Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TU WIEN

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Abstract Dialectical Frameworks:

Properties, Complexity, and Implementation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Stefan Ellmauthaler, BSc

Matrikelnummer 0525923

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Proj.Ass. Dipl.-Ing. Johannes Peter Wallner, BSc

Wien, 10.08.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Abstract Dialectical Frameworks:

Properties, Complexity, and Implementation

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Stefan Ellmauthaler, BSc
Registration Number 0525923

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Proj.Ass. Dipl.-Ing. Johannes Peter Wallner, BSc

Vienna, 10.08.2012 ____________________ (Signature of Author) ____________________ (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stefan Ellmauthaler, BSc
Quellenstraße 67/26, 1100 Wien

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

At first I want to thank my advisor *Stefan* for the introduction to abstract argumentation, the option to graduate under his advice, and the great support and helpful tips during the last months. In addition I want to thank my co-advisor *Johannes* for his invaluable feedback during the creative process of the thesis and the hints when I got stuck in some formal proof-details. Furthermore my thanks go to *Gerd* for the opportunities he gave me and his invitation to Leipzig, where I have got different views and new suggestions to the topics of the thesis.

I am also grateful to my parents *Erhard* and *Ilse*, who supported me during my whole study and made it possible to concentrate as much as possible on the graduation. I also want to thank my life companion *Nadine* for her love and her constant efforts to let me consume some of my spare time for other things than my study.

Last but not least, I want to say *"thank you"* to all my friends which have spent free time together with me and who have given me more than one suggestion for formal concerns regarding the finishing of the master study.

Stefan

Abstract

Over the last two decades the interest for *Abstract Argumentation* steadily raised in the field of Artificial Intelligence. The concept of Dung's *Argumentation Frameworks (AFs)*, where arguments and their relations are represented in a directed graph-structure, is a well-known, simple, and powerful concept. This framework is used to find acceptable sets of arguments, which have specific properties (e.g. being conflict free), defined by several semantics.

Recently *Abstract Dialectical Frameworks (ADFs)* were introduced, a generalization of Dung's approach, to overcome the limitation of attack-relations being the only type of native relations. To reach this goal, in addition to the relations, total functions are used to decide the acceptance of an argument. These functions are so called acceptance conditions. Due to the high expressiveness of this newly proposed theory, some semantics were only generalized for the restricted bipolar ADFs yet.

This work will give an exhaustive overview on ADFs. The restriction to bipolar ADFs for some of the semantics is not desired, so we try to develop a solution to gain the generalized stable model semantics. This semantics is particularly important because the other semantics which are restricted to bipolar ADFs, depend on stable models. To gain such a generalization, we will try to connect the foundations of ADFs to other fields of computer science. So we may relate subclasses of these fields to the bipolar ADF to overcome this obstacle. This connection also makes ADFs more accessible to other fields of computer science.

We will concentrate mainly on the introduction of the alternative representation of propositional-formula ADFs (*pForm-ADFs*), but we will also show that ADFs can be represented as hyper-graphs. Based on the new representation a transformation from ADFs to *pForm-ADFs*, together with a generalization of the stable model semantics will be presented. In addition some properties between semantics will be investigated and an overview of complexity results, enriched with new ones is given.

Currently there is no software system available to compute semantics for ADFs. So in addition to the formal results we also present an Answer Set Programming (ASP) based implementation to solve these highly complex computations. We will also present preliminary empirical experiments.

Kurzfassung

Abstract Argumentation konnte im Laufe der letzten zwei Dekaden stetig immer mehr Interesse im Forschungsbereich der Künstlichen Intelligenz gewinnen. Eines der wichtigsten Konzepte ist dabei Dung's Argumentation Framework. Hierbei handelt es sich um einen einfachen, jedoch mächtigen und gut entwickelten Ansatz zum Darstellen von Argumenten und deren Beziehungen. Diese Informationen sind hierbei in Form eines gerichteten Graphen kodiert, wo jede Kante einen Angriff auf ein anderes Argument symbolisiert. Mittels dieses Frameworks ist es ebenfalls möglich, durch Semantiken Mengen von Argumenten auszuwählen und zu prüfen ob sie gewisse Eigenschaften besitzen (z.B. ob die Menge konfliktfrei ist).

Vor kurzem wurde in Form von *Abstract Dialectical Frameworks (ADFs)* eine Verallgemeinerung dieses Konzepts vorgestellt. Dabei werden die Beziehungen mittels einer vollständigen Funktion definiert, wodurch sie nicht mehr nur auf Angriffe beschränkt sind. Durch diese Funktion, welche *Akzeptanzbedingung* genannt wird, ist es nun möglich sehr komplexe Beziehungen zu beschreiben. Aufgrund dieser Ausdrucksstärke gibt es nun jedoch Probleme gewisse Semantiken für ADFs zu definieren. Daher ist die Unterklasse der bipolaren ADFs eingeführt worden.

Im Rahmen dieser Arbeit wird das Konzept der ADFs nochmals genau vorgestellt. Da die Einschränkung auf bipolare ADFs nur eine vorübergehende Lösung darstellt, wird versucht eine allgemeine Form der stabilen Modell-Semantik zu finden. Da dies die grundlegende Semantik für alle anderen ist, welche auf bipolare ADFs beschränkt sind, wird hierfür ein generalisierter Ansatz am meisten benötigt. Um dies zu erreichen werden Konzepte von anderen Bereichen der theoretischen Informatik genutzt um deren Ergebnisse für die Probleme mit bipolaren ADFs zu nutzen. Dadurch können ADFs ebenfalls leichter in jenen Gebieten eingesetzt werden.

Hauptsächlich werden die aussagenlogischen ADFs behandelt werden, da es mit deren Hilfe möglich ist ADFs relativ natürlich in bipolare umzuwandeln. Darauf aufbauend wird dann die Entwicklung einer allgemeinen stabilen Modell-Semantik gezeigt. Zusätzlich werden noch einige Eigenschaften zwischen einzelnen Semantiken diskutiert. Eine Zusammenfassung bereits vorhandener komplexitätsanalytischer Resultate wird ebenfalls präsentiert um danach neue Ergebnisse zeigen zu können.

Da es derzeit kein adäquates Software-System zum Berechnen von Modellen für Semantiken auf ADFs gibt, wird zusätzlich zu den formalen Ergebnissen noch eine neue ASP (Answer Set Programming) basierte Implementierung präsentiert. Um ein Gefühl für die Effizienz des Systems zu bekommen werden außerdem noch empirische Experimente zur Laufzeit diskutiert.

Contents

1 Introduction **1**

2 Background **5**
2.1 Argumentation Formalism 5
2.2 Dung's Abstract Argumentation Frameworks 6
2.3 Abstract Dialectical Frameworks 10
2.4 Propositional Logic 17
2.5 Complexity Theory 22

3 Alternative Representations of ADFs **25**
3.1 ADFs with Propositional Formulae 25
3.2 ADFs as Hypergraphs 36

4 Properties and Complexity of ADFs **41**
4.1 Transformations to Subclasses 41
4.2 Generalization of the Stable Extension 50
4.3 Relations between Semantics 53
4.4 Complexity Analysis of ADFs 55

5 Implementation **61**
5.1 ASP - Encodings 61
5.2 Experiments 77

6 Related Work **91**
6.1 Related Concepts 91
6.2 Related Software Systems 95

7 Conclusion & Future Work **97**

A Listing of the ADF $\rightarrow$ AF Encodings **99**
A.1 Model for ADFs 99
A.2 ADF $\rightarrow$ AF transformation 99

B Listing of the ADF System Encodings **101**

B.1 Linktypes 101
B.2 Conflict-free set 103
B.3 Model 104
B.4 Stable model 105
B.5 Admissible set 106
B.6 Preferred model 107
B.7 Well-founded model 109

Bibliography **111**

CHAPTER 1

Introduction

The theory of *Argumentation* is situated at the intersection of Philosophy [Barth and Krabbe, 1982, Toulmin, 2003, Walton, 1996], Artificial Intelligence (AI) [Caminada and Amgoud, 2007], and several application domains, like law interpretation [Bench-Capon, 2002, Bench-Capon and Dunne, 2005, Bench-Capon et al., 2009]. Within AI several subfields are particularly relevant and benefit from studies of argumentation. In particular Knowledge Representation [Amgoud and Cayrol, 2002, Besnard and Hunter, 2005, Prakken and Sartor, 1997], Non-Monotonic Reasoning [Bondarenko et al., 1997, Chesñevar et al., 2000, Prakken and Vreeswijk, 2002], Decision Making [Dimopoulos et al., 2009], and Multi-Agent Systems [Amgoud et al., 2005, Kakas and Moraitis, 2006]. For an overview on the steady rise of interest in this field of AI over the last two decades see the survey article by Bench-Capon and Dunne [2007] or the books by Besnard and Hunter [2008] or Rahwan and Simari [2009].

The basic idea of *Argumentation* is to provide a concept to draw conclusions based on knowledge. This knowledge can consist of complex rules of requirements, relations, and inferences. Typically in argumentation this knowledge is represented in an abstract way as a set of arguments which may be accepted or not. These arguments stand in relations to each other to picture only the needed information from the knowledge base. Based on these relations conclusions on the acceptability of the arguments can be inferred. Together with the knowledge of the procedure of the instantiation of these arguments, the inferred acceptability of the arguments leads to conclusions for a given knowledge base.

The field of *abstract argumentation* is concerned with the representation of arguments and its conclusions only. This area typically utilizes *argumentation frameworks* as their formal modeling language. A particular well-known and widely studied representation is *Dung's Argumentation Framework* (*AF*) [Dung, 1995]. There arguments are related with each other via attack-relations. With this model it is possible to represent whether some arguments are in conflict with other arguments or not. Based on these conflicts acceptable (conflict-free) viewpoints are desired. These viewpoints are sets of acceptable arguments which have to fulfill different properties. They can be seen as a conflict-resolution and are introduced as semantics for the framework. In particular the sets of accepted arguments are known as extensions. This concept

has proved to be simple but yet powerful in expressiveness, so it evolved to one of the most used basic concepts in abstract argumentation.

Alas, the simplicity of Dung's AF has some disadvantages in more complex relations between arguments. It is not possible to represent natively concepts like support between arguments, distributed attacks, or other more complex dependencies than direct attacking relations between arguments. Indeed it is possible to evaluate these concepts with specific structures between the arguments, but these use artificial arguments which have no meaning for conclusions and are only means to an end.

To overcome the shortcomings of Dung's AF many different frameworks, semantics, and improvements emerged (e.g. the concept of Meta-Argumentation [Boella et al., 2009], Attack semantics for Dung's AF [Villata et al., 2011], and Constraint Argumentation Frameworks [Coste-Marquis et al., 2006]). Recently a generalization of Dung's AF, so called *Abstract Dialectical Frameworks* (*ADFs*) [Brewka and Woltran, 2010], was proposed. There the relations between now called *statements* are generalized in such way that a boolean function decides whether a statement should be accepted or not. With this generalized approach it is now possible to natively represent the above mentioned complex relations between statements. Due to the high grade of freedom in the definition of the acceptance via these boolean functions, it is complicated to define some of the semantics in a generalized form of Dung's extensions. Intuitively the complications come from the possibility to define relations which change their behavior in such way that they are sometimes attacking and sometimes supporting, based on the acceptance of other arguments. To present a consistent picture the subclass of *bipolar Abstract Dialectical Frameworks* (*BADFs*) has been introduced. There each relation between two arguments is intuitively either attacking or supporting. So the complicated relations are not allowed to occur in *BADFs*.

Although the introduction of *BADFs* was useful to define complex semantics of *ADFs* they are actually a restriction of the whole formalism. Despite that all Dung's AF are in the class of *BADFs* it is desirable to gain a generalization of the definitions for these restricted semantics, to remove this artificial restriction of *ADFs*. The aim of this work is to analyze the properties of *ADFs* and their semantics to find a generalization of the stable extension, which is one of the restricted semantics. As all other restricted semantics are based on this one, a generalized result for the stable extension is the most needed. In addition an implementation for further benchmarks and comparisons will be engineered. This is particularly important as there are currently no software systems available to work with *ADFs*, which makes empiric comparisons and benchmarks with other problem instances in the field of argumentation impossible. For efficient representations in the implementation boolean functions are too space expensive, so in addition alternative representations need to be found. Such alternative representations will also make the field of *ADFs* more accessible to other fields of computer science. This connection to other fields allows the usage of already known results of these disciplines to solve existing problems for *ADFs*.

Based on the introduced problems and the presented aim of the work, we want to summarize the main contributions of this thesis:

- alternative representations for *ADFs*
- a generalization of the stable extension
- an implementation of a system to find accepted sets with respect to different semantics

To achieve these aims, many parts of the work use formal methods to analyze and prove facts about the properties and the complexity of the *ADFs*. The computation of most of the semantics has a high computational effort (they reside at least in the first level of the polynomial hierarchy). To solve these hard problems, the highly sophisticated paradigm of *Answer Set Programming (ASP)* is used for the implementation of the system (for an ASP-overview see [Brewka et al., 2011c]). The already existing, competition-winning [Calimeri et al., 2011, Denecker et al., 2009, Gebser et al., 2007b] solver `clasp`[1] [Gebser et al., 2007a] is the underlying layer to solve the ASP problems. The system description of the implementation[2] [Ellmauthaler and Wallner, 2012] has been accepted as a demonstration for the International Conference on Computational Models of Argument (COMMA 2012), which covers the latest research results related to the computational aspects of argumentation.

The structure of the work will be as follows. In Chapter 2, the basic preliminaries and the background for the further thesis is presented. Here a more detailed introduction to Argumentation, Abstract Argumentation, Dung's AF, and *ADFs* will be pictured. In additional classical propositional logic together with a quick overview of the basics of complexity theory will be given. Afterwards Chapter 3 will propose two different representations for *ADFs*. One with the aim to connect *ADFs* to propositional formulae as *pForm-ADFs* and the other to show that the generalization of Dung's AFs can also be represented entirely by graph-theory as *Hyper-graphs*. Then the properties and the complexity of *ADFs* will be analyzed in Chapter 4. Here some transformations to subclasses are presented and our transformation from *ADFs* to *BADFs* will be proposed. This transformation is needed for the generalization of the stable extension which is also shown in this chapter. Then we will analyze some of the relations between extensions of *ADFs* and compare them to already known relations for their equivalents on Dung's AF. Finally some already known complexity results will be presented and then new results are provided. The last main contribution is presented in Chapter 5. Here the encodings for the software systems and benchmarks are presented in detail. Then Chapter 6 will discuss the related work and in Chapter 7 some concluding words together with an outlook to future work will be given.

[1] part of the Potsdam Answer Set Solving Collection (Potassco), may be downloaded at http://potassco.sourceforge.net

[2] see http://www.dbai.tuwien.ac.at/research/project/argumentation/adfsys/ for the sources of the running system

CHAPTER 2

Background

In this chapter we will give an overview on the field of formal argumentation. In particular we will review the widely used Abstract Argumentation Framework, proposed by Dung [1995], and a generalized framework, called an *Abstract Dialectical Framework*, introduced by Brewka and Woltran [2010]. The basics for propositional logic and complexity theory will also be reviewed in this chapter, as we will need them in further chapters.

2.1 Argumentation Formalism

Formal argumentation is based on the idea to construct and evaluate *arguments* to model the concept of (nonmonotonic) reasoning. In this context these arguments are defeasible, which means that their conclusion may change because of the existence of other arguments. This concept stands in a direct contrast to a proof theoretic approach, because a proven fact remains proven even in the light of additional information.

The argumentation formalism can be represented as a *three-step process* [Caminada and Wu, 2011]. Argumentation can be used for many different fields of application, therefore the input and the type of the output can vary, but the concept behind the steps will not change. We will use the nonmonotonic entailment problem as one application to explain how the three steps work (see also Figure 2.1). Intuitively an entailment problem is the task to find the sets of consequences which can be deduced from a given, possibly inconsistent, knowledge-base. For the deduction some form of logical reasoning is needed, which will be a nonmonotonic logic in our case. At first we have a knowledge-base which is the input for the entailment problem. Based on the elements of the knowledge base we can construct a set of arguments and relate them among themselves. This construction is the first step, the *instantiation*. The resulting set of arguments and their relations are represented as a framework for argumentation. Based on this framework sets of accepted arguments can be identified. How the *identification of arguments* (second step) is done is in general given by semantics for the framework. These selected sets are referred to as *extensions of arguments*. The third, and last, step is the *identification of accepted*

conclusions. There the extensions of the arguments are further analyzed and on their basis accepted conclusions are identified, which are represented as *extensions of conclusions*. These extensions yield the conclusion for the entailment problem and therefore the problem was solved via the argumentation formalism.

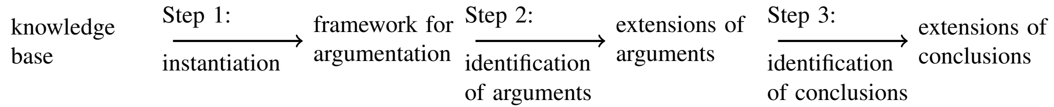


Figure 2.1: Three-step-process of argumentation

In this work we will not deal with the instantiation or the identification of conclusions. So the aim is on the framework itself and the identification of accepted arguments and we are talking only about the arguments and their relations and we do not care about the reason of their relation. This aim and point of view places this work in the field of *Abstract Argumentation*. For further details on the formal argumentation process see the work of Caminada and Amgoud [2007] as well as Caminada and Wu [2011].

2.2 Dung's Abstract Argumentation Frameworks

To model the process of abstract argumentation Dung introduced an Abstract Argumentation Framework (*Argumentation Framework - AF*). In this framework it is possible to model the arguments and their relations. These relations are modeled via binary attacks among two arguments. So we can represent that an individual argument is attacking another one or is attacked by it.

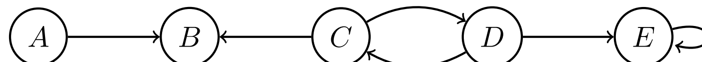
Definition 2.2.1 (Argumentation Framework [Dung, 1995])**.** *An Argumentation Framework is a pair*

$$AF = (AR, AT)$$

where AR is a set of arguments, and AT is a binary relation on AR, i.e. $AT \subseteq AR \times AR$

The meaning of $(A, B) \in AT$ for two arguments $A, B \in AR$, is that (A, B) represents an attack of A against B. As the definition of the framework is syntactically the same as for a directed graph, frameworks can be directly represented as such, where the nodes are the arguments and the attacks are the directed edges.

Example 2.2.2.
The $AF_1 = (\{A, B, C, D, E\}, \{(A, B), (C, B), (C, D), (D, C), (D, E), (E, E)\})$ can also be represented by the following graph:



In this example we can see that argument A is not attacked by anyone, while argument C and D have a mutual attack relation between them. Argument E is not only attacked by D, but also attacks itself.
This argumentation framework can model different situations. Maybe the 5 arguments stand for different employees in a company and they have to form a team. The attacks can stand for the preferences such that one employee does not want to work with another one for some distinct reasons. Based on the given AF, everyone will be fine with A in the team and maybe E does not want to be in the team (as he attacks himself).

Based on the *AF* we want to get a selection of arguments which is based on the relations between the different arguments. This selection shall model the reasoning process to determine consistent or reasonable sets which lead to a conclusion. Which arguments are selected is determined by the *semantics*, which are defined for the framework.

The most intuitive property is the *conflict-free set* (Definition 2.2.3), where no arguments are inside the set, which are attacking each other. Therefore there cannot be any direct conflicts among the selected arguments.

Definition 2.2.3 (Conflict-free set [Dung, 1995])**.** *Let $AF = (AR, AT)$ be an argumentation framework. A set $S \subseteq AR$ of arguments is said to be* conflict-free *in AF if there are no arguments A and B in S such that A attacks B. The set $cf(AF)$ is the set of all* conflict-free sets *for the argumentation framework AF.*

Example 2.2.4 (Conflict-free set, based on Example 2.2.2)**.**

$$cf(AF_1) = \{\emptyset, \{A\}, \{B\}, \{C\}, \{D\}, \{A, C\}, \{A, D\}, \{B, D\}\}$$

The notion of a *conflict-free set* is only avoiding attacks between any pair of selected arguments. As it does not care about attacks from the unselected arguments one can state that the constructed set can be attacked, and therefore invalidated easily. Based on the example (Example 2.2.2 and 2.2.4), $\{\emptyset\}$ and $\{A\}$ would be the only two *conflict-free sets*, where the selected arguments are not attacked from unselected ones. To ensure that the selected arguments are still prepared against attacks from unselected arguments we can *"defend"* them by an attack from one selected argument against the unselected one. If an argument inside the *conflict-free set* $M \in cf(AF)$ is *"defended"*, we say that it is *acceptable w.r.t.* M. The set $M_1 \subseteq AR$ is said to attack the set $M_2 \subseteq AR$ if a relation $(a, b) \in AT$ exists, such that $a \in M_1$ and $b \in M_2$.

Definition 2.2.5 (Acceptable argument [Dung, 1995])**.** *Let $AF = (AR, AT)$ be an argumentation framework. An argument $A \in AR$ is* acceptable *in AF with respect to a set $S \subseteq AR$ of arguments iff for each argument $B \in AR$: if B attacks A then B is attacked by S.*

Definition 2.2.6 (Admissible set [Dung, 1995])**.** *Let $AF = (AR, AT)$ be an argumentation framework. A conflict-free set of arguments $S \subseteq AR$ is* admissible *in AF iff each argument in S is acceptable with respect to S. The set of admissible sets of the framework AF is denoted by $adm(AF)$.*

Example 2.2.7 (Admissible set, based on Example 2.2.2)**.**

$$adm(AF_1) = \{\emptyset, \{A\}, \{C\}, \{D\}, \{A, C\}, \{A, D\}\}$$

We can see that in Example 2.2.7 some *admissible sets* of arguments are included in another one. To define the (credulous) semantics for the *AF* we need to represent the maximal *admissible set*, which is called the *preferred extension*.

Definition 2.2.8 (Preferred extension [Dung, 1995])**.** *A* preferred extension *of an argumentation framework AF is a maximal (with respect to set inclusion) admissible set of AF.*

Example 2.2.9 (Preferred extension, based on Example 2.2.2)**.** *Our example has two preferred extension $E^1_{AF_1} = \{A, C\}$ and $E^2_{AF_1} = \{A, D\}$.*

Another, even more restricted semantics is the *stable extension*.

Definition 2.2.10 (Stable extension [Dung, 1995])**.** *Let $AF = (AR, AT)$ be an argumentation framework. A conflict-free set of arguments $S \subseteq AR$ is called a* stable extension *in AF iff S attacks each argument which does not belong to S.*

Example 2.2.11 (Stable extension, based on Example 2.2.2)**.**
Here the stable extension is $SE_{AF_1} = \{A, D\}$.

It is obvious to see that the *stable extension* is also a *preferred extension* in the examples. Indeed one of the results by Dung states that *every stable extension is a preferred extension, but not vice versa.* [Dung, 1995].
In contrast to the credulous semantics another approach is used to introduce the skeptical semantics. This one is the *grounded extension* (Definition 2.2.13), which is based on a characteristic fix-point function.

Definition 2.2.12 (Characteristic function [Dung, 1995])**.** *The* characteristic function $F_{AF} : 2^{AR} \to 2^{AR}$ *of an argumentation framework $AF = (AR, AT)$ is defined as follows:*
$F_{AF}(S) = \{A | A$ *is acceptable with respect to* $S\}$.

Definition 2.2.13 (Grounded extension [Dung, 1995])**.** *The* grounded extension GE_{AF} *of an argumentation framework AF is the unique least fixed point of F_{AF}.*

Additionally there is another semantics defined, which relates the preferred extensions and the grounded extensions (i.e. relating credulous and skeptical semantics). This is achieved with the *complete extension*.

Definition 2.2.14 (Complete extension)**.** *An admissible set $S \subseteq AR$ of arguments is called a* complete extension *of an argumentation framework $AF = (AR, AT)$ iff each argument, which is acceptable with respect to S, belongs to S.*

Example 2.2.15 (Grounded and complete extensions, based on Example 2.2.2)**.** *In our example the grounded extension would be $GE_{AF_1} = \{A\}$ and the complete extensions are $CE_{AF_1} = \{\{A\}, \{A, C\}, \{A, D\}\}$.*

The mentioned relation between the extensions is summarized in Theorem 2.2.16.

Theorem 2.2.16 ([Dung, 1995])**.**

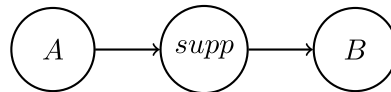
(I) Each preferred extension is a complete extension, but not vice versa.

(II) The grounded extension is the least (with respect to set inclusion) complete extension.

(III) The complete extensions form a complete semilattice[1] *with respect to set inclusion.*

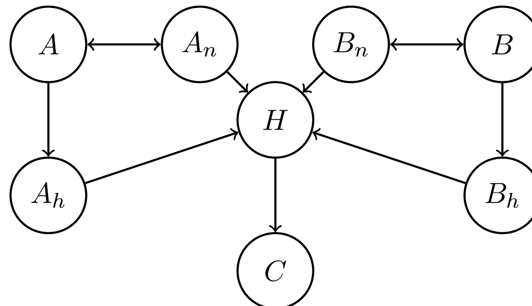
It is obvious that the AF is only capable of modeling binary attacks directly among two arguments. More sophisticated relations are represented via distinct structures which are specially tailored for a prior chosen semantics. In the following we will describe a few approaches for such *semantic structures* which are primary based on the stable extension.

Example 2.2.17 (Semantic structure for a support relation)**.**



The above AF *models a support between the node A and the node B under the stable extension. We might not be interested in the* support-argument "supp" *and remove it from the set of arguments in the final conclusion.*
The above structure has the following stable extension: $SE = \{A, B\}$

Example 2.2.18 (Semantic structure for a distributed attack)**.** *Here we want to give an example for a distributed attack (i.e. two arguments A and B attack together one argument C, but only if both are accepted) under the stable extension.*



[1]A partial order $(S, \leq)$ is a complete semilattice iff each nonempty subset of S has a greatest lower bound and each increasing sequence of S has a lower upper bound.

Here we have the following stable extensions (the interesting nodes are underlined):

$$SE^1 = \{\underline{A}, \underline{B}, H\}$$
$$SE^2 = \{\underline{C}, A_n, B_n, A_h, B_h\}$$
$$SE^3 = \{\underline{B}, \underline{C}, A_n, A_h\}$$
$$SE^4 = \{\underline{A}, \underline{C}, B_n, B_h\}$$

Example 2.2.17 shows that we can define a support with the help of another *helper* argument, which forces the stable extension to select B if we have selected A. Example 2.2.18 tries to define a structure which pictures a distributed attack. Here both attacking arguments, A and B, need to be selected by the semantics to have an applicable attack against the third argument C. These examples shall only motivate that it is possible to represent more relations between the arguments than only an attack. We still have to keep in mind that we need special structures which are bound to a specific semantics. For further observations and discussions about relation of the semantics of Dung's AF, see [Baroni et al., 2011a].

2.3 Abstract Dialectical Frameworks

One of the most criticized aspects of Dung's *AF* is that it is only possible to represent direct attacks among two arguments. We have already seen that it is possible to relate two arguments in a more complex manner, but this is highly dependent from the used semantics. To achieve more expressive power on the side of the framework, Brewka and Woltran proposed another, more general framework for argumentation, namely the *Abstract Dialectical Framework - ADF* [Brewka and Woltran, 2010]. This framework is based on the same basic ideas as Dung's *AF*, which are the utilization of arguments and the representation of relations with a binary relation. In addition to these basics, acceptance conditions are added to the framework. These abstract conditions cover any function to describe the relation between an argument and all its parents (i.e. all arguments it is dependent upon). Additionally Brewka and Woltran also have shown that every *AF* can be represented as an *ADF*. The semantics for *ADFs* are also generalizations of the semantics defined by Dung.

Characterization of Abstract Dialectical Frameworks

Now we will give a formal characterization of the *ADFs*. Again we use a graph-like structure, but now we have a set of statements instead of the set of arguments. With the change of the naming it shall be emphasized that we have positions, instead of arguments, which can be accepted or not. To describe the dependency for the acceptance, the statements are connected via links. How the status of a given statement looks like is determined by the dependencies among the statement and all direct parents in the graph.

To denote the set of parents for a statement s, *par(s)* is used. In addition to the links every node s has an *acceptance condition* C_s, which is associated to the node and depends on the links. This condition distinguishes whether a statement shall be accepted or not. C_s is a function

which maps each subset of $par(s)$ to either *in* or *out*. Intuitively we can say that if $R \subseteq par(s)$ are accepted, $par(s) \setminus R$ are not accepted and $C_s(R) = in$ (or $C_s(R) = out$) then s shall (not) be accepted.

Definition 2.3.1 (Abstract Dialectical Framework [Brewka and Woltran, 2010]). *An* abstract dialectical framework *is a tuple $D = (S, L, C)$ where*

- *S is a set of statements (positions, nodes)*
- *$L \subseteq S \times S$ is a set of links*
- *$C = \{C_s\}_{s \in S}$ is a set of total functions $C_s : 2^{par(s)} \rightarrow \{in, out\}$, one for each statement s. C_s is called acceptance condition of s.*

Previously we claimed that the *ADFs* are a generalization of Dung's *AFs*. So we want to show how Dung's *AFs* can be captured via *ADFs*. As both frameworks use a set of arguments of respectively statements, these can be taken over as they are. Dung's *AF*($AF = (AR, AT)$) only knows one type of relation between two arguments, so we can also use the attack relations as the links. For the acceptance conditions we need to generate the following conditions: For each $s \in AR : C_s(R) = in$ *iff for each* $r \in R : (r, s) \notin AT, C_s(R) = out$ otherwise.

Definition 2.3.2 (Dung Style ADF). *Let $AF = (AR, AT)$ be a Dung's AF. An ADF $D = (S, L, C)$ is a* Dung Style ADF *iff it is constructed by the following rules:*

(I) $S = AR$

(II) $L = AT$

(III) Use for all statements $s \in AR$, $C_s(\emptyset) = in$ if no statement $r \in AR$ exists, such that $(s, r) \in AT$. Otherwise $C_s(R) = out$ for all $\emptyset \subset R \subseteq par(s)$.

Example 2.3.3 (ADF based on Example 2.2.2). *We will now show how the ADF D_1 will look like, based on the AF AF_1:*
$D_1 = (S_1, L_1, C_1)$*, where*
$S_1 = \{A, B, C, D, E\}$*,*
$L_1 = \{(A, B), (C, B), (C, D), (D, C), (D, E), (E, E)\}$*, and*
$C_1 = \{C_A, C_B, C_C, C_D, C_E\}$*.*
The functions in C_1 have the following mapping:

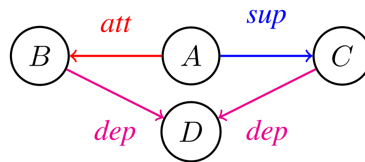
$C_A(\emptyset) = in$	$C_B(\emptyset) = in$	$C_C(\emptyset) = in$	$C_E(\emptyset) = in$
	$C_B(\{A\}) = out$	$C_C(\{D\}) = out$	$C_E(\{D\}) = out$
	$C_B(\{C\}) = out$	$C_D(\emptyset) = in$	$C_E(\{E\}) = out$
	$C_B(\{A, C\}) = out$	$C_D(\{C\}) = out$	$C_E(\{D, E\}) = out$

Bipolar Abstract Dialectical Frameworks

In the section about *ADFs* we have shown that we are able to model attacks between two statements. Indeed it is also possible to model supports, such that the acceptance of one statement will approve the acceptance of another one. In addition a third case can occur: The link is neither attacking nor supporting and is called *dependent*. How such an *ADF* with all three types of links can look like is demonstrated in Example 2.3.4

Example 2.3.4 (Link types)**.** *We use the ADF $D_2 = (S_2, L_2, C_2)$ to show how the three link types can look like:*
$D_2 = \{A, B, C, D\}$
$L_2 = \{(A, B), (A, C), (B, D), (C, D)\}$
$C_2 = \{C_A, C_B, C_C, C_D\}$

$C_A(\emptyset) = in$	$C_B(\emptyset) = in$	$C_C(\{A\}) = in$	$C_D(\emptyset) = out$
	$C_B(\{A\}) = out$	$C_C(\emptyset) = out$	$C_D(\{B\}) = in$
			$C_D(\{C\}) = in$
			$C_D(\{C, B\}) = out$



*The graphic shows which links are attacking (*att*), supporting (*sup*), and dependent (*dep*). Intuitively we have the attacking link between A and B, because the acceptance of A will switch the value of the acceptance condition of B from in to out. For the supporting link it is the same, but here the acceptance of A will switch the acceptance condition of C to be in. It can be a little bit irritating why the links to D are neither attacking nor supporting. We can see that D is not accepted if neither B nor C is accepted. If one of them is selected we will have to accept D, which is obviously a support. So lets suppose we had nothing accepted and now we are accepting B. Therefore we will accept D and so we have a supporting nature. If we accept now C, we will have to reconsider the acceptance of D and remove it. There we have an attacking nature of C against D. Now we change the order of acceptance, such that C is accepted before B. In this situation the attacking and supporting nature of the links will be interchanged between them. So the nature of the two links is neither attacking nor supporting, but it is dependent on other parents of the statement.*

As we already know that Dung's *AF* can be seen as a subclass of *ADFs* where only attacking links are allowed, we can also construct a subclass where only supports are allowed. These *ADFs* are called *monotonic ADFs* [Brewka and Woltran, 2010]. In Definition 2.3.5 a formal definition for attacking and supporting links is presented. Links which are not attacking and not supporting are supposed to be dependent. It was already mentioned in the example that this name comes from the point that the nature is changing, dependent from other parents of the linked node. These dependent links make some semantics difficult to be defined. So the subclass

of *ADFs* where every link has to be attacking or supporting was introduced. These *ADFs* are called *Bipolar Abstract Dialectical Frameworks (BADF)*.

Definition 2.3.5 ([Brewka and Woltran, 2010]). *Let $D = (S, L, C)$ be an ADF. A link $(r, s) \in L$ is*

(I) supporting iff for no $R \subseteq par(s)$ we have that $C_s(R) = in$ and $C_s(R \cup \{r\}) = out$,

(II) attacking iff for no $R \subseteq par(s)$ we have that $C_s(R) = out$ and $C_s(R \cup \{r\}) = in$.

Definition 2.3.6 (Bipolar Abstract Dialectical Framework [Brewka and Woltran, 2010]). *Let $D = (S, L, C)$ be an ADF. If for all links $(r, s) \in L$, (r, s) is either supporting or attacking, the ADF D is called a* Bipolar Abstract Dialectical Framework.

Semantics

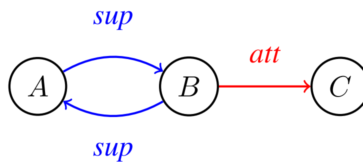
As we have defined *ADFs* and the subclass of *BADFs* we will now explain how the different semantics work and how they are related to the semantics from Dung's *AF*. We will discuss the specific semantics in an altered order (w.r.t. Section 2.2 - *AF*) as the stable (Definition 2.3.10) and preferred model (Definition 2.3.15) are only defined for *BADFs*. In addition we have to keep in mind that the *ADFs* are more expressive, so more properties have to be considered for the semantics.

Definition 2.3.7 (Conflict-free set [Brewka and Woltran, 2010]). *Let $D = (S, L, C)$ be an* ADF. *A set $M \subseteq S$ is* conflict free *if $\forall s \in M$ we have $C_s(M \cap par(s)) = in$. The set $cf_{ADF}(D)$ is the set of all* conflict-free sets *for the* ADF *D.*

The first semantics we are introducing will correspond to the stable extension if we only use *ADFs* which are direct representations of Dung's *AFs*. This will be done by the notion of a *model*, which is intuitively a set of statements which are satisfying the acceptance conditions for each node. Additionally it will also ensure that each satisfied node is in the set.

