FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Abstract Argumentation and Answer-Set Programming

Modelling the Resolution-Based Grounded Semantics

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Christian Weichselbaum
Matrikelnummer 0525522

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Stefan WOLTRAN
Mitwirkung: Projektass. Dipl.-Ing. Sarah Alice GAGGL

Wien, 24.01.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Abstract Argumentation and Answer-Set Programming

Modelling the Resolution-Based Grounded Semantics

MASTER THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Christian Weichselbaum
Registration Number 0525522

at the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Stefan WOLTRAN
Assistance: Projektass. Dipl.-Ing. Sarah Alice GAGGL

Vienna, 24.01.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Christian Weichselbaum
Brandfeldgasse 14, 2120 Obersdorf

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)

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of my advisor Stefan Woltran and his assistance Sarah A. Gaggl who guided me into the field of abstract argumentation and lead me to the goal of completing my studies in Computational Intelligence. I also deeply appreciate the help of Johannes Wallner, who did not hesitate to share his experience with me and swiftly responded anytime, I had a question.
Also, I am indepted to my parents for making it possible for me to obtain such great education, and I would like to thank them for the many years of support prior to my graduation. Last but not least, I owe my deepest gratitude to Sabrina, who always is there to patiently and with love support me in reaching my goals.

Abstract

Since its emergence in the 20th century, the field of argumentation in Artificial Intelligence (AI) has experienced significant growth. One central topic within this field is the acceptability of arguments for which P. M. Dung [19] introduced a theory based on the notion of abstract argumentation frameworks. These frameworks are simply consisting of a set of arguments and a binary relation constituting attacks, without any inner structure. Since arguments may attack each other, it is the case that not all of them might be able to stand together. Therefore, arguments are subject to an evaluation method, of which the formal definition is called an *argumentation semantics* . This method can either be declarative or procedural. In order to be able to compare those semantics, several formal principles have been defined [6] which should be fulfilled by a single extension or the whole set of extensions of a semantics. In their paper *On the resolution-based family of abstract argumentation semantics and its grounded instance* [5] P. Baroni, P. E. Dunne and M. Giacomin introduced a new family of semantics for abstract argumentation systems, among them the resolution-based grounded semantics, which has the unique property of satisfying all principles. There have already been encodings developed within the realm of answer-set programming (ASP) [39], a form of declarative programming oriented towards difficult search problems and based on the stable model semantics as presented by M. Gelfond and V. Lifschitz [36]. Yet there are still performance problems with the encodings of this particular semantics. Our task is to develop ASP encodings, based on an algorithm presented by P. Baroni and M. Giacomin [6] that compute the extensions of this semantics and provide performance gains compared to existing approaches. These encodings shall then be incorporated in the *ASPARTIX system*, introduced by U. Egly et al. [25]. Furthermore, we keep track of any optimizations which result in a performance gain since they might be applicable on other semantics as well.

Kurzfassung

Seit seiner Entstehung im 20. Jahrhundert hat das Feld der Argumentation innerhalb der KI ein deutliches Wachstum erlebt. Ein zentrales Thema ist die Akzeptanz von Argumenten für die P. M. Dung [19] eine neue Theorie, welche auf dem Begriff der Abstract Argumentation Frameworks beruht, entwickelt hat. Diese Frameworks bestehen aus zwei Mengen: einer Menge von Argumenten und einer binären Relation, welche die Angriffe - auch Attacks gennant - darstellt. Diese Argumente besitzen jedoch keine innere Struktur, daher auch der Name Abstract Argumentation. Da sich Argumente gegenseitig attackieren können, sind nicht alle Argumente gemeinsam haltbar. Aus diesem Grund wendet man auf die Argumente eine Evaluierungsmethode an, welche auch als Semantik bezeichnet wird. Diese Methode kann entweder deklarativ oder prozedural sein. Um diese Semantiken vergleichen zu können, wurden einige formale Prinzipien definiert [6], die von einer einzelnen Extension oder der ganzen Menge von Extensionen einer Semantik erfüllt werden können. In ihrer Arbeit *On the resolution-based family of abstract argumentation semantics and its grounded instance* [5] haben P. Baroni, P. E. Dunne, und M. Giacomin eine neue Familie von Semantiken für Abstract Argumentation Systeme eingeführt. Unter diesen Semantiken befindet sich die Resolution-Based Grounded Semantics, welche die einzigartige Eigenschaft hat, alle formalen Prinzipien zu erfüllen, die in ihrer bereits erwähnten Arbeit [6] präsentiert wurden. Im Bereich der Answer Set Programmierung wurden bereits Kodierungen dieser neuen Semantik entwickelt. Answer Set Programming (ASP) [39], ist eine deklarative Programmiersprache, welche für komplexe Suchprobleme entwickelt wurde. Diese Programmiersprache basiert auf der Stable Model Semantics, welche von M. Gelfond und V. Lifschitz [36] eingeführt wurde. Die existierenden Kodierungen leiden jedoch noch unter Performance Problemen, weshalb wir es uns zur Aufgabe gemacht haben ein eigenes ASP Programm mit besserer Leistung zu entwickeln. Unsere Kodierung basiert auf einem Algorithmus, der von P. Baroni und M. Giacomin [6] präsentiert wurde. Diese Kodierungen sollen dann im *ASPARTIX System* einbindbar sein, welches von U. Egly et al. [25] entwickelt wurde. Des weiteren versuchen wir jene Code-Segmente und Optimierungen zu identifizieren, welche zu einer Leistungssteigerung bei der Berechnung der Extensionen führen.

Contents

1 Introduction **1**
- 1.1 Main Contributions . . . 2
- 1.2 Related Work . . . 3
- 1.3 Organization . . . 4

2 Background **5**
- 2.1 Complexity . . . 5
- 2.2 Abstract Argumentation . . . 8
- 2.3 Answer-Set Programming . . . 24

3 ASP based Argumentation **47**
- 3.1 Encodings of Standard Semantics . . . 47
- 3.2 Previous Encodings of the Resolution Based Grounded Semantics . . . 52

4 New Encodings of the Resolution Based Grounded Semantics **57**
- 4.1 Outline . . . 57
- 4.2 Encoding . . . 58
- 4.3 Another Variant of a Verification-Algorithm Based Encoding . . . 64

5 Experimental Evaluation **69**
- 5.1 Test Set . . . 69
- 5.2 Test Environment . . . 69
- 5.3 The Candidate Encodings . . . 70
- 5.4 Results . . . 70
- 5.5 Observations . . . 83

6 Summary and Future Work **85**

Bibliography **87**

CHAPTER 1

Introduction

In our daily life we deal a lot with different forms of argumentation. Whether we have to find a common decision, are negotiating, are trying to pursuade somebody to accept our ideas or are simply asking for the reason behind some behavior.
Since Aristotle, science has tried to find mechanisms and methods to distinguish between legitimate arguments and flawed arguments, also called fallacies. Deductive logic and other similar formal approaches did not prove useful for this purpose, since in natural language we often deal with incomplete, vague or uncertain information. We may also often have to deal with enthymemes, which are implicit premises. So in the last third of the 20th century a new school of thought called informal logic, gained momentum in order to be able to analyze the structures that constitute argument components and to evaluate the argumentation processes. There are four major tasks in argumentation which can be identified:

1. The task of *identification*, which deals with identifying the premises and conclusions of an argument and also checks whether it fits the argumentation scheme in the corresponding text of discourse.

2. The task of *analysis*, which identifies implicit premises and conclusions, and makes them explicit, which is necessary for proper argument evaluation.

3. The task of *evaluation*, which determines whether an argument is weak or strong or if it can withstand the general criteria and principles, that are applied to it.

4. The task of *invention*, which deals with the construction of new arguments in order to support a desired conclusion.

But in order to be able to computationally deal with any arguments in natural language, we have to formalize them in some way. Some common approaches are *defeasible logic programming* as presented in an article by A. Garcia and G. Simari [29], *defeasible argumentation with specificity-based preferences* as presented in an article by G. Simari and R. Loui [48] or *Argumentation Based on Classical Logic* as presented by P. Besnard and A. Hunter [10] which we will take a

look at and shortly describe in Chapter 2: Background.

One central topic within the field of argumentation in artificial intelligence is the acceptability of arguments which, considering the four major tasks of argumentation, belongs to the task of evaluation. In argumentation, in order to formally define the acceptability of an argument, P. M. Dung [19] introduced a theory, which is based on the notion of abstract argumentation frameworks. These frameworks are simply consisting of a set of arguments and a binary relation constituting attacks, where arguments are not assumed to have any specific structure. Since arguments may attack each other, it is the case that not all of them might be able to stand together. Therefore one has to develop formal methods which, based on this theory, determine for a given argument whether it is accepted or not. Those formal methods are called *argumentation semantics*. Two major approaches can be identified in order to realize such semantics: The *reduction approach* and the direct approach. In the direct approach, one picks a programming language of choice and develops a customized algorithm which is tailored just to realize this particular theory. Whereas in the reduction approach one builds a solution based on existing software which has proven great performance in similar tasks. Such an approach makes use of answer-set programming (ASP) [39]. ASP is a form of declarative programming, oriented towards difficult search problems and based on the stable model semantics as presented by M. Gelfond and V. Lifschitz [36].
In *abstract argumentation* a variety of different semantics has been proposed. In order to be able to compare them, several formal principles have been defined [6] which should be fulfilled by a single extension or the whole set of extensions of a semantics. In their paper *On the resolution-based family of abstract argumentation semantics and its grounded instance* [5], P. Baroni, P. E. Dunne, and M. Giacomin introduced a new family of semantics for abstract argumentation systems, among them the resolution-based grounded semantics which has the unique property of satisfying all principles. There already exist ASP encodings of various semantics, including the resolution-based grounded semantics. This semantics, such as some others, is not easy to encode. Hence, there are still performance problems with the encodings of the resolution-based grounded semantics semantics. Our task is to develop ASP encodings, based on an algorithm presented by P. Baroni and M. Giacomin [6] that compute the extensions of this semantics and provide performance gains compared to existing approaches. These encodings shall then be incorporated in the *ASPARTIX system*, introduced by U. Egly et al. [25] .

1.1 Main Contributions

In this section we stipulate the contributions this thesis aims to make to the field of abstract argumentation within the realm of answer-set programming. As already mentioned, there are existing approaches to encoding the resolution-based grounded semantics but yet there is room for improvement.

1. Primarily with this thesis we expect to find an alternative way of realizing the resolution-

based grounded semantics based on an ASP encoding which provides a significant performance gain. Therefor, we will develop a Guess & Check answer-set program, based on the verification algorithm VER_{GR^*} presented in [6]. With this encoding, we will conduct performance comparisons with existing encodings for the same problem.

2. In the process of developing our novel approach, we will apply different ASP modeling techniques known to achieve performance gains of answer set programs, such as using aggregates or symmetry breaking, which aims at eliminating symmetric parts of the search space. We will retain the specific optimizations which can be identified successfully. Those optimizations do not only contribute to a better encoding for the resolution-based grounded semantics, but due to their relation to the encodings of other extension based semantics within the field of abstract argumentation, they might be applicable on semantics other than the resolution-based grounded semantics as well.

3. Finally, we will contribute a thourough analysis of which answer-set programming solver is best suited for computing the extensions of the resolution-based grounded semantics.

1.2 Related Work

Work related to this thesis, is situated in the task of evaluation within argumentation. As previously mentioned, the approaches of developing encodings in this field which evaluate the acceptability of arguments, are divided into the reduction approach and the direct approach. All the work related to our solution belongs to the reduction approach. One example within this realm is *Constraint Satisfaction Programming* (CSP), upon which several methods have been based. A CSP basically consists of a triple $\langle V, D, C\rangle$ that is composed of a finite set of variables V, a domain D for each variable $v \in V$ and finally a set of constraints, C. The constraints are imposed on an arbitrary set of variables $X \subseteq V$ and govern the assignment of values to the respecitve variables. A solution to a problem in CSP is found if all constraints are satisfied. Applications of CSP in the field of abstract argumentation can be found in the article by S. Bistarelli et al. [12] who specify constraints for several principles such as conflict-freeness or admissibility. Further applications of CSP within the same area can be found in [1, 13].
Another approach constituting the scope of work related to this thesis, is ASP-based. One of these related approaches has been proposed by C. Nieves et al. [42] and is based on a method relying on the use of propositional formulas, another one by T. Wakaki and K. Nitta, presented in their article [54]. Also within the scope of related work, one can find the approach of U. Egly et al. [24, 25] where answer-set programming encodings are proposed that compute the extensions of the conflict-free, admissible, preferred, stable, complete and grounded semantics. Other similar concepts also exist. The most closely related works are two existing approaches computing the resolution-based grounded semantics:

1. Proposed by W. Dvořák et al. [22], the first approach is based on an algorithm which was originally proposed by Baroni et al. [5]. It is primarily based on the Guess & Check paradigm and on the iteration over sets.

2. This second approach by W. Dvořák et al. [22] is based on the meta-modeling techniques as proposed in [32] by M. Gebser et al.

1.3 Organization

This thesis is structured into 6 chapters. The first chapter contains the introduction, some words on the main contributions, an overview of related work and this section, introducing the structure. The second chapter provides the necessary background on complexity theory, abstract argumentation and answer-set programming. We continue with a chapter on ASP based argumentation, where we first present approaches for encodings of the standard semantics of abstract argumentation with answer-set programming. Also in this chapter, we include an overview of alternative ways for encoding the resolution-based grounded semantics. We continue with a chapter, where we present our novel encoding of the resolution-based grounded semantics and a detailed description of the new approach, proposed by W. Dvořák et al. which has been simultaneously developed. After the chapter on the new encodings of the resolution-based grounded semantics follows a chapter on the experimental evaluation. We provide a thorough description of the test set, the environment in which we conducted our benchmarks and again provide a short overview of the candidate encodings which we compared to our implementation. We provide a detailed section on our results and a description of the observations we made, especially with respect to optimization possibilities of the encodings. At the end of our thesis, in an extra chapter, we shortly summarize the main discoveries and outline what may be open for future work.

CHAPTER 2

Background

In this chapter we provide an extensive overview over and the necessary background of the different fields and technologies that we encountered during the development of this thesis. Section 2.1 provides a short introduction into the realm of complexity theory and gives an overview of complexity classes and the polynomial hierarchy which we will refer to again later on. In the subsequent section 2.2 we provide a general introduction into the field of Argumentation in Artificial Intelligence. We hope to provide the reader with a brief glimpse on how to derive abstract arguments from real world arguments and how those arguments are then dealt with in abstract argumentation. The main part of this section constitutes a particular description of abstract argumentation. We provide a general point of view on the concept of labeling-based semantics and a more detailed explication of extension-based semantics along with its principles. In this section the reader may also find a detailed overview over the most important extension-based semantics and a table of their complexity results. Lastly we include a thorough introduction of the resolution-based grounded semantics. In the last section 2.3 we provide an introduction of Answer-Set Programming which we used to realize our encodings of the abstract argumentation semantics. We describe the general concept of answer-set programming and the stable model semantics. Furthermore in this section, the reader will find a general survey of the various language extensions of answer-set programming and different modeling techniques. Last but not least, we included a description of CLASP and DLV, two answer-set programming solvers which we used in our research and implementation.

2.1 Complexity

In this section we provide a short introduction of complexity theory and the different complexity classes which we will need later when describing argumentation, answer-set programming and our implementation.

> Complexity theory is the part of theoretical computer science that attempts to prove that certain transformations from input to output are impossible to compute using a reasonable amount of resources. [44]

Computational problems can be assigned to different classes, depending on their use of certain resources such as *run time* or *space*. Complexity theory deals with the properties and relations of those classes. The classification of each problem is based solely on the problem itself, regardless of any possible algorithm that may be used for its computation. In order to explain the decision problems of abstract argumentation we provide a brief overview over the complexity classes we will need in the subsequent section. These classes include *P*, *NP*, *NP-complete*, *co-NP*, *P-complete*, *co-NP-complete* and Π_2^P*-complete*. For further details about these classes please refer to C. H. Papadimitriou [43].

The complexity class P

The complexity class *P* is one of the most fundamental complexity classes. It contains all the problems *P* which are solvable in polynomial time. In other words, it contains those problems for which a solution can be found by an algorithm within $n^k + k$ computation steps, for some constant k.

Definition 1. *A decision problem D is in complexity class P, iff there exists a program Π that decides D and Π has a runtime for each instance I of D with an upper-bound $O(|I|^k)$, where k is a constant and $|I|$ denotes the size of an instance I.*

NP

Another fundamental complexity class is *NP*. It includes all decision problems D which are solvable in polynomial time by a non-deterministic Turing machine. The complexity class *P* is contained in the class of *NP*. Problems within this class are characterized by their property that they can be verified quickly whether any given instance I is a positive solution of D, but there is no fast algorithm known that can find all such solutions, at this time.

Definition 2. *A certificate relation R for a problem D is a relation $R \subseteq$ Instances$(D) \times$ Cert, where $Cert$ is a set of finite objects such that $I \in$ Instances(D) is a positive instance of D iff $\exists C \in$ Cert : $(I, C) \in R$.*

Definition 3. *In order for a problem P to be in NP it has to have a polynomially balanced certificate relation R that is polynomially decidable.*
A certificate relation R is polynomially decidable if there is a polynomial-time algorithm which checks whether a pair of an instance I and a certificate C, $(I, C) \in R$.
A certificate relation R is polynomially balanced if $(I, C) \in R \Rightarrow |C| \leq |I|^k$ for some constant $k \geq 1$.

NP-complete

The class of *NP-complete* (NP-c) decision problems is a class of problems where each problem D lies in *NP* and every other NP-complete problem can be reduced to D in polynomial time. These problems are as well characterized by the property that they can be verified quickly whether any given instance I is a positive solution of D, but there is no fast algorithm known that can find all such solutions, at this time.

Definition 4. *A decision problem D is NP-complete, iff the following two conditions hold:*

1. *D is in NP*
2. *Every NP-complete decision problem can be reduced to D in polynomial time.*

co-NP-complete

Each co-NP-complete problem D has a complement problem $\bar{D}$ that is NP-complete. Before we are able to introduce the definition of co-NP-complete problems, we have to define co-NP problems.

Definition 5. *The set C of co-NP problems is defined as follows: $C = \{D \mid \bar{D} \in NP\}$ where $\bar{D}$ denotes the complement of a problem D.*

Now that we have defined co-NP problems, we can continue with the definition of co-NP-complete problems.

Definition 6. *A decision problem D is co-NP-complete iff the following two conditions hold:*

1. *D is in co-NP*
2. *Every co-NP-complete decision problem can be reduced to D by a polynomial-time many-to-one reduction.*

P-complete

Another important notion is P-completeness, for which we also would like to provide a definition.

Definition 7. *A decision problem D is P-complete if following two conditions hold*

1. *D is in P*
2. *Every P-complete decision problem can be reduced to D by an appropriate reduction*

whereas an appropriate reduction is of lower complexity than P and the specific type of reduction may vary.

Π_2^P-complete

Π_2^P-complete is a class of the *polynomial-time hierarchy* which is defined in terms of polynomial-time bounded oracle machines. Oracles are supposed to solve a problem of their respective class within a single computational step and therefore their runtime can be neglected. Now, before we define Π_2^P-completeness we introduce the definition of the polynomial-time hierarchy as described by L. J. Stockmeyer in [50].

Definition 8. *The polynomial-time hierarchy is* $\{\Sigma_k^P, \Pi_k^P, \Delta_k^P : k \geq 0\}$
where $\Sigma_0^P = \Pi_0^P = \Delta_0^P = P$ *and* P *is the set of all decision problems, solvable in polynomial time.*
For any $k > 0$ *we define:*

$$\Sigma_{i+1}^P = NP^{\Sigma_i^P}$$
$$\Pi_{i+1}^P = co\text{-}NP^{\Sigma_i^P}$$
$$\Delta_{i+1}^P = P^{\Sigma_i^P}$$

where $i \geq 0$ *and* $P^{\Sigma_i^P}$ *for instance, denotes the set of decision problems, solvable by a Turing machine in class* P*, with the help of an oracle for some complete problem in class* Σ_i^P*.*

Π_2^P stands for co-NP$^{\text{NP}}$ which means that decision problems in Π_2^P can be solved by a co-NP decision procedure, that accesses an oracle with an NP-decision procedure.

Definition 9. *Now having defined* Π_2^P*, a problem is* Π_2^P*-complete if and only if it is a member of the set of the problems in* Π_2^P *and at the same time* Π_2^P*-hard.*

2.2 Abstract Argumentation

In this section we provide the necessary background of argumentation and a detailed description of the resolution based grounded semantics. As already mentioned in the introduction, before we continue with a detailed description of abstract argumentation, we shortly describe how one can acquire arguments from the natural language in order to formally deal with arguments. Some common approaches are *defeasible logic programming* as presented in an article by A. Garcia and G. Simari [29], *defeasible argumentation with specificity-based preferences* as presented in an article by G. Simari and R. Loui [48] or *Argumentation Based on Classical Logic* as presented by P. Besnard and A. Hunter [10]. We decided to focus on the latter one, so let us start with the technique described by P. Besnard and A. Hunter.

From Natural Language to Arguments - Argumentation Based on Classical Logic

When describing a method of transforming arguments from natural language into a representation in classical logic as presented in an article by P. Besnard and A. Hunter [10], we assume that the basics of classical logic are known to the reader. We will stick to the notations used

by P. Besnard and A. Hunter: Atoms are represented by lower case roman letters $(a, b, c, \ldots)$. Formulas will be represented by greek letters such as $(\alpha, \beta, \gamma, \ldots)$. We will use $\wedge$, $\vee$, $\rightarrow$ and $\neg$ as representation for the corresponding logical connectives *conjunction*, *disjunction*, *implication*, and *negation* . $\vdash$ denotes the binary consequence relation of classical logic. A fixed Δ is assumed to constitute the knowledge-base which is a finite set of formulae. There is no a priori restriction to the contents of Δ which can be arbitrarily complex. Formulae which represent both certain and uncertain information can be contained in the knowledge-base. Any arguments and counter-arguments will be formed from this knowledge-base. There is no restriction to the knowledge-base or even its sub-formulae to be consistent. No meta-level information, such as orderings or priorities are assumed about any formulae in Δ. Now we are able to define an argument for argumentation based on classical logic.

Definition 10. *An argument A is a pair $\langle \Phi, \alpha \rangle$ such that:*

- $\Phi \nvdash \bot$.
- $\Phi \vdash \alpha$.
- Φ *is a minimal subset of* Δ *satisfying* $\Phi \vdash \alpha$.

where we say that $A = \langle \Phi, \alpha \rangle$ is an argument *for α, which is called the* claim. *Furthermore Φ is called the support for the claim α. The pair $\langle \Phi, \alpha \rangle$ itself, in general is not an element of Δ.*

Example 1. *We present an example from the article by P. Besnard and A. Hunter [10]. Consider a discussion in a newspaper editorial office about whether or not to proceed with the publication of some indiscretion about a prominent politician. Suppose the key bits of information are captured by the following five statements.*

p	*Simon Jones is a Member of Parliament*
$p \rightarrow \neg q$	*If Simon Jones is a Member of Parliament then we need not keep quiet about details of his private life*
r	*Simon Jones just resigned from the House of Commons*
$r \rightarrow \neg p$	*If Simon Jones just resigned from the House of Commons then he is not a Member of Parliament*
$\neg p \rightarrow q$	*If Simon Jones is not a Member of Parliament then we need to keep quiet about details of his private life*

Now, we are able to form arguments from Example 1. For further use, in latter sections, we will assign letters to them:

- a: $\langle \{p, p \rightarrow \neg q\}, \neg q \rangle$
- b: $\langle \{r, r \rightarrow \neg p\}, \neg p \rangle$
- c: $\langle \{r, r \rightarrow \neg p, \neg p \rightarrow q\}, q \rangle$

We picked two different kinds of attacks, *undercuts* and *rebuttals*, within the realm of argumentation which we will shortly describe with the help of the previous examples. Argument a is formed of the first two statements from Example 1 and claims that we do not have to be quiet about Simon Jones' private life. Argument b is formed of the next two statements and claims that Simon Jones is not a Member of the Parliament, hence it attacks the support of argument a. This form of attack is called undercut.

Definition 11. *An undercut for an argument $\langle \phi, \alpha \rangle$ is an argument $\langle \psi, \neg(\phi_1 \wedge \ldots \wedge \phi_n) \rangle$ where $\{\phi_1, \ldots, \phi_n\} \subseteq \phi$.*

Argument c is formed of the last three statements and claims that we have to remain quiet about Simon Jones' private life. As a result it attacks the claim of our first argument. This form of attack, where arguments have opposed claims, is called rebuttal.

Definition 12. *An argument $\langle \psi, \beta \rangle$ is a* rebuttal *for an argument $\langle \phi, \alpha \rangle$ iff $\beta = \neg\alpha$*

For more in-depth knowledge about this kind of transformations, we refer to the already mentioned article by P. Besnard and A. Hunter [10]. P. Besnard and A. Hunter [10] also introduced structures, called *argument trees*, which provide means for evaluating such formal arguments. Argument trees are formal structures which represent all possible paths an argumentation might take, originating from an initial argument represented by the root node. They are an approach of casting the process of argumentation into a formal frame. The tree structure was aptly chosen, since an argumentation process usually starts with an initial argument from which grows some claim. Most of the time such arguments give rise to more counter-arguments which in turn again give rise to counter-arguments. Since an argument quite often has multiple counter-arguments, one can see that, from a single initial argument, multiple courses of argumentation can arise, similar to a tree growing from the root. Each branch of the tree continues until the point where no subsequent counter-argument needs a new item in its support.

Definition 13. *An argument tree for a claim α, is a tree where each node is an argument such that [10]:*

1. *The root is an argument for the claim α.*
2. *For no node $\langle \phi, \beta \rangle$ with ancestor nodes $\langle \phi_1, \beta_1 \rangle, \ldots \langle \phi_n \beta_n \rangle$, where $n \geq 0$, is ϕ a subset of $\phi_1 \cup \ldots \cup \phi_n$.*
3. *The children nodes of a node N consist of all canonical undercuts for N that obey the previous statement.*

For the definition of *canonical undercuts* and a more detailed description of argument trees, the interested reader may refer to [10]. Now, in order to determine whether a root argument is justified or *warranted*, as P. Besnard and A. Hunter call it, a judge function is introduced which marks each node of the tree as undefeated U or defeated D.

Definition 14. *The judge function labels each argument tree T either* warranted *or* unwarranted *such that each T is warranted by the judge function iff $Mark(A_r) = U$ where A_r is the root node of T. For all nodes A_i in T, if there exists a child A_j of A_i such that $Mark(A_j) = U$, then it follows that $Mark(A_i) = D$, otherwise $Mark(A_i) = U$. From that it follows that the root of a tree T is undefeated if all of its children are defeated. [10]*

From that definition one can see that argument trees provide means of evaluating the justification state of an argument solely based on the tree structure. But this method of evaluating an argumentation process is confined to argumentations which can be represented by a single tree structure. In the next section we will present a more widely applicable framework which can be used for evaulating argument structures, consisting of arbitrary graphs.

Abstract Argumentation

In the previous section we described how one can transform arguments from natural language into formal arguments. An argument may have a complex internal structure, or represent several intuitive meanings. In abstract argumentation an argument does not possess any internal structure and simply is anything which attacks other arguments or may be attacked by them. Those attacks do not possess any internal structure either. They may include undercuts, rebuttals and other types of attacks, and have no other meaning. While the method by P. Besnard and A. Hunter [10] resulted in a tree, other abstraction methods might result in richer structures such as directed graphs. We now turn our attention to such objects.

The *abstract argumentation framework* as introduced in his pivotal paper by P. Dung [19], is a pair consisting of two sets. A set of arguments and a set of binary relations constituting attacks. As we already mentioned, he abstracted away any internal structure, hence, the term *abstract* argumentation framework. This simple structure of an *abstract argumentation framework* is suited for a graph representation, where the arguments are represented by nodes and directed edges constitute the attack relations. Such a graphical representation of an *abstract argumentation framework* is called a *defeat graph*. We will frequently use it in order to visualize our examples in this thesis.

Definition 15. *An* Abstract Argumentation Framework (AF) *is a pair $\langle A, R \rangle$, where A is a set of elements which are called arguments and $R \subseteq A \times A$ is a set of binary relations, called attack relations. A pair $(a, b) \in R$ means that argument a attacks argument b, which is also denoted as $a \rightarrow b$. $(a, b) \notin R$ is also denoted as $a \not\rightarrow b$.*

In *abstract argumentation* there are two special notions of attacks. It might be the case that an argument a is attacking itself, as shown in Figure 2.1. Either because its conclusion itself is contradictory or its conclusion is a contradiction to its premise. We call such a situation a *self-defeating* attack. It might also be the case that two arguments a and b are attacking each other, as depicted in Figure 2.2. Such an attack situation is called a *mutual attack*.

