allow for a more compact representation, e.g., pooling or intervals.

Example 31. The following program does not contain any variables.

```
ex31a.gr                                                Gringo

myRange(1..3).
```

Nevertheless, due to the range term, the rule is transformed during grounding with `Gringo` in the following way.

```
ex31b.gr                                                Gringo

myRange(1).
myRange(2).
myRange(3).
```

■

A more severe difference of the concept of the naïve grounding and grounding perfomed by grounding tools are language features due to which the semantics of a rule in the grounding is determined by a partial evaluation of the program. We encounter this phenomenon in particular when using conditions in `Gringo`.

Example 32. Reconsider program `ex14.gr` of Example 14 on page 30.

```
ex14.gr                                                Gringo

bird(waldo).
bird(tux).
{older(X,Y),older(Y,X)}1 :- bird(X),bird(Y), X!=Y.
1{strongest(X):bird(X)}1.
{seasick(X):bird(X)}.
```

Given this program, `Gringo` produces the following output.

```
ex32a.gr                                               Gringo

bird(waldo).
bird(tux).
{older(tux,waldo),older(waldo,tux)}1.
1{strongest(tux),strongest(waldo)}1.
{seasick(tux),seasick(waldo)}.
```

If we add two further rules

```
penguin(tweety).
bird(X) :- penguin(X).
```

stating that tweety is a penguin and penguins are birds, we obtain the following `Gringo` grounding.

```
ex32b.gr                                               Gringo

bird(waldo).
bird(tux).
bird(tweety).
penguin(tweety).
{older(tweety,tux),older(tux,tweety)}1.
{older(tweety,waldo),older(waldo,tweety)}1.
{older(tux,waldo),older(waldo,tux)}1.
1{strongest(tux),strongest(tweety),strongest(waldo)}1.
{seasick(tux),seasick(tweety),seasick(waldo)}.
```

Hence, by adding information to the non-ground program, the grounding of other rules changes as the rule

```
1{strongest(tux),strongest(waldo)}1.
```

was replaced by

```
1{strongest(strongest(tux),tweety),strongest(waldo)}1.
```

and

```
{seasick(tux),seasick(waldo)}.
```

by

```
{seasick(tux),seasick(tweety),seasick(waldo)}.
```

■

6.2 An Abstraction of Non-Ground Programs

In the abstraction of non-ground programs that is introduced in the following, only little is assumed about their internal structure, essentially only that they consist of rules.

Definition 58. An *abstract non-ground rule*, or NG-*rule* for short, is an arbitrary sequence of symbols. $\Diamond$

Thus, in general, we allow for NG-rules of arbitrary shapes. For the sake of formal clarity, however, we assume that NG-rules differ from other concepts introduced in this work, in particular from C-rules. Nevertheless, we allow NG-rules to contain predicate symbols and terms.

In order to allow for C-rules to appear in our non-ground framework, we use a general notion of rule that combines NG-rules and C-rules.

Definition 59. A *general rule*, or G-*rule*, is either an NG-rule or a C-rule. A *general program*, or G-*program*, is a finite set Π of G-rules. The Herbrand Universe HU_Π of a G-program Π is given by the set of all ground functions containing only function symbols appearing in the G-rules of Π. The set of predicates appearing in Π is denoted by $\mathcal{P}(\Pi)$. $\Diamond$

6.3 Abstractions of Grounding

We present two types of abstract grounding that turn general programs into abstract-constraint programs. In both, we stick to the principle that a grounding of a program is the union of the groundings of its rules. However, they differ in the parameters that may influence the grounding of a rule. While the first type takes the whole program into account, the other might be influenced by a pre-evaluation of the program.

In ASP solver languages, there are some sets of non-ground rules that form a program which can be grounded with a grounding tool, while others cannot, e.g., due to function symbols that require infinite grounding or because domain independence properties are not met. Here, it can be the case that the same rule can be grounded in one context, while it cannot be grounded in another. In order to account for that, in both abstractions, we make use of *groundability relations* for non-ground programs, i.e., relations that determine whether a G-program can be grounded or not.

Definition 60. A *groundability relation* is a binary relation γ over the set of G-programs and $2^{\mathrm{HU}_{\mathscr{A}}}$. If $\langle \Pi, F \rangle \in \gamma$ for a G-program Π and a set F of ground terms, we call Π γ-*groundable* with respect to F. Moreover, if Π is γ-groundable with respect to HU_Π, then we say that Π is γ-*groundable*. $\Diamond$

6.3.1 Black-Box Grounding

The first type of abstraction of grounding we present is *black-box grounding*. Its main rationale is a high degree of abstraction in order to capture the behaviour of many current and probably future grounding tools. While in its original understanding a grounding only replaces variables in a non-ground rule by ground terms from a given set, the examples in Section 6.1 show that the grounding of a rule obtained by modern grounding tools is context sensitive. Hence, for black-box grounding, we propose a grounding function that takes the maximal context, i.e., the program to ground, as input for grounding a single rule.

Definition 61. Let γ be a groundability relation. An *abstract blackbox-grounding function* for γ-groundable G-programs is a mapping $gr_b(\cdot,\cdot)$ that assigns every G-rule ρ and every γ-groundable G-program Π such that $\rho \in \Pi$ a set $gr_b(\rho, \Pi) = P$ of C-rules with

(i) $D_P \subseteq \mathrm{B}^{\mathrm{HU}_\Pi}_{\mathcal{P}(\Pi)}$ and

(ii) whenever ρ is a C-rule then $gr_b(\rho, \Pi) = \{\rho\}$. $\Diamond$

Item (i) states that the domain of the grounding of a G-rule consists only of atoms constructible from predicates and terms in the program. As C-rules represent ground rules, they always coincide with their grounding, as expressed by Item (ii).

Next, we define a structure that specifies non-ground ASP languages with black-box grounding.

Definition 62. A *black-box non-ground semantics configuration* is a pair

$$\mathbb{S}_b = \langle \gamma, gr_b(\cdot, \cdot) \rangle,$$

where γ is a groundability relation and $gr_b(\cdot, \cdot)$ an abstract blackbox-grounding function for γ-groundable G-programs.

For an γ-groundable G-program Π, we call

$$gr_{b,\mathbb{S}_b}(\Pi) = \bigcup_{\rho \in \Pi} gr_b(\rho, \Pi)$$

the *grounding* of Π (with respect to $\mathbb{S}_b$). $\Diamond$

Answer sets are defined via the grounding.

Definition 63. Let $\mathbb{S}_b = \langle \gamma, gr_b(\cdot, \cdot) \rangle$ be a black-box non-ground semantics configuration and Π a G-program that is γ-groundable. An interpretation $I \subseteq \mathrm{B}^{\mathrm{HU}_\Pi}_{\mathcal{P}(\Pi)}$ is an *answer set* of Π (with respect to $\mathbb{S}_b$) if $I \in AS(gr_{b,\mathbb{S}_b}(\Pi))$. $\Diamond$

As the answer-sets of a G-program directly correspond to that of its grounding, several results for computations of C-programs carry over to G-programs as summarised below.

From now on, we assume that $\mathbb{S}_b = \langle \gamma, gr_b(\cdot, \cdot) \rangle$ is a fixed black-box non-ground semantics configuration, abbreviate $gr_{b,\mathbb{S}_c}(\Pi)$ by $gr_b(\Pi)$, and call a G-program *black-box groundable* if it is γ-groundable.

We extend the notions of Definition 54 (on page 60) to G-programs under black-box grounding as follows.

Definition 64. Let Π be a black-box groundable G-program and $C = S_0, \ldots, S_n$ a computation for $gr_b(\Pi)$. Then, C is a black-box computation for Π. Moreover,

- C has *black-box failed for* Π *at step* i if it has failed for $gr_b(\Pi)$ at step i;
- is *black-box complete for* Π if it is complete for $gr_b(\Pi)$;
- is *black-box stuck in* Π if it is stuck in $gr_b(\Pi)$;
- *black-box succeeded for* Π if it has succeeded for $gr_b(\Pi)$. $\Diamond$

The soundness result for non-ground computations using black-box grounding is obtained as a direct consequence of Theorem 14 (on page 61).

Corollary 7. *Let* Π *be a black-box groundable* G*-program and* $C = S_0, \ldots, S_n$ *a computation that has black-box succeeded for* Π. *Then,* I_{S_n} *is an answer set of* Π.

Proof. As C has black-box succeeded for Π it has succeeded for $gr_b(\Pi)$. Hence, by Theorem 14, I_{S_n} is an answer set of $gr_b(\Pi)$. Thus, by Definition 63 it is an answer set of Π. $\square$

Similarly, we can lift Theorem 15 (on page 61) as follows.

Corollary 8. *Let* Π *be a black-box groundable* G*-program and* S_0 *a state such that* $P_{S_0} \subseteq gr_b(\Pi)$*, and* I *an answer set of* Π *with* $I_{S_0} \subseteq I$ *and* $I \cap I^{-}{}_{S_0} = \emptyset$*. Then, there is a computation* $S_0, \ldots, S_n$ *that has black-box succeeded for* Π *such that* $P_{S_n} = gr_b(\Pi)^I$ *and* $I_{S_n} = I$.

Proof. As I is an answer set of Π, by Definition 63, it is an answer set of $gr_b(\Pi)$. Thus, the conjecture follows from Theorem 15. □

The following result is the non-ground variant of Corollary 4 (on page 62) under black-box grounding.

Corollary 9. *Let* Π *be a black-box groundable* G*-program and* I *an answer set of* Π*. Then, there is a computation* $S_0, \ldots, S_n$ *that has black-box succeeded for* Π *such that* $P_{S_n} = gr_b(\Pi)^{I_{S_n}}$ *and* $I_{S_n} = I$.

Proof. The claim follows immediately from Corollary 8 in case $S_0 = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. □

Next, we lift Theorem 16 on page 62 to G-programs under black-box grounding. Remember that this result provides the formal ground for the jumping technique that is introduced later in Chapter 7.

Corollary 10. *Let* Π *be a black-box groundable* G*-program,* $C = S_0, \ldots, S_n$ *a black-box computation for* Π*,* P *a set of* C*-rules with* $P \subseteq gr_b(\Pi)$*, and* I *an answer set of* $P_{S_n} \cup P$ *with* $I_{S_n} \subseteq I$ *and* $I \cap I^{-}{}_{S_n} = \emptyset$*. Then, there is a black-box computation* $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ *for* Π*, such that* S_m *is stable,* $P_{S_m} = P_{S_n} \cup P^I$ *and* $I_{S_m} = I$.

Proof. The result is a direct consequence of Theorem 16. □

Note that in this result for jumping under black-box grounding we require that $P \subseteq gr_b(\Pi)$. In the case of conditional grounding, as we also discuss in the context of Theorem 19 on page 82, I can also be an answer set of the union of P_{S_n} with a non-ground subset of Π. We cannot do this here, as it cannot be guaranteed that the respective rules have the same grounding in the context of this union as in the context of Π.

From Theorem 17 (on page 66) and Corollary 10 we get the following result guaranteeing the existence of a stable continuation of a computation.

Corollary 11. *Let* Π *be a black-box groundable* G*-program,* $C = S_0, \ldots, S_n$ *a black-box computation for* Π*, such that* S_n *is stable,* P *a set of* C*-rules with* $P \subseteq gr_b(\Pi)$*,* I *an answer set of* $P_{S_n} \cup P$ *with* $I_{S_n} \subseteq I$ *and* $I \cap I^{-}{}_{S_n} = \emptyset$ *such that* $P^I \setminus P_{S_n}$ *is a normal, convex, and absolutely tight* C*-program.*

Then, there is a black-box computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ *for* Π *where the subsequence* $S_n, \ldots, S_m$ *is a stable computation,* $P_{S_m} = P_{S_n} \cup P^I$*, and* $I_{S_m} = I$.

Proof. By Corollary 10 there is a black-box computation $C' = S_0, \ldots, S_n, S'_{n+1}, \ldots, S'_m$ for Π, such that S'_m is stable, $P_{S'_m} = P_{S_n} \cup P^I$ and $I_{S_m} = I$. By Theorem 17, since S_n and S'_m are stable and $P_\Delta = P_{S'_m} \setminus P_{S_n}$ is normal, convex, and absolutely tight, there is a stable computation $C' = S_n, \ldots, S'_m$ such that $S_m = S'_m$. Then, $C'' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ is the desired black-box computation for Π. □

For the special case where we start with the empty state, we can formulate the following result in the spirit of Corollary 9.

Corollary 12. *Let* Π *be a black-box groundable* G*-program and* I *an answer set of* Π *such that* $gr_b(\Pi)^I$ *is a normal, convex, and absolutely tight* C*-program. Then, there is a rooted stable computation* $C' = S_0, \ldots, S_n$ *that has black-box succeeded for* Π *such that* $P_{S_n} = gr_b(\Pi)^I$*, and* $I_{S_n} = I$.

Proof. The result holds by Corollary 11 for $C = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. □

6.3.2 Conditional Grounding

In our alternative abstraction of grounding that we refer to as *conditional grounding*, the grounding function does not consider the whole program for grounding each rule. Instead, the grounding of a rule may only be influenced by the set of ground terms with respect to which it is grounded (similar to how grounding was described in Section 3.3.2) and, additionally, by an interpretation that is obtained by partial evaluation of the given program. The idea is inspired by grounding tools that follow a similar principle, e.g., `Gringo` partially evaluates a program and expands constructs such as conditions according to what atoms are true in an intermediate interpretation. Computations using *black-box grounding* can be considered *static* in the sense that when extending a computation by a successor state we can consider an arbitrary rule from the overall grounding of the program. For *conditional grounding* we investigate a more *dynamic* setting. As mentioned, this type of grounding depends on an interpretation. During the evolution of a computation, at each state S, we may only add rules that are obtained by grounding a non-ground rule with respect to the state's interpretation I_S. In this sense, the grounding of a non-ground rule may change as the computation advances. The framework we introduce guarantees that only rules that appear in the overall grounding of the program can be choosen at each step and that every answer set can be reached by a so-called *settled computation* which is, roughly, a computation in which the rule added in each state is obtained as instance of a non-ground rule whose grounding will not change anymore.

While conditional grounding gives a fine-grained dynamic view on grounding and allows for exploiting non-ground ASP solving for extending a computation (see Theorem 19 on page 82 and Section 7.4 on jumping), there are a few constructs of solver languages that cannot be captured by this framework. We will discuss them at the end of the section.

Next we define groundability relations for non-ground programs under conditional grounding. As we aim for a less abstract view than in the black-box case we need to make further assumptions.

Definition 65. A *conditional groundability relation* is a groundability relation γ_c such that

- $\langle \Pi, F \rangle \in \gamma_c$ implies $\langle \Pi', F' \rangle \in \gamma_c$ for all G-programs Π and Π' with $\Pi' \subseteq \Pi$ and all sets F and F' of ground terms with $F' \subseteq F$, and
- for every C-rule r, it holds that $\langle \{r\}, F \rangle \in \gamma_c$. $\Diamond$

In the context of conditional grounding, we assume that each G-rule is associated with a set of ground atoms whose truth may influence the grounding of the rule with respect to a set of ground terms. In particular, let ρ be a G-rule and F a set of ground terms. Then, there is a set SB_ρ^F of ground atoms called the *sensitive base* of ρ with respect F such that

- $\mathrm{SB}_\rho^F = \emptyset$ if ρ is a C-rule and
- $\mathrm{SB}_\rho^{F'} \subseteq \mathrm{SB}_\rho^F$ for every $F' \subseteq F$.

Example 33. Let ρ be an abstract non-ground rule corresponding to the following `Gringo` rule from program `ex14.gr` of Example 14 on page 30:

```
1{strongest(X):bird(X)}1.
```

Then, for $F_1 = \{tux\}$ and $F_2 = \{tux, waldo\}$ we have $\mathrm{SB}_\rho^{F_1} = \{bird(tux)\}$ and $\mathrm{SB}_\rho^{F_2} = \{bird(tux), bird(waldo)\}$. ■

Next we define grounding functions for conditional grounding.

Definition 66. Let γ_c be a conditional groundability relation. An *abstract conditional grounding function* for γ_c-groundable G-programs is a mapping $gr_c(\cdot,\cdot,\cdot)$ that assigns every G-rule ρ, interpretation I, and set F of ground terms, for which $\langle\{\rho\}, F\}\rangle \in \gamma_c$ holds, a set $gr_c(\rho, I, F) = P$ of C-rules such that

(i) $D_P \subseteq \mathrm{B}_{\mathcal{P}(\{\rho\})}^{F \cup \mathrm{HU}_{\{\rho\}}}$,

(ii) whenever ρ is a C-rule then $gr_c(\rho, I, F) = \{\rho\}$, and

(iii) for every G-rule ρ, interpretations I_1, I_2, and sets F_1, F_2 of ground terms with $I_1|_{\mathrm{SB}_\rho^{F_2}} = I_2|_{\mathrm{SB}_\rho^{F_2}}$, $F_1 \subseteq F_2$, and $\langle\{\rho\}, F_2\rangle \in \gamma_c$, we have

- $gr_c(\rho, I_1, F_1) \subseteq gr_c(\rho, I_2, F_2)$, and
- $gr_c(\rho, I_1, F_1) = gr_c(\rho, I_2, F_2)$ for $F_1 = F_2$.

For a G-program Π such that $\langle \Pi, F\rangle \in \gamma_c$ holds, let $gr_c(\Pi, I, F)$ denote $\bigcup_{\rho\in\Pi} gr_c(\rho, I, F)$. $\Diamond$

Similar as in Definition 61 (on page 74) for black-box grounding, Item (i) specifies how the domain of a grounded rule may look like. Here, the domain of the grounding of a G-rule consists only of atoms constructible from predicates and ground terms in the non-ground rule and the terms in F. Moreover, also similar to the black-box case, Condition (ii) requires C-rules to always coincide with their grounding. Item (iii) expresses that only the truth of atoms in the sensitive base of a G-rule have influence on its grounding. Here, the second parameter of the grounding function, interpretation I, determines which atoms are considered true. Moreover, the condition characterises which effects their truth value may have on the grounding.

We assume the availability of a pre-evaluation function for G-programs.

Definition 67. Let γ_c be a conditional groundability relation. A *partial evaluation function* for γ_c-groundable G-programs is a function $I(\cdot)$ that assigns every γ_c-groundable G-program Π an interpretation $I(\Pi)$. $\Diamond$

Intuitively, the result $I(\Pi)$ of applying a pre-evaluation function on a G-program Π is an interpretation that fully determines the truth of the atoms appearing in the sensitive base of each rule from Π and hence determines how the program is grounded. We express that the truth values of the atoms in the sensitive base of a G-rule under some interpretation corresponds to those under $I(\Pi)$ in the notion of *settledness*.

Definition 68. Let γ_c be a conditional groundability relation, $I(\cdot)$ a partial evaluation function for γ_c-groundable G-programs, and Π an γ_c-groundable G-program.

A G-rule ρ is *settled* in Π under interpretation I (and $I(\cdot)$-pre-evaluation) if $\rho \in \Pi$ and $I|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}$.

Let $gr_c(\cdot,\cdot,\cdot)$ be an abstract conditional grounding function for γ_c-groundable G-programs. Then, we call a rooted computation $S_0, \ldots, S_n$ for $gr_c(\Pi, I(\Pi), \mathrm{HU}_\Pi)$ *settled* in Π (under $I(\cdot)$-pre-evaluation) if for every $0 \leq i < n$, there is some G-rule ρ that is settled in Π under I_{S_i} and $I(\cdot)$-pre-evaluation such that $r_{new}(S_i, S_{i+1}) \in gr_c(\rho, I(\Pi), \mathrm{HU}_\Pi)$. $\Diamond$

The intuition of a settled computation is that in each state only rules are added that are obtained from grounding a non-ground rule ρ that is settled under the previous state's interpretation. The aim is to ensure that only rules are added in a computation that belong to the overall grounding of the program. To guarantee that, we need to make further assumptions in the next definition that characterises non-ground ASP languages with conditional grounding.

Definition 69. A *conditional non-ground semantics configuration* is a triple

$$\mathbb{S}_c = \langle \gamma_c, gr_c(\cdot,\cdot,\cdot), I(\cdot)\rangle,$$

where γ_c is a conditional groundability relation, $gr_c(\cdot,\cdot,\cdot)$ an abstract conditional grounding function for γ_c-groundable G-programs, and $I(\cdot)$ a partial evaluation function for γ_c-groundable G-programs such that for every γ_c-groundable G-program Π

(i) there is no rooted computation $S_0,\dots,S_n$ for $gr_c(\Pi, I(\Pi), \mathrm{HU}_\Pi)$ such that for some $\rho \in \Pi$ it holds that $I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subset I_{S_n}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}$ and

(ii) there exists a rooted computation $S_0,\dots,S_n$ settled in Π with

- $I_{S_n} = I(\Pi)$,
- each C-literal $L \in \mathrm{B}(r)$ for $r \in P_{S_n}$ is convex, and
- for every rooted computation $S'_0,\dots,S'_m$ that is complete for $gr_c(\Pi, I(\Pi), \mathrm{HU}_\Pi)$, there is a computation $S_0,\dots,S_n,S_{n+1},\dots,S_m$ such that $S_m = S'_m$ and for indices $n{+}1 \le i \le m, n{+}1 \le j \le m, 1 \le i' \le m$, and $1 \le j' \le m$ with $r_{new}(S_{i-1},S_i) = r_{new}(S'_{i'-1},S'_{i'})$ and $r_{new}(S_{j-1},S_j) = r_{new}(S'_{j'-1},S'_{j'})$, we have that $i < j$ iff $i' < j'$.

We call $gr_{c,\mathbb{S}_c}(\Pi) = gr_c(\Pi, I(\Pi), \mathrm{HU}_\Pi)$ the *grounding* of Π (with respect to $\mathbb{S}_c$). $\Diamond$

Condition (i) ensures that the interpretation $I(\Pi)$ obtained by pre-evaluation contains all atoms possibly influencing the grounding that can be derived from the rules in the grounding of Π. Item (ii) ensures that there is a computation C settled in Π for computing the atoms in $I(\Pi)$. The intuition is that $C = S_0,\dots,S_n$ represents the pre-evaluation. The final condition of Item (ii) states that from every complete rooted computation C' in the grounding one can obtain another complete rooted computation C' with the same final state by evaluating the rules in P_{S_n} first, i.e., C'' has C as prefix and the following states introduce the remaining rules in the same order as in C'. Summarising, Item (ii) expresses that $I(\Pi)$ can always be computed in the beginning of a computation.

The answer sets of a G-program using conditional grounding are defined via the grounding, analogously to the case of black-box grounding.

Definition 70. Let $\mathbb{S}_c = \langle \gamma_c, gr_c(\cdot,\cdot,\cdot), I(\cdot)\rangle$ be a conditional non-ground semantics configuration and Π be a G-program that is γ_c-groundable. An interpretation $I \subseteq \mathrm{B}_{\mathcal{P}(\Pi)}^{\mathrm{HU}_\Pi}$ is an *answer set* of Π with respect to $\mathbb{S}_c$ if $I \in AS(gr_{c,\mathbb{S}_c}(\Pi))$. $\Diamond$

For the remainder of the chapter we assume an implicit conditional non-ground semantics configuration $\mathbb{S}_c = \langle \gamma_c, gr_c(\cdot,\cdot,\cdot), I(\cdot)\rangle$, abbreviate $gr_{c,\mathbb{S}_c}(\Pi)$ by $gr_c(\Pi)$, call a G-program conditionally groundable if it is γ_c-groundable, and denote the set of all answer sets of a G-program Π with respect to $\mathbb{S}_c$ by $AS_c(\Pi)$.

We also define the notions of Definition 54 (on page 60) for G-programs under conditional grounding. Unlike the case of black-box grounding here we require computations for a G-program to be settled.

Definition 71. Let Π be a conditionally groundable G-program and $C = S_0,\dots,S_n$ a computation settled in Π. Then, C is a *conditional computation* for Π. Moreover,

- C has *conditionally failed for* Π *at step* i if it has failed for $gr_c(\Pi)$ at step i;
- is *conditionally complete for* Π if it is complete for $gr_c(\Pi)$;
- is *conditionally stuck in* Π if it is stuck in $gr_c(\Pi)$;

- *conditionally succeeded for* Π if it has succeeded for $gr_c(\Pi)$. ◊

The next result makes explicit that the pre-evaluated interpretation is part of every answer set and determines the truth values for the sensitive bases of the rules of a G-program.

Proposition 4. *Let* Π *be a conditionally groundable* G*-program and* $I \in AS_c(\Pi)$*. Then,* $I(\Pi) \subseteq I$ *and for each* $\rho \in \Pi$ *it holds that* $I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}$.

Proof. As $I \in AS(gr_c(\Pi))$, by Corollary 4 (on page 62), there is a rooted computation $C' = S'_0, \ldots, S'_m$ that is complete for $gr_c(\Pi)$ such that $I_{S'_m} = I$. By Condition (ii) of Definition 69, there is a rooted computation $C = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ with $S_m = S'_m$ such that $I_{S_n} = I(\Pi)$. As $I_{S_n} \subseteq I_{S_m}$, we have $I(\Pi) \subseteq I$.

Consider some $\rho \in \Pi$. It must hold that

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subseteq I|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

By Condition (i) of Definition 69, as C is a rooted computation for $gr_c(\Pi)$, it cannot hold that

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subset I|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

Consequently,

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$ □

Also for conditional grounding, a soundness result follows as direct consequence of Theorem 14 (on page 61).

Corollary 13. *Let* Π *be a conditionally groundable* G*-program and* $C = S_0, \ldots, S_n$ *a computation that has succeeded for* $gr_c(\Pi)$*. Then,* I_{S_n} *is an answer set of* Π.

Proof. As C has succeeded for $gr_c(\Pi)$, by Theorem 14, I_{S_n} is an answer set of $gr_c(\Pi)$. Thus, by Definition 70 it is an answer set of Π. □

As we target settled computations, unlike Corollary 8 (on page 76), the result establishing the existence of a computation for every answer set takes not an arbitrary state of the grounding as given but a settled computation C. We show that C can be expanded to reach the given answer set.

Theorem 18. *Let* Π *be a conditionally groundable* G*-program,* $C = S_0, \ldots, S_n$ *a conditional computation for* Π*, and* I *an answer set of* Π *with* $I_{S_n} \subseteq I$ *and* $I \cap I^{-}{}_{S_n} = \emptyset$*. Then, there is a computation* $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ *that has conditionally succeeded for* Π *where* $P_{S_m} = gr_c(\Pi)^I$ *and* $I_{S_m} = I$.

Proof. By Theorem 15 (on page 61), there is a computation $C'' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ that is complete for $gr_c(\Pi)$ such that S_m is stable, $P_{S_m} = gr_c(\Pi)^I$, and $I_{S_m} = I$. As C'' is rooted, by Item (ii) of Definition 69, there is a computation $C''' = S'''_0, \ldots, S'''_k, S'''_{k+1}, \ldots, S'''_m$ such that $S_m = S'''_m$ and

(†) for indices $1 \leq j \leq m$, $1 \leq e \leq m$, $k+1 \leq j' \leq m$, and $k+1 \leq e' \leq m$ with $r_{new}(S_{j-1}, S_j) = r_{new}(S'''_{j'-1}, S'''_{j'})$ and $r_{new}(S_{e-1}, S_e) = r_{new}(S'''_{e'-1}, S'''_{e'})$, we have that $j < e$ iff $j' < e'$, where $S'''_0, \ldots, S'''_k$ is a rooted computation settled in Π with $I_{S'''_k} = I(\Pi)$ and each C-literal $L \in \mathrm{B}(r)$ for $r \in P_{S'''_k}$ is convex.

Next, we construct a computation C' that has C as prefix and continues by adding C-rules according to the order in C'''. Consider the sequence of C-rules

$$R''' = r_{new}(S'''_0, S'''_1), \dots, r_{new}(S'''_m - 1, S'''_m)$$

and for simplicity let us write $R''' = r'''_1, \dots, r'''_m$. Moreover, let $R' = r'_1, \dots, r'_{m-n}$ denote the sequence of C-rules obtained from R''' by removing all C-rules occurring in P_{S_n}. Consider the sequence $C' = S'_0, \dots, S'_n, S'_{n+1}, \dots, S'_m$ of state structures, where $S'_i = S_i$ for $1 \le i \le n$ and, for $n < i \le m$,

$$S'_i = \langle P_{S'_{i-1}} \cup \{r'_{i-n}\}, I_{S'_{i-1}} \cup (I|_{D_{r'_{i-n}}}), I^-{}_{S'_{i-1}} \cup ((D_{grc(\Pi)^I} \setminus I)|_{D_{r'_{i-n}}}), \Upsilon\rangle$$

where $X' \in \Upsilon$ iff $X' = X \cup \Delta'$, $X \in \Upsilon_{S'_{i-1}}$, $\Delta \subseteq I_{S'_i} \setminus I_{S'_{i-1}}$, and r'_{i-n} is not an external support for X' with respect to $I_{S'_i}$. We will show that C' is the targeted computation.

First, we show that C' is a computation. Towards a contradiction, assume that for some $1 \le i \le m$, S'_i is not a successor of S'_{i-1}. As $S'_0, \dots, S'_n$ is a computation it must hold that $n < i$. Let r denote the C-rule $r_{new}(S'_{i-1}, S'_i)$, for which we have $r = r'_{i-n}$ and $r = r'''_{e'}$ for some $1 \le e' \le m$. By construction of C' and since $r \in P_{S_m}$, the Conditions (i), (ii), (iii), (v), and (vi) of Definition 52 on page 58 for being a successor of S'_{i-1} are fulfilled by S'_i. It must hold that Condition (iv) is violated, hence $I_{S'_{i-1}} \not\models \mathrm{B}(r)$. Then, since $S'''_{e'}$ is a successor of $S'''_{e'-1}$, we know that $I_{S'''_{e'-1}} \models \mathrm{B}(r)$. Hence, there must be some $L \in \mathrm{B}(r)$ such that

$$I_{S'''_{e'-1}} \models L \quad \text{and} \quad I_{S'_{i-1}} \not\models L.$$

Note that, as $P_{S'''_{e'-1}} \subseteq P_{S'_{i-1}}$, by construction of C' we have that

$$I_{S'''_{e'-1}} \subseteq I_{S'_{i-1}}.$$

Consider the case that $r \in P_{S'''_k}$. Then, L must be convex. Note that $I \models L$. This is a contradiction to convexity of L, as we have

$$I_{S'''_{e'-1}} \subset I_{S'_{i-1}} \subseteq I.$$

Now, consider the case that $r \notin P_{S'''_k}$. Then, we have $k < e' \le m$. Since

$$I_{S'_{i-1}} \not\models L, \quad I_{S'''_{e'-1}} \models \mathrm{B}(r), \quad \text{and} \quad I_{S'''_{e'-1}} \subseteq I_{S'_{i-1}},$$

there must be some atom

$$a \in I_{S'_{i-1}} \setminus I_{S'''_{e'-1}}$$

such that $a \in D_L$. Let h be the smallest index $1 \le h \le i-1$ such that

$$a \in I_{S'_h}.$$

It must hold that $a \in D_{r'}$ for $r' = r_{new}(S'_{h-1}, S'_h)$. Consider the case that $r' \in P_{S'''_{e'-1}}$. Then, we have a contradiction to

$$a \notin I_{S'''_{e'-1}}.$$

There must be some index j' such that $r' = r'''_{j'}$ with $e' \le j' \le m$. Moreover, as

$$r' \in P_{S'_{i-1}} \setminus P_{S'''_{e'-1}},$$

it must hold that $r' \in P_{S_n}$. Let j be the index such that $r' = r_{new}(S_{j-1}, S_j)$ with $1 \le j \le n$ and e the index such that $r = r_{new}(S_{e-1}, S_e)$. As $r \notin P_{S_n}$, we have $n < e$ and consequently

$j < e$. As $r_{new}(S_{j-1}, S_j) = r_{new}(S'''_{j'-1}, S'''_{j'})$ and $r_{new}(S_{e-1}, S_e) = r_{new}(S'''_{e'-1}, S'''_{e'})$, by (†) we get that $j' < e'$, being a contradiction to $e' \leq j'$. It follows that C' is a computation.