Definition 2.3.8 (Model [Brewka and Woltran, 2010]). *Let $D = (S, L, C)$ be an* ADF. *$M \subseteq S$ is a* model *of D if $M \in cf_{ADF}(D)$ and for each $s \in S, C_s(M \cap par(s)) = in$ implies $s \in M$. $model_{ADF}(D)$ is the set of models for the* ADF *D.*

Example 2.3.9 (Model for different *ADFs* (based on Example 2.3.3 and 2.3.4)). *The unique model M_1 for the* ADF *D_1 and the unique model M_2 for the* ADF *D_2 are $M_1 = \{A, D\}$ and $M_2 = \{A, C, D\}$. Another interesting example is the* ADF *$D_3 = (S_3, L_3, C_3)$:*



$S_3 = \{A, B, C\}$, $L_3 = \{(A, B), (B, A), (B, C)\}$, *and* $C_3 = \{C_A(\emptyset) = C_B(\emptyset) = out,$ $C_A(\{B\}) = C_B(\{A\}) = in, C_C(\emptyset) = in, C_C(\{B\}) = out\}$.
Here we have two models $M_3^1 = \{A, B\}$ *and* $M_3^2 = \{C\}$. *It is hard to argue that* M_3^1 *is a model of* D_3*, because* A *is only in the set because* B *is inside the set and vice versa. This may be unintuitive when we reduce the circle to one element. Then expressions like "I am right because I postulate it" would be selected by the model as a valid extension.*

The examples illustrates the equality between the *stable extension* of Dung's *AF* with a corresponding *ADF*. We can also see that there are some problems with *self-supporting cycles* as the existence of such constructs is enough to qualify the members of the cycle to be in a model. To get rid of the cycles we will need to check whether the elements of the model are accepted as a result of the cycles or not. Brewka and Woltran [2010] utilized the idea of a reduction from the Gelfond-Lifschitz-Reduct [Gelfond and Lifschitz, 1988]. For the reduction they use the property of the existence of a *unique least model* for a *monotonic ADF*. This model can be constructed via an operator Th_D, which has a least fixed-point. Th_D is a function $Th_D : 2^S \to 2^S$ which is defined as

$$Th_D(M) = \{s \in S \mid C_s(M) = in\}.$$

Intuitively the function collects in each step all statements which can be accepted and the acceptance is only based on already accepted statements. We will start this operation with an empty set, so there always has to be a general accepted statement on which the acceptance has to build up. Therefore a self-supporting cycle would not be selected if there is no additional statement which is a supporting member of the cycle. With this definition of the function for the least model of a *monotonic ADF* we can define the whole transformation for the *stable model*:

Definition 2.3.10 (Stable model [Brewka and Woltran, 2010])**.** *Let* $D_B = (S, L, C)$ *be a bipolar ADF. A model* M *of* D_B *is a* stable model *if* M *is the least model of the reduced ADF* D_B^M *obtained from* D_B *by*

(I) eliminating all nodes not contained in M *together with all links in which any of these nodes appear,*

(II) eliminating all attacking links,

(III) restricting the acceptance conditions C_S *for each remaining node* s *to the remaining parents of* s*.*

$smodel_{ADF}(D)$ *will be used to refer to the set of all* stable models *of the* ADF D*.*

The elimination of nodes and links in the first and second reduction step is obtained by removing them from the set S respectively L. In addition the restriction in the last reduction step is done by deleting all acceptance condition mappings which contain removed nodes or links. Note that the other conditions are not altered.

Now it should be easy to see why the stable model is only defined for *BADFs*. During the elimination process all attacking links are removed. As we have a *BADF* we know that only supporting links remain. So we do know that the reduced *ADF* is monotonic and has a unique

least model. This would not be possible if there are also dependent links in the reduction as we do not have a monotonic *ADF*. The intuition behind the stable model is to check for the model of an *ADF* if the accepted statements where accepted because of a "traceable" reason or a support cycle.

Proposition 2.3.11. *For every Dung Style ADF D $model_{ADF}(D) = smodel_{ADF}(D)$ holds.*

Proof. (i) Let $M \subseteq S$ be an arbitrary model of the ADF $D = (S, L, C)$, where the ADF is based on a Dung's AF. We know that $M \in cf_{ADF}(D)$ which implies that the acceptance condition $C_m(M)$ of all statements $m \in M$ is mapping to *in*. L has only attacking links, so for no $C_S(R) = in$ a mapping $C_S(R') = out$, where $R' \subset R$ holds, can exist. The reduced ADF $D^M = (S^M, L^M, C^M)$ has no links because D only has attacking links, as it is based on Dung's AF. Therefore all remaining acceptance conditions in C^M have the mapping $C_S(\emptyset) = in$ for all $s \in S^M$ and so all arguments in S^M are selected by $Th_D(S^M)$. (ii) Every stable model is a model by definition. □

Example 2.3.12 (Stable models for ADFs, continuation of Example 2.3.9)**.** *D_2 is no* BADF*, so the definition does not apply here. Let us have a look at D_1: The reduced* ADF *D_1^M will be: $S_1^M = \{A, D\}, L_1^M = \{\}, C_1^M = \{C_A(\emptyset) = in, C_D(\emptyset) = in\}$, and the least model of D_1^M is $\{A, D\}$. This is the indicator that the stable model of D_1 is $\{A, D\}$.*
Now we will take a look on the ADF *D_3, which has the self supporting cycle. There we have two models, so we have to build two reductions. We first check the model $M_3^1 = \{A, B\}$. Here we have the reduced* ADF *$D_3^M = \{\{A, B\}, \{(A, B), (B, A)\}, \{C_A(\emptyset) = C_B(\emptyset) = out, C_A(B) = C_B(A) = in\}\}$. The least model of this* ADF *is $\{\}$, so M_3^1 is not a stable model of D_3. Now we have to check the second model $M_3^2 = \{C\}$. The reduced* ADF *$D_3^M = \{\{C\}, \{\}, \{C_C(\emptyset) = in\}\}$ has as the least model $\{C\}$ and therefore this one is a stable model. So the only stable model for D_3, as expected, is $\{C\}$.*

To define the *preferred model* (Definition 2.3.15) we will use a characterization based the relations between the admissible set, the stable extension and the preferred extension of Dung's *AF*:

Proposition 2.3.13 ([Brewka and Woltran, 2010])**.** *Let $AF = (AR, AT)$ be an argumentation framework. $E \subseteq AR$ is admissible in AF iff there is some $R \subseteq (AR \setminus E)$ such that*

(I) no element in R attacks an element in E, and

(II) E is a stable extension of the reduced argumentation framework

$$AF - R = (AR \setminus R, \{(a, b) \in AT \mid a, b \in AR \setminus R\}).$$

We can generalize the *admissible* property for Dung's *AF*, so we will get the definition for the *admissible* property for *ADFs*.

Definition 2.3.14 (Admissible set for BADFs [Brewka and Woltran, 2010])**.** *Let $D = (S, L, C)$ be a* BADF*. $M \subseteq S$ is admissible in D iff there is $R \subseteq S$ such that*

(I) no element in R attacks an element in M, and

(II) M is a stable model of $D - R$. $D - R$ is obtained by:

(i) deleting all statements $s \in R$ from D,

(ii) removing all links $(a, b) \in L$, where $a \in R$ or $b \in R$, and

(iii) restricting the acceptance conditions to the remaining parents.

Now it is trivial to tailor the definition for the *preferred model*, based on the definition for the *preferred extension*. But we still have to keep in mind that this only works for *BADFs*.

Definition 2.3.15 (Preferred model [Brewka and Woltran, 2010]). *M is a* preferred model *of D iff M is (inclusion) maximal among the sets admissible in D.*

Compared with the section on semantics for Dung's *AF* we are only missing the *grounded extension* and the *complete extension*. The latter extension was not generalized and will not be discussed. The generalization of the *grounded extension* will work for all *ADFs* and is not restricted to the subclass of *BADFs*. The basic idea of the *grounded extension* is to find the least fix point of a function which collects all arguments which can be accepted with respect to the current selection. We have generalized Dung's *AF* with the possibility to support other statements, so we will generalize the *grounded extension* in the same manner. Intuitively we want a function which collects all acceptable statements with respect to the currently selected statements (i.e. their acceptance condition says *in*) and rejects all statements which are definitely *out*. So the number of *"undecided"* elements, which are neither accepted nor rejected is reduced with each application of the function. In case there are no undecided statements left or the number of undecided statements cannot be reduced further, the fix point of the function and the desired result is reached. We will call this extension the *well-founded model*.

Definition 2.3.16 (Well-founded model [Brewka and Woltran, 2010]). *Let $D = (S, L, C)$ be an ADF. Consider the operator*

$$\Gamma_D(A, R) = (acc(A, R), reb(A, R))$$

where $acc(A, R) =$

$$\{r \in S | A \subseteq S' \subseteq (S \setminus R) \Rightarrow C_r(S' \cap par(r)) = in\}$$

and $reb(A, R) =$

$$\{r \in S | A \subseteq S' \subseteq (S \setminus R) \Rightarrow C_r(S' \cap par(r) = out\}$$

Γ_D is monotonic in both arguments and thus has a least fixed-point. E is the well founded model of D iff for some $E' \subseteq S$, (E, E') is the least fixed-point of Γ_D.

The following example will picture the *admissible set*, the *preferred model* and the *well-founded model*.

Example 2.3.17 (Admissible set, preferred model and well-founded model for ADFs, continuation of Example 2.3.9)**.** *D_2 is no* BADF*, so we will present only solutions for D_1 and D_3 under the* admissible set *and the* preferred model.
The admissible sets for D_1 are $\{a\}$, $\{c\}$, $\{d\}$, $\{a, c\}$, and $\{a, d\}$. The two sets $\{a, c\}$ and $\{a, d\}$ are the maximal sets w.r.t. the subsets, so they are the two preferred models. The well-founded model is $\{a\}$, as the fixed-point of Γ_{D_1} is $(\{a\}, \{b\})$.
For D_2 the well-founded model is $\{a, c, d\}$ with the fixed-point result of $(\{a, c, d\}, \{b\})$.
D_3 has one admissible set, namely $\{c\}$, which is also the preferred one. Its well-founded model is the empty set, as Γ_{D_3} is $(\emptyset, \emptyset)$

2.4 Propositional Logic

In the following we will introduce the widely used and common *classical propositional logic* and some of its most important properties. For an exhaustive introduction to classical logic from the mathematical point of view see [Church, 1996, Rothmaler, 2000].

The classical logic is a binary logic. This means any variable can only have one of two values (i.e. a variable can be either true or false). We can combine different variables and constants to some sort of logical sentence, which is a logical proposition. As in natural languages there are some rules how such a sentence has to look like. Here a valid sentence is called a well-formed formula.
To decide whether a formula is a well-formed propositional formula or not, we define its syntax which declares the allowed symbols and how they are connected. Note that the syntax only distinguishes between sequences of symbols which are allowed and those which are not allowed.

Definition 2.4.1 (Basic syntax of the propositional logic)**.**
Given a signature $\Sigma := (\Sigma_c, \Sigma_{pv}, \Sigma_{con})$, where $\Sigma_c = \{\top, \bot\}$ is a set of constant symbols, Σ_{pv} is a set of propositional variables, and $\Sigma_{con} = \{\vee, \wedge, \neg\}$ is a set of connectives.

Inductive definition of a well-formed propositional formula:

(i) Every $p \in \Sigma_{pv}$ is a formula.

(ii) Every $c \in \Sigma_c$ is a formula.

(iii) If ϕ is a formula, then (ϕ) is a formula too.

(iv) If ϕ is a formula, then $\neg\phi$ is one too.

(v) If ϕ and ψ are formulae and $\circ \in \{\vee, \wedge\}$, then $\phi \circ \psi$ is one too.

The formulae defined by (i) and (ii) are called atomic formulae. *Non-atomic formulae are* compound formulae. *The formulae used to build a compound formula are* sub-formulae *of the compound formula. To have a notation for all atoms in a formula ψ, we use the set $atoms(\psi)$. Atoms and negated atoms together are called* literals. *We will write Σ_{pv}^{ψ} to denote the propositional variables in the signature of the propositional formula ψ.*

We do want to give the above defined well-formed formula some sort of meaning. For this purpose we have to define the *semantics of propositional logic*. The goal of the semantics-definition is to resolve the truth-value of a formula, based on a mapping of the atoms which are occurring in the formula to a truth-value. We will use the values 0 and 1 to represent the truth-values *false* and *true*. This mapping is covered by the *interpretation* of a formula. Note that a formula has countable many interpretations.

Definition 2.4.2 (Interpretations of a formula)**.**
An interpretation *I is a set of propositional variables such that $I \subseteq \Sigma_{pv}$. A* partial interpretation *I_p is a pair of sets (T, F) such that $T \subseteq \Sigma_{pv}$ and $F \subseteq (\Sigma_{pv} \setminus T)$.*

The intended meaning for the interpretation is to have a set of all the variables which have been assigned the truth-value "true". The partial interpretation uses two sets to describe which variables are mapped to "true" and which are "false". Note that there can be some variables where no distinct mapping exists.

An (partial) interpretation can also be represented as a function, which maps the values *true* and *false* to the (sub-)set of propositional variables (i.e. $I : \Sigma_{pv} \mapsto \{0, 1\}$ and $I_p : PV \mapsto \{0, 1\}, PV \subseteq \Sigma_{pv}$). The latter way is the more common way, but we prefer the set approach of Definition 2.4.2 as it is more related to the set notation from the *ADFs*. To get the meaning of a compound formula under a specific interpretation we will need a method to calculate the value based on the truth assignments for the atoms. At first we will give a common definition, then we will characterize the calculation function in a more set theoretic manner.

Definition 2.4.3 (Semantics for propositional logic)**.** *The truth-value, based on an interpretation I is computed via the evaluation function V_I for the arbitrary formulae ϕ and ψ:*

(i) $V_I(p) = I(p), p \in \Sigma_{pv}$

(ii) $V_I(\top) = 1$ *and* $V_I(\bot) = 0$

(iii) $V_I(\neg\phi) = 1 - V_I(\phi)$

(iv) $V_I(\phi \wedge \psi) = min(V_I(\phi), V_I(\psi))$

(v) $V_I(\phi \vee \psi) = max(V_I(\phi), V_I(\psi))$

As already said, it is more convenient for us to stick closer to set-theoretic definitions, so we will define the truth value under a given interpretation for sets too:

Definition 2.4.4 (Semantics for propositional logic with sets)**.** *The truth-value $\mathcal{V}_I$ with respect to the interpretation I for the arbitrary formulae ψ and ϕ, where a denotes atomic formulae, is:*

(i) $\mathcal{V}_I(a)$ is true iff $a \in I$.

(ii) $\mathcal{V}_I(\top)$ is always true and $\mathcal{V}_I(\bot)$ is always false.

(iii) $\mathcal{V}_I(\neg\phi)$ is true iff ϕ is not true.

(iv) $\mathcal{V}_I(\phi \wedge \psi)$ *is true iff* ϕ *and* ψ *are both true.*

(v) $\mathcal{V}_I(\phi \vee \psi)$ *is true iff at least one of them is true.*

Any value which is not true has to be false as an interpretations captures all propositional variables.

With the semantics definition for the symbols $\neg$, $\vee$, and $\wedge$ it can be seen that these symbols are representing the negation, disjunction and conjunction. In fact we would only need the negation and one additional connective to simulate the third connective (e.g. $(\phi \wedge \psi)$ can be written as $\neg(\neg\phi \vee \neg\psi)$). This simulation can be done with the knowledge about the meaning of the different symbols. For more convenience we will use additional connectives, which should be seen as syntactic shortcuts for the semantically identical formulae. We will use the symbol $\equiv$ to represent the *semantic equivalence*. Two formulae are semantically equivalent if both formulae have the same set of models.

Definition 2.4.5 (Semantically identical syntactic shortcuts)**.** *Let* ϕ *and* ψ *be arbitrary formulae, then*

- $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$
- $\phi \veebar \psi \equiv (\phi \wedge \neg\psi) \vee (\neg\phi \wedge \psi)$
- $\phi \bar{\wedge} \psi \equiv \neg\phi \vee \neg\psi$
- $\phi \leftrightarrow \psi \equiv (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$

The above definitions allow us to identify which truth-value is represented by a formula with respect to a given interpretation. To reduce the count of parentheses we will define the strength of binding for the used connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \veebar, \bar{\wedge}$. This means without parentheses the conjunction binds stronger than the disjunction. If the value $\mathcal{V}_I(\phi)$ of the formula ϕ with the interpretation I is "true", we say that the interpretation is a propositional model ($I \in mod_p(\phi)$). In addition we will say that a formula is *satisfiable (SAT)* if it has at least one model and it is *valid (VALIDITY)* if every interpretation is a model.

Till now we have only dealt with the interpretation and the value of formulae, based on the interpretation. With the introduction of the SAT and VALIDITY problem we can also have a look on the value of formulae with respect to partial interpretations. With a partial interpretation it may happen that there are not enough truth-values assigned to the variables to get a result from the evaluation process. Indeed it can happen that there is a result (e.g. if it is known that one component of a disjunction is true, it does not matter what value the other component has).

Definition 2.4.6 (Evaluation value for partial assignments)**.** *Let* ψ *be an arbitrary formula and* $I_p = (T, F)$ *a partial interpretation for this formula. To determine the truth value, replace each* $t \in T$ *which occurs in* ψ *with* $\top$ *and each* $f \in F$ *with* $\bot$*. If the resulting formula is VALID, then* $\mathcal{V}_{I_p}(\psi)$ *is true. If it is not SAT, then* $\mathcal{V}_{I_p}(\psi)$ *is false. In case it is whether true or false, it can not be decided with the current mapped variables.*

It is obvious that the computation of the truth-value under a given partial interpretation is more complex than the computation of it under an interpretation. For the interpretation it is only needed to check the value of the compound formulae, based on the given truth-value assignment. In contrast to that it is required to check against each possible interpretation of the not assigned values for the partial interpretation. Therefore the partial interpretation can only result in *yes* or *no* if the model-check for every interpretation with respect to the partial assignment results in the same answer.

For propositional logic there exist some more properties which are interesting for us. A formula is in a *negated normal form - NNF* if all negations only occur directly in front of atoms. In general some sort of syntactical rewriting, based on semantic equivalences is used to construct a negated normal form (see Algorithm 2.4.9).

Definition 2.4.7 (DeMorgan's Laws)**.** *Let ψ and ϕ be arbitrary propositional formulae. Then the following truth-value equivalences hold:*

- $\neg(\psi \wedge \phi) \equiv (\neg\psi \vee \neg\phi)$
- $\neg(\psi \vee \phi) \equiv (\neg\psi \wedge \neg\phi)$

Definition 2.4.8 (Double Negation)**.** *Let ψ be an arbitrary propositional formula. Then the following truth-value equivalence holds:*

$$\neg\neg\psi \equiv \psi$$

Algorithm 2.4.9 (NNF-Transformation)**.** *Let ψ be an arbitrary propositional formula. To gain a* negated normal form $\mathcal{NNF}(\psi)$*, the following steps need to be done:*

(I) Apply De Morgan's Laws *till all negations only appear directly in front of atoms, then*

(II) use the Double Negation *as long as it is applicable.*

In addition we are interested in two other normal forms, namely the *conjunctive normal form - CNF* and the *disjunctive normal form - DNF*. The *CNF* is a negated normal form, where the literals are pooled together in groups (*clauses*). The clauses are connected via conjunctions and the literals inside the clauses are disjunctively connected. For the DNF the role of disjunction and conjunction is switched. The advantage of the two normal forms is the flat structure of the formula as there are not many nested connectives. The disadvantage is the potential exponential growth of the length of the formula during the transformation. Note that every formula can be transformed into a semantically equivalent formula in DNF (resp. CNF or NNF).

Definition 2.4.10. *Let ψ, ϕ, and ρ be arbitrary propositional formulae. Then the following truth-value equivalences hold:*

- $\psi \wedge (\phi \vee \rho) \equiv (\psi \wedge \phi) \vee (\psi \wedge \rho)$
- $\psi \vee (\phi \wedge \rho) \equiv (\psi \vee \phi) \wedge (\psi \vee \rho)$

Algorithm 2.4.11 (CNF-Transformation)**.** *Let ψ be an arbitrary propositional formula in NNF. Apply the distributive law such that the disjunctions move towards the atoms and the conjunctions connect these disjunctions.*

Example 2.4.12 (CNF)**.** *Suppose $\phi = a \veebar b$ and $\psi = (a \wedge b) \leftrightarrow (a \rightarrow (b \vee c))$. The formulae have the following CNF:*

$$(a \vee b) \wedge (\neg a \vee \neg b)$$

$$(\neg a \vee \neg b \vee \neg a \vee b \vee c) \wedge (a \vee a \vee \neg a \vee \neg b \vee \neg c) \wedge (b \vee a \vee \neg b \vee \neg c)$$

Due to the specified structure of a formula in *CNF* the formula can be represented in a more compact way, namely as a set of sets. This representation is the *clause form* (for a more exhaustive introduction and discussion of its properties see [Leitsch, 1997]). We will write $\mathcal{CF}(\psi)$ to represent the clause form of the formula ψ. Here the literals of one clause are represented as a set of literals and all clauses of the formula are represented as a set of clauses. One advantage of this representation is the easy readability, the elimination of multiple occurrences of a literal in one clause, and the elimination of multiple occurrences of the same clause.

The *CNF* has also some interesting properties, which are not present for arbitrary formulae. A formula in *CNF* is unsatisfiable if in any clause a contradiction exists. In addition we know that if a formula in *CNF* has an interpretation I which is a model of this formula, then I has to be a model for every clause too. In general an empty set of literals is seen to be false under every interpretation and an empty set of clauses is a tautology.

For propositional logic it is possible to resolve the truth-value of a formula with a truth-table. A truth-table has for each sub-formula of the formula in question a column. Each row is one interpretation for the propositional variables. So the truth-table shows all interpretations for the formula. In the fields the truth value to the corresponding formula with respect to the interpretation is listed.

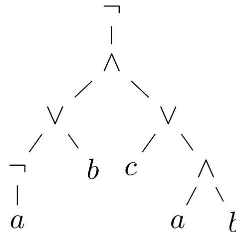
Example 2.4.13 (Truth-table)**.** *Suppose we have the formula $(\neg a \vee b) \wedge (\neg b \vee a)$ (which is semantically equivalent to $a \veebar b$). The truth-table would be:*

a	b	$\neg a$	$\neg b$	$\neg a \vee b$	$\neg b \vee a$	$(\neg a \vee b) \wedge (\neg b \vee a)$
0	0	1	1	1	1	1
0	1	1	0	1	0	0
1	0	0	1	0	1	0
1	1	0	0	1	1	1

Propositional logic has such a strong expressiveness that it is possible to find for each set of models an appropriate formula. In other words, every kind of sequence for the last column of a truth-table can be enforced by a formula.

Another representation for propositional formulae, which has the aim to show the structure of the sub-formulae, is the *formula-tree*. There the connectives are the inner nodes and the leaves are the atomic formulae.

Example 2.4.14 (Formula-tree)**.** *For the formula* $\neg((\neg a \vee b) \wedge c \vee (a \wedge b))$ *the corresponding tree is:*



Proposition 2.4.15. *Let* ψ *be a propositional formula, NNF*(ψ) *be the transformed NNF of* ψ*, and* α *be an arbitrary atom in* ψ*.* α *is a negated literal in* $nnf(\psi)$ *iff an odd number of negations occurs in the path from* α *to the root of the formula tree of* ψ*.*

Proof. To gain a negated normal form the negations need to move towards the atoms. The generally used DeMorgan rules preserve the number of negations in the path from the atom to the root. When all negations are directly above the atoms the double-negation rule will eliminate all pairs of negations. Therefore only one or no negation will be kept. As always two negations are deleted the parity of the number of negations does not change. □

Definition 2.4.16. *Let NNF*(ψ) *be an arbitrary negated normal form of the formula* ψ*. A literal* α *in a propositional formula* ψ *is said to have a* positive polarity *iff the literal is positive in the corresponding NNF*(ψ)*.*
A literal α *in a propositional formula* ψ *is said to have a* negative polarity *iff the literal is negated in the corresponding NNF*(ψ)*.*

Note that the above proposition and definition also hold for the CNF and DNF.

2.5 Complexity Theory

In this section we give an overview on complexity theory and the common complexity classes, which will be interesting for *ADFs*. A brief overview is given by Johnson [1992] and for an in-depth insight in complexity theory we refer to the book by Papadimitriou [1994].

In complexity theory we want to understand and show how complex the process to find a solution for a problem is. These problems are defined by an input description and a question to be answered. We will deal mostly with decision problems. There the question is formulated to get a “yes or no” answer. The complexity of such a problem is given by a function which is only dependent on the input and the method to solve the problem. The different types of functions are pooled together in so-called *complexity classes*.

One of the most important complexity class is $\mathbf{P}$. It is defined on a deterministic universal turing machine and is a class for decision problems.

Definition 2.5.1 (Complexity class $\mathbf{P}$)**.** *A problem P is in $\mathbf{P}$ if it can be solved by a deterministic universal turing machine in polynomial many working steps, with respect to the length of the input string.*

In other words a problem in $\mathbf{P}$ needs only polynomially longer to be computed than the length of the input string was. One example for a problem in $\mathbf{P}$ is the decision problem whether an interpretation I for a formula ϕ is a propositional model $I \in mod_p(\phi)$ or not.

Definition 2.5.2 (Complexity class $\mathbf{NP}$)**.** *A problem P is in $\mathbf{NP}$ if it can be solved by a non-deterministic universal turing machine in polynomial many working steps, with respect to the length of the input string.*

For $\mathbf{NP}$ we have another underlying working mode, namely the non-deterministic universal turing machine. Intuitively this means if we would try to solve the problem with deterministic turing machines, we would have to clone the machine at each step where a decision is done. For each decision path another turing machine is working and the problem is solved when one of the machines can answer the question in polynomial time. An example for a problem in the $\mathbf{NP}$ class is the SAT problem for propositional formulae (i.e. is a given formula ϕ satisfiable or not). Note that it is unknown if there exists an efficient way to compute problems which are in $\mathbf{NP}$, but current solutions to solve $\mathbf{NP}$ problems with deterministic methods take exponential time.

We will say a problem is $\mathbf{NP}$-*complete* if we know that the problem has a *membership* in this class and that the problem is $\mathbf{NP}$-*hard*.

Definition 2.5.3 (Membership and hardness)**.** *A problem P has a membership in a complexity class C, if an algorithm exists, whose complexity function is in the class of C.*
A problem P is said to be C-hard for a complexity-class C, if a program Π exists, which transforms P' to P, where P' is known to be a C-hard problem, and the answer to P' equals the answer to P. Additionally the complexity of Π must not be greater than $\mathbf{P}$.

One method to show the $\mathbf{NP}$-*membership* is to use a *"guess & check"* algorithm. This algorithm will guess a solution and afterwards checks whether the solution is correct or not. Note that the algorithm only checks one guess. If this *"guess & check"* algorithm has a polynomial runtime (i.e. has a $\mathbf{P}$ membership), then the problem is in $\mathbf{NP}$.

For each non-deterministic problem class a *complement class* exists (e.g. $\mathbf{coNP}$). There all answers are the complement of the original problem. One example of two complement problems is the SAT and the UNSAT problem. Note that the two complement problems can have a different difficulty to solve: To answer the SAT question, it is only needed to test the interpretations till one is a model, but to check for the UNSAT answer every interpretation must be tested to show that no interpretation is a model.

On top of the classes $\mathbf{P}$, $\mathbf{NP}$, and $\mathbf{coNP}$, we can now define one additional type of classes. We will use so-called *oracles*. Let us assume we have an oracle which can solve a problem in a complexity class with a constant computational effort of one unit of time. If we use such an oracle in our program the overall complexity of the program without the oracle would be higher than with the oracle. In case we have an program in $\mathbf{P}$ and an oracle which solves a problem in $\mathbf{NP}$, we would have the complexity class $\mathbf{P}^{\mathbf{NP}}$. Based on this notation for algorithms with oracles we can build the *polynomial hierarchy*.

Definition 2.5.4 (Polynomial hierarchy)**.**

$$\begin{aligned}\Delta_0\mathbf{P} = \Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}&\textit{; and for all } i \geq 0 : \\ \Delta_{i+1}\mathbf{P} &= \mathbf{P}^{\Sigma_i\mathbf{P}} \\ \Sigma_{i+1}\mathbf{P} &= \mathbf{NP}^{\Sigma_i\mathbf{P}} \\ \Pi_{i+1}\mathbf{P} &= \mathbf{coNP}^{\Sigma_i\mathbf{P}}\end{aligned}$$

To get a better understanding on the hierarchy, Figure 2.2 is sketching the relations of the different classes. In addition a less detailed notion for computational complexity exists for computational problems. Here the problems are assigned to be *tractable* or *intractable*. In general all problems in $\mathbf{P}$ are said to be *tractable* problems while all problems which are in a higher complexity class are *intractable* computational problems.

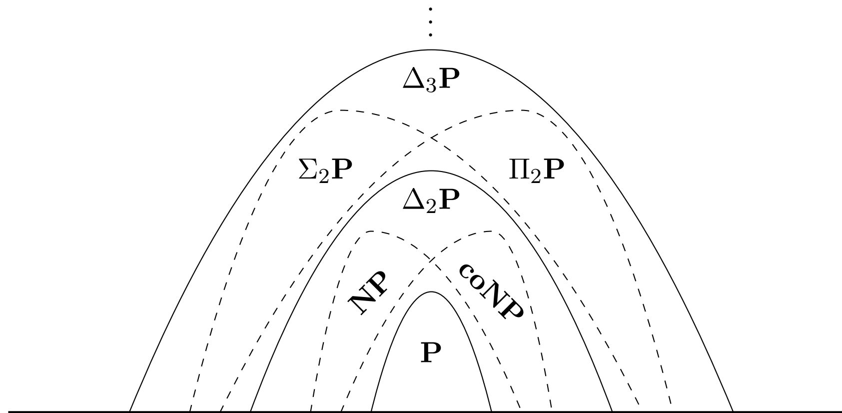


Figure 2.2: Relations of the classes in the polynomial hierarchy

Note that it is not known whether the polynomial hierarchy is a correct assumption or not. Indeed the question whether $\mathbf{P} \neq \mathbf{NP}$ holds or not is still not answered. Anyway, we use the notion of the polynomial hierarchy as it is assumed in the scientific community that it holds till something different is proven.

CHAPTER 3

Alternative Representations of ADFs

This chapter will emphasize on alternative representations for *ADFs*. The first alternative is based on the idea that the acceptance conditions for the statements are in fact binary functions and therefore it is possible to represent them as propositional formulae, which was only a short, not advocated side note in [Brewka and Woltran, 2010]. The idea was used to construct a transformation [Brewka et al., 2011a], but it was never investigated further. We will rewrite all important definitions and show that both representations are indeed equal. In addition we will also discuss a representation for *ADFs* as a hypergraph structure to have a stronger connection to the graph-like representation of *Dung's AFs*.

3.1 ADFs with Propositional Formulae

We will try to characterize the already introduced *ADFs* (see Section 2.3) with another representation for the acceptance condition. In the basic definition for *ADFs* we have used the links between statements to define that they are in a relation. Based on the relation it is possible to identify all parents of one statement and the acceptance condition defines for each subset of parents whether the statement is *in* or *out*. These two values for the subsets determine which combinations of accepted parents will lead to the acceptance or non-acceptance of the statement. We will now redefine the notion of an acceptance condition as well as the *ADF* to adjust it for propositional formulae.

Propositional formula ADF

Definition 3.1.1 (ADF with a propositional formula as acceptance condition)**.**
An ADF with propositional formula acceptance conditions *is a tuple* $D = (S, L, AC)$ *where*

- S *is a set of statements*
- $L \subseteq S \times S$ *is a set of links*

- $AC = \{AC_s\}_{s \in S}$ *is the set of acceptance conditions.*

Definition 3.1.2 (acceptance condition)**.** *Let $D = (S, L, AC)$ be an* ADF *with propositional acceptance conditions, then $AC_s = \psi$ and ψ is a propositional formula, where $(\Sigma_{pv}^s = par(s)) \subseteq S$ and $\forall_{a \in par(s)} a \in \Sigma_{pv}^s$.*

The new *acceptance condition* has only one propositional formula instead of the binary total function, which has to be defined for $2^{par(s)}$ different inputs. We will use the acceptance of the parents for the interpretation. To relate both representations it is important to see the value *in* as *true* and *out* as *false*. To be sure that every relation to the parents is represented we demand that each parent occurs as an atom in the formula. In addition we want to be sure that only the parents will take account to the truth-value of the formula, so we restrict the propositional variables to the set of parents.

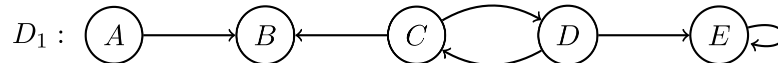
Definition 3.1.3. *An* ADF *$D = (S, L, C)$ is equivalent to an ADF with propositional formulas as acceptance conditions $D' = (S', L', AC)$ iff $S = S'$, $L = L'$, and for every $s \in S$ and every $M \subseteq par(s) : C_s(M) = in$ iff $M \in mod_p(AC_s)$ holds.*

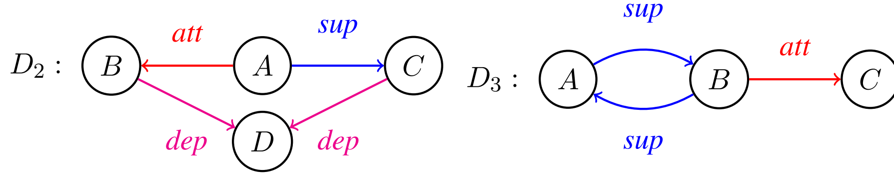
Proposition 3.1.4. *For every* ADF *$D = (S, L, C)$ a* ADF with propositional formula acceptance conditions *$D' = (S', L', AC)$ can be constructed such that D and D' are equivalent, and for every* ADF with propositional formula acceptance condition *$D' = (S', L', AC)$ an* ADF *$D = (S, L, C)$ can be constructed such that D' and D are equivalent.*

Proof. Both are only equivalent if the statements and the links are the same, so only the total functions and the propositional formulas have to be analyzed deeper. The input for the total functions is determined by the acceptance of the parent statements and possible inputs are all subsets of the set of parents. The interpretation of a propositional formula is defined as a subset of all atoms in the formula. As only parent statements are allowed to occur as atoms all interpretations of a propositional formula are the same sets as the possible inputs for the total function and vice versa. The total function defines for each input value an output value. In fact it is the same as a truth-table for the propositional logic. As a propositional formula can express every possible set of truth-values in a truth-table the appropriate formula can express the same as the function. In the other way each formula has a set of interpretations, where some of them are a model. So the total function needs to set all sets which correspond to a model to in and the others to out. □

As the construction can be done in both ways the expressiveness of the two representations has to be equal.

Example 3.1.5 (Some *ADFs* represented with propositional formula acceptance conditions)**.** *We will use the* ADFs *D_1, D_2, and D_3, we have already used before (see Examples 2.3.3, 2.3.4, and 2.3.9):*





These are the corresponding acceptance conditions:

D_1	D_2	D_3
$AC_A = \top$	$AC_A = \top$	$AC_A = B$
$AC_B = \neg A \wedge \neg C$	$AC_B = \neg A$	$AC_B = A$
$AC_C = \neg D$	$AC_C = A$	$AC_C = \neg B$
$AC_D = \neg C$	$AC_D = B \veebar C$	
$AC_E = \neg E \wedge \neg D$		

The example shows that we have to use the propositional constants $\bot$ and $\top$, if the statement has no parents. There we choose according to the basic definition whether it shall be always *in* or *out*. In addition it is obvious that the propositional formulae are easier to read and more compact than the total functions (to compare them see Examples 2.3.3, 2.3.4, 2.3.9, and 3.1.5).