Figure 2.1: An attack relation $(a, a) \in R$, where an argument a attacks itself.

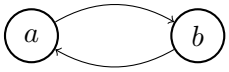

Figure 2.2: Both attack relations (a, b) and (b, a) are in R. Such attacks are called mutual.

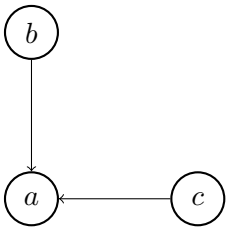

Figure 2.3: An argumentation framework, based on Example 1

Definition 16. *Let $G = \langle A, R \rangle$ be an AF. An attack relation $(a, a) \in R$, where an argument attacks itself is called self-defeating. And if both (a, b) and (b, a) are in R, those attacks are called mutual. The set of mutual attacks, of an argumentation framework G, M_G is defined as follows: $M_G = \{(a, b) \in R \mid a \neq b \wedge b \rightarrow a\}$.*

We now take a look at Example 1 which we introduced in the previous section and transform it into an abstract argumentation framework as depicted in Figure 2.3. As one can see, our argumentation framework clearly does not only represent our example. It could as well represent a totally different situation, with the same attack relations between the arguments, what basically is the idea behind abstract argumentation. The fact that we abstract away from any internal structure results in the possibility of analyzing certain properties independently, without dealing with any specific aspect of the arguments. Opposed to this advantage, we have the side effect of losing expressiveness which makes the direct use of abstract argumentation less attractive in any application context.

Abstract Argumentation Semantics

Now, that arguments may attack each other, there clearly might be arguments which are not able to stand together. Therefore the notion of the *justification state* of an argument has been introduced in the literature. Informally, an argument is *justified* if it is, solely or together with other arguments, able to survive the attacks it receives. Otherwise, the argument is regarded as

not being justified or rejected. In order to determine the state of each argument and the sets of arguments which are capable of surviving together, several semantics have been introduced which govern the evaluation process. Those semantics are formal methods which can either be declarative or procedural, and are called *abstract argumentation semantics*. Two different approaches exist for those semantics. The extension-based semantics are formal definitions of how to derive a set of extensions from an argumentation framework $AF = \langle A, R \rangle$, where a single extension is a subset of A. Whereas the labeling-based semantics are a formal definition of how to derive a set of labelings from an argumentation framework $AF = \langle A, R \rangle$. Where a labeling $L : A \to \mathbb{L}$ is the assignment of labels, taken from a predefined set of different labels $\mathbb{L}$, to arguments A of the argumentation framework.
Before we describe the different semantics, we define some notations for different properties of argument sets that we will use in our subsequent definitions. The notation we use is based on the article *On principle-based evaluation of extension-based argumentation semantics* by Baroni et al. [6] adapted and extended where needed to fit our implementation.

Definition 17. *Given an argumentation framework $AF = \langle A, R \rangle$, two subsets of arguments $S \subseteq A$ and $P \subseteq A$ and an argument $x \in A$, let us define some notations:*

- $S \to x \equiv \exists y \in S : y \to x$
- $x \to S \equiv \exists y \in S : x \to y$
- $S \to P \equiv \exists x \in S, y \in P : x \to y$
- *S^C, the complement of S is defined as $S^C = A \backslash S$*
- $x^- \equiv \{y \in A \mid y \to x\}$
- $x^+ \equiv \{y \in A \mid x \to y\}$

Definition 18. *Let $AF = \langle A, R \rangle$ be an argumentation framework and $S \subseteq A$ be a subset of its arguments. Then the set of attackers of S is denoted as $S^- = \{x \in A \mid x \to S\}$ and the set of arguments, attacked by S is denoted as $S^+ = \{x \in A \mid S \to x\}$. The range of a set S is denoted as $S^\oplus = S \cup S^+$.*

In some definitions, we will encounter the *restriction* of an argumentation framework $AF = \langle A, R \rangle$ to a subset $S \subseteq A$. It is defined as follows.

Definition 19. *The restriction of an argumentation framework $G = \langle A, R \rangle$ to a set $S \subseteq A$ is denoted as $G{\downarrow_S}$ and defined as following: $G{\downarrow_S} = \langle S, R \cap (S \times S) \rangle$.*

Definition 20. *Let $G = \langle A, R \rangle$ be an argumentation framework. A set $S \subseteq A$ is untattacked or initial iff $S^- = \emptyset$. The set of unattacked arguments in G is denoted as $\mathcal{IN}(G)$.*

Now that we have defined the notion of *unattacked* or *initial* arguments, we are able to formally define *externally unattacked sets* which are also needed later in this chapter in order to describe certain principles relying on them.

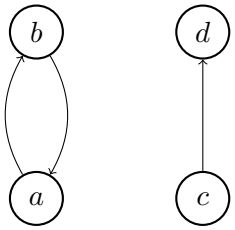

Figure 2.4: An argumentation framework $AF = \langle\{a, b, c, d\}, \{(a, b), (b, a), (c, d)\}\rangle$

Definition 21. *Given an argumentation framework $AF = \langle A, R\rangle$, then a non-empty set $S \subseteq A$ is regarded as externally unattacked iff $\nexists a \in (A \backslash S) : a \rightarrow S$. The set of* externally unattacked sets *of AF is denoted as $\mathcal{US}(AF)$.*

Labeling-Based Semantics

In this thesis, we will work with *extension-based semantics*, but in order to provide a proper overview of abstract argumentation, we now shortly introduce the *labeling-based semantics*. The basic idea underlying this semantics is to assign labels to each argument of an argumentation-framework from a predefined set of labels. An example for such a set of labels might be $\{\text{in}, \text{out}, \text{undecided}\}$. The labeling-based semantics are formal definitions of how to derive a set of labelings from an argumentation framework $AF = \langle A, R\rangle$, where a labeling $L : A \rightarrow \mathbb{L}$ is the assignment of labels, taken from a predefined set of different labels $\mathbb{L}$ to arguments A of the argumentation framework. The set of all possible label mappings from A to $\mathbb{L}$ is denoted as $\mathcal{L}(A, \mathbb{L})$. The labelings of an abstract argumentation framework $AF = \langle A, R\rangle$ with respect to a semantics S, is a subset of $\mathcal{L}(A, \mathbb{L})$ and denoted as $\mathcal{L}_S(AF) \subseteq \mathcal{L}(A, \mathbb{L})$. We present a short example, based on the argumentation framework, as depicted in Figure 2.4:

Example 2. *Let S_1 be a hypothetical labeling-based semantics, where an argument is labelled* in *only if it receives no attacks, and is labelled* out *if it is attacked by arguments labelled* in *and otherwise labelled* undecided*. Let $AF_1 = \langle\{a, b, c, d\}, \{(a, b), (b, a), (c, d)\}\rangle$ be an argumentation framework. Then applying semantics S_1 on our example AF_1 would result in following labeling $\mathcal{L}_{S_1}(AF_1) = \{\{(a, \textit{undecided}), (b, \textit{undecided}), (c, \textit{in}), (d, \textit{out})\}\}$.*

For further examples and a more comprehensive overview of *labeling-based* semantics, we refer to [4].

Extension-Based Semantics

An extension-based argumentation semantics is a formal definition of principles ruling the argument evaluation process and specifying which subsets of arguments of an argumentation framework $AF = \langle A, R\rangle$ are acceptable and able to survive together. Such a set is called an *extension*.

Definition 22. *Let $AF = \langle A, R\rangle$ be an argumentation framework and $\mathcal{S}$ a generic argumentation semantics, then the set of extensions prescribed by S for AF is denoted as $\mathcal{E}_{\mathcal{S}}(AF)$. $\mathcal{D}_{\mathcal{S}} = \{G \mid \mathcal{E}_{\mathcal{S}}(G) \neq \emptyset\}$ denotes the set of argumentation frameworks for which, with respect to semantics $\mathcal{S}$, at least one extension exists. If for a semantics $\mathcal{S}$ all argumentation frameworks belong to $\mathcal{D}_{\mathcal{S}}$, it is said to be* universally defined.

Now if there is exactly one extension by a semantics, this semantics belongs to the *unique-status approach*. Otherwise, if there are more than one extension prescribed by a semantics, it belongs to the *multiple-status approach*.

Definition 23. *A semantics belongs to the* unique status approach *if it fulfills the following criterion: $\forall G \in \mathcal{D}_{\mathcal{S}} : |\mathcal{E}_{\mathcal{S}}(G)| = 1$. Otherwise it belongs to the* multiple-status approach.

As we already noted, extensions describe the justification state of an argument. When we take the extension membership to distinguish the different justification states, we basically have two different types of *justification*. If an argument is contained in each extension of an argumentation framework, it is regarded as *skeptically justified*. We say that an argument is *credulously justified* if it is part of at least one extension. It can be deduced from the above definitions, that for unique-status approaches these two classifications coincide and therefore several refinements to this particular justification state exist. For further justification states, the reader may refer to [7], where a more detailed overview is provided.

Definition 24. *Let $AF = \langle A, R\rangle$ be an argumentation framework and $\mathcal{S}$ be a semantics, then an argument $x \in A$ is* skeptically justified *in AF w.r.t. $\mathcal{S}$ iff $\forall E \in \mathcal{E}_{\mathcal{S}}(AF) : x \in E$. An argument $x \in A$ is* credulously justified *iff $\exists E \in \mathcal{E}_{\mathcal{S}}(AF) : x \in E$.*

Principles for Extension-Based Semantics

Before introducing the different extension-based semantics of abstract argumentation frameworks and explaining the motivation that drove the development of the resolution-based grounded semantics, we will provide a general set of desirable properties as presented in [6], which are shared by some or all semantics. When presenting these principles, we will distinguish between properties of individual extensions or the whole set of extensions. First of all, we introduce the two most important principles, underlying the definition of extension-based semantics. These are *language independence* and the *conflict free* principles.

Definition 25. *Two argumentation frameworks $AF_1 = \langle A_1, R_1\rangle$ and $AF_2 = \langle A_2, R_2\rangle$ are called isomorphic which is denoted as $AF_1 \stackrel{\circ}{=}_m AF_2$, iff there exists a bijective mapping $m : A_1 \rightarrow A_2$ such that $(a, b) \in R_1, (m(a), m(b)) \in R_2$.*

Intuitively, the language independence principle stands for the fact that any extension prescribed by a semantics, solely depends on the attack relations between arguments. It essentially describes the idea of abstract argumentation, where our results only depend upon the structure of the defeat graph, described by an abstract argumentation framework.

Definition 26. *A semantics $\mathcal{S}$ satisfies the* language independence *principle if and only if* $\forall AF_1 \in \mathcal{D}_{\mathcal{S}}, \forall AF_2 \in \mathcal{D}_{\mathcal{S}} : AF_1 \stackrel{\circ}{=}_m AF_2, \mathcal{E}_{\mathcal{S}}(AF_2) = \{M(E) \mid E \in \mathcal{E}_{\mathcal{S}}(AF_1)\}$*, where* $M(E) = \{y \mid \exists x \in E, y = m(a)\}$.

The *conflict free* principle intuitively describes the idea of extensions being sets of arguments that are able of being acceptable to one another meaning that within their arguments there are no conflicts or attacks, respectively.

Definition 27. *Let $AF = \langle A, R \rangle$ be an argumentation framework and $P \subseteq A$ a subset of A. Then P is* conflict-free *in AF, denoted as $cf(P)$, iff $\nexists x, y \in P$ such that $x \rightarrow y$. The* conflict-free principle *is satisfied by a semantics $\mathcal{S}$ iff $\forall AF \in \mathcal{D}_{\mathcal{S}}, \forall E \in \mathcal{E}_{\mathcal{S}}(AF)$ E is conflict-free in AF.*

As stated in [6] all extension-based semantics proposed in the literature adhere to the *conflict free* principle. So does the *resolution-based grounded semantics*, introduced in [5] which we focus on in this thesis.

One of the basic principles is based on the expectations towards the arguments of an extension to be able to "survive attacks together" or in other words "to stand on their own". This requires the extension to be free of conflicts and it requests the arguments of an extension to attack each attack that it is receiving from outside. In order to formally define the principle of *admissibility* we first have to introduce the notion of an argument being acceptable with respect to a set $S \subseteq A$.

Definition 28. *Let $AF = \langle A, R \rangle$ be an argumentation framework, $a \in A$ an argument and $S \subseteq A$ a subset of the frameworks arguments. Then a is acceptable w.r.t. S in AF iff $\forall b \in A$: $b \rightarrow a \Rightarrow S \rightarrow b$.*

Now, that we have introduced the notion of acceptability of arguments w.r.t. a semantics in an argumentation framework, we are ready to describe a function that returns the set of all acceptable arguments of an AF w.r.t. a semantics $\mathcal{S}$.

Definition 29. *Let $AF = \langle A, R \rangle$ be an argumentation framework, then $\mathcal{F}_{AF} : 2^A \rightarrow 2^A$ is a function, called the* characteristic function *which, given a subset $S \subseteq A$ returns the set of acceptable arguments wrt. S in an AF.*

Having defined the acceptability of an argument and the characteristic function, we now define the *admissibility* of a set $S \subseteq A$:

Definition 30. *Let $AF = \langle A, R \rangle$ be an argumentation framework, then a set $S \subseteq A$ is* admissible*, iff it is conflict-free and $\forall a \in S : a \in \mathcal{F}_{AF}(S)$. The set of* admissible *sets of an argumentation framework AF is denoted as $\mathcal{AS}(AF)$.*

Having defined the *admissibility* of a set, we are now able to formally introduce the principle of *admissibility* for semantics of an abstract argumentation frameworks.

Definition 31. *A semantics $\mathcal{S}$ satisfies the* admissibility principle *iff* $\forall AF \in \mathcal{D}_{\mathcal{S}} : \mathcal{E}_{\mathcal{S}}(AF) \subseteq \mathcal{AS}(AF)$*, or stated differently,* $\forall E \in \mathcal{E}_{\mathcal{S}}(AF) : a \in E \Rightarrow (\forall b \in \{a\}^- : E \rightarrow b)$.

The *reinstatement principle* intuitively states that if the attackers of an argument a are in turn attacked by an extension E, then they do not have any effect on the justification state of a and a therefore can be seen as *reinstated* and be a part of the extension itself. The *reinstatement principle* constitutes the converse to the implication of the *admissibility principle*.

Definition 32. *A semantics $\mathcal{S}$ satisfies the* reinstatement principle *iff* $\forall AF \in \mathcal{D}_{\mathcal{S}}, \forall E \in \mathcal{E}_{\mathcal{S}}(AF) : (\forall b \in A, b \rightarrow a \Rightarrow E \rightarrow b) \Rightarrow a \in E$.

A further principle deals with another constraint towards the extensions of a semantics adhering to it. It deals with possible inclusion relationships and requires no extension within a set of extensions that can be a proper subset of another one. The *I-maximality principle* is fundamental when the evaluation of *skeptical justification* is desired.

Definition 33. *A set of extensions $\mathcal{E}$ is I-maximal iff* $\forall E_1, E_2 \in \mathcal{E} : E_1 \subseteq E_2 \Rightarrow E_1 = E_2$*. A semantics $\mathcal{S}$ satisfies the* I-maximality principle *iff* $\forall AF \in \mathcal{D}_{\mathcal{S}} : \mathcal{E}_{\mathcal{S}}$ *is I-maximal.*

The *directionality criterion* intuitively deals with the idea, that an argument's justification state shall only be affected by the justification state of its defeaters, of their defeaters justification state and so on. Whereby the arguments that were defeated by this argument should not have any effect on its justification state. Therefore, we can define the directionality criterion by imposing this requirement on any unattacked set of an argumentation framework to be unaffected by any remaining argument outside the set.

Definition 34. *A semantics $\mathcal{S}$ satisfies the* directionality criterion *iff* $\forall AF \in \mathcal{D}_{\mathcal{S}}, \forall U \in \mathcal{US}(AF) : \mathcal{AE}_{\mathcal{S}}(AF, U) = \mathcal{E}_{\mathcal{S}}(AF \downarrow_U)$*, where* $\mathcal{AE}_{\mathcal{S}}(AF, U) \triangleq \{(E \cap U) \mid E \in \mathcal{E}_{\mathcal{S}}(AF)\} \subseteq 2^U$.

In order to define the last two principles that we will need in order to describe the motivation behind the introduction of the *resolution-based grounded semantics*, we first have to introduce the notion of *skepticism*, which is elaborately defined by P. Baroni and M. Giacomin in [6]. Intuitively this notion is based on the idea that an abstract argumentation semantics is more skeptical than another semantics, if it makes less committed choices about the justification state of arguments in an argumentation framework. This means that this semantics is more likely to leave the justification state of an argument *undecided*.
We define a generic *skepticism-relation* $\preceq^E$ between two sets of extensions of an argumentation framework $AF = \langle A, R \rangle$: Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two arbitrary extensions of AF, then $\mathcal{E}_1 \preceq \mathcal{E}_2$ means, that $\mathcal{E}_1$ is at least as skeptical as $\mathcal{E}_2$. Note that, as stated in [6], these skepticism relations impose a partial, but not necessarily a total order, which means that there might be two sets of extensions which are not comparable.

Definition 35. *Two sets of extensions, $\mathcal{E}_1$ and $\mathcal{E}_2$ fulfill relation $\mathcal{E}_1 \preceq^E_\cap \mathcal{E}_2$ iff*

$$\bigcap_{E_1 \in \mathcal{E}_1} E_1 \subseteq \bigcap_{E_2 \in \mathcal{E}_2} E_2$$

$\mathcal{E}_1 \preceq^E_\cap \mathcal{E}_2$ represents the fact that arguments which are skeptically justified according to $\mathcal{E}_1$ are also skeptically justified according to $\mathcal{E}_2$.

Definition 36. *Two sets of extensions, $\mathcal{E}_1$ and $\mathcal{E}_2$, fulfill $\mathcal{E}_1 \preceq^E_W \mathcal{E}_2$ iff $\forall E_2 \in \mathcal{E}_2 \exists E_1 \in \mathcal{E}_1 : E_1 \subseteq E_2$.*

Definition 37. *Given an argumentation framework AF, two sets of extensions, $\mathcal{E}_1$ and $\mathcal{E}_2$ of AF, fulfill relation: $\mathcal{E}_1 \preceq^E_S \mathcal{E}_2$ iff $\mathcal{E}_1 \preceq^E_W \mathcal{E}_2$ and $\forall E_1 \in \mathcal{E}_1 \exists E_2 \in \mathcal{E}_2 : E_1 \subseteq E_2$.*

The skepticism-relation $\preceq^A$ is based on the following intuition: Consider one argumentation framework $AF = \langle A, R \rangle$ with two arguments $a, b \in A$ and a single attack $(a, b) \in R$. In that argumentation framework, any semantic would reject b and accept a with the highest commitment. Now if we add an attack (b, a) to R, we clearly create an argumentation framework where only a more skeptical commitment is appropriate. Generalizing this observation by transforming unidirectional attacks into mutual ones results in less committed justification states of the nodes involved.

Definition 38. *Let $AF = \langle A, R \rangle$ be an argumentation framework, then $\mathcal{CONF}(AF) \triangleq \{(a, b) \in R \mid a \to b \vee b \to a\}$.*

Before we are able to define the skepticism-relation $\preceq^A$, we have to define $\mathcal{CONF}(AF)$ the set of undirectional conflicts of an argumentation framework.

Definition 39. *Given two argumentation frameworks $AF_1 = \langle A_1, R_1 \rangle$ and $AF_2 = \langle A_2, R_2 \rangle$, then $AF_1 \preceq^A AF_2$ iff $\mathcal{CONF}(AF_1) = \mathcal{CONF}(AF_2)$ and $R_2 \subseteq R_1$.*

Equipped with preceding definitions, we are able to define the notion of skepticism adequacy which guarantees the preservation of the skepticism-relation when applying a semantics $\mathcal{S}$ to two argumentation frameworks subject to this relation. In other words skepticism adequacy states that if a skepticism-relation between two argumentation frameworks is present, the same relation also holds for their extensions.

Definition 40. *A semantics $\mathcal{S}$ is $\preceq^E$-*skepticism-adequate*, given a skepticism-relation $\preceq^E$ between sets of extensions, denoted as $\mathcal{SA}_{\preceq^E}(\mathcal{S})$ iff for any pair of argumentation frameworks AF and AF' such that $AF \preceq^A AF'$ the following condition holds: $\mathcal{E}_\mathcal{S}(AF) \preceq^E \mathcal{E}_\mathcal{S}(AF')$*

Since $\preceq^A$ consists of an equality and a set inclusion relation it describes a partial order which implies that there exist argumentation frameworks which are not comparable and within the comparable AFs multiple maximal AFs w.r.t. $\preceq^A$ can exist [6].

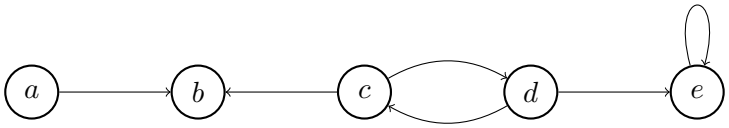

Figure 2.5: $AF = \langle\{a, b, c, d, e\},\{(a, b),(c, b),(c, d),(d, c),(d, e),(e, e)\}\rangle$

Definition 41. *Let $AF = \langle A, R\rangle$ be an argumentation framework, then $\mathcal{RES}(AF)$ denotes the set of argumentation frameworks which are comparable to AF and maximal w.r.t. $\preceq^A$.*

As last principle we describe the resolution adequacy criterion, as presented in [6], which intuitively states that an argument should be skeptically justified in AF, if it is skeptically justified in every possible resolution of AF, which is formally stated by the $\Leftarrow$ part of the following formula and proposed by S. Mogdil in [41]:

$$a \in \bigcap_{E \in \mathcal{E}_S(AF)} E \Leftrightarrow \forall \text{AF'} \in \mathcal{RES}(\text{AF}) : a \in \bigcap_{E \in \mathcal{E}_S(AF)} E$$

Now Baroni et al. [6] provide a generalized version of this formula which is made parametric w.r.t. skepticism relations between sets of extensions by the use of the following intermediary formulation and: $\mathcal{UR}(\text{AF}, S) = \bigcup_{\text{AF'} \in \mathcal{RES}(\text{AF})} \mathcal{E}_S(\text{AF'})$

Definition 42. *Given a skepticism relation $\preceq^E$ between sets of extensions, a semantics $\mathcal{S}$ is $\preceq^E$resolution adequate, denoted $\mathcal{RA}_{\preceq^E}(\mathcal{S})$, iff for any argumentation framework it holds that $\mathcal{UR}(AF, S) \preceq^E \mathcal{E}_\mathcal{S}(AF)$.*

Different Extension-Based Argumentation Semantics

Now that we have presented the different principles for extension-based argumentation semantics among Dung's traditional semantics [19], we now provide a collection of different extension-based semantics, including the resolution-based grounded semantics which is the only one fulfilling all the principles as presented in the previous section.
We introduce an example that we borrowed from [22] to point out the differences of the various semantics:

Example 3. *Our example abstract argumentation framework AF consists of following arguments $A = \{a, b, c, d, e\}$ and attack relations $\{(a, b), (c, b), (c, d), (d, c), (d, e), (e, e)\}$.*

The intuition of the *stable semantics* which is a *multiple-status approach*, is quickly explained. The extensions of this semantics are *conflict-free* and they attack every argument outside the extension. More formally it is defined as follows:

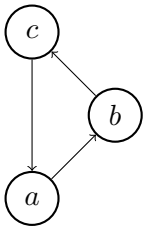

Figure 2.6: An argumentation framework $AF = \langle\{a, b, c\}, \{(a, b), (b, c), (c, a)\}\rangle$

Definition 43. *Let $AF = \langle A, R\rangle$ be an argumentation framework, then a set $S \subseteq A$ is a* stable extension *w.r.t. AF iff S is conflict-free and $\forall a \in A : a \notin S \Rightarrow S \rightarrow a$.*

When taking a look at our example 2.5, we have the following *conflict-free* sets: $\{\{a, c\}$, $\{a, d\}$, $\{b, d\}$, $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\emptyset$ $\}$. But only $\{a, d\}$ does attack all arguments of AF which are outside the extension.

Lets define the term *complete extension*, but since it is also frequently referred to as *complete semantics* in current literature as mentioned in [45], we will too stick to this term in this thesis. Its notion is based on the idea that its extension should be *conflict-free*, able to defend itself and include all arguments that it receives. Hence, the *complete semantics* fulfills the principles of *admissibility* and *reinstatement* and by P. Baroni and M. Giacomin [6] it has been proven to satisfy the *directionality principle*. The *complete extension* is a semantics which is a *multiple-status approach*

Definition 44. *Let $AF = \langle A, R\rangle$ be an argumentation framework, then a set $S \subseteq A$ is a* complete extension *of AF iff $S \in \mathcal{AS}(AF) \wedge \mathcal{F}_{AF}(E) \subseteq S$ or, in words, if and only if S is admissible and each argument of A which is acceptable wrt. S belongs to S*

Due to the fact that the *complete semantics* does include the initial arguments, the arguments defended by them, and the arguments defended by the defended arguments, in some special case the *complete semantics* includes the empty extension. In the case, where no argument is not attacked as can be seen in our Example 2.5, we have three resulting extensions: $\{\{a, c\}, \{a, d\}, \{a\}\}$.

Example 4. *For the argumentation framework $AF = \langle\{a, b, c\}, \{(a, b), (b, c), (c, a)\}\rangle$ we have the following admissible sets: $\mathcal{AS}(AF) = \{\{\emptyset\}\}$ and trivially every argument defended by the empty set belongs to the empty set, so the resulting complete extensions are: $\{\{\emptyset\}\}$.*

We already introduced the *stable semantics* which relies on the somewhat strict requirement that any extension S of an argumentation framework $AF = \langle A, R\rangle$ has to attack all the arguments $A \backslash S$ outside the extension. The *preferred extension* differs here by only specifying the extensions to be as large as possible and to be able to defend themselves. More formally:

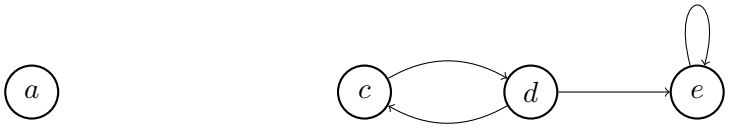

Figure 2.7: $AF_1 = \langle\{a, c, d, e\}, \{(c, d), (d, c), (d, e), (e, e)\}\rangle$

Definition 45. *Let $AF = \langle A, R\rangle$ be an argumentation framework, then a set $S \subseteq A$ is a* preferred extension *w.r.t. AF iff S is a maximal (w.r.t. set inclusion) element of $\mathcal{AS}(AF)$.*

Like the *stable semantics*, the *preferred semantics* is a traditional *multiple-status approach*. In Example 2.5 $AF = \langle\{a, b, c, d, e\}, \{(a, b), (c, b), (c, d), (d, c), (d, e), (e, e)\}\rangle$ does yield following *admissible sets*: $\{\{a, c\}, \{a, d\}, \{a\}, \{c\}, \{d\}, \emptyset\}$ of which only $\{\{a, c\}, \{a, d\}\}$ are maximal and hence preferred extensions.

The notion of the *grounded extension* which is a unique-status approach, is based on the intuition that, starting with the set $S = \mathcal{IN}(AF)$ of initial arguments of an argumentation framework AF, we incrementally increase S. We do so by suppressing the arguments attacked by S in a new, modified argumentation framework and identifying a possible new set of initial arguments. Those arguments are then added to S and the whole procedure is repeated until we reach a point where no new initial arguments are found. In fact this procedure corresponds to the iterated application of the characteristic function $\mathcal{F}_{AF}$ which is applied until a fixed point of $\mathcal{F}_{AF}$ has been reached.

Definition 46. *Let $AF = \langle A, R\rangle$ be an argumentation framework, then the* grounded extension *of AF is the least fixed point of the characteristic function $\mathcal{F}_{AF}$. The* grounded extension *of an argumentation framework AF is denoted as $\mathcal{GE}(AF)$.*

For any finitary argumentation framework $AF = \langle A, R\rangle$ it is proven in [19] that $\mathcal{GE}(AF) = \bigcup_{i=1}^{\infty} \mathcal{F}^i(\emptyset)$. (Let $AF = \langle A, R\rangle$ be an argumentation framework, then AF is finitary if for any $x \in A$, $\{x\}^-$ is finite.)