Towards a contradiction assume C' is not settled in Π. Clearly, C' is a rooted computation for $gr_c(\Pi)$. Let i be the smallest index with $0 \leq i < m$ such that there is no G-rule ρ that is settled in Π under $I_{S'_i}$ such that $r = r_{new}(S'_i, S'_{i+1}) \in gr_c(\rho, I(\Pi), \mathrm{HU}_\Pi)$. Note that, since C is settled in Π and C is a prefix of C', we have that $n \leq i$.

Consider the case that $r \in P_{S'''_k}$. Then, we have $r = r'''_{i'}$ for some $1 \leq i' \leq k$. As $S'''_0, \ldots, S'''_k$ is settled in Π, we know that there is some ρ that is settled in Π under $I_{S'''_{i'-1}}$ such that $r \in gr_c(\rho, I(\Pi), \mathrm{HU}_\Pi)$. Consequently, $\rho \in \Pi$ and

$$I_{S'''_{i'-1}}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

As $P_{S'''_{i'-1}} \subseteq P_{S'_i}$ and by construction of C' we have that $I_{S'''_{i'-1}} \subseteq I_{S'_i}$. Therefore,

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subseteq I_{S'_i}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

By Definition 69, as $S_0, \ldots, S'_i$ is a rooted computation for $gr_c(\Pi)$ it cannot hold that

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subset I_{S'_i}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

It must hold that

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I_{S'_i}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

This is a contradiction to ρ not being settled in Π under $I_{S'_i}$.

Finally, consider the case that $r \notin P_{S'''_k}$. Consequently, we have $k \leq i$. As $r \in gr_c(\Pi)$, there must be some $\rho \in \Pi$ with $r \in gr_c(\rho, I(\Pi), \mathrm{HU}_\Pi)$. As ρ must not be settled in Π under $I_{S'_i}$ we get

$$I_{S'_i}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \neq I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

From that, and since $I_{S'_k} = I(\Pi)$ implies $I(\Pi) \subseteq I_{S'_i}$, we get

$$I_{S'_i}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subset I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

As $S'_0, \ldots, S'_i$ is a rooted computation for $gr_c(\Pi)$, this is a contradiction to Condition (i) of Definition 69. □

Without a given prefix computation we get the following variant of Corollary 4 (on page 62) for the conditional grounding setting.

Corollary 14. *Let* Π *be a conditionally groundable* G*-program and* I *an answer set of* Π*. Then, there is a computation* $S_0, \ldots, S_n$ *that has conditionally succeeded for* Π *where* $P_{S_n} = gr_c(\Pi)^{I_{S_n}}$ *and* $I_{S_n} = I$*.*

Proof. The conjecture follows from Theorem 18 in case $C = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. □

Next, we give our result for the jumping technique under conditional grounding (cf. Section 7.4). Unlike in Corollary 10 (on page 76) for black-box grounding, based on this result, one can join the rules that are already considered in a computation with a non-ground subset of the overall program and use an ASP solver to obtain an extended computation from that.

Theorem 19. *Let* Π *be a conditionally groundable* G*-program,* $C = S_0, \ldots, S_n$ *a conditional computation for* Π*,* Π' *a set of* G*-rules settled in* Π *with respect to* I_{S_n}*, and* $I \in AS_c(P_{S_n} \cup \Pi')$ *with* $I_{S_n} \subseteq I$ *and* $I \cap I^-{}_{S_n} = \emptyset$*. Then, there is a conditional computation* $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ *for* Π *such that* S_m *is stable,* $P_{S_m} = gr_c(P_{S_n} \cup \Pi')^I$*, and* $I_{S_m} = I$*.*

Proof. At first, we show that C is settled in $\Pi'' = P_{S_n} \cup \Pi'$. Towards a contradiction, assume it is not and let i be the smallest index with $0 \leq i < n$ such that there is no G-rule ρ settled in Π'' under I_{S_i} with $r \in gr_c(\rho, I(\Pi''), \mathrm{HU}_{\Pi''})$ for $r = r_{new}(S_i, S_{i+1})$. As $r \in P_{S_n}$ and r is a C-rule, we also have $r \in \Pi''$ and $r \in gr_c(r, I(\Pi''), \mathrm{HU}_{\Pi''})$. Then,

$$I_{S_i}|_{\mathrm{SB}_r^{\mathrm{HU}_{\Pi''}}} \neq I(\Pi'')|_{\mathrm{SB}_r^{\mathrm{HU}_{\Pi''}}}$$

because r is not settled in Π'' under I_{S_i}. This is a contradiction, as r being a C-rule implies $\mathrm{SB}_r^{\mathrm{HU}_{\Pi''}} = \emptyset$. It follows that C is settled in Π''. Therefore, by Theorem 18, as Π'' is a conditionally groundable G-program, there is a computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ settled in Π'' such that S_m is stable, $P_{S_m} = gr_c(\Pi)^I$, and $I_{S_m} = I$. It remains to be shown that C' is settled in Π. Towards a contradiction, assume it is not and let j be the smallest index with $0 \leq j < m$ such that there is no G-rule ρ'' settled in Π under I_{S_j} with $r' \in gr_c(\rho'', I(\Pi), \mathrm{HU}_\Pi)$ for $r' = r_{new}(S_j, S_{j+1})$. As C is settled in Π and C is a prefix of C', we have that $n \leq j$. Therefore, it must hold that $r' \notin P_{S_n}$. As C' is settled in Π'', we have that $r' \in gr_c(\rho', I(\Pi''), \mathrm{HU}_{\Pi''})$ for some $\rho' \in \Pi'$ where ρ' is settled in Π'' under I_{S_j} and in Π under I_{S_n}. We next show that

$$I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} \subseteq I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}.$$

Consider some $a \in I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}$. As ρ' is settled in Π under I_{S_n}, we have

$$I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} = I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}.$$

Therefore, it holds that $a \in I_{S_n}$. Then, since C' is a computation and $n \leq j$ we have that $a \in I_{S_j}$. Therefore,

$$a \in I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} \quad \text{and} \quad I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} \subseteq I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}.$$

Consider the case that $r' \in gr_c(\rho', I(\Pi), \mathrm{HU}_\Pi)$. Then, ρ' must not be settled in Π under I_{S_j}. Hence, we have $I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} \neq I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}$ and therefore

$$I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} \subset I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}.$$

Now consider the case that $r' \notin gr_c(\rho', I(\Pi), \mathrm{HU}_\Pi)$. By Condition (iii) of Definition 66 (on page 78), since $\mathrm{HU}_{\Pi''} \subseteq \mathrm{HU}_\Pi$, we get that $r' \notin gr_c(\rho', I(\Pi), \mathrm{HU}_{\Pi''})$. From that and since $r' \in gr_c(\rho', I(\Pi''), \mathrm{HU}_{\Pi''})$, again by Condition (iii), it must hold that

$$I(\Pi'')|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} \neq I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}.$$

As $I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}} = I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_\Pi}}$ and $\mathrm{HU}_{\Pi''} \subseteq \mathrm{HU}_\Pi$, by the definition of a sensitive base, also

$$I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} = I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}.$$

It follows that $I(\Pi'')|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} \neq I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}$. As ρ' is settled in Π'' under I_{S_j} it holds that

$$I(\Pi'')|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} = I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}},$$

which implies $I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} \neq I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}$. Hence, since $I_{S_n} \subseteq I_{S_j}$, we get

$$I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} \subset I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}.$$

Consider some $b \in I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}$ such that $b \notin I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}$. As

$$I_{S_n}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}} = I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}},$$

we get $b \notin I(\Pi)$. Thus, it holds that $b \notin I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi}}}$. Moreover, from $b \in I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi''}}}$, we get

$$b \in I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi}}}$$

by the definition of a sensitive base because $\mathrm{HU}_{\Pi''} \subseteq \mathrm{HU}_{\Pi}$. Therefore, as

$$I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi}}} \subseteq I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi}}}$$

also in this case we get

$$I(\Pi)|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi}}} \subset I_{S_j}|_{\mathrm{SB}_{\rho'}^{\mathrm{HU}_{\Pi}}}.$$

This is a contradiction by Condition (i) of Definition 69, as $\rho' \in \Pi$ and $S_0, \ldots, S_j$ is a rooted computation for $gr_c(\Pi)$. □

We can also extend our results for the existence of stable computations to G-programs under conditional grounding.

Theorem 20. *Let* Π *be a conditionally groundable* G*-program,* $C = S_0, \ldots, S_n$ *a conditional computation for* Π *such that* S_n *is stable,* Π' *a set of* G*-rules settled in* Π *with respect to* I_{S_n}*, and* $I \in AS_c(P_{S_n} \cup \Pi')$ *with* $I_{S_n} \subseteq I$ *and* $I \cap I^-{}_{S_n} = \emptyset$ *such that* $P \setminus P_{S_n}$ *for* $P = gr_c(P_{S_n} \cup \Pi')^I$ *is a normal, convex, and absolutely tight* C*-program.*

Then, there is a conditional computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ *for* Π *such that* S_i *is stable for all* $n \leq i \leq m$*,* $P_{S_m} = gr_c(P_{S_n} \cup \Pi')^I$*, and* $I_{S_m} = I$*.*

Proof. By Theorem 19, there is a computation $C'' = S_0, \ldots, S_n, S'_{n+1}, \ldots, S'_m$ settled in Π such that S'_m is stable,

$$P_{S'_m} = gr_c(\Pi'')^I$$

for $\Pi'' = P_{S_n} \cup \Pi'$, and $I_{S'_m} = I$. By Theorem 17 (on page 66), there is a computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ such that $S_m = S'_m$ and S_i is stable for all $n \leq i \leq m$. Note that, since C'' is settled in Π, we have $P_{S_m} \subseteq gr_c(\Pi)$ and consequently C' is a computation for $gr_c(\Pi)$. It remains to be shown that C' is settled in Π. Let i be the smallest index with $0 \leq i < m$ such that there is no G-rule ρ' that is settled in Π under I_{S_i} such that $r = r_{new}(S_i, S_{i+1}) \in gr_c(\rho', I(\Pi), \mathrm{HU}_{\Pi})$. Note that, since C is settled in Π and C is a prefix of C', we have that $n \leq i$. From that we get that $r \notin P_{S_n}$ and consequently, since $r \in gr_c(\Pi'')$ it must hold that there is some $\rho \in \Pi'$ with $r \in gr_c(\rho, I(\Pi''), \mathrm{HU}_{\Pi''})$. As ρ is settled in Π under I_{S_n} we have $I(\Pi)|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi}}} = I_{S_n}|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi}}}$. From that and $I_{S_n} \subseteq I_{S_i}$ we get

$$I(\Pi)|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi}}} \subseteq I_{S_i}|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi}}}.$$

As $S_0, \ldots, S_n, S_{n+1}, \ldots, S_i$ is a rooted computation for $gr_c(\Pi)$, we get by Condition (i) of Definition 69 (on page 79) that

$$I(\Pi)|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi}}} = I_{S_i}|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi}}}.$$

It follows that ρ is settled in Π under I_{S_i}. As a consequence, as noted above, it must hold that $r \notin gr_c(\rho, I(\Pi), \mathrm{HU}_{\Pi})$. Hence, by Condition (iii) of Definition 66 (on page 78), as $\mathrm{HU}_{\Pi''} \subseteq \mathrm{HU}_{\Pi}$, also $r \notin gr_c(\rho, I(\Pi), \mathrm{HU}_{\Pi''})$. By the same condition and since $r \in gr_c(\rho, I(\Pi''), \mathrm{HU}_{\Pi''})$, we get that $I(\Pi)|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi''}}} \neq I(\Pi'')|_{\mathrm{SB}_{\rho}^{\mathrm{HU}_{\Pi''}}}$.

Assume there is some

$$a \in I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_{\Pi''}}}$$

such that $a \notin I(\Pi'')$. From $a \in \mathrm{SB}_\rho^{\mathrm{HU}_{\Pi''}}$ we get by the definition of a sensitive base that $a \in \mathrm{SB}_\rho^{\mathrm{HU}_\Pi}$. Therefore, as $a \in I(\Pi)$ and

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I_{S_n}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}},$$

we get $a \in I_{S_n}$. As $\rho \in \Pi''$, $a \in \mathrm{SB}_\rho^{\mathrm{HU}_{\Pi''}}$, $a \notin I(\Pi'')$, and $I_{S_m} \in AS_c(\Pi'')$, we get by Proposition 4 that $a \notin I_{S_m}$. This is a contradiction to $a \in I_{S_n}$ as $I_{S_n} \subseteq I_{S_m}$. Consequently, there must be some $b \in I(\Pi'')|_{\mathrm{SB}_\rho^{\mathrm{HU}_{\Pi''}}}$ such that $b \notin I(\Pi)$.

From $b \in \mathrm{SB}_\rho^{\mathrm{HU}_{\Pi''}}$ we get by the definition of a sensitive base that $b \in \mathrm{SB}_\rho^{\mathrm{HU}_\Pi}$. Moreover, as $b \in I(\Pi'')$ and $I_{S'_m} \in AS_c(\Pi'')$, by Proposition 4 it holds that $b \in I_{S'_m}$. As C' is a rooted computation for $gr_c(\Pi)$, by Condition (i) of Definition 69, it cannot hold that

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subset I_{S'_m}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

As $I_{S_n} \subseteq I_{S'_m}$ and $I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I_{S_n}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}$, we get

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} \subseteq I_{S'_m}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

It follows that

$$I(\Pi)|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}} = I_{S'_m}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}.$$

As $b \in I_{S'_m}|_{\mathrm{SB}_\rho^{\mathrm{HU}_\Pi}}$ this is a contradiction to $b \notin I(\Pi)$. □

Without a given prefix computation we get the following completeness result as a corollary.

Corollary 15. *Let* Π *be a conditionally groundable* G*-program such that* $gr_c(\Pi)^I$ *is a normal, convex, and absolutely tight* C*-program and* I *an answer set of* Π*. Then, there is a stable computation* $C' = S_0, \ldots, S_n$ *that has conditionally succeeded for* Π *such that* $P_{S_m} = gr_c(P_{S_n} \cup \Pi')^I$ *and* $I_{S_m} = I$*.*

Proof. The result holds by Theorem 20 for $C = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. □

Less abstract than black-box grounding, the framework for conditional grounding is still quite flexible and can be seen as an abstraction for most features of ASP solver languages. Yet, there exist programs in current solver languages that it cannot capture as illustrated in the next example. However, in these cases, black-box grounding would be a safe fallback option.

Example 34. Consider the following program that guesses for every bird whether it flies or not and derives the atom `multipleFliers` in the case that there are at least two birds that fly.

```
ex34a.dlv                                             DLV

bird(tux) v bird(tweety).
bird(waldo).
flies(X) v -flies(X) :- bird(X).
multipleFliers :- 2 <= #count{X:flies(X)}.
```

Grounding by `DLV` produces the following output.

```
ex34b.dlv                                                    DLV

bird(tux) v bird(tweety).
bird(waldo).
flies(waldo) v -flies(waldo).
flies(tux) v -flies(tux) :- bird(tux).
flies(tweety) v -flies(tweety) :- bird(tweety).
multipleFliers :- 2 <= #count{<tux:flies(tux)>,
                       <tweety:flies(tweety)>,
                       <waldo:flies(waldo)>} <= 2147483647.
```

The grounding of the aggregate in the rule deriving `multipleFliers` clearly depends on the first two rules in the program that define the `bird` predicate although the rule does not involve this predicate itself. In fact, `DLV` has to compute which terms could be relevant for the local grounding of the rule based on the remainder of the program. Moreover, the grounding of the rule does not only depend on atoms that must be true in every answer set as there are answer sets where `bird(tux)` (or `bird(tweety)`) is true and answer sets where it is false. Hence, a G-program that corresponds to `ex34a.dlv` would not be conditionally groundable.

Also `Gringo` offers similar functionality. In particular it allows for non-ground atoms to appear in a weight or cardinality constraint without using conditions. Similar to `DLV`, a local grounding is performed, where the ground terms used are computed from the remaining program. The following program is a `Gringo` variant of the `DLV` example.

```
ex34c.gr                                                  Gringo

1{bird(tux),bird(tweety)}1.
bird(waldo).
{flies(X)} :- bird(X).
multipleFliers :- 2{flies(X)}.
```

The resulting `Gringo` grounding is the following program.

```
ex34d.gr                                                  Gringo

1{bird(tux),bird(tweety)}1.
bird(waldo).
{flies(tweety)} :- bird(tweety).
{flies(tux)} :- bird(tux).
{flies(waldo)}.
multipleFliers :- 2{flies(waldo),flies(tux),
                                        flies(tweety)}.
```

The same observations as for `DLV` apply. Note that the rules

```
multipleFliers :- 2{flies(X):bird(X)}
```

or

```
multipleFliers :- 2{flies(X):flies(X)}
```

cannot be used as replacement of the final rule in program `ex34b.gr` as `Gringo` requires predicates in conditions to be stratified. This restriction corresponds to the requirement that atoms that influence grounding can be pre-evaluated deterministically in our conditional grounding framework. ■

7 Stepping Answer-Set Programs

We develop a methodology for stepping answer-set programs based on the computation model introduced in the previous chapter. Its main application is debugging but it is also beneficial in other contexts as it may improve the understanding of a given answer-set program and can help to improve the understanding of the answer-set semantics for beginners.

Step-by-step execution of a program is common practise in procedural programming languages, where developers can debug and investigate the behaviour of their programs in an incremental way. The stepping technique introduced in this work shows how this popular form of debugging can be applied to ASP, despite the genuine declarative semantics of answer-set programs that lacks a control flow. Moreover, it meets the requirements for a debugging approach as sketched in Section 2.4. Note that we have published previous versions of the stepping technique for normal logic programs (Oetsch et al., 2010b, 2011c) and DL-programs (Oetsch et al., 2012c) that are subsumed by the framework of this thesis.

Stepping is intended to be a practical support technique for answer-set programmers rather than a purely theoretical approach. As a consequence, we assume the *availability of a support environment* that assists a user in a stepping session. In this chapter, we sometimes refer to potential features of such a system. In Chapter 8, we describe the `SeaLion` IDE that has been developed in the context of this thesis and implements a prototype of a stepping support environment.

In the following section, we introduce two example problems that we make use of in the remainder of the thesis. In Section 7.2, we describe the general idea of stepping for ASP. There are two major ways for navigating in a computation in our framework: performing *steps*, discussed in Section 7.3, and *jumps* that we describe in Section 7.4. Building on these techniques, we discuss methodological aspects of stepping for debugging purposes in Section 7.5 including development guidelines for effective debugging. A number of debugging scenarios are presented in Section 7.6. Section 7.7 concludes the chapter with general guidelines for ASP development that also ease the use of stepping-based debugging.

7.1 Example Problems

Next, we introduce two problems related with grid puzzles that serve as running examples in this and the following chapter. The first problem is *maze generation*, a problem where deciding the existence of a solution is in NP. The other problem deals with a task in the setting of the game *minesweeper* where the corresponding decision problem is Σ_2^{P}-hard.

7.1.1 Maze Generation Problem

The maze generation problem that we deal with has been a benchmark problem of the second ASP competition (Denecker et al., 2009) to which it was submitted by Martin Brain. The original problem description is available on the competition's website:

```
http://dtai.cs.kuleuven.be/events/ASP-competition/
Benchmarks/MazeGeneration.shtml
```

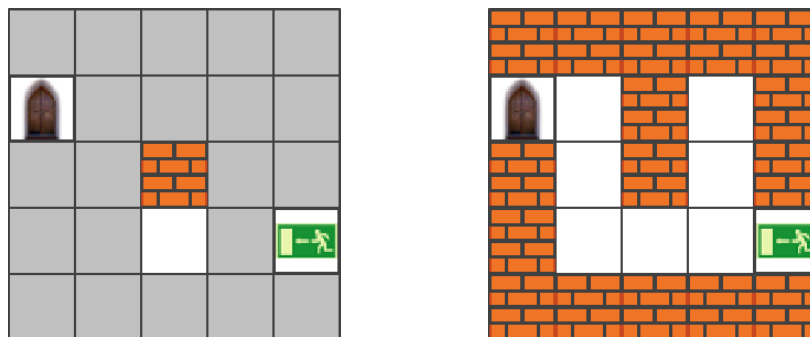

Figure 7.1: Left: A grid visualising an instance of the maze generation where white squares represent empty cells, whereas grey squares are yet undefined. Right: A solution for the instance on the left.

As the name of the problem indicates, the task we deal with is to generate a maze, i.e., a labyrinth structure in a grid that satisfies certain conditions. In particular, we deal with two-dimensional grids of cells where each cell can be assigned to be either an empty space or a wall. Moreover, there are two (distinct) empty squares on the edge of the grid, known as the entrance and the exit. A path is a finite sequence of cells, in which each distinct cell appears at most once and each cell is horizontally or vertically adjacent to the next cell in the sequence.

Such a grid is a valid maze if it meets the following criteria:

1. Each cell is a wall or is empty.
2. There must be a path from the entrance to every empty cell (including the exit).
3. If a cell is on any of the edges of the grid, and is not an entrance or an exit, it must contain a wall.
4. There must be no 2x2 blocks of empty cells or walls.
5. No wall can be completely surrounded by empty cells.
6. If two walls are diagonally adjacent then one or other of their common neighbours must be a wall.

The maze generation problem is the problem of completing a two-dimensional grid in which some cells are already decided to be empty or walls and the entrance and the exit are pre-defined to a valid maze. An example of a problem instance and a corresponding solution maze is depicted in Figure 7.1.

Next we describe the predicate schema that we use for ASP maze generation encodings. The predicates `col/1` and `row/1` define the columns and rows in the grid, respectively. They are represented by a range of consecutive, ascending integers, starting at `1`. The positions of the entrance and the exit are determined by predicates `entrance/2` and `exit/2`, respectively, where the first argument is a column index and the second argument is a row index. In a similar manner, `empty/2` and `wall/2` determine which cells are empty or contain walls. For example, the instance of Figure 7.1 can be encoded by the following facts:

```
exMazeInstance.gr                                    Gringo

col(1..5). row(1..5).
entrance(1,2). exit(5,4). wall(3,3). empty(3,4).
```

Moreover, the solution in the figure could be represented by the following interpretation:

Figure 7.2: Minesweeper configurations where the player has to guess.

Figure 7.3: Loosing a tricky Minesweeper game due to bad luck: In the first picture the player has (correctly) identified all but one mines and marked them with a flag. Only two covered cells remain - the chances to select the free cell are 1:1. The player chooses to uncover the upper right cell (highlighted with a red box). The right picture shows the result—unfortunately the cell contained a mine.

```
{wall(1,1), empty(1,2), wall(1,3), wall(1,4), wall(1,5),
 wall(2,1), empty(2,2), empty(2,3), empty(2,4), wall(2,5),
 wall(3,1), wall(3,2), wall(3,3), empty(3,4), wall(3,5),
 wall(4,1), empty(4,2), empty(4,3), empty(4,4), wall(4,5),
 wall(5,1), wall(5,2), wall(5,3), empty(5,4), wall(5,5)}
```

7.1.2 Fair Minesweeper

Minesweeper is a well-known computer game where the player has to uncover all cells in a two-dimensional grid of initially covered cells that do not contain mines. If the user uncovers a cell with a mine the game is over and the user has lost. The total number of mines is known. Moreover, each uncovered cell that does not contain mines shows the number of neighbouring cells that do contain mines, if any. Most implementations of minesweeper randomly create the grid and determine the positions of the mines before the player uncovers the first cell. A quite unsatisfying consequence of these implementations is that even if the player follows an optimal strategy, there are situations in which no cell is guaranteed to be mine free, i.e., the user depends on luck when uncovering further cells. For example, Figure 7.2 shows such scenarios. In the first case, the user has uncovered a single cell that has one neighbouring mine. The chances are 4:1 to select an empty cell among the cell's neighbours. In the scenario on the right-hand side chances are 1:1. While in these examples the player will probably just start a new game in case a mine was picked, Figure 7.3 shows a setting in which the game was almost successfully completed, however the last decision, offering a 1:1 chance, was wrong.

Our aim is creating a *fair* game variant, where the player is guaranteed that a cell being uncovered does not contain a mine whenever this cell is possibly empty and there is no *safe cell*,

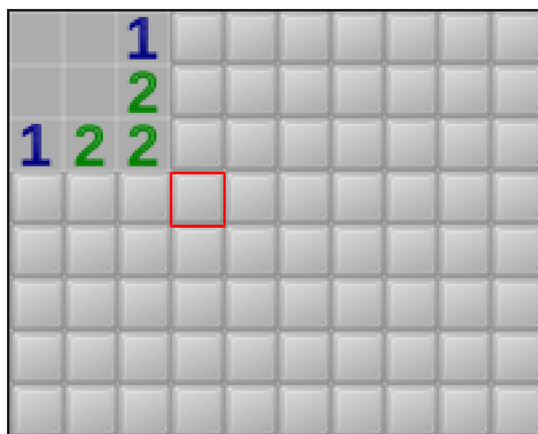

Figure 7.4: The cell $(4,4)$ that is highlighted with a red box is a safe cell. The twos in cells $(2,3)$ and $(3,2)$ in combination with the ones require that cells $(3,4)$ and $(4,3)$ are mines. As cell $(3,3)$ has only two neighbouring mines, $(4,4)$ cannot contain one.

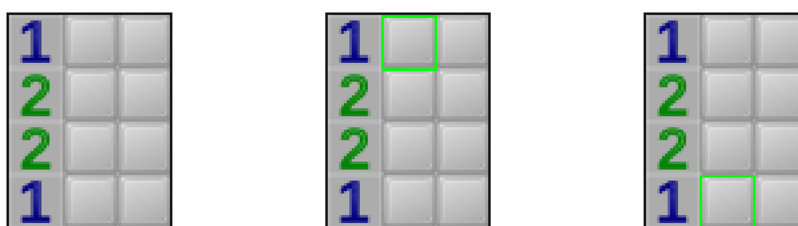

Figure 7.5: A fair minesweeper instance and its two solutions, indicating the safe cells $(2,1)$ and $(2,4)$ by green boxes.

i.e., a yet covered cell for which it can be derived that it does not contain any mine given the knowledge available to the player. For instance, the cell marked with a red box in Figure 7.4 is a safe cell. Hence, we aim for an implementation where the assignment of mines may change (for yet covered cells).

One particular task for implementing fair minesweeper could be to identify safe cells for a given game situation. We want to develop an answer-set program for this task, i.e., each answer set of the program joined with a set of facts that describes a minesweeper situation identifies a safe cell, and, conversely, for each safe cell there is such a corresponding answer set.

Like in the case of maze generation, we next describe the predicate schema that we use for encoding the fair minesweeper problem. First, the predicate `cell/2` is used to define the cells in the grid, where the first argument corresponds to the column and the second to the row of the cell. Also here, columns and rows are represented by a range of consecutive, ascending integers, starting at `1`. The number of neighbouring mines of uncovered cells is encoded using atoms of the predicate `number/3` where the first two arguments determine the cell and the third one a number between `0` and `8`. Uncovered cells are identified by atoms of the predicate `uncovered/2`. Moreover, the total number of mines is given as the argument of `nrOfMines/1`.

Example 35. For example, the instance of Figure 7.5 can be encoded by the facts of program `ex35.gr`:

```
ex35.gr                                                   Gringo

cell(1..3, 1..4).
covered(2..3, 1..4).
number(1, 1, 1).
number(1, 2, 2).
number(1, 3, 2).
number(1, 4, 1).
nrOfMines(4).
```

■

An answer set of the targeted encoding should contain exactly one atom of the predicate `safeCell/2`, indicating the position of a safe cell, e.g., for the example in Figure 7.5 there should be two answer sets, one containing `safeCell(2,1)`, the other `safeCell(2,4)`.

7.2 General Idea

We introduce stepping for ASP as a strategy to identify mismatches between the intended semantics of an answer-set program under development and its actual semantics. The general idea is to monotonically build up an interpretation by, in each step, adding literals derived by a rule that is active with respect to the interpretation obtained in the previous step. The process is interactive in the sense that at each such step the user chooses the active rule to proceed with and decides which literals of the rule should be considered true or false in the target interpretation. Hereby, the user only adds rules he or she thinks are active in an expected or an unintended actual answer set. The interpretation grows monotonically until it is eventually guaranteed to be an answer set of the overall program, otherwise the programmer is informed why and at which step something went wrong. This way, one can in principle without any backtracking direct the computation towards the interpretation one has in mind. In debugging, having the programmer in the role of an oracle is a common scenario (Shapiro, 1982). It is reasonable to assume that a programmer has good intuitions on where to guide the search if there is a mismatch between the intended and the actual behaviour of a program. We use the computation models of Chapters 5 and 6 to ensure that, if the interpretation specified in this way is indeed an answer set, the process of stepping will eventually terminate with the interpretation as its result. Otherwise, it will get stuck at some step where the user gets insight why the interpretation is not an answer set, e.g., when a constraint becomes irrevocably active or no further rule is active that could derive some desired literal.

Due to the declarativity of ASP, once one detects unintended semantics, it can be a tough problem to manually detect the reason. Stepping is a method for breaking this problem into smaller parts and structuring the search for an error. At the same time, relying on the user's intuition on which rules to proceed with, stepping can be guided such that the search quickly results in new insights. The approach is inspired by stepping-based debugging for procedural languages, where the behaviour of a program is analysed by executing statement by statement, following the program's control flow, and inspecting variable assignments. As the series of case studies provided later in Section 7.6 demonstrates, the declarativity in ASP is not in discrepancy with adapting a method from the imperative paradigm, but fruitful instead. On the one hand, with stepping the user always has guidance for starting the search for bugs, and, on the other hand, the interactive choice for the next rule makes stepping in ASP in a sense more flexible than traditional stepping, where the control flow dictates which statements are to be considered next. To still allow for fast debugging, procedural language debuggers allow for setting *breakpoints*, i.e., marking statements until which execution is done automatically. We also have a similar

feature in stepping for ASP, called *jumping*, that allows to jointly consider multiple rules which are assumed to be correct. Hence, we can speed up stepping by only inspecting suspicious parts of the program step-by-step.

7.3 Steps

By a *step* we mean the extention of a computation by a further state. We consider a scenario, where a programmer has written an answer-set program in a solver language for which a G-program Π is a groundable abstraction. Moreover, we assume that the programmer has obtained some (black-box/conditional) computation for Π that is neither stuck in Π nor complete for Π. For performing a step, one needs to find a successor state S_{n+1} for S_n such that $C' = S_0, \ldots, S_{n+1}$ is a (black-box/conditional) computation for Π.

In our stepping methodology we propose a sequence of three user actions to perform a step. These user actions, described next, can be supported by an interactive debugging environment. Intuitively, for finding a successor state, we suggest to

1. select a non-ground rule with active ground instances, then
2. choose an active ground rule among the instances of the non-ground rule, and
3. select for yet undefined atoms in the domain of the ground instance whether they are considered true or false.

The approach allows for quickly finding a successor with the help of a debugging system. In what follows we give a more detailed description of the three user actions and assume that G-programs represent non-ground programs in the solver language and C-programs represent corresponding groundings, following the abstraction approach of the previous chapters.

First, the user selects a non-ground rule $\rho \in \Pi$. In practise, this can be realised, e.g., by directly selecting ρ in the editor in which the program was written. In a conditional grounding setting, we require that ρ is settled in Π under I_{S_n}. A potential aid that could be given by a debugging system for this user action is to automatically determine the subset of G-rules in Π that have at least one C-rule r in their grounding such that could lead to a successor state, i.e., $r = r_{new}(S_n, S)$ for some successor S of S_n. Moreover, under conditional grounding, the system can indicate which rules of Π are already settled under I_{S_n}.

In the second choice, the user selects either a rule $r \in gr_b(\rho, \Pi)$ in the case of black-box grounding or $r \in gr_c(\rho, I_{S_n}, F)$ under conditional grounding. As the ground instances of ρ are not part of the original program Π, picking one requires a different approach as for choosing ρ. Here, a debugging system can display the ground rules in a dedicated area and, as before, restrict the choice of rule groundings of ρ to C-rules that could lead to a successor state. Filtering techniques can be used to restrict the amount of the remaining C-rules, e.g., by letting the user define partial assignments for the variables in ρ that determine a subset of the considered instances.