With the representation utilizing the propositional formula acceptance conditions, the information from the links becomes redundant. Every variable which occurs in the formula has to be a parent of the associated statement. In addition the links do not support us with specific information about the link type, as this has to be checked with the acceptance condition. Therefore we can omit the links and define the *propositional formula ADF (pForm-ADF)*.

Definition 3.1.6 (pForm-ADF)**.** *A pForm-ADF is a pair $D = (S, AC)$, where*

- *S is a set of statements*
- *$AC = \{AC_s\}_{s \in S}$ is the set of acceptance conditions, where each statement has exactly one associated condition.*

An acceptance condition AC_s is a propositional formula ψ, where $\Sigma^s_{pv} \subseteq S$.

One advantage of *pForm-ADFs* is the possibility to rewrite sub-formulae into semantically equivalent propositions. Some links may not have an impact to the decision whether a statement is accepted or not (e.g. $a \vee \neg a$, which is a tautology and has the same expressiveness as $\top$). These rewritings can reduce the complexity of formulae and be a method to analyze whether a link is meaningful or not. This kind of simplified rewriting of formulae indeed removes some kind of information, as a link may be removed by the removal of tautologies or absorptions. Although informations are removed, these changes have no impact on the introduced semantics as the relation between these two arguments is that they have no influence on each other.

Note that this is a simplification for the standard representation of *ADFs*. There are already variants of *ADFs* existing, which are utilizing weights or other forms of additional information

about the links. In these cases it is possible to add an additional structure for the extra information to the *pForm-ADF* or to reuse the omitted notion of links. Another workaround could be to remove the requirement that each variable in the signature occurs in the formula. So the signature would capture the information of the links and the formula itself contains the informations about the relations.

Propositional Formula BADF

We already said that the *pForm-ADF* has the same expressiveness as the original representation. We can use the definition for *BADFs* from the existing representation without changes, but we need to revamp the definition for a link and how its type is decided. In addition we will present a class of formulae where the decision can be done via a syntactic procedure.

Definition 3.1.7 (links and link types for *pForm-ADFs*)**.** *Let $D = (S, AC)$ be a* pForm-ADF, *where $a, b \in S$. A link (a, b) between a and b such that a is in a relation with b is present if $a \in \Sigma^b_{pv}$.*
The link (a, b) is attacking in D iff for no $I \subseteq (\Sigma^b_{pv} \setminus \{a\})$ the following holds:

$$I \notin mod_p(AC_b)$$
$$I \cup \{a\} \in mod_p(AC_b)$$

The link (a, b) is supporting in D iff for no $I \subseteq (\Sigma^b_{pv} \setminus \{a\})$ the following holds:

$$I \in mod_p(AC_b)$$
$$I \cup \{a\} \notin mod_p(AC_b)$$

We say a is attacking (resp. supporting) in D w.r.t. b. $att(AC_b)$ (resp. $sup(AC_b)$) denotes the set of all attacking (resp. supporting) links in D w.r.t. b.

The identification of the link-type is already shown to be intractable (see Section 4.4 for more details), so it is desirable to find a specific subclass of propositional formulae, where the link-type distinction is tractable. In addition this class shall be general enough to gain the same expressiveness as the class of bipolar *ADFs*. At first we will project the attacking and supporting notion to variables in propositional formulae to have a stronger focus on the formulae (which are the acceptance conditions).

Definition 3.1.8 (Attacking and supporting variable)**.** *Let ψ be a propositional formula. A variable $a \in \Sigma^\psi_{pv}$ is*

- *attacking if there exists no $I \subseteq \Sigma^\psi_{pv}$, such that $I \notin mod_p(\psi)$ and $I \cup \{a\} \in mod_p(\psi)$, and*
- *supporting if there exists no $I \subseteq \Sigma^\psi_{pv}$, such that $I \in mod_p(\psi)$ and $I \cup \{a\} \notin mod_p(\psi)$.*

The notion of attacking and supporting variables will identify the “role” of each variable in the formula. If we use the formula as an acceptance condition the “role” of each variable is in a one to one correspondence to the link-type of the variable (i.e. the link to another statement in the *ADF*).

We have already mentioned that there may exist relations between two statements which have no impact on each other (e.g. tautologies and contradictions in the acceptance condition formulae). These variables, which can be omitted without loss of information (see Proposition 3.1.12) can not impact the truth-value of the formulae. So we introduce the notion of an *informative variable and formula* and the complement concept of an *uninformative variable and formula*. Intuitively an informative variable is a variable which provides information to the formulae in some kind such that there exists an interpretation where the addition of this informative variable to the interpretation changes the truth-value of the formula.

Definition 3.1.9 (Informative variable)**.** *Let ψ be a propositional formula. A variable $a \in \Sigma_{pv}^{\psi}$ is* informative *in ψ if there exists an interpretation $I \subseteq \Sigma_{pv}^{\psi}$ such that $\mathcal{V}_I(\psi) \neq \mathcal{V}_{I\cup\{a\}}(\psi)$. It is* uninformative *if it is not informative.*

Corollary 3.1.10. *A variable $a \in \Sigma_{pv}^{\psi}$ of the propositional formula ψ is informative iff a is attacking, supporting, or neither attacking nor supporting. A variable $a \in \Sigma_{pv}^{\psi}$ of the propositional formula ψ is uninformative iff a is attacking and supporting at the same time.*

Proof. Follows by Definition 3.1.9 and Definition 3.1.8. A variable a in an arbitrary formula ψ is informative iff there exists a model $I \subseteq \Sigma_{pv}^{\psi}$ such that the truth-value changes between I and $I \cup \{a\}$. If such a change exists it is either a counter example for the fact that the variable is attacking or supporting. So a variable needs to be attacking or supporting, but it cannot be attacking and supporting. If a second interpretation $J \subseteq \Sigma_{pv}^{\psi}$ exists, such that $I \neq J$, $\mathcal{V}_I(\psi) \neq \mathcal{V}_J(\psi)$, and J is again an attacking or supporting counter example, then a is neither attacking nor supporting, but still informative. □

Definition 3.1.11 (Informative propositional formula)**.** *A propositional formula ψ is said to be* informative *if every variable a in ψ is informative.*

Proposition 3.1.12. *Every uninformative propositional formula ψ can be transformed to an informative propositional formula by substituting all not informative variables with the constant $\bot$, without changing the set of models.*

Proof. Let ψ be a propositional formula which is not informative. Suppose $a \in \Sigma_{pv}^{\psi}$ is an uninformative variable, then $\mathcal{V}_{I\setminus\{a\}}(\psi) = \mathcal{V}_{I\cup\{a\}}(\psi)$ holds for every interpretation $I \subseteq \Sigma_{pv}^{\psi}$. With the substitution of all occurrences of the variable a in ψ with the constant $\bot$, we do the same as the truth-value evaluation process for $\mathcal{V}_{I\setminus\{a\}}(\psi)$ does. So the set of models for the formula ψ remains the same. Through the removal of the variable a the link disappears and so the uninformative link is no longer existent. □

Definition 3.1.13 (Bipolar propositional formula)**.** *A propositional formula ψ is a bipolar propositional formula if every variable is supporting or attacking.*

Now we will connect the approach which has its focus on formulae and the *pForm-ADFs* such that both notions can be used to identify whether a *pForm-ADF* is bipolar or not.

Corollary 3.1.14. *A* pForm-ADF $D = (S, AC)$ *is bipolar if every statement* $s \in S$ *has a bipolar propositional formula as its acceptance condition* AC_s*.*

Proof. Follows from the Definitions 2.3.6, 3.1.7, 3.1.8 and 3.1.13. □

In the following we will introduce the class of *monotone propositional formulae*, which restricts the usage of the polarity of variables. There it is not allowed that one variable occurs with a positive and a negative polarity in the same formula. As a natural further step we also introduce the *monotone pForm-ADFs*, where every acceptance condition needs to be a monotone propositional formula. The idea behind this representation is, that we will be able to draw conclusions on the link types of variables based on the knowledge of their polarity (see Proposition 3.1.19).

Definition 3.1.15 (Monotone propositional formula)**.** *A propositional formula* ψ *is a monotone propositional formula if no variable occurs as a positive and negative polarity literal in the formula.*

Definition 3.1.16 (Monotone pForm-ADF)**.** *Let* $D = (S, AC)$ *be a* pForm-ADF*.* D *is called a* monotone pForm-ADF *if every statement* $s \in S$ *has an acceptance condition* AC_s *which is a monotone propositional formula.*

Algorithm 3.1.17 (Transformation of bipolar informative propositional formulae to monotone propositional formulae)**.** *Let* ψ *be a bipolar informative propositional formula represented in clause form. Remove all clauses which contain the same variable as a positive and a negative literal. Then do the following substitutions:*

- *If the variable* $a \in \Sigma_{pv}^{\psi}$ *is attacking, then remove all positive literals "*a*" from the clauses in* ψ*.*
- *If the variable* $a \in \Sigma_{pv}^{\psi}$ *is supporting, then remove all negative literals "*$\neg a$*" from the clauses in* ψ*.*

Proposition 3.1.18. *The transformation of a bipolar informative propositional formula* ψ *in clause form to a monotone propositional formula in clause form* ψ' *does not change the set of models, i.e.* ψ *and* ψ' *have the same set of models.*

Proof. Let ψ be a bipolar propositional formula in clause form, $a \in \Sigma_{pv}^{\psi}$ be a propositional variable of ψ, and $I \subseteq \Sigma_{pv}^{\psi}$ be an interpretation of ψ. The removal of a clause which is a tautology does not change the set of models as the truth-value of the clause is always true. ψ is bipolar and informative, so a can only be supporting or attacking. This holds for every variable, therefore it is trivial that no variable can occur as a positive and a negative literal after the transformation.
Now we distinct how the truth-values under some interpretation $I \subseteq \Sigma_{pv}^{\psi}$ behaves and then we will show that the transformation will not change the truth-value behavior of the overall formula under the interpretation. We let $a \notin I$ to argue about all interpretations where a is not in the set of variables. To capture all sets, we also investigate the interpretations $I \cup \{a\}$.

(I) Assume a is supporting. Choose one arbitrary clause $c \in \mathcal{CF}(\psi)$, where $\{\neg a\} \subseteq c$ and remove the $\neg a$-literal from it such that $c_n = c \setminus \{\neg a\}$ holds. We can distinguish between three cases which can occur for the clause:

(i) $\mathcal{V}_I(c) = \mathit{false}$: This state is not possible, because the literals in c are connected via disjunctions and $\{\neg a\} \subseteq c$ holds, so $\mathcal{V}_I(c) = \mathit{true}$ has to hold.

(ii) $\mathcal{V}_I(c) = \mathit{true}$ and $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{true}$: The substitution of c by c_n will not change the truth value, as $\mathcal{V}_I(c_n) = \mathcal{V}_{I\cup\{a\}}(c_n) = \mathcal{V}_I(c)$. $\mathcal{V}_I(c_n) = \mathcal{V}_I(c)$ holds because $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{true}$ and therefore there needs to be another literal except $\neg a$ in c which evaluates to true under I. So the formula still evaluates to true if we remove $\neg a$.

(iii) $\mathcal{V}_I(c) = \mathit{true}$ and $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{false}$: $I \in \mathit{mod}_p(\psi)$ and $I \cup \{a\} \notin \mathit{mod}_p(\psi)$ is a contradiction to the assumption that a is supporting, so there needs to be an $c' \in (\mathcal{CF}(\psi) \setminus \{c\})$ such that $\mathcal{V}_I(c') = \mathit{false}$. Therefore we know that $I \notin \mathit{mod}_p(\psi)$ and $I \cup \{a\} \notin \mathit{mod}_p(\psi)$ has to hold. Since $\mathcal{V}_I(c) = \mathit{true}$ and $\mathcal{V}_{I\cup\{a\}} = \mathit{false}$ holds, $\neg a$ is the only reason for c to be true under I. Therefore $\mathcal{V}_I(c_n) = \mathcal{V}_{I\cup\{a\}}(c_n) = \mathit{false}$.

(II) Assume a is attacking. Choose one arbitrary clause $c \in \mathcal{CF}(\psi)$, where $\{a\} \subseteq c$ and remove the a-literal from it such that $c_n = c \setminus \{a\}$ holds. We can distinct between three cases which can occur for the clause:

(i) $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{false}$: This state is not possible, because the literals in c are connected via disjunctions and $\{a\} \subseteq c$ holds, so $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{true}$ has to hold.

(ii) $\mathcal{V}_I(c) = \mathit{true}$ and $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{true}$: For this interpretation the variable a does not have an impact on the truth-value. So the substitution of c by c_n will not change the truth value, as $\mathcal{V}_I(c_n) = \mathcal{V}_I(c)$.

(iii) $\mathcal{V}_I(c) = \mathit{false}$ and $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{true}$: $I \notin \mathit{mod}_p(\psi)$ and $I \cup \{a\} \in \mathit{mod}_p(\psi)$ is a contradiction to the assumption that a is attacking, so there needs to be an $c' \in (\mathcal{CF}(\psi) \setminus \{c\})$ such that $\mathcal{V}_{I\cup\{a\}}(c') = false$. Therefore we know that $I \notin \mathit{mod}_p(\psi)$ and $I \cup \{a\} \notin \mathit{mod}_p(\psi)$ has to hold. Since $\mathcal{V}_I(c) = \mathit{false}$ and $\mathcal{V}_{I\cup\{a\}}(c) = \mathit{true}$ holds, a is the only reason for c to be true under $I \cup \{a\}$. Therefore $\mathcal{V}_I(c_n) = \mathcal{V}_{I\cup\{a\}}(c_n) = \mathit{false}$.

As one removal does not change the set of models every additional removal will not change the models neither. □

Proposition 3.1.19. *Let ψ be a monotone propositional formula and $a \in \Sigma_{pv}^{\psi}$ be a variable of ψ, then the following holds:*

(I) If a has a positive polarity, a has to be a supporting variable and

(II) if a has a negative polarity it has to be an attacking variable.

Proof. Let ψ be a monotone propositional formula and assume without loss of generality that ψ is in $\mathcal{NNF}$. In addition assume that ψ is represented as a formula tree. So negations only occur

as direct parents of the leaves. Therefore no negation occurs on the path from a variable with positive polarity and exactly one negation occurs on the path from a negative polarity variable.

(I) Suppose $a \in \Sigma_{pv}^{\psi}$ has a positive polarity and a is not a supporting variable. Then an interpretation $I \subseteq \Sigma_{pv}^{\psi}$ exists, such that $I \in mod_p(\psi)$ and $(I \cup \{a\}) \notin mod_p(\psi)$. a occurs only as a positive literal in ψ, therefore on each path from one occurrence of a to the root of the formula tree no negation occurs. As $I \in mod_p(\phi)$ holds, we know that the truth-value $\mathcal{V}_I(a) = \mathit{false}$ is not propagated to the root of the formula tree. By $I \cup \{a\} \notin mod_p(\phi)$ we need a subformula whose truth-value is changed from true to false by the addition of a to the interpretation. As $\mathcal{V}_{I\cup\{a\}}(a) = \mathit{true}$ and no negation occurs on the path from a to the root, no such truth value change to false can be caused by a. Therefore no such interpretation I can exist and a has to be supporting.

(II) Suppose $a \in \Sigma_{pv}^{\psi}$ has a negative polarity and a is not an attacking variable. Then an interpretation $I \subseteq \Sigma_{pv}^{\psi}$ exists, such that $I \notin mod_p(\psi)$ and $(I \cup \{a\}) \in mod_p(\psi)$. a occurs only as a negative literal in ψ, therefore on each path from one occurrence of a to the root of the formula tree exactly one negation occurs. As $I \notin mod_p(\phi)$ holds, we know that the truth-value $\mathcal{V}_I(\neg a) = \mathit{true}$ is not propagated to the root of the formula tree. By $I \cup \{a\} \in mod_p(\phi)$ we need a subformula whose truth-value is changed from false to true by the addition of a to the interpretation. As $\mathcal{V}_{I\cup\{a\}}(\neg a) = \mathit{false}$ and exactly this one negation occurs on the path from a to the root, no such truth value change to true can be caused by a. Therefore no such interpretation I can exist and a has to be attacking.

□

Note that the proposition only identifies that a positive polarity variable is supporting and a negative polarity variable is attacking. It is still possible that the variable is both, attacking and supporting. So we can not check with that syntactical trick whether a variable is informative or not.

Corollary 3.1.20. *Each variable $a \in \Sigma_{pv}^{\psi}$ in a monotone propositional formula ψ is attacking or supporting.*

Proof. Every variable a in a monotone propositional formula ψ has either a positive or negative polarity (Definition 3.1.15). By Proposition 3.1.19 the Corollary follows. □

Corollary 3.1.21. *Every* monotone pForm-ADF $D = (S, AC)$ *is a* BADF.

Proof. Follows by Definition 2.3.6, Corollary 3.1.14, and Corollary 3.1.20. □

Definition 3.1.22 (*pForm-ADF* equivalence)**.** *Let $D = (S, AC)$ and $D' = (S', AC')$ be* pForm-ADFs. *D is equivalent to D' if $S = S'$ and for all $I \subseteq S$ the following holds:*

$$\forall s \in S : I \in mod_p(AC_s) \iff I \in mod_p(AC'_s)$$

Theorem 3.1.23. *Each* pForm-BADF *can be transformed to an equivalent* monotone pForm-ADF *and every* monotone pForm-ADF *is a* pForm-BADF.

Proof. That each *pForm-BADF* can be transformed to an equivalent *monotone pForm-ADF* is shown by Proposition 3.1.18 and that every *monotone pForm-ADF* is a *pForm-BADF* is shown by Corollary 3.1.21.
Note that an *ADF* which is not a *BADF* can not be a *monotone pForm-ADF*. By the definition of *BADFs* (see Definition 2.3.6), an *ADF* is not a *pForm-BADF* if it has a link which is neither attacking nor supporting. Corollary 3.1.20 shows that each variable is attacking or supporting and so no such link can exist in a *monotone pForm-ADF*. □

Table 3.1 tries to connect the existing *ADF* representation with the *pForm-ADF* representation for a better understanding on the relation between attacking and supporting links. The topmost section of the table shows some kind of truth table. This one omits specific formulae which are analyzed and only enumerates in each column one possible pattern of models. In the bottom section of the table in each column an example for a propositional formula with this set of models is given. On the left side of the top section the standard enumeration of the possible truth assignments to the atoms is given, together with a representation as a set-theoretic interpretation. In the middle-section of the table the variables are checked against the definition of supporting and attacking links. The rows *"attacking ce"* and *"supporting ce"* will show one counter-example (if it exists) on which the attacking respectively supporting link property is not satisfied. The counter-example states exactly one interpretation $\mathcal{I}$. This means that the addition of the variable a (resp. b) will result in a change of the acceptance condition truth-value such that the property for attack or support is no longer satisfied for the link to a (resp. b). Finally the row *link-type* is summarizing the outcome of the counter-examples and gives an overview about the final link type of the variable.

In addition the table illustrates that not informative links may be removed without any impact to the set of models. In the examples it is also realized that each *BADF* is already represented as a *monotone pForm-ADF* and only the formulae with dependent links are not represented in this class (as it is not possible). Note the side-effect that the formulae which are not represented as monotone formulae are already the result of the monotone transformation procedure if we use them on the non-*BADFs*.

Semantics

We have shown that the *pForm-ADFs* are another representation for *ADFs*. In the prior section we have defined the *pForm-ADFs* and revamped some of the underlying definitions for *BADFs* to have a more natural approach via the new representation. Additionally we presented a subclass where deciding some properties of the link types is trivial. To introduce an alternative to the *ADFs* with total functions, we have to reformulate the definitions for the semantics too. We will present these alternative definitions in this section in the following two parts:

At first we will investigate the semantics which are defined on the set of all *ADFs* without restrictions. Then we introduce the semantics which only work for *BADFs*. In general an addition part about the *monotone pForm-ADFs* can be presented, but the semantics do have the same definitions for *pForm-BADFs* and *monotone pForm-ADFs*. In fact there are only differences in the complexity of the semantics with respect to the link type detection.

a	b	$\mathcal{I}$	possible model-patterns for $\mathcal{I}$															
0	0	∅	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	{b}	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	{a}	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	{a,b}	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
a		attack ce	-	b	∅	b	-	-	∅	∅	-	b	-	b	-	-	-	-
		support ce	-	-	-	-	b	-	b	-	∅	∅	-	-	∅	∅	b	-
		link type	as	s	s	s	a	as	d	s	a	d	as	s	a	a	a	as
b		attack ce	-	a	-	-	∅	∅	∅	∅	-	a	-	-	-	a	-	-
		support ce	-	-	a	-	-	-	a	-	∅	∅	∅	∅	-	-	a	-
		link type	as	s	a	as	s	s	d	s	a	d	a	a	as	s	a	as
prop. formula			$\bot$	$a \wedge b$	$a \wedge \neg b$	a	$\neg a \wedge b$	b	$(a \vee b) \wedge (\neg a \vee \neg b)$	$a \vee b$	$\neg(a \vee b)$	$(a \vee \neg b) \wedge (\neg a \vee b)$	$\neg b$	$a \vee \neg b$	$\neg a$	$\neg a \vee b$	$\neg(a \wedge b)$	$\top$

Table 3.1: Relation between propositional models and attacking/supporting links for two variables

Semantics on *pForm-ADFs*

Here we will have our focus on the *conflict-free* property as well on the *model* and the *well-founded model*. To get the *conflict-free* definition for *pForm-ADFs* we need to use the equivalence between the total functions and the propositional formulae, defined by Definition 3.1.3. So we get the adapted definition for the *conflict-free* property on *pForm-ADFs*.

Definition 3.1.24 (conflict-free set for *pForm-ADFs*)**.** *Let* $D = (S, AC)$ *be a* pForm-ADF. *A set* $M \subseteq S$ *is* conflict-free *if for all* $s \in M : M \in mod_p(AC_s)$ *holds.* $cf_{pADF}(D)$ *is the set of all* conflict-free sets *for* D*.*

The adaption for the model can be done in the same manner. In addition we have substituted the two implications with an iff to shorten the definition and make it more intuitive.

Definition 3.1.25 (model for *pForm-ADFs*)**.** *Let* $D = (S, AC)$ *be a* pForm-ADF. $M \subseteq S$ *is a model of* D *if for each* $s \in S$, $M \in mod_p(AC_s)$ *iff* $s \in M$*, holds.* $model_{pADF}(D)$ *is the set of models for the* pForm-ADF D*.*

At last we will have to take a further look on the *well-founded model*. This semantics is based on the operator $\Gamma_D(A, R)$ (see Definition 2.3.16). The idea behind Γ_D is to collect all statements which are accepted (resp. rejected) for sure. Iterated application of the operator on the collected set of statements results in a fixed-point which represents the set of statements

which are accepted (resp. rejected) regardless the selection of any other statement which is not known to be selected or rejected without doubt. Based on the equivalence between the notions of the model and the binary function value *in* and the definition of Γ_D for *ADFs*, it is easy to see that the concept of the partial interpretation reflects the condition that regardless of other assumptions the value is always *in* respectively *out*. So the definition can be adapted as follows:

Definition 3.1.26 (well-founded model for *pForm-ADFs*)**.** *Let $D = (S, AC)$ be a* pForm-ADF. *Consider the operator*

$$\Gamma_D(A, R) = (acc(A, R), rej(A, R)),$$

where $acc(A, R) =$

$$A \cup \{s \in (S \setminus (A \cup R)) \mid \mathcal{V}_{(A,R)}(AC_s) = \mathit{true}\},$$

and $rej(A, R) =$

$$R \cup \{s \in (S \setminus (A \cup R)) \mid \mathcal{V}_{(A,R)}(AC_s) = \mathit{false}\}.$$

Γ_D is monotonic in both arguments and thus has a least fixed-point. E is the well-founded model of D iff for some $E' \subseteq S$, (E, E') is the least fixed-point of Γ_D.

Semantics on *pForm-BADFs*

Now let us have a look on the remaining semantics, namely the *stable model* together with the *admissible set* and the *preferred model* which are defined on top of the *stable model*. The definition of the stable model for *BADFs* checks the stability of models. This check is done by the monotone operator $Th_D(M)$ which is applied to a reduction of the *ADF* based on the specific model. Only if the least fixed-point of this operator has the same set as the model, it is stable. At first we need to adapt the operator, which is already done with the knowledge of Definition 3.1.3:

$$Th_D(M) = \{s \in S \mid M \in mod_p(AC_s)\}$$

The reduction (defined in Definition 2.3.10) is done by removing all statements which are not in the currently checked model. Then all attacking links are removed from the set of links. In the last step all acceptance condition mappings are removed which contain a removed statement or refer to a parent which is no longer in relation (the removal of the links restricts the set of parents). To get the same behavior for the *pForm-ADFs*, we will not remove the variables which shall be removed by the last step. Instead we will substitute the variables with $\bot$.

Definition 3.1.27 (stable model for *pForm-ADFs*)**.** *Let $D = (S, AC)$ be a* pForm-BADF. *A model M of D is a* stable model *if M is the least model of the reduced* pForm-ADF $D^M = (S^M, AC^M)$ *obtained from D by*

(I) eliminating all nodes not contained in M, such that $S^M = S \cap M$,

(II) for all $s \in S^M$ substitute in AC_s all $a \in \Sigma_{pv}^{AC_s}$ with $\bot$ if $a \notin S^M$, to gain AC'_s,

(III) for all $s \in S^M$ substitute in AC'_s all $a \in \Sigma_{pv}^{AC'_s}$ with $\bot$ if $a \in att(AC_s)$, to gain AC_s^M.

$smodel_{pADF}(D)$ *is the set of all stable models of* D.

Lemma 3.1.28. *Let* $D = (S, L, C)$ *be an* ADF*,* $D' = (S, AC)$ *be a* pForm-ADF*, such that* D *is equivalent to* D'*, and* $s, t \in S$ *be two statements in the* ADFs D *and* D'*. The removal of all mappings in* C_s*, where* t *occurs results in the same acceptance condition as it is represented by* AC_s*, where each occurrence of* t *is substituted with* $\bot$.

Proof. Each mapping in the *ADF* acceptance condition C_s, where t occurs is in the *pForm-ADF* representation an interpretation I, where $t \in I$ holds. If the mapping of the acceptance condition is *out* then $I \notin mod_p(AC_s)$ and if it is *in* then $I \in mod_p(AC_s)$. The removal of all acceptance conditions where t occurs equals the removal of all interpretations which contain t. In every interpretation $J \subseteq (S \setminus \{t\})$, the truth value of t is mapped to $\bot$. Therefore the substitution of t with the constant $\bot$ will result in the same acceptance of s. □

Corollary 3.1.29. *Definition 2.3.10 and 3.1.27 accept the same set of models for two equivalent* ADFs*, where one is represented by binary functions and the other is a* pForm-ADF.

Proof. Follows by Definition 3.1.3 and Lemma 3.1.28. □

In the same manner as we have adapted the *stable model* we can alter the *admissible set*, as no new mechanics are used for them.

Definition 3.1.30 (admissible set for *pForm-ADFs*)**.** *Let* $D = (S, AC)$ *be a* pForm-BADF. $M \subseteq S$ *is* admissible *in* D *iff there is* $R \subseteq S$ *such that*

(I) for each $s \in M$ *holds, that for each* $r \in R : r \notin att(AC_s)$

(II) M *is a stable model of* $D - R = (S^R, AC^R)$ *and* $D - R$ *is obtained by:*

- *(i) deleting all statements* $s \in R$ *from* S *to gain* S^R,
- *(ii) substituting for all* $s \in S^R$ *in* AC_s *all* $a \in \Sigma_{pv}^{AC_s}$ *with* $\bot$ *if* $a \in R$.

Note that we can omit the explicit manipulation of the links, which is needed for *ADFs* in general, as these informations are preserved in the acceptance conditions of *pForm-ADFs*. The last extension does not change for the *pForm-ADFs*, as a *preferred model* is just a subset maximal *admissible set*.

3.2 ADFs as Hypergraphs

The *pForm-ADF* representation has a strong logical and mathematical focus and it has enough expressiveness to get rid off the necessity of the links. Indeed Dung's AF has a strong connection and similarity to graphs. We will present a representation of ADFs which will emphasize on the connection to graph theory. We already know that ADFs are a generalization of Dung's AFs. So we will also use a generalization of graphs: *Hypergraphs*. A hypergraph is a graph, which contains hyperedges. The difference between edges and hyperedges is that they connect a set of nodes instead of only two. To express the direction of the attack respectively support, we

will attach *signs* to each node in the hyperedge. A sign may be positive, negative or neutral. If it is neutral (zero) it identifies the statement which will be attacked or supported by other statements, a positive sign (one) stands for a supporting statement, and a negative sign (minus one) expresses an attacking statement. We will call an edge an *"incoming edge"* for a statement, if this statement has a neutral sign.

Definition 3.2.1 (ADF as Hypergraph)**.** *An* ADF as a Hypergraph *is a pair $H = (S, R)$, where*

- *S is a set of statements*
- *R is a set of hyperedges E*

A hyperedge E is a set of pairs (s, w), where

- *$s \in S$,*
- *$w \in \{-1, 0, 1\}$, and*
- *exactly one pair exists such that $w = 0$.*

Each edge can be seen as a joint attack and a "rescuing" support. If one statement has more than one edge, the edges together can be seen as a mutual support and a "strong" attack. With rescuing and strong we mean that one support respectively attack is enough to accept or reject the whole edge respectively set of edges. Intuitively we need to put this kind of semantics into the meaning of edges to achieve the same expressiveness as the propositional formulae have. Based on a set of selected statements we can decide whether an edge is acceptable or not.

Definition 3.2.2 (acceptable edge)**.** *A hyperedge $E = \{(s_i, w_i)\} \in R$ of an ADF $H = (S, R)$ is acceptable w.r.t. a set $M \subseteq S$ if the sum $esum(E, M) = \sum\{w \mid (s, w) \in E, s \in M\}$ is not equal to the number of negative weighted pairs multiplied by minus one in the edge E.*

Example 3.2.3 (acceptable edge)**.** *Let $H = (S, R)$ be a hypergraph* ADF*, with $S = \{a, b, c\}$ and the following edge $\{(a, 0), (b, 1), (c, -1)\} \in R$. This would be the same as the acceptance condition $AC_a = b \vee \neg c$. Now we will show how the acceptance of the edge is computed for two different cases. In case 1 we have a, b and c selected and case 2 has only a and c selected. At first we need to get the number of negative weighted pairs, which is 1. For case 1 we have to build the $esum$ which is 0 ($-1 + 1 + 0$). 0 does not equal -1, so this edge is acceptable under the set of selected statements. In case 2 $esum$ is -1 ($-1 + 0$), which equals -1. So case 2 is no example for an acceptable edge and therefore a cannot be accepted here.*

Definition 3.2.4 (acceptable statement)**.** *Given an ADF $H = (S, R)$ and a set $M \subseteq S$. A statement $s \in S$ is acceptable w.r.t. M if all incoming edges are acceptable w.r.t. to the set M. $acc_M(H)$ denotes the set of all acceptable statements in H w.r.t. M.*

Based on the definition of an acceptable statement we can now formulate how a *conflict-free set* will look like. We will show that it is possible to use this representation to decide whether a set is a *conflict-free set* or not. We will omit further investigations to more semantics as this form of representation is intended for graphical representation, although it would be possible to define them too.

Definition 3.2.5 (conflict-free set). *Given an ADF $H = (S, R)$. A set $M \subseteq S$ is* conflict-free *if $\forall s \in M$ we know that $s \in acc_M(H)$ holds.*

As we want to have a relation between these hypergraph ADFs and the pForm ADFs, we will show that both are equivalent.

Definition 3.2.6. *A pForm ADF $D = (S, AC)$ is equivalent to a hypergraph ADF $H = (S', R)$ iff $S = S'$ and for every $s \in S$ and every $M \subseteq S : M \in mod_p(AC_s)$ iff $s \in acc_M(H)$ holds.*

Proposition 3.2.7. *For every* pForm-ADF *$D = (S, AC)$ an hypergraph ADF $H = (S', R)$ can be constructed such that D and H are equivalent and vice versa.*

Proof. In general S and S' are the same sets as this is one requirement for the equivalence of both representations.
Without loss of generality let the acceptance condition formulae be in $\mathcal{CF}$.In addition consider an auxiliary function, which maps a negative literal to -1 and a positive literal to 1:

$$weight(lit) = \begin{cases} 1 & \text{if } lit \text{ is positive} \\ -1 & \text{if } lit \text{ is negative} \end{cases}$$

At first we will show that we can construct a *pForm-ADF* $D = (S, AC)$ with an arbitrary hypergraph ADF $H = (S', R)$: Let e be a hyperedge in R, then we can construct the set of variables via

$$e_v = \{a \mid (a, 1) \in e\} \cup \{\neg a \mid (a, -1) \in e\}.$$

As all acceptance conditions are represented in $\mathcal{CF}$, every e_v can be seen as a clause. So we can construct the acceptance condition AC_s in $\mathcal{CF}$:

$$AC_s = \{e_v \mid (s, 0) \in e\}$$

So we have created for each statement an acceptance condition, based on the hypergraph edges.

The construction of an hypergraph ADF $H = (S', R)$ from an arbitrary *pForm-ADF* $D = (S, AC)$ is done as follows: For each $s \in S$ and there for each clause $c \in AC_s$, we construct a hyperedge

$$e_{s,c} = \{(atom(a), weight(a)) \mid a \in c\} \cup \{(s, 0)\},$$

where *atom* represents the variable in an arbitrary literal. Now we can create a set of hyperedges for each acceptance condition $H_s = \{e_{c,s} \mid (s, 0) \in e_{c,s}\}$. Then $H = \bigcup_{s \in S} H_s$.

To show that both are equal, we need to show that for all $M \subseteq S : M \in mod_p(AC_s)$ *iff* $s \in acc_M(H)$ holds.
To evaluate if a an interpretation I is a model of a CNF, we need at least one literal in each conjunct to be $true$ with respect to I. So in each conjunct we need an atom $a \in atom(AC_s)$ which is in the interpretation and occurs as a positive literal, or which is not in the interpretation and occurs as a negative literal. To ensure that $M \notin mod_p(AC_s)$, each literal in every conjunct needs to evaluate to $false$ under M. So all negative literals have to be in the interpretation and all positive literals are not allowed to be inside the interpretation. So $esum(E, M)$ of the corresponding edge E will be the inverse number of all negative literals, which is the only case where the edge is not acceptable and therefore also $s \notin acc_M(H)$. □

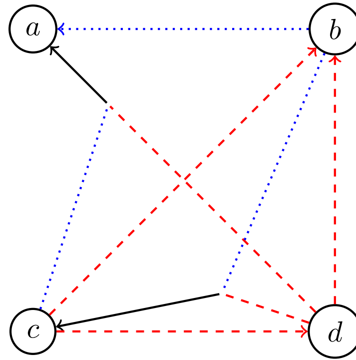
Now we will give a small example of such a hypergraph ADF.