We will show you how to acquire the *grounded extension* of an argumentation framework, based on our example 2.5 $AF = \langle\{a, b, c, d, e\}, \{(a, b), (c, b), (c, d), (d, c), (d, e), (e, e)\}\rangle$. The set S of initial arguments for AF is $\{a\}$. Then we construct a new argumentation framework $AF_1 = \langle\{a, c, d, e\}, \{(c, d), (d, c), (d, e), (e, e)\}\rangle$, shown in Figure 2.7, by suppressing the arguments attacked by a.
At this step, the set of initial arguments does not grow. Hence the resulting extension simply consists of the argument $\{a\}$

Besides the four traditional Dung's semantics which we already explained in more detail, there exist various other semantics in abstract argumentation. Those further semantics include the *CF2-semantics*, introduced by P. Baroni et al. [8] which is based on conflict-freeness and the graph-theoretical notion of *strongly connected components* (SCCs). Other examples are for instance the *semi-stable semantics*, further examined by M. Caminada in [16]. Furthermore,

the *ideal-semantics* as presented by P. M. Dung et al. in [20], which provides a unique-status approach that is based on the admissibility and skeptical justification under preferred semantics, and the *prudent-semantics* as introduced by S. Coste-Marquis et al. [17] which considers a more extensive notion of attack.
Finally, there is the *resolution-based grounded semantics* [5] for which we provide a more detailed description.

Resolution-Based Grounded Semantics

In section 2.2 we have already introduced all the desirable properties for abstract argumentation semantics and we have already presented the various existing semantics. Unfortunately they do not satisfy the whole set of properties. In their article, P. Baroni et al. [5] asked themselves the question, as to whether those desirable properties are even achievable by one single semantics and whether it was feasible and practical to drive the definition of abstract argumentation semantics by formal criteria rather than basic intuitions. They answered both of these questions positively and introduced the *resolution-based grounded semantics* which possesses all the desirable properties.
Before we are able to define the *resolution-based grounded semantics*, we have to define the notion of *resolution* of an argumentation framework. This notion is based on the idea that each mutual attack represents an undecided situation in the abstract argumentation framework. These undecided situations are resolved by *resolution*, where one of the arguments in the mutual-attack is suppressed in favor of the other argument, thereby converting a mutual attack into an unidirectional one.

Definition 47. *Let $G = \langle A, R\rangle$ be an AF. A* partial resolution *of this AF is defined as any subset $\beta \subset M_G$ where, if $(x, y) \in \beta$ then $(y, x) \notin \beta$. The AF where partial resolution is applied, is denoted as $G_\beta = \langle A, R\backslash\beta\rangle$. A* full resolution *$\gamma$ is any partial resolution β where exactly one of each mutual attacks in M_G occurs or $M_{G_\gamma} = \emptyset$ respectively. $\mathcal{FR}(G)$ is the set of all full resolutions of G. The set of argumentation frameworks, resulting from the set of full resolutions is denoted as $\mathcal{FRAF}(G) = \{G_\beta \mid \beta \in \mathcal{FR}(G)\}$.*

Definition 48. *Let $AF = \langle A, R\rangle$ be an argumentation framework. Then a set $\mathcal{S} \subseteq A$ is called a resolution-based grounded extension of AF, denoted as $\mathcal{E}_{\mathcal{GR}*}(AF)$, if there exists a resolution β such that $\mathcal{GE}(A, R\backslash\beta) = \mathcal{S}$ and $\nexists\beta'$ for which $\mathcal{GE}(A, R\backslash\beta') \subset \mathcal{S}$.*

We will provide a short example, which will show how the *resolution-based grounded semantics* works.

Example 5. *The example framework 2.8 is composed as follows:*

$$AF = \langle\{a, b, c, d, e, f\}, \{(a, b), (c, b), (d, b), (c, d), (d, c), (c, e), (d, e), (e, f)\}\rangle$$

There is exactly one mutual attack between c and d. From that we derive following two, non-empty resolutions $\mathcal{FR}(AF) = \{\{(c, d)\}, \{(d, c)\}\}$. Now we determine the grounded extensions of

$$AF_1 = \langle\{a, b, c, d, e, f\}, \{(a, b), (c, b), (d, b), (d, c), (c, e), (d, e), (e, f)\}\rangle$$

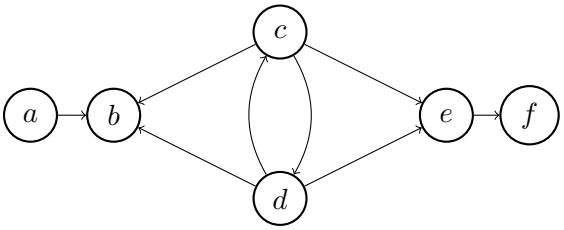

Figure 2.8: An argumentation framework to illustrate the application of resolution based grounded semantics [22]

and

$$AF_2 = \langle \{a, b, c, d, e, f\}, \{(a, b), (c, b), (d, b), (c, d), (c, e), (d, e), (e, f)\} \rangle$$

for which we get following results: $\mathcal{GE}(AF_1) = \{a, d, f\}$ *and* $\mathcal{GE}(AF_2) = \{a, c, f\}$*, which compose the extension of the* resolution-based grounded semantics*:* $\{\{a, d, f\}, \{a, c, f\}\}$.

Computational Properties of the Semantics

Since we are interested in the performance of our encoding of the *resolution-based grounded semantics*, we will provide a selection of the computational properties of the four traditional semantics and the *resolution-based grounded semantics*. We focus on the complexity results of three decision problems for a semantics σ which arise in the field of abstract argumentation.

1. *Credulous Acceptance* Cred_σ: Let $G = \langle A, R \rangle$ be an AF and $a \in A$, an argument of AF. Is a in at least one $S \in \sigma(G)$?

2. *Skeptical Acceptance* Skept_σ: Let $G = \langle A, R \rangle$ be an AF and $a \in A$, an argument of AF. Is a element of each $S \in \sigma(G)$?

3. *Verification of an extension* Ver_σ: Let $G = \langle A, R \rangle$ be an AF and $S \subseteq A$ be a set. Is $S \in \sigma(G)$?

The computational properties of the stable and preferred semantics have been summarized by P. E. Dunne and M. Wooldridge in [21]. For further information on the complexity results of the complete semantics we refer to the article of S. Coste-Marquis et al. [18]. We obtained the computational properties of the grounded semantics from [23], an article by W. Dvořák and S. Woltran. When introducing the resolution-based grounded semantics in [5], Baroni et al. provided elaborate proofs for its computational properties.

Relation between Semantics

In Figure 2.9 we depict the *is-a relationship*, between the different semantics of abstract argumentation. The semantics, which we described in more detail, are highlighted in Figure 2.9. Other semantics which we do not present in this figure are not dealt with in this thesis.

	complete	stable	grounded	preferred	GR*
Cred_σ	NP-c	NP-c	P-c	NP-c	NP-c
Skept_σ	P-c	NP-c	P-c	Π_2^P-c	coNP-c
Ver_σ	L	P-c	P-c	coNP-c	P

Table 2.1: Complexity table of abstract argumentation semantics, where *-c* stands for the completeness of the respective class

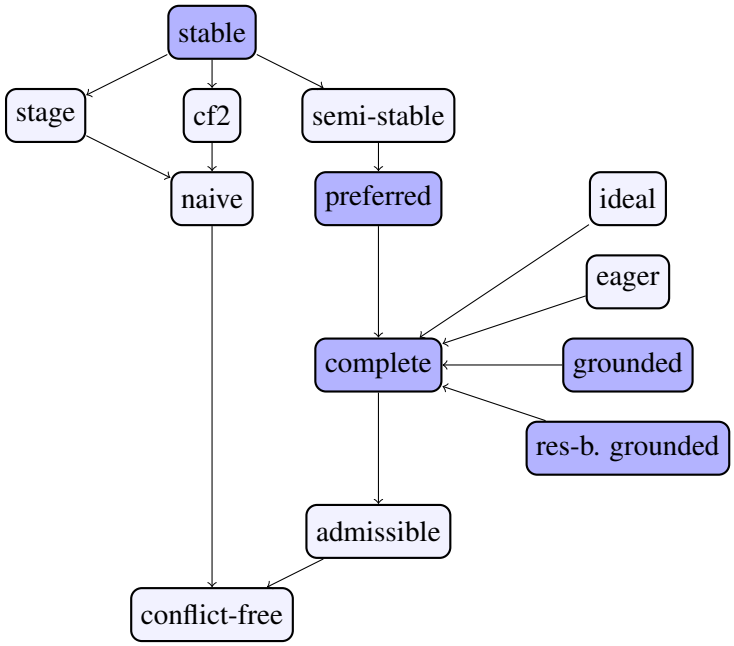

Figure 2.9: Relation between abstract argumentation semantics.

2.3 Answer-Set Programming

In this section we present the necessary background knowledge on answer-set programming, its extensions and modeling techniques. We start by explaining how ASP is based on the stable model semantics as introduced by M. Gelfond and V. Lifschitz in [36] and introduce the notational conventions that we are going to use throughout this thesis when dealing with answer-set programming. Then we present some of the language extensions of ASP: Integrity constraints, choice rules, weight constraints and optimization statements. We continue by describing the architecture of an ASP solver and conclude the background of answer-set programming with the presentation of a variety of different modeling techniques.

> Answer set programming (ASP) is a form of declarative programming oriented towards difficult search problems. [39]

Answer-Set Programming is a *declarative problem solving* approach with origins in Nonmonotonic Reasoning and Logic Programming, where as opposed to imperative programming, one formalizes the problem and a problem instance and transfers the task of finding a solution to the answer-set programming framework. Answer-Set Programming is based on the stable model semantics as introduced by M. Gelfond and V. Lifschitz in [36], a semantics for normal logic programs with negation, and on the default theories described by R. Reiter in [47].

But before we start to formally describe answer-set programming, we introduce the notational conventions that we use throughout this thesis. Furthermore we suppose that the reader has a basic understanding of Logic and Logic Programming (especially of classical and first order logic). $\top$ and $\bot$ stands for *true* and *false*. $\leftarrow$ stands for the conditional *if* in logic programs, , and "," stands for the conjunction. The disjunction simply is denoted by $\vee$. The default-negation is denoted by not or $\sim$.

The formal definition of answer-set programming is based on the extension of *classical logic programs* which are also called *horn logic programs*, as described in [27]. Classical logic programs are programs that are built of rules, called *horn clauses* which are clauses with exactly one positive literal, called head.

Definition 49. *A classical logic program P is a finite set of clauses of the following form:*

$$A \leftarrow B_1, \ldots, B_m$$

where A and $B_1, \ldots, B_m$ are atoms of a first-order language Σ. Literal A is also called the head of the rules, where $B_1, \ldots, B_m$ is called the body. Rules with an empty body are called facts.

These logic programs are the problem specification and the problem instances. The solutions to the problem are represented by models of those logic programs. These solutions are called stable models or answer sets.

Definition 50. *Given a logic program P and Σ the signature of P, then the* Hebrand Universe *of P, denoted as $\mathcal{HU}(P)$, is the set of all terms which can be formed over all constants in Σ.*
The Herbrand Base *of P, denoted as $\mathcal{H}(P)$, is the set of all ground atoms which can be formed from predicates occurring in P and the terms in $\mathcal{HU}(P)$.*
The Herbrand Interpretation *$\mathcal{I}$, is an interpretation over the Herbrand Universe $\mathcal{HU}(P)$ and a subset $\mathcal{I} \subseteq \mathcal{H}(P)$.*
A Herbrand Interpretation $\mathcal{I}$ is a Herbrand Model *M of P, if $M \models_\Sigma P$.*
$\mathcal{M}(P)$ is the set of all Herbrand Models of P.

Since in classical logic programs we deal with atoms of a first-order language we have to ground these programs in order to determine their models. The process of grounding can be regarded as a universal quantification over all variable occurrences.

Definition 51. *A ground instance of a horn clause C, as defined above, is any clause C obtained by the following substitution, which we denote as $grnd$:*

$$grnd : VAR(C) \rightarrow \mathcal{HU}(P)$$

where $VAR(C)$ denotes the set of variables, contained in a clause C and $\mathcal{HU}(P)$ denotes the Herbrand universe.
The ground instantiation of a program P is denoted by $grd(P) = \bigcup_{r \in P} grd(r)$. [27]

For each rule r of the program P, $grd(r)$ denotes the set of ground instances of a rule r. It is obtained by replacing all variables in r by ground terms, until no variable is left.

The solution of a classical logic program is represented by the *smallest Herbrand model*, which can be determined by the use of the fixed point semantics on the grounded instance of a logic program, which intuitively is an iterative approach, where in each step new atoms are derived from the rules and facts of the programs, as well the previously derived atoms until we can not derive any more atoms. When determining the least fixed point we apply a monotonic, *immediate consequence operator* T_P as elaborately explained in [27].

Definition 52. $T_P : 2^{\mathcal{H}(P)} \rightarrow 2^{\mathcal{H}(P)}$ *is an operator, defined as follows:*

$$T_P(I) = \left\{ A \;\middle|\; \begin{array}{c} \text{there exists a rule } A \leftarrow B_1, \ldots, B_m \in grnd(P) \\ \text{such that} \{B_1, \ldots, B_m\} \subseteq I \end{array} \right.$$

where $T_P^0 = \emptyset$ and $T_P^{i+1} = T_P(T_P^i)$ for $i \geq 0$. [27]

The least fixed point $lfp(P)$ of a program P is its smallest Herbrand model and therefore its result.

Definition 53. *The recursive application of T_P, $T_P^{i+1} = T_P(T_P^i)$ for $i \geq 0$ has a least fixed point $lfp(T_P)$ and converges to it. There exists no model $I \in \mathcal{M}(P)$ such that $I \subset lfp(P)$.*

When dealing with knowledge representation and reasoning, an important notion is the *Closed World Assumption*, as described in [46]. The closed world assumption states that anything that is not explicitly known to be true is assumed to be false.

> Under the closed world assumption, certain answers are admitted as a result of failure to find a proof. More specifically, if no proof of a positive ground literal exists, then the negation of that literal is assumed to be true. [46]

Definition 54. *The* closed world assumption *(CWA) is the assumption that anything which is not known to be true, is assumed to be false. In other words, any proposition that fails to be proven, leads to its negation being true.*

Classical logic programs are subject to the closed world assumption. This changes in normal logic programs, with the introduction of the *negation-as-failure*. Negation-as-failure extends classical logic programming by the possibility of directly dealing with incomplete information. Hence, the closed world assumption is no longer applied automatically to all predicates. We distinguish between those predicates which are explicitly false, and those which fail to be proven. Whereas in classical logic programs, we only can derive a single model, the smallest *Herbrand model*, in normal logic programs we can get multiple *stable models*.

Answer-Set Programming is based on extended logic programs, which extend normal logic programs by the notion of classical negation. Now, an extended logic program P is a set of rules of the following form:

$$r : H \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m.$$

where H is called head of r:

$$head(r) = H$$

$A_1, \ldots, A_n, B_1, \ldots, B_m$ are literals and not stands for the default negation. A literal l is either an atom p or a negated atom $\neg p$. An atom p is an expression $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms. In an extended logic program $not\ A$ is true for a literal A if A is not explicitly $true$, whereas $\neg A$ is true if A explicitly is false. $pos(r) = \{A_1, \ldots, A_n\}$ is the set of positive literals, $neg(r) = \{B_1, \ldots, B_m\}$ is the set of negative literals. The literals $P(t_1, ..., t_n)$ and $\neg P(t_1, ..., t_n)$ are called *complementary*. If l is a literal, the complementary literal is denoted as $\bar{l}$. A set of literals is called consistent, if it does not contain any complementary literals. A consistent set is also called a *state*. [9]
A state S of an extended logic program P with no *default negation* is closed w.r.t. P if for each rule $r \in P$ it holds that if $pos(r) \subseteq S$ it follows that $head(r) \cap S \neq \emptyset$.

The programs we are dealing with are ASP programs, which are guaranteed to have finite ground programs. Therefore rule safety is necessary, which requires that each rule r of a program P fulfills the following criterion:

Definition 55. *A rule is* safe *if each variable in that rule appears in at least one positive literal within the body. A program P is safe, if each rule $r \in P$ is safe.*

From now on, in this section, when describing programs, we are strictly referring to safe programs.

Definition 56. *Let P be an extended logic program we define the reduct P^S of P w.r.t.. a state S as follows:*

$$P^S := \{H \leftarrow A_1, ..., A_n | H \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m \in P, \{B_1, ..., B_m\} \cap S = \emptyset\}$$

This reduct also is known as Gelfond-Lifschitz-Reduct.

The Gelfond-Lifschitz-Reduct is informally constructed as follows:

- First we delete each rule r that has a negative literal $not\ B_i$ in its body with $B_i \in S$.
- Secondly, we eliminate all negative literals of the form $not\ B_i$ from the remaining rules.

Now that we have defined the Gelfond-Lifschitz-Reduct, we can describe the notion of stable models. Stable models, which mostly are referred to as answer sets of a program, are defined as follows [27]:

Definition 57. *An interpretation S of a program P is a stable model of P if $S = lfp(P^S)$, where P^S stands for the Gelfond-Lifschitz-Reduct of P w.r.t. S.*

Before we continue with various language extensions and later on in this chapter introduce the *guess & check* methodology, we find it necessary to present the notion of *unstratified negation*. Therefore we have to take a look at what makes a program P stratified. A program P is stratified if an evaluation ordering can be found such that we are able to evaluate each negated relation with the relation's meaning being determined beforehand. Formally stratified programs can be defined by the use of a dependency graph as presented in [3]:

Definition 58. *Let P be a program, then we define the dependency graph $G = \langle V, E \rangle$ as follows:*

- *The vertices V of the graph represent the predicates p of P.*
- *Two predicates p and q share an edge $(p, q) \in E$ iff p is in the head of a rule $R \in P$ and q is in the body of R.*
- *An edge $(p, q) \in E$ may be positive or negative. An edge (p, q) is positive iff there is a rule $R \in P$ for which p is in the head of a rule R and q occurs positively in the body of R. (p, q) is negative iff there is a rule $R \in P$ for which p is in the head of R and q occurs negatively in the body of R.*

Now that the dependency graph has been defined, we present the following definition:

Definition 59. *A program P is stratified iff in its dependency graph there are no cycles containing a negative edge. [3]*

Therefore we have an *unstratified negation* if for the negated relation in a rule no evaluation ordering exists such that the relations truth value can be predetermined beforehand. We present you with an example for a program with unstratified negation and one example for a program with stratified negation:

Example 6. *Let P be a program with the following two rules:*

$$p(X) \leftarrow q(X).$$

$$q(X) \leftarrow not\ p(X).$$

Here the dependency graph would include following set of edges:$\{(p, q), (q, p)\}$. As can be seen the graph would not contain a cycle with a negative edge, since for (q, p). p occurs negatively in the body. Hence P is regarded as unstratified.

Example 7. *We consider another program P with the rules:*

$$p(X) \leftarrow q(X).$$

$$q(X) \leftarrow not\ o(X).$$

$$o(X).$$

The dependency graph of P contains the following edges: $\{(p, q), (q, o)\}$. It clearly does not contain any cycle. And since the dependency graph does not contain any negative cycle, dependency graph P is stratified.

Language Extensions

Various language extensions for answer-set programming have been introduced, in order to increase compressiveness and to explore possible applications. They serve different purposes, such as eliminating unwanted models from the answer sets or performing optimization tasks. Some of these language extensions could result in an elevated level of complexity. The extensions which we describe are:

- Integrity constraints
- Choice rules
- Disjunctive rules
- Weight constraints
- Optimization statements

One important extension are *integrity constraints*. Their purpose is to eliminate models that do not meet the specified constraints. This elimination is based on the use of auxiliary predicates as shown below and described in detail in [27, 39].

$$p \leftarrow A_1, ..., A_n, not\ p, not\ B_1, ..., not\ B_m.$$

In this formula, by using the auxiliary predicate p, we create a situation in which each model that does satisfy $A_1, ..., A_n, not\ B_1, ..., not\ B_m$. is rendered unstable.

Example 8. *The program P_i: $p \leftarrow not\ p$. As explicitly explained in [27], constraints work as follows: We consider two different kinds of interpretations M_1 and M_2 for P_i. The first type M_1 is an interpretation, where $p \notin M$, thus the body of $p \leftarrow not\ p$ is satisfied and p should be true in M this however is contradictory to p not being in M, which was the condition for the rules body to be true. The second type M_2 is an interpretation where $p \in M$, therefore the body of the rule is false and P_i is satisfied by M_2. But remember the definition of stable models: An interpretation S of P is a stable model of P if $S = lfp(P^S)$. $lfp(P^{M_2}) = \emptyset$ which is different to M_2, hence M_2 does not constitute a stable model of P_i.*

For convenience reasons, the auxiliary predicate is omitted and constraints are rules of the following form:

$$\leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m.$$

Below, we show how integrity constraints can be used to check the validity of a given 3-coloring of a graph $G = \langle V, E \rangle$, where V are the vertices and $E \subseteq V \times V$ are the edges of the graph. A valid 3-coloring of a graph G is an assignment of colors, three in number, to vertices of G, such that no two adjacent vertices share the same color.

Example 9. *Let $G = \langle V, E \rangle$ be our graph and $L : V \rightarrow C$ be our color assignment, where C is a set of three colors $C = \{red, green, blue\}$. Now, when translating the problem of finding a valid 3-coloring of G, our color assignment L is represented by the predicates: $red(X)$, $green(X)$ and $blue(X)$, where X is a variable. Vertices are represented by the unary predicate $v/1$ and the edges by a binary predicate $e/2$. We expect a color assignment, where each vertex has a color assigned. In order to check for a valid 3-coloring, we first check whether each node has at least one color from the set C assigned:*

$$\leftarrow v(X), not\ red(X), not\ green(X), not\ blue(X).$$

And now make use of following integrity constraints to check that no neighboring vertices share the same color:

$$\leftarrow red(X), red(Y), v(X), v(Y), e(X, Y), X! = Y.$$

$$\leftarrow green(X), green(Y), v(X), v(Y), e(X, Y), X! = Y.$$

$$\leftarrow blue(X), blue(Y), v(X), v(Y), e(X, Y), X! = Y.$$

If an answer-set remains, we have got a valid 3-coloring.

Another extension are *Choice rules* [52]. Choice rules are used to encode choices over subsets of head literals. These rules are of the following form:

$$\{A_1, ..., A_m\} \leftarrow A_{m+1}, ..., A_n, not\ A_{n+1}, ..., not\ A_l$$

where $0 \leq m \leq n \leq l$ and A_i for $i = 0 \ldots l$ are literals. The semantical meaning of a choice rule is given the body literals are satisfied that each possible subset of the head can be included in the stable model of a program. To illustrate their function, we present an example:

Example 10. *We choose a very simple rule with an empty body:*

$$\{a, b\} \leftarrow$$

This rule results in the answer-sets $\{\}$, $\{a\}$, $\{b\}$ and $\{a, b\}$.

A helpful extension are disjunctive logic programs, where the rule heads are allowed to contain disjunctive information. They usually are used to represent indefinite knowledge and are of the following form:

$$A_1 \vee \ldots \vee A_m \leftarrow A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_l$$

where $0 \leq m \leq n \leq l$ and A_i for $i = 0 \ldots l$ are literals. We illustrate the use of disjunctive rules, with the following example:

Example 11. *The following rule encodes the possibility that a switch s is either on or off which is represented by the predicates* `on/1` *and* `off/1`*:*

$$on(s) \vee off(s) \leftarrow switch(s).$$

An answer set X of a disjunctive logic program P is a $\subseteq$-minimal set of atoms being closed under P^X [2]. The rule $r : A_1 \vee \ldots \vee A_m \leftarrow A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_l$ results in a different answer set for each literal $A_1, \ldots, A_m$, if $pos(r) \in S$ and $neg(r) \notin S$ for a state S of a disjunctive logic program. So semantically for each answer set we conclude that one of the alternatives from the rule head is true. The complexity of the underlying decision problem is raised by the usage of disjunctive rules [26]. For a detailed overview over the different complexity results, we refer to Table 2.2 of the complexity subsection.

Furthermore, weight constraint rules [49] provide another extension of normal logic programs. Those rules come with a lower and an upper bound, both of which can be omitted. Lower and upper bounds may theoretically be real numbers, but so far we encountered only integer implementations in the common answer-set programming solvers. We will provide the syntax of those rules, which is as follows:

$$A_0 \leftarrow l\{a_1 = w_{A_1}, \ldots, A_n = w_{A_n}, B_1 = w_{B_1}, \ldots, B_m = w_{B_m}\}$$
$$A_0 \leftarrow l\{a_1 = w_{A_1}, \ldots, A_n = w_{A_n}, B_1 = w_{B_1}, \ldots, B_m = w_{B_m}\}u$$

where A_i for $i = 0 \ldots n, n \geq 0$ are positive literals and B_j for $j = 1 \ldots m, m \geq 0$ are literals with default negation. l and u represent the lower and upper bound. $w_1, \ldots, w_n$ can be positive as well as negative integers or real numbers respectively. Albeit, if negative weights are used, the complexity of the program is increased by one level in the polynomial hierarchy. Semantically, a weight constraint rule is satisfied, if the sum of all weights contained in this rule, lies within the given bounds.

$$l \geq \sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i} \leq u$$

A special form of weight constraint rules are *cardinality rules* [49]. Those rules are used for controlling the cardinality of subsets and can be applied by only providing a lower bound, or by providing both, lower and upper bound as shown below:

$$A_0 \leftarrow l\{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}$$

$$A_0 \leftarrow l\{A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n\}u$$

where $0 \leq m \leq n$ and A_i for $i = 0 \ldots n$ are literals. l and u are integer and represent the lower and upper bound. Intuitively a cardinality rule is a weight constraint rule with all its weights set to 1, as depicted below:

$$A_0 \leftarrow l\{A_1 = 1, \ldots, A_m = 1, not\ A_{m+1} = 1, \ldots, not\ A_n = 1\}$$

$$A_0 \leftarrow l\{A_1 = 1, \ldots, A_m = 1, not\ A_{m+1} = 1, \ldots, not\ A_n = 1\}u$$

Sometimes we want to find the optimum among all possible solutions. Therefore answer-set programming comes with another powerful extension, optimization statements [49]. Here we optimize w.r.t. the weights of a subset of literals and have the choice between minimizing and maximizing. Although, as is well known, a minimization problem can easily be transformed into a maximization problem. We are for convenience reasons provided with both possibilities:

$$\text{minimize}\{a_1 = w_{A_1}, \ldots, A_n = w_{A_n}, B_1 = w_{B_1}, \ldots, B_m = w_{B_m}\}$$

$$\text{maximize}\{a_1 = w_{A_1}, \ldots, A_n = w_{A_n}, B_1 = w_{B_1}, \ldots, B_m = w_{B_m}\}$$

where A_i for $i = 0 \ldots n, n \geq 0$ are positive literals and B_j for $j = 1 \ldots m, m \geq 0$ are literals with default negation. These rules allow us to derive the stable models S with the smallest or largest weight with respect to some statement M, which is either subject to maximization or minimization:

$$w(M, S) = \sum_{a_i \in S} w_{a_i} + \sum_{b_i \notin S} w_{b_i}$$

Another language extension, which can be combined with other rules such as cardinality rules or optimization statements and disjunctive rule heads are conditional literals.

$$\ell : d$$

where ℓ is a literal and the conditional d is constituted by a domain predicate. Such a conditional literal is evaluated by conducting a substitution on $\ell : d$ in a program P where for the resulting $\ell' : d'$, d' is in the stable model of P. [49]

Aggregates are the last extension we would like to mention. Aggregates are used to express or to determine properties of a set of elements and are similar to aggregates in SQL databases [14]. Both Answer-Set Programming solvers that we describe in this chapter provide this particular language extension, although with slightly different syntax and usage. For this reason we omit a detailed description here and refer to the respective subsections on DLV and CLASP.