In the third user action for performing a step, the programmer chooses the truth values for the atoms in D_ρ that are neither in I_{S_n} nor in $I^-{}_{S_n}$. This choice must be made in a way such that there is a successor S_{n+1} of S_n with $P_{S_{n+1}} = P_{S_n} \cup \{r\}$, $I_{S_{n+1}} = I_{S_n} \cup \Delta$, and $I^-{}_{S_{n+1}} = I^-{}_{S_n} \cup \Delta^-$, where Δ contains the atoms the user chose to be true and Δ^- the atoms considered false. That is, S_n, Δ, and Δ^- must fulfil the conditions of Definition 52 on page 58. Here, the user needs only to ensure that Condition (v) of Definition 52 holds, i.e., $I_{S_{n+1}} \models \mathrm{B}(r)$ and $I_{S_{n+1}} \models^{\exists} \mathrm{H}(r)$, as the other conditions automatically hold once all unassigned atoms have been assigned to Δ and Δ^-. In particular, note that the set of unfounded sets, $\Upsilon_{S_{n+1}}$ can always be automatically computed following Condition (vi) of Definition 52 and does not impose restrictions on the choice of Δ and Δ^-. A support system for stepping should check

whether Condition (v) holds for the truth assignment specified by the user. Additional help can be given by automatically assigning atoms to Δ or Δ^- whenever their truth values are the same for all successor states that are based on adding r.

Example 36. As a first step for developing the maze-generation encoding, we want to identify border cells and guess an assignment of walls and empty cells. Our initial program is `ex36.gr`, given next.

```
ex36.gr                                                    Gringo
maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).

wall(X,Y) :- border(X,Y), not entrance(X,Y),
                                         not exit(X,Y).
{ wall(X,Y) : col(X): row(Y) : not border(X,Y) }.
empty(X,Y) :- col(X), row(Y), not wall(X,Y).
```

The first two rules extract the numbers of columns and rows of the maze from the input facts of predicates `col/1` and `row/1`. The next four rules derive `border/2` atoms that indicate which cells form the border of the grid. The final three rules derive `wall/2` atoms for border cells except entrance and exit, guess `wall/2` atoms for the remaining cells, and derive `empty/2` atoms for non-wall cells, respectively.

We use `ex36.gr` in conjunction with the facts in program `exMazeInstance.gr` as shown on page 88 that determine the problem instance. Recall, that we use the + operator as a means for referring to compositions of programs with explicit filename, i.e., we deal with the program `exMazeInstance.gr + ex36.gr` in this case.

We start a stepping session with the computation $C_0 = S_0$ consisting of the empty state $S_0 = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$. Following the scheme of user actions described above for performing a step, we first look for a non-ground rule with instances that are active under I_{S_0}. As $I_{S_0} = \emptyset$, only the facts from `exMazeInstance.gr` have active instances. We choose the rule `entrance(1,2).` In this case, the only (active) instance of the rule is (under black-box or conditional grounding) identical to the rule, i.e., the fact:

```
entrance(1,2).
```

The only atom in the domain of the rule instance is `entrance(1,2)`. Therefore, when performing the final user action for a step one has to decide the truth value of this atom. In order to fulfil Condition (v) of Definition 52 (on page 58), the rule head, i.e., `entrance(1,2)`, must be true in the successor state. Thus, our first step results in the computation $C_1 = S_0, S_1$ where

$$S_1 = \langle \{\texttt{entrance(1,2).}\}, \{\texttt{entrance(1,2)}\}, \emptyset, \{\emptyset\} \rangle.$$

For the next step, we choose the rule

```
col(1..5).
```

from `exMazeInstance.gr`. Similar as in Example 31 on page 72, the rule has multiple instances under `Gringo` grounding:

```
col(1).
col(2).
col(3).
col(4).
col(5).
```

We select the instance `col(5).` Since the head of the rule must be true under the successor state, as before, atom `col(5)` must be considered true in the successor state of S_1. The resulting computation after the second step is $C_2 = S_0, S_1, S_2$, where

$$S_2 = \langle\{\texttt{entrance(1,2).col(5).}\}, \{\texttt{entrance(1,2), col(5)}\}, \emptyset, \{\emptyset\}\rangle.$$

Under I_{S_2} a further rule in `exMazeInstance.gr + ex36.gr` has active instances:

```
maxCol(X) :- col(X), not col(X+1).
```

That is, it has the active instance

```
maxCol(5) :- col(5), not col(6).
```

that we choose for the next step. In order to ensure that Condition (v) of Definition 52 is satisfied, we need to ensure that head and body are satisfied under the successor state. Hence, atom `maxCol(5)` has to be considered true, whereas `col(6)` must be considered false. We obtain the computation $C_3 = S_0, S_1, S_2, S_3$, where

$$S_3 = \langle\{\texttt{entrance(1,2).col(5).maxCol(5) :- col(5), not col(6).}\},$$
$$\{\texttt{entrance(1,2),col(5),maxCol(5)}\}, \{\texttt{col(6)}\}, \{\emptyset\}\rangle. \quad \blacksquare$$

7.4 Jumps

If one wants to simulate the computation of an answer set I in a stepping session using steps only, as many steps are necessary as there are active rules in the grounding under I. Although, typically the number of active ground instances is much less than the total number of rules in the grounding, still many rules would have to be considered. In order to focus on the parts of a computation that the user is interested in we introduce a *jumping* technique for quickly considering rules that are of minor interest, e.g., for rules that are already considered correct. We say that we *jump through* these rules. By performing a jump, we mean to find a state that could be reached by a computation for the program at hand that extends the current computation by possibly multiple states. If such a state can be found, one can continue to expand a computation from that while it is ensured that the same states could be reached by using steps only. Jumps can be performed exploiting the results of Corollary 10 (on page 76) in the case of black-box grounding and Theorem 19 (on page 82) for conditional grounding. In essence, jumping can be done as follows.

1. Select rules that you want to jump through (i.e., the rules you want to be considered in the state to jump to),

2. an auxiliary answer-set program is created that contains the selected rules and the active rules of the current computations final state, and

3. a new state is computed from an answer set of the auxiliary program.

Next, we describe the items in more detail. We assume that a (black-box/conditionally)-groundable G-program Π and a (black-box/conditional)-computation $S_0, \ldots, S_n$ for Π are given.

The first user action essentially differs for the two types of grounding. In the black-box case, one may only choose a subset $P \subseteq gr_b(\Pi)$ of the black-box grounding of Π. Hence, an implication for a stepping support environment is the necessity of means to select the ground instances that form P. In order to keep memory resources and the amount of rules that have to be considered by the user low, the system could split the selection of a C-rule for P in two phases. First, the user selects a non-ground rule ρ, similar as in the first user action of defining a step. Then, the system provides so far unconsidered rules of $gr_b(\rho, \Pi)$ for selection, where similar filtering techniques as sketched for the second user action for performing a step can be applied. Under conditional grounding, we are not restricted to choose rules from the grounding but can choose non-ground rules (in case they are settled) to be considered in the jump. Hence, the user can select a set Π' of G-rules settled in Π with respect to I_{S_n}. As $\Pi' \subseteq \Pi$, this could, e.g., be done in the editor in which the answer-set program is written.

The auxiliary program of the second item can be automatically computed. For black-box testing it is a C-program P_{aux} given by $P_{aux} = P_{S_n} \cup P \cup P_{con}$, where

$$\begin{aligned} P_{con} = &\{\leftarrow not\ a \mid a \in I_{S_n}\} \cup \\ &\{\leftarrow a \mid a \in I^{-}{}_{S_n}\} \end{aligned}$$

is a set of constraints that ensure that for every answer set I of P_{aux} we have $I_{S_n} \subseteq I$ and $I \cap I^{-}{}_{S_n} = \emptyset$. Under conditional testing, the auxiliary program is the G-program $\Pi_{aux} = P_{S_n} \cup \Pi' \cup P_{con}$, where P_{con} is identical to the black-box case.

After computing an answer set I of the auxiliary program, Corollary 10 and Theorem 19 ensure the existence of a (black-box/conditional) computation $C' = S_0, \ldots, S_n, S_{n+1}, \ldots, S_m$ for Π such that S_m is stable and $I_{S_m} = I$. In the case of black-box grounding, $P_{S_m} = P_{S_n} \cup P^I$, and for conditional grounding, $P_{S_m} = gr_c(P_{S_n} \cup \Pi')^I$. In either case, the user can proceed with further steps or jumps extending the computation $S_0, \ldots, S_m$ as if S_m had been reached by steps only.

Note that non-existence of answer sets of the auxiliary program does not imply that Π has no answer sets as shown next.

Example 37. Consider the (black-box/conditionally) groundable G-program Π consisting of the C-rules

$$a \leftarrow$$

and

$$\leftarrow not\ a$$

that has $\{a\}$ as its unique answer set. Assume we want to jump through the second rule starting from the computation $C = \langle \emptyset, \emptyset, \emptyset, \{\emptyset\} \rangle$ consisting of the empty state. Then, $P_{aux} = \{\leftarrow not\ a\}$ has no answer set. ■

The example shows that jumping only makes sense when the user is interested in a computation reaching an answer set of the auxiliary program. In case of multiple answer sets of the auxiliary program, the user could pick any or a stepping environment can choose one at random. For practical reasons, the second option seems more preferable. On the one hand, presenting multiple answer sets to the user can lead to a large amount of information that has to be stored and processed by the user. And on the other hand, if the user is not happy with the truth value of some atoms in an arbitrary answer set of the auxiliary program, he or she can use steps to define the truth of these atoms before performing the jump.

Example 38. We want to continue computation C_3 for program `exMazeInstance.gr + ex36.gr` from Example 36. As we are interested in the final three rules of `ex36.gr` that derive `empty/2` and `wall/2` atoms but these rule depend on atoms of predicate `border/2`, `entrance/2`, and `exit/2` that are not yet considered in C_3, we want to jump through the facts from `exMazeInstance.gr` and the rules

```
maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).
```

of program `ex36.gr`. Assume we use conditional grounding. Then, Π' is formed by the program `exMazeInstance.gr` and the rules above and the resulting auxiliary program is given by:

```
ex38aux.gr                                                    Gringo

% P_S3
entrance(1,2).
col(5).
maxCol(5) :- col(5), not col(6).

% Pi':
maxCol(X) :- col(X), not col(X+1).
maxRow(Y) :- row(Y), not row(Y+1).
border(1,Y) :- col(1), row(Y).
border(X,1) :- col(X), row(1).
border(X,Y) :- row(Y), maxCol(X).
border(X,Y) :- col(X), maxRow(Y).

% P_con
:- not entrance(1,2).
:- not col(5).
:- not maxCol(5).
:- col(6).
```

The program `ex38aux.gr` has the single answer set I_{aux} consisting of the atoms:

```
col(1), col(2), col(3), col(4), col(5), maxCol(5),
row(1), row(2), row(3), row(4), row(5), maxRow(5),
empty(3,4), wall(3,3), entrance(1,2), exit(5,4),
border(1,1), border(2,1), border(3,1), border(4,1),
border(5,1), border(1,2), border(5,2), border(1,3),
border(5,3), border(1,4), border(5,4), border(1,5),
border(2,5), border(3,5), border(4,5), border(5,5),
```

We obtain the new state $S_4 = \langle P_{S_4}, I_{aux}, D_{P_{S_4}} \setminus I_{aux}, \{\emptyset\}\rangle$, where P_{S_4} consists of the following rules:

```
col(1). col(2). col(3). col(4). col(5).
row(1). row(2). row(3). row(4). row(5).
wall(3,3). empty(3,4). entrance(1,2). exit(5,4).
maxCol(5) :- col(5), not col(6).
maxRow(5) :- row(5), not row(6).
border(1,1) :- col(1), row(1).
border(2,1) :- col(2), row(1).
border(3,1) :- col(3), row(1).
border(4,1) :- col(4), row(1).
```

```
border(5,1) :- col(5), row(1).
border(1,2) :- col(1), row(2).
border(5,2) :- row(2), maxCol(5).
border(1,3) :- col(1), row(3).
border(5,3) :- row(3), maxCol(5).
border(1,4) :- col(1), row(4).
border(5,4) :- row(4), maxCol(5).
border(1,5) :- col(1), row(5).
border(5,1) :- row(1), maxCol(5).
border(5,5) :- row(5), maxCol(5).
border(1,5) :- col(1), maxRow(5).
border(2,5) :- col(2), maxRow(5).
border(3,5) :- col(3), maxRow(5).
border(4,5) :- col(4), maxRow(5).
border(5,5) :- col(5), maxRow(5).
```

Theorem 19 (on page 82) ensures the existence of a conditional computation $C_4 = S_0, S_1, S_2, S_3, \ldots, S_4$ for program `exMazeInstance.gr + ex36.gr`. ■

7.5 Methodology

In this section, we describe how the framework we have introduced can be used in a stepping methodology. We describe stepping on three conceptual levels. Section 7.5.1 identifies the iterative advancement of a computation as the technical level of our methodology that we refer to as stepping cycle. In Section 7.5.2, we describe how our technique applies to debugging and program analysis and the embedding of stepping in the ASP development context is subject of Section 7.5.3. Finally, in Section 7.5.4, we compile practical guidelines for our methodology in an illustrative chart. Application scenarios are provided later in Section 7.6.

7.5.1 Stepping Cycle

The iterative extension of a computation in the stepping methodology using steps and jumps can be described as a *stepping cycle* that is depicted in Figure 7.6. It summarises how a user may advance a computation along the lines of Sections 7.3 and 7.4, i.e., it provides a *technical level* representation of the stepping methodology. We assume that a stepping session always starts with the computation consisting of the empty state.

7.5.2 Program Analysis and Debugging Level Methodology

The main purpose for stepping in the context of this thesis is its application for debugging and analysing answer-set programs. In this section, we describe how insight into a program is gained using stepping. During stepping, the user follows his or her intuitions on which rule(s) to apply next and which atoms to consider true or false. In this way, an interpretation is built up that either is or is not an answer set of the program. In both cases, stepping can be used to analyse the interplay of rules in the program in the same manner, i.e., one can see which rule instances become active or inactive after each step or jump. In the case that the targeted interpretation is an answer set of the program, the computation will never fail (in the sense of Definitions 54, 64, and 71 on pages 60, 75, respectively 79) or get stuck and will finally succeed. It can, however, happen that intermediate states in the computation are unstable (cf. Example 28 on page 64). For debugging, stepping towards an answer set is useful if the answer set is unwanted. In particular, one can see why a constraint (or a group of rules supposed to have a constraining effect) does not become active. For instance, stepping reveals other active rules that derive atoms that make

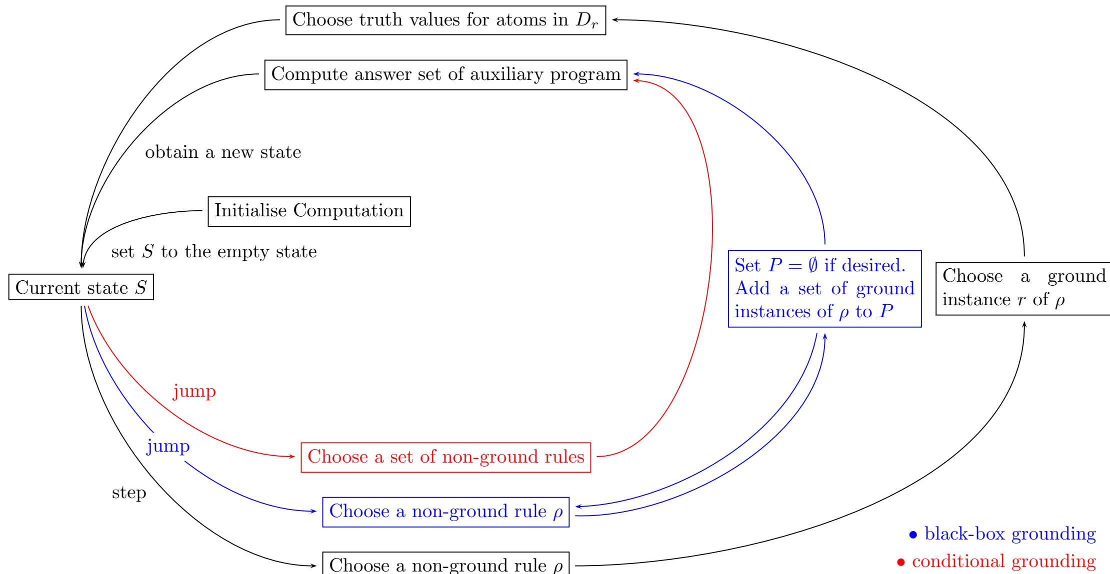

Figure 7.6: Stepping cycle

some literal in the constraint false or rules that fail do derive atoms that activate the constraint. Stepping towards an actual answer set of a program is illustrated in Example 39 on page 101.

In the case that there is an answer set that the user expects to be different, i.e., certain atoms are missing or unwanted, it makes sense to follow the approach that we recommend for expected but missing answer sets, i.e., stepping towards the interpretation that the user wants to be an answer set. Then, the computation is guaranteed to fail at some point, i.e., there is some state in the computation from which no more answer set of the program can be reached. If a stepping support environment automatically computes whether a computation has failed, the user immediately realises which of his or her decisions for which rule lead to the absence of answer sets that extend the current state's interpretation. This will in many cases reveal a bug. In other situations, the computation can already have failed before the bug can be found, e.g., the computation can have failed from the beginning in case the program has no answer sets at all. Nevertheless, the error can be found when stepping towards the intended interpretation. In most cases, there will be either a rule instance that becomes active that the user considered inactive, or the other way around, i.e., a rule instance never becomes active or is deactivated while the computation progresses. Eventually, due to our completeness results in the previous chapter, the computation will either get stuck or ends in an unstable state S such that no active external support for a non-empty unfounded set from Υ_S is available in the program's grounding. Stepping towards an interpretation that is not an answer set of the overall program can be seen as a form of hypothetical reasoning: the user can investigate how rules of a part of the program support each other before adding a further rule instance would cause an inconsistency. Moreover, in doing so one can disregard stability of the model that is built up as the stability condition of the semantics is captured in the unfounded sets component of a state that does not influence the choices the user has in stepping. Example 40 illustrates stepping towards an intended but non-existing answer set for finding a bug. Another illustration of hypothetical reasoning is given in Example 41 where a user tries to understand why an interpretation that is not supposed to be an answer set is indeed no answer set.

As mentioned in Section 7.2, if one has a clear idea on the interpretation one expects to be an answer set, stepping allows for building up a computation for this interpretation without backtracking. In practise, one often lacks a clear vision on the truth value of each and every atom with respect to a desired answer set. As a consequence, the user may require revising the decisions he or she has taken on the truth values of atoms as well as on which rules to add to the

computation. Hence, ideally, a stepping environment allows for retracting a computation to a previous state, i.e., let the user select one of the states in the computation and continue stepping from there. This way a tree of states can be built up, where every path from the root node to a leaf node is a rooted computation.

7.5.3 Top-Level Methodology

Stepping must be understood as embedded in the programming and modelling process, i.e., the technique has to be recognised in the *context* of developing answer-set programs. A practical consequence of viewing stepping in the big picture are several possibilities for exploiting information obtained during the development of a program for doing stepping faster and more accurate.

While an answer-set program evolves, the programmer will in many cases compute answer sets of preliminary versions of the program for testing purposes. If this answer sets are persisted, they can often be used as a starting point for stepping sessions for later versions of the program. For instance, in case the grounding P of a previous version of a program is a subset of the current grounding P' it is obvious that a successful computation C for P is also a computation for P'. Hence, the user can initiate a stepping session starting from C. Also in case that $P \not\subset P'$, a stepping support system could often automatically build a computation that uses an answer set of P (or parts of it) as guidance for deciding on the rules to add and the truth values to assign in consecutive states. Likewise, (parts of) computations of stepping sessions for previous versions of a program can be stored and re-used either as a computation of the current program if applicable or for computing such a computation. The idea is that previous versions of a program often constitute a part of the current version that one is already familiar with and "trusts" in. By starting from a computation that already considers a well-known part of the program, the user can concentrate on new and often more suspicious parts of the program.

7.5.4 The Stepping Guide

We next give advice how users can exploit stepping for analysing and debugging their code. In particular, Figure 7.7 synthesises practical guidelines for stepping from the methodological aspects of stepping described so far. It can be seen as a user-oriented view on the stepping technique. Depending on the goals and the knowledge of the user, this guide gives concise yet high-level suggestions on how to proceed in a stepping session. The upper area of the figure is concerned with clarifying the best strategy for a stepping session and for choosing the computation to start from. The lower area, on the other hand, guides the user through the stepping process.

The diagram differentiates between four tasks a user may want to perform.

(i) Debugging a program that lacks a particular answer set: in this case, a good strategy is to step and jump through rules that the user thinks should build up this answer set. Eventually, the computation will fail and get stuck, indicating the reason for the bug.

(ii) Debugging a program that lacks any answer set: if an intended answer set is known, we advise using the strategy of Item (i). Otherwise, the user should choose rules and truth values during stepping that he or she thinks should be consistent, i.e., lead to a successful computation. Also here, the computation is guaranteed to fail and get stuck, indicating a reason for the inconsistency of the program.

(iii) Debugging a program with an unintended answer set: In case that the unintended answer set I is similar to an intended but missing answer set I', thus if I is intuitively a wrong version of I', then we recommend stepping towards I', following the strategy of Item (i). Otherwise, the user can step towards I. Unlike in the previous cases, the computation is

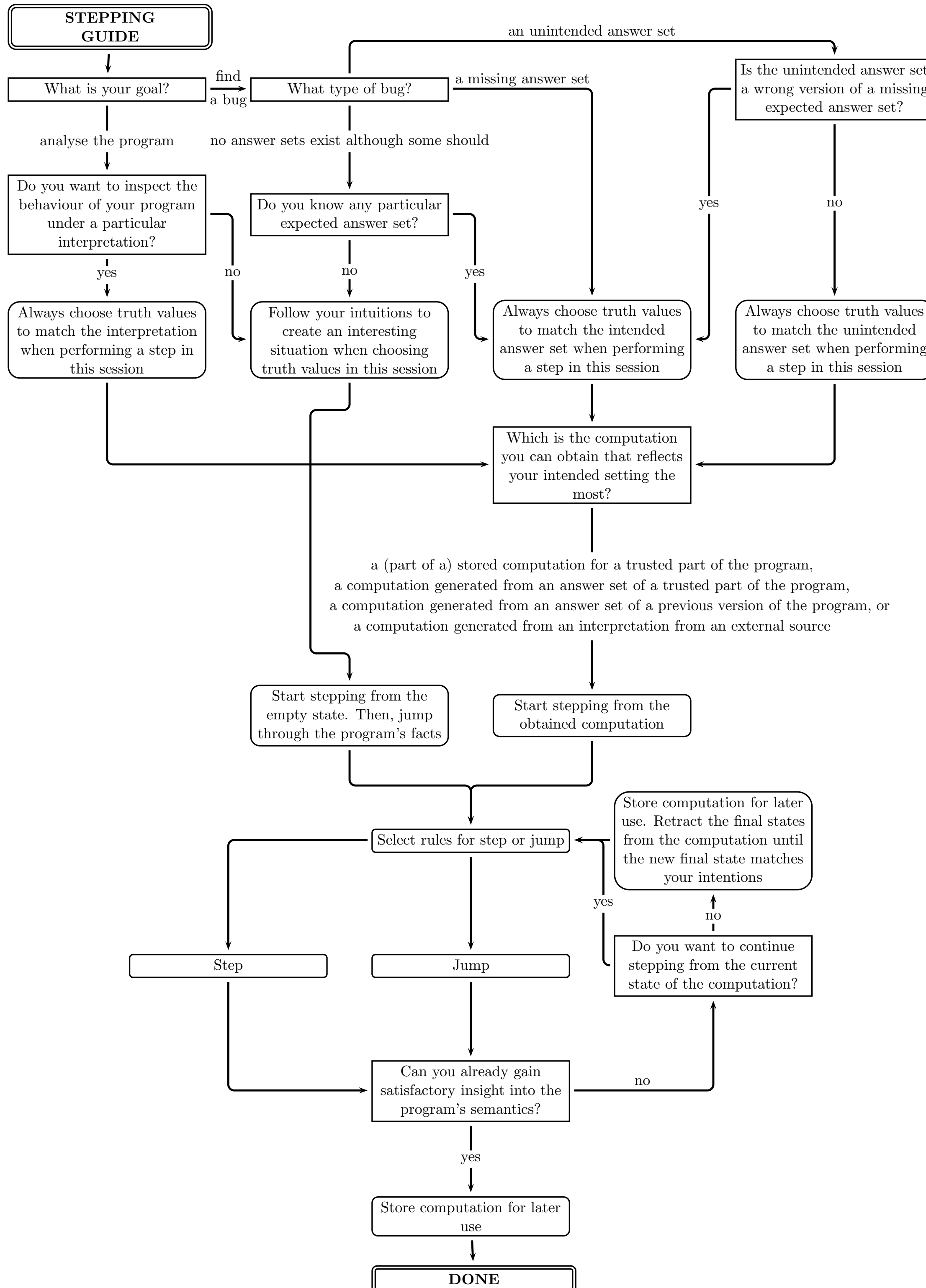

Figure 7.7: Stepping guide

guaranteed to eventually succeed. Here, stepping acts as a disciplined way to inspect how the atoms of I could be derived and why no rule is preventing I from being an answer set. Moreover, if I is intended to be a model of the program but not stable, then the stepping process will reveal which rules provide external support for sets of atoms that are supposed to be unfounded.

(iv) Analysing a program: In case that the user is interested in the behaviour of the program under a particular interpretation, it is reasonable to step towards this interpretation. Otherwise, rules and truth assignments should be chosen that drive the computation towards states that the user is interested in.

Note that the procedures suggested above and in Figure 7.7 are meant as rough guidelines for the inexperienced user. Presumably, good knowledge about the own source code and some practice in stepping will give the user a good intuition on how to find bugs efficiently.

7.6 Use Cases

In this section we show application scenarios of stepping using our running examples.

The first scenario illustrates stepping towards an interpretation that is an answer set of the program under consideration.

Example 39. We want to step towards an answer set of our partial encoding of the maze generation problem, i.e., of the program `exMazeInstance.gr + ex36.gr`. Therefore, we continue our stepping session with computation C_4, i.e., we start stepping from state S_4 that we obtained in Example 38. In particular, we want to reach an answer set that is compatible with the maze generation solution depicted in Figure 7.1. To this end, we start with stepping through the active instances of the rule

```
{wall(X,Y) : col(X): row(Y) : not border(X,Y)}.
```

Note that, as we have already considered all rule instances in S_4 that may derive atoms of predicates `col/1`, `row/1`, and `border/2`, the rule is settled in the overall program with respect to I_{S_4}. The only active instance of the rule is

```
{wall(2,2), wall(3,2), wall(4,2), wall(2,3), wall(3,3),
             wall(4,3), wall(2,4), wall(3,4), wall(4,4)}.
```

Thus, we next choose a truth assignment for the atoms appearing in the instance's choice atom. Note that we do not need to decide for the truth value of `wall(3,3)` as it is already contained in I_{S_4} and therefore already considered true. As can be observed in Figure 7.1 on page 88, among the remaining cells the rule deals with, only the one at position $(3,2)$ is a wall in our example. Hence, we obtain a new state S_5 from S_4 by extending P_{S_4} by our rule instance, I_{S_4} by `wall(3,2)`, and $I^{-}{}_{S_4}$ by `wall(2,2)`, `wall(4,2)`, `wall(2,3)`, `wall(4,3)`, `wall(2,4)`, `wall(3,4)`, `wall(4,4)`. As in S_4, the empty set is the only unfounded set in state S_5. It remains to jump through the rules

```
wall(X,Y) :- border(X,Y), not entrance(X,Y), not exit(X,Y).
```

and

```
empty(X,Y) :- col(X), row(Y), not wall(X,Y).
```

that leads to the addition of instances

```
wall(1, 1) :- border(1, 1), not entrance(1, 1), not exit(1, 1).
wall(2, 1) :- border(2, 1), not entrance(2, 1), not exit(2, 1).
wall(3, 1) :- border(3, 1), not entrance(3, 1), not exit(3, 1).
wall(4, 1) :- border(4, 1), not entrance(4, 1), not exit(4, 1).
wall(5, 1) :- border(5, 1), not entrance(5, 1), not exit(5, 1).
wall(5, 2) :- border(5, 2), not entrance(5, 2), not exit(5, 2).
wall(1, 3) :- border(1, 3), not entrance(1, 3), not exit(1, 3).
wall(5, 3) :- border(5, 3), not entrance(5, 3), not exit(5, 3).
wall(1, 4) :- border(1, 4), not entrance(1, 4), not exit(1, 4).
wall(1, 5) :- border(1, 5), not entrance(1, 5), not exit(1, 5).
wall(2, 5) :- border(2, 5), not entrance(2, 5), not exit(2, 5).
wall(3, 5) :- border(3, 5), not entrance(3, 5), not exit(3, 5).
wall(4, 5) :- border(4, 5), not entrance(4, 5), not exit(4, 5).
wall(5, 5) :- border(5, 5), not entrance(5, 5), not exit(5, 5).
empty(1, 2) :- col(1), row(2), not wall(1, 2).
empty(2, 2) :- col(2), row(2), not wall(2, 2).
empty(4, 2) :- col(4), row(2), not wall(4, 2).
empty(2, 3) :- col(2), row(3), not wall(2, 3).
empty(4, 3) :- col(4), row(3), not wall(4, 3).
empty(2, 4) :- col(2), row(4), not wall(2, 4).
empty(3, 4) :- col(3), row(4), not wall(3, 4).
empty(4, 4) :- col(4), row(4), not wall(4, 4).
empty(5, 4) :- col(5), row(4), not wall(5, 4).
```

to a new state S_6. I_{S_6} extends I_{S_5} by the head atoms of these rules that are not yet in I_{S_5}. Likewise, ${I^-}_{S_6}$ extends ${I^-}_{S_5}$ by the default negated atoms appearing in the rules that are not yet in ${I^-}_{S_5}$. As $\Upsilon_{S_6} = \{\emptyset\}$ and no rule in `exMazeInstance.gr + ex36.gr` has further active instances under I_{S_6}, the computation $S_0, \ldots, S_6$ has succeeded and hence I_{S_6} is an answer set of the program. ■

In the next example, a bug is revealed by stepping towards an intended answer set.

Example 40. As a next feature, we (incorrectly) implement rules that should express that there has to be a path from the entrance to every empty cell and that 2×2 blocks of empty cells are forbidden.

```
ex40.gr                                                    Gringo

adjacent(X,Y,X,Y+1) :- col(X), row(Y), row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y), row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y), col(X+1).
adjacent(X,Y,X-1,Y) :- col(X), row(Y), col(X-1).
reach(X,Y)    :- entrance(X,Y), not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y),
                                    not wall(XX,YY).

:- empty(X,Y), not reach(X,Y).
:- empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1).
```

The first six rules formalise when an empty cell is reached from the entrance, and the two constraints should ensure that every empty cell is reached and that no 2×2 blocks of empty cells exist, respectively.