Example 3.2.8 (hypergraph ADF). *ADF $D = (S, AC)$, where $S = \{a, b, c, d\}$, and*

$$\begin{aligned} AC_a &= b \wedge (c \vee \neg d) \\ AC_b &= \neg c \wedge \neg d \\ AC_c &= b \vee \neg d \\ AC_d &= \neg c \end{aligned}$$

The corresponding R for the ADF $H = (S, R)$ is:

$E_{a1} = \{(a, 0), (b, 1)\}$ $\quad E_{a2} = \{(a, 0), (c, 1), (d, -1)\}$
$E_{b1} = \{(b, 0), (c, -1)\}$ $\quad E_{b2} = \{(b, 0), (d, -1)\}$
$E_{c1} = \{(c, 0), (b, -1), (d, -1)\}$ $\quad E_{d1} = \{(d, 0), (c, -1)\}$



In the graphical representation we swap the numbers with red dashed and blue dotted lines for the attack respectively the support. If attacking and supporting links are in the hyperedge, the last part of the edge is black, as it is dependent on the set, whether an attack or an support will be important. Due to the visualization we can already see that b is attacked two times directly, so if c or d is in the set, b can not be an acceptable statement. Another situation can be seen for the node a: b can only support a if c is selected to "neutralize" an attack by d or if d is not present.

CHAPTER 4

Properties and Complexity of ADFs

Based on the presented *pForm-ADF* in the previous chapter, we will emphasize on some properties of *ADFs* now. This chapter will start with transformations to subclasses, where the approach of [Brewka et al., 2011a] will be presented. In addition we will introduce a new transformation to construct from every *ADF* a *BADF*. We then apply the idea of the transformation of *ADFs* to *BADFs* in order to generalize the stable extension such that it will work on arbitrary *ADFs*. Then we will investigate how some of the semantics relate to each other in comparison to Dung's Theorem (Theorem 2.2.16) and the definition of the stable extension. At last we will give a short overview on already existing computational complexity results and then extend these.

4.1 Transformations to Subclasses

The basic idea behind transformations to subclasses is the possibility to map the whole generalized class to one specific subclass without loss of information. Indeed this is not possible in every case as some restrictions can not be simulated in subclasses. If some kind of transformation can be done this may help to generalize well understood properties from the subclasses.

ADF $\rightarrow$ AF

Brewka, Dunne and Woltran proposed a transformation of *ADFs* to *AFs* [Brewka et al., 2011a]. This approach is tailored for a specific pair of semantics, namely the model semantics for *ADFs* and the stable extension for *AFs*. It extends the concept of semantical structures which was already mentioned in Section 2.2. To simulate *ADFs* via *AFs* the acceptance conditions of the statements are represented as boolean circuits which are directly encoded into the *AF*. For each statement in the *ADF* the acceptance of the corresponding argument in the *AF* is based on the truth-value output of the encoded circuit. The intuitive idea on the transformation is to encode the result of this circuit as a boolean network in an *AF*. For better readability they also used the fact that the boolean total functions (i.e. acceptance conditions) of *ADFs* can also be represented as a propositional formula. Based on these formulae, they constructed the semantic structures

such that the *ADF-model* corresponds to the *stable extension* of the transformed *AF* with respect to the statements of the *ADF* (i.e. taking only the arguments into account which are present in the *ADF*). Although this approach seems to work, there are some cases where the proposed transformations do not result in an *AF* with the desired properties. A fix on the structures for these issues is already done by the authors, but it is yet not publicized[1]. In the following we will introduce their procedure on the *pForm-ADFs* and present the correct semantical structures.

In the first step of the transformation for each statement an additional one needs to be created which represents the negation of the original statement. Intuitively this statement will be an argument selected by the stable extension if the corresponding statement is not selected by the *ADF-model*. If the acceptance condition of the statement is a truth-constant (i.e. $\bot$ or $\top$), we have to connect the two arguments, i.e. m and its negated variant $\neg m$, with an unidirectional attack (see Figure 4.1). In the case where the acceptance condition is not a truth-constant, we have to connect the two arguments with a mutual attack. After the arguments and their counter-

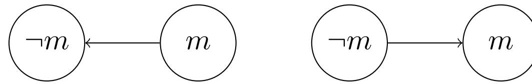


Figure 4.1: Semantic structure for the truth constants $\top$ (on the left) and $\bot$

arguments are built and related to each other, the structures which represent the boolean networks need to be attached to the arguments. There the network represents the acceptance of the argument, which together with its counter-argument is at the "output" position of the network. The related arguments are then the "inputs" for the network. The used semantical structures are illustrated in Figure 4.2 and 4.3. The first type of structures is used if the acceptance condition

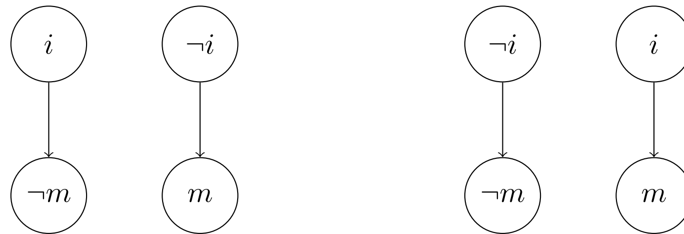


Figure 4.2: Semantic structures for non-compound literals

is only one literal. In the figure we suppose that the statement m of the *pForm-ADF* has the acceptance condition i respectively $\neg i$. These structures intuitively take advantage of the stable extension. So the left structure is a projection of the acceptance condition i for the statement m. If i is selected, then also m will be selected by the stable extension. Reversely the right structure projects the acceptance condition $\neg i$ for the statement m and the stable extension can only accept m if $\neg i$ is accepted. The structures in the second figure (Figure 4.3) construct an

[1]The used structures are taken from their work in progress paper and they are used with the permission of the authors (G. Brewka, P. Dunne and S. Woltran), without any claim of my own contribution.

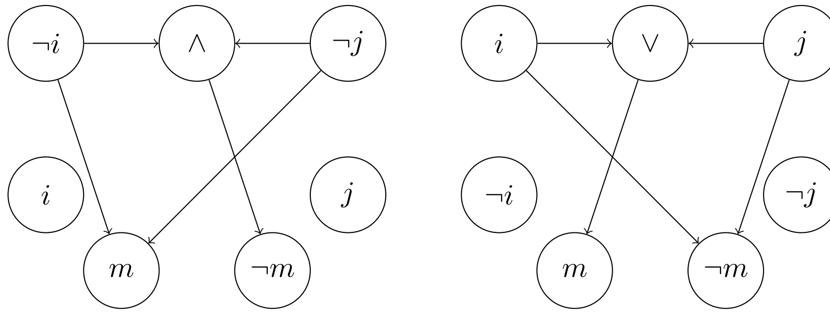
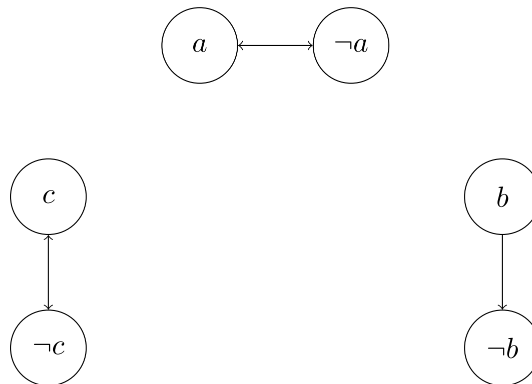


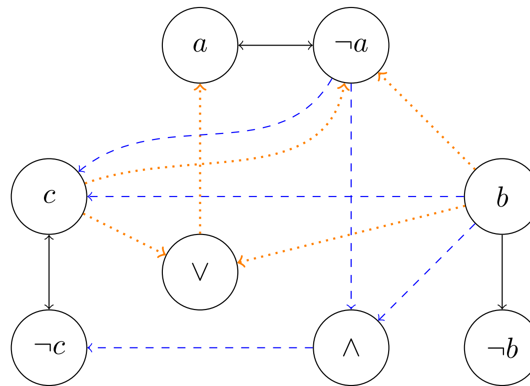
Figure 4.3: Semantic structures for conjunction and disjunction

AF for a statement m in a *pForm-ADF*, where the acceptance condition AC_m is either $i \wedge j$ or $i \vee j$. Intuitively they connect the arguments with some helper arguments, whose acceptance in the stable extension will enforce the acceptance of either m or $\neg m$. Indeed this acceptance is based on the acceptance of the arguments i and j. Note that on i and j again such a boolean network will be attached.

Example 4.1.1 (Transformation of an *ADF* to an *AF*)**.**
The pForm-ADF $D = (\{a, b, c\}, \{AC_a, AC_b, AC_c\})$ *with the acceptance conditions* $AC_a = b \vee c$*,* $AC_b = \top$*, and* $AC_c = a \wedge \neg b$ *can be transformed to the* AF *with the above described steps. At first we need to generate the arguments and their counter-arguments. In addition, based on the acceptance condition, the structural rules for the attacks need to be applied.*



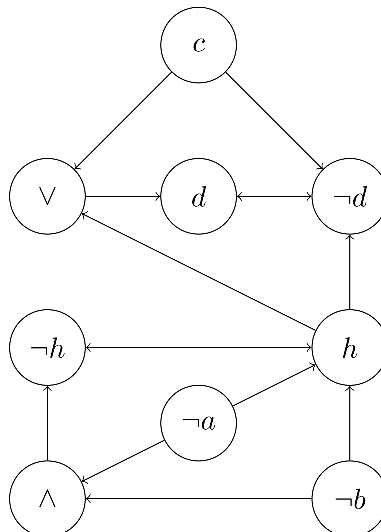
Then we insert the structures for the acceptance condition of a *and* b*. So we will get the following* AF D'*:*



For better readability we have used orange dotted lines for the attack relations based on the disjunctive structure and blue dashed lines for the attack relations on the conjunctive structures. Now we can compare the model of D, which is $\{a, b\}$ and the stable extension of D', which is $\{a, b, \neg c\}$. As there does not exist a statement $\neg c$ in D, we only compare a and b and they are indeed the model of D.

Although the example only shows how this structures are built on simple formulae with only one connective, it is indeed possible to build a semantic structure for each configuration of nested connectives. Then the structure needs to be constructed step by step and for each connective an additional helping argument needs to be introduced, which is some kind of a temporary output node for the subformula. In the following example we will show how such a pattern for a compound formula looks like.

Example 4.1.2 (Syntactical structure for compound formulae)**.** *We will omit the definition of the acceptance conditions of all statements but one. Lets look at the node d with the acceptance condition $AC_d = (a \wedge b) \vee c$. Then we have to construct the following semantic structure for the* AF*:*



The new helper argument h represents the value of the subformula $a \wedge b$ and $\neg h$ is the counter argument for the case where an argument with the acceptance condition $a \wedge b$ would not be accepted. Note that in this example we omit all arguments which are not in relation to arguments in the semantic structures. Indeed in a full ADF *to* AF *transformation there would be also relations between $\neg a$ and a (as well as for b and c), but they are dependent from the acceptance conditions for a (resp. b and c).*

Note that the construction of the semantical structures is based on the analysis of the modus operandi of the stable extension. The introduced auxiliary arguments and their relations are in fact an application of *Meta-Argumentation* (for further details see Section 6.1).

In Chapter 5, we will present an implementation together with benchmark results, to analyze how well the transformation approach performs compared with a direct implementation of the *ADF*-model semantics.

ADF $\rightarrow$ BADF

In the Sections 2.3 and 3.1 we have reviewed and presented the *ADF* and the *pForm-ADF*. There we had the distinction between *BADFs* and those *ADFs* which are not bipolar. We needed this kind of restriction to the subclass of *BADFs* to define some properties and semantics. As these semantics are not capable for the scope of all *ADFs*, a procedure to transform every *ADF* into a *BADF* will be presented. To gain this transformation an approach from the concept of *monotone pForm-ADFs*, where we already have shown that the *monotone pForm-ADFs* are equal to the class of *BADFs* (see Theorem 3.1.23), will be utilized.

Motivation of the transformation

We do know that every *monotone pForm-ADF* is a *BADF* and that an *ADF* which is not bipolar can not be represented as a *monotone pForm-ADF*. Note that it is computationally hard ($\mathbf{coNP}$-hard) to decide if a *pForm-ADF* is bipolar.

Our goal is to transform an arbitrary *ADF* to a corresponding *BADF*. It turned out to be useful to base this transformation on the transformation from *pForm-BADFs* to *monotone pForm-ADFs*. Borrowing ideas from the *ADF* $\rightarrow$ *AF* procedure, we will utilize auxiliary structures to accomplish this. To this end we apply the transformation for monotone formulae (see Definition 3.1.18) on the formulae. Every bipolar formula will be transformed to a monotone formula, and every formula which is not bipolar will be transformed to a *quasi-monotone* formula. This quasi-monotone formula satisfies for all attacking and supporting variables the monotone condition and only the variables which are dependent violate it.

Based on this distinction between monotone (i.e. bipolar) and quasi-monotone (i.e. not bipolar) formulae, acceptance conditions and *pForm-ADFs* it is easy to see that the dual polarity of the variable seems to be the central problem. In fact both polarities for the variable are necessary to represent the dependent nature of the link. Based on the interpretation, roughly speaking, the variable is “choosing” whether it “uses” its positive or negative polarity to change the truth value of the formula. This kind of changing attitude is also the intuitive reason behind the property that the variable is dependent. Since both polarities are needed to preserve the dependent

link behavior we cannot change the formula such that the variable satisfies the monotone condition. Instead of changing the formula we just split the statement into two parts: In one part we will change the formula such that the variable only utilizes its positive polarity and in the other one only the negative polarity occurs. In the quasi-monotone case the variable has some kind of choice, so we also let the statement choose: if one of the two parts is accepted, then the statement will be accepted. In practice these two parts are two new statements which have the customized formulae as acceptance conditions and the original statement gets a new acceptance condition, which is the disjunction of the two new helping statements.

One thing is still missing, namely the description of the customization (or transformation) of the formulae to resolve the dependent link. Intuitively we partition the set of interpretations of the formula to be transformed into two parts, namely one where the dependent variable evaluates to true and one where the variable evaluates to false under an arbitrary interpretation. So we get a formula where every model is an interpretation where the dependent variable occurs and another formula where every model is an interpretation where the dependent variable does not occur. Due to this partition it is easy to see that the variable has in the new formulae no interpretations where the supporting respectively attacking property can be refuted. Since this partition works on the semantical tier and not the syntactical it will also work for the set of all formulae and not only for the quasi-monotone ones. We just used this idea to motivate the approach.

Specification of the algorithm

In the following we will describe the detailed algorithm for arbitrary formulae. Algorithm 4.1 covers the basic mechanics and steps for the transformation. The basic idea is to add auxiliary statements for each dependent variable to resolve the dependent link into an attacking and supporting part. Note that we used the notation of *statement.AC* to refer to the acceptance condition for the statement. Every allocation to this structure is indeed a manipulation of the set of acceptance conditions.

The input for the algorithm is a *pForm-ADF* and at the end of the algorithm a transformed *pForm-BADF* is the result. In line 2, it is ensured that every statement will be checked whether it has to be transformed or not. Newly created statements through the transformation are checked too, since they are added to the set of statements in the line 11. The check whether the acceptance condition of a statement has dependent variables or not is done in line 4. Here as a preparation step it is possible to compute this information once before the algorithm (*dependency-method 1*) or every time the check is done (*dependency-method 2*). Then the first dependent variable is taken and the acceptance condition of the current statement is copied two times and the two copies of the acceptance conditions formulae are altered such that the dependent variable is conjunctively attached to one formula and the negated variable to the second one (lines 6 and 7). Then the two new acceptance conditions are attached to two new statements and the acceptance condition of the current statement is altered to be the disjunction of the two new ones. Finally we add the two new statements to the set of statements in the *pForm-BADF* (line 11). In line 13 it is ensured that every statement of the origin *pForm-ADF* is also in the *pForm-BADF*.

Based on the used dependency-method some additional things have to be taken into account. If method 1 was used, then s' and s'' inherit all the dependent variables of s, except the one which was resolved by the construction of s' and s''. In addition we have to mark s dependent-variable-

Algorithm 4.1: *ADF* $\rightarrow$ *BADF* algorithm

Input : A *pForm-ADF* $D = (S, AC)$
Output: The transformed *pForm-BADF* $D' = (S', AC')$, which corresponds to D

```
1  S' ← ∅
2  while S != ∅ do
3  |   s ∈ S
4  |   if s has dependent variables then
5  |   |   α ← first dependent variable of s
6  |   |   ψ ← s.AC ∧α
7  |   |   φ ← s.AC ∧¬α
8  |   |   s.AC ← s' ∨ s''
9  |   |   s'.AC ← ψ
10 |   |   s''.AC ← φ
11 |   |   S ← S ∪ {s', s''}
12 |   end
13 |   S' ← S' ∪ {s}
14 |   S ← S \ {s}
15 end
```

free, because the acceptance condition shifted to the new statements s' and s''. Method 2 needs to check every time how the link types of the *pForm-ADF* have changed due to the change of models for the two formulae. Both methods have their advantages and disadvantages. Method 1 does not need any more checking and can work without further investigations, but it is possible that due to the change of models some variables are not dependent any more. So more additional statements would be generated than needed. This results in a bigger *pForm-BADF* but will not affect the correctness of the algorithm. The second method is more aware of the changes and only creates new statements if they are really needed. But the check has to be done for all variables (except these which are already transformed). Note additionally that we can not use the concept of quasi-monotone formulae for this check as the lines 4-11 do not create monotone formulae.

Correctness of the approach

Now we will show that the approach is a transformation which does not change the acceptance values for the original statements (i.e. those in the *ADF*). Intuitively the transformation carries the acceptance decision away from the current statement and delegates it to two helping statements. These helping statements only accept one of the two cases (i.e. one time it only accepts the interpretations which include the dependent variable, and once where it only accepts interpretations which do not contain the dependent variable). To show the correctness of this approach we have to prove that this delegation on the level of statements does not interfere with the set of interpretations which are accepted by the delegating statement. Note that we can only compare those statements which occur in both *ADFs*.

Definition 4.1.3 (ADF $\rightarrow$ BADF transformation step)**.** *Let $D = (S, AC)$ be a* pForm-ADF *and $s \in S$ be a statement, such that AC_s is a propositional formula ψ which has a dependent variable a. Then the transformation creates a* pForm-ADF $D' = \delta(D)$ *in the following way:*

(I) create $AC'_{s'}, AC'_{s''}$, such that $AC'_{s'} = \psi \wedge a$ and $AC'_{s''} = \psi \wedge \neg a$

(II) $AC'_s = s' \vee s''$

(III) $S' = S \cup \{s', s''\}$, $AC' = (AC \setminus \{AC_s\}) \cup \{AC'_{s'}, AC'_{s''}, AC'_s\}$

Lemma 4.1.4. *Let $D = (S, AC)$, $D' = (S', AC')$ be* pForm-ADFs*, $s \in S$ be statement, such that AC_s is a propositional formula ψ which has a dependent variable a, and $D' = \delta(D)$. For the acceptance conditions the following holds: $mod_p(\psi) = mod_p(AC'_{s'}) \cup mod_p(AC'_{s''})$.*

Proof. Let the signature of ψ be Σ^{ψ}_{pv}, $I \subseteq \Sigma^{\psi}_{pv}$ be an arbitrary interpretation of ψ. In addition let $\psi' = \psi \wedge a$ and $\psi'' = \psi \wedge \neg a$. We can distinct between two cases:

(I) $I \notin mod_p(\psi)$: If I is not a model of ψ, then in ψ' and ψ'' one component of the conjunction (i.e. ψ) evaluates to $false$ under the interpretation I and so it is not possible for the whole conjunction to get the truth-value $true$ under I. Therefore I is neither a model for ψ' nor for ψ''.

(II) $I \in mod_p(\psi)$: If I is a model of ψ, we have to investigate two cases:

  (i) $a \in I$: As $I \in mod_p(\psi)$ and $a \in I$: $\mathcal{V}_I(a) = true$ and $\mathcal{V}_I(\psi) = true$. So $\mathcal{V}_I(\psi \wedge a) = true$ also holds.

  (ii) $a \notin I$: As $I \in mod_p(\psi)$ and $a \notin I$: $\mathcal{V}_I(\neg a) = true$ and $\mathcal{V}_I(\psi) = true$. So $\mathcal{V}_I(\psi \wedge \neg a) = true$ holds too.

□

Intuitively the next Lemma will show that the transformation does not change the set of conflict-free sets with respect to the statements which occur in both *pForm-ADFs*.

Lemma 4.1.5. *Let $D = (S, AC)$, $D' = (S', AC')$ be* pForm-ADFs*, $s \in S$ be statement, such that AC_s is a propositional formula ψ which has a dependent variable a, and $D' = \delta(D)$. Let $M \subseteq S$, $M' \subseteq S'$ be two sets in D, D' such that $(M' \setminus \{s', s''\}) = M$. Then $M \in cf_{pADF}(D)$ iff $M' \in cf_{pADF}(D')$.*

Proof.

(I) Assume $M \in cf_{pADF}(D)$: If M is conflict-free, then for every $m \in M$ we know by Definition 3.1.24 that $M \in mod_p(AC_m)$ holds. The transformation step only changed the acceptance condition of s. So M still satisfies all acceptance conditions of the statements, except AC'_s. If $s \in M$, then s' or s'' needs to be added to M' to satisfy AC'_s. By Lemma 4.1.4 we know that $mod_p(AC_s) = mod_p(AC'_{s'}) \cup mod_p(AC'_{s''})$ holds. So M is

either a model for $AC'_{s'}$ or for $AC'_{s''}$. The addition of the satisfied statement (i.e. s' respectively s'') will satisfy AC'_s and does not interfere with the other acceptance conditions. If $s \notin M$ we do not need to change anything, as all acceptance conditions will stay satisfied under M.

(II) Assume $M' \in cf_{pADF}(D')$: If M' is conflict free, and none of $\{s, s', s''\}$ are in the conflict free set, then it will be also a conflict free set in D, as the acceptance conditions are satisfied without them. If $s' \in M$ or $s'' \in M$, but $s \notin M$, the removal of s' respectively s'' will not change the truth-value of the other statements with respect to M and so the removal will not change the property of being conflict-free. If $s \in M$, then $s' \in M$ or $s'' \in M$ has to hold too, because of AC'_s. Again by Lemma 4.1.4 we know that if $AC'_{s'}$ or $AC'_{s''}$ is satisfied, also AC_s is satisfied. So the removal of s' resp. s'' will not set the truth-value of AC_s to false.

□

Based on the result for conflict-free sets we will advance to the notion of models.

Lemma 4.1.6. *Let $D = (S, AC)$, $D' = (S', AC')$ be* pForm-ADFs, *$s \in S$ be statement, such that AC_s is a propositional formula ψ which has a dependent variable a, and $D' = \delta(D)$. Then, for all $M \subseteq D$ holds: $M \in model_{pADF}(D)$ iff $M' \in model_{pADF}(D')$, where $(M' \setminus \{s', s''\}) = M$.*

Proof. The model is a conflict-free set, where each statement whose acceptance condition is satisfied, has to be in the set of statements. If $s \in M$, then $M \in mod_p(AC_s)$, therefore $M \in mod_p(AC'_{s'})$ or $M \in mod_p(AC'_{s''})$ holds and has to be collected into the set of statements (by Lemma 4.1.4). So AC'_s is also satisfied and the property of a model is satisfied. If $s \notin M$, then neither s' nor s'' can be added to the set of statements and nothing changes for the set of models.
If $s \notin M'$, then also s' and s'' can not be in M'. In this case M' and M are the same. If s' or s'' is selected to be in M', then also s has to be added to the model ($AC'_s = s' \vee s''$). So the addition of one of the two helping statements does enforce the addition of s. One of the helping statements was selected because the set of statements satisfied its acceptance condition. Again by Lemma 4.1.4 we know that this selection of statements would also satisfy AC_s. So again these two models are equal with respect to the statements in D. □

Lemma 4.1.7. *Let $D = (S, AC)$, $D' = (S', AC')$ be* pForm-ADFs, *$s \in S$ be statement, such that AC_s is a propositional formula ψ which has a dependent variable a, and $D' = \delta(D)$.
Let (A, R), (A', R') be the least fixed-point of Γ_D respectively $\Gamma_{D'}$. Then $A = A' \setminus \{s', s''\}$ and $R = R' \setminus \{s', s''\}$ holds.*

Proof. Assume we begin with an empty set of accepted and rejected statements. Each step collects the statements whose acceptance conditions are true respectively false under the partial interpretation, based on the accepted and rejected statements. If we collect one of the unchanged statements in D or D' they have the same acceptance conditions and so the sets of accepted and rejected statements will be the same. If we have to accept s' or s'', then we can also accept s

in the next step. This will only happen if a is already accepted or rejected, as this is a variable on the top-level conjunction (i.e. it is a switch to accept or reject the statement). In addition the acceptance of s' will reject s'' and vice versa. If both, s' and s'' are rejected, then the partial interpretation for both acceptance conditions $AC'_{s'}$ and $AC'_{s''}$ validate to false (i.e. it is unsatisfiable under the current partial interpretation). So the same has to hold for AC_s, as they share the same set of propositional models with s' and s''. □

Now we can show that the transformation $\delta(D)$ of an *ADF* D does not change the sets of statements under the semantics defined for *ADFs* with respect to the statements in D.

Theorem 4.1.8. *Let* $D = (S, AC)$, $D' = (S', AC')$ *be* pForm-ADFs, $s \in S$ *be statement, such that* AC_s *is a propositional formula* ψ *which has a dependent variable* a, *and* $D' = \delta(D)$. *Then*

$$\sigma_{ADF}(D) = (\sigma_{ADF}(D') \setminus \{s', s''\})$$

holds for $\sigma = \{cf, model, wf\}$

Proof. Follows by Lemma 4.1.5, 4.1.6, and 4.1.7. □

We have shown that one application of the transformation δ does not change the set of accepted sets under different semantics with respect to the statements in the origin *ADF*. In addition we have shown that the transformation removes the dependent variable and splits it into one supporting and one attacking variable. At next we want to show that the repeated application of δ will result in a fixed-point, where the set of accepted sets under the semantics with respect to the statements in the origin *ADF*, does not change and that the resulting *ADF* is bipolar.

Theorem 4.1.9. *Let* $D = (S, AC)$ *be a* pForm-ADF, $D' = (S', AC')$ *be a* pForm-BADF, *such that* D' *is the fixed-point of* $\delta(D)$, *denoted by* $Fix_\delta(D)$, *where* γ *is the set of all auxiliary statements which are created during the applications of* δ. *Then*

$$\sigma_{ADF}(D) = (\sigma_{ADF}(D') \setminus \gamma)$$

holds for $\sigma = \{cf, model, wf\}$.

Proof. Based on Theorem 4.1.8 each application of δ does not change the set of acceptable sets. As every iteration of δ removes one dependent variable and it only changes the *ADF* if a dependent variable occurs, a fixed-point will be reached. D' has no dependent variables, so it is bipolar by Corollary 3.1.14. □

4.2 Generalization of the Stable Extension

Based on the concept of the transformation of *ADFs* to *BADFs*, we will now present a generalized approach for the stable extension. Indeed we will try to generalize the stable model semantics to work on *ADFs* and not only on *BADFs*. The basic idea is to use the results of the transformation to refine the steps of the already existing stable model semantics. In the following we will show the concept of the generalized semantics, followed by a proof that the refined definition does inherit the results captured by Theorems 4.1.8 and 4.1.9.

Definition 4.2.1 (Generalized stable model semantics for *pForm-ADFs*)**.** *Let $D = (S, AC)$ be a* pForm-ADF. *A model M of D is a* stable model *if M is the least model of the reduced* pForm-ADF *$D^M = (S^M, AC^M)$ obtained from D by*

(I) eliminating all nodes not contained in M, such that $S^M = S \cap M$,

(II) for all $s \in S^M$ substitute in AC_s all $a \in \Sigma_{pv}^{AC_s}$ with $\bot$ if $a \notin S^M$, to gain AC'_s,

(III) for all $s \in S^M$ substitute in AC'_s all $a \in \Sigma_{pv}^{AC'_s}$ with $\bot$ if $a \in att(AC_s)$, to gain AC''_s.

(IV) for all $s \in S^M$, if $\{a_1, ..., a_n\}$ is the set of all selected dependent variables in AC_s (i.e. $a_i \notin att(AC_s)$, $a_i \notin sup(AC_s)$, $a_i \in atoms(AC_s)$, and $a_i \in M$ for $1 \leq i \leq n$) then $AC_s^M = AC''_s \wedge a_1 \wedge \cdots \wedge a_n$

$smodel_{pADF_g}(D)$ is the set of all stable models of D obtained by the generalization.

Proposition 4.2.2. *For a* pForm-BADF *D the following holds:*

$$smodpadf\, D = smodel_{pADF_g}(D)$$

Proof. A *pForm-BADF* consists of bipolar acceptance condition formulae. If an acceptance condition formula has a variable which is neither supporting nor attacking, it is not a bipolar acceptance condition formula. So the fourth step in the reduction process will never be applied. As the fourth step is the only addition to the stable model semantics both definitions need to have the same results. □

At next we will show that the generalized stable model semantics will also w

Proposition 4.2.3. *Let $D = (S, AC)$ be a* pForm-ADF *which is not bipolar and let $D' = Fix_\delta(D)$ be a* pForm-BADF. *Then*

$$M' \in smodel_{pADF}(D') \iff M \in smodel_{pADF_g}(D)$$

holds for $M' \cap S = M$. For ADFs *which are not bipolar the generalized model semantics treat the acceptance condition formulae which have dependent variables as if the* pForm-ADF *would have been transformed to a* pForm-BADF.

Proof. Let $D = (S, AC)$ be a *pForm-ADF*, $s \in S$ be an arbitrary statement such that $AC_s = \psi$ has a dependent variable a. $D' = Fix_\delta(D)$. The reduction only works on selected statements, so we assume that $s \in M$.

(I) Assume that $a \notin M$. As $s \in M$, we know by Lemma 4.1.6 that s'' with $\psi \wedge \neg a$ needs to be in M too ($\mathcal{V}_I(\neg a)$ is only true iff $a \notin I$). a is not in M, so all occurrences of a in $AC'_{s'}$ are substituted by $\bot$. $\neg a$ turns into $\top$ and so the original formula before the transformation is restored. In addition all other occurrences of a are substituted too. So the substitution of a with $\bot$ in AC_s would do the same changes as in the transformed acceptance condition $AC'_{s''}$.

(II) Assume that $a \in M$. By Lemma 4.1.6 we know that s' with $\psi \wedge a$ would be in M. The transformed supporting version of the acceptance condition was selected, so there exists in this model a supportive relation between s and a. To be sure that the acceptance condition will only use its supportive character, the fourth step in the reduction process transforms the AC_s to its supportive counter part $AC'_{s'}$.

The proof has shown that for one dependent variable the right decision is done. As this is done for every dependent variable, the above reasoning holds for every dependent variable too. □

Intuitively we judge on the accepted statements of the model whether the attacking nature of the dependent link was taken in account or not. If both statements are in the model, then the attacking nature of the dependent link could not be taken into account, as otherwise the selection of both would result in a set which invalidates the conflict-free property. In the case where the dependent link was not attacking we have to assume that the supporting nature was present in some extend and so we have to force the variable to be supporting in the reduction. In the case where the dependent link statement was not selected it is treated in the same way as an attack (It was refused to avoid an invalidation of the conflict-free property).

To give a better insight on the process we will present some examples of *ADFs* which are not bipolar and then we will evaluate the stable models of them.

Example 4.2.4 (Small *ADFs* and their stable models)**.** *We will have look on the* pForm-ADF $D_1 = (S_1, AC_1)$ *and* $D_2 = (S_2, AC_2)$*, where*

$$
\begin{aligned}
S_1 = S_2 &= \{a, b, c\}, \\
AC_1 &= \{AC_{1a}, AC_{1b}, AC_{1c}\}, \\
AC_{1a} &= \neg b, AC_{1b} = \neg a, AC_{1c} = (a \wedge \neg b) \vee (\neg a \wedge b), \\
AC_2 &= \{AC_{2a}, AC_{2b}, AC_{2c}\}, \text{ and} \\
AC_{2a} &= b, AC_{2b} = a, AC_{2c} = (a \wedge b) \vee (\neg a \wedge \neg b)
\end{aligned}
$$

These pForm-ADFs *have the following models:*
$model_{pADF}(D_1) = \big\{\{a, c\}, \{b, c\}\big\}$
$model_{pADF}(D_2) = \big\{\{c\}, \{a, b, c\}\big\}$
Based on these models we can use the reduction steps and achieve the following acceptance conditions (Note that we omit the set of statements and just identify it by the presence of an acceptance condition):

$$
\begin{aligned}
AC_1^{\{a,c\}} &= \{AC'_{1a}, AC'_{1c}\}, AC_1^{\{b,c\}} = \{AC''_{1b}, AC''_{1c}\} \\
AC'_{1a} &= \top, AC'_{1c} = ((a \wedge \neg\bot) \vee (\neg a \wedge \bot)) \wedge a \equiv (a \wedge \top) \equiv a \\
AC''_{1b} &= \top, AC''_{1c} = ((\bot \wedge \neg b) \vee (\neg\bot \wedge b)) \wedge b \equiv (b \wedge \top) \equiv b \\
AC_2^{\{c\}} &= \{AC'_{2c}\}, AC_2^{\{a,b,c\}} = \{AC''_{2a}, AC''_{2b}, AC''_{2c}\} \\
AC'_{2c} &= (\bot \wedge \bot) \vee (\top \wedge \top) \equiv \top \\
AC''_{2a} &= b, AC''_{2b} = a, AC''_{2c} = ((a \wedge b) \vee (\neg a \wedge \neg b)) \wedge a \wedge b \equiv a \wedge b
\end{aligned}
$$

Now we can construct the least models of the reductions and see whether the models are stable or not, so we get the set of stable models:
$smodel_{pADF}(D_1) = \{\{a, c\}, \{b, c\}\}$
$smodel_{pADF}(D_2) = \{\{c\}\}$
These two examples show that self-support-cycles are still removed and that the dependent links are really reduced to supporting links if they where selected in a model.

4.3 Relations between Semantics

This section will analyze relations between semantics of *ADFs* and studying differences to semantics defined by Dung. We present some properties which hold for Dung's AF but do not hold for *ADFs*. In addition we will try to justify the results and give some ideas why these properties do not hold. Note that the following inconsistencies with the properties of Dung's AF are not serious enough to abandon the work or consideration of *ADFs*. They should be seen as a reminder that some details of the concept should be revisited to fix some unwanted side effects or unintuitive behavior.

One very basic property follows directly from Dung's definition of the stable extension (see Definition 2.2.10). Informally the property claims that no stable extension is a proper subset of another stable extension. For better readability we state the corresponding property for *ADFs*.

Definition 4.3.1. *Let* $D = (S, AC)$ *be a* pForm-ADF *and* $M' \subset M \subseteq S$ *be two sets of statements of* D*. If* $M \in smodel_{pADF}(D)$ *then* $M' \notin smodel_{pADF}(D)$.

In the following we will give a counter-example for this property, i.e. show that it does not hold. Note that the *ADF* in the counter-example is a *BADFs*.

Example 4.3.2 (Counter-example to Definition 4.3.1)**.** *The* pForm-BADF $D = (S, AC)$ *with the statements* $S = \{a, b\}$ *and the acceptance conditions*
$AC_a = \top$,
$AC_b = \neg a \vee b$
has the following stable models: $\{a\}, \{a, b\}$.