Architecture of Answer-Set Programming Solvers

Figure 2.10 depicts the whole process of solving a problem with the help of answer-set programming. The problem itself, first of all, has to be translated into a logic program. Several

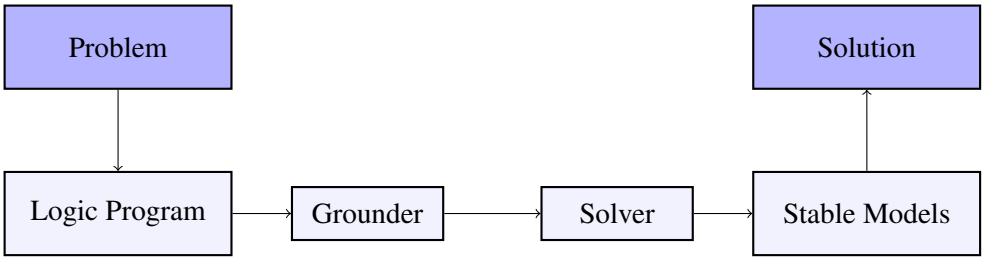

Figure 2.10: The ASP process

answer-set programming solvers offer slightly different syntax and may provide several language extensions, but basically they all are fed with extended logic programs. The process of determining the answer sets of those logic programs can basically be split into two separate stages. In the first stage, the logic program undergoes the grounding process, where a program P which may contain variables, is replaced by its ground instance $grnd(P)$. In the second stage a solver conducts the model search, which may result in none, a single or multiple stables model, the answer sets of an ASP program. These resulting answer sets may then be interpreted, depending on the application context. Some of the solvers combine those two stages in one single executable, others consist of two executables where one does the grounding and the other conducts the propositional model search.
Basically the grounder simply replaces the program P by its ground instance $grnd(P)$, but for efficiency reasons intelligent grounding techniques have been developed, which result in much smaller grounded instances. The underlying idea is that many of the resulting literals such as repeated literals in the body of rules, or tautological rules can be omitted without having any effect on the resulting answer sets. Advances in database technology also had an effect on the development of efficient grounders, since many problems in grounding are shared with database applications [14].
After the grounding process, the propositional program is processed by a solver. These solvers are based on sophisticated SAT-solvers which make use of techniques such as backtracking, clause-learning and heuristics [14].

Answer-Set Programming Modeling Techniques

Now that we have presented various language extensions of answer-set programming, we are ready to introduce a collection of four modeling techniques, which partially may involve the use of these extensions. The modeling techniques, which we present are:

- guess & check
- saturation
- iteration over sets

The most common modeling technique is the *Guess and Check* methodology, which is also referred to as *Generate-and-Test* and is elaborately explained by V. Lifschitz in [38]. In this

technique we employ two steps. In the first step, a set of candidate solutions is generated, which is then filtered in the second step by constraints as follows:

1. Nondeterminism, obtained by the use of unstratified negation or choice rules, is used to generate a set of candidate solutions.

2. The "Check" or "Test" in this methodology primarily consists of constraint rules. Those constraint rules may include plain integrity constraints, cardinality constraints, weight constraints or optimization statements. This part of the process might include auxiliary predicates as well.

Again, as an example, we refer to the problem of 3-colorability of a graph $G = \langle V, E \rangle$, where V are the vertices and $E \subseteq V \times V$ are the edges of the graph again and we look for a valid 3-coloring of a graph G which is an assignment of colors such that no pair of adjacent vertices is assigned the same color.

Example 12. *Let $G = \langle V, E \rangle$ be a graph and our color assignment $L : V \rightarrow C$, where C is a set of three colors $C = \{red, green, blue\}$. Now, when translating the problem of finding a valid 3-coloring of G, our colors are represented by the predicates: $red(X)$, $green(X)$ and $blue(X)$, where X is a variable. The nodes are represented by $node(X)$ and edges between two nodes X and Y are encoded as follows: $edge(X, Y)$. Now that we have our graph representation, we are ready to encode our "Guess" and "Check" program:*
The "Guess" consists of a disjunctive/choice rule:

$$red(X), green(X), blue(X) \leftarrow node(X).\} \textit{ Guess}$$

And the "Check" consists of the following three rules:

$$\left.\begin{array}{l} \leftarrow red(X), red(Y), edge(X, Y). \\ \leftarrow green(X), green(Y), edge(X, Y). \\ \leftarrow blue(X), blue(Y), edge(X, Y). \end{array}\right\} \textit{ Check}$$

Example 13. *Here we encode the same example again, but instead of choice rules, we make use of unstratified and double negation. Again we let $G = \langle V, E \rangle$ be a graph and our color assignment $L : V \rightarrow C$, where C is a set of three colors $C = \{red, green, blue\}$. Now, when translating the problem of finding a valid 3-coloring of G, our colors are represented by the predicates:$red(X)$, $green(X)$ and $blue(X)$, where X is a variable. The nodes are represented by $node(X)$ and edges between two nodes X and Y are encoded as follows: $edge(X, Y)$. Now that we have our graph representation, we are ready to encode our "Guess" and "Check" program:*

$$
\left.
\begin{aligned}
red(X) &\leftarrow not\neg red(X), node(X).\\
\neg red(X) &\leftarrow not\, red(X), node(X).\\
green(X) &\leftarrow not\neg green(X), node(X).\\
\neg green(X) &\leftarrow not\, green(X), node(X).\\
blue(X) &\leftarrow not\neg blue(X), node(X).\\
\neg blue(X) &\leftarrow not\, blue(X), node(X).
\end{aligned}
\right\} \textit{Guess}
$$

$$
\left.
\begin{aligned}
&\leftarrow red(X), red(Y), edge(X,Y).\\
&\leftarrow green(X), green(Y), edge(X,Y).\\
&\leftarrow blue(X), blue(Y), edge(X,Y).
\end{aligned}
\right\} \textit{Check}
$$

The next technique that we present is the *saturation technique*. It is used to test whether a property holds for all possible guesses in an ASP program. This means for instance that one can check whether or not a solution to a problem exists and if so provide a single answer set for that case. When considering 3-colorability as an example, one can check whether or not a graph is 3-colorable by the use of saturation.
To check whether a property holds for all guesses of a program P, we have to define an answer set candidate which constitutes a witness for its satisfaction. We call it M_{sat}, which is the single answer set if the wanted property is present in all guesses. Otherwise, if the desired property does not hold for every guess, there will be an answer set M which is not fully saturated and $M_{\neg sat} \subset M_{sat}$. Due to the definition of stable models this yields the result that M_{sat} is not a stable model anymore and is therefore not included in the answer sets of P. [27]

Also in [27] a general design rule for the saturation process in answer-set programming is presented for performing a check whether a given property Pr holds:

1. Define the search space of all guesses, by either choice rules or the use of unstratified negation. We call that search space subprogram P_{guess}.

2. Use a subprogram P_{check} in order to verify if property Pr holds for an arbitrary guess M_G.

3. If Pr holds, we perform the saturation on M_G to generate M_{sat}.

4. If the property Pr does not hold for M_G, then program P does yield a strict subset $M_{\neg sat} \subset M_{sat}$.

For this technique we choose the example of checking whether a given graph is bipartite. A graph $G = \langle V, E\rangle$ is bipartite if it can be divided into two disjoint sets $U_1 \subseteq V$ and $U_2 = V \backslash U_1$ such that every edge of E connects a vertex $x \in U_1$ to a vertex $y \in U_2$. This problem is the same as checking whether a graph is two-colorable and can be encoded as follows:

Example 14. *Let $G = \langle V, E\rangle$ be a graph. Then V are the vertices and $E \subseteq V \times V$ are the edges of the graph. The vertices are represented by facts $v/1$ and the edges are represented by*

a binary predicate $e/2$. Now the two disjunctive subsets of V, $U_1 \subseteq V$ and $U_2 = V \backslash U_1$ are represented by the predicates $u1/1$ and $u2/1$.

$$u1(X) \vee u2(X) \leftarrow v(x).\} P_{guess}$$

$$\left.\begin{array}{l} not\_bip \leftarrow u1(X), u1(Y), edge(X,Y). \\ not\_bip \leftarrow u2(X), u2(Y), edge(X,Y). \end{array}\right\} P_{check}$$

$$\left.\begin{array}{l} u1(X) \leftarrow not\_bip, v(X). \\ u2(X) \leftarrow not\_bip, v(X). \end{array}\right\} P_{saturate}$$

In the first subprogram P_{guess} we use a choice rule to guess every possible subset-assignment for the vertices of G. In the subsequent subprogram P_{check} we now check whether the assignment of subsets to vertices is legal according to the definition of a bipartite graph. If the assignment at some point does constitute a witness for a bipartite graph, not_bip does not hold at that point and we do not saturate this answer set. In the next step $P_{saturate}$ and P together with the graph does not yield the saturated answer set M_{sat}. Otherwise, for each step we saturate every possible answer set and the single answer set M_{sat} is the result which indicates that the graph is not bipartite.

The last technique we present is *iteration over a set*, which may serve several purposes such as testing whether a property holds for all answer-sets without the use of negation. There are two primary reasons for using such a method. The first one is to avoid the use of negation where it could lead to undesired behavior such as cyclic negation. The second one is the use of iteration over a set in combination with the saturation technique. [27].
Another application of this technique is to recursively generate answer-sets. Here a successor relation is used to label the answer-set of a program in order to iteratively apply the program again on its result in order to compute a new answer-set.
The effect of iteration is reached by the use of an ordering, which is imposed on some part of the domain of the problem encoding, as we can see in the example below. Then the smallest element of a domain is determined to be a starting point for the iteration. With the help of the successor relationship from the ordering, the successor variable of the variable labeling the current iteration step is determined.

Example 15. *Here an order is put on the vertices of a graph $G = \langle V, E \rangle$, where the vertices V are represented by the predicate* `node/1`.

$$lt(X,Y) \leftarrow node(X), node(Y), X < Y.$$
$$nsucc(X,Z) \leftarrow lt(X,Y), lt(Y,Z).$$
$$succ(X,Y) \leftarrow lt(X,Y), not\ nsucc(X,Y).$$
$$ninf(X) \leftarrow lt(Y,X).$$
$$nsup(X) \leftarrow lt(X,Y).$$
$$inf(X) \leftarrow not\ ninf(X), node(X).$$
$$sup(X) \leftarrow not\ nsup(X), node(X).$$

The predicate `succ/2` *denotes the successor relationship, where the predicates* `inf/1` *and* `sup/1` *represent the smallest, respectively the largest domain element.*

Complexity of ASP

In this subsection we include a brief look at complexity results within the field of Answer-Set Programming as provided by [22]. Since we too are dealing with fixed programs, there is no need to for adaptations and therefore we can focus on data complexity. More specifically we are dealing with the data complexity of checking whether $P(D) \models A$ with P being a logic program, D being the input database and A a set of ground atoms. We distinguish between two types of decision problems. Here $\models_c$ stands for credulous reasoning and $\models_s$ stands for skeptical reasoning.

e	normal programs	disjunctive programs	optimization programs
$\models_c$	NP	Σ_2^P	Σ_2^P
$\models_s$	coNP	Π_2^P	Π_2^P

Table 2.2: Complexity for logic programs. [22]

Answer-Set Programming Systems

In this subsection we present the Answer-Set Programming systems which we use for computing our encodings. The first such system is an implementation of *Disjunctive Logic Programming* which was developed by the *Technische Universität Wien* and the *University of Calabria*, called DLV. The second system, which we use is CLASP in combination with GRINGO. Both are part of the "Potsdam Answer Set Collection" [30, 33] and have been developed, as the name implies, at the University of Potsdam.

DLV

DLV [11] is an implementation of *Disjunctive Logic Programming* which was developed by the *Technische Universität Wien* and the *University of Calabria*.
Disjunctive Logic Programs are programs where disjunction is allowed in the head of the rules and negation may occur in the body of the rules. The semantics, which is implemented in DLV is the answer set semantics as presented by V. Lifschitz [39]. As a result a disjunctive logic program may have several models called answer sets. Furthermore, *DLV* is extended with *weak constraints* [37] .
The syntax of *DLV* is constructed as follows:
Variables are denoted as strings, which start with an uppercase letter. Strings which start with a lowercase letter denote constants. Positive integer constants and strings are surrounded with double quotes. Terms are either a variable or a constant. Atoms are predicates p of the form $p(t_1, ..., t_n)$. A literal l is either an atom p or a negated atom $\neg p$. Negation as failure is denoted by literals l with a preceding not: $not\ l$. The complementary of a classic literal l, $\neg l$ is defined

as $\neg p$ if $l = p$ and as p otherwise.

The disjunction $a_1 \vee ... \vee a_n$ is called the *head* of the rule and $b_1, ..., b_k,$ not $b_{k+1}, ...,$ not b_m denote the body. A rule without *head* is called an *integrity constraint* and a rule, which has precisely one head literal is called *normal*. A rule with empty body is called *fact* (":-" can be omitted). A special type of integrity constraint are weak constraints. As opposed to standard constraints, these constraints do not necessarily need to be satisfied. Hence their violation does not remove an answer-set from the solution. Yet the solver aims to minimize the weight of violated weak constraints. Furthermore it is possible to prioritize weak constraints. When applying prioritization, the violations of the weak constraints with the highest priority are minimized first. This minimization continues with the lower priority levels in descending order.

Definition. *Weak constraints contain the symbol ":$\sim$" instead of ":$-$". They are an expression of following form:*
$:\sim b_1, ..., b_k,$ *not* $b_{k+1}, ...,$ *not* $b_m.[w : l]$
where w stands for weight and l for the layer. w and l are both positive integer constants.

Before we present the various aggregates supported by DLV, we have to define *symbolic sets* which aggregates work with. A symbolic set is defined in DLV as follows [11]:

Definition 60. *Let $vars$ be a set of local variables and $conj$ is a set of non-aggregate literals, then*

$$\{vars : conj\}$$

syntactically denotes a symbolic set. A variable is considered local if it occurs in at least one literal of $conj$, and does not appear outside the symbolic set. A global variable, is a variable that occurs outside the symbolic set. Literals of $conj$ may contain constants, local variables, and global variables.

DLV offers aggregate predicates which allow one to express properties over a set of elements. They can be contained in the body of rules and constraints. Aggregates may also be negated by the *default negation*. The aggregates which *DLV* supports are: $\#count$, $\#min$, $\#max$, $\#sum$ and $\#times$. Their meaning and function is described in Table 2.3.

An example of such a predicate use might be:

Example 16. *Suppose we want to count the number of directed edges, connected to a vertex of a graph. A vertex is represented by the predicate v/1 and an edge is represented by the predicate e/2. Then the following rule determines the number N of edges (outgoing and incoming) for each vertex X.*

```
1 edges(X,N) :- N = N1 + N2, N1 = #count{Z : e(X,Z)}, N2 = #count{Z : e(Z,X)},
    v(X).
```

Furthermore DLV supports a variety of comparative predicates (see Table 2.4) to enable the comparison of constants. Those constants encompass integers as well as any other type of symbols, but note that only for the integer comparison the semantics is determined. For all other

aggregate	function of the aggregate
#count	cardinality of the local set
#min	computes the minimum value of the first local variable to be aggregated over in the symbolic set
#max	computes the maximum value of the first local variable to be aggregated over in the symbolic set
#sum	returns the sum of the first local variable to be aggregated over in the symbolic set
#times	computes the product of the first local variable to be aggregated over in the symbolic set

Table 2.3: built in aggregates

type of constants only a fixed ordering can be guaranteed.

predicate	meaning
== or =	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Table 2.4: built in comparative predicates [11]

Another set of supported predicates are arithmetic predicates as depicted in Table 2.5. In order to make use of this extension provided by DLV, an upper limit N for integers has to be provided in the command line. DLV only does work with positive integers and if one provides integer constants outside this range in the input, DLV responds by issuing a warning. The arguments of arithmetic predicates always consist of zero or more input arguments (which have to fulfill the safety criterion) and exactly one output argument. Note that arithmetic predicates in DLV will never return a negative output argument. [11]

The last functionality of DLV we would like to mention is *guards*. Guards are either variables or numeric values. They are a means to compare values which aggregate predicates return. We distinguish between two types of guards, ordinary guards and *assignment guards*, which are used as variables to store the return value of an aggregate. Aggregates with assignment guards always evaluate to true.

CLASP

As an alternative answer-set programming solver technology, we chose CLASP and GRINGO. Both are part of the ”Potsdam Answer Set Collection” [30, 33] and have been developed, as the name implies, at the University of Potsdam. All the programs of the ”Potsdam Answer Set

predicate	meaning
#int(X)	is true, iff X is a known integer.
#succ(X, Y)	is true, iff $X + 1 = Y$ holds.
#prec(X, Y)	is true, iff $X - 1 = Y$ holds.
#mod(X, Y, Z)	is true, iff $(X mod Y) = Z$ holds.
#absdiff(X, Y, Z)	is true, iff $\|X - Y\| = Z$ holds.
+(X,Y,Z), or alternatively: Z=X+Y	is true, iff $Z = X + Y$ holds.
*(X,Y,Z), or alternatively: Z=X*Y	is true, iff $Z = X \times Y$ holds.
-(X,Y,Z), or alternatively: Z=X-Y	is true, iff $Z = X - Y$ holds.
/(X,Y,Z), or alternatively: Z=X/Y	is true, iff $Z = X \div Y$ holds.

Table 2.5: built in arithmetic predicates [11]

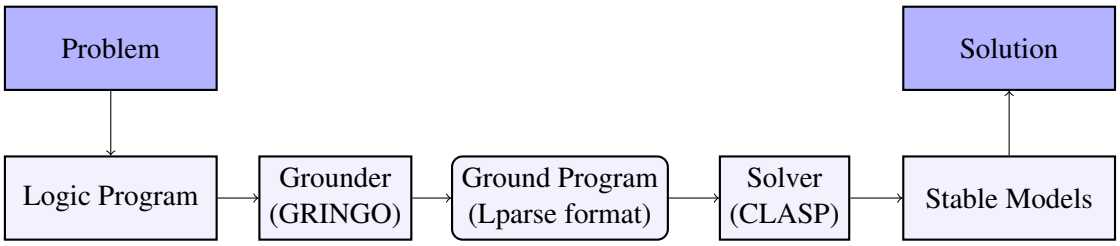

Figure 2.11: The ASP solving process with CLASP and GRINGO

Collection" are written in C++ and published under the GNU General Public License [31]. CLASP is a conflict-driven answer-set solver which combines the high-level modeling capacities of Answer-Set Programming with state-of-the-art techniques from the area of Boolean constraint-solving. [34]. GRINGO is a grounder for logic programs under answer sets semantics and has originally been developed as a combination and extension of the grounding approaches of lparse and DLV [35]. GRINGO's primary purpose is to preprocess given logic programs and perform variable substitution. The task performed by GRINGO can be split into four different phases. In the first phase, the input program is checked syntactically for correctness and an internal representation of it is generated. In the second phase, GRINGO verifies that a finite equivalent ground instantiation exists for the input. Then the ground instances are computed according to a predetermined schedule in order to evaluate in the 4th and last phase, the newly derived ground instances [35]. The output of gringo can then be passed to CLASP in an lparse [51] format. CLASP then applies techniques, such as conflict driven clause learning [40] and various heuristics. The application of CLASP on the grounded output of gringo then results in the desired answer sets, as depicted in Figure 2.11.

The input language of GRINGO consists of rules, facts, integrity constraints and a variety of meta-statements. Rules, facts and integrity constraints are constructed from terms, the most basic ones of which are integers, constants and variables. A special type of variables are *anonymous variables*, which are denoted as '_'. Anonymous variables are similar to standard variables, with

one difference: each occurrence of '_' is treated as a newly introduced variable. Furthermore, special constants exist such as '#infinum' and '#supremum' which stand for the smallest and the largest possible value. A rule is represented by a construct of the following form:

$$A_0 \text{: } -L_1, \ldots, L_n$$

where A_0 is an atom that constitutes the head of the rule. The body is composed of a conjunction of literals, separated by a comma. Any literal is denoted as L_i for $1 \leq i \leq n$ and consists of an atom A or a negated atom $notA$, where not stands for the default negation. It remains to note, that GRINGO expects the rules of a program to be safe. Facts are rules with an empty body and of the following form:

$$A_0.$$

Again A_0 is an atom. Integrity constraints are rules with an empty head and are represented in the following way:

$$: -L_1, \ldots, L_n$$

A term which does not contain any variable is denoted as a *ground term*. Subsequently, we will further describe the various parts, which extend the input language of GRINGO.

Classical negation in front of atoms and denoted as $-$ (not to confuse with default negation) is permitted in front of atoms $-A$. In contrast to default negation, where the negation of an atom holds as long as the atoms truth itself cannot be established, classical negation of an atom only holds if the complementary of the atom can be established. Classical negation constitutes merely a extra syntactic feature, which can be constructed by the use of integrity constraints [31].

CLASPD an extended version of CLASP does come with support for disjunctive rules. As already explained they are rules with *disjunctions* in their heads. In the rule heads, atoms of a disjunction are separated by a pipe symbol '|'. Those disjunctions are usually used to represent indefinite knowledge and are one possible way of performing the guess & check technique. As already mentioned in our section on complexity of answer-set programming, disjunction in rule heads does increase the complexity of programs. Again we return to our example of 3-colorability and show how a guess & check procedure is encoded with the help of disjunctive rules as in CLASPD.

Example 17. *Let $G = \langle V, E \rangle$ be a graph and our color assignment $L : V \rightarrow C$, where C is a set of three colors $C = \{red, green, blue\}$. Now, when translating the problem of finding a valid 3-coloring of G, our colors are represented by the predicates:$red(X)$, $green(X)$ and $blue(X)$, where X is a variable. The nodes are represented by $node(X)$ and edges between two nodes X and Y are encoded as follows: $edge(X, Y)$. Now that we have our graph representation, we are ready to encode our "Guess" and "Check" program:*
The "Guess" consists of a disjunctive rule:

```
1 red(X) | green(X) | blue(X) :- node(X).
2
3 :- red(X), red(Y), edge(X,Y).
4 :- green(X), green(Y), edge(X,Y).
5 :- blue(X), blue(Y), edge(X,Y).
```

GRINGO comes with a great number of different arithmetic functions, which are presented in Table 2.3. Please note that it is not allowed to bind variables in the scope of an arithmetic

operation	symbols
addition	+
subtraction	-
unary minus	-
multiplication	*
division	/ or $\#div$
modulo	\ or $\#mod$
absolute	$\|\cdot\|$ or $\#abs$
power	** or $\#pow$
bitwise and	&
bitwise or	?
bitwise xor	^
bitwise complement	~

Table 2.6: built-in arithmetic functions

function by a corresponding atom. [31]

Example 18. *We present some small examples of how these arithmetic predicates can be used:*

```
1  value1(4). value2(8).
2  addition(V1 + V2)    :- value1(V1), value2(V2).
3  subtraction(V2 - V1)    :- value1(V1), value2(V2).
4  multiplication(V1 * V2) :- value1(V1), value2(V2).
5  division(V1 / V2) :- value1(V1), value2(V2).
6  modulo(#mod(V1,V2)) :- value1(V1), value2(V2).
7  modulo(V1 #mod V2) :- value1(V1), value2(V2).
8  modulo(V1 \ V2) :- value1(V1), value2(V2).
9  absolute(#abs(V)) :- value1(V).
10 power(V1 ** V2)    :- value1(V1), value2(V2).
```

As the reader can see various notations for the same operation are valid.

Furthermore the following comparison predicates are supported:

predicate	meaning
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Table 2.7: built-in comparison predicates

Comparison predicates can be evaluated with integers as well as arbitrary ground terms, where constants are ordered lexicographically and the function symbols are ordered according to their arity first and then lexicographically. If the name is the same, they are ordered component wise. Integers are always smaller than constants and constants are smaller than function symbols. Note: Arithmetic functions are always evaluated before the application of comparison predicates [31].

Furthermore, built-in predicates "=" and " :=", used as assignment operators, are provided. These predicates unify a term on the right side of the predicate to a non-ground term on the left-hand side [31].

GRINGO offers some convenience when defining a great number of facts through the use of intervals of integers. We provide an example:

Example 19. *Let $fact/1$ be an arbitrary unary predicate, then we define a list of facts over a range of integers from i to j as follows:*

$$fact(i..j).$$

Another feature that is supported by GRINGO, are *conditions*, which allow for instantiating variables to collections of terms within a single rule. This feature is quite convenient in order to define more compact representations of aggregates or for defining conjunctions or disjunctions over many arbitrarily ground atoms. Note three important limitations for a correct application of conditions [31]:

- All predicates on the right-hand side of a condition must either be domain predicates or built-in.
- Each variable within an atom in front of a condition must occur on the right-hand side or be global.
- Global variables are prioritized over local ones. One has to choose local variables with care in order to avoid accidental conflicts.

Example 20. *We present an example where conditions are used for compact representations of aggregates. Let $G = \langle V, E \rangle$. The nodes are represented by $node(X)$ and edges between two nodes X and Y are encoded as follows: $edge(X, Y)$.*

```
1 edgecount(X,V1 + V2) :- V1 = #count {edge(X,_) : node(X) }, V2 = #count {edge
    (X,_) : node(X) }.
```

Here we count the number of edges for a given node.

Pooling, which is represented by the symbol ";" is a method to achieve a more compact representation for atoms where we encode several options. Pooled arguments in a term of the rule body are expanded to a conjunction of the different options, where pooled arguments of a term in the head are expanded to multiple rules with the various options in the head [31].

An extensive collection of aggregates is provided, which in general are of the following form:

$$l \text{ op } [L_1 = w_1, \ldots L_n = w_n]u$$

where l and u are the lower and the upper bound. L_i, for $1 \leq i \leq n$ are a multi-set of literals, for which w_i denotes the weight. An aggregate evaluates to true, if the operation op applied to the multi-set lies within the specified bounds. The fact that multi-sets are used results in the possibility of the same literals occurring multiple times within that multi-sets. In the table below you will find the various aggregates which are supported: If aggregates, which are allowed to

aggregate	function of the aggregate
#count	count literals (allowed, without bounds, on right-hand side of assignments)
#min	minimum weight (allowed, without bounds, on right-hand side of assignments)
#max	maximum weight (allowed, without bounds, on right-hand side of assignments)
#sum	sum of the weights (allowed, without bounds, on right-hand side of assignments)
#avg	average of the weights
#even	true if the number of true literals within the multi-set is even
#odd	true if the number of true literals within the multi-set is odd

Table 2.8: built in aggregates

be used in that way are placed on the right-hand side of an assignment, it is advised to only use domain predicates within that aggregate, because otherwise a space-blow up may be the result. $\#count$ is a special aggregate of the form:

$$\text{\#count } \{L_1, \ldots L_n\}u$$

which comes only with an upper bound u and all weights set to 1 by default. Another special set of aggregates are $\#even$ and $\#odd$, which are aggregates with all weights set to 1 and no lower or upper bounds specified. Like the $\#count$ aggregate, these aggregates have curly brackets instead of square brackets. In the aggregates where curly brackets are used repeated occurrence of literals is not counted. Regarding variables in aggregates: here an atom occurring within an aggregate behaves similar to an atom of the left side of a rule. Any variable occurring within an aggregate is local and has to be bound by a variable which occurs outside the aggregate or by a variable occurring on the right-hand side of a condition [31].

Optimization statements are a language extension which is of course included as well. One kind of optimization statements is built upon multi-sets and denoted with square brackets. Here weights might be provided. The other kind is denoted with curly brackets and no extra weight assignments are possible. For both types of optimization statements, priority levels might be provided. For compatibility reasons with LPARSE, default priorities are assigned, if multiple optimization statements are used. Their priorities are then determined by the the occurrence of the statements in the program. The later a statement occurs, the higher its priority is. Syntactically optimization statements are constructed as follows:

$$\text{opt } [L_1 = w_1@p_1, \ldots L_n = w_n@p_n]$$

$$\text{opt}\ \{L_1@p_1, \ldots L_n@p_n\}$$

where again L_i, for $1 \leq i \leq n$ are a multi-set or a set of literals. w_i denotes the weight and p_i denotes the priority levels.

Example 21. *We present an example which concisely depicts a program that makes use of optimization statements. Suppose a project manager of some company wants to build a project team based on following criteria: The team should consist of at least 5 employees of which at least one is an electrical engineer. The team should have the best experience, but incur minimal cost. Experience is of more importance than minimizing the costs.*

```
1  #hide.
2  #show in_team/1.
3  employee(1..10).
4  experience(1,4). experience(2,3). experience(3,3). experience(4,3).
       experience(5,3).
5  experience(6,3). experience(7,3). experience(8,3). experience(9,3).
       experience(10,3).
6  cost(1,5). cost(2,2). cost(3,7). cost(4,6). cost(5,4).
7  cost(6,3). cost(7,4). cost(8,5). cost(9,9). cost(10,3).
8  electrical_engineer(3).
9  electrical_engineer(5).
10
11 in_team(X) :- not -in_team(X), employee(X).
12 -in_team(X) :- not in_team(X), employee(X).
13
14 electricalengineer :- in_team(X), electrical_engineer(X).
15 #maximize[in_team(X) : experience(X,Y)=Y@1 ].
16 #minimize[ in_team(X) : cost(X,Y)=Y@2 ].
17
18 :- V = #count{in_team(X)}, V < 5.
19 :- not electricalengineer.
```

Lines 1 and 2 are meta-statements which we will describe subsequently. In line 3, we generate 10 employees. In lines 4 to 10 we use three different predicates to assign to each employee an experience value, a cost factor and the qualification of being an electrical engineer. Line 11 and 12 constitute the guess of the assignment of employees to the team. Line 14 is a rule which tells us, if our team includes an electrical engineer. In line 15 and 16, we provide the optimization statements. We want to maximize the experience in the team and at the same moment minimize the cost. Line 18 and 19 are integrity constraints which assure that the team is not smaller than the size of 5 and that an electrical engineer is included.