Assume that we did not spot the bug in the second constraint—in the third body literal the term `Y+1` was mistaken for `X+1`. This could be the result of a typical copy-paste error. It turns

out that `exMazeInstance.gr + ex36.gr + ex40.gr` has no answer set. In order to find a reason, one can start stepping towards an intended answer set. We assume that the user already trusts the program `exMazeInstance.gr + ex36.gr` from Example 39. Hence, he or she can reuse the computation $S_0, \ldots, S_6$ for `exMazeInstance.gr + ex36.gr` as starting point for a stepping session because all rules in P_{S_6} are also ground instances of rules in the extended program `exMazeInstance.gr + ex36.gr + ex40.gr`. Then, when the user asks for rules with active ground instances a stepping support environment would present the following rules:

```
adjacent(X,Y,X,Y+1) :- col(X), row(Y), row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y), row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y), col(X+1).
adjacent(X,Y,X-1,Y) :- col(X), row(Y), col(X-1).
reach(X,Y)    :- entrance(X,Y), not wall(X,Y).

:- empty(X,Y), not reach(X,Y).
:- empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1).
```

The attentive observer will immediately notice that two constraints are currently active. There is no reason to be deeply concerned about

```
:- empty(X,Y), not reach(X,Y).
```

being active because the rule defining the `reach/2` predicate—that can potentially deactivate instances of the constraint—has not been considered yet. However, the constraint

```
:- empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1).
```

contains only atoms of predicate `empty/2` that has already been fully evaluated. Even if `empty/2` was only partially evaluated, an active instance of the constraint could not become inactive in a subsequent computation for the sole ground that it only contains monotonic literals. When the user inspects the single ground instance

```
:- empty(1,2), empty(2,2), empty(1,2), empty(2,3).
```

of the constraint the bug becomes obvious. A less attentive observer would maybe not immediately realise that the constraint will not become inactive again. In this case, he or she would in the worst case step through all the other rules before the constraint above remains as the last rule with active instances. Then, at the latest, one comes to the same conclusion that `X+1` has to be replaced by `Y+1`. Moreover, a stepping environment could give a warning when there is a constraint instance that is guaranteed to stay active in subsequent states. We refer to the corrected version of program `ex40.gr` by `ex40b.gr`. ■

Compared to traditional software, programs in ASP are typically very succinct and often authored by a single person. Nevertheless, people are sometimes confronted with ASP code written by another person, e.g., in case of joint program development, software quality inspection, legacy code maintenance, or evaluation of student assignments in a logic-programming course. As answer-set programs can model complex problems within a few lines of code, it can be pretty puzzling to understand someone else's ASP code, even if the program is short. Here, stepping can be very helpful to get insight into how a program works that was written by another programmer, as illustrated by the following example.

Example 41. Imagine the full encoding of the maze generation encoding that is composed by the programs `ex36.gr + ex40b.gr` and the constraints in `ex41.gr`, given next, has been written by another author.

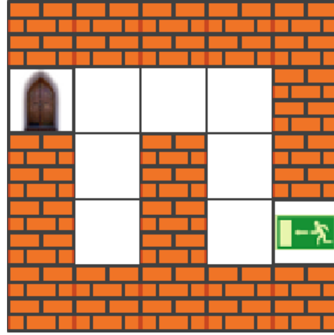

Figure 7.8: A valid maze—but not a solution for the instance depicted in Figure 7.1 as that requires $(3, 4)$ to be an empty cell.

```
ex41.gr                                                          Gringo
:- exit(X,Y), wall(X,Y).
:- wall(X,Y), wall(X+1,Y), wall(X,Y+1), wall(X+1,Y+1).
:- wall(X,Y), empty(X+1;X-1,Y), empty(X,Y+1;Y-1),
                                    col(X+1;X-1), row(Y+1;Y-1).
:- wall(X,Y), wall(X+1,Y+1), not wall(X+1,Y),
                                                 not wall(X,Y+1).
:- wall(X+1,Y), wall(X,Y+1), not wall(X,Y),
                                               not wall(X+1,Y+1).
```

Note that the guess whether a cell is a wall or empty in the program `ex36.gr + ex40b.gr + ex41.gr` is realised by guessing for each non-border cell whether it is a wall or not and deriving that a cell is empty in case we do not know that it is a wall. Moreover, observe that facts of predicate `empty/2` may be part of a valid encoding of a maze generation problem instance, i.e., they are a potential input of the program. As a consequence, it seems plausible that the encoding could guess the existence of a wall for a cell that is already defined to be empty by a respective fact in the program input. In particular, there is no constraint that explicitly forbids that a single cell can be empty and contain a wall. The encoding would be incorrect if it would allow for answer sets with cells that are empty and a wall, as that would be inconsistent with the maze generation problem specification. However, it turns out that the answer sets of the program are exactly the intended ones. Let us find out why by means of stepping.

Reconsider the problem instance depicted in Figure 7.1 that is encoded in the program `exMazeInstance.gr`. It requires that cell $(3, 4)$ is empty. If it did not, the maze shown in Figure 7.8 that contains a wall at cell $(3, 4)$ would be a valid solution. We start a stepping session for program `exMazeInstance.gr + ex36.gr + ex40b.gr + ex41.gr`, and step towards an interpretation encoding the maze of Figure 7.8 to see what is happening if we consider $(3, 4)$ to be a wall despite the presence of fact `empty(3,4)`. We can reuse the computation C_4 obtained in Example 38 whose final state S_4 considers already the facts describing the input instance and the rules needed for deriving `border/2` atoms. As in Example 39, we continue with a step for considering the ground instance of the rule

```
{wall(X,Y) : col(X): row(Y) : not border(X,Y)}.
```

that guesses whether non-border cells are walls. This time, instead of choosing `wall(3,2)` to be true, we only add `wall(3,4)` to the atoms considered true. Then, for the resulting state S'_5, both `empty(3,4)` and `wall(3,4)` are contained in $I_{S'_5}$. A visualisation of $I_{S'_5}$ if given in the centre of Figure 7.9. In order to derive the remaining atoms of predicates `empty/2` and `wall/2` we then jump through the rules

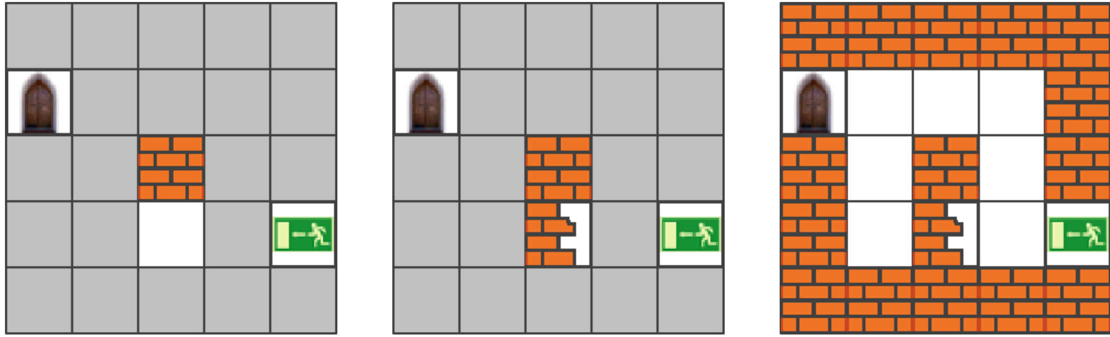

Figure 7.9: The stepping session described in Example 41: Starting from the maze generation instance we step towards an interpretation encoding the wrong solution of Figure 7.8. After stepping through the guessing rule the resulting interpretation contains atoms `empty(3,4)` and `wall(3,4)` stating that cell $(3,4)$ is both a wall and empty.

```
wall(X,Y) :- border(X,Y), not entrance(X,Y), not exit(X,Y).
```

and

```
empty(X,Y) :- col(X), row(Y), not wall(X,Y).
```

to obtain state S_6', where $I_{S_6'}$ is illustrated in the right subfigure of Figure 7.9.

Now, the user sees that constraint

```
:- empty(X,Y), not reach(X,Y).
```

has active instances. This comes as no surprise as the rules defining reachability between empty cells have not been considered yet. We decide to do so now and initiate a jump through the rules

```
adjacent(X,Y,X,Y+1) :- col(X), row(Y), row(Y+1).
adjacent(X,Y,X,Y-1) :- col(X), row(Y), row(Y-1).
adjacent(X,Y,X+1,Y) :- col(X), row(Y), col(X+1).
adjacent(X,Y,X-1,Y) :- col(X), row(Y), col(X-1).
reach(X,Y)    :- entrance(X,Y), not wall(X,Y).
reach(XX,YY)  :- adjacent(X,Y,XX,YY), reach(X,Y),
                                      not wall(XX,YY).
```

We obtain the new state S_7' and observe that under interpretation $I_{S_7'}$ the constraint still has an active instance, namely

```
:- empty(3,4), not reach(3,4).
```

Obviously, the atom `reach(3,4)` has not been derived in the computation. When inspecting the rules defining `reach/2` it becomes clear why the answer sets of the encoding are correct: the atom `reach(X,Y)` is only derived for cells that do not contain walls. Consequently, whenever there is an empty cell which was guessed to contain a wall it will be considered as not reachable from the entrance. As every empty cell has to be reachable, a respective answer-set candidate will be eliminated by an instance of the constraint

```
:- empty(X,Y), not reach(X,Y).
```

Although the encoding of the maze generation problem is correct one could consider it to be not very well designed. Conceptually, the purpose of the constraint above is forbidding empty cells to be unreachable from the entrance and not forbidding them to be walls. Moreover, if one would replace the rules

```
reach(X,Y)    :- entrance(X,Y), not wall(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y),
                                   not wall(XX,YY).
```

by

```
reach(X,Y)    :- entrance(X,Y), empty(X,Y).
reach(XX,YY) :- adjacent(X,Y,XX,YY), reach(X,Y),
                                   empty(XX,YY).
```

which seem to be equivalent in the terms of the problem specification, the program would not work. A more natural encoding would be to explicitly forbid empty cells to contain walls either by an explicit constraint or a modified guess where non-border cell is guessed to be either empty or contain a wall but not both. ■

Example 42. The program `ex42.gr`, given in Figure 7.10, provides an encoding for the fair-minesweeper problem, i.e., it identifies safe cells in a given game situation. The encoding is based on the *saturation technique* (Eiter and Gottlob, 1995) which is a strategy for expressing problems in Σ_2^P using disjunctive logic programs. While the method provides a systematic approach for encoding complex problems, arguably, this technique results in programs that are relatively difficult to understand, especially for novices of the stable-model semantics. Confronted with such an encoding, it is helpful have tools to examine its behaviour for learning how it works. Thus, we explain `ex42.gr` by means of stepping.

For our examination we will use the game setting depicted in Figure 7.5. Joining `ex42.gr` with program `ex35.gr` consisting of facts that encode the instance (given in Section 7.1.2) yields two answer sets, one indicating that $(2,1)$ is a safe cell, while $(2,4)$ is identified to be safe by the other answer set. Thus, the results are as intended. We start a stepping session by jumping through all the facts in `ex35.gr`. Based on those facts, the first eight rules in `ex42.gr` compute which cells are neighbours, the total number of covered cells, the number of covered neighbours of uncovered cells, and further auxiliary information that is uniquely defined by the instance facts. Thus, we can safely jump through these rules. The rule

```
1{safeCell(X,Y):covered(X,Y)}1.
```

allows for choosing one of the covered cells as a candidate for being a safe cell. As a first case we will consider the cell $(3,1)$ to be the candidate cell, despite knowing that $(3,1)$ is no safe cell for the given instance, i.e., there is an assignment of the yet covered cells to be either empty or mines that respects the minesweeper specifications where $(3,1)$ is assigned a mine. Note that there is also a valid assignment such that $(3,1)$ is empty. We perform the respective step and thereby consider atom `safeCell(3,1)` to be true. When stepping through the disjunctive rule

```
mine(X,Y)|empty(X,Y) :- covered(X,Y).
```

one can consider a covered cell (X,Y) to be a mine, empty, or both. While the latter seems to be counterintuitive from a knowledge representational point of view, indeed every answer set of the encoding has to contain both a `mine/2` and an `empty/2` atom, for every covered cell. This aspect is natural for saturation encodings and one reason why they are not always easy to understand. But for now, in our stepping session, let us consider an interpretation that contains either a `mine/2` and an `empty/2` atom for each uncovered cell by stepping through every active instance of the disjunctive rule. In doing so, we describe a valid minesweeper configuration, placing exactly as many mines in neighbouring fields as indicated by the respective number in a cell and overall exactly 4 mines as required in the chosen problem instance. In particular, let us choose the candidate cell $(3,1)$ and cells $(2,2)$, $(2,3)$, and

```
ex42.gr                                                    Gringo

% establish neighbourhood and count covered
neighbour(X,Y,NX,NY) :- cell(X,Y),cell(NX,NY),|NX-X|<=1,
                              |NY-Y|<=1,|NX-X|+|NY-Y|>=1.
potentialNrOfNeighbours(1..8).
nrCols(X) :- cell(X,Y),not cell(X+1,Y).
nrRows(Y) :- cell(X,Y),not cell(X,Y+1).
nrOfCells(X*Y) :- nrCols(X),nrRows(Y).
cellIndex(1..X) :- nrOfCells(X).

nrOfCovered(N) :- N={covered(X,Y):cell(X,Y)}.
nrOfCoveredNeighbours(X,Y,N) :- number(X,Y,_),
                  N={covered(NX,NY):neighbour(X,Y,NX,NY)}.

% guess candidate
1{safeCell(X,Y):covered(X,Y)}1.

% guess mines
mine(X,Y) | empty(X,Y) :- covered(X,Y).

atLeastNCoveredNeighboursMines(X,Y,N) :- number(X,Y,_),
                              potentialNrOfNeighbours(N),
      N {mine(NX,NY):neighbour(X,Y,NX,NY):covered(NX,NY)}.
atLeastNCoveredNeighboursFree(X,Y,N) :- number(X,Y,_),
                              potentialNrOfNeighbours(N),
     N {empty(NX,NY):neighbour(X,Y,NX,NY):covered(NX,NY)}.
atLeastNCoveredMines(N) :- cellIndex(N), nrOfCovered(NC),
                         N<=NC,N{mine(X,Y):covered(X,Y)}.
atLeastNCoveredFree(N) :- cellIndex(N), nrOfCovered(NC),
                       N<=NC, N{empty(X,Y):covered(X,Y)}.

% spoiling conditions
spoil :- mine(X,Y),empty(X,Y).
spoil :- number(X,Y,N),
                  atLeastNCoveredNeighboursMines(X,Y,N+1).
spoil :- number(X,Y,N),nrOfCoveredNeighbours(X,Y,NCN),
             atLeastNCoveredNeighboursFree(X,Y,(NCN-N)+1).
spoil :- nrOfMines(N),atLeastNCoveredMines(N+1).
spoil :- nrOfMines(N),nrOfCovered(NC),
                           atLeastNCoveredFree((NC-N)+1).
spoil :- safeCell(X,Y), empty(X,Y).

% spoiling
mine(X,Y) :- covered(X,Y),spoil.
empty(X,Y) :- covered(X,Y),spoil.

:- not spoil.
```

Figure 7.10: Program `ex42.gr` encodes the fair-minesweeper problem.

$(3,2)$ to contain mines and the remaining covered fields to be empty. We can then step or jump through the rules defining the predicates `atLeastNCoveredNeighboursMines/3`, `atLeastNCoveredNeighboursFree/3`, `atLeastNCoveredMines/1`, and the predicate `atLeastNCoveredFree/1`. In doing so we have no more decisions to take as these rules only depend on predicates that have already been evaluated in our computation. At this point, the computation gets stuck in a state, that we later refer to as S_1, where the only remaining active instance is the constraint

```
:- not spoil.
```

Inspecting this constraint, one sees that the atom `spoil` must be present in any potential answer set of the encoding. Moreover, the rules

```
mine(X,Y) :- covered(X,Y),spoil.
empty(X,Y) :- covered(X,Y),spoil.
```

enforce that if `spoil` is true then every covered cell has to be considered both as containing a mine and as being empty. Thus, we restart the stepping session from the point where we added instances of the rule

```
mine(X,Y)|empty(X,Y) :- covered(X,Y).
```

This time, we consider both head atoms to be true for every covered cell. As before, after deciding truth for `mine/2` and `empty/2` predicates, there is no choice left when stepping through the remaining rules. After stepping until no more remaining rule is active, we end up in a state, S_2, with 770 unfounded sets. Hence, the interpretation that was built up, I_{S_2}, is a model of the program but no answer set. In the light of Definition 34 on page 41, the interpretation that we reached in the previous stepping session, I_{S_1}, prevents I_{S_2} from being an answer set because it is a proper subset of I_{S_2} and, I_{S_1},I_{S_2}, and the program satisfy Condition ($\star$) of Definition 34 as the constraint

```
:- not spoil.
```

is not part of the FLP-reduct of the program with respect to I_{S_2}.

We restart the stepping session for the second time, but now consider $(2,1)$ as candidate for a safe cell for which we know that it actually is one. Again, we first try to establish a valid minesweeper configuration with either a `mine/2` or an `empty/2` atom true for each covered cell. Thus, as $(2,1)$ is a safe cell, we can only choose a configuration in which $(2,1)$ is empty. As a consequence, the rule instance

```
spoil :- safeCell(2,1), empty(2,1).
```

becomes active and `spoil` has to be derived. When continuing the computation we will end up in some stuck state S_3, as the rules

```
mine(X,Y) :- covered(X,Y),spoil.
empty(X,Y) :- covered(X,Y),spoil.
```

become active for each covered cell (X,Y) that would derive the atoms `empty(X,Y)` and `mine(X,Y)`.

Now we restart again, keeping $(2,1)$ as safe cell candidate, and try to reach an invalid minesweeper configuration without deriving `spoil`, i.e., an assignment for the covered cells such that for none of them both the `mine/2` and the `empty/2` atom are true but the minesweeper constraints are violated, i.e., the total number of mines or the given number of neighbouring mines is violated for some cell. Then, however, some instance of the rules

```
spoil :- number(X,Y,N),
                    atLeastNCoveredNeighboursMines(X,Y,N+1).
spoil :- number(X,Y,N),nrOfCoveredNeighbours(X,Y,NCN),
               atLeastNCoveredNeighboursFree(X,Y,(NCN-N)+1).
spoil :- nrOfMines(N),atLeastNCoveredMines(N+1).
spoil :- nrOfMines(N),nrOfCovered(NC),
                              atLeastNCoveredFree((NC-N)+1).
```

that detect invalid minesweeper configurations becomes active and forces `spoil` to be true. Consequently, we will end up in some stuck state S_4 again.

Finally, we step towards the interpretation that is obtained by applying all active rules where $(2,1)$ is considered the safe cell candidate and for every covered cell both the `mine/2` and the `empty/2` atom are chosen to be true. This time, the computation succeeds, i.e., we arrive at a state, let us call it S_5, where no further rule is active and, unlike S_2, we do not have any non-empty unfounded set left. The explanation why I_{S_5} is an answer set, in contrast to I_{S_2}, can be given using our example stepping sessions to the stuck states S_3 and S_4. We have seen that for S_2 there exists state S_1 that corresponds to a valid minesweeper configuration where the candidate cell is a mine and I_{S_1} prevents I_{S_2} from being an answer set. For S_5, there is no such preventing state like S_1 is for S_2 because there is no valid minesweeper configuration where the candidate cell is a mine. Trying to step to a preventing state will result in a stuck state like S_3 where the candidate cell is considered empty or a stuck state like S_4 where one of the minesweeper constraints is violated. In both cases the rule instance that causes the state to be stuck is part of the FLP-reduct of the program with respect to I_{S_5}. It is the essence of the saturation technique that for a desired solution like I_{S_5} no state like S_1 can be reached in which the constraint

```
:- not spoil.
```

that is not part of the FLP-reduct with respect to I_{S_5} is the only unsatisfied rule.

We do not claim that an ASP novice will learn the saturation technique by himself or herself by just applying stepping in an uneducated way. Nevertheless, stepping provides an ASP instructor with the means to give students insight into partial assignments for a saturation encoding. This potentially leads to a better comprehension of its behaviour that is sometimes perceived as a form of black magic when one just looks at the resulting answer sets. ■

7.7 General Guidelines for Development

As a debugging technique, stepping can help in many situations where tracking a bug manually is cumbersome. It is natural to ask how big an answer-set program can get such that it is still suitable for stepping. Due to the vague nature of the question, answers cannot be clearly established. From a complexity theoretic point of view, the problems that need to be solved in a stepping support environment for and after performing a step or a jump, e.g., computing a new state from a jump, determining rules with active instances, or checking whether a computation has failed, are not harder than computing an answer set of the program under development. Under this observation, our technique appears to be an appropriate approach for debugging ASP. However, in some applications, solving times of multiple minutes or even hours are acceptable. Certainly, having waiting times of these lengths for individual debugging steps is undesirable. On the positive side, often, following a few guidelines during the development of an answer-set program can significantly reduce the likelihood of introducing bugs, the amount of information the user has to deal with, and also the computational resources required for stepping. Next, we summarise some measures in this line, some of which were already discussed in a paper by Brain et al. (2009) who explored pragmatic methodologies for ASP development.

Make Encodings Scalable and Start with Small Examples

A developer should aim for a program that can be scaled in the sense that the size of its grounding and its answer sets depends on parameters like constants or input programs. A very common setting is to write an answer-set program as a *uniform problem-encoding*, i.e., the program itself is fixed and able to solve every instance of a computational problem, where solving a concrete instance is realised by joining the program with a set of facts that encode the instance as input. Typically, when using problem instances that can be considered to be small in their respective domain, also their encoding and the resulting grounding as well as answer sets are small. It is recommended to continuously test the evolving program with a few small example input programs or parameters. If the examples represent different corner cases the chances are good that bugs are detected early on as witnessed by an evaluation of the *small-scope hypothesis* for ASP (Oetsch et al., 2012a). The hypothesis states that a high proportion of errors can be found by testing a program for all test inputs within some small scope. Although making a program scalable is often a good design choice, the decision to do so should be evaluated on a case-by-case basis, as it sometimes leads to less natural, respectively, more complicated problem encodings. As an example, consider programs dealing with Sudoku. This game is based on a 9×9 grid of cells. While it is possible to base one's encodings on some generalisation of the game that works with $n \times n$ grids, it might not be worth the effort if one will in practise only deal with standard 9×9 Sudokus.

Visualise Answer Sets and Stepping States

Answer-set solvers provide their output in a textual format, representing answer sets by a list of its atoms. As this type of representation is often cumbersome for the user to interpret, tools like `Kara` (cf. Section 8.2.3) (Kloimüllner et al., 2013), `ASPVIZ` (Cliffe et al., 2008), `IDPDraw` (Wittocx, 2009), or `Lonsdaleite` (Smith, 2011) were developed that allow for visualising interpretations. These tools allow for concisely specifying graphical representations of interpretations. Examples for this type of visualisation include the figures for the maze generation and fair minesweeper use cases in this thesis that were created using the `Kara` system that is part of the `SeaLion` IDE presented in the next chapter. Such visualisations allow for quickly understanding which solution is encoded in an interpretation and, consequently, to easily spot when an answer set differs from what is expected. It is advisably to specify visualisations in a way that also interpretations that are not supposed to be answer sets have a meaningful graphical representation. For example consider Figures 7.1 and 7.9 on pages 88 and 105. Cells for which there is no `empty/2` or `wall/2` atom are visualised by a grey square although in an expected answer set every cell has to have either exactly one of these atoms. Likewise, we also visualise cells that have both an `empty/2` and a `wall/2` atom, as shown for cell $(3, 4)$ in Figure 7.9. This ability to visualise partial or wrong solutions can be exploited in stepping sessions for visualising interpretation I_S after reaching a new state S following a step or jump. Having a visual feedback allows a user to easily capture the semantic essence of the current state in a stepping session.

Test often and Keep Intermediate Information

"Test early, test often" is a piece of advice that is often given in the context of software engineering and arguably applies also for developing answer-set programs. If the program is run for the first time when it is already supposed to cover much of the intended functionality, the likelihood for having a bug is quite high, there are no parts of the program that the user can already trust, and one would need to start stepping from scratch, as there is no information available that allows for generating an initial computation. If the user regularly computes answer sets after adding some new functionality, he or she will recognise wrong behaviour earlier. As a

consequence, smaller parts of the program need to be searched for an error, answer sets and stepping sessions of previous versions of the program can be stored and reused for subsequent stepping sessions, and already tested parts of the program can be considered as less suspicious for containing a bug which allows for focusing only on new program parts.

7.8 Comparison to other Debugging Approaches for ASP

Previous approaches for debugging answer-set programs have been discussed in Section 2.3. Here, we focus on their differences to the stepping method proposed in this work and, for doing so, categorise them with respect to different aspects. In general, stepping can be seen as orthogonal to the basic ideas of all the other approaches we discussed. That is, it is reasonable to have a development kit that supports stepping and other debugging methods simultaneously. Nevertheless, stepping as presented in this work, overcomes current restrictions regarding language restriction and practical applicability, as discussed in Section 2.4, that previous methods still face.

For the comparison, we first examine how different approaches behave with respect to the particularities of ASP as discussed in Section 2.2. That is, how they are dealing with, one the one hand, non-determinism in the sense that a program may have multiple answer sets, and, on the other hand, with the declarative flavour of ASP. Regarding both aspects, stepping has an exceptional position due to its interactive flavour. Approaches that focus on actual answer sets of the program to be debugged include the algorithm by Brain and De Vos (2005) that aims at explaining the presence of atoms in an answer set. Also, justifications (Pontelli et al., 2009) are targeted towards explanations in a given actual answer set, with the difference that they focus on a single atom but can not only explain their presence but also their absence. The approach by Caballero et al. (2008) can also be seen to target a single actual answer set. However, handling non-determinism is irrelevant in their setting, as they target a language fragment for which every program is guaranteed to have exactly one answer set. Due to their focus on actual answer sets of the debugged program, the methods mentioned so far cannot be applied on (erroneous) programs without any answer set. The previous meta-programming based debugging technique (Gebser et al., 2008; Pührer, 2007) and follow-up works (Oetsch et al., 2010a; Polleres et al., 2013) deal with a single intended but non-actual answer set of the debugged program. The question why a set of atoms does not jointly occur in any answer set, as raised by Brain and De Vos (2005), is related to the work of Wittocx et al. (2009) on debugging first-order theories with inductive definitions Denecker (2000); Denecker and Ternovska (2008). In their approach, the user can specify a class of intended semantic structures which are not preferred models of the theory at hand (corresponding to actual answer sets of the program to be debugged in ASP terminology).

Syrjänen's diagnosis technique (Syrjänen, 2006) considers the setting when a program has no answer set at all but does not consider intentions of the user on how an answer set should look like. In contrast, stepping does not require actual or intended answer sets as a prerequisite, as the user can explore the behaviour of his or her program under different interpretations that may or may not be extended to answer sets by choosing different rules instances. In the interactive setting summarised in Figure 7.6 on page 98, where one can retract a computation to a previous state and continue stepping from there that is also implemented in `SeaLion`, a stepping session can thus be seen as an inspection across arbitrary interpretations rather than an inquiry about a concrete set of actual or non-existent answer sets. Nevertheless, if one has a concrete interpretation in mind, the user is free to focus on that. The ability to explore rule applications for partial interpretations that cannot become answer sets amounts to a form of hypothetical reasoning. A related form of this type of debugging is also available in one feature of the tagging approach (Brain et al., 2007b) that aims at extrapolating non-existent answer sets by switching off rules and guessing further atoms. Here, the stepping technique can be considered more focused, as the interpretation under investigation is determined by the choices of the

user in stepping but is essentially arbitrary in the tagging approach if the user does not employ explicit restrictions.

Most existing debugging approaches for ASP can be seen as declarative in the sense that a user can pose a debugging query, and receives answers in terms of different declarative definitions of the semantics of answer-set programs, e.g., in terms of active or inactive rules with respect to some interpretation. In particular, the approaches do not take the execution strategy of solvers into account for reasons discussed in Section 2.2. This is also the case for the stepping approach, however stepping as well as online justifications (Pontelli et al., 2009) are exceptional as both methods involve a generic notion of computation which adds a procedural flavour to debugging. Nonetheless, the computation model we use for stepping can be seen as a declarative characterisation of the answer-set semantics itself as it does not apply a fix order in which to apply rules to build up an answer set.

A computation in the sense of Pontelli et al. (2009) is a sequence of three-valued interpretations in which monotonously more atoms are considered true, respectively, false. The information carried in these interpretations corresponds to that of the second and third component of a state in a computation in our framework. The purpose of using computations in the two approaches differs. While computations in stepping are used for structuring debugging process in a natural way, where the choices how to proceed remains with the user, the computations of Pontelli et al. are abstractions of the solving procedure. Their goal is to allow a solver that is compatible with their computation model to compute justifications for its intermediate results. Thus, similar to their offline versions, online justifications are non-interactive, i.e., they are computed automatically and used for post-mortem debugging. As our computation model is compatible with that for online justifications, it seems very promising to combine the two approaches in practise. While debugging information in stepping focuses on violation of rules and unfounded sets, our method leaves the reasons for an atom being true or false as implicit consequences of a user's decision. Here, online justifications could keep track of the reasons for truth values at each state of a stepping session and presented to the user during debugging on demand.

Besides stepping, also the approach by Wittocx et al. (2009) can be considered interactive. While in their approach a fixed proof is explored interactively, the interaction in our method has influence on the direction of the computation. Although not interactive in the sense of an interleaved communication between system and user, further approaches allow the user to provide information for filtering the amount of debugging information (Brain et al., 2007b; Gebser et al., 2008). The approaches mentioned in this paragraph realise declarative debugging in the sense of Shapiro (1982), where the user serves as an oracle for guiding the search for errors (cf. Section 2.2).

Finally, we compare the ASP languages supported by different approaches. First, the language of theories with inductive definitions used in one of the debugging approaches (Wittocx et al., 2009) differs from the remaining approaches that are based on logic-programming rules. Many of these works deal only with the basic ASP setting of debugging ground answer set programs, supporting only normal rules (Brain and De Vos, 2005; Pontelli et al., 2009), disjunctive rules (Gebser et al., 2008), or simple choice rules (Syrjänen, 2006). The work on tagging-based debugging (Brain et al., 2007b) sketches how to apply the approach to programs with variables by means of function symbols. The approach by Caballero et al. (2008) deals with non-ground normal programs which have to be stratified. Explicit support for variables is also given in an extension (Oetsch et al., 2010a) of the meta-programming approach for disjunctive programs. It was later extended to allow for weight constraints (Polleres et al., 2013) by compiling them away to normal rules. A commonality of these approaches is that they target ASP languages that can be considered idealised proper subsets of current solver languages. In this respect, stepping is the first debugging approach that overcomes these limitations as the use of C-programs and abstract grounding make the framework generic enough to be applied to ASP solver languages.

While this does not mean that other approaches cannot be adapted to fit a solver language, it is no always immediately clear how. For our approach, instantiating our abstractions to the language constructs and the grounding method of a solver is sufficient to have a ready-to-use debugging method.

8 Stepping in the Integrated Development Environment SeaLion

In this chapter, we describe SeaLion, an *integrated development environment* (IDE) for ASP. It was developed as one of the major goals of the MMDASP project (see Section 1.2) and comes with an implementation of the stepping framework developed in this thesis. The system's name SeaLion is composed of "Sea" which stands for *Support Environment for ASP* and "Lion" which symbolises the strength and good-naturedness that we aimed to integrate into the environment. A preliminary report on SeaLion was given by Oetsch et al. (Oetsch et al., 2011b, 2013), where initial functionality and an outline on features that are planned to be incorporated are presented. The majority of these plans have been realised in the meanwhile—most importantly, a debugging system that can cope with real-world answer-set programs based on the stepping framework. An up-to-date discussion of SeaLion is given in a recent paper (Busoniu et al., 2013).

The remainder of this chapter is organised as follows. Section 8.1 provides general information about the implementation, including design principles, motivation for the chosen infrastructure, system architecture, and availability. The main features of SeaLion are described in Section 8.2 except for the stepping plugin that is illustrated in greater detail in Section 8.3. Finally, in Section 8.4, we compare the SeaLion system with other integrated development environments for ASP.

8.1 Design, Architecture, and Availability of SeaLion

A major design principle for SeaLion, in alignment with the objectives of the MMDASP project, was interoperability with most of the available popular ASP solvers. That is, SeaLion and its features are not tailored towards a specific ASP solver language but designed generic enough to allow instantiations for differing dialects of input languages. For most features of the IDE, including stepping functionality, such instantiations are provided for the languages of the state-of-the-art solvers Clasp (in conjunction with Gringo) and DLV (compare Sections 3.6.2 and 3.6.3 on the solver languages).