We will now investigate the counter-example in more detail. The set $\{a\}$ is obviously a model and due to the acceptance condition it is also the least fixed-point of the monotonic operator Th_D on the reduced *ADF*. It seems a little bit unintuitive that $\{a, b\}$ is a stable model too, so lets check what happens: $\{a, b\}$ is a model, as both acceptance conditions are satisfied. The reduction will not remove any statements and so it only has to remove the attacks for the reduct. $\neg a \vee b$ is substituted by $\neg\bot \vee b$ which is equivalent to $\top \vee b$ respectively $\top$. So it is a legal action to accept both statements together as a stable model, because the monotonic operator Th_D has approved a and b, as both are always acceptable due to their reduced acceptance condition. From an intuitive point of view the acceptance condition of b means that b is accepted if a is not selected or b is selected. So the only case where b may not be selected is when a is selected and b not. If we remove the attacking link (a, b) from AC_b during the reduction step, information about the self-support of b should be preserved. Due to the fact that a disjunction only evaluates

to $false$ if both sub-formulae evaluate to $false$, the removal of the attack transforms the disjunction to a valid formula. Note that this behavior also holds for *ADFs* with binary total function acceptance conditions. It seems the removal of attacks by substituting the attacks by top (resp. removing the total function mappings where the attacking variable occurs) removes in the case of the disjunctive connective too much information of the acceptance condition. One possible fix for the stable model semantics on *pForm-ADFs* may be to analyze which changes have to be done by the reduct to reflect an intuitive behavior of the reduced formulae, instead of the substitution of atoms with constants.

Based on Dung's theorem (see Definition 2.2.16) we can analyze another well known property for Dung's AF to *ADFs*. Point (I) and (II) of the theorem together relate the preferred extension with the grounded extension via the complete one. Although there is no *ADF*-counter-part for the complete extension, we want to investigate if the following well-known property for *AFs* also holds for *ADFs*. Shortly sketched, we do know that each preferred extension is a complete extension. In addition the grounded extension is least complete extension with respect to set inclusion. Therefore each grounded extension needs to be a subset of each preferred extension.

Definition 4.3.3. *Let* $D = (S, AC)$ *be a* pForm-ADF*,* $W \subseteq S$ *be the well-founded model of* D*. If* $P \subseteq S$ *is a* preferred model *of* D*, then* $W \subseteq P$*.*

In the same manner as above we will now present a counter-example to proof that this property does not hold.

Example 4.3.4 (Counter-example to Definition 4.3.3)**.** *The* pForm-ADF $D = (S, AC)$ *with the statements* $S = \{a, b, c, d\}$ *and the acceptance conditions*
$AC_a = \neg b \vee \neg c$*,*
$AC_b = \top$*,*
$AC_c = d$*, and*
$AC_d = \top$
has the following preferred models: $\{a, b\}$ *and* $\{b, c, d\}$ *The well-founded model is* $\{b, c, d\}$*.*

The above example obviously violates the property, as $\{b, c, d\} \not\subseteq \{a, b\}$. The violation may come from two sources: The well-founded or the preferred semantics. The well-founded model is intuitively just the natural step to improve the functionality of the grounded extension with the concept of supports. For Property 4.3.3 to hold we now will study possible adoptions. First let us try to modify the well-founded mode and let the preferred stay the same. For the example this would imply that either the set $\{b\}$ or the empty set $\emptyset$ is the well-founded model. The former is unintuitive since it would be hard to argue why b is selected and d not, although they have the same acceptance condition. The empty set is likewise undesired since then the well-founded model would not accept statements with tautological acceptance conditions. So let us have a look on the preferred semantics. For the preferred model we need to compute the subset-maximal admissible sets. Intuitively the admissible set is found by choosing a subset of the statements and try to cut unwanted statements of the framework out till the chosen subset is a stable model of this reduced framework (Note that there are some restrictions to the selection of the unwanted statements which are cut out). It seems that this is the problem of the admissible set, as we can remove d from the framework (it has no attack relation), and then $\{a, b\}$ is a

stable model. To improve the admissible set, maybe some additional restrictions for the removal of statements (i.e. restrictions for elements in R) can solve this problem. Another approach can be an independent development of the admissible set semantics, as the current concept is based on relations between the admissible set and the stable extension of Dung's AF.

4.4 Complexity Analysis of ADFs

This part is dedicated to the computational complexity of *ADFs*. We will review the currently known results on common decision problems for *ADFs* and then we will present new results. Note that we will describe the decision problems of already existing results and questions for *pForm-ADFs*, although they were original defined for *ADFs* respectively *AFs*.

Review of existing results

As described in Section 2.5, we are interested in problems which consist of a question and result in a *"yes or no"*-answer. These decision problems are generally defined as a given input and one question. In argumentation several decision problems are of relevance. In order to present currently available results we will first introduce the most important problems which are the *existence* problem, as well as the *skeptical* and *credulous acceptance* problems.

The *existence* problem is the question for the existence of an model for a given semantics. We will denote this problem with $\mathbf{Exist}_\sigma$.

$\mathbf{Exist}_\sigma$ decision problem	
Given:	A *pForm-ADF* $D = (S, AC)$ and a semantics σ.
Question:	Does a σ-extension for D exist?

This problem is in many cases like the well-founded semantics or the admissible sets trivial, as in every case at least the empty set is computed as an legit model. So some authors refine the existence problem to ask whether a non-empty σ-extension of D exists or not.

$\mathbf{Exist}_\sigma{}^{\neg\emptyset}$ decision problem	
Given:	A *pForm-ADF* $D = (S, AC)$ and a semantics σ.
Question:	Does a non-empty σ-extension of D exist?

Sometimes we want to know more detailed information about the accepted statements in the model. The *credulous acceptance* problem asks if a specific statement occurs in one model under a given semantics. The formal problem is denoted by $\mathbf{Cred}_\sigma$.

$\mathbf{Cred}_\sigma$ decision problem	
Given:	A *pForm-ADF* $D = (S, AC)$, a statement $s \in S$ and a semantics σ.
Question:	Does s occur in at least one σ-extension of D?

Another interesting problem is the question whether a statement occurs in every model under a given semantics or not. This would give a hint that this statement is crucial for the construction of answers as it gets accepted by every answer set of statements. This problem is called the *skeptical acceptance* problem $\mathbf{Skept}_\sigma$.

$\mathbf{Skept}_\sigma$ **decision problem**	
Given:	A *pForm-ADF* $D = (S, AC)$, a statement $s \in S$ and a semantics σ.
Question:	Does s occur in every σ-extension of D?

Note that the answers to some of these problems may lead to misleading conclusions. For example the $\mathbf{Skept}_\sigma$-question is answered with *"yes"* if the $\mathbf{Exist}_\sigma$-question for the same *pForm-ADF* is answered with *"no"*. At first this may seem a little bit unintuitive, but if there is no extension then every statement is in every extension. So in practical use one should not only depend on one decision problem.

One can also ask whether a given set of statements is a model under a given semantics. So we want to *verify* if a given set is a model. We denote $\mathbf{Ver}_\sigma$ for the *verification* problem.

$\mathbf{Ver}_\sigma$ **decision problem**	
Given:	A *pForm-ADF* $D = (S, AC)$, a set $M \subseteq S$ and a semantics σ.
Question:	Is M a σ-extension of D?

Till now we have presented decision problems which are of relevance in general for argumentation. Next we will describe a central *ADF*-specific problem. During the work with *ADFs* it is hard to miss that some semantics depend on the restriction of *BADFs*. A natural question therefore is if an *ADF* is a *BADF*.

bipolarity decision problem	
Given:	A *pForm-ADF* $D = (S, AC)$
Question:	Is D a *BADF*?

All currently known complexity results were proposed in the introductory work of *ADFs*. Table 4.1 gives an overview on these complexity results [Brewka and Woltran, 2010]. In the table we have one row for each semantics and every column stands for one decision problem. The cells where the lines and columns intersect represent the currently available results. Note that we have omitted the $\mathbf{Exist}_\sigma{}^{\neg\emptyset}$ problem, as there are no results yet. The asterisk symbol in the table means that the results assume that the link types are known before. The table also shows that there are many open problems. An additional result which is not shown in the table is that the **bipolarity decision problem** was shown to be $\mathbf{coNP}$-hard.

	$\mathbf{Cred}_\sigma$	$\mathbf{Skept}_\sigma$	$\mathbf{Ver}_\sigma$	$\mathbf{Exist}_\sigma$
Model	?	?	?	?
Stable	$\mathbf{NP}$-c*	$\mathbf{coNP}$-c*	?	?
Preferred	$\mathbf{NP}$-c*	$\Pi_2\mathbf{P}$*	?	trivial
Well-founded	?	?	$\mathbf{coNP}$-hard	trivial

Table 4.1: Complexity results on decision problems for *ADFs*

New results

Based on the given overview we will now refine the results. Many of the complexities incorporate the assumption that the link types are known. As there are no results for this decision, we will show the completeness of this problem. To decide the link type we need to evaluate once if it is attacking and once if it is supporting. Our goal is to show the complexity of deciding, whether a link is attacking or not.

attack link decision problem	
Given:	A *pForm-ADF* $D = (S, AC)$ and two statements $a, s \in S$.
Question:	Is $a \in att(AC_s)$.

In order to gain a completeness result, we need to show the membership and the hardness to a complexity class.

Proposition 4.4.1. *The **attack link decision problem** has a* $\mathbf{coNP}$*-membership.*

Proof. Let $D = (S, AC)$ be a *pForm-ADF*, and $a, s \in S$ be two statements. To show that $a \notin att(AC_s)$ holds, an interpretation $I \subseteq \Sigma_{pv}^{\psi}$ where $a \notin I$ holds can be guessed. Then it may be checked if $I \notin mod_p(AC_s)$ and $(I \cup \{a\}) \in mod_p(AC_s)$. Both checks can be done in $\mathbf{P}$ and therefore the complement problem is in $\mathbf{NP}$. As the complement problem is in $\mathbf{NP}$ our original problem, namely the decision if $a \in att(AC_s)$, is in $\mathbf{coNP}$. □

Proposition 4.4.2. *The **attack link decision problem** is* $\mathbf{coNP}$*-hard.*

To show the hardness in the following proof we will need to use another problem which is known to be $\mathbf{coNP}$-hard and then we have to find a polynomial reduction to get a *pForm-ADF*, where the answer to the attack question is always the same as the answer to the question asked in the already known problem.

Proof. The **validity** problem is known to be $\mathbf{coNP}$-complete.

validity decision problem	
Given:	A propositional formula ψ.
Question:	Is ψ a valid formula?

Let ψ be an arbitrary propositional formula, then we construct a *pForm-ADF* $D = (S, AC)$, such that

$$\begin{aligned} S &= \Sigma_{pv}^{\psi} \cup \{a, t\}, \text{ where } a, t \notin \Sigma_{pv}^{\psi}, \\ AC_s &= s \text{ for all } s \in (S \setminus \{t\}), \text{ and} \\ AC_t &= \psi \vee a. \end{aligned}$$

The question for our decision problem is now: "Is $a \in att(AC_t)$?"
Now we show that $a \in att(AC_t) \iff \psi$ *is valid*: Based on the construction of AC_t it follows that $\Sigma_{pv}^{AC_t} = \Sigma_{pv}^{\psi} \cup \{a\}$ and therefore every interpretation $I \subseteq \Sigma_{pv}^{\psi}$ is also a legal interpretation of the acceptance condition AC_t. The answer to the validity problem may be one of the following cases.

- ψ is not valid: So we know that for at least one interpretation $I \subseteq \Sigma_{pv}^{\psi}$ it holds that $I \notin mod_p(\psi)$, then $I \notin mod_p(AC_t)$. Since $a \notin I$ and $I \cup \{a\} \in mod_p(AC_t)$ this is a counter example for an attack, i.e. $a \notin att(AC_t)$.
- ψ is valid: It follows that for every interpretation $I \subseteq \Sigma_{pv}^{\psi}$ it holds that $I \in mod_p(\psi)$. AC_t is a disjunction of ψ and a. If ψ always evaluates to true, then $I \in mod_p(AC_t)$ and $I \cup \{a\} \in mod_p(AC_t)$ for all $I \subseteq \Sigma_{pv}^{AC_t}$ holds. So no counter example for the attacking link can be found, i.e. $a \in att(AC_t)$ holds.

The answer of the attack is in this case purely dependent on the validity of the formula ψ therefore both will give in all cases the same answer. □

The membership proof for the quite similar *support link decision* problem is completely analogous and for the proof of its hardness only the acceptance condition for AC_t needs to be modified a little bit. Therefore we will omit these proofs to reduce redundancy. With these two questions we can now check for every link which type it has. So we get some additional sense about the costs for the knowledge which is assumed to be given by some of the existing results. Although the information about the links can be acquired once for all the problems at least the intractability of this subproblem shall be kept in mind.

$\mathbf{Cred}_{stable}^{m}$ **decision problem**	
Given:	A *monotone pForm-ADF* $D = (S, AC)$ and a statement $s \in S$.
Question:	Is s in at least one stable model of D?

The notion of *monotone pForm-ADFs* (see Definition 3.1.16) has been introduced in this work. We want to show that the computational complexity for the *credulous acceptance* problem on the stable model semantics ($\mathbf{Cred}_{stable}^{m}$) remains for *monotone pForm-ADFs* the same as for *pForm-ADFs* with the assumption of the link type knowledge. In this way we want to show that it is slightly easier to solve the problem for *monotone pForm-ADFs* because they do not need the assumption of the link type knowledge. In addition we also want to show that there is no reason that the complexity decreases as a result of the restriction to monotone formulae.

Proposition 4.4.3. $\mathbf{Cred}^{m}_{stable}$ *for* monotone pForm-ADFs *is* $\mathbf{NP}$*-hard.*

Proof. Every Dung's *AF* can be transformed to a *pForm-ADF*. As all variables occur as pure attacks, every acceptance condition formula is a conjunction of literals with negative polarity, which is a *monotone pForm-ADF*. So the hardness results on Dung's *AF* (see [Dimopoulos and Torres, 1996]) carry over to *monotone pForm-ADFs*. □

Proposition 4.4.4. $\mathbf{Cred}^{m}_{stable}$ *for* monotone pForm-ADFs *has an* $\mathbf{NP}$*-membership.*

Proof. Let $D = (S, AC)$ be a *monotone pForm-ADF* and $s \in S$ be a statement in D. Guess a model $M \subseteq S$, where $s \in M$. The check if M is a model corresponds to evaluating if a given interpretation is a model for propositional formulae. The construction of D^M is computationally easy: substitute in every formula all variables which have a negative polarity occurrence in the formula (see Proposition 3.1.19). The computation of the (unique) least fixed-point of D^M by the application of Th_D also takes maximal $m \in M$ many steps, where each step is a truth-value evaluation of propositional formulae under a given interpretation. □

CHAPTER 5

Implementation

This chapter will emphasize on the implementation of software systems to compute the sets of accepted statements, defined by the semantics, for specific instances. In addition we present experiments for the evaluation of the performance. At first we describe how the encodings for the systems are done (Section 5.1) and in Section 5.2 the results of the experiments are shown and discussed.

5.1 ASP - Encodings

As a preliminary we will first introduce the paradigm of logic programming with strong emphasis on Answer Set Programming. In the following we will present an implementation of the ADF $\rightarrow$ AF transformation [Brewka et al., 2011a] (see Section 4.1). After this rather small program, we will present and describe our new system `ADFsys` [Ellmauthaler and Wallner, 2012]. Note that we only show parts of the whole encodings in this chapter. For the full listings see Appendix A and B.

Introduction to Logic Programming

The paradigm of logic programming has the aim to describe in a declarative way the facts and inference rules. With this description all inferences which satisfy these declarations can be determined. A slightly older, but highly sophisticated and well-known approach is LOGPROG (for details see [Apt, 1997]). In LOGPROG the declaration of propositional theories consisting of variables, predicates and functions is possible. Alas, to achieve efficient programs in LOGPROG a programmer needs to use complex propositional data-structures which have tendencies to become unintuitive. In addition it is not always declarative, as there exist constructs where the order in the source code matters.

The paradigm of Answer Set Programming - ASP (for an overview see [Brewka et al., 2011c]) does not try to provide that much freedom for data structures compared to LOGPROG. So it works generally on a flat data-structure and so it does not allow function symbols in its

core version. The basic idea behind ASP is to solve computational hard problems with a declarative approach, which is modular such that the problem definition and instances are two different components where each component may be changed.

Let us begin with basic definitions for the ASP programming paradigm: We begin with the propositional case, as answer set programming allows in addition the usage of predicates. Note that despite predicates are allowed no other concepts of first-order logic like quantifiers are allowed. An ASP program consists of a set of rules. A rule is an expression of the form

$$a \leftarrow b_1, ..., b_m, not\ c_1, ..., not\ c_n.$$

where a, all b_i, and all c_i are atoms. The rule consists of two parts, the so-called *head* and the *body*, separated by the $\leftarrow$-symbol. Intuitively a rule can be read that the head (a in the above rule) can only be derived if the body ($b_1, ..., b_m, not\ c_1, ..., not\ c_n$ in the rule) is satisfied by the already derived knowledge. Note that the not prefix before the c_i atoms is not a negation in the classical logic way. It can be seen as some sort of nonmonotonic negation, as it means that c_i can not be derived. An atom a is a positive literal and $not\ a$ is a "default" negated literal (i.e. weak negation). A special form of a rule is the rule in the form

$$a \leftarrow .$$

where a is not dependant of any other atom and always true. In this case the $\leftarrow$ can be omitted and we may write

$$a.$$

which is called a *fact*. A further natural idea is to use an empty head, such that

$$\leftarrow b_1, ..., b_m, not\ c_1, ..., not\ c_n.$$

is the form of this rule. Intuitively this can be seen as a rule which infers $\bot$, which is the same as a contradiction. This type of rule is called a *constraint* and it can be used to identify impossible, unwanted, or contradicting inferences.

A set of rules is called an ASP program, which results in potentially multiple results, so-called *answer sets*. An answer set is a set of atoms which are the derived inferences and the knowledge base together. Due to the allowed usage of weak negation it is possible to create negation loops in rules such that different answer sets can be derived (see the example below).

Example 5.1.1 (ASP rules with more than one answer-set)**.**

$$r_1 :\ a \leftarrow not\ b.$$

$$r_2 :\ b \leftarrow not\ a.$$

In this example we have two rules. If we take a look on the program $p = \{r_1\} \cup \{r_2\}$, we see that neither a nor b is derived and so we can apply both rules to derive additional atoms. If we derive r_1 first, then we can not derive r_2 because now a exists. The same holds for the other case where we first have derived r_2. So the program p would provide two answer sets: $\{\{a\}, \{b\}\}$

It can get quite complex to deal with such negations, because during the collection of new derived knowledge some already used (and before applicable) rule will no longer be applicable and its knowledge is no longer derivable and therefore the answer set may be inconsistent. This means ASP is a non-monotonic formalism. Answer sets which are stable can be computed via the Gelfond-Lifschitz reduct [Gelfond and Lifschitz, 1988]. The reduct is built for every model M of the program Π. There all rules with at least one negated literal which is also in the model M are removed. In the remaining rules all negated literals are deleted from the bodies. So a negation-free, reduced program is constructed which has a unique least model. If this model of the reduced program equals M, then M is stable.

We now consider in addition predicates and variables, i.e. an atom can now be a predicate $p(t_1, ..., t_n)$ with arity $n \geq 0$. The terms $t_1, ..., t_n$ can either be constants or variables. In the programming paradigm, we will use for facts (atoms, constants and predicate names) lower case letters and variables for general rules are denoted by names beginning with an upper case letter. So a rule $a \leftarrow p(X)$. will infer a if any predicate $p(c)$ exists, where c is a constant. To handle variables it is important for ASP to solvers to get a *grounded* input. This means that the variables are substituted by all possible constants. Indeed in a naive way this would end in an exponential blowup, so the specialized grounders utilize many tricks to produce as compact and small as possible grounded versions of the program. As we need to use a grounder before we can use these problems, also all rules need to be groundable. So every weakly negated atom (i.e. those that have a *not* before them) also needs to occur positively in the body. If we are not interested in some of the variables in a relation, we may use the underline (_) to express this. (e.g. $a(X) \leftarrow rel(_, X)$ creates the fact $a(X)$ if there exists any relation $rel(Y, X)$, but we do not care which value Y has. Note that there may be more than one relation which can satisfy this rule, but this does not matter because we are dealing with sets).

The paradigm we have shown till now has the computational power to solve problems in $\mathbf{NP}$. Now we will present extensions of this concept, where some of these methods may increase the computational power to higher levels of the polynomial hierarchy. The first extension we want to introduce are aggregates together with optimization. With aggregates the number of relations can be counted. These aggregates can be used in a way like they are used in SQL. $\#count\{a, b, c\}$ is one example, which would give the answer 3. As variables and underlines are allowed to occur, it is possible to count the number of relations which has something in common (e.g. $\#count\{rel(_, X)\}, X$. may count how many elements are related to X). The results of these aggregates may be used to refine rules with boundaries in the form of $L\{number\}U$ to only make the body applicable if the number is between the boundaries L and U (e.g. $a \leftarrow 2\{\#count\{rel(_, b)\}\}5$. may only derive a if there are $2, 3, 4$, or 5 relations where b occurs as the second element). In addition to the usage of aggregates for more distinct definitions of rules, it is also possible to add weights to some derived facts and let the ASP solver find a solution with a minimal, respectively optimal, score with respect to the weights. The second extension we want to present is the extension to *disjunctive logic programs*. There it is allowed to use rules with disjunctions in the head:

$$a_1 \vee ... \vee a_i \leftarrow b_1, ..., b_m, not\ c_1, ..., not\ c_n.$$

The first intuition says that these disjunctive programs do the same as the construct shown in

Example 5.1.1. With disjunctive programs stronger properties for the elements in the head are enforced. The above example may be extended with additional rules to have a, b and both together in one answer set. In contrast to the approach with weak negation loops, the disjunctive program only computes answer sets which are subset-minimal with respect to the elements in the disjunctive rule head.

Note that ASP solves problems which are known to be in $\mathbf{NP}$ or harder. With a special coding technique, the *saturation* (see [Eiter and Gottlob, 1995] for additional insight on saturation), it is possible to also solve $\mathbf{coNP}$ problems. We will use saturation for some of the implementations and we will describe it when we use it the first time. Notice that the use of disjunctive programs already increases the expressiveness of the logic program to $\Sigma_2\mathbf{P}$.

Introduction to the used Solver

The below described encodings are done with the Potsdam Answer Set Solving Collection (`Potassco`[1]) [Gebser et al., 2011b]. We have decided to use this package of software, because it is one of the best solvers currently available [Calimeri et al., 2011, Denecker et al., 2009, Gebser et al., 2007b]. In addition some nice features like an embedded `LUA`[2] [Ierusalimschy et al., 2007] code processor to generate additional information with an imperative approach are included too. Here the idea is to call an imperative function when a rule is applied and use the return value of this function as the derived knowledge.

We have used a selection of the tools from `Potassco`. To translate the ASP-program to a grounded one, we use `gringo`. It creates an `lparse` conform output, which is needed as the input for the solver of the `Potassco`-package. The used solver is `claspD`, which is `clasp` [Gebser et al., 2007a] together with the power to work with disjunctive programs. In addition we use `metasp` [Gebser et al., 2011a] to get the subset-minimal property of the preferred semantics.

Each solver has some own additions to the syntax of the language to define the ASP-program. So we will give a short overview on some of the needed standards we have used. In general we have to use `:-` instead of $\leftarrow$ to separate the head from the body. For better readability we will use in the listing $\leftarrow$ although it has to be the double-dot followed by the dash. For conjunctive programs we have to use $|$ for the $\vee$ symbol. Again we will print in the listings the $\vee$ instead to achieve a more convenient presentation. The symbol $\%$ is used to handle the line from the symbol till its end as a comment .

To use the mentioned `LUA`-code, we need to define a region for it. The listing shows how such a `LUA`-code may be embedded. Here we just define the LUA function *foo* which does return anything. In addition line 6 shows how this function is invoked. If b is derived as a fact, then we will derive $a($`returnvalue_of_foo`$)$. Note that the return value of *foo* is not cached, so the function is invoked every time it occurs. The last line shows another rule where a clasp-specific syntax is used. In the first predicate of the body we see a semicolon. This is a shorthand to reduce the number of predicates. This line can also be written as

$$ismodel(F) : -subformula(F), subformula(F1), F := neg(F1), nomodel(F1).$$

[1] see http://potassco.sourceforge.net/ for the system and further documentation

[2] see http://www.lua.org/ for the language definition

```
1 #begin_lua
2   function foo()
3      return anything
4   end
5 #end_lua
6 a(@foo) ← b.
7 ismodel(F) ← subformula(F;F1), F:=neg(F1), nomodel(F1).
```

Listing 1: Clasp specific syntax

which needs much more space than the above notation. The used `:=` is the symbol for an assignment. So F needs to be $neg(F1)$ to get the rule applicable. An additional handy syntax pattern is the colon and the count-shortcut (instead of writing the count aggregate it is sufficient to write curly brackets). The expression `{rel(S,X):fact(S)},X:=foo` would be the number of relations relations $rel(S, X)$ for the facts S, where X is foo.

We need to use $\#maximize[e@p]$ and $\#minimize[e@p]$ to utilize the optimization extension of clasp. Here e is some element, which will be counted (e.g. $p(X)$ to count the number of derived predicates p), and p is a number which declares the priority of the optimization. The optimization will first be done for the highest priority and then those sets which are maximal are optimized for the next smaller priority and so on.

At last we want to give a short overview on how to use the programs in form of an example.

Example 5.1.2 (Usage `gringo` and `clasp`)**.** *Let `program1.dl` and `instance1.dl` be ASP-programs. Then the answer sets are computed with the following command from a commonly used shell:*

```
$> gringo program1.dl instance1.dl | claspD 0
```

As already said we need to ground the programs first to get them solved by `claspD`, so we called `gringo` first and then we used its output as the input for `claspD`. Note that `claspD` is the solver for disjunctive programs. It can also solve programs without disjunctive rules, so we do not change between `clasp` and `claspD` calls. The zero after the program call is an argument to `claspD`, which forces the enumeration of all answer sets instead of just the first one found.

ADF → AF

This section will describe how the encoding of the ADF→AF-transformation is implemented and give an insight on the used coding techniques and mechanics. This implementation was done as a preliminary study on *ADFs*. The study had the aim to empirically test whether a native encoding of the model semantics for *ADFs* performs better than the transformation to *AFs* together with the computation of the stable extension or not. For easier computation of the acceptance conditions we restricted them to *CNF*. Based on this easier representation of

acceptance conditions, we will present the native encoding for the model semantics for these *ADFs*, together with the transformation to *AFs*.

To represent a *pForm-ADF*, we need to use the following encoding.

Definition 5.1.3 (Representation of the statements)**.** *Let* $D = (S, AC)$ *be a* pForm-ADF*. The ASP-program* $\hat{S}$ *contains for each statement one fact such that* $\hat{S} = \{statement(s). \mid s \in S\}$.

Definition 5.1.4 (Representation of the acceptance conditions)**.** *Let* $D = (S, AC)$ *be a* pForm-ADF*, where the acceptance conditions are represented in clause form. In addition let* π *be a mapping to get an unique name for each clause over all acceptance conditions.*

$$\begin{aligned}\hat{AC} &= \{cl(s, \pi(c)). \mid s \in S, c \in AC_s\}\\ \hat{C}_{pos} &= \{pos(\pi(c), a). \mid c \in AC_s, a \in c, a \textit{ is a positive literal}\}\\ \hat{C}_{neg} &= \{neg(\pi(c), a). \mid c \in AC_s, a \in c, a \textit{ is a negative literal}\}\\ \hat{A} &= \hat{AC} \cup \hat{C}_{pos} \cup \hat{C}_{neg}\end{aligned}$$

The program $\hat{A}$ *represents the clause form of each acceptance condition in* D*.*

Based on the two programs to represent the statements and the acceptance conditions the program $\hat{E} = \hat{S} \cup \hat{A}$ can be used to encode the facts about one *pForm-ADF*. Note that the truth-constants $\bot$ and $\top$ can not be used as symbols, but omitting the literals $\hat{C}_{pos}$ and $\hat{C}_{neg}$ equals an empty set of literals in the clause form and omitting the clauses $\hat{AC}$ for one statement equals an empty set of clauses in the clause form. So it is possible to project these two truth-constants for the acceptance condition.

Now we will explain how the direct encoding of the model for the presented representation of *pForm-ADFs* is done. Listing 2 shows the whole source of the direct encoding. Here we use a *"guess & check"* approach. Lines 1 and 2 incorporate the guess, where for each statement it is guessed whether it is in the model or not. Based on the introduction to ASP it should be clear that every subset of the set of statements is guessed as a possible answer set. In the lines 9 and 10 the constraints check whether the guessed answer set fulfills the desired property or not. So only the "right" sets stay as answer sets. Before this check can occur we derive which clauses validate $true$ under the current guess (line 4 and 5). Then we infer for each formula the predicate `fconj` if it is $false$ under the current guess. Based on this inference now the two constraints can check whether a statement was selected whose acceptance condition validates to $false$ or a statement was not selected although its acceptance condition has the truth-value $true$ under the current guess.

Note that we use the already existing software system ASPARTIX[3] [Egly et al., 2010] to compute the stable extension of the *AF*. Therefore we will follow their representation of *AFs*, where `arg(x)` is the definition of one argument x and `att(a,b)` projects that a attacks b.
To encode the transformation, a creation of new arguments is needed to get the negated arguments and the helping nodes for the logical connectives. To obtain this in an easy to read and understandable way, we use the underline "_" as the prefix for a negated argument. In addition

[3]see http://rull.dbai.tuwien.ac.at:8080/ASPARTIX for a web front-end of ASPARTIX

```
1  in(X) ← not out(X), statement(X).
2  out(X) ← not in(X), statement(X).
3
4  vdisj(X,Y) ← cl(X,Y), pos(Y,Z), in(Z).
5  vdisj(X,Y) ← cl(X,Y), neg(Y,Z), out(Z).
6
7  fconj(X) ← cl(X,Y), not vdisj(X,Y).
8
9  ← in(X), fconj(X).
10 ← out(X), not fconj(X).
```

Listing 2: Direct Encoding of the model semantics

the "*h_*" prefix is used to identify helping arguments which are needed for the transformation. To generate this new knowledge we used the embedded LUA-engine. Listing 3 shows how these functions are built. Line 3 to 9 captures the function to create a negated atom and also resolves double negation such that the negation of a negative literal is again a positive literal. The used `Val`-object is some kind of interface to the solver to add the new knowledge. The function `dhelper` is used to create a new argument for the disjunction in one clause. Note that there also exist similar functions for the other elements of the created arguments during the transformation, but they are in fact the same as the `dhelper` routine.

```
1  #begin_lua
2  local i=-1
3  function neg(s)
4    if (string.sub(Val.name(s),1,1)) == "_" then
5      return Val.new(Val.ID, string.sub(Val.name(s),2))
6    else
7      return Val.new(Val.ID, "_"..Val.name(s))
8    end
9  end
10
11 function dhelper(s)
12   i=i+1
13   return Val.new(Val.ID, "h_v_" .. i)
14 end
15 #end_lua.
```

Listing 3: LUA-functions for the creation of arguments

Listing 4 shows the program part with the rules to create the arguments which correspond to the statements of the *ADF*. In addition the negated arguments are inferred too. Lines 4 and 5 resolve the number of clauses (`numcon`) and the count of literals in each clause (`numdis`).

```
1 arg(X) ← statement(X).
2 arg(@neg(X)) ← statement(X).
3
4 numcon(X,Y) ← statement(X), Y:=#count{cl(X,_)}.
5 numdis(Y,Z) ← statement(X), cl(X,Y),
     Z:=#count{pos(Y,_),neg(Y,_)}.
6
7 verum(X) ← statement(X), numcon(X,Y), Y==0.
8 falsum(X) ← statement(X), cl(X,Y), numdis(Y,Z), Z==0.
```

Listing 4: Rules for the statement transformation

With this counts we can identify how the structure of the acceptance condition formula looks like. The knowledge about the structure is used in line 7 and 8 to infer predicates for statements which have a truth-constant as their truth-value.

Based on the transformation we know that we have to check whether the acceptance condition formula is based on variables or truth-constants. In the latter case only a unidirection attack relation between the argument and its negated version is added. These additions are shown in Listing 5 at the first two lines. The other two lines in the listing create a bidirectional attack relation between the two arguments.

```
1 att(X,@neg(X)) ← verum(X).
2 att(@neg(X),X) ← falsum(X).
3 att(X,@neg(X)) ← statement(X), not verum(X), not falsum(X).
4 att(@neg(X),X) ← statement(X), not verum(X), not falsum(X).
```

Listing 5: Attack relations between arguments and counterarguments

Next, we will have a look on the transformation of the relation between two statements where the acceptance condition consists only of one literal. This is the last case where we do not need any additional helping arguments (see Listing 6). The first two rules infer on basis of the count of the clauses and literals the knowledge that there is only one literal which is negative respectively positive. Then in the following lines the attacks between the arguments and their negated arguments are done as described in the section about the transformation.

At last the rules for acceptance conditions with compound formulae in CNF are missing. Here we need to use the both structures introduced by the transformation. A more detailed look on the structures for conjunction and disjunction shows that both structures are identical in the arrangement of the attacks and differ in the position of the arguments. So a generic attack structure for these two is used. Based on the arguments passed as a generic attack, other arguments are put in place for the structure (Listing 7). Now the following rules only need to use this generic attack predicate to build one semantic attack structure. To have a simpler way to deal with positive and negative literals the predicate `lit` is introduced, where the argument respectively the negated argument is used. This predicate is just a substitution for the `pos` and

```
1 neglitatt(Z,X) ← statement(X), cl(X,Y), numcon(X,A),
     numdis(Y,B), neg(Y,Z), A==1, B==1.
2 poslitatt(Z,X) ← statement(X), cl(X,Y), numcon(X,A),
     numdis(Y,B), pos(Y,Z), A==1, B==1.
3
4 att(Z,X) ← neglitatt(Z,X).
5 att(@neg(Z),@neg(X)) ← neglitatt(Z,X).
6 att(@neg(Z),X) ← poslitatt(Z,X).
7 att(Z,@neg(X)) ← poslitatt(Z,X).
```

Listing 6: Attack relations between arguments with single literals as acceptance conditions

```
1 att(H,T) ← genatt(I,T,H).
2 att(I,H) ← genatt(I,T,H).
3 att(I,@neg(T)) ← genatt(I,T,H).
```

Listing 7: Generic attack relations

`neg` predicates. To reduce the number of needed arguments, we have a distinction of the shape of the acceptance condition formula. This means that we analyze whether there exists only one clause (i.e. the formula only has disjunctions), or there are more than one clause existing. In the latter case a distinction is done between those clauses which only contain one literal and those who contain more. Listing 8 shows these distinct rules. Intuitively spoken the rules produce new helper arguments (`dhelper`, `chelper`, `disjunct`) which are connected with the arguments via generic attacks. In the last two lines the attacks between the newly created arguments and their negations are added, as they could not been inferred before.