The meta-statements which are supported include the symbols which denote a line as a comment "%" and the symbols "%*" and "*%" which denote a multi-line comment. Furthermore, the statements "#show" and "#hide" can be used to explicitly state which predicates shall be shown or hidden in the answer sets. Below we present an example we borrowed from [31], to show how these statements are used.

Example 22. *An example how the meta-statements "#show" and "#hide" are used:*

```
1 #hide.                    % Suppress all atoms in output
2 #hide p/3.                % Suppress all atoms of predicate p/3 in output
3 #hide p(X,Y) : q(X).      % Suppress p/3 if the condition holds
4 #show p/3.                % Include all atoms of predicate p/3 in output
5 #show(X,Y) : q(X).        % Include p/3 if the condition holds
```

CHAPTER 3

ASP based Argumentation

In this chapter, we provide an extensive overview over the existing approaches with respect to abstract argumentation in answer-set programming. In the first section we present an overview following the survey on abstract argumentation in answer-set programming by F. Toni and M. Sergot , provided in [53], which deals with the approach by C. Nieves et al. [42], which is based on a method relying on the use of propositional formulas, another by T. Wakaki and K. Nitta, presented in their article [54]. Furthermore, the survey summarizes the approach of U. Egly et al. [24, 25] to compute the extensions of the conflict-free, admissible, preferred, stable, complete and grounded semantics. Last-mentioned is the encoding of the ideal semantics proposed by W. Faber and S. Woltran in [28].
We continue with a section in which we present two existing approaches to computing the resolution-based grounded semantics with the help of answer-set programming. The first one was developed by S. Woltran et al. and is based on an algorithm which was originally proposed by Baroni et al. [5]. It is primarily based on the Guess & Check paradigm and on the iteration over sets. The second approach is based on the meta-modeling technique as proposed in [32] by M. Gebser et al.

3.1 Encodings of Standard Semantics

In this section we present various approaches to encoding the standard semantics of abstract argumentation, basically following the article of F. Toni and M. Sergot [53]. This article provides an extensive survey on those approaches, using answer-set programming for computing the extensions. These encodings provide a (mostly) one-to-one correspondence between the answer-sets of the different encodings and the extensions of the semantics. When describing the different approaches, F. Toni and M. Sergot differentiate between them by the kind of extension these encodings focus on and by type of mappings and the correspondences they define. They categorize them into two distinct groups. All the approaches mentioned by F. Toni and M. Sergot, rely on DLV as answer-set solver.

The first approach to be explained is the computation of the preferred extension proposed by C. Nieves et al. [42], which is based on a method using propositional formulas. With those propositional formulas conditions for sets of arguments of an abstract argumentation framework $G = \langle A, R \rangle$ are described. The answer-sets of this approach are in one-to-one correspondence with the extensions of the preferred extension. It consists of a mapping, which results in a disjunctive logic program that defines a predicate def, where each pair $(x, y) \in R$ is represented by a rule:

$$def(x) \vee def(y).$$

For each pair $(x, y) \in R$ furthermore another rule is added:

$$def(x) \leftarrow def(z_1), \ldots, def(z_k).$$

where $(k \geq 0)$ and the arguments $z_1, \ldots z_k$ are defenders of x against y, which means for each $z_i : att(z_i, y) \in R$ there does not exist another attack of y in R. Then the preferred extension is defined as the complement of each answer-set of the encoding described, meaning an argument x is in the extension if the corresponding predicate $def(x)$ is not part of the answer-set. For a detailed description please refer to [53].

In their article T. Wakaki and K. Nitta [54], present encodings for the complete, stable, preferred, grounded and semi-stable semantics. Their approach is based on the *reinstatement labelings* as introduced by M. Caminada [15], which essentially are total functions, mapping from arguments of an AF to labels $\{in, out, undec\}$. Where arguments are labelled in all its attackers are labelled out and arguments are labelled out if there exists an argument which is attacking it and is labelled in. For each encoding, an argumentation framework $AF = \langle A, R \rangle$ is translated into an input database $\hat{F}$ as follows:

$$\hat{F} = \{arg(a) \mid a \in A\} \cup \{att(a, b) \mid (a, b) \in R\}$$

Then the complete extensions, for instance, are determined by the following encoding:

$$\begin{aligned}
\Pi_{compl} = \{&in(X) \leftarrow arg(X), not\ ng(X).\\
&ng(X) \leftarrow in(Y), att(Y, X).\\
&ng(X) \leftarrow undec, att(Y, X).\\
&out(X) \leftarrow in(Y), att(Y, X).\\
&undec(X) \leftarrow arg(X), not\ in(X), not\ out(X).\}
\end{aligned}$$

Here, as well, the resulting answer-sets are in one-to-one correspondence with the extensions of the complete semantics. Each argument labelled in is an argument of the extension. As opposed to the latter approach by U. Egly et al. (see below) and where maximality checks are performed implicitly, T. Wakaki and K. Nitta perform the maximality checks separately in a form of processing. For a more in-depth overview, we again refer to [53].

In their articles U. Egly et al. [24, 25] propose encodings for the extensions of the conflict-free, admissible, preferred, stable, complete and grounded semantics. The first step transforms the input argumentation framework $AF = \langle A, R \rangle$ into answer-set programming rules:

$$\hat{F} = \{arg(a) \mid a \in A\} \cup \{att(a, b) \mid (a, b) \in R\}$$

For computing the extensions of the different semantics, they follow a modular approach. By combining the input AF $\hat{F}$ with one or more sets of rules each of which responsible for a certain semantics.

We begin by describing the encoding for computing the conflict-free extensions for a given AF which is constructed as follows:

$$\begin{aligned}\Pi_{cf} = \{&in(X) \leftarrow not\ out(X), arg(X).\\ &out(X) \leftarrow not\ in(X), arg(X).\\ &\leftarrow in(X), in(Y), defeat(X, Y).\}\end{aligned}$$

Π_{cf} starts with a guess, and then discards all models which are not conflict-free. The resulting extensions are obtained by computing the answer sets, for a logic program $\hat{F} \cup \Pi_{cf}$, which are in a one-to-one correspondence with the respective extensions. For the remaining semantics admissible, preferred, stable, complete and grounded, the technique works in the same fashion.

We continue with the description of the encoding for the admissible semantics. This encoding relies on the computation of the conflict-free extensions and adds a rule-set for discarding all answer-set where arguments within an extension are not defended by the extension itself.

$$\begin{aligned}\Pi_{adm} = \Pi_{cf} \cup \{&defeated(X) \leftarrow in(Y), defeat(Y, X).\\ &not\_defended(X) \leftarrow defeat(Y, X), not\ defeated(Y).\\ &\leftarrow in(X), not\_defended(X).\}\end{aligned}$$

The encoding for the complete extension Π_{comp} builds on the encoding of the admissible extension Π_{adm} and adds a constraint which discards all answer-set where arguments defended by the extension are not contained in that extension.

$$\Pi_{comp} = \Pi_{adm} \cup \{\leftarrow out(X), not\ not\_defended(X).\}$$

The two answer-set programming modeling techniques saturation and iteration over sets, require an order $<$ over the domain elements, in order to be able to construct some form of loops. This order, together with some helper predicates `succ/2`, `inf/1`, `sup/1`, `lt/1`, is provided by the following module $\Pi_<$ which is also used by all the encodings which we subsequently describe.

$$\begin{aligned}\Pi_{<} = \{&lt(X,Y) \leftarrow arg(X), arg(Y), X < Y.\\ &nsucc(X,Z) \leftarrow lt(X,Y), lt(Y,Z).\\ &succ(X,Y) \leftarrow lt(X,Y), not\ nsucc(X,Y).\\ &ninf(X) \leftarrow lt(Y,X).\\ &nsup(X) \leftarrow lt(X,Y).\\ &inf(X) \leftarrow not\ ninf(X), arg(X).\\ &sup(X) \leftarrow not\ nsup(X), arg(X).\}\end{aligned}$$

Before we are able to present the encoding of the grounded semantics, we have to present $\Pi_{defended}$ which includes the needed predicate `defended/1`. The predicate `defended/1` itself is based on another predicate that is included in $\Pi_{defended}$, `defended_upto/2`. For instance $defended\_upto(X,Y)$ encodes the fact that an argument X is defended by a current assignment with respect to all arguments $U \leq Y$ [24].

$$\begin{aligned}\Pi_{defended} = \{&defended\_upto(X,Y) \leftarrow inf(Y), arg(X), not\ defeat(Y,X).\\ &defended\_upto(X,Y) \leftarrow inf(Y), in(Z), defeat(Z,Y),\\ &\qquad\qquad defeat(Y,X).\\ &defended\_upto(X,Y) \leftarrow succ(Z,Y), defended\_upto(X,Z),\\ &\qquad\qquad not\ defeat(Y,X).\\ &defended\_upto(X,Y) \leftarrow succ(Z,Y), in(V), defeat(V,Y),\\ &\qquad\qquad defeat(Y,X).\\ &defended(X) \leftarrow sup(Y), defended\_upto(X,Y).\}\end{aligned}$$

Having defined `defended/1`, the encoding of the grounded semantics [24] can be presented:

$$\Pi_{ground} = \Pi_{<} \cup \Pi_{defended} \cup \{in(X) \leftarrow defended(X).\}$$

The last encoding, proposed by U. Egly et al., which we describe carries out the task of computing the preferred extensions. For encoding the preferred semantics, U. Egly et al. make use of the saturation technique. Since the encoding of the preferred extension is not as trivial as the computation of the previous semantics, we recall the definition of the preferred semantics: Let $AF = \langle A, R\rangle$ be an argumentation framework, then a set $S \subseteq A$ is a *preferred extension* w.r.t. AF iff S is a maximal (w.r.t. set inclusion) element of $\mathcal{AS}(AF)$. Built upon the computation of the admissible extensions Π_{adm} U. Egly et al. construct a second guess using the new predicates `inN/1` and `outN/1`. This guess of a subset $S' \subset S$ is based on disjunction instead of default negation. The use of disjunction allows both predicates, `inN/1` and `outN/1`, to hold for an argument. The saturation is performed such that both predicates, `inN/1` and `outN/1`, can be derived for each argument $a \in S'$ where S' does not characterize an admissible extension. Now if this saturation is successful for each $S' \subset S$, the underlying interpretation is considered an

answer-set for the encoding. This procedure is perfomed with the help of the newly introduced predicate `spoil/0`, which is our witness for the saturation:

$$\begin{aligned}\Pi_{spoil} = \{&inN(X) \vee outN(X) \leftarrow out(X).\\ &inN(X) \leftarrow in(X).\\ &spoil \leftarrow eq.\\ &spoil \leftarrow inN(X), inN(Y), defeat(X,Y).\\ &spoil \leftarrow inN(X), outN(Y), defeat(Y,X), undefeated(Y).\\ &inN(X) \leftarrow spoil, arg(X).\\ &outN(X) \leftarrow spoil, arg(X).\\ &\leftarrow not\ spoil\}\end{aligned}$$

The constraint $\leftarrow not\ spoil$ makes sure that a guess only survives if it is saturated. Π_{spoil} does need some additional intermediary predicates which are introduced by the module $\Pi_{helpers}$. `eq/0` is a predicate, relying on `eq_upto/1`, which holds if $S' = S$. The unary predicate `undefeated/1` holds for an argument if it is undefeated by the elements of S'.

$$\begin{aligned}\Pi_{helpers} = \Pi_{<} \cup \{&eq\_upto(Y) \leftarrow inf(Y), in(Y), inN(Y).\\ &eq\_upto(Y) \leftarrow inf(Y), out(Y), outN(Y).\\ &eq\_upto(Y) \leftarrow succ(Z,Y), in(Y), inN(Y), eq\_upto(Z).\\ &eq\_upto(Y) \leftarrow succ(Z,Y), out(Y), outN(Y), eq\_upto(Z).\\ &eq \leftarrow sup(Y), eq\_upto(Y).\\ &undefeated\_upto(X,Y) \leftarrow inf(Y), outN(X), outN(Y).\\ &undefeated\_upto(X,Y) \leftarrow inf(Y), outN(X), not\ defeat(Y,X).\\ &undefeated\_upto(X,Y) \leftarrow succ(Z,Y), undefeated\_upto(X,Z),\\ &\qquad\qquad outN(Y).\\ &undefeated\_upto(X,Y) \leftarrow succ(Z,Y), undefeated\_upto(X,Z),\\ &\qquad\qquad not\ defeat(Y,X).\\ &undefeated(X) \leftarrow sup(Y), undefeated\_upto(X,Y)\}\}\end{aligned}$$

The encoding of the preferred extension now is composed as follows:

$$\Pi_{pref} = \Pi_{adm} \cup \Pi_{helpers} \cup \Pi_{spoil}$$

For further details and an elaborate documentation, we refer the interested reader to [24] .

The last approach, which is summarized by F. Toni and M. Sergot, is the encoding of the ideal semantics, proposed by W. Faber and S. Woltran in [28], which is based on manifold answer set programs. Those manifold answer set programs are encodings that allow multiple forms of meta-reasoning. For a detailed description please refer to [53].

3.2 Previous Encodings of the Resolution Based Grounded Semantics

In this subsection we are presenting two alternative approaches for encoding the resolution-based grounded semantics. The first is a simple ASP program without any optimization statements and based on a verification algorithm proposed by Baroni et al. [5]. The second approach is based on the meta modeling techniques provided by Gebser et al. [32].
Before we present these two different kinds of encodings we will describe how an argumentation framework, for which we want to determine the extensions, is represented in ASP. Let $AF = \langle A, R \rangle$ be the fixed input argumentation framework, then it is translated into an input database $\hat{F}$ as follows:

$$\hat{F} = \{arg(a) \mid a \in A\} \cup \{defeat(a, b) \mid (a, b) \in R\}$$

When describing the ASP encodings we stick to the conventions used in [22] and split them into modules which we describe and treat them separately.
Prior to describing the different encodings, we provide three modules which will be part of all the approaches we present, including our realization of the verification algorithm [5].
The first module π_{cf} together with $\hat{F}$ constitutes a program, which guesses a set $S \in A$, where the unary predicates `in/1` and `out/1` indicate that an argument a is either $a \in S$ or $a \notin S$. With the help of a constraint rule Π_{cf}, which we introduced in the previous subsection, one removes all answer sets which are not conflict-free.

A last common module Π_{range} computes the range $S^{\oplus} = S \cup S^{+}$ of a subset $S \subseteq A$.

$$\begin{aligned}\Pi_{range} = \{&in\_range(X) \leftarrow \arg in(X).\\ &in\_range(X) \leftarrow in(Y), defeat(Y, X).\\ &not\_in\_range(X) \leftarrow arg(X), not\ in\_range.\}\end{aligned}$$

Now we provide definitions which we will need to describe the encodings. The first definition is the cut of an argumentation framework G, denoted as $CUT(G)$ and obtained by suppressing the arguments to a specific range in the grounded extensions.

Definition 61. *The cut of an argumentation framework $G = \langle A, R \rangle$, denoted as $CUT(G)$ is defined as the argumentation framework which is obtained by the restriction of G to the arguments A of G, where the arguments of $\mathcal{GE}(G)^{\oplus}$ are excluded. $CUT(G) = G \downarrow_{A \setminus \mathcal{GE}(G)^{\oplus}}$*

The next definition deals with the notion of stability of a set $S \in A$ in another set $T \in A$ w.r.t. to an argumentation framework $G = \langle A, R \rangle$, and is denoted as $st_{(G)}(S, T)$.

Definition 62. *[5] Given an argumentation framework $G = \langle A, R \rangle$ and two sets $S, T \in A$ then S is stable in T w.r.t. G, denoted as $st_{(G)}(S, T)$, iff $\forall a \in (T \setminus S) : a \in (S \cap T)^{+}$.*

Taken from [5] we provide a formal characterization of the extensions of the resolution-based grounded semantics, which consists of three conditions checked in the verification algorithm in [5], this constitutes the foundation of the encoding provided by S. Woltran et al. [22] as well as of the encoding, which is proposed in this thesis.

Definition 63. *[5] Let G be an argumentation framework $G = \langle A, R \rangle$, such that $CUT(G) \neq \langle \emptyset, \emptyset \rangle$ and $MR(CUT(G)) = \emptyset$, letting $W = \Pi_G$, where $\Pi_G = \bigcup_{V \in MR(CUT(G))} V$, $S = \mathcal{GE}(G)$ and $T = U \backslash S^{\oplus}$, then $U \in \mathcal{E}_{\mathcal{GR}*}(G)$ if the following three conditions hold:*

1. $U \cap S^{\oplus} = S$.
2. $st_{CUT(G)}(T, W)$.
3. $(T \cap W^C) \in \mathcal{E}_{\mathcal{GR}*}(CUT(G) \downarrow_{W^C \backslash (T \cap W)^+})$

Furthermore we provide the definition of the set of strongly connected components of an argumentation framework $G = \langle A, R \rangle$, denoted as $SCCS(G)$, which is taken from graph theory, describing the set of vertices, or arguments in our case, that are the maximal connected subgraphs of a graph or an argumentation framework, respectively.

Definition 64. *[5] The set of strongly connected components of an argumentation framework $G = \langle A, R \rangle$ is denoted as $SCCS(G)$. The* strongly connected component decomposition *of G partitions the arguments A into equivalence classes induced by the relation $p(x, y)$, defined over $A \times A$. $p(x, y)$ holds iff $x = y$ or there exists a directed path from x to y and a directed path from y to x in G.*

Based on the following lemma, we define the set of minimal relevant components $MR(G)$ of an argumentation framework G.

Lemma 1. *[5] Given a non-empty argumentation framework $G = \langle A, R \rangle$ with $|SCCS(G)| = 1$, the condition (i) for any full resolution β of G $\exists x$ such that $\{x\}^-_{G_\beta} = \emptyset$ is equivalent to the conjunction (ii) of the following three conditions:*

(a) $\forall x \in A, \langle x, x \rangle \notin R$;

(b) R *is symmetric, i.e.* $\langle x, y \rangle \in R \Leftrightarrow \langle y, x \rangle \in R$;

(c) the undirected graph $\bar{G}$ formed by replacing each (directed) pair $\{\langle x, y \rangle, \langle y, x \rangle\}$ with a single undirected edge $\{x, y\}$ is acyclic.

Now, as a last definition before we start describing the encodings, we provide the set of minimal relevant strongly connected components $MR(G)$ of an argumentation framework G.

Definition 65. *[5, 22] Given a non-empty argumentation framework $G = \langle A, R \rangle$, a set $S \in SCCS(G)$ is minimal relevant, if S is a minimal element of $\prec$ and $G \downarrow_S$ satisfies the conditions (a)-(c) of Lemma 1. $\prec$ denotes a partial order over the set $SCCS(G) = \{S_1, \ldots, S_n\}$, denoted as $(S_i \prec S_j)$ for $i \neq j$, which is defined if $\exists x \in S_i, y \in S_j$ such that there is a direct path from x to y in G.*
The set of minimal relevant strongly connected components of an argumentation framework G is denoted as $MR(G)$.

Verification Algorithm Based Encoding

In their article [22] W. Dvořák et al. propose an encoding of the resolution-based grounded semantics which is based on the verification algorithm listed in Listing 4.1, like the encoding that we introduce in this thesis. This verification algorithm was originally proposed by Baroni et al. in their article *On the resolution based family of abstract argumentation and its grounded instance* [5]. The Guess & Check paradigm is the major modeling technique applied in this realization of the semantics. As mentioned before, this encoding incorporates the modules Π_{cf} and $\Pi_{<}$ the first one being used to guess all possible conflict-free sets and the latter one is the foundation of the iterative applied check procedure. This procedure cheks whether the conditions of Definition 65 and 63 are met by any guess. Due to the fact that our encoding, is based on the same verification algorithm, we provide a detailed description of this approach in Chapter 4 to support the reader with better means for understanding both encodings and their differences.

Meta ASP Encodings

In their article *Complex optimization in answer set programming* [32], M. Gebser et al. propose an approach to address complex optimization and preference handling criteria such as inclusion-based minimization. Without the meta ASP modeling techniques this would require other, more cumbersome modeling techniques, such as saturation. The metasp approach is based on a series of reusable ASP encodings, which handle those different optimization and preference handling criteria, and enable their availability for use by other ASP programs. The meta-modeling technique is basically composed of three steps: First, the ASP encoding is reified by the grounder gringo, which generates the ground version of the given program and returns the resulting facts. In a second step the grounder is executed again but this time with the result of the previous grounding process together with the meta-programs which realize the optimization part. In the third step claspD is executed in order to perform the solving task.
The meta-modeling approach offers a collection of encodings which provides answer-set inclusion minimization among other complex optimization capacities. Therfore it poses an efficient approach of tackling the problem of encoding the resolution-based grounded semantics. In this subsection we describe two meta ASP encodings of resolution-based grounded semantics as proposed by W. Dvořák et al. in [22], which make use of the answer-set inclusion minimization and omits prioritization and weights. These two encodings use inclusion minimization for determining the resolutions and they differ in the fact that the first encoding calculates the grounded extension for a guessed resolution explicitly, while the second encoding acquires the complete extensions for the guessed resolution, and subsequently applies subset minimization to retrieve the grounded extensions.

Both meta ASP encodings use the same ASP module Π_{res} to determine the resolutions of an argumentation framework :

$$\Pi_{res} = \{defeat\_minus\_beta(X,Y) \leftarrow defeat(X,Y), not\ defeat\_minus\_beta(Y,X), X \neq Y.$$
$$defeat\_minus\_beta(X,Y) \leftarrow defeat(X,Y), not\ defeat(Y,X).$$
$$defeat\_minus\_beta(X,X) \leftarrow defeat(X,X).\}$$

We continue by describing the modules which used realize the first metasp approach. We again need an order, which is put on the domain of arguments `arg/1`, and therefor use the module $\Pi_{<}$, which we already defined in the previous subsection, and its helper predicates `inf/1`, `succ/2` and `sup/1`. Now that an order has been defined, we are able to provide the next module $\Pi_{defended}$:

$$\begin{aligned}\Pi_{defended} = \{&defended\_upto(X,Y) \leftarrow inf(Y), in(X), not\ defeat\_minus\_beta(Y,X).\\ &defended\_upto(X,Y) \leftarrow inf(Y), in(Z), defeat\_minus\_beta(Z,Y),\\ &\qquad defeat\_minus\_beta(Y,X).\\ &defended\_upto(X,Y) \leftarrow succ(Z,Y), defended\_upto(X,Z),\\ &\qquad not\ defeat\_minus\_beta(Y,X).\\ &defended\_upto(X,Y) \leftarrow succ(Z,Y), in(V), defeat\_minus\_beta(V,Y),\\ &\qquad defeat\_minus\_beta(Y,X).\\ &defended(X) \leftarrow sup(Y), defended\_upto(X,Y).\}\end{aligned}$$

Having defined $\Pi_{defended}$, one is able to explicitly determine the grounded extension as follows:

$$\Pi_{grd} = \Pi_{<} \cup \Pi_{defended} \cup \{in(X) \leftarrow defended(X)\}$$

Π_{grd*_metasp} combines all the previous modules and adds the statement for minimizing `in/1`:

$$\Pi_{grd*\_metasp} = \Pi_{grd} \cup \Pi_{res} \cup \{\#minimize[in]\}$$

The second encoding does differ slightly and as already mentioned it initially acquires the complete extensions for the guessed resolutions and then uses subset minimization. But, since complete extensions are based on admissible sets, for this case an intermediary module is used. A module which has already been introduced in 3.1 and which has been slightly adapted to fit into the metasp approach:

$$\begin{aligned}\Pi_{adm} = \Pi_{cf} \cup \{&defeated(X) \leftarrow in, defeat(Y,X).\\ &\leftarrow in(X), defeat(Y,X), not\ defeated(Y).\}\end{aligned}$$

Now the module Π_{com} is composed as follows:

$$\Pi_{com} = \Pi_{adm} \cup \{ undefended(X) \leftarrow defeat\_minus\_beta(Y, X), not\ defeated(Y). \\ \leftarrow out(X), not undefended(X) \}.$$

The extensions in the second encoding are now computed by acquiring the subset minimization from the complete extensions.

$$\Pi'_{grd*\_metasp} = \Pi_{com} \cup \Pi_{res} \cup \{ \#minimize[in] \}$$

Both versions of meta ASP encodings are computed by applying Π to the given argumentation frameworks $\hat{F}$, where Π stands for either Π_{grd*_metasp} or Π'_{grd*_metasp}. This step is performed by calling `gringo` with the `-reify` option on $\Pi(\hat{F})$. Now the output is redirected with the *pipe* command once more to gringo, where *meta.lp*, *metaO.lp*, *metaD.lp* are applied together with the statement `(echo "optimize(1,1,incl).")`, triggering the use of subset inclusion for the optimization step with *priority* and *weight* set to 1:

```
gringo -reify Π(F̂) | gringo -{meta.lp, metaO.lp, metaD.lp} \
<(echo "optimize(1,1,incl).")  | claspD 0
```

Finally claspD with the option 0 (which tells clasp to compute all answer sets) is called to conduct the solving process.

CHAPTER 4

New Encodings of the Resolution Based Grounded Semantics

In this chapter we present our new approach for encoding the resolution-based grounded semantics. This approach is fully based on the verification Algorithm 4.1, proposed by Baroni et al [5] and is realized with answer-set programming as previously mentioned. Like S. Woltran et al. [22] for their implementation of the resolution-based grounded semantics, we also follow the ASPARTIX approach which was proposed by U. Egly et al. [25]. To begin we present a brief outline before presenting the details of our implementation. Since the approach approach by S. Woltran et al. [22] has been developed in parallel to our solution, we conclude this chapter with a detailed description of this realization of the resolution-based grounded semantics. Thus the reader is provided with the neccesary means for understanding any differences and the latter performance comparison.

4.1 Outline

We describe how the argumentation framework is represented for which we want to determine the extensions. Let $AF = \langle A, R\rangle$ be the fixed input argumentation framework, then it is translated into an input database $\hat{F}$ as follows:

$$\hat{F} = \{arg(a) \mid a \in A\} \cup \{att(a,b) \mid (a,b) \in R\}$$

As we did with the previous encodings, we stick to the conventions, used in [22], with this novell approach, and split our encoding into modules which we describe and treat separately.

Our encoding starts with guessing all possible sets $S \subseteq A$ of $G = \langle A, R\rangle$ and then uses the verification algorithm to filter out the sets, not belonging to the set of the resolution-based grounded extensions $\mathcal{E}_{\mathcal{GR}^*}(G = \langle A, R\rangle)$, by checking the conditions of Definition 65 and 63 (Definition 63 demands the iterative structure of our check procedure).

Listing 4.1: Verifying that $U \in \mathcal{E}_{\mathcal{GR}^*}(G = \langle A, R \rangle)$

```
1  procedure GR* − VER(G = ⟨A, R⟩, U) returns boolean
2  S := GE(G)
3  if (U ∩ S^⊕ ≠ S) then
4          return false
5  end if
6  T := U\S
7  if CUT(G) = ⟨∅, ∅⟩ or MR(CUT(G)) = ∅ then
8          if T = ∅ then
9                  return true
10         else
11                 return false
12         end if
13 else
14         W := Π_G
15 end if
16 if ¬st_CUT(G)(T, W) then
17         return false
18 else
19         return GR* − VER(CUT(G) ↓_{W^C \ (T∩W)^+} (T ∩ W^C))
20 end if
21 end
```

4.2 Encoding

Now we start with the detailed description of our implementation. The first module Π_{cf}, together with $\hat{F}$ constitutes a program, which guesses a set $S \in A$, where the unary predicate `u/1` indicates that an argument a is either $a \in S$ or $a \notin S$. With the help of a constraint rule, Π_{cf} removes all answer sets which are not conflict free.

$$\begin{aligned}\Pi_{cf} = \{&u(X) \leftarrow arg(X), not\ \neg u(X).\\ &\neg u(X) \leftarrow arg(X), not\ u(X).\\ &\leftarrow u(X), u(Y), att(X,Y).\}\end{aligned}$$

Based on the order imposed on the domain elements, we introduce the module Π_{init}, which initializes the iteration by supplying predicates representing the arguments of the argumentation framework and the arguments of set S, with an extra variable I. This variable, is assigned the smallest domain element in order to represent the first step in the iteration, due to predicate `inf/1` from module $\Pi_{<}$. In the remaining modules of our encoding, we will use the last variable of each predicate to label it with the current domain element associated with the iteration.

$$\begin{aligned}\Pi_{init} = \{&iarg(X,I) \leftarrow arg(X), inf(I).\\ &iu(X,I) \leftarrow u(X), inf(I).\} \cup \Pi_{<}\end{aligned}$$

The predicates `iarg/2` and `iu/2` represent the labelled copies of arguments of the input framework and the guessed set S. The following module Π_{grd} is a module we borrowed from the existing encoding of the grounded extension and modified to be repeatedly used, once in each iteration, by labeling its predicates with the variable I, (like U. Egly et al. [24, 25]).