The target audience for SeaLion are software developers new to ASP yet familiar with support tools as used in procedural and object-oriented programming. As a consequence, it was our aim to create an environment that is similar to well-established development tools. In particular, this was one reason why SeaLion is implemented as plugin of the *Eclipse* platform (Eclipse Project, 2014), which is popular among software engineers and can be considered the standard environment for Java development. Arguably, people who are familiar with Eclipse and basic ASP skills will easily adapt to SeaLion. The decision to build on Eclipse rather than writing a stand-alone application from scratch had further benefits. For one, we profit from software reuse as SeaLion makes heavy use of existing functionality that we adapted to our needs. Examples include the text editor framework, source-code highlighting, problem reporting, project management, the undo-redo mechanism, the console view, the refactoring and the navigation frameworks (Outline), and launch configurations. Moreover, much functionality of

Eclipse can be used without any adaptions, e.g., workspace management, the possibility to define working sets, i.e., grouping arbitrary files and resources together, software versioning and revision control (e.g., based on SVN or CVS), and task management.

Regarding the user interface, the aim was to make the usage of SeaLion as smooth as possible. We paid attention that the methods of the features of SeaLion can be performed with as few mouse clicks or other user interaction as necessary and followed Eclipse conventions for shortcuts. Moreover, we wanted to give the ASP developer much freedom in how to use the system. For example, SeaLion avoids functionality that patronises the ASP developer like imposing a certain coding style, i.e., every valid ASP source file should be usable in the IDE. Furthermore, we aimed at interoperability, e.g., through the use of standards or the framework for external tool configurations that allows for using arbitrary external tools, e.g., for postprocessing computed answer sets.

A key aspect in the design of SeaLion is extensibility. While it is implemented as a plugin of Eclipse, the SeaLion implementation follows itself a modular principle, where features can be added by further Eclipse plugins. Moreover, the API framework is tailored to support, on the one hand, further ASP languages with little effort and, on the other hand, allows for embedding future features easily. To this end, we defined a hierarchy of classes and interfaces that represent *program elements*, i.e., fragments of ASP languages. This is done in a way such that we can use common interfaces and base classes for representing similar program elements of different ASP languages. For instance, we have different classes for representing literals of the Gringo language and literals of the DLV language in order to be able to handle subtle differences. For example, as DLV is unaware of conditions, an object of class DLVStandardLiteral has no support for them, whereas a GringoStandardLiteral object keeps a list of condition literals. Substantial differences in other language features, like aggregates, optimisation, and filtering support, are also reflected by different classes for Gringo and DLV, respectively. However, whenever possible, these classes are derived from a common base class or share common interfaces. Therefore, plugins can, for example, use a general interface for aggregate literals to refer to aggregates of both languages. Hence, current and future feature implementations can make use of high-level interfaces and stay independent of the concrete ASP language to a large extent.

Figure 8.1 depicts the technology stack of SeaLion. The *main plugin* of SeaLion as well as several plugins implementing different features of the IDE are embedded in Eclipse. Being a Java application, the system is executed in a virtual machine (Java Runtime Environment). Currently, there are two plugins implementing the support for concrete ASP languages, the *DLV plugin* and the *Gringo plugin*. They contain language specific functionality such as parsers for the respective languages that are based on the ANTLR framework (Parr, 2007), means to translate internal representations back to ASP source code, support for launching solvers, customisation of source code editors, and adaptations of the graphical user interface. The plugins *DLV tools* and *Potassco tools* are convenience packages that automatically install and configure solvers and grounders for use with SeaLion. Current plugins realising features are the following:

- the *stepping plugin* that is discussed in Section 8.3,
- the *Kara plugin* that deals with result visualisation (see Section 8.2.3),
- the *CNL plugin* dealing with a controlled natural language representation of answer sets based on LANA annotations (the LANA language is discussed in Section 8.2.2) that is currently not part of the standard SeaLion distributions,
- the *explanation plugin* and the *Ouroboros plugin* that implement debugging features complementary to stepping (see Section 8.2.4), and

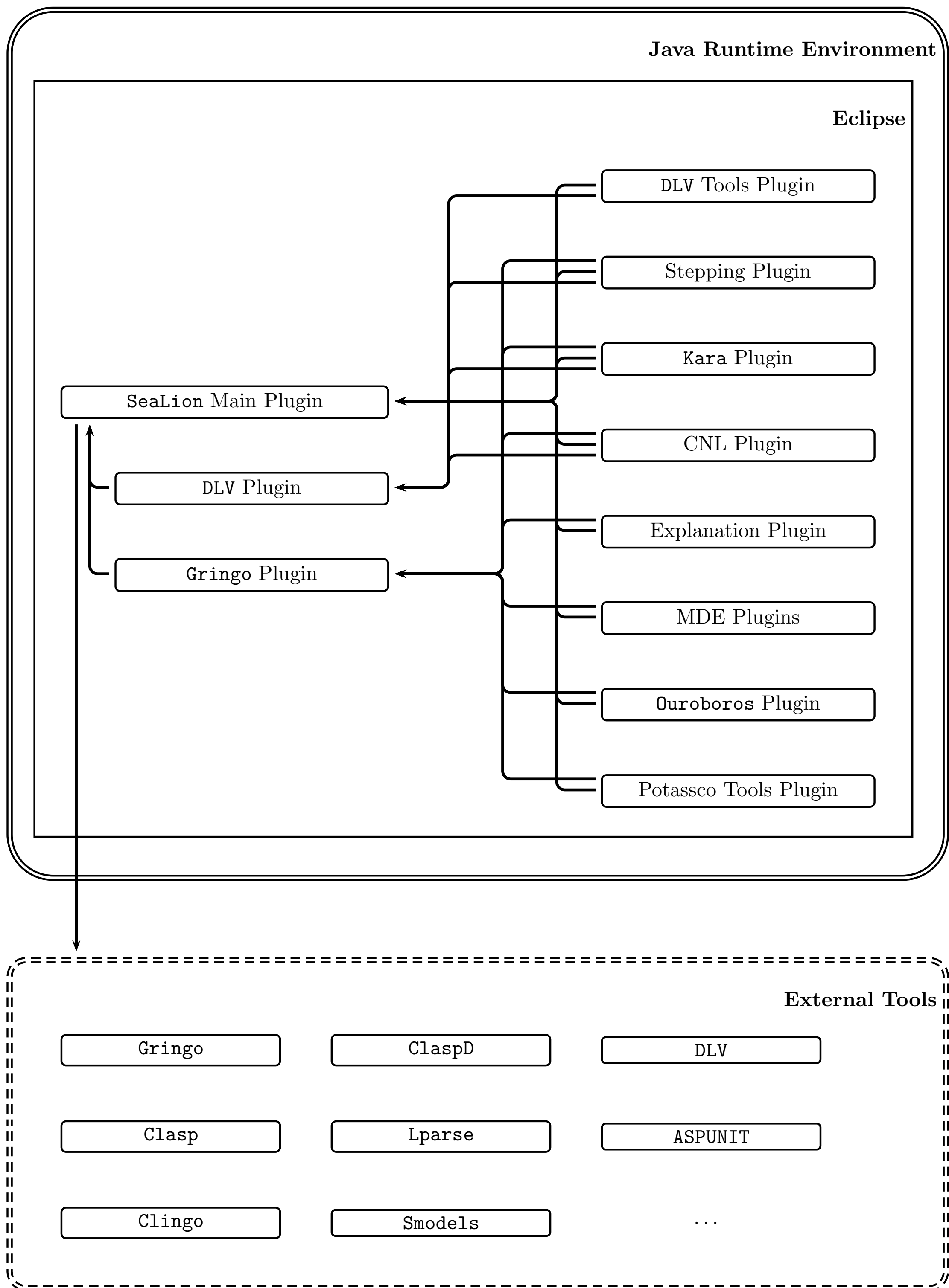

Figure 8.1: Technology Stack of SeaLion.

- multiple plugins that realise the model-driven development approach that is discussed in Section 8.2.2.

An arrow in Figure 8.1 symbolises the usage relation between components of the system. Note that the explanation plugin as well as the plugin for model-driven development do not have links to the DLV plugin. Nevertheless, these modules support the DLV language as their implemen-

tation is based on the abstract data structures representing ASP programs that are provided in the main plugin. In fact, these plugins only use the Gringo plugin as they need special treatment for handling Gringo specific language constructs such as conditions that are not already covered in the main plugin.

SeaLion is free software published under the GNU General Public License version 3. There are two major options to install SeaLion. Users of Eclipse can obtain it using the update site

http://sealion.at/update

using Eclipse's update mechanism. Alternatively, standalone packages of SeaLion are provided for different operating systems and architectures that only require a Java Runtime Environment. Both installation variants support automatic updates and come with pre-configured ASP solvers. Information on installation instructions and links to the source code can be obtained from the project web site

http://www.sealion.at.

8.2 IDE Features

The central element in SeaLion is the *source-code editor* for logic programs. Although there are current endeavours to harmonise solver languages (cf. also Section 3.6), up-to-now the languages of Gringo and DLV differ in their presentation of aggregates and many other small details. That is why the SeaLion editor comes in two variants, one for DLV and one for Gringo. A screenshot of a Gringo source file in SeaLion's editor is given in Figure 8.2.

The editors provide typical conveniences of IDEs, like context-sensitive *syntax highlighting*, *syntax checking*, and *problem reporting*. Terms and predicates appearing in the program are proposed for *autocompletion*. SeaLion also offers functionality for *refactoring* answer-set programs. In particular, we implemented functionality for uniform and safe renaming of predicates, constants, function symbols, and variables throughout a user-defined set of files containing answer-set programs. Once a new name is chosen, the user has the possibility to directly apply the changes implied by renaming or revise them on a preview page. Here, one can inspect the effects file by file where the original as well as the new source code are displayed next to each other and all hypothetical changes are highlighted as depicted in Figure 8.3. An overview of the edited answer-set program is given in Eclipse's *Outline View* in a tree-shaped graphical representation that can be seen in the bottom-right corner of Figure 8.2. Clicking on a node of the tree selects the source code corresponding to the represented program element in the editor such that the programmer can proceed editing there. Another convenient editor feature is the temporary highlighting of code the programmer might be interested in. For instance, if the cursor is positioned over a literal, the positions of all literals of the same predicate in the overall document are indicated.

8.2.1 Solver Interaction

In order to interact with solvers, grounders, and other ASP-related tools, SeaLion has a mechanism for handling *external tools*. One can define *external tool configurations* that specify the path to an executable as well as default command-line parameters. Arbitrary command-line tools are supported; however, there are special configuration types for some programs such as Gringo, Clasp, and DLV. The SeaLion website offers packages that include or automatically install a variety of popular grounders and solvers for which external tool configurations are pre-defined. In addition to external command-line tools, one can also define tool configurations that represent pipes between external tools. This is needed when grounding and solving

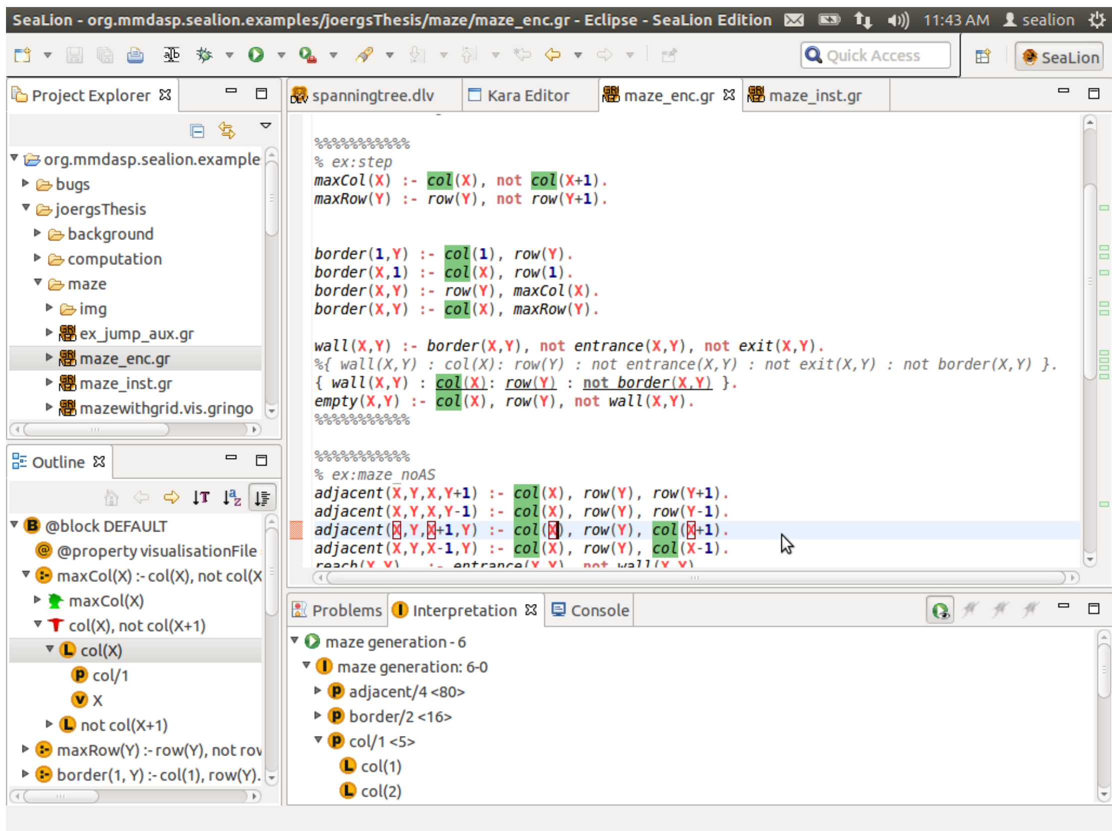

Figure 8.2: A screenshot of `SeaLion`'s editor, the program outline, and the interpretation view.

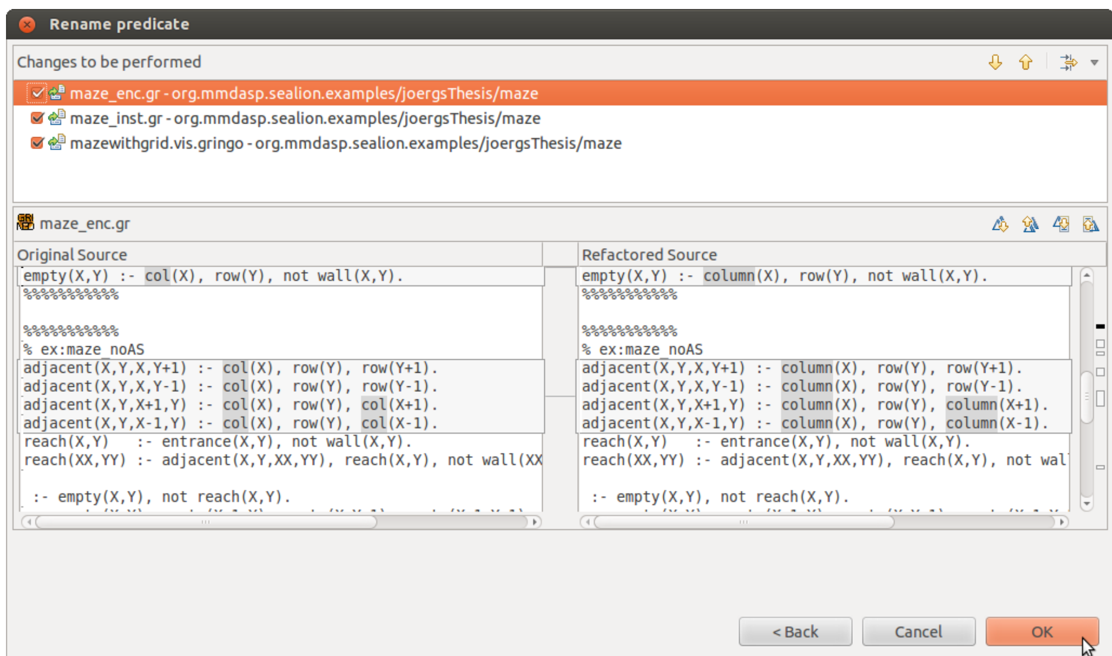

Figure 8.3: Reviewing file changes implied by renaming predicate `col/1` to `column/1`.

are provided by separate executables. For example, one can define two separate tool configurations for `Gringo` and `Clasp` and define a piped tool configuration for using the two tools in a pipe. Pipes of arbitrary length are supported such that arbitrary pre- and post-processing

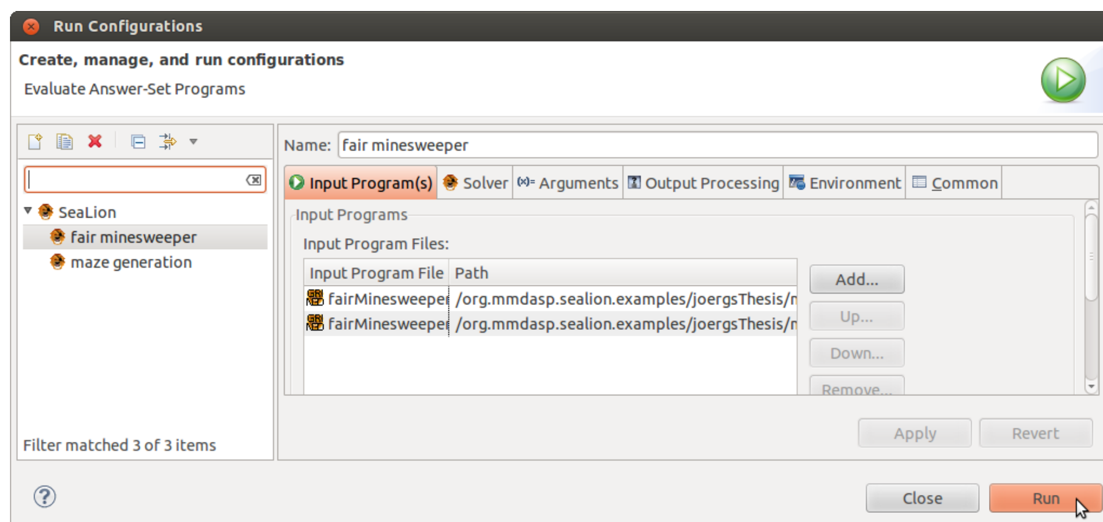

Figure 8.4: Selecting two source files in Eclipse's launch configuration dialog.

can be done when needed. As arbitrary tools can be piped, this mechanism allows for post-processing or handling solver output as needed, e.g., opening external visualisation tools like `IDPDraw` (Wittocx, 2009) or `ASPVIZ` (Cliffe et al., 2008) (Note that `SeaLion` also comes with its own visualisation component `Kara` that is described below). Default solvers for different solver languages can be set in the preference menu of `SeaLion` depending on file content types in the "Content Type Preferences" section.

For executing answer-set solvers and other tools, we make use of Eclipse's *launch configuration framework*, i.e., the user can create re-usable *launch configurations* that define which programs should be executed using particular external-tool configurations, the command-line arguments to use, and other settings. Figure 8.4 shows the page of the launch configuration editor on which input files for a solver invocation can be selected. Moreover, a launch configuration contains information how the output of the solver should be treated. One option is to print the solver output as it is in Eclipse's *console view*. The other option is to parse the resulting answer sets for further use in `SeaLion`. In this case, the answer sets obtained from the solvers are stored in `SeaLion`'s *interpretation view* as well as in the *interpretation comparison view*. In both, interpretations are visualised as expandable trees, where the literals of each interpretation are grouped by their predicates. Compared to a standard textual representation, this way of visualising answer sets provides a well-arranged overview of the individual interpretations. While the interpretation view lists interpretations in rows, the interpretation comparison view places them in columns. By horizontally arranging trees for different interpretations next to each other, it is easy to compare two or more interpretations.

Besides defining launch configurations, `SeaLion` also offers the possibility to invoke a solver right away on a selection of files in the workspace using the default settings of an external tool configuration. This is realised using the so-called *Launch Shortcuts* mechanism of Eclipse. The user selects the files that should be evaluated in the project explorer and selects the `SeaLion` entry of their "Run As" context menu. The entry is available as soon as an external tool configuration is set as default solver for the selected file content type.

8.2.2 LANA Support, Documentation Generation, and Model-Driven Engineering

The editors of `SeaLion` are capable of processing LANA annotations (De Vos et al., 2012a,b). LANA stands for "Language for ANnotating Answer-set programs" and is, as the name sug-

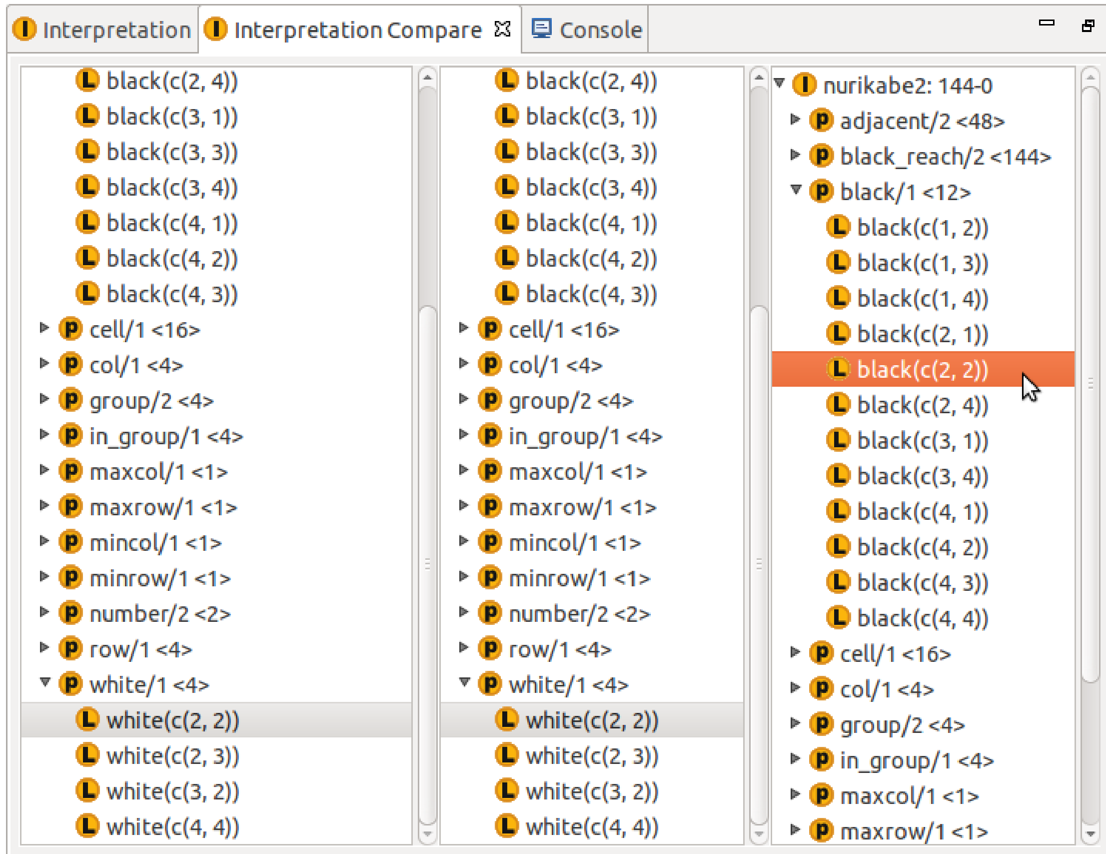

Figure 8.5: `SeaLion`'s interpretation compare view.

gests, a dedicated annotation language for ASP. LANA annotations are specially marked program comments augmenting an answer-set program with meta-information. Being program comments, they are invisible to an ASP solver and therefore not altering the semantics of the program. However, different tools may interpret and use the annotated information to various ends like documentation, testing, verification, code completion, or other development support needs. Moreover, LANA annotations allow for structuring ASP programs by grouping rules into coherent blocks and for specifying, e.g., language signatures, assertions, as well as unit tests for such blocks. `SeaLion` implements different functionality based on LANA. For instance, blocks are reflected in the outline view of `SeaLion` and LANA descriptions of terms and predicates appear next to autocompletion proposals. Moreover, `SeaLion` allows for automatically generated source-code documentation for answer-set programs, similar as tools like `JavaDoc` or `Doxygen` do for other programming languages, based on LANA annotations. For this purpose, the IDE incorporates the `ASPDOC` documentation generator that takes LANA-annotated ASP code as input and produces HTML files as output (De Vos et al., 2012a). The documentation contains descriptions of all (nested) blocks of the answer-set program. Also, a summary of the block structure of the entire answer-set program is presented at the beginning of the documentation to provide an overview. For each block, descriptions of the used atoms and types of involved terms, as well as for assertions, are given. The documentation also includes HTML versions of the program's source code, which can be particularly useful for sharing ASP code online. There are links from the documentation to the source code and vice versa. Likewise, rules for defining pre- and postconditions can be inspected by using respective links. ASP documentation generation can be accessed through Eclipse's export menu.

LANA is also used by `SeaLion`'s modelling framework that adopts techniques from *model-driven engineering* (MDE) (Schmidt, 2006) for supporting the development of answer-set pro-

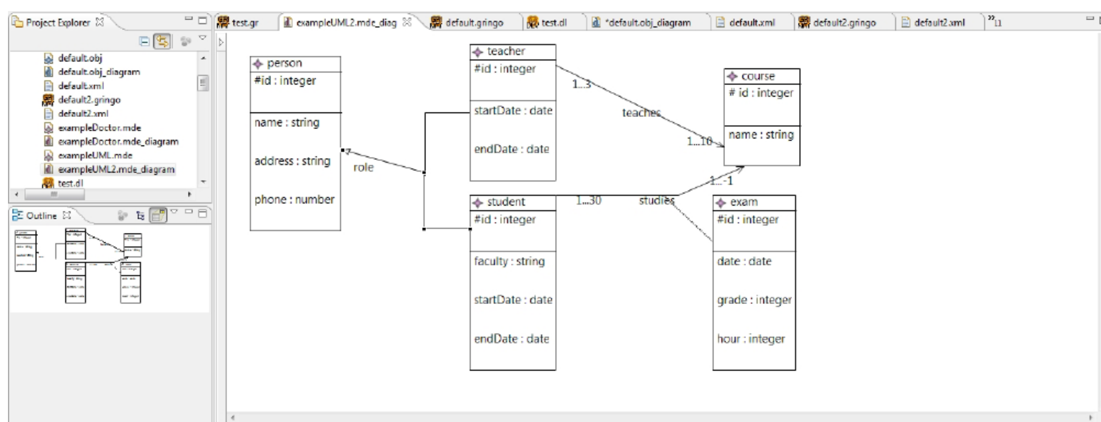

Figure 8.6: Modelling in the SeaLion domain diagram editor.

grams. SeaLion's MDE plugin allows for guiding the ASP development process by graphical models, starting from modelling the problem domain and ending at the visualisation of problem solutions. In object-oriented programming, it is common to model the data structures needed by means of graphical models like *UML class diagrams* (Fowler, 2004). These *domain models* serve as primary development artefacts from which parts of the code can be generated. In SeaLion, a graphical editing framework for modelling the domain of an answer-set program in an extended UML class diagram is implemented. That is, there is a graphical editor (cf. Figure 8.6) in which the user can start the development by creating a UML class diagram that models the problem domain. In a second step, the model can be translated into an ASP source file that contains LANA annotations that define an ASP predicate schema for the problem domain. The translation includes descriptions of the predicates and assertions representing constraints expressed in the model such as cardinalities of associations or disjointedness and completeness of generalisations (e.g., that a person is a man or a woman but not both). Moreover, assertions detecting key violations are generated: we allow for defining *key attributes* in our UML diagrams that are not part of the UML standard as in object-oriented languages class instances are typically uniquely identified by an implicit key that represents an address in memory. The translation from the domain model to an ASP predicate schema is similar in spirit to well-known translations from entity-relationship models to database schemas. That is, adding foreign keys to predicates for relationships, mapping every class to a set of predicates, and mapping all attributes to terms. To give flexibility to the user, the mapping is configurable, e.g., the user may choose by how many and by which predicates a class is represented.

After generating the code file, the developer may proceed with completing his or her answer-set program. In the further course of development, the created domain model can be reused for visualising answer sets that use the generated predicate schemas by means of UML object diagrams. That is, based on the UML class diagram and the configuration of the translation, instances of the classes defined in the model that are encoded in the answer set are detected and displayed (compare Figure 8.7). Likewise, their relationships are visualised. Furthermore, the answer set is automatically checked against LANA assertions that were created, and violations of constraints are highlighted in the UML object diagram (Figure 8.8). To open the diagram, the user opens a corresponding dialog from the context menu of the answer set that should be visualised directly in the interpretation view. As answer sets can become very large, it is also possible to pre-select an interesting subset of the answer set. In this case, only instances are shown in the diagram whose keys appear in the selected atoms.

Figure 8.7: A UML object diagram based on the model of Figure 8.6.

Figure 8.8: Another UML object diagram based on the model of Figure 8.6. Concepts and relations displayed in red indicate different violations of the domain constraints in the visualised interpretation.

8.2.3 Visualisation and Visual Editing of Answer Sets

Visualisation of answer sets is provided by the `Kara` plugin of `SeaLion` that can create user-defined graphical representations of interpretations (Kloimüllner et al., 2013; Kloimüllner, 2012,

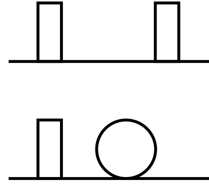

Figure 8.9: The visualisation of interpretation I from Example 43.

2013). Additionally, Kara offers generic visualisations and allows for graphically manipulating interpretations.

User-defined visualisation in Kara is based on ASP itself. Indeed, we follow a similar approach as the tools ASPVIZ (Cliffe et al., 2008) and IDPDraw (Wittocx, 2009). We next describe this method on an abstract level.

Assume we want to visualise an interpretation I that is defined over a first-order alphabet $\mathscr{A}$. We join I, interpreted as a set of facts, with a visualisation program V that is defined over $\mathscr{A}' \supset \mathscr{A}$, where $\mathscr{A}'$ may contain auxiliary predicates and function symbols, as well as predicates from a fixed set $\mathcal{P}_v$ of reserved *visualisation predicates* that vary for the three tools.

The rules in V are used to derive different atoms with predicates from $\mathcal{P}_v$, depending on I, that control the individual graphical elements of the resulting visualisation including their presence or absence, position, and all other properties. An actual visualisation is obtained by post-processing an answer set I_v of $V \cup I$ that is projected to the predicates in $\mathcal{P}_v$. We refer to I_v as a *visualisation answer set* for I. Note that since V is an arbitrary answer-set program it might be non-deterministic in the sense that multiple visualisation answer sets may exist. In the current implementation only one of them is used for visualisation.

The language allows for high-level graphical specifications, supporting graph structures, grids, and relative positioning of graphical elements. Next, we give a simple example of a visualisation program.

Example 43. Assume we deal with a domain program whose answer sets correspond to arrangements of items on two shelves. Consider interpretation I consisting of the atoms

```
book(s1,1). book(s1,3). book(s2,1). globe(s2,2).
```

stating that two books are located on shelf s1 in positions 1 and 3 and that there is another book and a globe on shelf s2 in positions 1 and 2, respectively. The goal is to create a simple graphical representation of this and similar interpretations, depicting the two shelves as two lines, each book as a rectangle, and globes as circles. Consider the following visualisation program:

```
% Rule 1-2: static lines representing shelves
visline(shelf1_line, 10, 40, 80, 40, 0).
visline(shelf2_line, 10, 80, 80, 80, 0).

% Rule 3-5: display books
visrect(f(S,P), 20, 8) :- book(S, P).
visposition(f(s1,P), 20 * P, 20, 0) :- book(s1, P).
visposition(f(s2,P), 20 * P, 60, 0) :- book(s2, P).

% Rule 6-8: display globes
visellipse(f(S,P), 20, 20) :- globe(S, P).
visposition(f(s1,P), 20 * P, 20, 0) :- globe(s1, P).
visposition(f(s2,P), 20 * P, 60, 0) :- globe(s2, P).
```

Rules 1 and 2 create two lines with the identifiers `shelf1_line` and `shelf2_line`, representing the top and bottom shelf. The second to fifth arguments of `visline/6` represent the origin and the target coordinates of the line.[1] The last argument of `visline/6` is a z-coordinate determining which graphical element is visible in case two or more overlap. Rule 3 generates the rectangles representing books, and Rules 4 and 5 determine their position depending on the shelf and the position given in the interpretation. Likewise, Rules 6 to 8 generate and position globes. The resulting visualisation of I is depicted in Figure 8.9. ■

Note that the first argument of each visualisation predicate is a unique identifier for the respective graphical element. By making use of function symbols with variables, like `f(S,P)` in Rule 3 above, these labels are not limited to constants in the visualisation program but can be generated on the fly, depending on the interpretation to visualise. While some visualisation predicates, like `visline/6`, `visrect/3`, and `visellipse/3`, define graphical elements, others, e.g., `visposition/4`, are used to change properties of the elements, referring to them by their respective identifiers. An exhaustive list of visualisation predicates available in `Kara` is given in Appendix A.