ADF System

Here we will introduce our software system `ADFsys`[4] [Ellmauthaler and Wallner, 2012] which offers encodings to compute the already discussed and presented semantics of *pForm-ADFs*. To make it easier to see which listings belong to each other, we will continue the line numbering for closely related parts. For the encodings we removed the restriction of the acceptance condition formulae to CNF. So every formulae may be used as an acceptance condition. Therefore the input format is different from the format for the transformation. The encoding of input instances (i.e. *pForm-ADFs*) is as follows:

Definition 5.1.5 (Representation of a *pForm-ADF* for `ADFsys`)**.** *Let* $D = (S, AC)$ *be a* pForm-ADF*. The ASP-program* $\hat{S}$ *contains for each statement one fact, such that*

$$\hat{S} = \{statement\{s\}. \mid s \in S\}.$$

[4]see http://www.dbai.tuwien.ac.at/research/project/argumentation/adfsys/ for sources and some examples

```
1  % if the acceptance condition only consists of a serie of
       disjunctions
2  datt(@dhelper(T),T,X) ← statement(T), cl(T,X), numcon(T,A),
       numdis(X,B), A==1, B>1.
3  genatt(I,T,H) ← datt(H,T,X), lit(X,I).
4
5  % if the acceptance condition has at least one conjunction
6  catt(@chelper(T),T) ← statement(T), not falsum(T),
       numcon(T,A), A>1.
7
8  % if the subformula from a catt conjunction is only one
       literal
9  genatt(@neg(I),@neg(T),H) ← catt(H,T), cl(T,X),
       numdis(X,B), lit(X,I), B==1.
10
11 % if the subformula from a catt conjunction consists of a
       serie of disjunctions
12 dincatt(@disjunct(X),X) ← catt(H,T), cl(T,X), numdis(X,B),
       B>1.
13 datt(@dhelper(T),T,X) ← dincatt(T,X).
14 genatt(@neg(I),@neg(T),H) ← dincatt(I,X), cl(T,X),
       catt(H,T).
15
16 % create the mutual attacks for the "disjunct"-nodes
       (dincatt)
17 att(X,@neg(X)) ← dincatt(X,_).
18 att(@neg(X),X) ← dincatt(X,_).
```

Listing 8: Generic attacks for different shapes of acceptance condition formulae

The ASP-program $\hat{A}$ represents the acceptance conditions, such that

$$\hat{A} = \{ac(s, ac_s) \mid s \in S, ac_s \in AC\}.$$

For the acceptance condition formula every formula can be used, which is a formula by the following induction:

(i) Every $s \in S$ is an acceptance condition formula.

(ii) c(v) and c(f) are acceptance condition formulae.

(iii) If ϕ is an acceptance condition formula, then $neg(\phi)$ is one too.

(iv) If ϕ and ψ are acceptance condition formulae, then these are one too:

- *$and(\phi, \psi)$,*
- *$or(\phi, \psi)$,*
- *$imp(\phi, \psi)$,*
- *$xor(\phi, \psi)$, and*
- *$iff(\phi, \psi)$.*

The ASP-program $\hat{I}$ is the representation of one pForm-ADF*, where $\hat{I} = \hat{S} \cup \hat{A}$.*

Note that the atoms `c(v)` and `c(f)` stand for the constant-symbols verum (i.e. $\top$) and falsum (i.e. $\bot$).

Formula representation and evaluation

Before we will explain how the distinct semantics are computed we will first show how the handling with the acceptance condition formulae is done and how it can be checked whether a given interpretation is a model for a formula or not. The predicates to construct the given formula act in a way like function symbols. Listing 9 shows how the acceptance condition formula is broken down into its subformulae recursively. Line 1 takes the acceptance condition as the first

```
1 subformula(X,F) ← ac(X,F),statement(X).
2 subformula(X,F) ← subformula(X,and(F,_)).
3 subformula(X,F) ← subformula(X,and(_,F)).
4 subformula(X,F) ← subformula(X,neg(F)).
```

Listing 9: Acceptance condition and its subformulae

"subformula". Note that the `subformula`-predicate keeps track of the original statement to which the formula and its parts belong to. In the next lines the formula is split into its parts, and for each part a new subformula is inferred. Keep in mind that we only show how this concept works on the conjunction and the negation, as the other connectives work in the same manner. After all predicates are resolved we have intuitively spoken the whole formula tree as our knowledge.

With this formula tree it is now easy to check whether a set of statements satisfies the acceptance condition formulae or not. The distinct rules for conjunction, disjunction, and negation, as well as the rules for atoms are shown in Listing 10. To check whether an interpretation is a model for a formula or not, we need some set of statements which are *true* respectively *false*. So we use the predicates `in` and `out` to define which statements are selected and which are not. How these two sets are constructed will be shown later. The listing shows some of the rules to infer those formulae which evaluate to *true* under the current assignment. Indeed there are also rules for the case where they evaluate to *false*. These will infer the predicate `nomodel`, whose definition is omitted to reduce redundancy. Lines 5 and 6 deal with formulae which are atoms. If the atom itself is selected or the atom is the truth-constant verum it is obvious that the selection is a model for this atom, meaning this subformula evaluates to true. The following rule

```
5  ismodel(X) ← atom(X), in(X).
6  ismodel(X) ← atom(X), X:=c(v).
7  ismodel(F) ← subformula(F;F1), F:=neg(F1), nomodel(F1).
8  ismodel(F) ← subformula(F), F:=and(F1,F2), ismodel(F1;F2).
9  ismodel(F) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F1).
10 ismodel(F) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F2).
```

Listing 10: Verification whether a set of statements is a model or not

deals with the negation. Intuitively the rule fires and induces new knowledge if there are two subformulae `F` and `F1`, where `F` is the formula `neg(F1)`, and the current selection of elements is not a model of `F1`. To resolve whether a conjunction evaluates to true, line 8 takes the formulae which are conjunctions and demands both of the subformulae to be true under the current statement assignment too. For the disjunction we need two rules as it is satisfied if at least one of the two subformulae validates to $true$.

Link-type distinction

Some of the semantics are defined for *BADFs* and so we will need a link-type distinction. This computation is also implemented and before we will picture the semantics we show how this encoding works. We will have to check every link. The property of being attacking respectively supporting is dependent on two interpretations, so we also need to distinct between these two evaluations. To have a simple method to check whether the interpretations are models or not, we revamped the `ismodel` and `nomodel` rules such that each predicate is defined for one formula with respect to a link and the current evaluation. The link predicates `link(X,S)` represent one link, where the statement `X` occurs in the acceptance condition of `S`. The problem to decide whether a link is attacking (resp. supporting) or not is in $\mathbf{coNP}$, so we need to use the *saturation* coding technique.

The basic idea behind this technique is to use a disjunctive rule to guess the sets where we want to show the $\mathbf{coNP}$-hard property. Then we have to infer a predicate with positive rules (i.e. no weak negations occur in the rule) if the property holds in this case. If the property is not satisfied we remove it with a constraint from the answer set. In the case where the property is satisfied, we will saturate the answer set, which means we set all elements to be in and out at the same time. Intuitively this means that we have only saturated sets as answer sets (which are all identical and therefore collapse to one answer set) if the property holds and if the property does not hold, at least one answer set candidate was not saturated and is therefore a minimal set w.r.t. the disjunctive guess. As this minimal answer set has a constraint which removes it, we do get no answer set as the answer of the computation.

Let us go back to the encoding for the link-types. Before we can apply the saturation, we have to guess which type the link is (Listing 11). Based on this guess we can now check for the guess whether it was right or not. Precisely for this check we will need the saturation mentioned above. Listing 12 pictures the saturation for our needs. To keep it simpler we only show the rules for the case where we guessed an attack link. The supporting and uninformative links (i.e. attack

```
1  att(X,S) ∨ supp(X,S) ∨ att_supp(X,S) ∨ dep(X,S) ← link(X,S).
```

Listing 11: Guess which link type the link has

and support) are similar to the attacking (Note that the rules in the lines 8 to 11 have to be copied and adapted for the other link-types). Line 2 is the disjunctive guess to select a set of statements

```
2  in(X,I,J,1) ∨ out(X,I,J,1) ← link(I,J), statement(X),
       atom(X), subformula(J,X).
3  ← in(X,X,J,1).
4  in(X,I,J,2) ← in(X,I,J,1).
5  out(X,I,J,2) ← out(X,I,J,1), X != I.
6  in(X,X,J,2) ← out(X,X,J,1).
7
8  noattclash(I,J) ← link(I,J), ac(J,F), ismodel(F,I,J,1).
9  noattclash(I,J) ← link(I,J), ac(J,F), nomodel(F,I,J,1),
       nomodel(F,I,J,2).
10
11 ok(I,J) ← noattclash(I,J), att(I,J).
12
13 ← not ok(I,J), check(I,J), not dep(I,J).
14
15 in(X,I,J,1) ← ok(I,J), out(X,I,J,1), X!=I.
16 out(X,I,J,1) ← ok(I,J), in(X,I,J,1), X!=I.
```

Listing 12: Saturation for the attacking link property

for the link and then every guess where the linked statement is selected is removed. Then a second set of statements is created, where the link statement is added to the guessed set. For this pair of sets the attack property is checked. If the property holds there is no clash. The guess was right if we check whether the link is attacking and no clash is detected. In the case where the check was not right, the constraint in line 13 removes the guess from the acceptable answer sets. In the last two lines we saturate, i.e. we set all statements to in and out at the same time if the check was right. At last we have to deal with the case where the link is dependent. This is an $\mathbf{NP}$-hard problem and therefore we do not need saturation. If we check for the dependency we try to find a set of interpretations to show that it is neither attacking nor attacking. In order to do that we need an additional guess for a second interpretation. Note that we use the guess for the saturation together with a second, identical guess to gain the two interpretations. The guess together with the check for the dependency can be found in Listing 13.

Conflict-free sets and the model semantics

After the description of the preliminary encoding parts, we can continue with the semantics. Naturally the easiest property and semantics have also the shortest and easiest encodings. So

```
17  in(X,I,J,3) ∨ out(X,I,J,3) ← link(I,J), dep(I,J),
        statement(X), atom(X), subformula(J,X).
18 ← in(X,X,J,3).
19  in(X,I,J,4) ← in(X,I,J,3).
20  out(X,I,J,4) ← out(X,I,J,3), X != I.
21  in(X,X,J,4) ← out(X,X,J,3).
22
23  isdep(I,J) ← link(I,J), ac(J,F), ismodel(F,I,J,1),
        nomodel(F,I,J,2), nomodel(F,I,J,3), ismodel(F,I,J,4).
24 ← dep(I,J), not isdep(I,J).
```

Listing 13: Dependent link check

we will start with the conflict-free sets and the model semantics. Both are based on the "guess & check" technique. In Listing 14 we start with a guess of selected statements. The constraint in line 4 removes all sets where a conflict emerges (i.e. a statement is selected although its acceptance condition is not satisfied). Without the last line we obtain the encoding for the

```
1  in(X) ← not out(X), statement(X).
2  out(X) ← not in(X), statement(X).
3
4 ← in(X), ac(X,F), nomodel(F).
5 ← out(X), ac(X,F), ismodel(F).
```

Listing 14: Model computation for an *ADF*

conflict-free sets. This last line is the check to be sure that no statement was forgotten, as every statement whose acceptance condition is satisfied has to be accepted by the model.

Based on the model semantics we can now add the check for the stable model semantics. At first we have revamped the `nomodel` and `ismodel` predicates to give the possibility to evaluate the same formula more than once. So we added the elements to the current subformula formula `F`, the statement `S` to which the formula belongs and the current evaluation `I`. Note that the predicates `in` and `out` now also have to specify in which evaluation step they are selected or not. In addition we have to add `ismodel` and `nomodel` rules to remove attacks. One example of such a rule is presented in Listing 15. The used evaluation step `model` is the evaluation of

```
1  nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I),
       I!=model, att(X,S), in(s,model).
```

Listing 15: One attack elimination rule

the model, as the basis for the stable model reduction. We can see that the rule evaluates to $false$ under the interpretation, even if it was in the model, because it is an attacking link-type.

The removal of links which are not selected is not needed, due to the fact that they are already set to be `out`. After this reduction we need to construct the least model of the reduction. This construction is an iterative procedure and so we will also use an iteration. This reflects the computation of the least fixed-point of the Th_D operator on the model-reduct (see Section 2.3 and 3.1). We start with the number of the statements (Listing 16) and set every statement to be out. Afterwards we check the models for this *empty* interpretation. For every iteration step, the

```
2 modelStatementCount(I) ← I:={in(S,model):statement(S)}.
3 iteration(I) ← modelStatementCount(I).
4 out(S,I) ← statement(S), modelStatementCount(I).
5 modelcheck(I) ← iteration(I).
```

Listing 16: First iteration step

next iteration is created till the last (i.e. evaluation iteration zero) is reached. Listing 17 shows this procedure. In line 7 and 8 we add every statement which had a true acceptance condition in the step before to the set of accepted statements. In addition we copy all the remaining not selected elements to the current iteration. We do not need to check more iterations, as we have to

```
6 iteration(J) ← iteration(I), J:=I−1, I>0.
7 in(S,J) ← iteration(I), iteration(J), J:=I−1,
      ismodel(F,S,I), ac(S,F).
8 out(S,J) ← iteration(I), iteration(J), J:=I−1, not in(S,J),
      out(S,I).
```

Listing 17: Iteration and model construction

add at least one statement per step till the fixed-point is reached. So we will not need more steps than the number of statements. After we have reached and evaluated the last iteration, we add all statements which are selected in this step to the set of accepted elements in the least model. Finally we have to check whether all elements of the model are also in the least model or not (Listing 18).

```
9 in(S) ← in(S,J), J:=0.
10 ← in(S,model), not in(S).
```

Listing 18: Least model and stability check

Semantics based on the stable model semantics

Next we will present the admissible set property together with the preferred semantics. The admissible set needs to work with the two sets R and M. So these two are guessed. In addition we shift the check that no element in R may attack an element in M into the guess method for

```
1  in(X,m) ← not out(X,m), statement(X).
2  out(X,m) ← not in(X,m), statement(X).
3
4  in(X,r) ← not out(X,r), statement(X), in(Y,m).
5  out(X,r) ← not in(X,r), statement(X).
6
7 ← in(X,r), in(Y,m), att(X,Y).
8 ← in(X,r), in(Y,m), att_supp(X,Y).
9
10 in(X,model) ← in(X,m), out(X,r).
11 out(X,model) ← statement(X), not in(X,model).
```

Listing 19: Guess for R and M

R (Listing 19). Based on the guess of the two sets, we can infer the set which has to be checked to be a stable model or not. At the moment where we have a guess for the model, we can reuse the above described encoding for the stable semantics. In order to get the subset maximization to obtain the preferred semantics, we use `metasp`. So we only need to add a maximization for the elements we want to get maximized. As it is easier to define it with minimization we just minimize the inverse set in Listing 20 (i.e. the statements which are out). To use `metasp` the

```
12 #minimize[out(X)].
```

Listing 20: Maximization of `in(X)`

encoding for it needs to be grounded and passed to the solver too. In addition some additional strings need to be inserted in the shell command to tell `metasp` what has to be done. How the commands look alike can be seen on the homepage and will be mentioned in the experiments-section (Section 5.2).

Well-founded semantics

The last semantics we have encoded is the well-founded one. Its definition uses the concept of partial interpretations, so we need a validity test as well as an unsatisfiability test. In addition the function Γ_D is constructed step by step, so we will need an iteration too.

Listing 21 shows how the iteration is tailored. There in every step one undecided statement is guessed and then it is checked whether the statement still belongs to the undecided set, or to the set of accepted respectively rejected statements (see Definition 3.1.26).

For the validity test (which is **coNP**-hard) the saturation technique is used once more (see Listing 22). To check the unsatisfiability we use again the solving power of the solver together with saturation. Due to the fact that the order in which the undecided statements are checked matters some answer sets may result in a subset of the well-founded model. To prevent this we apply the optimization feature of `clasp`. Additional it may be the case that an undecided

```
1  snum(I) ← I:={statement(Y)}.
2  iteration(I) ← snum(J), I:=J−1.
3  iteration(I) ← iteration(J), I:=J−1, I>=0.
4  % create undecided set of variables at the starting point
      of the function
5  undec(X,I) ← snum(I), statement(X).
6  % iterate the function one step further, and guess an
      additional element for A or R
7  inA(X,I) ← inA(X,J), J:=I+1, iteration(I).
8  inR(X,I) ← inR(X,J), J:=I+1, iteration(I).
9  select(X,I) ← not deselect(X,I), statement(X),
      iteration(I), undec(X,J), J:=I+1.
10 deselect(X,I) ← not select(X,I), statement(X),
      iteration(I), undec(X,J), J:=I+1.
11 ← A:={select(_,I)}, iteration(I), A>1.
12 undec(X,I) ← iteration(I), undec(X,J), J:=I+1,
      deselect(X,I).
```

Listing 21: Iteration and guess attempt for the well-founded semantics

```
13 in(X,I) ∨ out(X,I) ← undec(X,J), J:=I+1, iteration(I).
14 in(X,I) ← iteration(I), J=I+1, inA(X,J).
15 out(X,I) ← iteration(I), J=I+1, inR(X,J).
16 okA(I) ← select(X,I), ac(X,F), ismodel(F,I).
17 okA(I) ← A:={select(_,I)}, iteration(I), A=0.
18 inA(X,I) ← okA(I), select(X,I).
19 in(X,I) ← okA(I), undec(X,J), J:=I+1, iteration(I).
20 out(X,I) ← okA(I), undec(X,J), J:=I+1, iteration(I).
```

Listing 22: Acceptance of statements whose partial interpretation is $true$

statement is not checked at all, which is prevented by a second optimization parameter. So only the correct answer set remains.

5.2 Experiments

All experiments were done on an `openSUSE` machine with eight `Intel Xeon` processors (2.33 GHz) and 49 GB memory. For the computation of the answer sets, we used `gringo` (version 3.0.3) and `clasp` (version 2.0.4) for the *ADF* → *AF* transformation tests, and `gringo` (version 3.0.3) together with `claspD` (version 1.1.2) was used for the *ADF* system.

ADF → AF

Preliminary tests indicate that the transformation takes in general more time than the computation of a model in the *ADF* respectively in the *AF*. We have chosen to implement the transformation with a declarative approach to make it easier to verify the program integrity. It is no surprise that the transformation needs a high amount of time due to the fact that almost all work is done by the grounder together with the embedded `LUA` interpreter. Indeed an imperative approach for the transformation may lead to better performance. Our goal was to compare the solving time of an *ADF* model with a corresponding *AF* stable extension. Based on these results we wanted to see whether it is more useful to compute semantics based on *ADFs* or to compute them based on corresponding *AFs*.

Test setup

To test the performance of the computation, we have written a generator to produce random *ADFs* with acceptance conditions in CNF. The randomization is dependent on three parameters:

- **number of statements (s):** The number of statements which are in the *ADF*.
- **max. number of neighbors (n):** This is the number of the maximal allowed different variables occurring in one acceptance condition formula.
- **probability for easy formulas (p):** The probability to produce an *easy* formula, which means that only one literal or the truth constants verum or falsum are the acceptance condition.

Based on these parameters the generator produces formulae with s statements, where each statement has either an *easy* formula (with probability p) or a 3-CNF formula. In the latter case this formula is a CNF where exactly three literals are in each clause. Each generated CNF formula has at most n different variables and the number of clauses equals the effectively used number of variables. To reduce the chance of easy unsatisfying clauses or redundant clauses it is in addition ensured that no variable occurs twice in one clause. Note that the actual used number of variables may be smaller than n, but it is more likely to be close to n. During the instance generation each variable has a 0.5 chance to be used as a positive (respectively negative) literal.

The tests were done with ten instances per class of parameter choices. We have chosen to use the following parameters:

- $s = \{500, 1000\}$
- $n = \{10, 20, 50\}$
- $p = \{0, 10, 20, 50, 80, 100\}$

In addition we reversed the meaning of the input format such that the formula is represented as a DNF and we have adjusted the transformation as well as the *ADF*-model computation to work with the DNF-representation too. This was done to reduce the possibility of biased results based

to the restriction of CNFs. Due to the inverted meaning of the input format we have used the same instances for the DNF and CNF based computations.

To get reasonable results, we limited the maximal computation time for each task to ten minutes. The invocation of the different tasks is done by the following commands, where $> at the beginning of the line is meant to be the command line prompt in a `Unix`-like environment:

```
$> gringo instance model.dl | clasp 0
$> gringo instance transform.dl | clasp 0
$> gringo transformed_instance stable_extension.dl | clasp 0
```

The argument `0` for `clasp` means that all answer sets are computed and not only the first one and the vertical line `|` is the pipe-symbol to pass the output of `gringo` to `clasp` as input. Notice that the output of the transformation is not a valid input format for gringo, so it needs to be adjusted (i.e. add a full stop (.) after each predicate).

Results

Our empirical results showed that most of the generated *ADFs* have exactly one model. Only 8 out of the 360 instances resulted in two models. The following figures will show how long the computation of each class of parameters took on average. On the X-axis of the shown diagrams the probability for complex formulae is assigned and the Y-axis incorporates the average time for the computation of the problem. The solid lines in the diagram represent the data for the native computation of the models on *ADFs* and the dashed lines show the computation times of `ASPARTIX`[5] for the transformed *AF*. In addition each color (i.e. blue, green, and red) has its own symbol (i.e. circle, triangle and cross) which represents the different instances with respect to their maximal number of variables in their acceptance conditions. Note that we have named the different datasets in the key of the diagram by their number of statements and their maximal variables in the acceptance condition. The trailing t symbolizes that this is the transformed *AF*. The figures shall be investigated with caution, because the y-axis has a different scaling to present the different results in a more meaningful way. Each scaling change is denoted by the *zigzag* on the axis. To easily reconstruct the current scaling at each *zigzag* the current value is written down. Between two of these values the scaling is linear.

Figure 5.1 shows the results for all instances with 500 statements, where the acceptance condition formulae are in CNF. The first important observation is the better performance of the native computation of the model compared to the computation of the stable extension on the transformed framework. Roughly speaking almost all model computations were finished in about ten seconds. Indeed the computation on the transformed frameworks took about 300 seconds to complete for the hardest to solve instances. In general it can be said that the computation gets harder the more variables are in the acceptance condition of the formulae. This is also reflected by the increased runtime for all instances where *easy* acceptance condition formulae are less likely to appear.

[5]for further details see Section 6.2

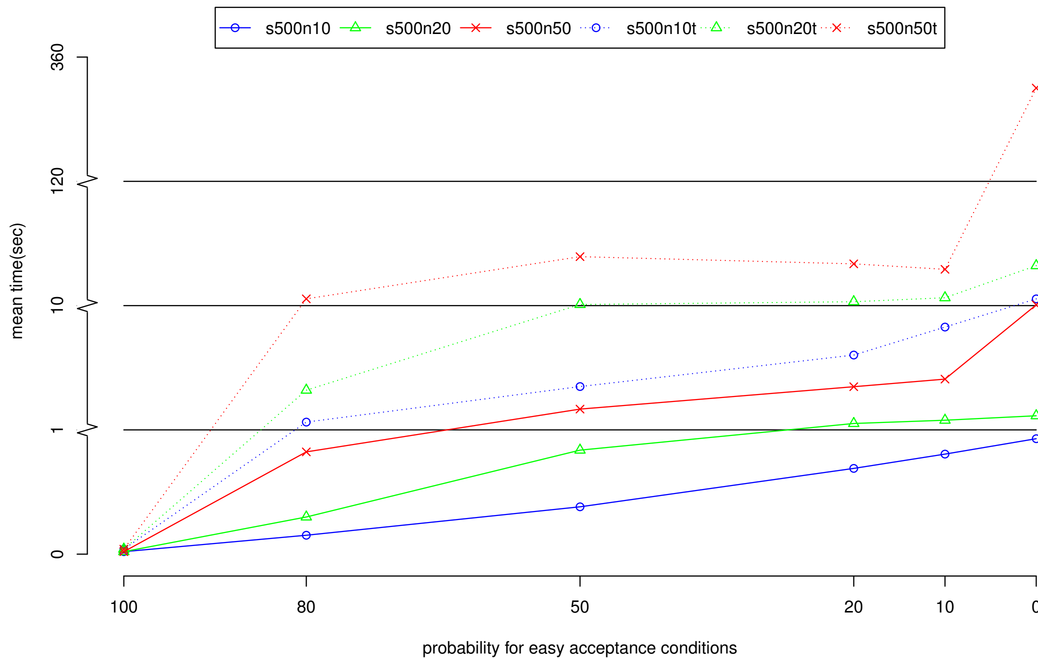


Figure 5.1: Computation times for 500 statements in CNF

The results for the instances with 500 statements, which use DNF acceptance conditions (shown in Figure 5.2) feature a similar picture. In fact all observations for the CNF-case hold in this case too. So it seems that we do not have to face a bias on basis of the chosen normal form.

Finally we want to have a look on the instances with 1000 statements. As the results for DNF and CNF were similar for the 500 statement instances, we have only done the tests with the CNF-representation. In general it seems that the proportion of the computation is similar to the instances with 500 statements. Although the graphs for both instancetypes are related, we have to keep in mind that even the easiest instance took over a second to compute for problems where a small number of *easy* acceptance conditions occur. A very surprising result is seen for the computation times of the instances with the highest number of variables in the acceptance conditions (i.e. $n = 50$). There the average computation time decreases when the probability for easy formulae decreases from 50% to 20% and then it even gets faster in the step to 10%. We speculate that we have some kind of easy to solve cases for the ASP-solver and therefore the computation time gets better. Indeed, the computation time in general seems to get worse, because every instance for 1000 statements, 50 variables and 0% probability for *easy* acceptance conditions exceeded the maximal runtime of ten minutes (600 seconds).

The presented results also relate in some extend to the filesize of the instances compared to

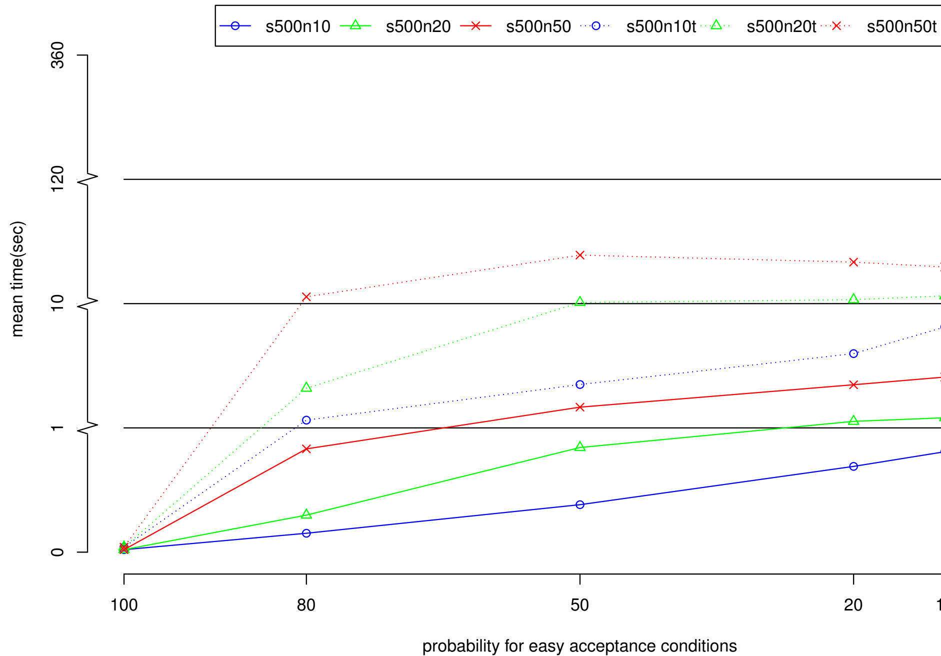


Figure 5.2: Computation times for 500 statements in DNF

the transformed instances. All generated instances together need about 1.2 GB of space, while the transformed *AF*-instances are with 4.5 GB about three to four times bigger. Notice that our transformation process is done on flat formulae structures, so the semantic structures remain very compact. So an implementation with arbitrary formulae would result in more complex semantic structures. The differences in the performance for arbitrary formulae may increase even more, as the semantic structures will get more complex.

ADF System

In the following we will present preliminary tests as a first benchmark of the computational behavior of the `ADFsys` encodings. In contrast to the *ADF* $\rightarrow$ *AF* transformation results, we will differentiate between the grounding and the solving time, to get a better sense on the computational expensive mechanics. For example the whole syntactical handling of the formulae, like subformulae resolution, is purely dedicated to the grounder while the propositional model check and other semantical concepts are almost purely done by the solver.

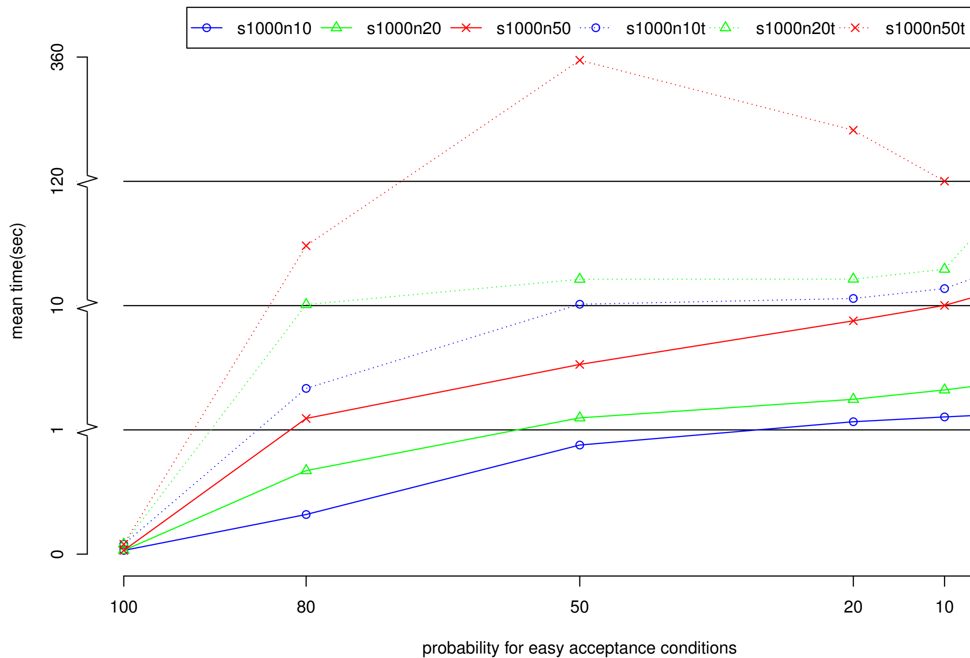


Figure 5.3: Computation times for 1000 statements in CNF

Test setup

In order to get randomized instances we have reused the already existing `8-grid-graph generator`, written for the generation of *AFs* to get benchmarks for the systems `ASPARTIX` and `CEGARTIX`[6]. The *AFs* are generated by arranging arguments in a grid. Then every direct connection between two neighbors has a chance to become an attack between those two arguments. In the 8-grid version of the generator all eight adjacent neighbors in the grid are potential attacking arguments to the center argument. In addition each attack has a chance to be mutual. Figure 5.4 illustrates how this grid concept works and how the adjacent neighbors are defined. This grid consists of 31 arguments, which are arranged in a matrix with a width of 7. The two dark gray arguments are two arbitrary arguments and the direct neighbors are colored in light gray. If the argument is on the edge of the grid, it can be seen that it has less adjacent arguments than those which are not on the edge.

For the generation of *pForm-ADFs* we use the `grid-generator` output for the underlying definition of links between two statements. Based on this graph we add some randomized information to the attack relations such that each attack relation represents one link between two

[6] for further details see Section 6.2

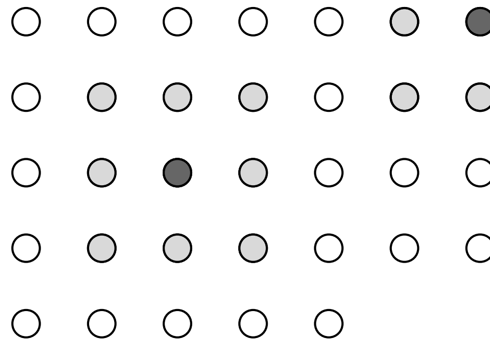


Figure 5.4: Grid-concept of the grid-generator

statements. Intuitively each incoming edge to a statement is one variable in the corresponding acceptance condition formula. How this variable is represented in the acceptance condition is determined by the additional information. So each variable has the chance of 0.5 to be a negated literal. All of the variables are connected via disjunctions or conjunctions and the decision which connective is picked between two variables is also based on a probability. At last in *ADFs* it is possible to have truth constants as variables, therefore there is a chance that a variable is substituted with a constant. The grid structure with the extended information on the links can be easily transformed to the already presented input format for the `ADFsys` encodings. Summarizing there are several parameters from which the generation of one test instance is dependent of:

- number of statements
- grid width
- probability of symmetric links
- probability of a constant substitution
- probability of the use of a disjunction instead of a conjunction

Notice that every *pForm-ADF* generated by this generator is a *monotone pForm-ADF* and therefore a *BADF* (see Theorem 3.1.23).

In order to create a set of instances to get preliminary results, we have chosen the following parameters for the *ADF* generation:

- statements= $\{10, 15, 20, 25, 30\}$
- $\{0, 0.5, 1\}$ for each of the three probabilities

The width of the grid is determined by the number of statements. For 10 statements the width is 3, for 15 statements a width of 4 is chosen, and for the other instances 5 is the used width. Based on the possible parameter combinations we have 135 different instance classes. To get more meaningful results, we created for each instance class 10 different *ADFs*. Based on first

impressions on the execution time we limited the time for one computation (i.e. grounding respectively solving) to five minutes (300 seconds). For our tests we decided to compute the model and the well-founded semantics as well as the stable model. To invoke the different tasks, we used the following commands (again we assume that a `UNIX`-like shell is used, with `$>` as the prompt):

```
$> gringo semantic-encoding.dl instance.dl > grounded_instance
$> claspD 0 -f grounded instance
```

The first command is the invocation of the grounder `gringo` for a specific instance and an encoding for the semantics. The result is stored in the `grounded_instance` file. In the second command `claspD` is used with the parameter `0` to compute all results and `-f` to use a given file as the input source. Notice that we have to compute the linktypes before it is possible to find a solution for the stable model semantics. To use the output of `claspD` again as an input for `gringo` we need to add the period at the end of each fact. We have done this with an `awk`-script, which is a script-language with similarities to `Java`. Although we do not present results for the preferred semantics, we want to mention how it can be invoked. Again we assume that the linktypes are already computed:

```
$> gringo --reify preferred.dl input.dl | \
          gringo - {meta.lp,metaO.lp,metaD.lp} \
          <(echo "optimize(1,1,incl).") | claspD 0
```

Note that we use the files `meta.lp`, `metaO.lp`, and `metaD.lp`. These are part of the `metasp` encoding package and can be downloaded from the `Potassco` site as well.

In the following we will see that the models are computed very fast and without any timeouts, therefore we have created an additional set of instances, where each *ADF* has 100 statements to get a better sense of the computation times for this efficient semantics.

Results

In the following we will present empirical results for the preliminary benchmarks on the 1350 instances. Each semantics is analyzed for its own and we have computed for each number of statements the average computation time. There we omitted all timed out instances. To reflect the failed instances we use the red, dashed line where the data-points are stars. So in general the more instances failed, the higher is the bias on the average values (i.e. the more instances failed the higher the real mean computation time would be if we do not restrict the maximal computation time). The blue lines with a circle denote the data-points for the average computation time of the grounder while the cross notation for the data-points reflects the time for the solver. The overall computation time is denoted by the dark magenta, dotted line with a plus-symbol for the data-points. In the figures the x-axis represents the number of statements while we have up to two different y-axis. The left y-axis denotes the mean computation time in seconds and the right y-axis projects the number of failed instances due to timeouts. Note that we have chosen the number of statements for the x-axis as the collected data suggested that the other parameters do not show the same grade of scaling in the computation time.