The first predicate of Π_{grd} is `defeat/3` which denotes any defeated arguments in iteration step I. With the remaining rules of the module, we determine the grounded extension of iteration N, which is the least fixed point of the characteristic function F_{AF} that is represented by the predicate `defendedN/2`, and computed via `defended_upto/3`. The predicate `in/2` simply represents the extension membership of an argument in the grounded extension of the current iteration.

$$
\begin{aligned}
\Pi_{grd} = \{ & \mathit{defeat}(X,Y,I) \leftarrow \mathit{iarg}(X,I), \mathit{iarg}(Y,I), \mathit{att}(X,Y). \\
& \mathit{defended\_upto}(X,Y,I) \leftarrow \mathit{inf}(Y), \mathit{iarg}(X,I), \mathit{not}\ \mathit{defeat}(Y,X,I). \\
& \mathit{defended\_upto}(X,Y,I) \leftarrow \mathit{inf}(Y), \mathit{in}(Z,I), \mathit{defeat}(Z,Y,I), \mathit{defeat}(Y,X,I). \\
& \mathit{defended\_upto}(X,Y,I) \leftarrow \mathit{succ}(Z,Y), \mathit{defended\_upto}(X,Z,I), \\
& \qquad\qquad \mathit{not}\ \mathit{defeat}(Y,X,I). \\
& \mathit{defended\_upto}(X,Y,I) \leftarrow \mathit{succ}(Z,Y), \mathit{defended\_upto}(X,Z,I), \\
& \qquad\qquad \mathit{in}(V,I), \mathit{defeat}(V,Y,I), \mathit{defeat}(Y,X,I). \\
& \mathit{defended}(X,I) \leftarrow \mathit{sup}(Y), \mathit{defended\_upto}(X,Y,I). \\
& \mathit{in}(X,I) \leftarrow \mathit{defended}(X,I).\}
\end{aligned}
$$

The components Π_{cf}as well as Π_{grd}, in their original form, have already been proposed by Egly et al. [24, 25]. Their predicate names just have been adapted to fit our encoding and their rule heads and bodies have been retrofitted to comply with our approach by performing an iteration over sets.

We continue with Π_{unps} where we check whether $(U \cap S^{\oplus}) \neq S$. Therefore we first determine $S^{\oplus}$, the range of S which is represented in our encoding by `susp/2`. Now we are ready to compute $(U \cap S^{\oplus})$, which is denoted by the predicate `unps/2`. Now with the constraint rules of Π_{unps} we discard all answer-sets, where $(U \cap S^{\oplus}) \neq S$.

$$
\begin{aligned}
\Pi_{unps} = \{ & \mathit{susp}(X,I) \leftarrow \mathit{in}(X,I). \\
& \mathit{susp}(X,I) \leftarrow \mathit{iarg}(X,I), \mathit{in}(S,I), \mathit{att}(S,X). \\
& \mathit{unps}(X,I) \leftarrow \mathit{iu}(X,I), \mathit{susp}(X,I). \\
& \leftarrow \mathit{unps}(X,I), \mathit{not}\ \mathit{in}(X,I). \\
& \leftarrow \mathit{in}(X,I), \mathit{not}\ \mathit{unps}(X,I).\}
\end{aligned}
$$

The next module, Π_{cut} determines the cut of our given argumentation framework, G. Recall that $CUT(G) = G \downarrow_{A \setminus \mathcal{GE}(G)^{\oplus}}$.

$$\Pi_{cut} = \{cut(X, I) \leftarrow iarg(X, I), not\ susp(X, I).\}$$

Now that we defined the module for deriving the cut, we can continue by determining the set of minimal relevant strongly connected components $SCCS(CUT(G))$, of the cut of G. Again we recall that for a non-empty argumentation framework $G = \langle A, R\rangle$, a set $S \in SCCS(G)$ is minimal relevant, if S is a minimal element of $\prec$ and $G \downarrow_S$ satisfies the conditions (a)-(c) of Lemma 1.
The strongly connected components of an argumentation framework are defined per Definition 64 as equivalence classes, induced by a relation $p(x, y)$ which holds if $x = y$ or there exists a directed path from x to y and a directed path from y to x in G. First we determine each directed path, which is represented by the predicate `path3`. Since we want to determine $MR(CUT(G))$, we restrict the set of arguments which we consider for the path, to the arguments of the cut. This is achieved by the use of the predicate `cut/2` in the rules body. The last two rules introduce the predicate `p/3` which represents the relation $p(x, y)$.

$$\begin{aligned}\Pi_{scc\_opt} = \{&path(X, Y, I) \leftarrow cut(X, I), cut(Y, I), att(X, Y).\\ &path(X, Z, I) \leftarrow cut(X, I), cut(Z, I), path(X, Y, I), path(Y, Z, I).\\ &p(X, Y, I) \leftarrow cut(X, I), cut(Y, I), X == Y.\\ &p(X, Y, I) \leftarrow path(X, Y, I), path(Y, X, I), X < Y.\}\end{aligned}$$

After initial test runs we decided to look for ways of improving our encoding. The original draft of Π_{scc_opt} left room for optimization. It is denoted as Π_{scc_opt} and a comparison of its performance to Π_{scc_opt} can be found in the subsequent chapter. We applied an optimization technique, which is known as symmetry breaking within the answer-set programming community. It is a technique aiming to eliminate redundant parts in the search space and thus reducing computation time. In this case we generated a large amount of answer sets where a redundant symmetry was present within the `p/3` predicates. Those predicates represent the symmetric relation $p(x, y)$ which, as already stated, holds if there exists a bidirectional path between x and y or if $x = y$. Now since this relation is symmetrical it is sufficient to derive only one relation of the pair $p(x, y)$ and $p(y, x)$, which is achieved by exploiting the order of the domain elements once more.

$$\begin{aligned}\Pi_{scc} = \{&path(X, Y, I) \leftarrow cut(X, I), cut(Y, I), att(X, Y).\\ &path(X, Z, I) \leftarrow cut(X, I), cut(Z, I), path(X, Y, I), path(Y, Z, I).\\ &p(X, Y, I) \leftarrow cut(X, I), cut(Y, I), X == Y.\\ &p(X, Y, I) \leftarrow path(X, Y, I), path(Y, X, I).\}\end{aligned}$$

As can be seen above, to acquire an optimized answer-set program Π_{scc_opt} we modified our initial approach Π_{scc} for replacing the following rule

$$p(X, Y, I) \leftarrow path(X, Y, I), path(Y, X, I).$$

with

$$p(X,Y,I) \leftarrow path(X,Y,I), path(Y,X,I), X < Y.$$

As can be seen this change permits only one of the equivalent relations $p(x,y)$ and $p(y,x)$ to be included in the answer-set. With these results, we can continue with the module Π_{minscc} that is used to filter out from the SCCs, which were determined in Π_{scc_opt}, and are a minimal element of $\prec$. With the help of the rule which introduces the predicate `sccattacked/2`, we check whether any of the elements of an SCC is attacked from an element of another SCC. Hence for each minimal SCC, represented in this encoding by the predicate `me`, the default negation of `sccattacked/2` must hold. Furthermore, we introduce a predicate `label/2` to label each SCC with a selected argument, which we determine with the help of the order that we put on the domain of elements with $\Pi_{<}$ and the predicate `bigger/2`.

$$\begin{aligned}\Pi_{minscc} = \{&sccattacked(X,I) \leftarrow p(X,Y,I), p(O,\_,I), att(O,Y), not\ p(O,X,I).\\ &me(X,I) \leftarrow p(X,\_,I), not\ sccattacked(X,I).\\ &bigger(X,Y,I) \leftarrow p(X,Y,I), p(Y,Z,I), X < Y, Y < Z.\\ &label(Y,I) \leftarrow p(X,Y,I), not\ bigger(X,Y,I), me(Y,I), X < Y.\\ &scc(X,L,I) \leftarrow p(X,L,I), label(L,I).\}\end{aligned}$$

We split the task of checking whether $MR(CUT(G))$ satisfies the conditions (a)-(c) of Lemma 1 into three modules, which we merge with module Π_{mr} later on. Let us begin with Π_{mr1}, with which we cover conditions (a) and (b). Condition (a), $\forall x \in A, \langle x,x\rangle \notin R$, states that no self-attacks are allowed within a minimal relevant component. With the help of the predicate `sccatt/3` we identify all the attacks within a SCC and the head of the second rule Π_{mr1}, `eselfatt/2` determines if self-attacks exist. The predicate `scclemma8a/2` is derived if there are no single self-attacks in a strongly connected component and condition (a) is satisfied. Condition (b), states that R, the attack relations within the SCC, have to be symmetric, i.e. $\langle x,y\rangle \in R \Leftrightarrow \langle y,x\rangle \in R$. With the second to last rule we check whether there is an attack $\langle x,y\rangle \in R$ for which there does not exist an $\langle y,x\rangle \in R$ and in that case the head of the rule can be derived. We use the last rule to label a SCC with the predicate `scclemma8b` if no asymmetries are present.

$$\begin{aligned}\Pi_{mr1} = \{&sccatt(X,Y,I) \leftarrow att(X,Y), scc(X,L,I), scc(Y,L,I).\\ &eselfatt(L,I) \leftarrow scc(X,L,I), att(X,X).\\ &scclemma8a(L,I) \leftarrow label(L,I), not\ eselfatt(L,I).\\ &asymmetric(L,I) \leftarrow sccatt(X,Y,I), not\ sccatt(Y,X,I), scc(X,L,I),\\ &\qquad\qquad scc(Y,L,I).\\ &scclemma8b(L,I) \leftarrow label(L,I), not\ asymmetric(L,I).\}\end{aligned}$$

It remains to check the last condition (c) of Lemma 1, which is a little bit more cumbersome to check than the previous conditions. This involves the use of aggregates in this encoding. We

recollect that the undirected graph $\bar{G}$ formed by replacing each (directed) pair $\{\langle x, y\rangle, \langle y, x\rangle\}$ with a single undirected edge $\{x, y\}$ has to be acyclic. In this case, G is the strongly connected component and we describe with the first rule of the next module, the undirected graph $\bar{G}$ by replacing each (directed) pair $\{\langle x, y\rangle, \langle y, x\rangle\}$ with a single undirected edge $\{x, y\}$. Condition (b) of Lemma 1 guarantees that all the directed attacks of the current SCC are symmetric. Hence vertices that have been connected in the initial graph remain connected in the undirected graph. Together with the fact that we are dealing with strongly connected components, condition (b) assures that the acquired undirected graph is connected, which makes the cycle detection much simpler than it usually would be, since it only requires a rule to check whether the number of arguments is greater than the number of undirected edges in the graph $\bar{G}$. The first rule of Π_{cyclic} transforms the pairs of attacks into an undirected edge. This enables us to check whether the resulting graph is acyclic. `cyclic/2` can be derived for the current SCC if the number of undirected edges is greater than or equal to the number of arguments in the SCC.

$$\begin{aligned}\Pi_{cyclic} = \{&udgscc(X,Y,I) \leftarrow sccatt(X,Y,I), sccatt(Y,X,I), X < Y.\\ &cyclic(L,I) \leftarrow \#count\{Z : scc(Z,L,I)\} = V,\\ &\qquad V <= \#count\{X,Y : scc(X,L,I), scc(Y,L,I),\\ &\qquad\qquad udgscc(X,Y,I)\},\\ &\qquad \#int(V), label(L,I).\}\end{aligned}$$

$$\begin{aligned}\Pi_{mr2} = \{&scclemma8c(L,I) \leftarrow label(L,I), not\ cyclic(L,I).\\ &mr(X,I) \leftarrow scc(X,L,I), scclemma8a(L,I), scclemma8b(L,I),\\ &\qquad scclemma8c(L,I).\}\end{aligned}$$

Module Π_{mr2} now makes use of Π_{cyclic} in order to check for a given SCC whether it is acyclic and then introduces a new predicate `mr/2`, which marks any strongly connected component in the current iteration I that fulfills all the conditions of Lemma 1, making it a minimal relevant SCC. For convenience reasons we unite the modules responsible for this task into a single module:

$$\Pi_{mr} = \Pi_{mr1} \cup \Pi_{cyclic} \cup \Pi_{mr2} \cup \Pi_{minscc} \cup \Pi_{scc\_opt} \cup \Pi_{cut}$$

Having derived all the arguments belonging to $MR(CUT(G))$, we can continue checking whether $U \in \mathcal{E}_{\mathcal{GR}*}(G)$ according to Definition 63. The next module conducts exactly the same checks as listed in the lines 7 to 15 from the algorithm 4.1, proposed by Baroni et al. [5].

Listing 4.2: Excerpt from Listing 4.1 (lines 7-15)

```
7  if CUT(G) = ⟨∅,∅⟩ or MR(CUT(G)) = ∅ then
8          if T = ∅ then
9                  return true
10         else
11                 return false
```

```
12          end if
13  else
14          W := Π_G
15  end if
```

The predicate `ecut/1` can be derived if $CUT(G) \neq \langle \emptyset, \emptyset \rangle$ and the predicate `emr/1` if $MR(CUT(G)) \neq \emptyset$. The predicate `et/1` can be derived for an iteration, if $T \neq \emptyset$. Now the two last rules of Π_{check} are constraints to remove all answer sets where $CUT(G) = \langle \emptyset, \emptyset \rangle$ or $MR(CUT(G)) = \emptyset$ and $T \neq \emptyset$.

$$\begin{aligned}\Pi_{check} = \{&t(X,I) \leftarrow iu(X,I), not\ in(X,I).\\ &emr(I) \leftarrow mr(\_,I).\\ &ecut(I) \leftarrow cut(\_,I).\\ &escclemma8c(I) \leftarrow scclemma8c(\_,I).\\ &et(I) \leftarrow t(\_,I).\\ &\leftarrow arg(I), et(I), not\ emr(I).\\ &\leftarrow arg(I), et(I), not\ ecut(I).\}\end{aligned}$$

As one can conclude from the listing below, our next module Π_{stable} incorporates the check whether $st_{CUT(G)}(T,W)$ holds for the current iteration. Otherwise if $CUT(G) \neq \langle \emptyset, \emptyset \rangle$ and $MR(CUT(G)) \neq \emptyset$ the next iteration is instantiated in $\Pi_{iterate}$ and the algorithm proceeds.

Listing 4.3: Excerpt from Listing 4.1(lines 16-21)

```
16  if ¬st_CUT(G)(T,W) then
17          return false
18  else
19          return GR* − VER(CUT(G) ↓_{W^C \ (T∩W)+}, (T ∩ W^C))
20  end if
21  end
```

We recall Definition 62: given an argumentation framework $G = \langle A, R \rangle$ and two sets $S, T \in A$ then S is stable in T w.r.t. G, denoted as $st_{(G)}(S,T)$, iff $\forall a \in (T \backslash S) : a \in (S \cap T)^+$. Now, for the check whether $st_{CUT(G)}(T,W)$ holds, we first have to determine the set $(W \backslash T)$. It is represented by the predicate `wot/2`. Next we want to determine $(T \cap W)$ before we acquire $(T \cap W)^+$. $(T \cap W)$ is computed by the second rule and represented by the predicate `tnw/2`. Now $(T \cap W)^+$ is derived with the second to last rule and represented by the predicate `tnwp/2`. With the constraint rule in Π_{stable} we now discard all answer-sets where $\neg st_{CUT(G)}(T,W)$ holds.

$$\begin{aligned}\Pi_{stable} = \{&wot(X,I) \leftarrow mr(X,I), nott(X,I).\\ &tnw(X,I) \leftarrow t(X,I), mr(X,I).\\ &tnwp(X,I) \leftarrow tnw(Y,I), iarg(X,I), att(Y,X).\\ &\leftarrow wot(X,I), not\ tnwp(X,I).\}\end{aligned}$$

The first two rules of the module $\Pi_{iterate}$ are our stopping criteria which mean that we do have an answer-set to our input and no further iteration is necessary. The stopping criteria are met if either $CUT(G) = \langle \emptyset, \emptyset \rangle$ and $T = \emptyset$ or $MR(CUT(G)) = \emptyset$ and $T = \emptyset$.
The remaining rules of $\Pi_{iterate}$ build the arguments for the next iteration $CUT(G) \downarrow_{W^C \setminus (T \cap W)^+}$ and $(T \cap W^C)$. We start with computing the set W^C, which is represented by the predicate `wc/2`. With the next rule we determine the predicate `wcotnwp/2` which is the set of arguments in $W^C \setminus (T \cap W)^+$. Now we are ready to compute the cut $CUT(G) \downarrow_{W^C \setminus (T \cap W)^+}$, which is represented by `cutgwcotnwp/2` and constitutes the arguments of the argumentation framework for the next iteration which are represented by `iarg/2` and computed in the succeeding rule, where they are assigned by the successor label for the next iteration with help of the predicate `succ/2` which is provided by the order we imposed on the domain elements in $\Pi_<$. It then remains to compute the set $(T \cap W^C)$ which is done in the second to last rule with the predicate `tnwc/2` in its rule head. The last rule takes the arguments for which the predicate `tnwc/2` can be derived, assigns them a succeeding iteration label, as we did for `iarg` and equips them with the predicate `iu/2`, which constitutes the set U in the next iteration.

$$
\begin{aligned}
\Pi_{iterate} = \{ & stop(I) \leftarrow arg(I), not\ escclemma8c(I), not\ et(I). \\
& stop(I) \leftarrow arg(I), not\ ecut(I), not\ et(I). \\
& wc(X,I) \leftarrow iarg(X,I), not\ mr(X,I). \\
& wcotnwp(X,I) \leftarrow wc(X,I), not\ tnwp(X,I). \\
& cutgwcotnwp(X,I) : -cut(X,I), wcotnwp(X,I). \\
& iarg(X,II) \leftarrow cutgwcotnwp(X,I), succ(I,II), not\ stop(I). \\
& tnwc(X,I) \leftarrow wc(X,I), t(X,I). \\
& iu(X,II) \leftarrow tnwc(X,I), succ(I,II), not\ stop(I).\}
\end{aligned}
$$

In the end we combine all the modules, which result in our encoding of the resolution-based grounded semantics $\Pi_{veruegr_opt}$:

$$\Pi_{veruegr} = \Pi_{mr} \cup \Pi_{check} \cup \Pi_{stable} \cup \Pi_{iterate} \cup \Pi_{unps} \cup \Pi_{grd} \cup \Pi_{init} \cup \Pi_{grd} \cup \Pi_{cf} \cup \Pi_<$$

For testing how our improved encoding performs in comparison to the initial version, we also combine the modules of the initial encoding without the optimization to form a fully functional encoding $\Pi_{veruegr}$. $\Pi_{veruegr}$ consists of the same modules as $\Pi_{veruegr_opt}$, with the only difference being that Π_{mr} uses the initial module Π_{scc} instead of Π_{scc_opt}:

$$\Pi_{mr} = \Pi_{mr1} \cup \Pi_{cyclic} \cup \Pi_{mr2} \cup \Pi_{minscc} \cup \Pi_{scc} \cup \Pi_{cut}$$

4.3 Another Variant of a Verification-Algorithm Based Encoding

Due to the fact that the encoding which is proposed by W. Dvořák et al. [22] shares a lot of similarities with our realization of the resolution-based grounded semantics, we decided to discuss

its details in our implementation chapter. As already mentioned before, this encoding incorporates the initially introduced modules Π_{cf} and $\Pi_{<}$, since the first one is used to guess all possible conflict-free sets and the latter one is the foundation of the iteratively applied check procedure. It is checked whether the conditions of Definition 65 and 63 are met by any guess.

The first module we deal with is Π_{arg_set}, which introduces three new predicates `arg_set/2`, `inU/2` and `defeatN/3`. These predicates are a copy of the arguments present in the guess and the defeats, and are equipped with an additional variable N, which is needed for the iterative procedure. The predicate `inf/1` is used to acquire the smallest identifier of the order resulting from $\Pi_{<}$.

$$\begin{aligned}\Pi_{arg\_set} = \{&arg\_set(N,X) \leftarrow arg(X), inf(N).\\ &inU(N,X) \leftarrow in(X), inf(N).\\ &defeatN(n,Y,X) \leftarrow arg\_set(N,X), arg_set(N,Y), defeat(Y,X).\}\end{aligned}$$

The following two modules, $\Pi_{defendedN}$ and $\Pi_{groundN}$ determine the grounded extension of iteration N, which is the least fixed point of the characteristic function F_{AF} that is represented by the predicate `defendedN/2` and computed via `def_uN/3`.

$$\begin{aligned}\Pi_{defendedN} = \{&def\_uN(N,X,Y) \leftarrow inf(Y), arg\_set(N,X), not\ defeatN(N,Y,X).\\ &def\_uN(N,X,Y) \leftarrow inf(Y), inS(N,Z), defeatN(N,Z,Y),\\ &\qquad\qquad defeatN(N,Y,X).\\ &def\_uN(N,X,Y) \leftarrow succ(Z,Y), not\ defeatN(N,Y,X),\\ &\qquad\qquad def\_uN(N,X,Z).\\ &def\_uN(N,X,Y) \leftarrow succ(Z,Y), not\ def\_uN(N,X,Z), inS(N,V),\\ &\qquad\qquad defeatN(N,V,Y), defeatN(N,Y,X).\\ &defendedN(N,X) \leftarrow sup(Y), def\_uN(N,X,Y).\}\end{aligned}$$

Again each predicate carries the variable N, identifying the current iteration for the predicate `inS/2` which stands for the membership of an argument in the grounded extension.

$$\Pi_{groundN} = \Pi_{cf} \cup \Pi_{<} \cup \Pi_{arg\_set} \cup \Pi_{defendedN} \cup \{inS(N,X) \leftarrow defendedN(N,X).\}$$

The next module, $\Pi_{F\_minus\_range}$, introduces three new predicates. It computes the arguments in $CUT(G)$ which are represented by the predicate `not_in_SplusN/2`. Furthermore, $U \cap S^{\oplus}$ is represented by `u_cap_Splus/2` and $S^{\oplus}$ is represented by the predicate `in_SplusN/2`. Finally, $\Pi_{F\_minus\_range}$ discards all answer-sets where $U \cap S^{\oplus} \neq S$.

$$\begin{aligned}\Pi_{F\_minus\_range} = \{&in\_SplusN(N,X) \leftarrow inS(N,X).\\ &in\_SplusN(N,X) \leftarrow inS(N,Y), defeatN(N,Y,X).\\ &u\_cap\_Splus(N,X) \leftarrow inU(N,X), in\_SplusN(N,X).\\ &\leftarrow u\_cap\_Splus(N,X), not\ inS(N,X).\\ &\leftarrow not\ u\_cap\_Splus(N,X), inS(N,X).\\ &not\_in\_SplusN(N,X) \leftarrow arg\_set(N,X), not\ in\_SplusN(N,X).\}\end{aligned}$$

The module Π_{MR} computes the set of minimal relevant strongly connected components of an argumentation framework G which for our case is the cut of AF G and denoted as $MR(CUT(G)$, computed by $\Pi_{F\_minus\_range}$ at the current iteration N. We recall that for a non-empty argumentation framework $G = \langle A, R\rangle$, a set $S \in SCCS(G)$ is minimal relevant, if S is a minimal element of $\prec$ and $G\downarrow_S$ satisfies the conditions (a)-(c) of Lemma 1.
In the module Π_{MR} the predicate `mr/2` denotes the membership of an argument in the set of minimal relevant SCCs, for an iteration step N. The first two rules of Π_{MR} compute the SCCs, which are represented by predicate `reach/3`, where $reach(N,X,Y)$ states that there is a path in $CUT(G)$ between the arguments X and Y at iteration step N. In the subsequent rules, it is verified whether the criteria of Lemma 1 are met. Condition a) is checked by the rule which contains the predicate `self_defeat/2` as a rule head, which states that an argument is self-defeating. Condition b) is verified by the predicate `nsym/2` , which indicates that an argument does not have a symmetric attack to an argument within the same component. The predicates `reachnotvia/4` and `cyc/4`, are responsible in the cycle detection, which is necessary to check whether condition c) of Lemma 1 holds - the undirected graph, formed by replacing each (directed) pair $\{\langle x,y\rangle, \langle y,x\rangle\}$ of a component with a single undirected edge $\{x,y\}$ has to be acyclic. The predicate `bad/2` is used to mark a component as violating condition c) or a). With the rule containing the predicate $pos\_mr(N,X)$ in the rule head, an argument x is said to be a possible candidate for being in $MR(CUT(G))$ if it is in $CUT(G)$, not self-defeating, acyclic w.r.t. to the definition of condition c) in Lemma 1 and for each $y \in CUT(G)$ it holds that $\langle x,y\rangle \in CUT(G) \Leftrightarrow \langle y,x\rangle \in CUT(G)$. The second to last rule is used to determine whether a strongly connected component is minimal w.r.t. $\prec$. Now the last predicate `mr/2` marks which argument belongs to a minimal relevant strongly connected component by checking whether the predicate `pos_mr/2` holds and `notminimal/2` does not hold.

$$
\begin{aligned}
\Pi_{MR} = \{ & reach(N,X,Y) \leftarrow not\_in\_SplusN(N,X), not\_im\_SplusN(N,Y), defeatN(N,X,Y).\\
& reach(N,X,Y) \leftarrow not\_in\_SplusN(N,X), defeatN(N,X,Z), reach(N,Z,Y),\\
& \qquad X! = Y.\\
& self\_defeat(N,X) \leftarrow not\_in\_SplusN(N,X), defeatN(N,X,X).\\
& nsym(N,X) \leftarrow not\_in\_SplusN(N,X), not\_in\_SplusN(N,Y), defeatN(N,X,Y),\\
& \qquad not\ defeatN(N,Y,X), reach(N,X,Y), reach(N,Y,X), X! = Y.\\
& nsym(N,Y) \leftarrow not\_in\_SplusN(N,X), not\_in\_SplusN(N,Y), defeatN(N,X,Y),\\
& \qquad not\ defeatN(N,Y,X), reach(N,X,Y), reach(N,Y,X), X! = Y.\\
& reachnotvia(N,X,V,Y) \leftarrow defeatN(n,X,Y), not\_im\_SplusN(N,V),\\
& \qquad reach(N,X,Y), reach(N,Y,X), X! = V, Y! = V.\\
& reachnotvia(N,X,V,Y) \leftarrow reachnotvia(N,X,V,Z), reach(N,X,Y),\\
& \qquad reachnotvia(N,Z,V,Y), reach(N,Y,X),\\
& \qquad Z! = V, X! = V, Y! = V.\\
& cyc(N,X,Y,Z) \leftarrow defeatN(N,X,Y), defeatN(N,Y,X),\\
& \qquad defeatN(N,Y,Z), defeatN(N,Z,Y),\\
& \qquad reachnotvia(N,X,Y,Z), X! = Y, Y! = Z, X! = Z.\\
& bad(N,Y) \leftarrow cyc(N,X,U,V), reach(N,X,Y), reach(N,Y,X).\\
& bad(N,Y) \leftarrow self\_defeat(N,X), reach(N,X,Y), reach(N,Y,X).\\
& pos\_mr(N,X) \leftarrow not\_in\_SplusN(N,X), not\ bad(N,X), not\ self\_defeat(N,X),\\
& \qquad not\ nsym(N,X).\\
& notminimal(N,Z) \leftarrow reach(N,X,Y), reach(N,Y,X),\\
& \qquad reach(N,X,Z), not\ reach(N,Z,X).\\
& mr(N,X) \leftarrow pos\_mr(N,X), not\ notminimal(N,X).\}
\end{aligned}
$$

Now the main difference between our novell encoding and this approach is the cycle detection, as can be seen in the following excerpt of our approach:

$$
\begin{aligned}
\Pi_{cyclic} = \{ & udgscc(X,Y,I) \leftarrow sccatt(X,Y,I), sccatt(Y,X,I), X < Y.\\
& cyclic(L,I) \leftarrow \#count\{Z : scc(Z,L,I)\} = V,\\
& \qquad V <= \#count\{X,Y : scc(X,L,I), scc(Y,L,I),\\
& \qquad\qquad udgscc(X,Y,I)\},\\
& \qquad \#int(V), label(L,I).\}
\end{aligned}
$$

As already mentioned the undirected graph in which we are looking for cycles is connected, which makes the cycle detection much simpler than it usually would be. So with the first rule of Π_{cyclic} we transform the pairs of attacks into an undirected edge. Then we continue to count edges and vertices in the transformed graph in order to check whether this graph is acyclic. For performing this step, as can be seen above, we make use of aggregates. This use of aggregates and the utilization of the fact that we only look for cycles in connected acyclic graphs constitute the two major differences which can be identified.