Example 44. `Kara` was also used for creating several visualisations for this thesis, including those for the maze generation problem introduced in Section 7.1.1 and the fair minesweeper problem from Section 7.1.2. We used the following visualisation problem for maze generation. Note that it makes use of external image files.

```
#const cellSize=60.

% Compute Nr of Cols and Rows
maxCol(X) :- col(X),not col(X+1).
maxRow(Y) :- row(Y),not row(Y+1).

% Draw lines as frame for grid
visline(frame(1),4,4,MAXROW*cellSize+5, 4,2) :- maxRow(MAXROW).
visline(frame(2),
        4,MAXCOL*cellSize+5,
        MAXROW*cellSize+5,MAXCOL*cellSize+5,
        2) :- maxRow(MAXCOL),maxRow(MAXROW).
visline(frame(3),4,4,4,MAXCOL*cellSize+5,2) :- maxCol(MAXCOL).
visline(frame(4),
        MAXROW*cellSize+5,4,
        MAXROW*cellSize+5,
        MAXCOL*cellSize+5,2) :- maxCol(MAXCOL), maxRow(MAXROW).

% Set colour of frame lines
viscolor(frame(1..4), darkgray).

% Set up the grid
visgrid(maze,
        MAXROW, MAXCOL,
        MAXROW*cellSize+5,
        MAXCOL*cellSize+5) :- maxCol(MAXCOL), maxRow(MAXROW).
visposition(maze,0,0,1).
```

[1] The origin of the coordinate system is at the top-left corner of the illustration window with the x-axis pointing to the right and the y-axis pointing downwards.

```
% Define image for wall cells
visimage(wall, "img/wall36.png").
visscale(wall, cellSize, cellSize).

% Define rectangle for empty cells
visrect(empty, cellSize, cellSize) :- empty(_,_).
viscolor(empty, darkgray).

% Define images for entrance and exit
visimage(entrance, "img/entrance36.png").
visscale(entrance, cellSize, cellSize).
visimage(exit, "img/exit36.png").
visscale(exit, cellSize, cellSize).

% Define rectangle for undefined cells
visrect(undefined, cellSize, cellSize) :- undefined(_,_).
visbackgroundcolor(undefined, lightgray).
viscolor(undefined, darkgray).

% Define image for cells that are both wall and empty
% (inconsistent configuration)
visimage(halfwall, "img/halfwall36.png").
visscale(halfwall, cellSize, cellSize).

% Compute undefined cells
undefined(X,Y) :- col(X),row(Y),not wall(X,Y),not empty(X,Y),
                  not entrance(X,Y),not exit(X,Y).

% Fill the cells of the grid
visfillgrid(maze, empty, Y, X) :- empty(X,Y),not wall(X,Y).
visfillgrid(maze, wall, Y, X) :- wall(X,Y),not empty(X,Y).
visfillgrid(maze, halfwall, Y, X) :- empty(X,Y), wall(X,Y).
visfillgrid(maze, entrance, Y, X) :- entrance(X,Y).
visfillgrid(maze, exit, Y, X) :- exit(X,Y).
visfillgrid(maze, undefined, Y, X) :- undefined(X,Y).
```

As argued in the development guidelines in Section 7.7, during writing an answer-set program it often makes sense to have means to visualise also partial or wrong answer sets. Indeed, the maze generation visualisation program above does not only allow for visualising answer sets as, e.g., in the right subfigure of Figure 7.1 on page 88, but also partial solutions as in the left subfigure of Figure 7.1 or interpretations not supposed to be answer sets like the one in Figure 7.8 on page 104. ■

`SeaLion` displays the resulting visual representation of an interpretation in a graphical editor that also allows for manipulating the visualisation. That is, properties such as colours can be manipulated and graphical elements can be re-positioned, deleted, or even created. Such manipulations are useful for two different purposes. First, for fine-tuning the visualisation before saving it as a scalable vector graphic (SVG) by means of the SVG export functionality of `Kara`. Second, modifying the visualisation can be used to obtain a modified version of the visualised interpretation by abductive reasoning. In fact, a feature is implemented that allows for abducing an interpretation that would result in the modified visualisation (Kloimüllner et al., 2013; Kloimüllner, 2013).

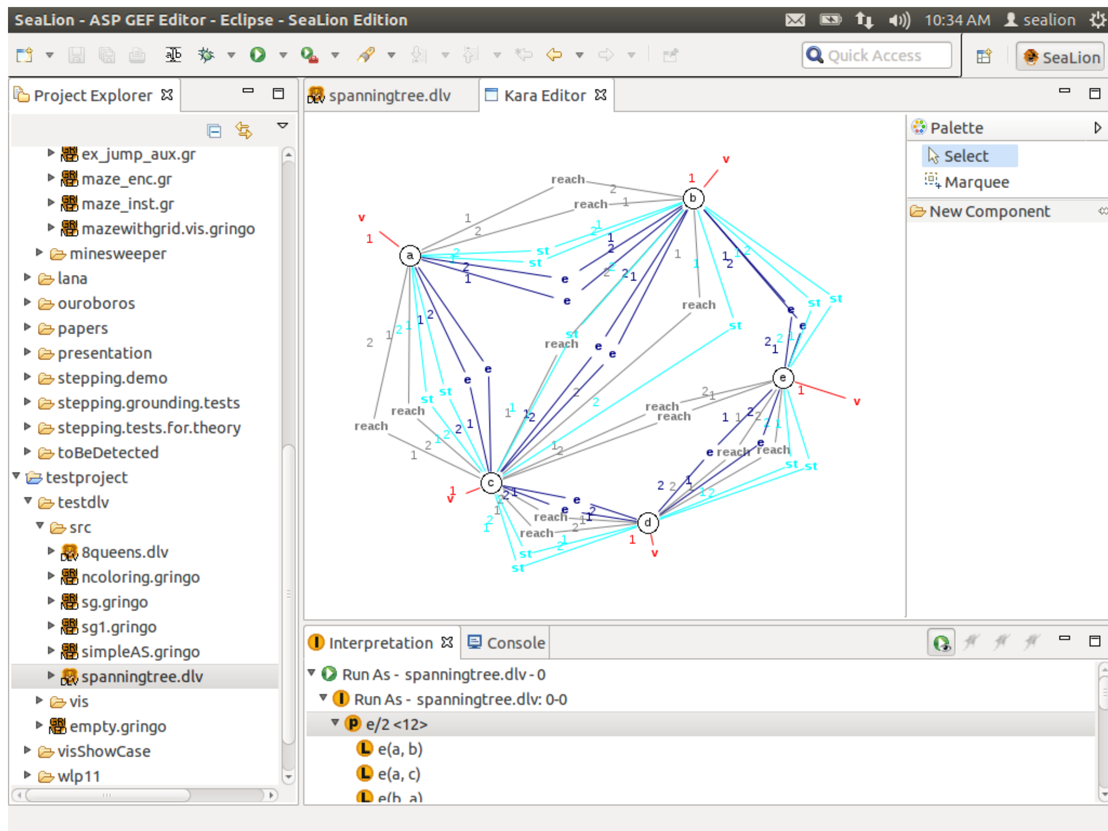

Figure 8.10: `SeaLion`'s visual interpretation editor showing a generic visualisation of the graph colouring interpretation of Example 45 (the layout has been manually optimised).

`Kara` also offers a *generic visualisation* that visualises an arbitrary interpretation without the need for defining a visualisation program. In such a case, the interpretation is represented as a labelled hypergraph. Its nodes are the individuals appearing in the interpretation and the edges represent the literals in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of the edge are used for expressing the term position of the individual. To distinguish between different predicates, each edge has an additional label stating the predicate. Edges of the same predicate are of the same colour.

Example 45. The following atoms form an answer set of an encoding for a graph colouring problem.

```
colour(1, lightblue). colour(2, yellow). colour(3, yellow).
colour(4, red). colour(5, lightblue). colour(6, red)
node(1). node(2). node(3). node(4). node(5). node(6).
edge(1, 2). edge(1, 3). edge(1, 4).
edge(2, 4). edge(2, 5). edge(2, 6).
edge(3, 1). edge(3, 4). edge(3, 5).
edge(4, 1). edge(4, 2). edge(5, 3).
edge(5, 4). edge(5, 6).
edge(6, 2). edge(6, 3). edge(6, 5).
```

The generic visualisation of the interpretation is given in Figure 8.10. ■

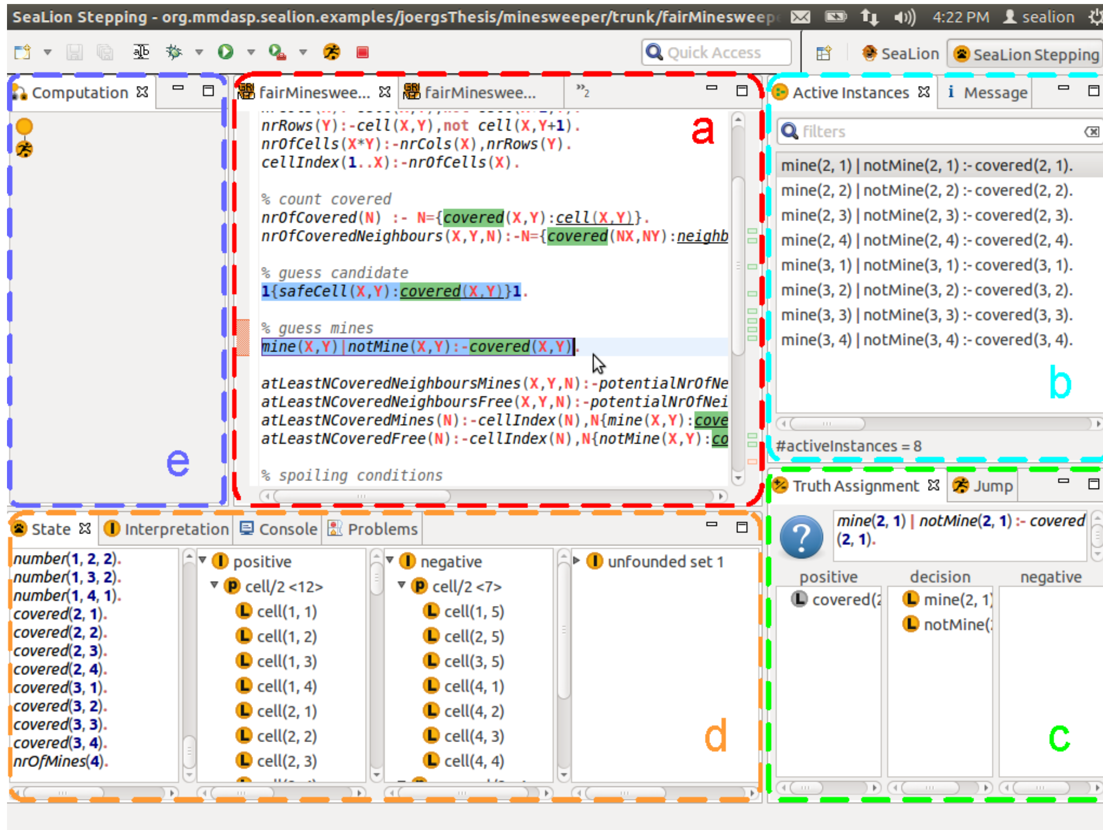

Figure 8.11: SeaLion's stepping view is devided into five areas (a-e).

8.2.4 Debugging Features other than Stepping

Besides stepping, SeaLion implements two further features for debugging purposes. For one, the SeaLion plugin Ouroboros (Frühstück et al., 2013) is a prototype implementation of a debugging method (Polleres et al., 2013) that tackles the question why a given interpretation is not an answer set. The approach extends earlier work addressing this question for propositional (Pührer, 2007; Gebser et al., 2008) and non-ground answer-set programs (Oetsch et al., 2010a) by additional support for weight constraints. The technique is based on ASP meta-programming and determines for a given answer-set program P and a given interpretation I, why I is no answer set of P in terms of unsatisfied rules and unfounded sets. While the Ouroboros plugin provides additional debugging functionality for SeaLion, it also profits from the stepping plugin which can help in building up the interpretation in question.

The other simple yet handy debugging feature of SeaLion, that is complementary to stepping, is the search for a rule that (potentially) derives a particular atom (Eder, 2013). The plugin exploits Eclipse's search framework and supports two settings. First, it can perform a syntactic search for rules in a given program whose head contains a given literal or predicate. Second, if the user also specifies an interpretation, e.g., by clicking on a computed answer set in the interpretation view of SeaLion, the search can be restricted to rules with instances that are active under the given interpretation.

8.3 Practical Stepping in SeaLion

SeaLion implements an environment for stepping as described in Chapter 7 based on the framework of computations introduced in Chapters 5 and 6 using the conditional grounding method (cf. Section 6.3). Although black-box grounding is conceptually simpler as conditional

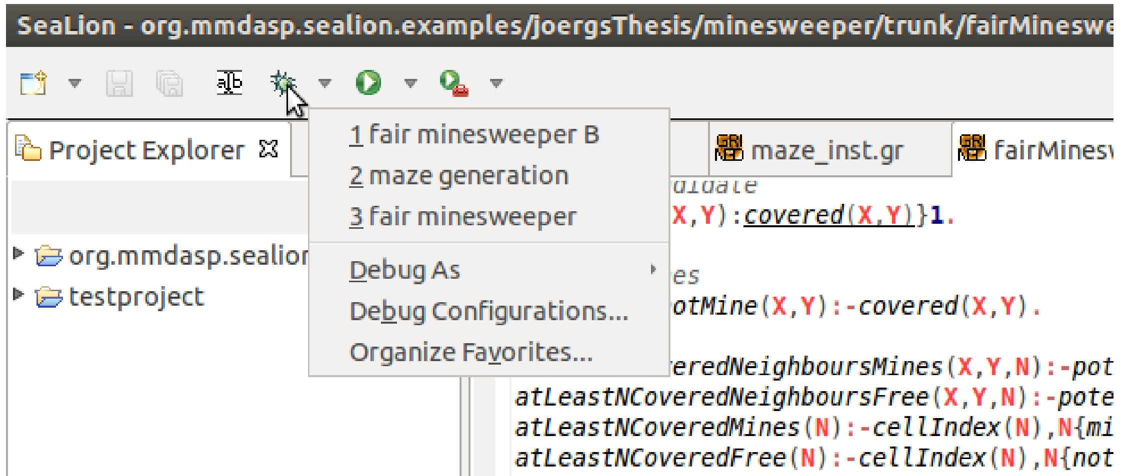

Figure 8.12: Initiating a stepping session by choosing an existing launch configuration.

grounding, we decided to implement only the latter for technical reasons. In particular, the external grounders we exploit in SeaLion do not provide information about which ground rule originates from which non-ground rule—information that is required for black-box grounding. We can do without such a feature for conditional grounding as we only need external grounders for evaluating the partial evaluation function (see Definition 67 on page 78), whereas the grounding function itself is implemented in SeaLion.

The stepping plugin supports the languages of Gringo and DLV and has been implemented by Peter Skočovský under the guidance of the author of this thesis (cf. Section 1.3). While it is the first implementation of the stepping technique for ASP and hence still a prototype, it is tailored for intuitive and user-friendly usage and able to cope with real-world answer-set programs. While the methodological aspects of stepping are discussed in the previous chapter, this section focuses on the user interface of SeaLion's stepping plugin.

8.3.1 Initiating Stepping

A stepping session in SeaLion can be started in a similar fashion as debugging Java programs in Eclipse using the launch configuration framework. SeaLion launch configurations that are used for defining which program files should be run with which solvers (as described in Section 8.2.1) can be re-used as *debug configurations*. That is, the user selects an existing launch configuration, e.g., that of a run that resulted in undesired answer sets, and choses to start stepping the ASP program formed by the input files defined in the configuration. Moreover, as the stepping plugin depends on external tools for grounding, also the grounder and solver settings of the launch configuration are needed. As for launching, stepping can also be initiated using launch shortcuts, i.e., the system tries to automatically generate an appropriate launch configuration when the user selects "Debug As" from the context menu of one or multiple ASP files. Figure 8.12 shows the selection of a pre-existing launch configuration for initiating stepping.

Like many IDEs, Eclipse comes with a multiple document interface in which inner frames, in particular Eclipse editors and views, can be arranged freely by the user. Such configurations can be persisted as *perspectives*. Eclipse plugins often come with default perspectives, i.e., arrangements of views and editors that are tailored to a specific user task in the context of the plugin. Also the stepping plugin has a preconfigured perspective that is opened automatically once a stepping session has been initiated. The next subsection gives an overview of the individual stepping related subframes in this perspective.

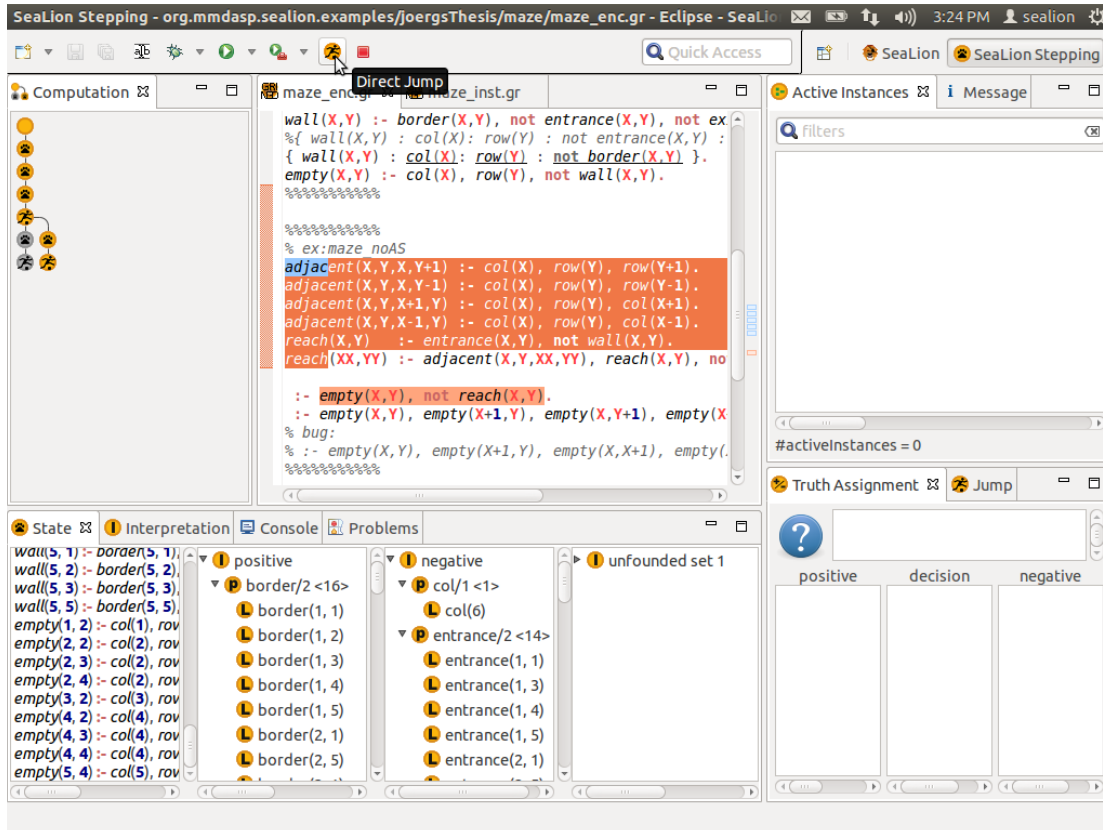

Figure 8.13: Clicking on the “Direct Jump” button for jumping from state S'_6 to state S'_7 in the context of Example 41 on page 103.

8.3.2 Stepping Perspective

Figure 8.11 shows SeaLion in the *stepping perspective*. The illustration distinguishes five regions (marked by supplementary dashed frames and labelled by letters) for which we give an overview in what follows.

The source code editor (Figure 8.11a) is the same as used for writing answer-set programs but extended with additional functionality during stepping mode for the ASP files involved in the active stepping session. In particular it indicates rules with ground instances that are active under the interpretation of the current stepping state. Constraints with active instances are highlighted by a red background (cf. Figure 8.17 on page 134), other rules with active instances have a blue background (as, e.g., in Figure 8.15). The editor remains functional during stepping, i.e., the program can be modified while debugging. Note, however, that the system does not guarantee that the current computation is still a valid computation in case of a modification of the answer-set program after stepping has been initiated. The source code editor is also the starting point for performing a step or a jump as it allows for directly selecting the non-ground rule(s) to be considered in the step or jump in the source code. The choice of non-ground rules corresponds to the initial step in the stepping cycle (see Section 7.5.1). For selecting a single rule for either a step or a jump it suffices to click on the rule. Selecting multiple non-ground rules for a jump comes in two variants. First, if the rules are consecutive in the source code it is sufficient to select the text area containing the rules in the source code editor. In this case a jump is performed by clicking on the “Direct Jump” button in the toolbar (Figure 8.13) or via a similar entry in the “Stepping” menu after the selection of rules. If the rules are non-consecutive, one has to use the second variant for performing a jump. To this end, the user can collect rules in the *jump view* (Figure 8.14) located in area c of Figure 8.11 as the second tab. For adding rules to this view

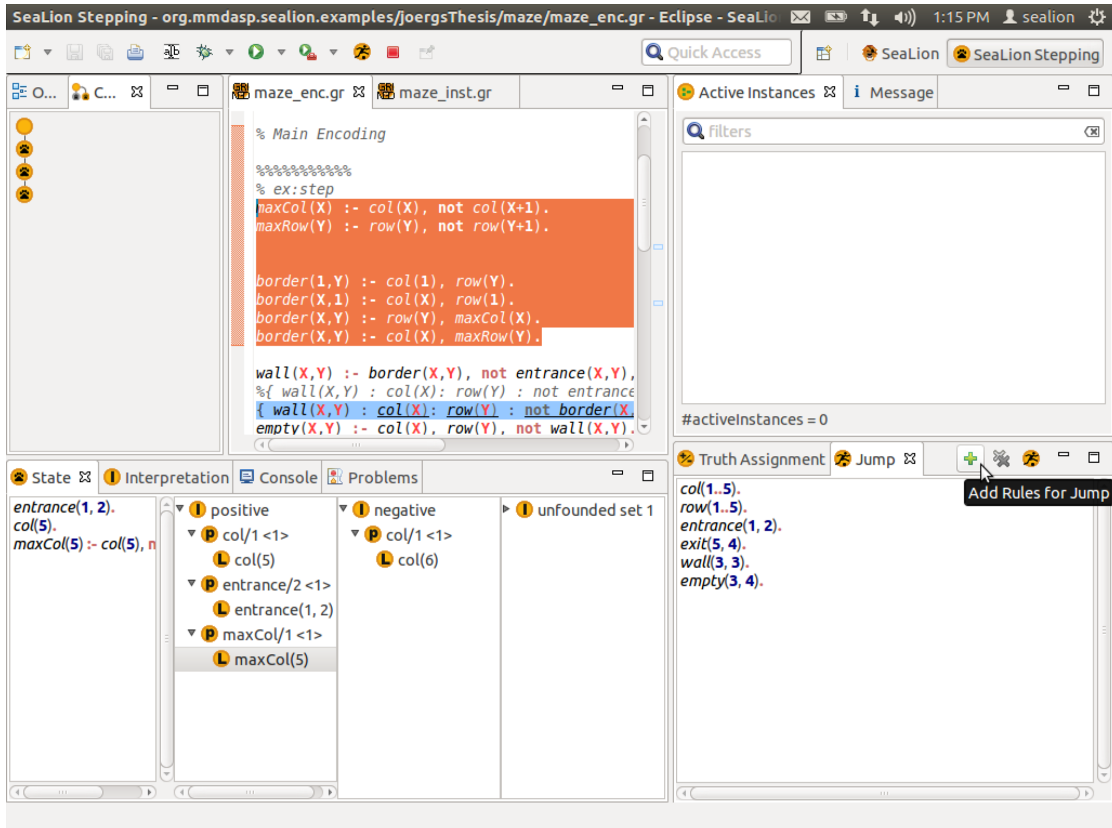

Figure 8.14: Adding rules to the jump view for jumping from state S_3 to state S_4 in the context of Example 38 on page 95.

one has to select them in the source editor and click the respective button in the view. Clicking another button in the jump view initiates the actual jump.

Choosing a ground instance for performing a step is done in the *active instances view* (Figure 8.11b). It contains a list with all active ground instances (with respect to conditional grounding) of the currently selected rule in the source editor. As these are potentially many, the view has a textfield for filtering the list of rules. Filters are given as dot-separated list of variable assignments of the form `X=t` where `X` is a variable of the non-ground rule and `t` is the ground term that the user considers `X` to be assigned to. Only ground instances are listed that obey all variable substitutions of the entered filters.

Once a rule instance is selected in the active instances view the atoms in the rule's domain are displayed in three lists of the *truth assignment view* (Figure 8.11c). The list in the centre shows atoms whose truth value has not already been determined in the current state. The user can decide whether they should be true, respectively false, in the next step by putting them into the list on the left, respectively, on the right. These atoms can be transferred between the lists by using keyboard cursors or drag-and-drop (Figure 8.15). After the truth value has been decided for all the atoms of the rule instance and only in case that the truth assignment leads to a valid successor state (cf. Definition 52 on page 58), a button labelled "Step" appears. Clicking this button computes the new state.

The state view (Figure 8.11d) shows the current stepping state of the debugging session. Hence, it is updated after every step or jump. It comprises four areas, corresponding to the components of the state (cf. Definition 50 on page 57), the list of active rules instances, a tree-shaped representation of the atoms considered true, a tree-shaped representation of the atoms considered false, both in a similar graphical representation as that of interpretations in the interpretation view, and an area displaying the unfounded sets in a similar way. The sets of atoms

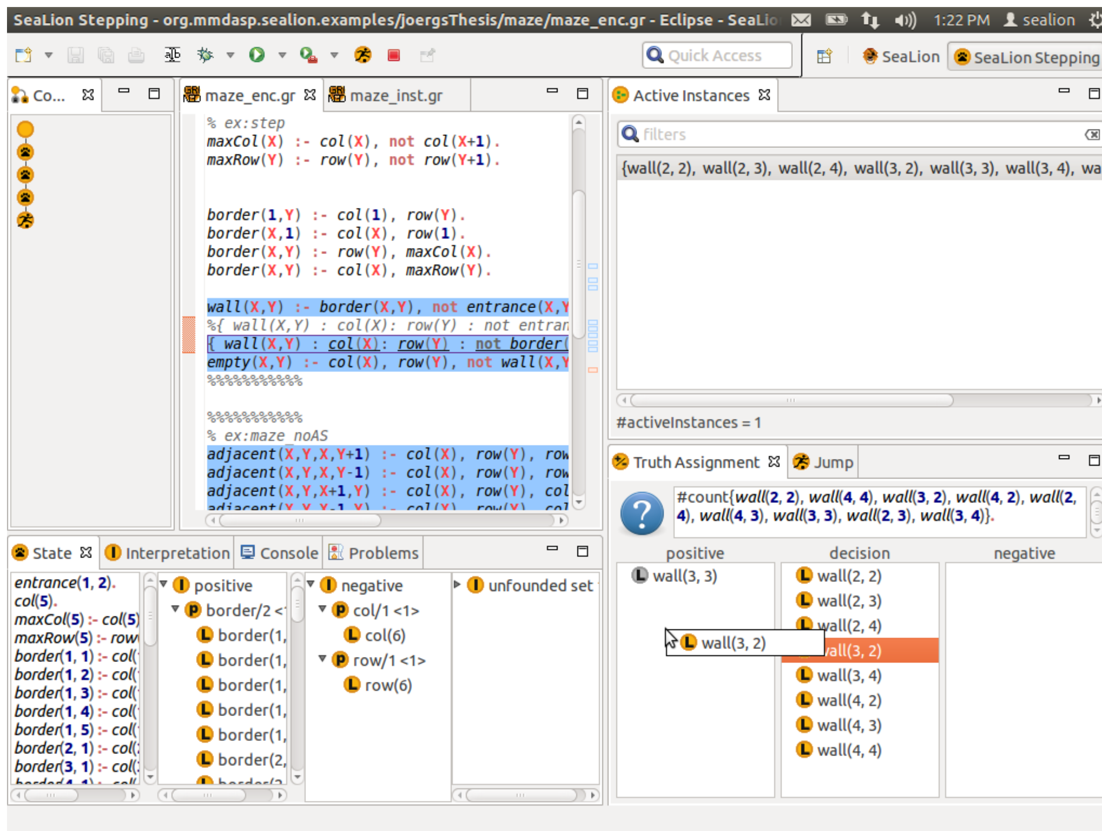

Figure 8.15: Deciding to consider atom `wall(3,2)` to be true by dragging it from the middle list of atoms in the truth assignment view and dropping it in the left list. The current state of the stepping session is S_4 from Example 39 on page 101 and the step prepared is that to state S_5. Note that atom `wall(3,3)` is greyed in the list of positive atoms. Greyed atoms in the positive or negative list cannot be be dragged away from there again because their truth value is already considered positive, respectively negative, in the current state (here S_4). Note that a step can only be completed once the truth value has been decided for all atoms in the rule instance.

displayed in this view can also be visualised using Kara (via options in the context menu). This is a handy feature as it allows to easily monitor the evolvement of the interpretation that is build up during the stepping session, provided the user specified an appropriate visualisation program.

Finally, the computation view (Figure 8.11e) gives an overview of the steps and jumps performed so far. Importantly, the view implements an undo-redo mechanism. That is, by clicking on one of the nodes displayed in the view, representing a previous step or jump, the computation can be reset to the state after this step or jump has been performed. Moreover, after performing an undo operation, the undone computation is not lost but becomes an inactive branch of the tree-shaped representation of steps and jumps. Thus, one can immediately jump back to any state that has been reached in the stepping session by clicking on a respective node in the tree (Figure 8.16).