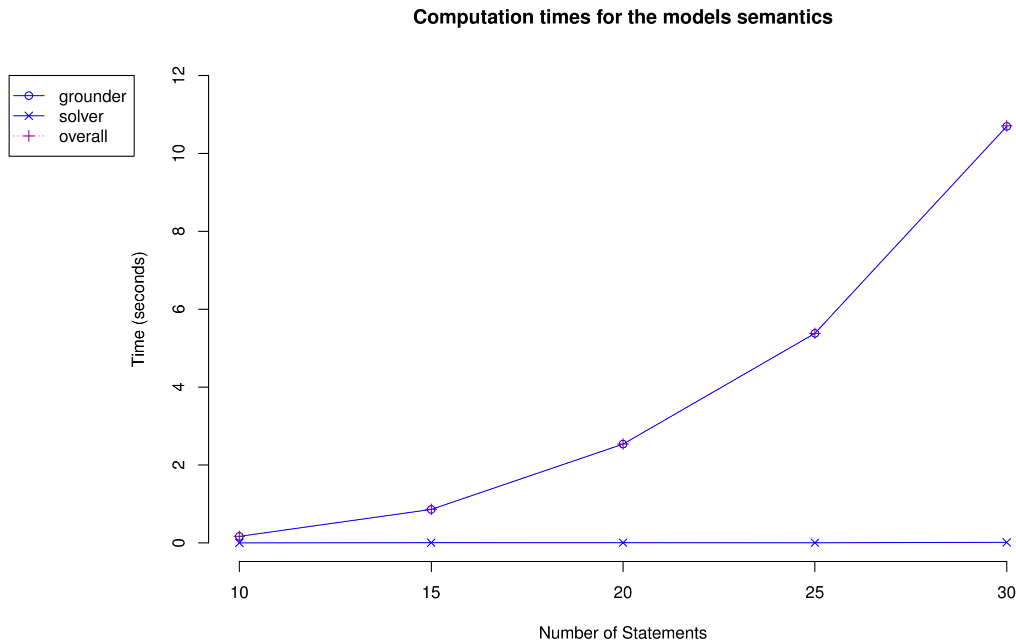


Figure 5.5: Mean computation times for the model semantics

Model semantics. As already mentioned, the model semantics finished very quickly compared to the other semantics. It is the only one where every instance could be solved in the given 300 seconds for each computation step. Figure 5.5 shows the results for the computation times. Notice that the y-axis only shows values between zero and 12 seconds. But do note that the slowest grounding procedure for one of the instances took about 70 seconds. Indeed there are so few instances with this high computation effort such that the mean value is under 12 seconds. It can be seen that most of the effort for the computation comes from the grounding process. In fact many calls of the solver were computed and finished in under 0.1 seconds. The relative high time for the grounding process may come from the formula handling, as every formula needs to be broken down into its sub formulae in order to check if an interpretation is a model or not. This is also reflected in relative big grounded instances compared to the input. So it seems that a lot of information is grounded which results in big, but relatively easy to solve grounded instances. As the computation of 500 and more statements for the CNF-based approach could be done in reasonable time, we also tried to solve *ADFs* with 100 statements and the other parameter choice as before. There the effort for the solving task is also negligible compared to the grounding times. In the 300 seconds 180 of the 270 instances could be solved. The reason why the computation of the models for the `ADFsys` encoding seems to be not as efficient as the encoding used in the $ADF \rightarrow AF$ transformation is that the input format is different. The restriction to use only formulae in CNF makes the propositional model verification easier and

the representation of the CNF in the input file also has a constant formula depth. So there is no need to compute the models of the subformulae. Indeed this is needed for the representation of the *ADFs* in the encoding for arbitrary formulae. Note that the construction of a CNF from an arbitrary formula may be exponential in the resulting size, so the CNF-restriction is not for every case a good idea. In addition an arbitrary formula is typically more human-readable than a CNF.

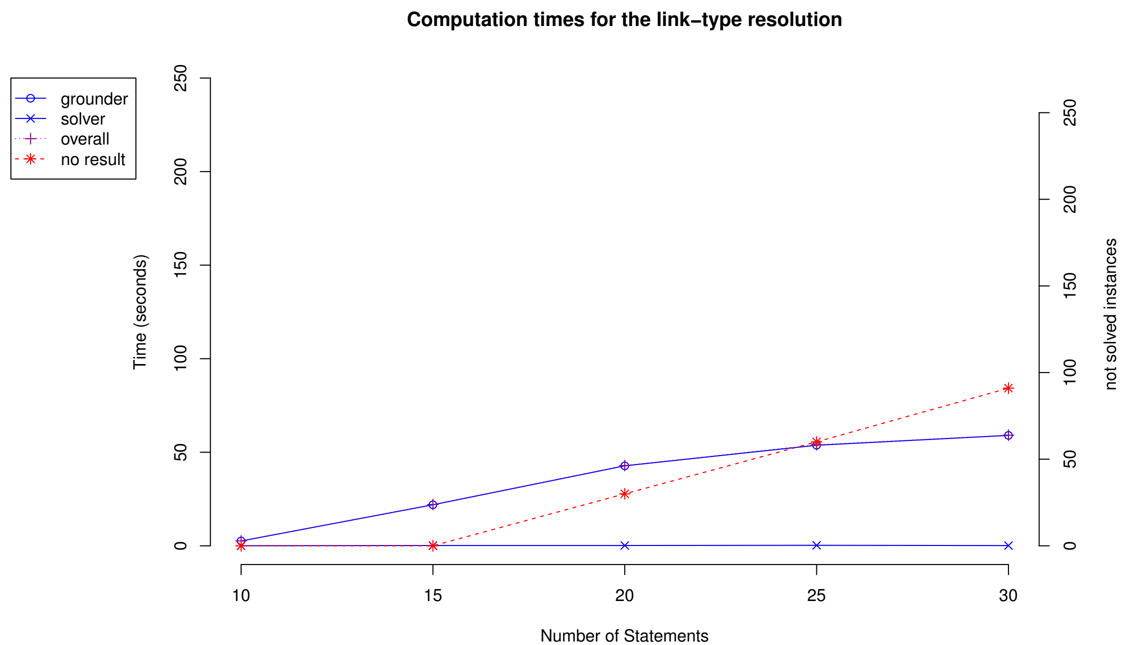


Figure 5.6: Mean computation times for the link-type resolution

Link-type resolution. In order to compute the stable model semantics, we first need to resolve the link-types for a given *ADF*. How this encoding performs is illustrated by Figure 5.6. Again we can observe that the grounding takes most of the computation time. In addition we encounter at 20 statements the first instances where the 300 seconds computation time are not enough for the grounding task. Based on the relatively low mean computation time of under 100 seconds it seems that the instances which encountered a timeout may be particular explicitly hard to solve instances. Indeed, all instances which failed to be computed for 20 statements had no constants and all links where symmetric. So a very high grade of dependency between all the statements is given. For 25 statements most of the timed out instances had the same properties as those for the 20 statements, however some of them had a chance of 0.5 that constants may appear in the acceptance conditions. More or less the same picture can be seen at the 30 statement instances.

In fact there appeared for the first time instances where the mutual dependency of statements have only a chance of 0.5 which timed out.

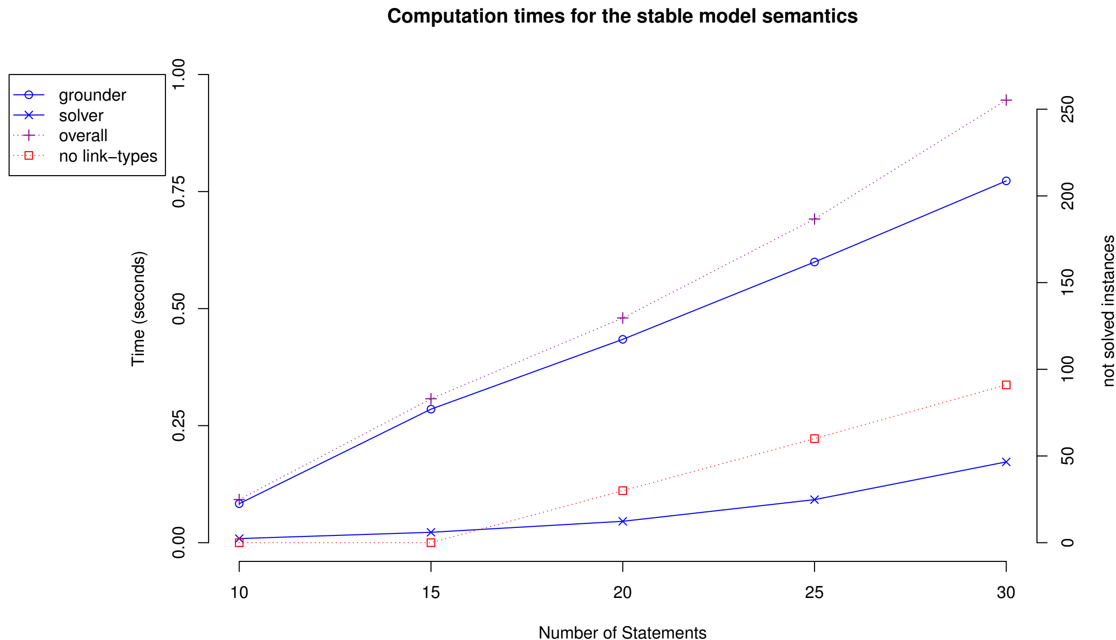


Figure 5.7: Mean computation times for the stable model semantics

Stable model semantics. Based on the link-type resolution we may now compute the stable models for the *ADFs*. As we can not start the computation of some instances due to the lack of a result for the link-types, we will reflect this variable in Figure 5.7. The instances which had no chance to start the computation are denoted by the red dotted line with the square symbol as the data-points. The surprising fact that the average computation for the stable models is within one second can also be explained very well. We have compared the computation times for the models with the computation times for the stable models. In the comparison between them almost all instances which took more than four seconds on the models to finish were instances where the link-type resolution failed ($\sim 92\%$). So some kind of connection between the grounding time for the model semantics and the grounding time for the link-type resolution can be seen. This suggests that the average computation time in Figure 5.7 excludes the difficult instances entirely. Another interesting fact is that the grounding process for the stable model was in many instances ($\sim 50\%$) faster than the process for the model, which is maybe based on some optimization in the grounding process of `gringo` because of the additional rules.

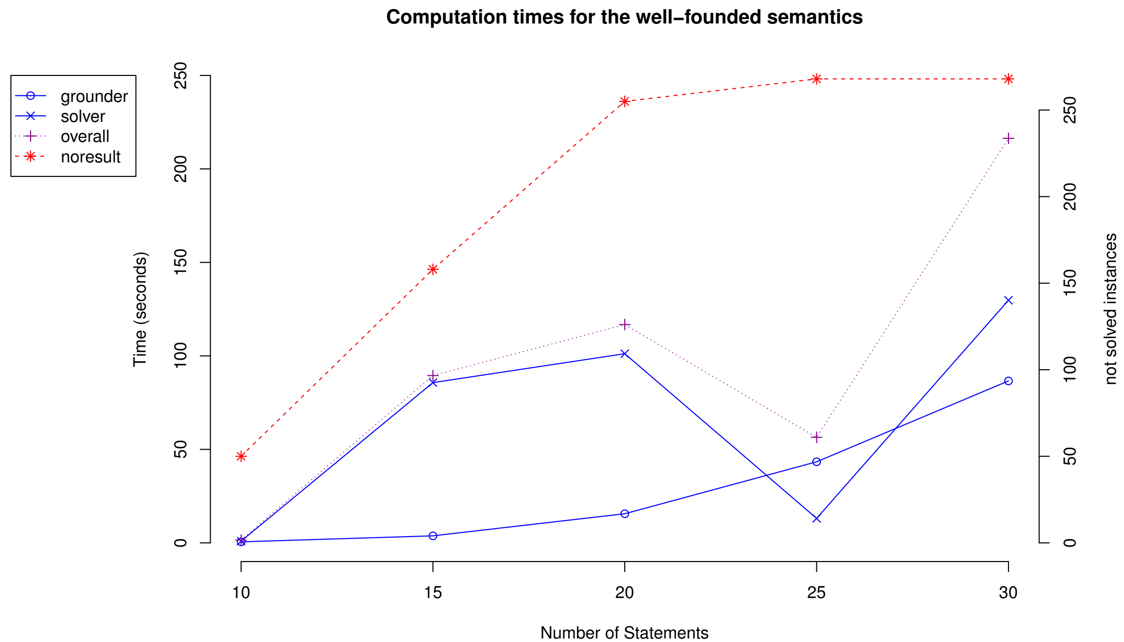


Figure 5.8: Mean computation times for the well-founded semantics

Well-founded semantics. At last we want to present the computation times for the well-founded semantics. In Figure 5.8 the computation for this semantics is illustrated. During the tests with the well-founded semantics we encountered an additional problem beside the high computation times for this encoding: The solver `claspD` was not able to solve 179 of the instances ($\sim 13\%$ of all instances), because of a *segmentation fault* during the execution. As these failures occurred in all different instance classes, we have counted them in the same way as a timeout, because it means that the computation was not possible. It can be seen in the statistics that the well-founded semantics seems to be the hardest semantics to solve. Especially this figure needs to be studied with care, because of the very high amount of instances with 20, 25, and 30 statements timed out. In fact only two instances for the 25 and 30 statements instances could be solved and for the 20 statement instances only 15 computations where successful in the given time ($\sim 6\%$). Therefore the decreasing mean computation time is mainly due to timeouts and errors. In addition for this encoding it can be observed that the solving time exceeds the grounder time. This is also reflected by the number of timeouts. From the 999 instances ($\sim 74\%$ of all instances) which had no computation result, occurred 193 timeouts ($\sim 14\%$ of all instances) during the grounding process and 627 timeouts ($\sim 44\%$ of all instances) during the solving process.

As we encountered a high percentage of failed computations for this semantics, we want to investigate further, how the un-computability in the given time relates to the other parameters we

have chosen for the generation of the test instances. During the review of the different parameters it appeared that the graphs grouped by the probability for constants was very similar to the graphs grouped by the probability of symmetric link relations. So it seems that the combination of the number of statements together with one of these probabilities behave in the given time window very similar. Although this needs to be seen with caution, because we do not know how long the computation of the failed instances would have taken. In addition it seems that there is no big difference between the type of connectives used in the instances. Based on these observations we present in Figure 5.9 how much successful computations were done, based on the probability for the substitution of variables by constants. This diagram shows again on the x-axis the number of statements, but the y-axis shows how much computations were successful. The three lines correspond to the instances where the probability for the occurrence of variables is 0 (blue line with boxed data-points), 0.5 (magenta line with a plus for th data-point symbols), and 1 (orange line with a cross for the data-points). Due to the additional split of the instances into the three groups with fixed probabilities for the constant occurrences, the maximum number of successful computations is 90. The first observation is that the computations for *ADFs* where every acceptance conditions only consists of constants performs for the small instances with 10 and 15 statements best. Those instances where the chance for variables equals the chance for constants seems to behave more linear then those with the more extreme chances for the constant occurrence.

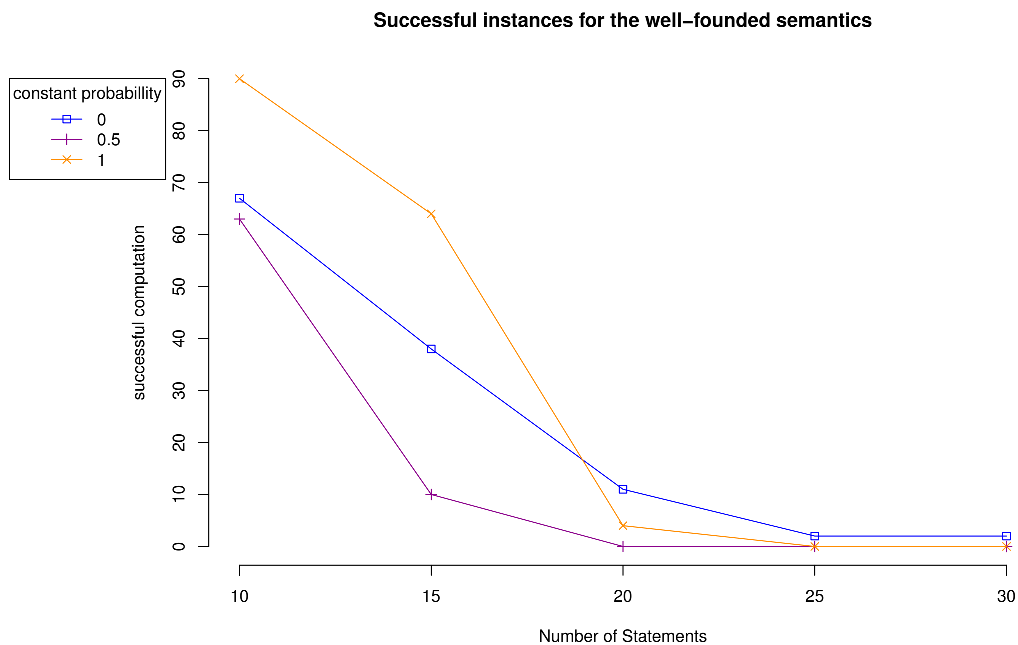


Figure 5.9: Successful computation instances for the well-founded semantics

Concluding remarks. One problem of the current encodings can be seen very well with this statistics: The formulae need to be broken down into subformulae, even if it only consists of constants and further the constant value is evaluated for every interpretation. So especially for instances which only consist of constant acceptance conditions way too much computations are done. In addition the above results also reflect the computational complexity of the problems very well. It can be observed that the different problems do not behave linear and in addition it can be observed that the underlying problem of the link-type resolution seems to be a very expensive one compared to the problem of finding the stable models. Note that this is also reflected in the fact that we need the saturation technique for the link-type resolution as well as for the well-founded semantics. Although the $\mathbf{coNP}$-problems (e.g. satisfiability for formulae in the well-founded semantics) do not rely on a higher stage of the polynomial hierarchy, we need to use disjunctive programs to solve them with ASP (Remember that the problem of solving disjunctive programs is in $\Sigma_2\mathbf{P}$).

CHAPTER 6

Related Work

In this chapter we will give an overview on related work in the field of abstract argumentation. The majority of this part will present other frameworks and approaches which are related to abstract argumentation and in some cases to *ADFs*. Then some other software systems for the automated computation of semantics for frameworks will be the point of interest.

6.1 Related Concepts

For the sake of completeness, at first we want to mention again that many of the currently used and proposed frameworks are based on the sophisticated and well-understood Dung's Argumentation Framework [Dung, 1995]. Several semantics and concepts emerged to overcome the drawbacks of the simple and powerful basic AF. In the following we will present at first similar ideas and concepts in comparison to *ADFs*. They are motivated through the idea to increase the expressiveness of Dung's AF. Subsequently we will present another semantic concept which mainly concentrates on an evaluation of the attack relations. Then we will picture approaches which are focused on the properties of the attack relations. Formalisms with multiple argumentation frameworks are the point of interest in the following paragraph. Afterwards we will step to a concept which has its emphasis on argumentation on the meta level of argumentation. In a final step we will sketch a concept which is only indirectly related to Dung's AF, but can be simulated with *ADFs*.

Concepts to increase the expressiveness of Dung's *AF*

With the aim to reinstate the semantics of Dung's AF, Martin Caminada proposed a concept of labeling for the arguments [Caminada, 2006]. These labels work in a similar way as arguments are in respectively out in the semantics of Dung's AF, but they are three-valued and allow the representation of accepted, rejected and undecided arguments. In his work he shows that properties for the labeling correspond to the extensions defined by Dung. One additional outcome

of these labelings is a more sophisticated formulation of acceptance, than just credulous and skeptical acceptance. For more details on this refinement see [Dvořák, 2011].

Following a similar motivation as *ADFs*, Coste-Marquis, Devred, and Marquis introduced *Constraint Argumentation Frameworks - CAFs* [Coste-Marquis et al., 2006]. In contrast to *ADFs*, a *CAF* consists of arguments, attack-relations and one propositional formula. The addition of a formula (i.e. the constraint) to Dung's AF is a restriction to the set of extensions. So every set of accepted arguments also needs to be an interpretation which satisfies the constraint formula. Indeed the constraints refine the expressiveness of the argumentation framework, but it is less expressive as *ADFs*, as they can specify the relations between the used statements in a more distinct way.

Another, apparently similar concept to *ADFs* are the *abstract bipolar argumentation frameworks* (for an overview see the survey article [Amgoud et al., 2008]). Here the lack of native support relations is the main motivation behind this approach. These argumentation frameworks add to Dung's AF an additional relation to represent a direct support between two arguments. The evaluation of supports and attacks is generally done by a notion of attack and support quality. Intuitively every attack on an argument increases the attack quality against this argument and support qualities are determined in the same way. If an argument has both, an attack and a support quality, then the higher one is taken into account for the semantic evaluation of extensions. Due to the possibility of cycles in the dependency of arguments and the calculation of their quality, cycles are not permitted. Note that for the cycle detection supports and attacks are counted as a relation. The main difference to *ADFs* is the reduced control of the link between different attack and support relations. In *ADFs* it can be specified very precisely when the attack is taken into account or not.

A more abstract approach are the highly generalized *Hyper Frameworks* [Weydert, 2011]. Here the author tried to overcome the general restriction that argumentation frameworks are finite. His basic motivation is the idea that we also have to deal with the fact that there exist infinitely many arguments for and against some statements. To get a formalism which can deal with an infinite domain, the relation definition between arguments is defined in first order logic. It is already shown that Dung's AF can be simulated with a *Hyper Framework*, but due to its highly abstract definition many definitions and properties are still open and are up to the user to be defined. Additionally there is the question about the role of support relations between arguments.

Approaches to conceptualize relational properties

Till now the presented argumentation frameworks had arguments and relations between the arguments. In addition it is a natural piece of reasoning that these relations may be the target of attacks. This idea is the motivation of *Extended Argumentation Frameworks - EAF* [Modgil, 2009]. Again this concept is based on Dung's AF and allows that attack relations may not only attack arguments, but also other attack relations between two arguments. The improved version of *EAFs* are the *Argumentation Frameworks with Recursive Attacks - AFRAs* [Baroni et al., 2011b], where recursive attacks on relations are permitted (e.g. attacks on attacks on attacks and similar things are possible). In addition in that paper of *AFRA* there is also a method presented to represent such a framework as a Dung's AF. For *ADFs* it shall be possible due to

the expressiveness of the acceptance conditions that they can simulate the attacks on attacks per *EAFs*. For example, if the attack from a statement a on b is attacked by c, it could be encoded in the acceptance condition AC_b as $\neg a \vee c$. Note that if this really works well with the different definitions of the semantics needs to be investigated further. In addition we suspect that it is also possible to simulate *AFRAs* with *ADFs*.

Villata, Boella, and van der Torre introduced another approach to work with attacks. They conceptualize successful and unsuccessful attacks for Dung's AF. The basic idea behind these attack semantics [Villata et al., 2011] is to only accept an argument if there is no successful attack on this argument. Intuitively there are properties which are defined to hold if and only if the property holds for every possible subset of arguments. With this general valid property a distinction of attacker dependencies, successful, unsuccessful, and defended attacks is possible. The most semantics (except the admissible extension) of Dung's AF are revamped for this approach such that the acceptance is no longer based on the acceptance of other arguments, but on the properties of the attacks.

Multiple Argumentation Framework concepts

In contrast to the previously presented concepts, we will now emphasize on approaches which work on multiple argumentation frameworks and deal with the exchange of knowledge which was inferred by these distinct frameworks. The foundation of the work of *Fibring Argumentation Frames* [Gabbay, 2009] is the conception of networks. A network may be any formalism for reasoning (e.g. argumentation frameworks, neural networks, Kripke models). The underlying idea is that each network is designed for different tasks and so it is not hard to imagine that one network is embedded in another one. A natural question is now how to deal with these nested networks. Is the inner or the outer network computed first? How are elements treated which occur in two networks? How are cycles resolved? To overcome these questions *Fibring Argumentation Frameworks* are a generalization of Dung's AF to represent every other network in an abstract way. To deal with the problems which occur from the nested networks, the *Frames* are in addition closed under substitution.

A similar idea is behind *Context Based Argumentation* [Brewka and Eiter, 2009]. They stick to Dung's AF and conceptualized the idea to have different argumentation frameworks which work parallel or in some sort of hierarchy to solve problems. The approach borrows mechanics from *Multicontext Systems* [Brewka and Eiter, 2007, Giunchiglia and Serafini, 1994] and presents a formal method to model different argumentation frameworks with knowledge exchange. The concept adds to each framework a so-called *mediator* which is responsible for the communication and the relations to other frameworks. One framework together with its mediator is called a *context*. The *Context Based Argumentation* offers bridge-rules as the method to relate the different contexts with each other. In addition it also provides inconsistency handling which is important for multiple contexts. Certainly it is possible to update the knowledge base of other contexts, based on the results of one context. So the *Context Based Argumentation* offers ways to add additional knowledge to the knowledge bases of other contexts. To allow heterogeneous constructs of contexts with different argumentation frameworks the highly abstract generalization of *Managed Multi Context Systems - mMCS* was introduced [Brewka et al., 2011b]. This

concept offers in addition mechanisms to modify the knowledge bases in a way like relational databases can manipulate informations.

Meta-Argumentation

All of the presented approaches used properties based on the arguments and their relations for their inferences. Now we want to discuss a concept which allows to express statements on the relations between arguments and their properties under a semantics. The *Metalevel Argumentation* [Modgil and Bench-Capon, 2011] intuitively allows us to utilize claims like “argument a defeats y” or “a is rejected” for a given or all semantics. In addition between these claims are implicit attack relations. In the previous two claims “a is rejected” would attack the other claim (“argument a defeats y”). A *Structured Argumentation Framework - SAF* is a Dung’s AF, together with a language definition for the additional claims, a mapping function to map arguments to well founded formulae in the language, and constraint rules defined in the language which defines attack relations. The proposed *Metalevel Argumentation Framework - MAF* is such a *SAF* with a specified set of allowed predicates and a specific set of rules to determine how the predicates have to be used in the language. To show that this concept is a generalization of Dung’s AF, it is also shown that a *MAF* can represent every Dung’s AF.

We have also mentioned in Section 4.1 that the $ADF \rightarrow AF$ transformation is closely related to the *Metalevel Argumentation* approach, because the semantical structures are built to control the acceptance of arguments. So it is needed to understand how the semantics (in this case the stable extension) works and how the acceptance is influenced by the relations between other arguments.

Carneades

At last we want to present another framework, which is special, because it is not based on Dung’s AF. *Carneades* [Gordon et al., 2007] is introduced as an abstract argumentation framework to model arguments in the anglo-american law system. Since the first introduction it was under a steady improvement (see e.g. [Gordon and Walton, 2009]) and so the explicit definitions for the framework and the acceptance of arguments are evolving and changing in different publications. Although the basic idea is still the same: Due to the connection to the law system, the framework tries to find arguments which are pro or con some assertion. These arguments have different types, depending on the kind of the information of the argument. Based on this type the needed premises to accept such an argument is different and in addition the burden of proof can be shifted. Intuitively the amount (or the quality) of arguments needed to accept an argument is dependent on the type of the argument. To project this concept, every accepted argument needs some kind of a argument-based argumentation-chain which is the proof of the acceptability of the argument. To model this framework there are some elementary differences to Dung’s approach. Beside both are modeled as directed graphs, in Carneades the nodes do have needed informations for the acceptability. In addition it is basically forbidden to use cycles in the argumentation graphs of Carneades. The argumentation-chains mentioned above are subgraphs of the graph, which can be seen as some kind of proof-tree to fulfill the burden of proof. A further difference is the concept that Carneades may be seen as a dialog and therefore it has stages and states.

Although it is a completely alternative approach, Carneades are quite closely related to *ADFs*, because it is possible to translate Carneades at one stage into an *ADF* [Brewka and Gordon, 2010]. So it was shown that Carneades can be represented by *ADFs*.

6.2 Related Software Systems

In this section we will give a quick overview on already existing systems which compute extensions and semantics for different related concepts. At first we will present other systems developed at the Vienna University of Technology and then we will present a software system for the University of Bologna and the Uppsala University. Afterwards a software system of the Oxford University as well as one from the University of Groningen will be discussed.

The `DLV`-based solver based system `ASPARTIX`[1] [Egly et al., 2008, 2010] uses the answer set solving paradigm to solve different argumentation frameworks. It is not only capable of computing different extensions of Dung's AF, as it can also work with preference based, value-based and bipolar Argumentation Frameworks. In addition an encoding to translate an *AFRA* to a Dung's AF is part of the system. As mentioned before, this system was also used as a benchmark comparison of the $ADF \rightarrow AF$ transformation.

`dynPARTIX`[2] [Dvořák et al., 2011] is another system which solves argumentation frameworks. This one can calculate the admissible, stable, complete and preferred extensions of Dung's AF. In addition it may answer the question whether an argument is skeptically (resp. credulously) accepted. The difference between `dynPARTIX` and `ASPARTIX` is the used computation model. `dynPARTIX` utilizes the `SHARP` framework, which is a library for heuristic methods and tree decompositions. The implementation is based on theoretical results for fixed parameter tractable algorithms for argumentation [Dvořák et al., 2010]. Note that this system uses the same syntax as `ASPARTIX` does.

The third system from the Vienna University of Technology is `CEGARTIX`[3] [Dvořák et al., 2012]. This one uses `clasp` or `MiniSat` as an $\mathbf{NP}$-oracle in an iterative fashion and is able to compute the skeptical acceptance for the preferred semantics of Dung's AF as well as both skeptical and credulous acceptance for stage and semi-stage semantics of Dung's AF for a specified argument.

A metalogic implementation for argumentation [Lundström et al., 2011a] is the next system we want to introduce. Their system is based on a metalogic approach, which puts the argumentation into a two player dialogue game environment to solve the acceptance problem of arguments [Lundström, 2009]. There one player tries to accept one argument and the other tries to refute it with other arguments. As the system also gives the possibility to browse the game-tree after the evaluation, this program offers a way to review why a specific argument is accepted or not. Due to the similarity between the dialogue and legal argumentation, this system is also used in this context [Lundström et al., 2011b].

[1] available at http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/
[2] available at http://www.dbai.tuwien.ac.at/research/project/argumentation/dynpartix/
[3] available at http://www.dbai.tuwien.ac.at/research/project/argumentation/cegartix/

Based on JAVA the solver with graphical user interface Dungine[4] [South et al., 2008] can compute the grounded skeptical and preferred credulous acceptance of arguments. For the reasoning-process a game-based structure is again used. A nice feature of the graphical user interface is the option to work with the argumentation framework in an illustrated representation and not only with text.

At last we want to present a credulous acceptance and rejection solver[5] [Verheij, 2007], written in Delphi 7. The tool computes the credulous acceptance for Dung's *AF* for the minimal admissible set, the grounded extension, the stable and semi-stable extension as well as for the preferred extension. Beside the acceptance it also computes the rejection of several arguments.

[4]available at http://www.argkit.org/

[5]available at http://www.ai.rug.nl/ verheij/comparg/

CHAPTER 7

Conclusion & Future Work

Conclusion

In this work we have reviewed the concept of *abstract argumentation* and the approach of *ADFs*. To refine this generalization of Dung's *AF*, we have shown that the frameworks can be conceptualized on the foundation of propositional logic and in a short excursion that it is even possible to represent them with hyper-graphs.

Based on our *pForm-ADF* representation we then have investigated some of the properties of the frameworks and introduced the subclass of *monotone pForm-ADFs*. For those frameworks we have proven that they are *BADFs* and that they can express every *BADF* without loss of generality. Based on the idea of *monotone pForm-ADFs* we then proposed an algorithm to transform an arbitrary *ADF* to a *BADF*. To overcome the shortcoming that the *stable model* semantics and all semantics based on it are only defined for *BADFs*, we have presented and proven the correctness of a *generalized stable model* semantics on basis of the outcome from our novel *ADF* to *BADF* transformation.

Afterwards we have investigated some inter-semantics relations which hold for Dung's *AFs*, and we showed that they do not carry over to *ADFs*. This holds for the stable model semantics as well as for the preferred semantics, which calls for further investigation.

To reduce the open gaps for complexity results on *ADFs*, we have presented results for the link-type resolution of *ADFs*, as they are a prerequisite for the complexity results of the *stable model* semantics. Due to the introduction of *monotone pForm-ADFs* we have shown that the complexity for the $\mathbf{Cred}^{m}_{stable}$-decision problem, if restricted to the subclass of *monotone pForm-ADFs*, is $\mathbf{NP}$-complete and is therefore in the same complexity class as the $Cred_{stable}$ problem on *BADFs* if the linktypes are known beforehand.

Furthermore we have proposed a software system which provides ASP encodings to compute the different semantics for *pForm-ADFs*. This system is using mechanisms which correspond to the known complexities of the different problems and it also provides some sense of possible memberships for currently unknown complexity results. The preliminary tests which are provided as well in this work give first impressions on the mean computation time for differ-

ent semantics. So we could observe that the *model* semantics may solve *ADFs* with up to 100 statements in five minutes, while the more complex semantics can only compute frameworks with maximal 30 statements in the same time. This shows that the high expressiveness of *ADFs* compared to *AFs* comes with the price of more involved computations

At last we have given an overview on other approaches for argumentation frameworks and have discussed the differences and similarities with respect to *ADFs*.

Future Work

Our preliminary test results have showed that the encodings should be optimized for some special cases of formulae (e.g. the acceptance conditions are truth-value constants) to reduce the unnecessary computations of the same results over and over. Furthermore we have seen that the grounding seems to be a bottle-neck for many semantics. So it is desirable to do additional optimization of the encodings. In addition more investigations of the involved relations of statements and acceptance conditions under different semantics are needed to create meaningful, computationally hard to solve, and general benchmark instances for *ADFs* to be able to do exhaustive and comparable tests. Alternatively it may also lead to a better performance if we try to utilize existing software systems like `CEGARTIX` [Dvořák et al., 2012] or `DYNPARTIX` [Dvořák et al., 2011].

Another topic to further investigate is the inter-semantics properties with respect to Dung's *AF*. Based on the presented counter-examples it is also important to revisit the *stable model* semantics to fulfill the properties given by Dung. The whole approach of the *stable model* is very similar to the Gelfond-Lifschitz reduct. So it would be interesting to consider the correspondence between those two concepts to gain a deeper understanding on the relation between *ADFs* and logic programming. To strengthen the connection between the different related argumentation approaches an investigation of a possible simulation of *CAFs* with *ADFs* may be considered. In addition investigations of the relation between *ADFs* and *EAFs* respectively *AFRAs* may be considered too.