We continue with module $\Pi_{stableN}$ in which parts of Definition 63 are checked. The first rule of the module generates the set $T = U \backslash S^{\oplus}$ and the second rule introduces the predicate `nemptyT/1`, telling whether T is not empty for an iteration N. The third rule introduces `emptyT/1`, that is present for an iteration N if `nemptyT/1` is not the case. With the help of `emptyT/1` and the predicate `not_exists_mr` introduced in rule 5, all guesses where $T \neq \emptyset$ but $MR(CUT(G)) = \emptyset$ are removed by the third to last rule in $\Pi_{stableN}$. With the rule containing the newly introduced predicate `defeated/2` in its rule-head, all the arguments which are defeated by T are determined. The last rule eliminates all the answer-sets where $\neg st_{CUT(G)}(T, \Pi_G)$ is the case.

$$\begin{aligned}\Pi_{stableN} = \{ & t(N,X) \leftarrow inU(N,X), not\ inS(N,X).\\ & nemptyT(N) \leftarrow t(N,X).\\ & emptyT(N) \leftarrow not\ empty(N), arg\_set(N,X).\\ & existsMR(N) \leftarrow mr(N,X), not\_in\_SplusN(N,X).\\ & not\_exists\_mr(N) \leftarrow not\ existsMR(N), not\_in\_Splus(N,X).\\ & true(N) \leftarrow emptyT(N), not\ existsMR(N).\\ & \leftarrow not\_exists\_mr(N), nemptyT(N).\\ & defeated(N,X) \leftarrow mr(N,X), mr(N,Y), t(N,Y), defeatN(N,Y,X).\\ & \leftarrow not\ t(N,X), not\ defeated(N,X), mr(N,X).\}\end{aligned}$$

Predicate`true/1` of module $\Pi_{stableN}$ is used to check whether the iteration realized by module $\Pi_{iterate}$ has to halt. The first rule of $\Pi_{iterate}$ computes the set $(T \cap \Pi_G)^{\oplus}$ and with the second rule, the next iteration is initiated with the predicates `arg_set/2` and `inU/2`, if the iteration is not ihalted by the presence of $true(N)$ for iteration N. The predicate `succ/2` is used to acquire a new domain element M, labeling the next iteration.

$$\begin{aligned}\Pi_{iterate} = \{ & t\_mrOplus(N,Y) \leftarrow t(n,X), mr(N,X), defeatN(N,X,Y).\\ & arg\_set(M,X) \leftarrow not\_in\_SplusN(N,X), not\ mr(N,X),\\ & \qquad not\ t\_mrOplus(N,X), succ(N,M), not\ true(N).\\ & inU(M,X) \leftarrow t(N,X), not\ mr(N,X), succ(N,M), not\ true(N).\}\end{aligned}$$

The last module Π_{grd*} combines all the modules, and these all constitute the encoding.

$$\Pi_{grd*} = \Pi_{groundN} \cup \Pi_{F\_minus\_range} \cup \Pi_{MR} \cup \Pi_{stableN} \cup \Pi_{iterate}.$$

In our results section we provide a detailed performance comparison and investigate the major differences between our encoding and the encoding of W. Dvořák et al.

CHAPTER 5

Experimental Evaluation

5.1 Test Set

For testing our encoding and comparing its performance to the previous encoding by W. Dvořák et al. we fully relied on a subset of the test set provided by W. Dvořák et al. in [22]. The argumentation frameworks of this test set have all been randomly generated with an argument size between 20 and 110 arguments. Two parametrized methods have been used for generating them.

1. Arbitrary AF $G = \langle A, R \rangle$: For any pair $a, b \in A$ the attack (a, b) is inserted into G with a given probability p.

2. Grid-structured AF $G = \langle A, R \rangle$: Every argument is arranged in a $(n \times m)$ grid, i.e. connected to its neighbors via attacks. The actual values for n and m depend on the total number of arguments; more precisely we let n be either 5, 15, or 25 and m is then chosen accordingly. For every neighbor b of a either the mutual attack $\{(a, b), (b, a)\}$ is added with a probability p, or a single attack is inserted into G. The direction of the single attack is chosen randomly with equal chance. For the grids we consider two neighborhoods, namely 4-neighborhood and 8-neighborhood. The former connects all arguments horizontally and vertically, the latter is connecting them also diagonally.

For both methods, the probability p has been 0.1 and 0.4.

5.2 Test Environment

The tests were executed on an openSUSE based machine with eight Intel Xeon processors (2.33 GHz) and 49 GB memory. The DLV version we used was the following release*DLV [build BEN/Dec 21 2011 gcc 4.6.1]*
The clasp version we used is *clasp 2.0.6 (64-bit)*
The claspD version we used is *claspD - Version: 1.1.2*

The gringo version we used is *gringo 3.0.4*

5.3 The Candidate Encodings

In this section we briefly recapitulate the encodings we compared in our experiments. First of all, there is our encoding $\Pi_{veruegr\_opt}$ and its initial version without the optimization $\Pi_{veruegr}$, which, as already mentioned, are both based on the verification algorithm 4.1, proposed by Baroni et al.
Then there are Π_{grd*_metasp} and Π'_{grd*_metasp} two metasp encodings of resolution-based grounded semantics that we already described, which were proposed by W. Dvořák et al. in [22] and are based on the metasp approach, proposed by M. Gebser et al. [32].
Finally, there is the encoding Π_{grd*}, proposed by W. Dvořák et al. in their article [22]. For the encodings $\Pi_{veruegr}$, $\Pi_{veruegr\_opt}$ and Π_{grd*} we compare CLASP and DLV. In order for $\Pi_{veruegr}$ and $\Pi_{veruegr\_opt}$ to be compatible with CLASP we had to slightly modify the rules containing aggregates.

$$\begin{aligned}\Pi_{cyclic} = \{&udgscc(X,Y,I) \leftarrow sccatt(X,Y,I), sccatt(Y,X,I), X < Y.\\ &cyclic(L,I) \leftarrow \#count\{Z : scc(Z,L,I)\} = V,\\ &\qquad V <= \#count\{X,Y : scc(X,L,I), scc(Y,L,I),\\ &\qquad\qquad udgscc(X,Y,I)\},\\ &\qquad \#int(V), label(L,I).\}\end{aligned}$$

Since the only aggregates we use appear in Π_{cyclic}, we only have to adapt this part of our encodings to make them CLASP-compatible, resulting in a new module $\Pi_{cyclic'}$. For further details on the differences between CLASP and DLV, please refer to our background section.

$$\begin{aligned}\Pi_{cyclic'} = \{&udgscc(X,Y,I) \leftarrow sccatt(X,Y,I), sccatt(Y,X,I), X < Y.\\ &cyclic(L,I) \leftarrow V = \#count\{scc(Z,L,I) : arg(Z)\},\\ &\qquad W = \#count\{udgscc(X,Y,L,I) : arg(X) : arg(Y) : d(z)\},\\ &\qquad V <= W, label(L,I).\}\end{aligned}$$

In order to differentiate between the DLV and the CLASP version of the respective encodings in the benchmark figures, we altered the names of the encodings as described in Table 5.1.

5.4 Results

To obtain results which are easily comparable to the results provided by W. Dvořák et al. in [22], we also stick to calculating the average computation time and the timeout percentage. Since we are also interested in the differences with respect to the time it takes for grounding the various encodings with the input instances, one of our benchmarks, focused on the grounding time percentage.

encoding	CLASP version	DLV version
Π_{grd*}	$\Pi_{grd*(lp)}$	$\Pi_{grd*(dl)}$
$\Pi_{veruegr}$	$\Pi_{veruegr(lp)}$	$\Pi_{veruegr(dl)}$
$\Pi_{veruegr\_opt}$	$\Pi_{veruegr\_opt(lp)}$	$\Pi_{veruegr\_opt(dl)}$

Table 5.1: Names of the encodings in the benchmarks.

Initial benchmarks

The first benchmark we performed is a comparison between the encodings $\Pi_{veruegr}$, Π_{grd*}, Π_{grd*_metasp} and Π'_{grd*_metasp}. Whereas for $\Pi_{veruegr}$ and Π_{grd*}, we compared the CLASP and DLV version of the encodings. As can be seen in Figure 5.1, we here are interested in the average computation time as well as the timeout percentage achieved by the different encodings.

In Figure 5.1 one can see that Π'_{grd*_metasp} showed the best performance with respect to computation time, whereas for the smaller instances $\Pi_{veruegr}$ in combination with CLASP performed slightly better. When we take a look at the timeouts we can clearly see that the best performance was achieved by $\Pi_{veruegr}$ when executed with CLASP. We also noticed that $\Pi_{veruegr}$ performed much better than Π_{grd*}. Another trend which can be seen is that the encodings in general performed better when executed with CLASP, but we will discuss this in more detail later.

Performance Gain, Achieved by our Encoding

In this section we take a deeper look into performance differences between $\Pi_{veruegr\_opt}$, Π_{grd*} and the metasp encodings. We wanted to better understand the reasons for the observerd performance gap between $\Pi_{veruegr\_opt}$ and the other encodings for the resolution-based grounded semantics, since this knowledge might be of use in other semantics encodings, in abstract argumentation as well.
The first apparent difference between Π_{grd*} and $\Pi_{veruegr\_opt}$ is the cycle detection. For a more detailed analysis of the question which differences bear the most significant gain, we omit our symmetry breaking optimization and use our initial encoding $\Pi_{veruegr}$. Since we want to verify how much of an effect the cycle detection (based on aggregates) had on computation time performance, we constructed a second version Π_{grd*_cx} from Π_{grd*}. In Π_{grd*_cx} we exchanged the rules responsible for the cycle detection which is required for checking whether $MR(CUT(G))$ satisfies condition (c) of Lemma 1. The exchange of rules only affects the module Π_{MR} of Π_{grd*}, which in Π_{grd*_cx} is substituted by Π_{MR_CX}.

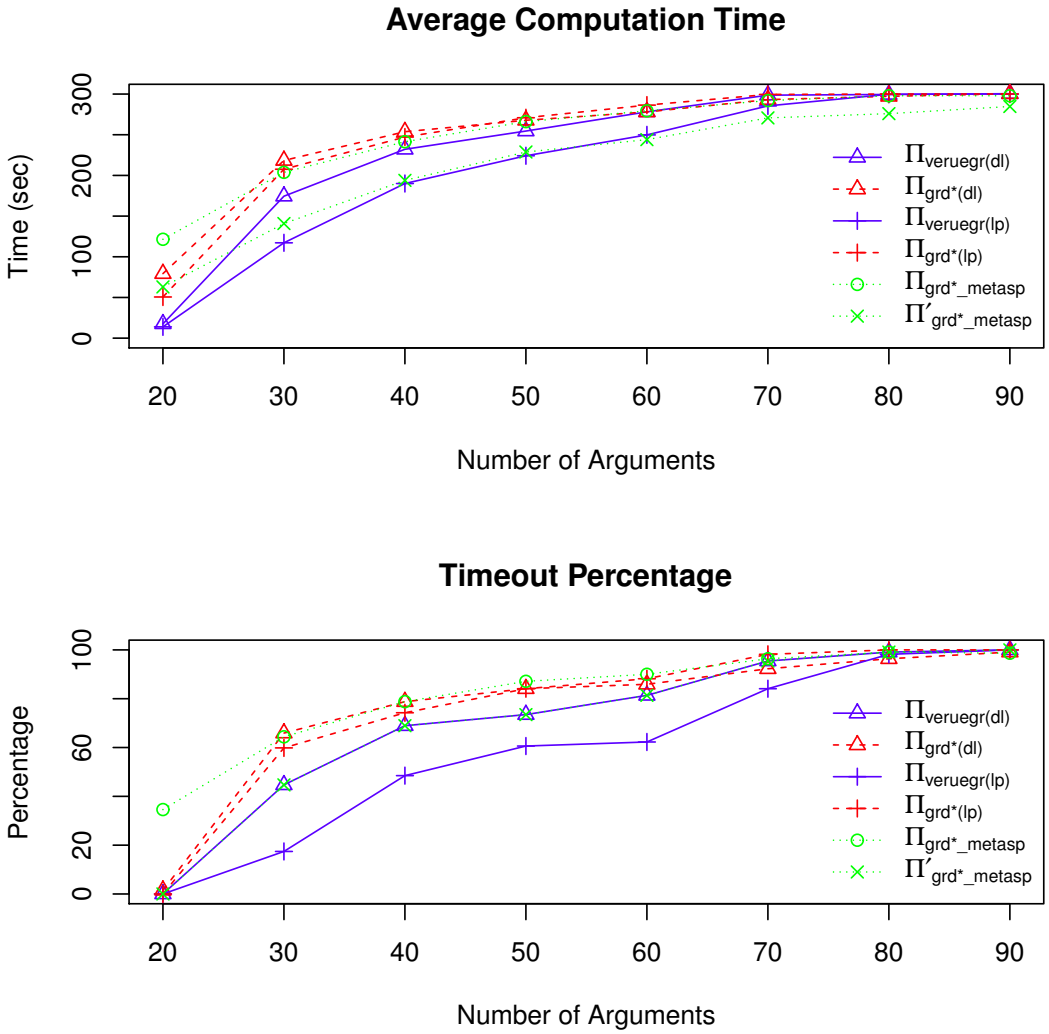

Figure 5.1: Computation time and timeout statistics of $\Pi_{veruegr}$, Π_{grd*}, Π_{grd*_metasp} and Π'_{grd*_metasp}.

$$
\begin{aligned}
\Pi_{MR} = \{ & \\
& \vdots \\
& reachnotvia(N,X,V,Y) \leftarrow defeatN(n,X,Y), not\_im\_SplusN(N,V), \\
& \qquad\qquad reach(N,X,Y), reach(N,Y,X), X! = V, Y! = V. \\
& reachnotvia(N,X,V,Y) \leftarrow reachnotvia(N,X,V,Z), reach(N,X,Y), \\
& \qquad\qquad reachnotvia(N,Z,V,Y), reach(N,Y,X), \\
& \qquad\qquad Z! = V, X! = V, Y! = V. \\
& cyc(N,X,Y,Z) \leftarrow defeatN(N,X,Y), defeatN(N,Y,X), \\
& \qquad\qquad defeatN(N,Y,Z), defeatN(N,Z,Y), \\
& \qquad\qquad reachnotvia(N,X,Y,Z), X! = Y, Y! = Z, X! = Z. \\
& bad(N,Y) \leftarrow cyc(N,X,U,V), reach(N,X,Y), reach(N,Y,X). \\
& \vdots \\
\} &
\end{aligned}
$$

For readability reasons we omit the rules responsible for the cycle detection, which were not affected by the exchange. It can easily be seen what adaptions we made and where the aggregates are used. For a more detailed explanation, we refer to the section which describes our implementation, where we extensively describe how the cycle detection operates with the help of aggregates.

$$
\begin{aligned}
\Pi_{MR\_CX} = \{ & \\
& \vdots \\
& bigger(N,X,Y) \leftarrow reach(N,X,Y), reach(N,Y,Z), X < Y, Y < Z. \\
& label(N,Y) \leftarrow reach(N,X,Y), not\ \ bigger(N,X,Y), X < Y. \\
& scc(N,X,L) \leftarrow reach(N,X,L), label(N,L). \\
& udgscc(N,X,Y) \leftarrow defeatN(N,X,Y), defeatN(N,Y,X), X < Y. \\
& cyc(N,L) \leftarrow \#count\{Z : scc(N,Z,L)\} = V, \\
& \qquad\qquad V <= \#count\{P,Q : scc(N,P,L), scc(N,Q,L), \\
& \qquad\qquad\qquad\qquad udgscc(N,P,Q)\}, \\
& \qquad\qquad \#int(V), label(N,L). \\
& bad(N,Y) \leftarrow cyc(N,L), scc(N,Y,L). \\
& \vdots \\
\} &
\end{aligned}
$$

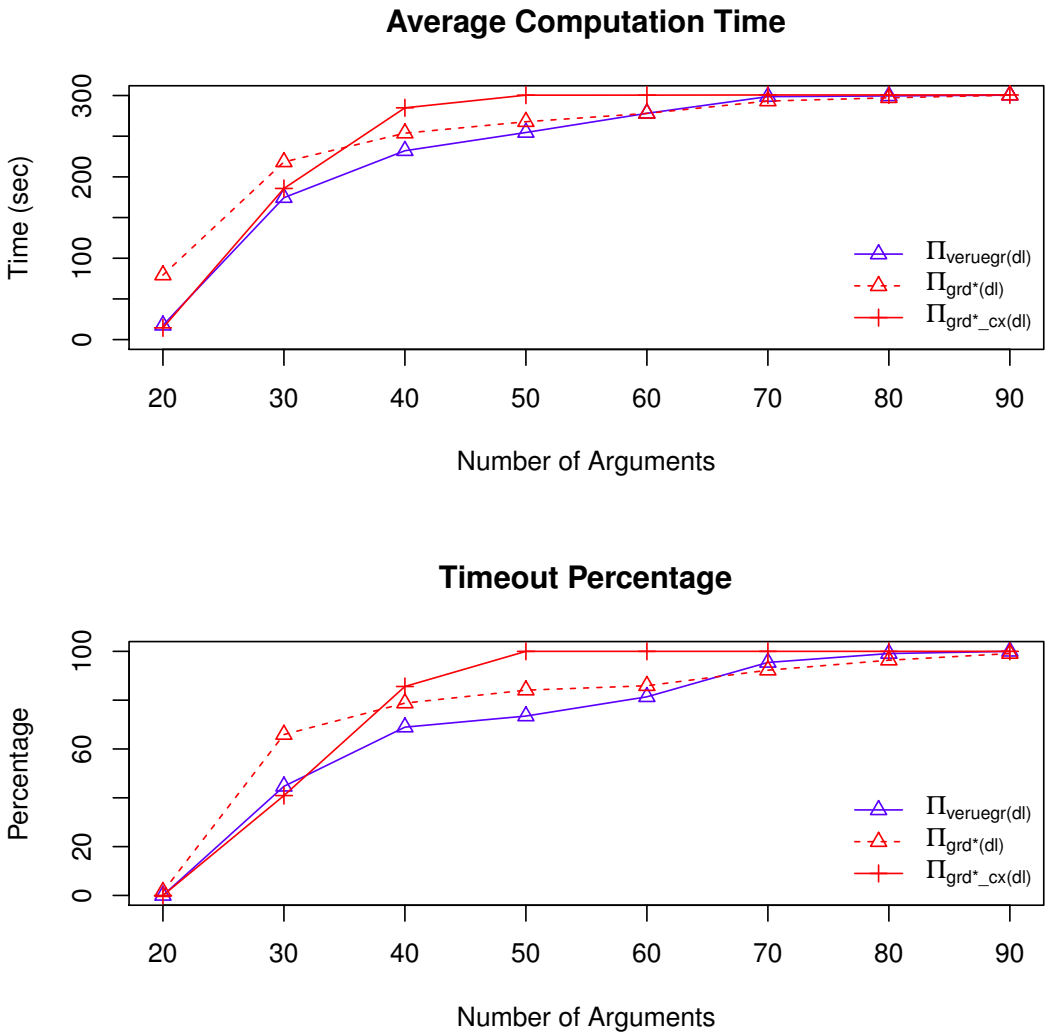

Figure 5.2: Computation time and timeout statistics of Π_{grd*}, Π_{grd*_cx} and $\Pi_{veruegr}$

In Figure 5.2 we provide benchmarks for comparing the performance of Π_{grd*}, Π_{grd*_cx} and $\Pi_{veruegr}$. We entirely relied on DLV for this set of test runs. Again, we were interested in computation time and timeout percentage.

From the results depicted in Figure 5.2, we can see that, for small instances, the exchange of the cycle detection resulted in the expected performance gain of Π_{grd*_cx}. Why this is not the case for larger instances remains a subject of further investigation in future work.

In Figure 5.3 we depict the results from showing us how our encoding with the symmetry breaking optimization $\Pi_{veruegr\_opt}$ performs, compared to our initial encoding.

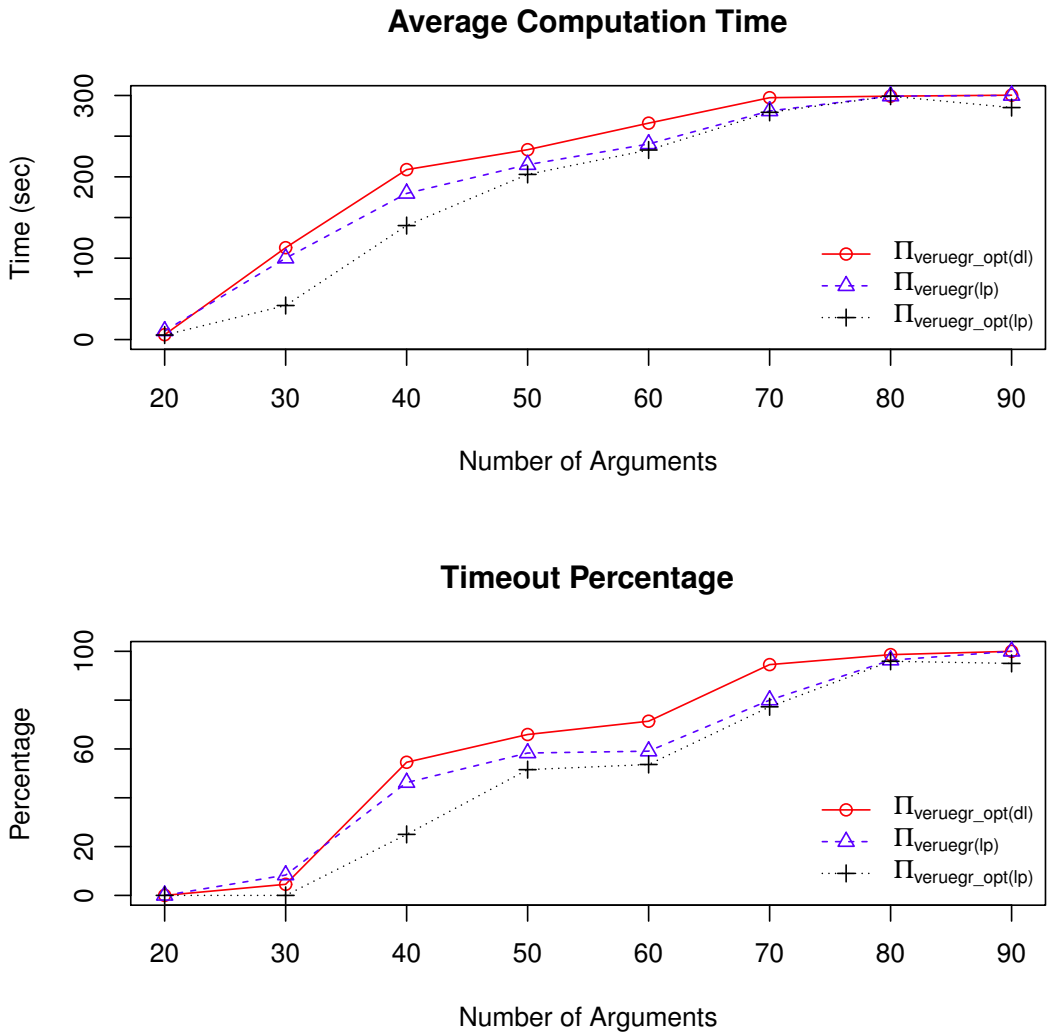

Figure 5.3: Symmetry breaking optimization.

We can see that this small optimization comes with a significant performance gain. Likewise one can notice that the encodings so far perform better when executed by CLASP. This leads to our decision to create more detailed statistics on the performance difference between CLASP and DLV. In Figure 5.4 we depict separate computation time and timeout statistics for the different kinds of AFs. The first two graphs show a comparison of the encodings for the arbitrarily generated AFs. Here it can be seen that each encoding regardless of whether it was run in combination with CLASP or DLV, performed poorly and quickly timed out. Yet one can notice, that CLASP always performed slightly better. This observation can be seen for the 4-grid and 8-grid AFs as well, whereby the performance was better when compared to the arbitrarily generated instances. There is only one exception in the statistics for the 4-grid AFs, where DLV performed better than CLASP for larger instances in combination with the encoding Π_{grd*}. With respect to this discovery, the remaining statistics compare the metasp encodings to the CLASP version of Π_{grd*} and $\Pi_{veruegr\_opt}$.

In Figure 5.5, we present a boxplot for the encodings: $\Pi_{veruegr\_opt}$, $\Pi_{veruegr}$, Π_{grd*}, Π_{grd*_metasp} and Π'_{grd*_metasp}. We chose to generate this representation of our benchmarks to get a proper visualization of the spread of the results in various measurements. We compared the computation time performance of each encoding with respect to the individual kinds of AFs. We notice that with our novel encoding of the resolution-based grounded semantics, $\Pi_{veruegr\_opt}$, we achieve the best overall performance for the given test instances of AFs. As expected, the application of symmetry breaking resulted in equally improved performance for all the test instances. Now we take a look at the metasp encodings. One can notice that the overall performance of them is worse than the performance of $\Pi_{veruegr\_opt}$, but for a great number of outliers, the computation time is considerably small. So we surmise that there is a kind of argumentation frameworks that is equally distributed across the different kinds of test instances we created on which the metasp encodings have an superior performance but the deeper analysis of this fact is not within the scope of this thesis.
When dealing with answer-set programming an important index is the grounding time of answer-set programs. Hence in Figure 5.6, we provide box plots depicting the average grounding time percentage of the different encodings. One can clearly see that with our encoding symmetry breaking only slightly affected the grounding time percentage. Also, one can observe that Π_{grd*} and our novel encoding share almost the same grounding time ratio as opposed to the metasp encodings. Their grounding time percentage is remarkably short, which means that the solver consumes the most computation time.

The last statistics we present are a detailed comparison of the encodings Π_{grd*}, Π_{grd*_metasp}, Π'_{grd*_metasp} and $\Pi_{veruegr\_opt}$. Like in the previous statistics we provide a comparison of those encodings in the Figures 5.7, 5.8 and 5.9 with respect to average computation time and timeout percentage. Each figure deals with one particular type of instances. Again, the types of instances we compare are arbitrary, 4-grid and 8-grid AFs. On the arbitrary AFs and for larger instances, the metasp encodings show the best performance, which are clearly shown by the average computation times as well in the timout percentage. For smaller instances our novel encoding $\Pi_{veruegr\_opt}$ performs better. When analyzing the results for the 4-grid AFs, we can

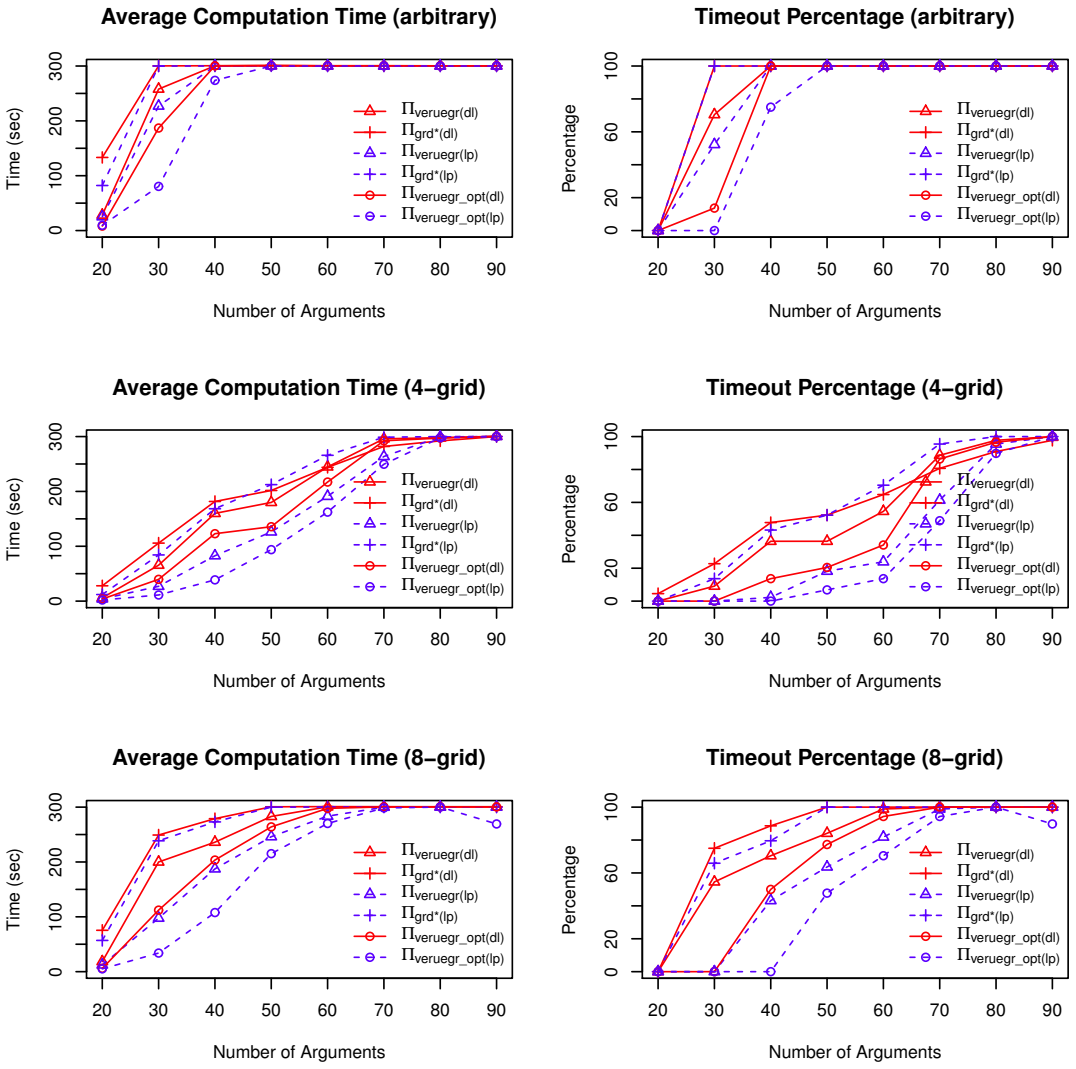

Figure 5.4: Detailed statistics for comparing DLV with GRINGO/CLASP.