Mismatches between the users intentions (reflected in the current stepping state) and the actual semantics of the program can be detected in different parts of the stepping perspective. If the user thinks a rule instance should be active but it is not, this can already be seen in the source code editor if the non-ground version of the rule does not have any active instance. Then, the rule is not highlighted in the editor. If the non-ground version does have active instances but not the one the user has in mind, this can be detected after clicking on the non-ground rule if they are missing in the active instances view. The computation is stuck if only rules are highlighted in the source editor that are constraints (cf. Figure 8.17) or for all of its instances, no truth assignment

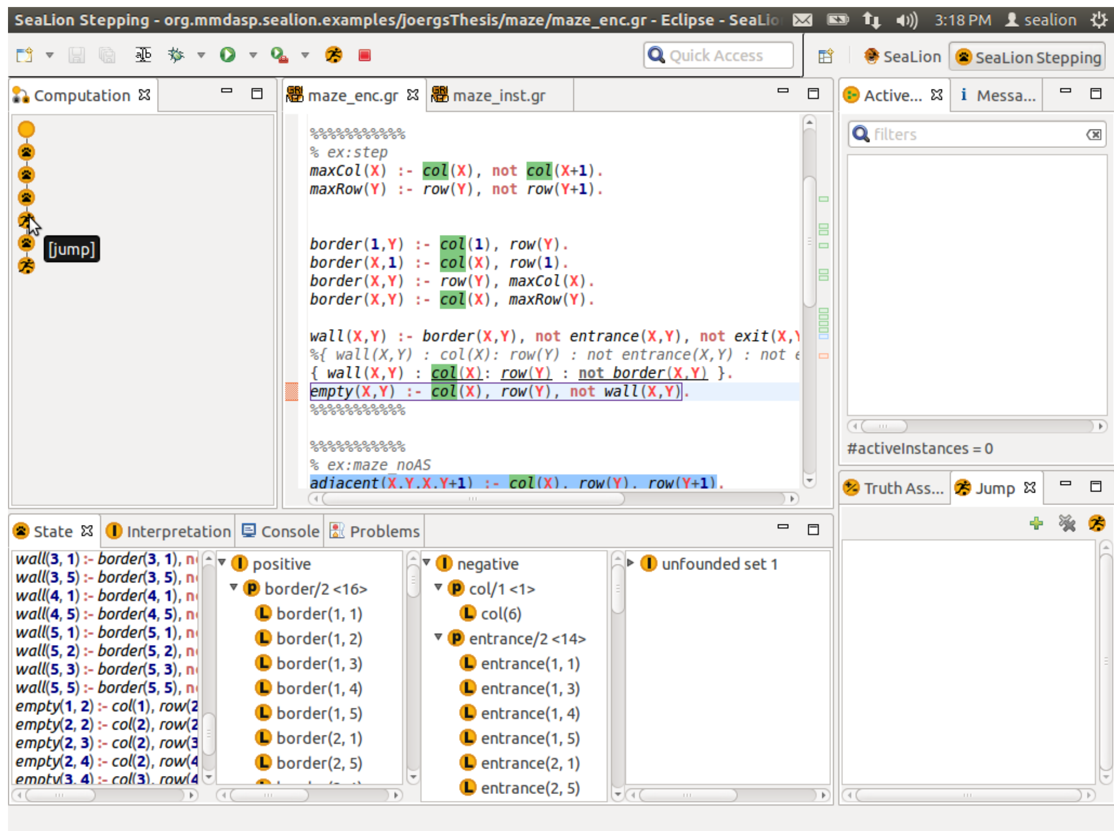

Figure 8.16: In SeaLion, we can retract the computation to a previous state by clicking on the node in the computation view representing the step or jump that created the target state. The screenshot shows reverting the last two states in Example 41 on page 103, where we reused a part of the computation $S_1, \ldots, S_6$ for exploring an alternative setting starting from S_4. The final state of the alternative computation presented in the example is depicted in Figure 8.17.

can be established such that the "Step" button appears. Finally, if no further rule is highlighted and there is no non-empty unfounded set visible in the state view, the atoms considered positive form an answer set of the overall program. If there are further unfounded sets, the user sees that the constructed interpretation is not stable. The unfounded sets indicate which atoms would need external support (see Figure 8.18).

8.4 Comparison of SeaLion to other IDEs for ASP

Besides SeaLion, also other systems have been proposed that provide support for writing answer-set programs. The first two systems in this respect that targeted on ASP languages, APE (Sureshkumar et al., 2007) and VisualDLV (Perri et al., 2007), have been presented at the first international workshop on software engineering for answer-set programming (SEA'07) and offer basic IDE functionality.

To begin with, the tool APE has been developed at the University of Bath as a student project (Sureshkumar, 2006). It is—like SeaLion—implemented as an Eclipse plugin. APE supports the language of the grounder Lparse (Syrjänen, 2002) and hence a sub-language of Gringo (cf. Section 3.6.2). The system offers syntax highlighting, syntax checking, a program outline, auto completion, and launch configurations. The output of the solver can be displayed in a console view or piped to an external script. These functionalities are also available in SeaLion. Additionally, APE has a feature to display the predicate dependency graph of a

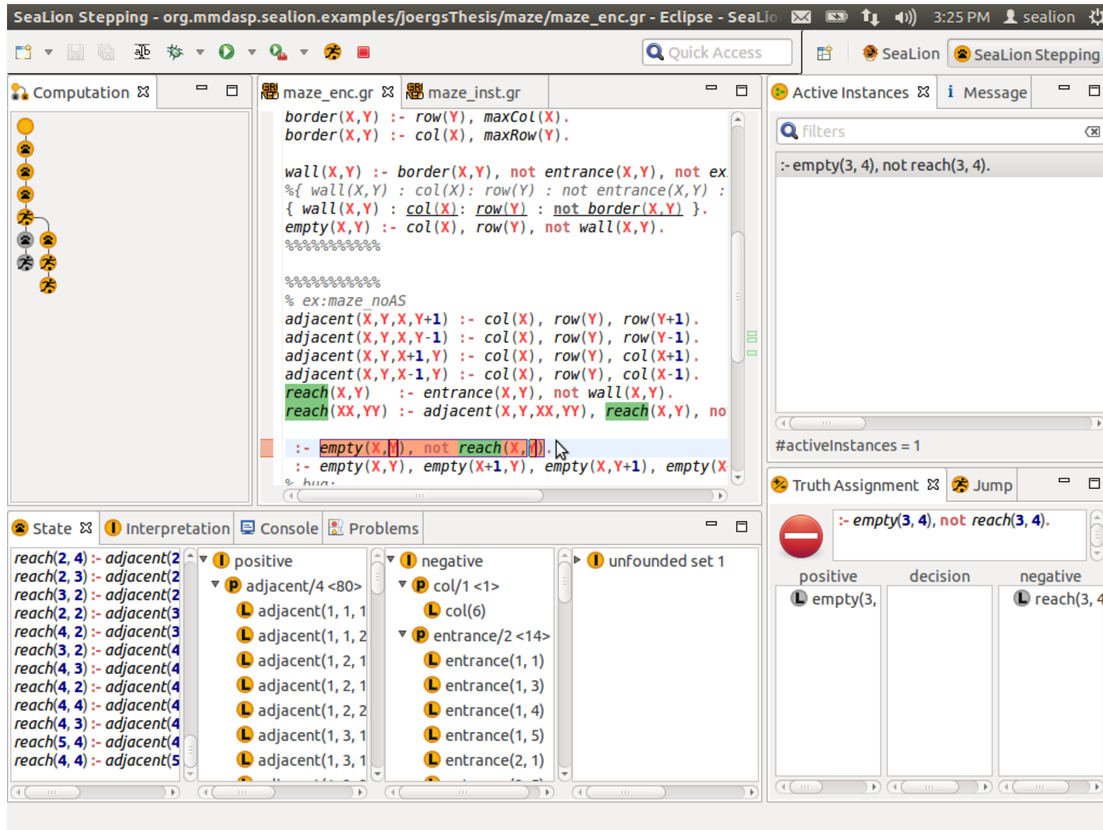

Figure 8.17: The computation is stuck is state S_7' from Example 41 on page 103 as only a constraint has active instances (highlighted in red in the source editor). The "No entry" symbol in the truth assignment view indicates that the instance is not satisfied under the current state. Note that the tree in the computation view has an inactive branch. The current computation (nodes with yellows background) is an alternative branch for the computation in Figure 8.16. Clicking on the final greyed branch would set $S_1, \ldots, S_6$ to be the active computation again.

program.

VisualDLV has been developed at the University of Calabria and is tailored towards the DLV solver. It is a standalone tool with a source code editor that also has syntax checking and auto completion for DLV programs. In addition to standard DLV, the tool is compatible with its two extensions DLV^{DB} and DLV^{IO} that allow for interacting with databases. Here, VisualDLV provides a limited SQL front-end to display and edit database tables and allows for generating auxiliary directives that specify the solver's connection to the database.

A more advanced IDE for DLV is ASPIDE, a recent standalone system (Febbraro et al., 2011) that builds on previous tools (Febbraro et al., 2010; Calimeri et al., 2009; Gebser et al., 2009b). Among the IDEs for ASP that we are aware of, ASPIDE is the closest to SeaLion regarding richness of features. Besides the basic functionalities (syntax highlighting, syntax checking, and code completion), one nice feature of ASPIDE is the support of customised code templates (Calimeri and Ianni, 2006). Moreover, ASPIDE provides a framework for unit tests (Febbraro et al., 2013) that is similar in spirit to unit testing with our annotation language LANA (De Vos et al., 2012a). Like SeaLion, ASPIDE supports refactoring of predicates and variables and allows for embedding external scripts for output processing. ASPIDE comes with a visual program editor, i.e., rules can be generated using a graphical interface. We do not aim for comprehensive visual source-code editing in SeaLion but consider the use of customisable program templates that allow for expressing common programming patterns in future releases of SeaLion. Unfortunately, the profiling component of the IDE (Calimeri et al., 2009), that

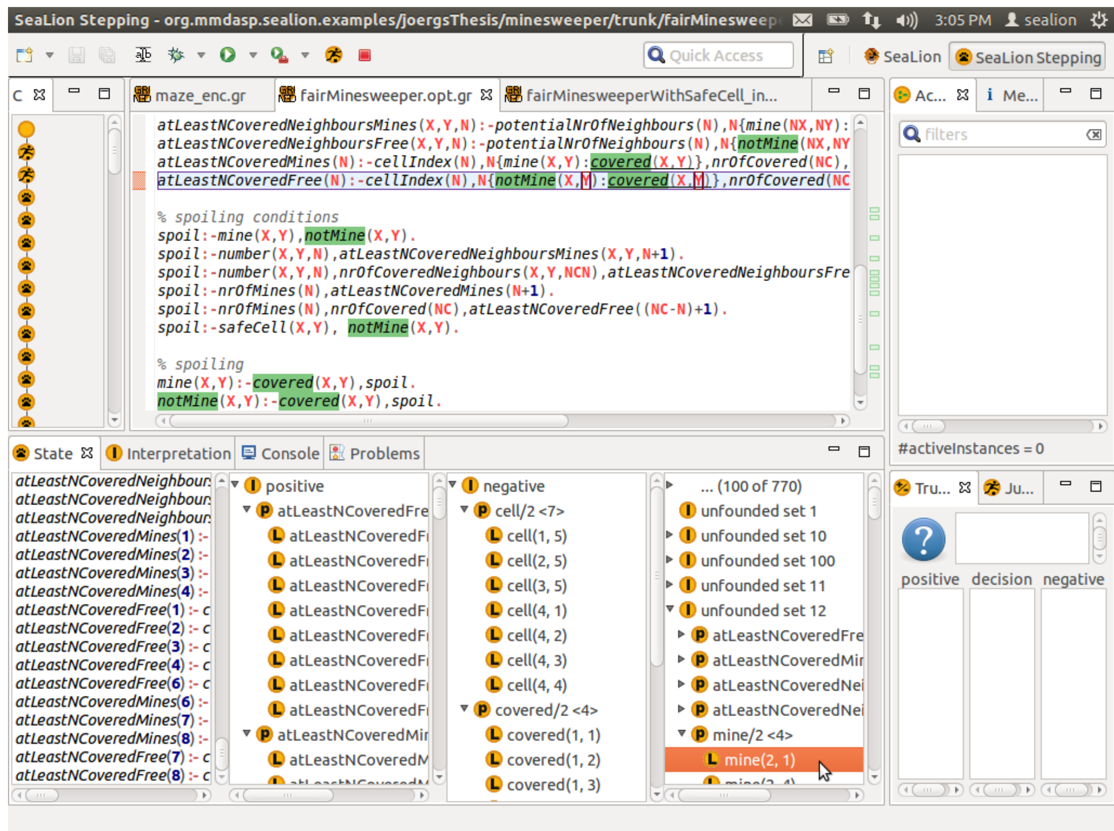

Figure 8.18: In state S_2 of Example 42 on page 106, no further rule has active instances. However, interpretation I_{S_2} is no answer set as there are 770 unfounded sets left. SeaLion only shows the first 100 unfounded sets that were computed because the respective GUI elements would draw much working memory.

is closely linked with DLV, is not publicly available. The developers of ASPIDE integrated the early prototype tool Spock (Brain et al., 2007a; Gebser et al., 2009b) that realises two debugging approaches for a basic language fragment (cf. Section 2.3).

A further system worth mentioning is iGrom (Koziarkiewicz, 2011) that has been developed as a student project at Vienna University of Technology independently from SeaLion. It is based on Eclipse as well and provides basic syntax highlighting and syntax checking for the languages of both Lparse and DLV. A speciality of iGrom is the support for the front-end languages for planning and diagnosis of DLV.

SeaLion and iGrom are the only IDEs that support both the Lparse/Gringo and the DLV language families. All the systems that we discussed provide some form of syntax checking and allow for launching ASP solvers. Built-in visualisation of answer sets, documentation generation, and support for model-driven engineering is only available in SeaLion. Likewise, no other IDE integrates a practical debugging approach for non-ground answer-set programs. Note that the model-based engineering plugin of SeaLion is a refined follow-up project of the VIDEAS system (Oetsch et al., 2011a) that used ER diagrams to model domains of answer-set programs.

Finally, we discuss systems that are not targeted towards core ASP but towards related knowledge representation languages. Among those are two commercial development environments for ontology reasoning on top of logic programming, OntoStudio (Weiten, 2009) and OntoDLV (Ricca et al., 2009). OntoStudio allows for modelling ontologies using semantic web languages (OWL,RDF(S),RIF) and the ObjectLogic language, an extension of frame logic (Kifer et al., 1995). The formalism is based on normal logic programs under the well-

founded semantics (Van Gelder et al., 1991). OntoStudio is Eclipse-based and has a strong focus on visual modelling. OntoDLV is an environment for modelling ontologies using OntoDLP, an extension of the DLV language with ontology features like classes, inheritance, relations and axioms and also focuses on interoperability with OWL and supports connections to relational databases, similar as other DLV based environments (VisualDLV and ASPIDE). Several IDEs for PROLOG have been developed, e.g., PDT (Kniesel et al., 2014), J-Prolog (Bartram, 2004), ProDT (Cancinos, 2012), ProClipse (Bendisposto et al., 2008), and the proprietary Amzi!Prolog (Amzi! inc, 2011). Among those only PDT seems to be under active development. With the exception of J-Prolog, all this systems are Eclipse plugins. Some of this environments provide front-ends for the tracing based debugging system of the PROLOG interpreter.

9 Summary and Conclusion

This chapter summarises the results of this work and discusses potential topics for future work.

9.1 Summary

In this thesis, we introduced the stepping technique for ASP that can be used for debugging and analysis of answer-set programs. Like stepping in imperative programming, where the effects of consecutive statements are watched by the user, our stepping technique allows for monitoring the effect of rules added in a step-by-step session. In contrast to the imperative setting, stepping in our sense is interactive in the sense that a user decides in which direction to branch, by choosing which rule to consider next and which truth values its atoms should be assigned. On the one hand, this breaks a general problem of debugging in ASP, namely how to find the cause for an error, into small pieces. On the other hand, this user interaction allows for focussing on interesting parts of the debugging search space from the beginning. This is in contrast to the imperative setting, where the order in which statements are considered in a debugging session is fixed. Nevertheless, also in our setting, the choice of the next rule is not entirely arbitrary, as we require the rule body to be active first. Stepping-based debuggers for procedural languages often tackle the problem that many statements need to be considered before coming to an interesting step by ignoring several steps until pre-defined breakpoints are reached. We developed an analogous technique in our approach that we refer to as jumping and that allows the user to consider multiple rules at once.

Besides developing the technical framework for stepping, we also discussed methodological aspects, thereby giving guidelines for the usage of the technique, and for setting the latter in the big picture of ASP development.

We have discussed `SeaLion`, an integrated development environment for ASP, that has been developed in the context of the same research project as this thesis and comes with an implementation of the stepping technique. We describe the design and the implementation of the system, give an overview of its features, and show how stepping can be realised in practise using `SeaLion`.

The main goal of this work was the realisation of a ready-to-use debugging approach for ASP that is compatible with the rich languages of modern ASP solvers. In order to achieve it, in the form of the stepping technique, several problems had to be solved on the way whose solutions can be considered interesting in themselves.

For one, it was necessary to overcome the differences between different solver languages. Here, we could make use of abstract-constraint programs as a well-established formalism for representing special literals such as aggregates or weight constraints. However, none of the existing semantics defined for abstract-constraint programs was sufficient to cover the semantics of all of our three solvers languages of interest, the `Gringo` language, the `DLV` language, and the language of `DLVHEX`. Therefore, we extended the FLP-semantics to disjunctive abstract-constraint programs with arbitrary abstract constraints in rule heads. The resulting formalism, for which we studied several properties, can be seen as an abstraction of ground programs in our three target solver languages.

Not only the gap between languages of different solvers but also the differences between solver languages and formal ASP languages needed to be bridged. In this respect, a key difference is the grounding step, where in formal ASP languages every rule is replaced with its instances resulting from naïve grounding, i.e., all combinations of rules obtained by substituting variables by all available terms. The grounding components of real answer-set solvers however apply many different forms of simplifications and even create new terms, e.g., by evaluating interpreted function symbols or accessing external knowledge sources. Also here, we decided to solve the problem by means of abstraction. That is, we defined a class of abstract non-ground programs and two types of abstractions of the grounding step that turn abstract non-ground into abstract-constraint programs. By developing our stepping framework to work for these abstractions, it can easily be applied to concrete solver languages.

The framework of computations developed in Chapters 5 and 6 provides the formal base for stepping. From the user perspective, stepping boils down to selecting one active rule instance at a time and fixing the truth assignment for performing a step or simply selecting a set of rules in the case of a jump. We showed properties that guarantee that this procedure is semantically adequate, i.e., computations fully characterise the fixpoint-based definition of answer sets. An analysis of the computational complexity of our semantics showed that checking answer-set existence for abstract-constraint programs is Σ_2^P-complete. Due to this fact, unless the polynomial hierarchy collapses, the problem is not in NP. To capture the full complexity of the semantics, computations keep track of unfounded sets. In order to eventually reach an answer-set of the program, each non-empty unfounded set must be eliminated by choosing a respective rule instance providing external support. We singled out a large class of programs in which we can rely on stable computations only, i.e., computations without non-empty unfounded sets, to reach every answer set.

Finally, we also compared the stepping technique, the new semantics for abstract-constraint programs, the framework of computations, as well as the `SeaLion` system with related approaches.

9.2 Outlook

In this section we conclude this work with an outlook on possible future work.

While unstable computation are often not needed, they offer great opportunities for further work. For one, the use of unfounded sets for distinguishing states in unstable computations is a natural first choice for expressing the lack of stability. Arguably, when a user arrives in a state with a non-empty unfounded set, he or she only knows that some external support has to be found for this set but there is no information which atoms of the unfounded sets are the crucial ones. It might be worthwhile to explore alternative representations for unstability information such as elementary loops (Gebser et al., 2011d) that would possibly provide more pinpoint information. This would also require lifting a respective notion to the full language of C-programs first.

Another issue regarding unstable computations that would deserve further attention is that in the current approach jumps can only result in stable states. Thus, unstable states in a computation can only be reached by individual steps at present. Here, it would be interesting to study methods and properties for computations that allow for jumping to states that are not stable.

Regarding the realisation of stepping, we next discuss functionality that could be helpful for stepping which are not yet implemented in `SeaLion`. One such feature would be semi-automatic stepping, i.e., the user could push a button and then the system searches for potential steps for which no further user interaction is required and applies them automatically until an answer set is reached, the computation is stuck, or user interaction is required.

It would also be convenient to have automated checks whether the computation of a debugging session is still a computation for the debugged program after a program update. In this

respect, when the computation for the old version became incompatible, a feature would be advantageous that builds up a computation for the new version that resembles the old one as much as possible. Unlike semi-automatic stepping and compatibility checks for computations which could be implemented without further studies, the latter point gives also rise to further theoretical research.

Further convenient features would be functionality that highlights the truth values of atoms that cause a rule not to be active for a given substitution and methods for predicting whether a rule can become active in the future, i.e., in some continuation of the computation.

We received positive feedback on the usability of stepping from students involved in the development of `SeaLion`. However, these were just a few replies that might be considered biased. Hence, a task left for future work is to carry out a systematic user study for the stepping plugin of `SeaLion`.

We also see potential for future work based on our results beyond stepping. For one, the semantics introduced in Chapter 4 can serve as a basis for extending current answer-set solvers to a richer language as today's solvers do not jointly allow for disjunction and aggregates in the rule heads. In this respect, the characterisations in terms of unfounded sets could be beneficial, as, e.g., the algorithms of `DLV` operates with unfounded sets.

Moreover, our framework of computations for G-programs could be implemented as an ASP solver that grounds the program on-the-fly. In many ASP programs the rules for guessing a solution prevent optimisations in the grounding step which leads to huge groundings that can easily exceed a reasonable amount of resources. These problems could be prevented by solvers that do grounding while solving and solvers that follow this principle have already been proposed (Lefèvre and Nicolas, 2009; Dao-Tran et al., 2012). However, they use only restricted core languages, in particular they do not allow for aggregate literals, weight constraints, or external atoms which became very important in practise. Based on our framework of computations it is in principle easy to develop such a solver in which the choices of the user in the stepping setting are replaced by choice points in which the solver takes a decision. We see potential that such a solver could outperform state-of-the art ASP systems if it came with adequate heuristics and learning techniques.

A Predefined Visualisation Predicates in `Kara`

Atom	Intended meaning
$visellipse(id, height, width)$	Defines an ellipse with specified height and width.
$visrect(id, height, width)$	Defines a rectangle with specified height and width.
$vispolygon(id, x, y, ord)$	Defines a point of a polygon. The ordering defines in which order the defined points are connected with each other.
$visimage(id, path)$	Defines an image given in the specified file.
$visline(id, x_1, y_1, x_2, y_2, z)$	Defines a line between the points (x_1, y_1) and (x_2, y_2).
$visgrid(id, rows, cols, height, width)$	Defines a grid with the specified number of rows and columns; $height$ and $width$ determine the grid size.
$visgraph(id)$	Defines a graph.
$vistext(id, text)$	Defines a text element.
$vislabel(id_g, id_t)$	Sets the text element id_t as a label for graphical element id_g. Labels are supported for the following elements: $visellipse/3$, $visrect/3$, $vispolygon/4$, and $visconnect/3$.
$visisnode(id_n, id_g)$	Adds the graphical element id_n as a node to a graph id_g for automatic layouting. The following elements are supported as nodes: $visrect/3$, $visellipse/3$, $vispolygon/4$, $visimage/2$.
$visscale(id, height, weight)$	Scales an image to the specified height and width.
$visposition(id, x, y, z)$	Puts an element id on the fixed position (x, y, z).
$visfontfamily(id, ff)$	Sets the specified font ff for a text element id.
$visfontsize(id, size)$	Sets the font size $size$ for a text element id.
$visfontstyle(id, style)$	Sets the font style for a text element id to bold or italics.
$viscolor(id, color)$	Sets the foreground colour for the element id.
$visbackgroundcolor(id, color)$	Sets the background colour for the element id.
$visfillgrid(id_g, id_c, row, col)$	Puts element id_c in cell (row, col) of the grid id_g.
$visconnect(id_c, id_{g_1}, id_{g_2})$	Connects two elements, id_{g_1} and id_{g_2}, by a line such that id_{g_1} is the source and id_{g_2} is the target of the connection.
$vissourcedeco(id, deco)$	Sets the source decoration for a connection.
$vistargetdeco(id, deco)$	Sets the target decoration for a connection.
$visleft(id_l, id_r)$	Ensures that the x-coordinate of id_l is less than that of id_r.
$visright(id_r, id_l)$	Ensures that the x-coordinate of id_r is greater than that of id_l.
$visabove(id_t, id_b)$	Ensures that the y-coordinate of id_t is smaller than that of id_b.
$visbelow(id_b, id_t)$	Ensures that the y-coordinate of id_b is greater than that of id_t.
$visinfrontof(id_1, id_2)$	Ensures that the z-coordinate of id_1 is greater than that of id_2.
$vishide(id)$	Hides the element id.
$visdeletable(id)$	Defines that the element id can be deleted in the visual editor.
$viscreatable(id)$	Defines that the element id can be created in the visual editor.
$vischangable(id, prop)$	Defines that the property $prop$ can be changed for the element id in the visual editor.
$vispossiblegridvalues(id, id_e)$	Defines that graphical element id_e is available as possible grid value for a grid id in the visual editor.

Bibliography

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA.

Amzi! inc (2011). Amzi! Prolog. `http://amzi.com/AmziPrologLogicServer/`. [Online; accessed 31-March-2014].

ANSI/IEEE (1983). *Standard Glossary of Software Engineering Terminology*. IEEE, New York, NY, USA.

Apt, K. R., Blair, H. A., and Walker, A. (1988). Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann.

Balduccini, M. and Gelfond, M. (2003). Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461.

Balint, A., Belov, A., Diepold, D., Gerber, S., Järvisalo, M., and Sinz, C., editors (2012). *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki.

Baral, C., Dzifcak, J., and Son, T. C. (2008). Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions. In Fox, D. and Gomes, C. P., editors, *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (AAAI'08), Chicago, Illinois, USA, July 13-17, 2008, pages 818–823. AAAI Press.

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

Baral, C. and Hunsaker, M. (2007). Using the probabilistic logic programming language p-log for causal and counterfactual reasoning and non-naive conditioning. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (IJCAI'07), Hyderabad, India, January 6-12, 2007, pages 243–249.

Bartholomew, M., Lee, J., and Meng, Y. (2011). First-order extension of the FLP stable model semantics via modified circumscription. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence* (IJCAI'11), pages 724–730. AAAI Press.

Bartram, J. (2004). J-prolog editor. `http://www.trix.homepage.t-online.de/JPrologEditor/`. [Online; accessed 31-March-2014].

Beierle, C., Dusso, O., and Kern-Isberner, G. (2005). Using answer set programming for a decision support system. In Baral, C., Greco, G., Leone, N., and Terracina, G., editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'05), Diamante, Italy, September 5-8, 2005, volume 3662 of *Lecture Notes in Computer Science*, pages 374–378. Springer.

Ben-Eliyahu, R. and Dechter, R. (1994). Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87.

Bendisposto, J., Endrijautzki, I., Leuschel, M., and Schneider, D. (2008). A semantics-aware editing environment for prolog in Eclipse. In *Proceedings of the 18th Workshop on Logic-based Methods in Programming Environments* (WLPE'08), Udine, Italy, December 12, 2008.

Berre, D. L., Roussel, O., and Simon, L. (2009). SAT competition 2009. `http://www.satcompetition.org/2009/`. [Online; accessed October 8, 2013].

Boenn, G., Brain, M., De Vos, M., and Fitch, J. (2011). Automatic music composition using answer set programming. *Theory and Practice of Logic Programming*, 11(2-3):397–427.

Bögl, M., Eiter, T., Fink, M., and Schüller, P. (2010). The `MCS-IE` system for explaining inconsistency in multi-context systems. In Janhunen, T. and Niemelä, I., editors, *Proceedings of the 12th European Conference on Logics in Artificial Intelligence* (JELIA'10), volume 6341 of *Lecture Notes in Computer Science*, pages 356–359. Springer.

Brain, M., Cliffe, O., and De Vos, M. (2009). A pragmatic programmer's guide to answer set programming. In De Vos, M. and Schaub, T., editors, *Proceedings of the 2nd International Workshop on Software Engineering for Answer-Set Programming* (SEA'09), Potsdam, Germany, pages 49–63.

Brain, M. and De Vos, M. (2005). Debugging logic programs under the answer set semantics. In De Vos, M. and Provetti, A., editors, *Proceedings of the 3rd International Workshop on Answer Set Programming* (ASP'05), Advances in Theory and Implementation, Bath, UK, September 27-29, 2005, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Brain, M., Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., and Yilmaz, C. (2012). Event-sequence testing using answer-set programming. *International Journal On Advances in Software*, 5(3–4):237–251.

Brain, M., Gebser, M., Puehrer, J., Schaub, T., Tompits, H., and Woltran, S. (2007a). That is illogical Captain! The debugging support tool spock for answer-set programs – System description. In De Vos, M. and Schaub, T., editors, *Proceeding of the 1st International Workshop on Software Engineering for Answer Set Programming* (SEA'07), Tempe, AZ, USA, May 14, 2007, pages 71–85.

Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., and Woltran, S. (2007b). Debugging ASP programs by means of ASP. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'07), Tempe, AZ, USA, May 15-17, 2007, volume 4483 of *Lecture Notes in Computer Science*, pages 31–43, Berlin-Heidelberg, Germany. Springer.

Brewka, G. (2007). Preferences, contexts and answer sets. In Dahl, V. and Niemelä, I., editors, *Proceedings of the 23rd International Conference on Logic Programming* (ICLP'07), Porto, Portugal, September 8-13, 2007, volume 4670 of *Lecture Notes in Computer Science*, page 22. Springer.

Brewka, G., Eiter, T., and Fink, M. (2011). Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In Balduccini, M. and Son, T. C., editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*, pages 233–258. Springer.

Brewka, G., Niemelä, I., and Truszczyński, M. (2008). Preferences and nonmonotonic reasoning. *AI Magazine*, 29(4):69–78.

Busoniu, P.-A. (2013). On supporting the development of answer-set programs using model-driven engineering techniques. Master's thesis, Vienna University of Technology, Vienna, Austria.

Busoniu, P.-A., Oetsch, J., Pührer, J., Skočovský, P., and Tompits, H. (2013). Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, 13(4-5):657–673.

Caballero, R., García-Ruiz, Y., and Sáenz-Pérez, F. (2008). A theoretical framework for the declarative debugging of datalog programs. In Schewe, K.-D. and Thalheim, B., editors, *Revised Selected Papers of the 3rd International Workshop on Semantics in Data and Knowledge Bases* (SDKB'08), Nantes, France, March 29, 2008, volume 4925 of *Lecture Notes in Computer Science*, pages 143–159, Berlin-Heidelberg, Germany. Springer.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. (2012). ASP-Core-2, Input language format. `https://www.mat.unical.it/aspcomp2013/ASPStandardization/`. [Online; accessed October 8, 2013].

Calimeri, F. and Ianni, G. (2006). Template programs for disjunctive logic programming: An operational semantics. *AI Communications*, 19(3):193–206.

Calimeri, F., Leone, N., Ricca, F., and Veltri, P. (2009). A visual tracer for DLV. In De Vos, M. and Schaub, T., editors, *Proceedings of the 2nd International Workshop on Software Engineering for Answer-Set Programming* (SEA'09), Potsdam, Germany, pages 79–93.

Cancinos, C. (2012). Prolog Development Tools - ProDT. `http://prodevtools.sourceforge.net/`. [Online; accessed 31-March-2014].

Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., and Scarcello, F. (1997). The DLV system: Model generator and advanced frontends (system description). In *Proceedings of the 12th Workshop on Logic Programming* (WLP'97), pages 128–137.

Clark, K. L. (1978). Negation as failure. In Gallaire, H. and Minker, J., editors, *Logic and Data Bases*, pages 292–322, New York. Plenum Press.

Cliffe, O., De Vos, M., Brain, M., and Padget, J. A. (2008). ASPVIZ: Declarative visualisation and animation using answer set programming. In de la Banda, M. G. and Pontelli, E., editors, *Proceedings of the 24th International Conference on Logic Programming* (ICLP'08), Udine, Italy, December 9-13, 2008, volume 5366 of *Lecture Notes in Computer Science*, pages 724–728. Springer.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., and Weinzierl, A. (2012). OMiGA: An open minded grounding on-the-fly answer set solver. In del Cerro, L. F., Herzig, A., and Mengin, J., editors, *Proceedings of the 12th European Conference on Logics in Artificial Intelligence* (JELIA'12), Toulouse, France, September 26-28, 2012, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer.

De Vos, M., Crick, T., Padget, J. A., Brain, M., Cliffe, O., and Needham, J. (2006). LAIMA: A multi-agent platform using ordered choice logic programming. In Baldoni, M., Endriss, U., Omicini, A., and Torroni, P., editors, *Selected and Revised Papers of the Third International Workshop on Declarative Agent Languages and Technologies* (DALT'05), Utrecht, The Netherlands, July 25, 2005, volume 3904 of *Lecture Notes in Computer Science*, pages 72–88. Springer.

De Vos, M., Kısa, D. G., Oetsch, J., Pührer, J., and Tompits, H. (2012a). Annotating answer-set programs in LANA. *Theory and Practice of Logic Programming*, 12(4-5):619–637.

De Vos, M., Kısa, D. G., Oetsch, J., Pührer, J., and Tompits, H. (2012b). LANA: A language for annotating answer-set programs. In Rosati, R. and Woltran, S., editors, *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning* (NMR'12), Rome, Italy, June 8-10,2012.