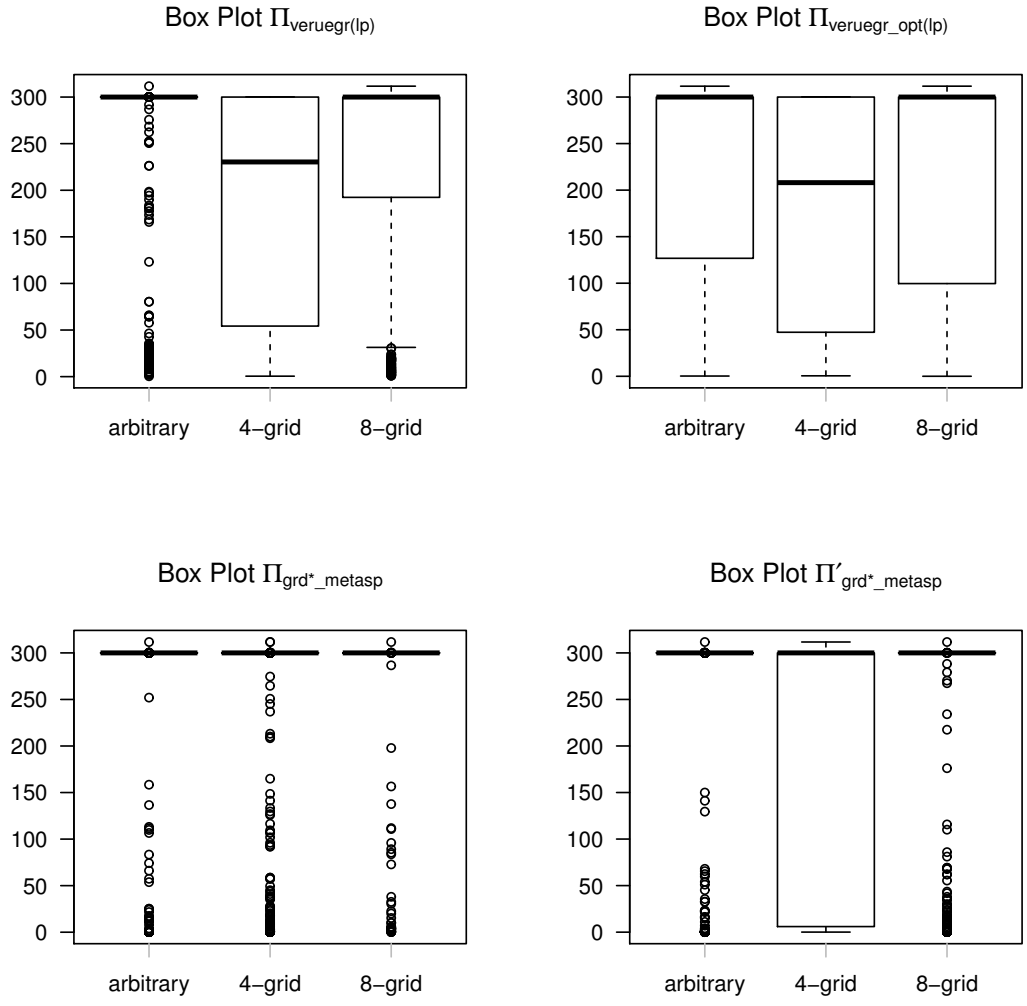

Figure 5.5: Boxplot of computation times.

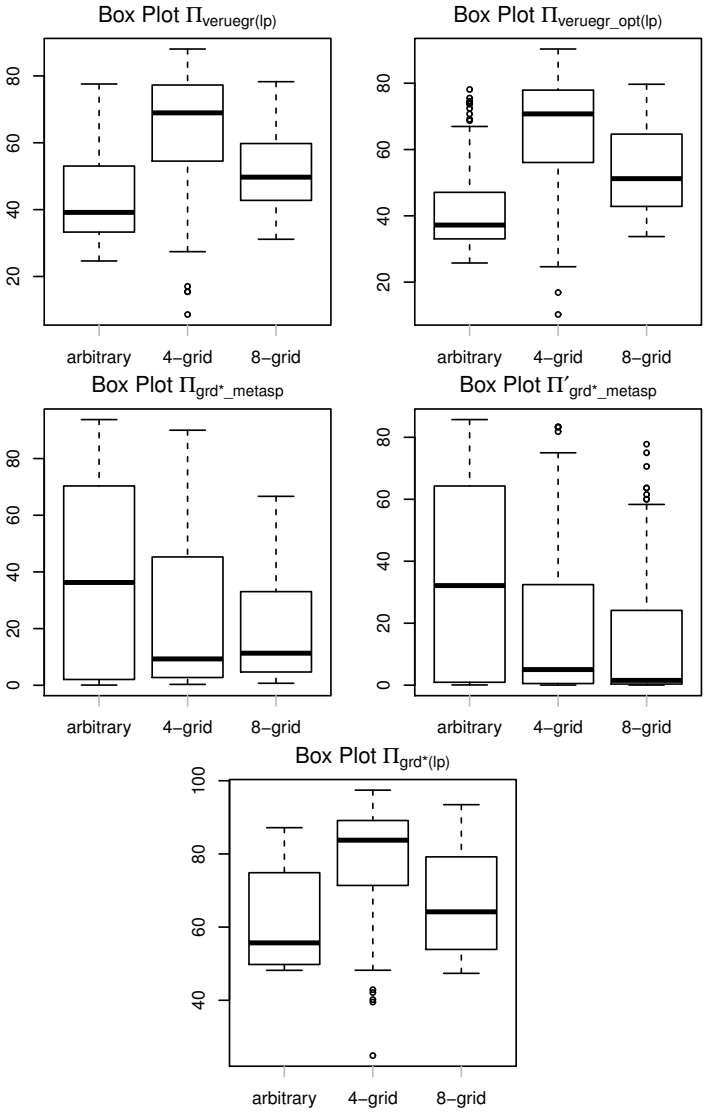

Figure 5.6: Average grounding time percentage.

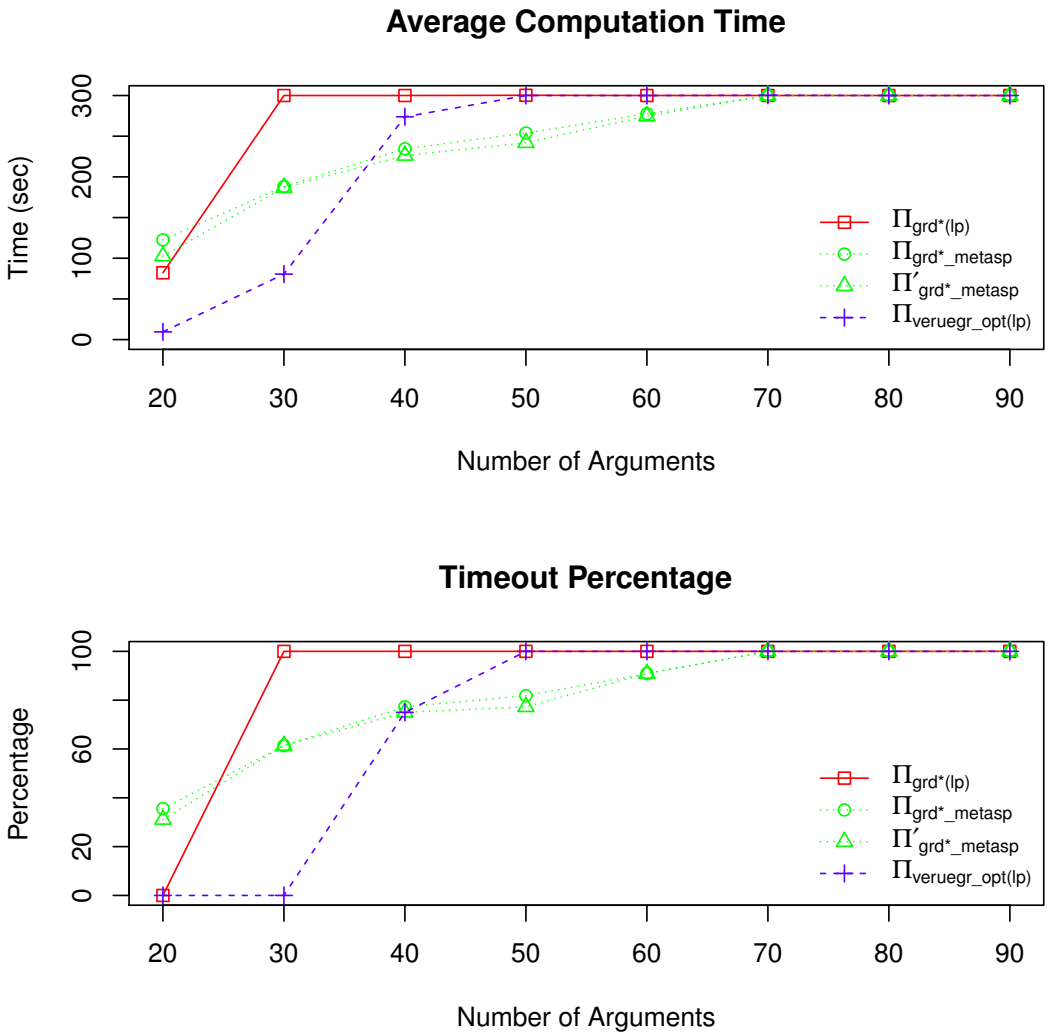

Figure 5.7: Average computation time and timeout percentage (arbitrary).

see that our novel encoding performs significantly better for the smaller instances up to a size between 70 and 80 arguments. Here for the largest instances of our benchmark, Π'_{grd*_metasp} shows the best performance with respect to average computation time. Although we point out that Π'_{grd*_metasp} also shows a much higher timeout percentage throughout the whole range of argumentation framework sizes. And recalling the box plot comparison of the overall computation time, we surmise the good average computation time is due to a great number of outliers with remarkably short computation time. On the 8-grid AFs our encoding did yield the best results with a timeout percentage of 0 up to instances of argument size 40. The average computation time as well as the timeout percentage always remained below that of the other encodings.

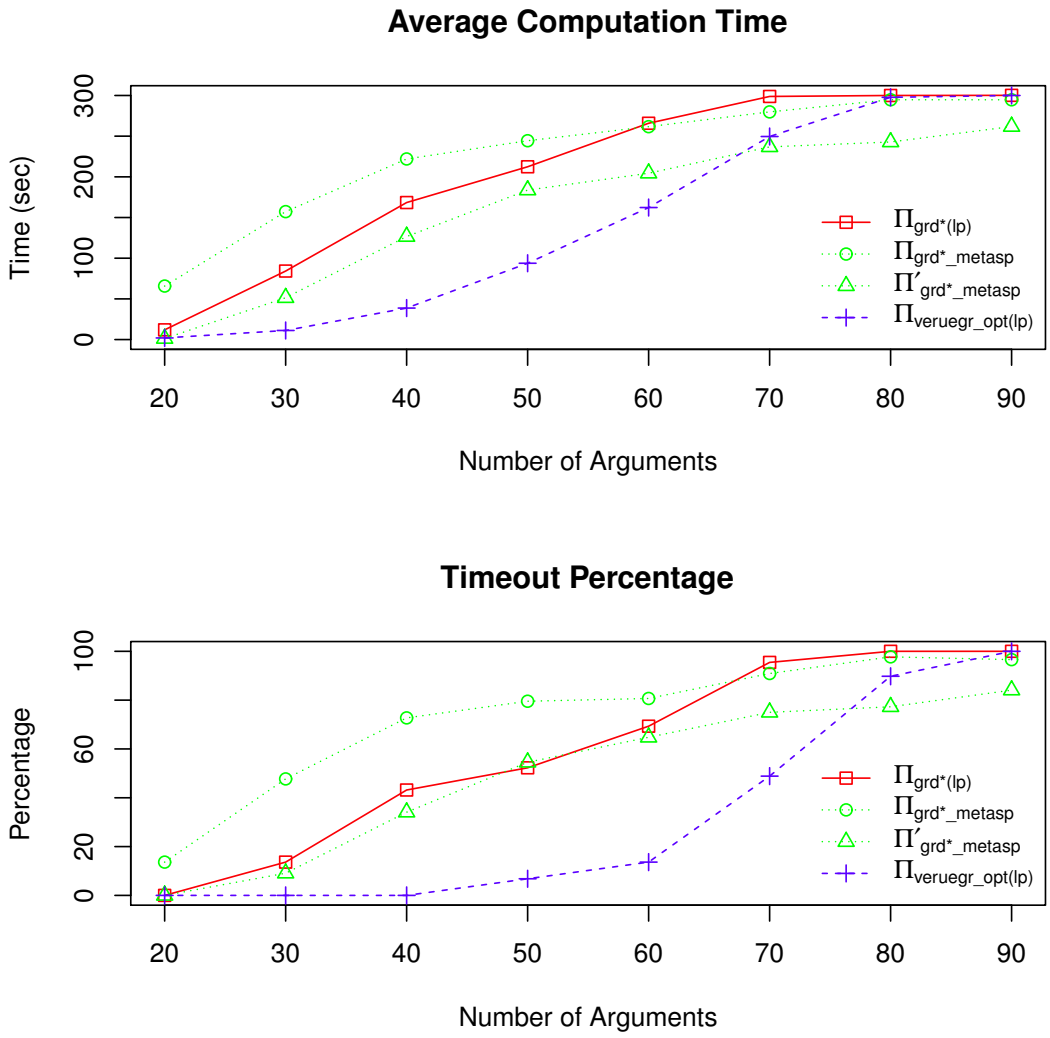

Figure 5.8: Average computation time and timeout percentage (4-grid).

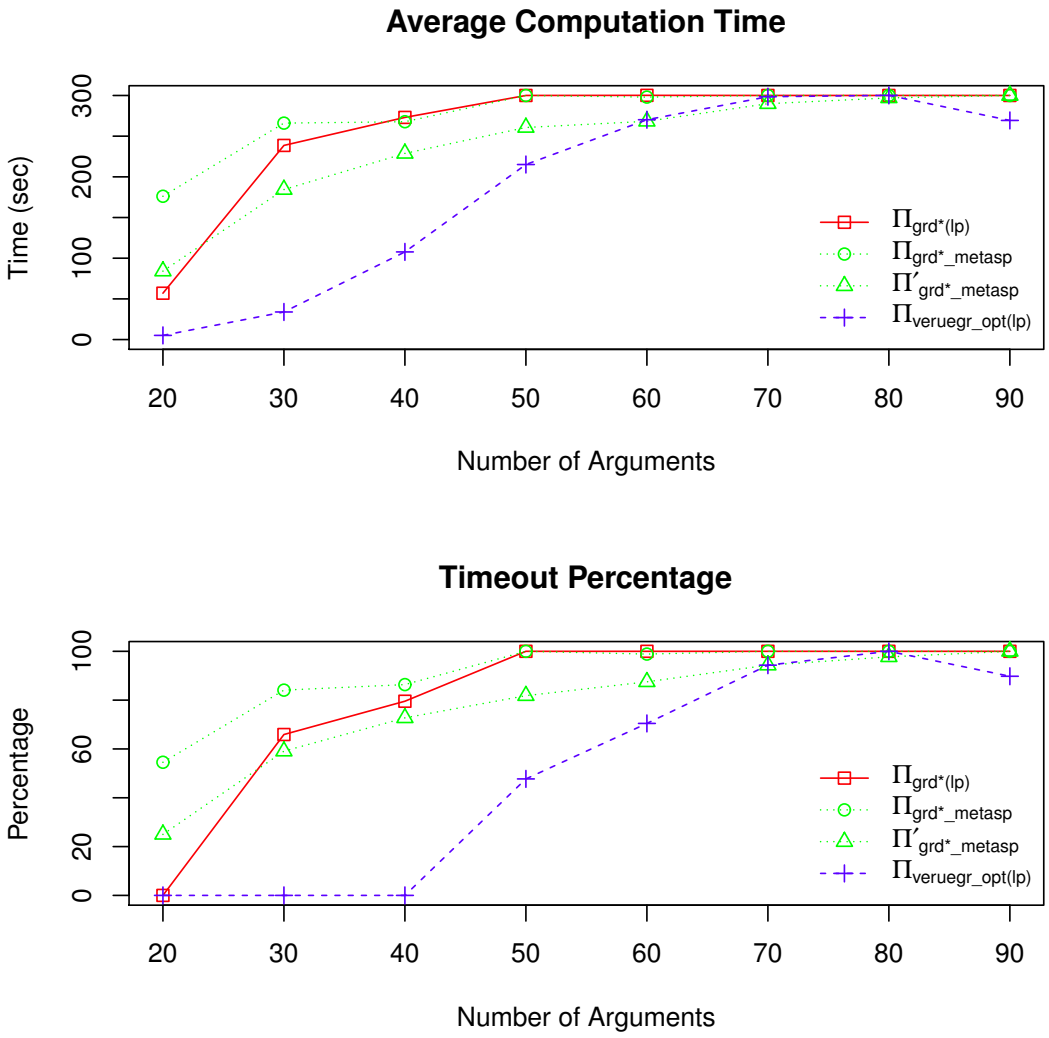

Figure 5.9: Average computation time and timeout percentage (8-grid).

5.5 Observations

In this section we summarize the different observations which could be made with respect to our benchmarks. The first observation we would like to mention is the performance gap between CLASP and DLV. Then we present the observations we made when comparing the average computation time and timeout percentage for $\Pi_{veruegr\_opt}$, Π_{grd*} and the metasp encodings. Finally, we want to mention the difference between metasp and the remaining encodings of the resolution-based grounded semantics with respect to the grounding time percentage.

DLV vs. CLASP

In the initial benchmarks we performed we noticed a slight performance advantage with CLASP. In the statistics shown in Figure 5.4 we performed a detailed comparison, and came to the conclusion that CLASP in general yielded a shorter computation time for the different encodings. We observed for each encoding executed with CLASP and GRINGO, regardless which type of argumentation frameworks were used, resulted in a performance gain. There was only one exception, which we encountered in the statistics for the 4-grid AFs. Here DLV performed slightly better than CLASP for larger instances in combination with Π_{grd*}.

Comparison to other Encodings

Based on the statistics for the detailed comparison of the encodings Π_{grd*}, Π_{grd*_metasp}, Π'_{grd*_metasp} and $\Pi_{veruegr\_opt}$, we made the observation that our novel encoding, $\Pi_{veruegr\_opt}$, outperformed all the other encodings of the resolution-based grounded semantics for the smaller instances of AFs with an argument size up to 40. For arbitrary and 4-grid AFs, the metasp encodings performed slightly better on the larger instances. But based on the statistics, shown in the box plots in Figure 5.5, we surmise that this performance advantage of Π_{grd*_metasp}, Π'_{grd*_metasp} for large instances is due to a great number of outliers with a relatively low computation time. We also confirmed in our results, that for small instances our aggregate-based cycle detection resulted in a performance gain.

Grounding Time Percentage

When dealing with answer-set programming an important index is the grounding time of answer-set programs. Hence, in Figure 5.6 we provide box plots depicting the average grounding time percentage of each encoding. We observed that Π_{grd*} and our novel encoding share almost the same grounding time ratio as opposed to the metasp encodings. Their grounding time percentage is remarkable small. From that we conclude that the metasp encodings possess a great advantage with respect to grounding time, but considering the average computation times as a whole, in comparison to our encoding the solver had a much larger computation time. We noticed that the metasp encodings seem to shift the workload from the grounder to the solver.

CHAPTER 6

Summary and Future Work

With this thesis we documented our development of a Guess & Check answer set program, based on the verification algorithm presented in [6] for computing the extensions of the resolution-based grounded semantics. We performed an elaborate performance comparison with various encodings of the resolution-based grounded semantics in the realm of answer-set programming. We compared our solution to Π_{grd*_metasp} and Π'_{grd*_metasp}, two meta ASP encodings of the resolution-based grounded semantics, which were proposed by by W. Dvořák et al. in [22] and are based on the metasp approach, proposed by M. Gebser et al. [32]. We too compared our realization to the encoding Π_{grd*}, proposed by W. Dvořák et al. in their article [22], which also is based on the verification algorithm presented in [6] and follows the Guess & Check paradigm. During the process of development we discovered the positive effects of an optimization technique which is known as symmetry breaking within the field of answer-set programming. We observed that for existing encodings of the resolution-based grounded semantics CLASP together with GRINGO performed significantly better than DLV. A deeper analysis of why this is the case remains subject to future work. We furthermore came to the conclusion that our novel encoding performed remarkably well compared to existing solutions. For smaller instances its performance was unmatched, while for larger instances the performance advantage depended on the type of argumentation framework.

Bibliography

[1] Leila Amgoud and Caroline Devred. Argumentation frameworks as constraint satisfaction problems. In *Proceedings of the 5th international conference on Scalable uncertainty management*, SUM'11, pages 110–122, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Christian Anger, Kathrin Konczak, Thomas Linke, and Torsten Schaub. A glimpse of answer set programming. *KI*, 19(1):12–, 2005.

[3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.

[4] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. Review: an introduction to argumentation semantics. *The Knowledge Engineering Review*, 26(4):365–410, December 2011.

[5] Pietro Baroni, Paul E. Dunne, and Massimilano Giacomin. On the resolution-based family of abstract argumentation semantics and its grounded instance. *Artificial Intelligence*, 175:791–813, March 2011.

[6] Pietro Baroni and Massimiliano Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence*, 171:675–700, July 2007.

[7] Pietro Baroni and Massimiliano Giacomin. Semantics of abstract argument systems. In Guillermo Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 25–44. Springer US, 2009.

[8] Pietro Baroni, Massimiliano Giacomin, and Giovanni Guida. SCC-recursiveness: a general schema for argumentation semantics. *Artificial Intelligence*, 168(1):162–210, October 2005.

[9] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme - Grundlagen, Algorithmen, Anwendungen*. Vieweg+Teubner Verlag, 4., verbesserte Auflage, 2008.

[10] Philippe Besnard and Anthony Hunter. Argumentation based on classical logic. In Guillermo Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 133–152. Springer US, 2009.

[11] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. dlv - user manual. `http://www.dlvsystem.com`, 10 2012.

[12] Stefano Bistarelli and Francesco Santini. Conarg: A constraint-based computational framework for argumentation systems. In *Proceedings of the IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011*, pages 605–612, 2011.

[13] Stefano Bistarelli and Francesco Santini. Modeling and solving afs with a constraint-based tool: conarg. In *Proceedings of the First international conference on Theory and Applications of Formal Argumentation*, TAFA 11, pages 99–116, Berlin, Heidelberg, 2012. Springer-Verlag.

[14] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[15] Martin Caminada. On the issue of reinstatement in argumentation. In *Proceedings of the 10th European conference on Logics in Artificial Intelligence*, JELIA'06, pages 111–123, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] Martin Caminada. Semi-stable semantics. In *Proceedings of the 2006 conference on Computational Models of Argument*, pages 121–130, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

[17] Sylvie Coste-Marquis, Caroline Devred, and Pierre Marquis. Prudent semantics for argumentation frameworks. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '05, pages 568–572, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Sylvie Coste-Marquis, Caroline Devred, and Pierre Marquis. Symmetric argumentation frameworks. In *Proceedings of the 8th European conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, ECSQARU'05, pages 317–328, Berlin, Heidelberg, 2005. Springer-Verlag.

[19] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artifical Intelligence*, 77(2):321–357, September 1995.

[20] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. A dialectic procedure for sceptical, assumption-based argumentation. In *Proceedings of the 2006 conference on Computational Models of Argument*, pages 145–156, Amsterdam, The Netherlands, 2006. IOS Press.

[21] Paul E. Dunne and Michael Wooldridge. Complexity of abstract argumentation. In Guillermo Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 85–104. Springer US, 2009.

[22] Wolfgang Dvořák, Sarah Alice Gaggl, Johannes Peter Wallner, and Stefan Woltran. Making use of advances in answer-set programming for abstract argumentation systems. *CoRR*, abs/1108.4942, 2011.

[23] Wolfgang Dvořák and Stefan Woltran. On the intertranslatability of argumentation semantics. *Journal of Artificial Intelligence Research*, 41(2):445–475, May 2011.

[24] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. *Argument & Computation*, 1(2):147–177, 2010.

[25] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Aspartix: Implementing argumentation frameworks using answer-set programming. In *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pages 734–738, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.

[27] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer Berlin Heidelberg, 2009.

[28] Wolfgang Faber and Stefan Woltran. Manifold answer-set programs for meta-reasoning. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 115–128, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] Alejandro J. García and Guillermo R. Simari. Defeasible logic programming: an argumentative approach. *Theory and Practice of Logic Programming*, 4(2):95–138, January 2004.

[30] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Marius T. Schneider. Potassco: The Potsdam answer set solving collection. `http://potassco.sourceforge.net`, 12 2012.

[31] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo and iclingo, 2008.

[32] Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(Special Issue 4-5):821–839, 2011.

[33] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, April 2011.

[34] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Clasp: a conflict-driven answer set solver. In *Proceedings of the 9th international conference on Logic programming and nonmonotonic reasoning*, LPNMR'07, pages 260–265, Berlin, Heidelberg, 2007. Springer-Verlag.

[35] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo: a new grounder for answer set programming. In *Proceedings of the 9th international conference on Logic programming and nonmonotonic reasoning*, LPNMR'07, pages 266–271, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[37] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

[38] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:2002, 2002.

[39] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 3*, pages 1594–1597. AAAI Press, 2008.

[40] David G. Mitchell. A sat solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[41] Sanjay Modgil. Hierarchical argumentation. In Michael Fisher, Wiebe van der Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 of *Lecture Notes in Computer Science*, pages 319–332. Springer, 2006.

[42] Juan C. Nieves, Ulises Cortés, and Mauricio Osorio. Preferred extensions as stable models. *Theory and Practice of Logic Programming*, 8(4):527–543, 2008.

[43] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[44] Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK, 2003.

[45] Iyad Rahwan and Guillermo R. Simari, editors. *Argumentation in Artificial Intelligence*. Springer-Verlag, Berlin, 2009.

[46] Raymond Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum, New York / London, 1978.

[47] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81 – 132, 1980. Special Issue on Non-Monotonic Logic.

[48] Guillermo R. Simari and Ronald P. Loui. A mathematical treatment of defeasible reasoning and its implementation. *Artificial Intelligence*, 53(2-3):125–157, February 1992.

[49] Patrik Simons, Ilkka Niemelá, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, June 2002.

[50] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1 – 22, 1976.

[51] Tommi Syrjánen. Lparse 1.0 user's manual.

[52] Tommi Syrjánen and Ilkka Niemelá. The smodels system. In *Logic Programming and Nonmotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer Berlin Heidelberg, 2001.

[53] Francesca Toni and Marek Sergot. Argumentation and answer set programming. In Marcello Balduccini and TranCao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 164–180. Springer Berlin Heidelberg, 2011.

[54] Toshiko Wakaki and Katsumi Nitta. New frontiers in artificial intelligence. In Hiromitsu Hattori, Takahiro Kawamura, Tsuyoshi Idé, Makoto Yokoo, and Yohei Murakami, editors, *Lecture Notes in Computer Science*, pages 254–269. Springer-Verlag, Berlin, Heidelberg, 2009.