De Vos, M. and Vermeir, D. (2002). Dynamic decision-making in logic programming and game theory. In McKay, B. and Slaney, J. K., editors, *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence* (AI'02), Advances in Artificial Intelligence, Canberra, Australia, December 2-6, 2002, volume 2557 of *Lecture Notes in Computer Science*, pages 36–47. Springer.

Delgrande, J. P. (2010). A program-level approach to revising logic programs under the answer set semantics. *Theory and Practice of Logic Programming*, 10(4-6):565–580.

Delgrande, J. P., Grote, T., and Hunter, A. (2009). A general approach to the verification of cryptographic protocols using answer set programming. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'09), Potsdam, Germany, September 14-18, 2009, volume 5753 of *Lecture Notes in Computer Science*, pages 355–367. Springer.

Denecker, M. (2000). Extending classical logic with inductive definitions. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L. M., Sagiv, Y., and Stuckey, P. J., editors, *Proceedings of the 1st International Conference on Computational Logic* (CL'10), London, UK, July 24-28, 2000, volume 1861 of *Lecture Notes in Computer Science*, pages 703–717. Springer.

Denecker, M. and Ternovska, E. (2008). A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, 9(2).

Denecker, M., Vennekens, J., Bond, S., Gebser, M., and Truszczyński, M. (2009). The second answer set programming competition. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'09), Potsdam, Germany, September 14-18, 2009, volume 5753 of *Lecture Notes in Computer Science*, pages 637–654, Berlin-Heidelberg, Germany. Springer.

Dix, J., Eiter, T., Fink, M., Polleres, A., and Zhang, Y. (2003). Monitoring agents using declarative planning. *Fundamenta Informaticae*, 57(2-4):345–370.

Dix, J., Faber, W., and Subrahmanian, V. S. (2012). Privacy preservation using multi-context systems and default logic. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 195–210. Springer.

Dworschak, S., Grell, S., Nikiforova, V. J., Schaub, T., and Selbig, J. (2008). Modeling biological networks by action languages via answer set programming. *Constraints*, 13(1-2):21–65.

Eclipse Project (2014). `http://www.eclipse.org/eclipse`. [Online; accessed October 6, 2014].

Eder, P. (2013). A SeaLion plugin for determining generating rules. Bachelor's thesis, Vienna University of Technology, Vienna, Austria.

Egly, U., Gaggl, S., and Woltran, S. (2010). Answer-set programming encodings for argumentation frameworks. *Argument & Computation*, 1(2):147–177.

Eiter, T., Faber, W., Leone, N., and Pfeifer, G. (1999). The diagnosis frontend of the DLV system. *AI Communications*, 12(1-2):99–111.

Eiter, T. and Gottlob, G. (1995). On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323.

Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., and Tompits, H. (2008). Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539.

Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2005). A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In Kaelbling, L. P. and Saffiotti, A., editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (IJCAI'05), Edinburgh, Scotland, UK, July 30-August 5, 2005, pages 90–96. Professional Book Center.

Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006). Effective integration of declarative rules with external evaluations for semantic-web reasoning. In Sure, Y. and Domingue, J., editors, *The Semantic Web: Research and Applications, Proceedings of the 3rd European Semantic Web Conference* (ESWC'06), Budva, Montenegro, June 11-14, 2006, volume 4011 of *Lecture Notes in Computer Science*, pages 273–287. Springer Verlag.

Eiter, T. and Wang, K. (2008). Semantic forgetting in answer set programming. *Artificial Intelligence*, 172(14):1644–1672.

Erdem, E., Aker, E., and Patoglu, V. (2012). Answer set programming for collaborative housekeeping robotics: Representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291.

Erdem, E., Lifschitz, V., and Ringe, D. (2006). Temporal phylogenetic networks and logic programming. *Theory and Practice of Logic Programming*, 6(5):539–558.

Erdem, E. and Wong, M. D. F. (2004). Rectilinear steiner tree construction using answer set programming. In Demoen, B. and Lifschitz, V., editors, *Proceedings of the 20th International Conference on Logic Programming* (ICLP'04), Saint-Malo, France, September 6-10, 2004, volume 3132 of *Lecture Notes in Computer Science*, pages 386–399. Springer.

Faber, W. (2005). Unfounded sets for disjunctive logic programs with arbitrary aggregates. In Baral, C., Greco, G., Leone, N., and Terracina, G., editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'05), Diamante, Italy, September 5-8, 2005, volume 3662 of *Lecture Notes in Computer Science*, pages 40–52. Springer.

Faber, W., Leone, N., and Perri, S. (2012). The intelligent grounder of DLV. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer.

Faber, W., Leone, N., and Pfeifer, G. (2004). Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence* (JELIA'04), volume 3229 of *Lecture Notes in Computer Science*, pages 200–212. Springer.

Faber, W., Pfeifer, G., and Leone, N. (2011). Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, 175(1):278–298.

Fang, M. (2013). A controlled natural language approach for interpreting answer sets. Bachelor's thesis, Vienna University of Technology, Vienna, Austria.

Febbraro, O., Leone, N., Reale, K., and Ricca, F. (2013). Unit testing in ASPIDE. In Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., and Wolf, A., editors, *Revised Selected Papers of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'11) and the 25th Workshop on Logic Programming (WLP'11), Vienna, Austria, September 28-30, 2011, volume 7773 of *Lecture Notes in Computer Science*, pages 345–364. Springer.

Febbraro, O., Reale, K., and Ricca, F. (2010). A visual interface for drawing ASP programs. In *Proceedings of the 25th Italian Conference on Computational Logic* (CILC'10).

Febbraro, O., Reale, K., and Ricca, F. (2011). ASPIDE: Integrated development environment for answer set programming. In Delgrande, J. P. and Faber, W., editors, *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16-19, 2011, volume 6645 of *Lecture Notes in Computer Science*, pages 317–330. Springer.

Ferraris, P. (2011). Logic programs with propositional connectives and aggregates. *ACM Transactions on Computational Logic*, 12(4):25.

Ferraris, P. and Lifschitz, V. (2005). Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5(1-2):45–74.

Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional.

Fritzson, P., Gyimothy, T., Kamkar, M., and Shahmehri, N. (1991). Generalized algorithmic debugging and testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming-Language Design and Implementation,* (PLDI'91), Toronto, Canada, pages 317–326. ACM Press.

Frühstück, M. (2013). Debugging in answer-set programs. Master's thesis, Alpen-Adria Universität Klagenfurt, Klagenfurt, Austria.

Frühstück, M., Pührer, J., and Friedrich, G. (2013). Debugging answer-set programs with Ouroboros – Extending the SeaLion plugin. In Cabalar, P. and Son, T. C., editors, *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'13), Corunna, Spain, September 15-19, 2013, volume 8148 of *Lecture Notes in Computer Science*, pages 323–328. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. (2010). *A user's guide to gringo, clasp, clingo, and iclingo*. Potassco Team. Available at `http://potassco.sourceforge.net` [Online; accessed October 6, 2014].

Gebser, M., Kaminski, R., König, A., and Schaub, T. (2011a). Advances in *gringo* series 3. In Delgrande, J. P. and Faber, W., editors, *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16-19, 2011, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer.

Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., and Thiele, S. (2009a). On the input language of ASP grounder gringo. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'09), Potsdam, Germany, September 14-18, 2009, volume 5753 of *Lecture Notes in Computer Science*, pages 502–508. Springer.

Gebser, M., Kaminski, R., and Schaub, T. (2011b). aspcud: A linux package configuration tool based on answer set programming. In Drescher, C., Lynce, I., and Treinen, R., editors, *Proceedings of the 2nd Workshop on Logics for Component Configuration* (LoCoCo'11), volume 65 of *EPTCS*, pages 12–25.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. T. (2011c). Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. (2007a). Conflict-driven answer set solving. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (IJCAI'07), Hyderabad, India, January 6-12, 2007, pages 386–392.

Gebser, M., Kaufmann, R., and Schaub, T. (2012). Gearing up for effective ASP planning. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 296–310. Springer.

Gebser, M., Lee, J., and Lierler, Y. (2011d). On elementary loops of logic programs. *Theory and Practice of Logic Programming*, 11(6):953–988.

Gebser, M., Pührer, J., Schaub, T., and Tompits, H. (2008). A meta-programming technique for debugging answer-set programs. In Fox, D. and Gomes, C. P., editors, *Proceedings of the 23rd AAAI Conference on Artificial Intelligence* (AAAI'08), Chicago, IL, USA, July 13-17, 2008, pages 448–453, Menlo Park, CA, USA. AAAI Press.

Gebser, M., Pührer, J., Schaub, T., Tompits, H., and Woltran, S. (2009b). spock: A debugging support tool for logic programs under the answer-set semantics. In Seipel, D., Hanus, M., and Wolf, A., editors, *Revised Selected Papers of the 17th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'07) and 21st Workshop on (Constraint) Logic Programming (WLP'07), volume 5437 of *Lecture Notes in Computer Science*, pages 247–252. Springer.

Gebser, M., Schaub, T., and Thiele, S. (2007b). Gringo : A new grounder for answer set programming. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'07), Tempe, AZ, USA, May 15-17, 2007, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. A. and Bowen, K., editors, *Proceedings of the 5th International Conference on Logic Programming,* (ICLP'88)), Seattle, WA, USA, pages 1070–1080. The MIT Press.

Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386.

Heljanko, K. and Niemelä, I. (2003). Bounded ltl model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550.

Hughes, T. P. (1989). *American Genesis: A History of the American Genius for Invention*. Penguin Books.

Ierusalimschy, R. (2006). *Programming in Lua, Second Edition*. Lua.org.

Janhunen, T. (2006). Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86.

Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., and Tompits, H. (2010). On testing answer-set programs. In Coelho, H., Studer, R., and Wooldridge, M., editors, *Proceedings of the 19th European Conference on Artificial Intelligence* (ECAI'10), Lisbon, Portugal, August 16-20, 2010, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 951–956. IOS Press.

Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., and Tompits, H. (2011). Random vs. structure-based testing of answer-set programs: An experimental comparison. In Delgrande, J. P. and Faber, W., editors, *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16-19, 2011, volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer.

Janhunen, T., Niemelä, I., Seipel, D., Simons, P., and You, J.-H. (2006). Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic*, 7(1):1–37.

Järvisalo, M., Berre, D. L., and Roussel, O. (2011). SAT competition 2011. `http://www.satcompetition.org/2011/`. [Online; accessed October 8, 2013].

Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843.

Kloimüllner, C. (2012). Visualisation and graphical editing of answer sets: The Kara system. Master's thesis, Vienna University of Technology, Vienna, Austria.

Kloimüllner, C. (2013). *Visualisation and Graphical Editing of Answer Sets: The Kara System: An advanced approach*. AV Akademikerverlag.

Kloimüllner, C., Oetsch, J., Pührer, J., and Tompits, H. (2013). Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *Revised Selected Papers of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'11) and the 25th Workshop on Logic Programming (WLP'11), Vienna, Austria, September 28-30, 2011, volume 7773 of *Lecture Notes in Computer Science*, pages 325–344. Springer.

Kniesel, G., Rho, T., Degener, L., Mühlschlegel, F., Stöwe, E., Noth, F., Becker, A., and Aliev, I. (2014). PDT - A Prolog IDE for Eclipse. `https://sewiki.iai.uni-bonn.de/research/pdt/docs/v2.1/start`. [Online; accessed 31-March-2014].

König, A. and Schaub, T. (2013). Monitoring and visualizing answer set solving. In *Technical Communications of the 29th International Conference on Logic Programming* (ICLP'13), Istanbul, Turkey, August 24-29, 2013, number 4-5-Online-Supplement in 13.

Koziarkiewicz, M. (2011). iGROM. `http://igrom.sourceforge.net/`. [Online; accessed 31-March-2014].

Lee, J. (2005). A model-theoretic counterpart of loop formulas. In Kaelbling, L. P. and Saffiotti, A., editors, *Proceedings of the 19th International Joint Conference on Artificial Intelligence* (IJCAI'05), Edinburgh, Scotland, UK, July 30-August 5, 2005, pages 503–508, Denver, CO, USA. Professional Book Center.

Lee, J. and Meng, Y. (2009). On reductive semantics of aggregates in answer set programming. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'09), volume 5753 of *Lecture Notes in Computer Science*, pages 182–195. Springer.

Lefèvre, C. and Nicolas, P. (2009). The first version of a new ASP solver : ASPeRiX. In Erdem, E., Lin, F., and Schaub, T., editors, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'09), Potsdam, Germany, September 14-18, 2009, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer.

Leone, N., Greco, G., Ianni, G., Lio, V., Terracina, G., Eiter, T., Faber, W., Fink, M., Gottlob, G., Rosati, R., Lembo, D., Lenzerini, M., Ruzzi, M., Kalka, E., Nowicki, B., and Staniszkis, W. (2005). The infomix system for advanced integration of incomplete and inconsistent data. In Özcan, F., editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 915–917. ACM.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562.

Leone, N., Rullo, P., and Scarcello, F. (1997). Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Information and Computation*, 135(2):69–112.

Lierler, Y. (2005). CMODELS - SAT-based disjunctive answer set solver. In Baral, C., Greco, G., Leone, N., and Terracina, G., editors, *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'05), Diamante, Italy, September 5-8, 2005, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer.

Lierler, Y. (2011). Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming*, 11(2-3):135–169.

Lierler, Y. and Schüller, P. (2012). Parsing combinatory categorial grammar via planning in answer set programming. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 436–453. Springer.

Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54.

Lin, F. and Zhao, Y. (2002). ASSAT: Computing answer sets of a logic program by SAT solvers. In Dechter, R. and Sutton, R. S., editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence* (AAAI'02), July 28 - August 1, 2002, Edmonton, Alberta, Canada, pages 112–118. AAAI Press / The MIT Press.

Lin, F. and Zhao, Y. (2004). ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137.

Liu, L., Pontelli, E., Son, T. C., and Truszczyński, M. (2010). Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence*, 174(3-4):295–315.

Liu, L. and Truszczyński, M. (2006). Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research*, 27:299–334.

Marek, V. W., Niemelä, I., and Truszczyński, M. (2008). Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*, 8(2):167–199.

Marek, V. W. and Remmel, J. B. (2004). Set constraints in logic programming. In Lifschitz, V. and Niemelä, I., editors, *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'04), Fort Lauderdale, FL, USA, Jan 6-8, 2004, volume 2923 of *Lecture Notes in Computer Science*, pages 167–179. Springer.

Marek, V. W. and Remmel, J. B. (2012). Disjunctive programs with set constraints. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 471–486. Springer.

Marek, V. W. and Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In Apt, K. R., Marek, V. W., Truszczyński, M., and Warren, D. S., editors, *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer.

Marek, V. W. and Truszczyński, M. (2004). Logic programs with abstract constraint atoms. In Ferguson, G. and McGuinness, D., editors, *In Proceedings of the 19th National Conference on Artificial Intelligence* (AAAI'04), San Jose, CA, USA, July 25-29, 2004, pages 86–91. AAAI Press.

McCarthy, J. (1980). Circumscription - A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39.

Mikitiuk, A., Moseley, E., and Truszczyński, M. (2007). Towards debugging of answer-set programs in the language PSpb. In Arabnia, H. R., Yang, M. Q., and Yang, J. Y., editors, *Proceedings of the 2007 International Conference on Artificial Intelligence* (ICAI'07), Volume II, Las Vegas, NV, USA, June 25-28, 2007, pages 635–640, Bogart, GA, USA. CSREA Press.

Mileo, A., Schaub, T., Merico, D., and Bisiani, R. (2011). Knowledge-based multi-criteria optimization to support indoor positioning. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):345–370.

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., and Miller, K. J. (1990). Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4):235–244.

Moore, R. C. (1985). Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94.

Naish, L. (1992). Declarative debugging of lazy functional programs. In *Proceedings of the 4th Workshop on Logic Programming Environments* (WLPE'92), Washington, DC, USA, pages 29–34.

Naqvi, S. A. (1986). A logic for negation in database systems. In Mincker, J., editor, *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming, Washington, DC, August 1986*, pages 378–387. Morgan Kaufman.

Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273.

Niemelä, I. and Simons, P. (1996). Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming* (JICSLP'96), pages 289–303. MIT Press.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., and Barry, M. (2001). An A-Prolog decision support system for the Space Shuttle. In Ramakrishnan, I. V., editor, *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages* (PADL'01), Las Vegas, Nevada, March 11-12, 2001, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer.

Oetsch, J., Prischink, M., Pührer, J., Schwengerer, M., and Tompits, H. (2012a). On the small-scope hypothesis for testing answer-set programs. In Brewka, G., Eiter, T., and McIlraith, S. A., editors, *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning* (KR'12), Rome, Italy, June 10-14, 2012. AAAI Press.

Oetsch, J., Pührer, J., Seidl, M., Tompits, H., and Zwickl, P. (2011a). VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In Delgrande, J. P. and Faber, W., editors, *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16-19, 2011, volume 6645 of *Lecture Notes in Computer Science*, pages 382–387. Springer.

Oetsch, J., Pührer, J., and Tompits, H. (2010a). Catching the Ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming*, 10(4-6):513–529.

Oetsch, J., Pührer, J., and Tompits, H. (2010b). Let's break the rules: Interactive procedural-style debugging of answer-set programs. In Abdennadher, S., editor, *Proceedings of the 24th Workshop on (Constraint) Logic Programming* (WLP'10), Cairo, Egypt, September 14-16, 2010, Technical Report, Faculty of Media Engineering and Technology, German University in Cairo, pages 77–87.

Oetsch, J., Pührer, J., and Tompits, H. (2011b). The SeaLion has landed: An IDE for answer-set programming—Preliminary report. In *Proceedings of the 25th Workshop on Logic Programming* (WLP'11), Vienna, Austria, September 28-30, 2011, volume 1843-11-06 of *INFSYS Research Report 1843-11-06*, pages 141–151.

Oetsch, J., Pührer, J., and Tompits, H. (2011c). Stepping through an answer-set program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'11), Vancouver, Canada, May 16-19, 2011, volume 6645 of *Lecture Notes in Computer Science*, pages 134–147. Springer.

Oetsch, J., Pührer, J., and Tompits, H. (2012b). An FLP-style answer-set semantics for abstract-constraint programs with disjunctions. In Dovier, A. and Costa, V. S., editors, *Technical Communications of the 28th International Conference on Logic Programming* (ICLP'12), Budapest, Hungary, volume 17 of *LIPIcs*, pages 222–234. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Oetsch, J., Pührer, J., and Tompits, H. (2012c). Stepwise debugging of description-logic programs. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 492–508. Springer.

Oetsch, J., Pührer, J., and Tompits, H. (2013). The SeaLion has landed: An IDE for answer-set programming—Preliminary report. In *Revised Selected Papers of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (INAP'11) and the 25th Workshop on Logic Programming (WLP'11), Vienna, Austria, September 28-30, 2011, volume 7773 of *Lecture Notes in Computer Science*, pages 305–324. Springer.

Osorio, M. and Cuevas, V. (2007). Updates in answer set programming: An approach based on basic structural properties. *Theory and Practice of Logic Programming*, 7(4):451–479.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition.

Perri, S., Ricca, F., Terracina, G., Cianni, D., and Veltri, P. (2007). An integrated graphic tool for developing and testing DLV programs. In De Vos, M. and Schaub, T., editors, *Proceeding of the 1st International Workshop on Software Engineering for Answer Set Programming* (SEA'07), Tempe, AZ, USA, May 14, 2007, pages 86–100.

Polleres, A., Frühstück, M., Schenner, G., and Friedrich, G. (2013). Debugging non-ground ASP programs with choice rules, cardinality constraints and weight constraints. In Cabalar, P. and Son, T. C., editors, *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'13), Corunna, Spain, September 15-19, 2013, volume 8148 of *Lecture Notes in Computer Science*, pages 452–464. Springer.

Pontelli, E., Son, T. C., Baral, C., and Gelfond, G. (2012). Answer set programming and planning with knowledge and world-altering actions in multiple agent domains. In Erdem, E., Lee, J., Lierler, Y., and Pearce, D., editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 509–526. Springer.

Pontelli, E., Son, T. C., and El-Khatib, O. (2009). Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(1):1–56.

Pührer, J. (2007). On debugging of propositional answer-set programs. Master's thesis, Vienna University of Technology, Vienna, Austria.

Pührer, J., Heymans, S., and Eiter, T. (2010). 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., and Tudorache, T., editors, *Proceedings of the 7th Extended Semantic Web Conference* (ESWC'10), Part I, The Semantic Web: Research and Applications, Heraklion, Crete, Greece, May 30 - June 3, 2010, volume 6088 of *Lecture Notes in Computer Science*, pages 183–197. Springer.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132.

Ricca, F., Gallucci, L., Schindlauer, R., Dell'Armi, T., Grasso, G., and Leone, N. (2009). OntoDLV: An ASP-based system for enterprise ontologies. *Journal of Logic and Computation*, 19(4):643–670.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., and Leone, N. (2012). Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, 12(3):361–381.

Ruzicka, B. (1990). Entwurf und Implementierung eines Debugger in einer Expertensystemumgebung. Master's thesis, Vienna University of Technology, Vienna, Austria.

Sakama, C. (2001). Learning by answer sets. In Provetti, A. and Son, T. C., editors, *Proceedings of the 1st International Workshop on Answer Set Programming* (ASP'01), Towards Efficient and Scalable Knowledge Representation and Reasoning, Stanford, March 26-28, 2001.

Sakama, C. (2005). Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic*, 6(2):203–231.

Sakama, C. and Inoue, K. (2009). Brave induction: A logical framework for learning from incomplete information. *Machine Learning*, 76(1):3–35.

Schaub, T. and Wang, K. (2001). A comparative study of logic programs with preference. In Nebel, B., editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (IJCAI'01), Seattle, Washington, USA, August 4-10, 2001, pages 597–602. Morgan Kaufmann.

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, 39(2):41–47.

Shapiro, E. Y. (1982). *Algorithmic Program Debugging*. PhD thesis, Yale University, New Haven, CT, USA.

Shen, Y.-D. and You, J.-H. (2007). A generalized Gelfond-Lifschitz transformation for logic programs with abstract constraints. In Holte, R. C. and Howe, A., editors, *Proceedings of the 22nd Conference on Artificial Intelligence* (AAAI'07), Menlo Park, California, USA, July 22-26, 2007, pages 483–488. AAAI Press.

Shen, Y.-D., You, J.-H., and Yuan, L.-Y. (2009). Characterizations of stable model semantics for logic programs with arbitrary constraint atoms. *Theory and Practice of Logic Programming*, 9(4):529–564.

Simkus, M. (2009). Fusion of logic programming and description logics. In Hill, P. M. and Warren, D. S., editors, *Proceedings of the 25th International Conference on Logic Programming*(ICLP'09), Pasadena, CA, USA, July 14-17, 2009, volume 5649 of *Lecture Notes in Computer Science*, pages 551–552. Springer.

Simons, P., Niemelä, I., and Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234.

Skočovský, P. (2014). Realisation of stepping for real-world ASP languages. Master's thesis, Universidade Nova de Lisboa, Lisbon, Portugal.

Smith, A. (2011). Lonsdaleite. `https://github.com/rndmcnlly/Lonsdaleite`. [Online; accessed October 8, 2013].

Soininen, T. and Niemelä, I. (1999). Developing a declarative rule language for applications in product configuration. In Gupta, G., editor, *Proceedings or the 1st International Workshop on Practical Aspects of Declarative Languages* (PADL'99), San Antonio, Texas, USA, January 18-19, 1999, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer.

Son, T. C., Pontelli, E., and Tu, P. H. (2007). Answer sets for logic programs with arbitrary abstract constraint atoms. *Journal of Artificial Intelligence Research*, 29:353–389.

Sonenberg, L. and Topor, R. W. (1988). On domain independent disjunctive databases. In Gyssens, M., Paredaens, J., and Gucht, D. V., editors, *Proceedings of the 2nd International Conference on Database Theory* (ICDT'88), Bruges, Belgium, August 31 - September 2, 1988, volume 326 of *Lecture Notes in Computer Science*, pages 281–291. Springer.

Sureshkumar, A. (2006). AnsProlog* Programming Environment (APE): Investigating software tools for answer set programming through the implementation of an integrated development environment. Bachelor's thesis, University of Bath, Bath, UK.

Sureshkumar, A., De Vos, M., Brain, M., and Fitch, J. (2007). APE: An AnsProlog* environment. In De Vos, M. and Schaub, T., editors, *Proceeding of the 1st International Workshop on Software Engineering for Answer Set Programming* (SEA'07), Tempe, AZ, USA, May 14, 2007, pages 71–85.

Syrjänen, T. (2000). Including diagnostic information in configuration models. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L. M., Sagiv, Y., and Stuckey, P. J., editors, *Proceedings of the 1st International Conference on Computational Logic* (CL'00), London, UK, 24-28 July, 2000, volume 1861 of *Lecture Notes in Computer Science*, pages 837–851. Springer.

Syrjänen, T. (2002). *Lparse 1.0 User's Manual*. Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland. `http://www.tcs.hut.fi/Software/smodels` [Online; accessed October 6, 2014].

Syrjänen, T. (2006). Debugging inconsistent answer set programs. In Dix, J. and Hunter, A., editors, *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning* (NMR'06), Lake District, UK, May 30-June 1, 2006, pages 77–83, Clausthal, Germany. Institut für Informatik, Technische Universität Clausthal, Techical Report.

Tang, C. K. F. and Ternovska, E. (2007). Model checking abstract state machines with answer set programming. *Fundamenta Informaticae*, 77(1-2):105–141.

Terracina, G., Francesco, E. D., Panetta, C., and Leone, N. (2008a). Enhancing a DLP system for advanced database applications. In Calvanese, D. and Lausen, G., editors, *Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems* (RR'08), Karlsruhe, Germany, October 31-November 1 2008, volume 5341 of *Lecture Notes in Computer Science*, pages 119–134. Springer.

Terracina, G., Leone, N., Lio, V., and Panetta, C. (2008b). Experimenting with recursive queries in database and logic programming systems. *Theory and Practice of Logic Programming*, 8(2):129–165.

Topor, R. W. and Sonenberg, L. (1988). On domain independent databases. In *Foundations of Deductive Databases and Logic Programming.*, pages 217–240. Morgan Kaufmann.

Tran, N. and Baral, C. (2004). Reasoning about triggered actions in ansprolog and its application to molecular interactions in cells. In Dubois, D., Welty, C. A., and Williams, M.-A., editors, *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning* (KR'04), Whistler, Canada, June 2-5, 2004, pages 554–564. AAAI Press.

Truszczyński, M. (2010). Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artificial Intelligence*, 174(16-17):1285–1306.

Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265.

Van Gelder, A. (1989). Negation as failure using tight derivations for general logic programs. *J. Log. Program.*, 6(1&2):109–133.

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

Weiten, M. (2009). Ontostudio® as a ontology engineering environment. In Davies, J., Grobelnik, M., and Mladenic', D., editors, *Semantic Knowledge Management*, pages 51–60. Springer.

Wertz, H. (1982). Stereotyped program debugging: an aid for novice programmers. *International Journal of Man-Machine Studies*, 16(4):379–392.

Wittocx, J. (2009). IDPDraw, a tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation`. [Online; accessed October 8, 2013].

Wittocx, J., Vlaeminck, H., and Denecker, M. (2009). Debugging for model expansion. In Hill, P. M. and Warren, D. S., editors, *Proceedings of the 25th International Conference on Logic Programming* (ICLP'09), Pasadena, CA, USA, July 14-17, 2009, volume 5649 of *Lecture Notes in Computer Science*, pages 296–311, Berlin-Heidelberg, Germany. Springer.

Curriculum Vitae

Personal Information

Name	Jörg Pührer
Adresse	Ploßstrasse 50, D-04347 Leipzig, Germany
E-mail	puehrer@informatik.uni-leipzig.de
Phone	+49 176 66366150
Date of Birth	15.08.1981
Place of Birth	Bad Ischl, Austria
Gender	Male
Citizenship	Austrian

Education

2008–	PhD Student at the Vienna University of Technology.
2004–2007	Student of Computer Intelligence at Vienna University of Technology, graduation as a Diplomingenieur (M.Sc.) with distinction, Thesis: On Debugging of Propositional Answer-Set Programs.
2004	Graduation as a Bakkalaureus der technischen Wissenschaften (B.Sc.), Project: A Genetic-Programming Plugin for the HeuristicLab Optimisation framework.
2002–2003	Exchange Student at the University of Edinburgh, United Kingdom, under the ERASMUS-programme with special emphasis on Artificial Intelligence.
2002	First Diploma exam passed with distinction.
2000–2004	Student of Computer Science at Johannes Kepler University, Linz.

Professional Experience

2013–	**Leipzig University, Leipzig, Germany** Institute of Computer Science, Intelligent Systems Group *Research Assistant*
2008-2013	**Vienna University of Technology, Vienna, Austria** Institute of Information Systems, Knowledge-based Systems Group *Research Assistant*
2007	**Vienna University of Technology, Vienna, Austria** Institute of Information Systems, Knowledge-based Systems Group *Tutor*

Services to the Community

Organisation	Co-chair of the International Workshop on Reactive Concepts in Knowledge Representation (ReactKnow 2014).
Committees	Program Committee of the 21st European Conference on Artificial Intelligence (ECAI 2014), Program Committee of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), Organising Committee of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011).
Reviewing	AAAI, CLIMA, DATALOG, DL, ECAI, FOIKS, GTTV, ICCSW, ICLP, IJCAI, INAP, ISWC, JELIA, KR, LPAR, LPNMR, PADL, POPL, WLP, WoLLiC.

Research Projects

2013–	**Abstract Dialectical Frameworks: Advanced Tools for Formal Argumentation.** Funded by the German Research Foundation (DFG).
2009-2013	**Methods and Methodologies for Developing Answer-Set Programs.** Funded by the Austrian Science Fund.
2011	**Net2: A Network for Enabling Networked Knowledge.** Funded by the EU 7th Framework Programme under the Marie Curie action "International Research Staff Exchange Scheme (IRSES)".
2009-2011	**ONTORULE - ONTOlogies meet Business RULEs.** Funded by the European Union's 7th Framework Programme under the Information and Communication Technologies Call 3.
2008	**Formal Methods for Comparing and Optimizing Nonmonotonic Logic Programs.** Funded by the Austrian Science Fund.

Grants and Scholarships

2006–2007	3 Student Grants of Project P18019 of the Austrian Science Fund (FWF).
2004–2005	Merit scholarship, Fakultät für Informatik, Vienna University of Technology.
2001–2004	Merit scholarship, Technisch-Naturwissenschaftliche Fakultät, Johannes Kepler University Linz.

Awards

2014	Best Presentation Award, Imperial College Computing Student Workshop (ICCSW 2014).

Research Visits

March-May 2011	Pontificia Universidad Católica de Chile, Santiago, Chile.

Guest Talks

2013	SeaLion: An Eclipse-based IDE for ASP with Advanced Debugging Support. Leipzig University, Leipzig, Germany.

2012	The SeaLion has Landed: An IDE for Answer-Set Programming—Status Report. University of Calabria, Rende, Italy.
2012	The SeaLion has Landed: An IDE for Answer-Set Programming—Status Report. University of Bath, Bath, UK.
2011	Invited Student Talk: Stepping through an Answer-Set Program. Fifth Alberto Mendelzon Workshop on Foundations of Data Management (AMW'11), Santiago, Chile.
2011	Answer-Set Programming: Basics, Combinations with Ontologies, and Debugging. Universidad de Chile, Santiago, Chile.
2010	Let's Break the Rules: Interactive Procedural Style Debugging of Answer-Set Programs. Together with Johannes Oetsch and Hans Tompits. Aalto University, Helsinki, Finland.
2010	Let's Break the Rules: Interactive Procedural Style Debugging of Answer-Set Programs. Sabanci University, Istanbul, Turkey
2009	Methods and Methodologies for Developing Answer-Set Programs. Together with Johannes Oetsch and Hans Tompits. Helsinki University of Technology, Helsinki, Finland.
2006	Debugging of Logic Programs under the Answer-Set Semantics. University of Potsdam, Germany.