APPENDIX A

Listing of the ADF $\rightarrow$ AF Encodings

A.1 Model for ADFs

```
1  % encoding for a model for an ADF where the formulas are in CNF
2
3  % guessing which statements are in the model
4
5  in(X) ← not out(X), statement(X).
6  out(X) ← not in(X), statement(X).
7
8
9  vdisj(X,Y) ← cl(X,Y), pos(Y,Z), in(Z).
10 vdisj(X,Y) ← cl(X,Y), neg(Y,Z), out(Z).
11
12 % every conjunction which evaluates to false under the assignemnt
13 fconj(X) ← cl(X,Y), not vdisj(X,Y).
14
15 % remove all assignments which contain statements whose formulae are not valid
16 ← in(X), fconj(X).
17 % remove all assignments where statements are not selected although their formulas are valid
18 ← out(X), not fconj(X).
```

A.2 ADF $\rightarrow$ AF transformation

```
1  #begin_lua
2  local i=-1
3  local j=-1
4  local k=-1
5  function neg(s)
6    if (string.sub(Val.name(s),1,1)) == "_" then
7      return Val.new(Val.ID, string.sub(Val.name(s),2))
8    else
9      return Val.new(Val.ID, "_"..Val.name(s))
10   end
11 end
12
13 function dhelper(s)
14   i=i+1
15   return Val.new(Val.ID, "h_v_" .. i)
16 end
17
18 function chelper(s)
19   j=j+1
20   return Val.new(Val.ID, "h_c_"..j)
```

```
21  end
22
23  function disjunct(s)
24    k=k+1
25    return Val.new(Val.ID, "h_V"..k)
26  end
27  #end_lua.
28
29  % generate for each statement of the ADF the argument and its negation for the AF
30  arg(X) ← statement(X).
31  arg(@neg(X)) ← statement(X).
32
33  % determine the number of conjunctions and disjunctions in the given CNFs:
34  numcon(X,Y) ← statement(X), Y:=#count{cl(X,_)}.
35  numdis(Y,Z) ← statement(X), cl(X,Y), Z:=#count{pos(Y,_),neg(Y,_)}.
36
37  % identify statements which have verum and falsum as their acceptance condition
38  verum(X) ← statement(X), numcon(X,Y), Y==0.
39  falsum(X) ← statement(X), cl(X,Y), numdis(Y,Z), Z==0.
40
41  % attack relations for the arguments and its negations
42  att(X,@neg(X)) ← verum(X).
43  att(@neg(X),X) ← falsum(X).
44  att(X,@neg(X)) ← statement(X), not verum(X), not falsum(X).
45  att(@neg(X),X) ← statement(X), not verum(X), not falsum(X).
46
47  % generate the attack relations if the acceptance condition is one literal
48  neglitatt(Z,X) ← statement(X), cl(X,Y), numcon(X,A), numdis(Y,B), neg(Y,Z), A==1, B==1.
49  poslitatt(Z,X) ← statement(X), cl(X,Y), numcon(X,A), numdis(Y,B), pos(Y,Z), A==1, B==1.
50  att(Z,X) ← neglitatt(Z,X).
51  att(@neg(Z),@neg(X)) ← neglitatt(Z,X).
52  att(@neg(Z),X) ← poslitatt(Z,X).
53  att(Z,@neg(X)) ← poslitatt(Z,X).
54
55  % generate the attack relations for a generic attack
56  att(H,T) ← genatt(I,T,H).
57  att(I,H) ← genatt(I,T,H).
58  att(I,@neg(T)) ← genatt(I,T,H).
59
60  % generate literals which are pos or neg.
61  lit(X,Y) ← pos(X,Y).
62  lit(X,@neg(Y)) ← neg(X,Y).
63
64  % generate the attack relations, ...
65
66  % if the acceptance condition only consists of a serie of disjunctions
67  datt(@dhelper(T),T,X) ← statement(T), cl(T,X), numcon(T,A), numdis(X,B), A==1, B>1.
68  genatt(I,T,H) ← datt(H,T,X), lit(X,I).
69
70  % if the acceptance condition has at least one conjunction
71  catt(@chelper(T),T) ← statement(T), not falsum(T), numcon(T,A), A>1.
72
73  % if the subformula from a catt conjunction is only one literal
74  genatt(@neg(I),@neg(T),H) ← catt(H,T), cl(T,X), numdis(X,B), lit(X,I), B==1.
75
76  % if the subformula from a catt conjunction consists of a serie of disjunctions
77  dincatt(@disjunct(X),X) ← catt(H,T), cl(T,X), numdis(X,B), B>1.
78  datt(@dhelper(T),T,X) ← dincatt(T,X).
79  genatt(@neg(I),@neg(T),H) ← dincatt(I,X), cl(T,X), catt(H,T).
80
81  % create the mutual attacks for the "disjunct"-nodes (dincatt)
82  att(X,@neg(X)) ← dincatt(X,_).
83  att(@neg(X),X) ← dincatt(X,_).
84
85  % create the "missing" arguments
86  arg(X) ← att(X,Y).
87  arg(Y) ← att(X,Y).
```

APPENDIX B

Listing of the ADF System Encodings

B.1 Linktypes

```
1  % splitting a formula into its subformulas
2  subformula(X,F) ← ac(X,F),statement(X).
3  subformula(X,F) ← subformula(X,and(F,_)).
4  subformula(X,F) ← subformula(X,and(_,F)).
5  subformula(X,F) ← subformula(X,or(_,F)).
6  subformula(X,F) ← subformula(X,or(F,_)).
7  subformula(X,F) ← subformula(X,neg(F)).
8  subformula(X,F) ← subformula(X,xor(F,_)).
9  subformula(X,F) ← subformula(X,xor(_,F)).
10 subformula(X,F) ← subformula(X,imp(F,_)).
11 subformula(X,F) ← subformula(X,imp(_,F)).
12 subformula(X,F) ← subformula(X,iff(F,_)).
13 subformula(X,F) ← subformula(X,iff(_,F)).
14 subformula(F) ← subformula(_,F).
15
16 % decide whether a subformula is an atom or not
17 noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18 noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19 noatom(F) ← subformula(F;F1), F:=neg(F1).
20 noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21 noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22 noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24 atom(X) ← subformula(X), not noatom(X).
25 atom(X) ← subformula(X), X:=c(v).
26 atom(X) ← subformula(X), X:=c(f).
27
28 step(I,J,C) ← in(_,I,J,C).
29 step(I,J,C) ← out(_,I,J,C).
30
31 % check whether an interpretation is a model or not
32 % ismodel (X,I,J,C)
33 % I,J ... step in the below defined ordering
34 % C ... counter for the c−th needed evaluation
35 ismodel(X,I,J,C) ← atom(X), in(X,I,J,C).
36 ismodel(X,I,J,C) ← atom(X), X:=c(v), step(I,J,C).
37 ismodel(F,I,J,C) ← subformula(F;F1), F:=neg(F1), nomodel(F1,I,J,C).
38 ismodel(F,I,J,C) ← subformula(F), F:=and(F1,F2), ismodel(F1;F2,I,J,C).
39 ismodel(F,I,J,C) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F1,I,J,C).
40 ismodel(F,I,J,C) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F2,I,J,C).
41 ismodel(F,I,J,C) ← subformula(F), F:=xor(F1,F2), ismodel(F1,I,J,C), nomodel(F2,I,J,C).
42 ismodel(F,I,J,C) ← subformula(F), F:=xor(F1,F2), ismodel(F2,I,J,C), nomodel(F1,I,J,C).
43 ismodel(F,I,J,C) ← subformula(F;F1;F2), F:=imp(F1,F2), nomodel(F1,I,J,C).
```

```
44  ismodel(F,I,J,C) ← subformula(F), F:=imp(F1,F2), ismodel(F1;F2,I,J,C).
45  ismodel(F,I,J,C) ← subformula(F), F:=iff(F1,F2), ismodel(F1;F2,I,J,C).
46  ismodel(F,I,J,C) ← subformula(F), F:=iff(F1,F2), nomodel(F1;F2,I,J,C).
47
48  nomodel(X,I,J,C) ← atom(X), out(X,I,J,C).
49  nomodel(X,I,J,C) ← atom(X), X:=c(f), step(I,J,C).
50  nomodel(F,I,J,C) ← subformula(F;F1), F:=neg(F1), ismodel(F1,I,J,C).
51  nomodel(F,I,J,C) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F1,I,J,C).
52  nomodel(F,I,J,C) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F2,I,J,C).
53  nomodel(F,I,J,C) ← subformula(F), F:=or(F1,F2), nomodel(F1,I,J,C), nomodel(F2,I,J,C).
54  nomodel(F,I,J,C) ← subformula(F), F:=xor(F1,F2), ismodel(F1,I,J,C), ismodel(F2,I,J,C).
55  nomodel(F,I,J,C) ← subformula(F), F:=xor(F1,F2), nomodel(F1,I,J,C), nomodel(F2,I,J,C).
56  nomodel(F,I,J,C) ← subformula(F), F:=imp(F1,F2), ismodel(F1,I,J,C), nomodel(F2,I,J,C).
57  nomodel(F,I,J,C) ← subformula(F), F:=iff(F1,F2), nomodel(F1,I,J,C), ismodel(F2,I,J,C).
58  nomodel(F,I,J,C) ← subformula(F), F:=iff(F1,F2), nomodel(F2,I,J,C), ismodel(F1,I,J,C).
59
60  % generate links between statements and atoms
61  link(X,S) ← statement(S), atom(X), subformula(S,X), statement(X).
62  %att(X,S) ∨ sup(X,S) ∨ att_supp(X,S) ∨ dependent(X,S) ← link(X,S).
63
64  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
65  %% linktype check
66  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
67
68  % guess which link type it is
69  att(X,S) ∨ supp(X,S) ∨ att_supp(X,S) ∨ dep(X,S) ← link(X,S).
70
71  % check whether the guess was right or not:
72
73  in(X,I,J,1) ∨ out(X,I,J,1) ← link(I,J), statement(X), atom(X), subformula(J,X).
74  ← in(X,X,J,1).
75  in(X,I,J,2) ← in(X,I,J,1).
76  out(X,I,J,2) ← out(X,I,J,1), X != I.
77  in(X,X,J,2) ← out(X,X,J,1).
78
79  noattclash(I,J) ← link(I,J), ac(J,F), ismodel(F,I,J,1).
80  noattclash(I,J) ← link(I,J), ac(J,F), nomodel(F,I,J,1), nomodel(F,I,J,2).
81  nosuppclash(I,J) ← link(I,J), ac(J,F), nomodel(F,I,J,1).
82  nosuppclash(I,J) ← link(I,J), ac(J,F), ismodel(F,I,J,1), ismodel(F,I,J,2).
83  noatt_suppclash(I,J) ← link(I,J), ac(J,F), nomodel(F,I,J,1), nomodel(F,I,J,2).
84  noatt_suppclash(I,J) ← link(I,J), ac(J,F), ismodel(F,I,J,1), ismodel(F,I,J,2).
85
86
87  ok(I,J) ← noattclash(I,J), att(I,J).
88  ok(I,J) ← nosuppclash(I,J), supp(I,J).
89  ok(I,J) ← noatt_suppclash(I,J), att_supp(I,J).
90
91  ← not ok(I,J), link(I,J), not dep(I,J).
92
93  in(X,I,J,1) ← ok(I,J), out(X,I,J,1), X!=I.
94  out(X,I,J,1) ← ok(I,J), in(X,I,J,1), X!=I.
95
96
97  % extended check for dep:
98
99  % no saturation as the problem whether a link is dep or not is in NP and not in coNP
100
101 in(X,I,J,3) ∨ out(X,I,J,3) ← link(I,J), dep(I,J), statement(X), atom(X), subformula(J,X).
102 ← in(X,X,J,3).
103 in(X,I,J,4) ← in(X,I,J,3).
104 out(X,I,J,4) ← out(X,I,J,3), X != I.
105 in(X,X,J,4) ← out(X,X,J,3).
106
107 isdep(I,J) ← link(I,J), ac(J,F), ismodel(F,I,J,1), nomodel(F,I,J,2), nomodel(F,I,J,3),
        ismodel(F,I,J,4).
108 ← dep(I,J), not isdep(I,J).
109
110 #maximize[att_supp(X,Y)].
111 #hide.
112 #show att/2.
113 #show supp/2.
114 #show att_supp/2.
115 #show dep/2.
```

B.2 Conflict-free set

```
1  % splitting a formula into its subformulas
2  subformula(X,F) ← ac(X,F),statement(X).
3  subformula(X,F) ← subformula(X,and(F,_)).
4  subformula(X,F) ← subformula(X,and(_,F)).
5  subformula(X,F) ← subformula(X,or(_,F)).
6  subformula(X,F) ← subformula(X,or(F,_)).
7  subformula(X,F) ← subformula(X,neg(F)).
8  subformula(X,F) ← subformula(X,xor(F,_)).
9  subformula(X,F) ← subformula(X,xor(_,F)).
10 subformula(X,F) ← subformula(X,imp(F,_)).
11 subformula(X,F) ← subformula(X,imp(_,F)).
12 subformula(X,F) ← subformula(X,iff(F,_)).
13 subformula(X,F) ← subformula(X,iff(_,F)).
14 subformula(F) ← subformula(_,F).
15
16 % decide whether a subformula is an atom or not
17 noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18 noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19 noatom(F) ← subformula(F;F1), F:=neg(F1).
20 noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21 noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22 noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24 atom(X) ← subformula(X), not noatom(X).
25 atom(X) ← subformula(X), X:=c(v).
26 atom(X) ← subformula(X), X:=c(f).
27
28 % check whether an interpretation is a model or not
29 ismodel(X) ← atom(X), in(X).
30 ismodel(X) ← atom(X), X:=c(v).
31 ismodel(F) ← subformula(F;F1), F:=neg(F1), nomodel(F1).
32 ismodel(F) ← subformula(F), F:=and(F1,F2), ismodel(F1;F2).
33 ismodel(F) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F1).
34 ismodel(F) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F2).
35 ismodel(F) ← subformula(F), F:=xor(F1,F2), ismodel(F1), nomodel(F2).
36 ismodel(F) ← subformula(F), F:=xor(F1,F2), ismodel(F2), nomodel(F1).
37 ismodel(F) ← subformula(F;F1;F2), F:=imp(F1,F2), nomodel(F1).
38 ismodel(F) ← subformula(F), F:=imp(F1,F2), ismodel(F1;F2).
39 ismodel(F) ← subformula(F), F:=iff(F1,F2), ismodel(F1;F2).
40 ismodel(F) ← subformula(F), F:=iff(F1,F2), nomodel(F1;F2).
41
42 nomodel(X) ← atom(X), out(X).
43 nomodel(X) ← atom(X), X:=c(f).
44 nomodel(F) ← subformula(F;F1), F:=neg(F1), ismodel(F1).
45 nomodel(F) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F1).
46 nomodel(F) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F2).
47 nomodel(F) ← subformula(F), F:=or(F1,F2), nomodel(F1), nomodel(F2).
48 nomodel(F) ← subformula(F), F:=xor(F1,F2), ismodel(F1), ismodel(F2).
49 nomodel(F) ← subformula(F), F:=xor(F1,F2), nomodel(F1), nomodel(F2).
50 nomodel(F) ← subformula(F), F:=imp(F1,F2), ismodel(F1), nomodel(F2).
51 nomodel(F) ← subformula(F), F:=iff(F1,F2), nomodel(F1), ismodel(F2).
52 nomodel(F) ← subformula(F), F:=iff(F1,F2), nomodel(F2), ismodel(F1).
53
54 %guessing, whether a statement is in or not
55
56 in(X) ← not out(X), statement(X).
57 out(X) ← not in(X), statement(X).
58
59 % encoding for the conflict free sets for an ADF with an arbitrary formula as AC
60
61 ← in(X), ac(X,F), nomodel(F).
62 #hide.
63 #show in/1.
```

B.3 Model

```
1  % splitting a formula into its subformulas
2  subformula(X,F) ← ac(X,F), statement(X).
3  subformula(X,F) ← subformula(X,and(F,_)).
4  subformula(X,F) ← subformula(X,and(_,F)).
5  subformula(X,F) ← subformula(X,or(_,F)).
6  subformula(X,F) ← subformula(X,or(F,_)).
7  subformula(X,F) ← subformula(X,neg(F)).
8  subformula(X,F) ← subformula(X,xor(F,_)).
9  subformula(X,F) ← subformula(X,xor(_,F)).
10 subformula(X,F) ← subformula(X,imp(F,_)).
11 subformula(X,F) ← subformula(X,imp(_,F)).
12 subformula(X,F) ← subformula(X,iff(F,_)).
13 subformula(X,F) ← subformula(X,iff(_,F)).
14 subformula(F) ← subformula(_,F).
15
16 % decide whether a subformula is an atom or not
17 noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18 noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19 noatom(F) ← subformula(F;F1), F:=neg(F1).
20 noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21 noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22 noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24 atom(X) ← subformula(X), not noatom(X).
25 atom(X) ← subformula(X), X:=c(v).
26 atom(X) ← subformula(X), X:=c(f).
27
28 % check whether an interpretation is a model or not
29 ismodel(X) ← atom(X), in(X).
30 ismodel(X) ← atom(X), X:=c(v).
31 ismodel(F) ← subformula(F;F1), F:=neg(F1), nomodel(F1).
32 ismodel(F) ← subformula(F), F:=and(F1,F2), ismodel(F1;F2).
33 ismodel(F) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F1).
34 ismodel(F) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F2).
35 ismodel(F) ← subformula(F), F:=xor(F1,F2), ismodel(F1), nomodel(F2).
36 ismodel(F) ← subformula(F), F:=xor(F1,F2), ismodel(F2), nomodel(F1).
37 ismodel(F) ← subformula(F;F1;F2), F:=imp(F1,F2), nomodel(F1).
38 ismodel(F) ← subformula(F), F:=imp(F1,F2), ismodel(F1;F2).
39 ismodel(F) ← subformula(F), F:=iff(F1,F2), ismodel(F1;F2).
40 ismodel(F) ← subformula(F), F:=iff(F1,F2), nomodel(F1;F2).
41
42 nomodel(X) ← atom(X), out(X).
43 nomodel(X) ← atom(X), X:=c(f).
44 nomodel(F) ← subformula(F;F1), F:=neg(F1), ismodel(F1).
45 nomodel(F) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F1).
46 nomodel(F) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F2).
47 nomodel(F) ← subformula(F), F:=or(F1,F2), nomodel(F1), nomodel(F2).
48 nomodel(F) ← subformula(F), F:=xor(F1,F2), ismodel(F1), ismodel(F2).
49 nomodel(F) ← subformula(F), F:=xor(F1,F2), nomodel(F1), nomodel(F2).
50 nomodel(F) ← subformula(F), F:=imp(F1,F2), ismodel(F1), nomodel(F2).
51 nomodel(F) ← subformula(F), F:=iff(F1,F2), nomodel(F1), ismodel(F2).
52 nomodel(F) ← subformula(F), F:=iff(F1,F2), nomodel(F2), ismodel(F1).
53
54 %guessing, whether a statement is in or not
55
56 in(X) ← not out(X), statement(X).
57 out(X) ← not in(X), statement(X).
58
59 % encoding for a model for an ADF with an arbitrary formula as AC
60
61 ← in(X), ac(X,F), nomodel(F).
62 ← out(X), ac(X,F), ismodel(F).
63 #hide.
64 #show in/1.
```

B.4 Stable model

```
1  % splitting a formula into its subformulas
2  subformula(X,F) ← ac(X,F),statement(X).
3  subformula(X,F) ← subformula(X,and(F,_)).
4  subformula(X,F) ← subformula(X,and(_,F)).
5  subformula(X,F) ← subformula(X,or(_,F)).
6  subformula(X,F) ← subformula(X,or(F,_)).
7  subformula(X,F) ← subformula(X,neg(F)).
8  subformula(X,F) ← subformula(X,xor(F,_)).
9  subformula(X,F) ← subformula(X,xor(_,F)).
10 subformula(X,F) ← subformula(X,imp(F,_)).
11 subformula(X,F) ← subformula(X,imp(_,F)).
12 subformula(X,F) ← subformula(X,iff(F,_)).
13 subformula(X,F) ← subformula(X,iff(_,F)).
14 subformula(F) ← subformula(_,F).
15
16 % decide whether a subformula is an atom or not
17 noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18 noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19 noatom(F) ← subformula(F;F1), F:=neg(F1).
20 noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21 noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22 noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24 atom(X) ← subformula(X), not noatom(X).
25 atom(X) ← subformula(X), X:=c(v).
26 atom(X) ← subformula(X), X:=c(f).
27
28 % check whether an interpretation is a model or not
29 ismodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I=model.
30 ismodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, not att(X,S), not att_supp(X,S),
       in(S,model).
31 ismodel(X,S,I) ← subformula(S,X), atom(X), X:=c(v), modelcheck(I), I=model.
32 ismodel(X,S,I) ← subformula(S,X), atom(X), X:=c(v), modelcheck(I), I!=model, in(S,model).
33 ismodel(F,S,I) ← subformula(S,F;F1), F:=neg(F1), nomodel(F1,S,I).
34 ismodel(F,S,I) ← subformula(S,F), F:=and(F1,F2), ismodel(F1;F2,S,I).
35 ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=or(F1,F2), ismodel(F1,S,I).
36 ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=or(F1,F2), ismodel(F2,S,I).
37 ismodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F1,S,I), nomodel(F2,S,I).
38 ismodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F2,S,I), nomodel(F1,S,I).
39 ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=imp(F1,F2), nomodel(F1,S,I).
40 ismodel(F,S,I) ← subformula(S,F), F:=imp(F1,F2), ismodel(F1;F2,S,I).
41 ismodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), ismodel(F1;F2,S,I).
42 ismodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F1;F2,S,I).
43
44 nomodel(X,S,I) ← subformula(S,X), atom(X), out(X,I), I=model.
45 nomodel(X,S,I) ← subformula(S,X), atom(X), out(X,I), I!=model, in(S,model).
46 nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, att(X,S), in(s,model).
47 nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, att_supp(X,S), in(s,model).
48 nomodel(X,S,I) ← subformula(S,X), atom(X), X:=c(f), modelcheck(I), I=model.
49 nomodel(X,S,I) ← subformula(S,X), atom(X), X:=c(f), modelcheck(I), I!=model, in(S,model).
50 nomodel(F,S,I) ← subformula(S,F;F1), F:=neg(F1), ismodel(F1,S,I).
51 nomodel(F,S,I) ← subformula(S,F;F1;F2), F:=and(F1,F2), nomodel(F1,S,I).
52 nomodel(F,S,I) ← subformula(S,F;F1;F2), F:=and(F1,F2), nomodel(F2,S,I).
53 nomodel(F,S,I) ← subformula(S,F), F:=or(F1,F2), nomodel(F1,S,I), nomodel(F2,S,I).
54 nomodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F1,S,I), ismodel(F2,S,I).
55 nomodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), nomodel(F1,S,I), nomodel(F2,S,I).
56 nomodel(F,S,I) ← subformula(S,F), F:=imp(F1,F2), ismodel(F1,S,I), nomodel(F2,S,I).
57 nomodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F1,S,I), ismodel(F2,S,I).
58 nomodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F2,S,I), ismodel(F1,S,I).
59
60 % guess whether a statement is in or out (for the model semantics)
61 in(X,model) ← not out(X,model), statement(X).
62 out(X,model) ← not in(X,model), statement(X).
63 modelcheck(model).
64
65 % check if the guess was right
66 ← in(S,model), ac(S,F), nomodel(F,S,model).
67 ← out(S,model), ac(S,F), ismodel(F,S,model).
68
69 % removal of all not selected elements is implicit done by setting them to "out"
```

```
70  modelStatementCount(I) ← I:={in(S,model):statement(S)}.
71  iteration(I) ← modelStatementCount(I).
72  out(S,I) ← statement(S), modelStatementCount(I).
73  modelcheck(I) ← iteration(I).
74
75  iteration(J) ← iteration(I), J:=I−1, I>0.
76  in(S,J) ← iteration(I), iteration(J), J:=I−1, ismodel(F,S,I), ac(S,F).
77  out(S,J) ← iteration(I), iteration(J), J:=I−1, not in(S,J), out(S,I).
78
79  in(S) ← in(S,J), J:=0.
80
81  ← in(S,model), not in(S).
```

B.5 Admissible set

```
1   % splitting a formula into its subformulas
2   subformula(X,F) ← ac(X,F),statement(X).
3   subformula(X,F) ← subformula(X,and(F,_)).
4   subformula(X,F) ← subformula(X,and(_,F)).
5   subformula(X,F) ← subformula(X,or(_,F)).
6   subformula(X,F) ← subformula(X,or(F,_)).
7   subformula(X,F) ← subformula(X,neg(F)).
8   subformula(X,F) ← subformula(X,xor(F,_)).
9   subformula(X,F) ← subformula(X,xor(_,F)).
10  subformula(X,F) ← subformula(X,imp(F,_)).
11  subformula(X,F) ← subformula(X,imp(_,F)).
12  subformula(X,F) ← subformula(X,iff(F,_)).
13  subformula(X,F) ← subformula(X,iff(_,F)).
14  subformula(F) ← subformula(_,F).
15
16  % decide whether a subformula is an atom or not
17  noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18  noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19  noatom(F) ← subformula(F;F1), F:=neg(F1).
20  noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21  noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22  noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24  atom(X) ← subformula(X), not noatom(X).
25  atom(X) ← subformula(X), X:=c(v).
26  atom(X) ← subformula(X), X:=c(f).
27
28  % check whether an interpretation is a model or not
29  ismodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I=model.
30  ismodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, not att(X,S), not att_supp(X,S),
        in(S,model).
31  ismodel(X,S,I) ← subformula(S,X), atom(X), X:=c(v), modelcheck(I), I=model.
32  ismodel(X,S,I) ← subformula(S,X), atom(X), X:=c(v), modelcheck(I), I!=model, in(S,model).
33  ismodel(F,S,I) ← subformula(S,F;F1), F:=neg(F1), nomodel(F1,S,I).
34  ismodel(F,S,I) ← subformula(S,F), F:=and(F1,F2), ismodel(F1;F2,S,I).
35  ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=or(F1,F2), ismodel(F1,S,I).
36  ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=or(F1,F2), ismodel(F2,S,I).
37  ismodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F1,S,I), nomodel(F2,S,I).
38  ismodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F2,S,I), nomodel(F1,S,I).
39  ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=imp(F1,F2), nomodel(F1,S,I).
40  ismodel(F,S,I) ← subformula(S,F), F:=imp(F1,F2), ismodel(F1;F2,S,I).
41  ismodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), ismodel(F1;F2,S,I).
42  ismodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F1;F2,S,I).
43
44  nomodel(X,S,I) ← subformula(S,X), atom(X), out(X,I), I=model.
45  nomodel(X,S,I) ← subformula(S,X), atom(X), out(X,I), I!=model, in(S,model).
46  nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, att(X,S), in(s,model).
47  nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, att_supp(X,S), in(s,model).
48  nomodel(X,S,I) ← subformula(S,X), atom(X), X:=c(f), modelcheck(I), I=model.
49  nomodel(X,S,I) ← subformula(S,X), atom(X), X:=c(f), modelcheck(I), I!=model, in(S,model).
50  nomodel(F,S,I) ← subformula(S,F;F1), F:=neg(F1), ismodel(F1,S,I).
51  nomodel(F,S,I) ← subformula(S,F;F1;F2), F:=and(F1,F2), nomodel(F1,S,I).
52  nomodel(F,S,I) ← subformula(S,F;F1;F2), F:=and(F1,F2), nomodel(F2,S,I).
53  nomodel(F,S,I) ← subformula(S,F), F:=or(F1,F2), nomodel(F1,S,I), nomodel(F2,S,I).
54  nomodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F1,S,I), ismodel(F2,S,I).
55  nomodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), nomodel(F1,S,I), nomodel(F2,S,I).
```

```
56  nomodel(F,S,I) ← subformula(S,F), F:=imp(F1,F2), ismodel(F1,S,I), nomodel(F2,S,I).
57  nomodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F1,S,I), ismodel(F2,S,I).
58  nomodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F2,S,I), ismodel(F1,S,I).
59
60  % guess a set M and R
61  in(X,m) ← not out(X,m), statement(X).
62  out(X,m) ← not in(X,m), statement(X).
63
64  in(X,r) ← not out(X,r), statement(X), in(Y,m).
65  out(X,r) ← not in(X,r), statement(X).
66
67  ← in(X,r), in(Y,m), att(X,Y).
68  ← in(X,r), in(Y,m), att_supp(X,Y).
69
70  % M can only be a model if no element of R is also in M
71  in(X,model) ← in(X,m), out(X,r).
72  out(X,model) ← statement(X), not in(X,model).
73  modelcheck(model).
74
75  % every M which is not checked to be a model has to be removed (as it cant be a stable model)
76  ← in(X,m), not in(X,model).
77
78  % check if the guess was right
79  ← in(S,model), ac(S,F), nomodel(F,S,model).
80  ← out(S,model), ac(S,F), ismodel(F,S,model), not in(S,r).
81
82  % removal of all not selected elements is implicit done by setting them to "out"
83  modelStatementCount(I) ← I:={in(S,model):statement(S)}.
84  iteration(I) ← modelStatementCount(I).
85  out(S,I) ← statement(S), modelStatementCount(I).
86  modelcheck(I) ← iteration(I).
87
88  iteration(J) ← iteration(I), J:=I−1, I>0.
89  in(S,J) ← iteration(I), iteration(J), J:=I−1, ismodel(F,S,I), ac(S,F), not in(S,r).
90  out(S,J) ← iteration(I), iteration(J), J:=I−1, not in(S,J), out(S,I).
91
92  in(S) ← in(S,J), J:=0.
93
94  ← in(S,model), not in(S).
```

B.6 Preferred model

```
1   % splitting a formula into its subformulas
2   subformula(X,F) ← ac(X,F),statement(X).
3   subformula(X,F) ← subformula(X,and(F,_)).
4   subformula(X,F) ← subformula(X,and(_,F)).
5   subformula(X,F) ← subformula(X,or(_,F)).
6   subformula(X,F) ← subformula(X,or(F,_)).
7   subformula(X,F) ← subformula(X,neg(F)).
8   subformula(X,F) ← subformula(X,xor(F,_)).
9   subformula(X,F) ← subformula(X,xor(_,F)).
10  subformula(X,F) ← subformula(X,imp(F,_)).
11  subformula(X,F) ← subformula(X,imp(_,F)).
12  subformula(X,F) ← subformula(X,iff(F,_)).
13  subformula(X,F) ← subformula(X,iff(_,F)).
14  subformula(F) ← subformula(_,F).
15
16  % decide whether a subformula is an atom or not
17  noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18  noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19  noatom(F) ← subformula(F;F1), F:=neg(F1).
20  noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21  noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22  noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24  atom(X) ← subformula(X), not noatom(X).
25  atom(X) ← subformula(X), X:=c(v).
26  atom(X) ← subformula(X), X:=c(f).
27
28  % check whether an interpretation is a model or not
29  ismodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I=model.
```

```
30  ismodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, not att(X,S), not att_supp(X,S),
        in(S,model).
31  ismodel(X,S,I) ← subformula(S,X), atom(X), X:=c(v), modelcheck(I), I=model.
32  ismodel(X,S,I) ← subformula(S,X), atom(X), X:=c(v), modelcheck(I), I!=model, in(S,model).
33  ismodel(F,S,I) ← subformula(S,F;F1), F:=neg(F1), nomodel(F1,S,I).
34  ismodel(F,S,I) ← subformula(S,F), F:=and(F1,F2), ismodel(F1;F2,S,I).
35  ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=or(F1,F2), ismodel(F1,S,I).
36  ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=or(F1,F2), ismodel(F2,S,I).
37  ismodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F1,S,I), nomodel(F2,S,I).
38  ismodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F2,S,I), nomodel(F1,S,I).
39  ismodel(F,S,I) ← subformula(S,F;F1;F2), F:=imp(F1,F2), nomodel(F1,S,I).
40  ismodel(F,S,I) ← subformula(S,F), F:=imp(F1,F2), ismodel(F1;F2,S,I).
41  ismodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), ismodel(F1;F2,S,I).
42  ismodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F1;F2,S,I).
43
44  nomodel(X,S,I) ← subformula(S,X), atom(X), out(X,I), I=model.
45  nomodel(X,S,I) ← subformula(S,X), atom(X), out(X,I), I!=model, in(S,model).
46  nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, att(X,S), in(s,model).
47  nomodel(X,S,I) ← subformula(S,X), atom(X), in(X,I), I!=model, att_supp(X,S), in(s,model).
48  nomodel(X,S,I) ← subformula(S,X), atom(X), X:=c(f), modelcheck(I), I=model.
49  nomodel(X,S,I) ← subformula(S,X), atom(X), X:=c(f), modelcheck(I), I!=model, in(S,model).
50  nomodel(F,S,I) ← subformula(S,F;F1), F:=neg(F1), ismodel(F1,S,I).
51  nomodel(F,S,I) ← subformula(S,F;F1;F2), F:=and(F1,F2), nomodel(F1,S,I).
52  nomodel(F,S,I) ← subformula(S,F;F1;F2), F:=and(F1,F2), nomodel(F2,S,I).
53  nomodel(F,S,I) ← subformula(S,F), F:=or(F1,F2), nomodel(F1,S,I), nomodel(F2,S,I).
54  nomodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), ismodel(F1,S,I), ismodel(F2,S,I).
55  nomodel(F,S,I) ← subformula(S,F), F:=xor(F1,F2), nomodel(F1,S,I), nomodel(F2,S,I).
56  nomodel(F,S,I) ← subformula(S,F), F:=imp(F1,F2), ismodel(F1,S,I), nomodel(F2,S,I).
57  nomodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F1,S,I), ismodel(F2,S,I).
58  nomodel(F,S,I) ← subformula(S,F), F:=iff(F1,F2), nomodel(F2,S,I), ismodel(F1,S,I).
59
60  % guess a set M and R
61  in(X,m) ← not out(X,m), statement(X).
62  out(X,m) ← not in(X,m), statement(X).
63
64  in(X,r) ← not out(X,r), not att(X,Y), not att_supp(X,Y), statement(X), in(Y,m).
65  out(X,r) ← not in(X,r), statement(X).
66
67  % M can only be a model if no element of R is also in M
68  in(X,model) ← in(X,m), out(X,r).
69  out(X,model) ← statement(X), not in(X,model).
70  modelcheck(model).
71
72  % every M which is not checked to be a model has to be removed (as it cant be a stable model)
73  ← in(X,m), not in(X,model).
74
75  % check if the guess was right
76  ← in(S,model), ac(S,F), nomodel(F,S,model).
77  ← out(S,model), ac(S,F), ismodel(F,S,model), not in(S,r).
78
79  % removal of all not selected elements is implicit done by setting them to "out"
80  modelStatementCount(I) ← I:={in(S,model):statement(S)}.
81  iteration(I) ← modelStatementCount(I).
82  out(S,I) ← statement(S), modelStatementCount(I).
83  modelcheck(I) ← iteration(I).
84
85  iteration(J) ← iteration(I), J:=I-1, I>0.
86  in(S,J) ← iteration(I), iteration(J), J:=I-1, ismodel(F,S,I), ac(S,F), not in(S,r).
87  out(S,J) ← iteration(I), iteration(J), J:=I-1, not in(S,J), out(S,I).
88
89  in(S) ← in(S,J), J:=0.
90  out(S) ← statement(S), not in(S).
91
92  ← in(S,model), not in(S).
93
94  %subset maximization of in(x).
95  #minimize[out(X)].
```

B.7 Well-founded model

```
1  % splitting a formula into its subformulas
2  subformula(X,F) ← ac(X,F),statement(X).
3  subformula(X,F) ← subformula(X,and(F,_)).
4  subformula(X,F) ← subformula(X,and(_,F)).
5  subformula(X,F) ← subformula(X,or(_,F)).
6  subformula(X,F) ← subformula(X,or(F,_)).
7  subformula(X,F) ← subformula(X,neg(F)).
8  subformula(X,F) ← subformula(X,xor(F,_)).
9  subformula(X,F) ← subformula(X,xor(_,F)).
10 subformula(X,F) ← subformula(X,imp(F,_)).
11 subformula(X,F) ← subformula(X,imp(_,F)).
12 subformula(X,F) ← subformula(X,iff(F,_)).
13 subformula(X,F) ← subformula(X,iff(_,F)).
14 subformula(F) ← subformula(_,F).
15
16 % decide whether a subformula is an atom or not
17 noatom(F) ← subformula(F;F1;F2), F:=and(F1,F2).
18 noatom(F) ← subformula(F;F1;F2), F:=or(F1,F2).
19 noatom(F) ← subformula(F;F1), F:=neg(F1).
20 noatom(F) ← subformula(F;F1;F2), F:=xor(F1,F2).
21 noatom(F) ← subformula(F;F1;F2), F:=imp(F1,F2).
22 noatom(F) ← subformula(F;F1;F2), F:=iff(F1,F2).
23
24 atom(X) ← subformula(X), not noatom(X).
25 atom(X) ← subformula(X), X:=c(v).
26 atom(X) ← subformula(X), X:=c(f).
27
28 % check whether an interpretation is a model or not at a specific iteration
29 ismodel(X,I) ← atom(X), in(X,I).
30 ismodel(X,I) ← atom(X), X:=c(v), iteration(I).
31 ismodel(F,I) ← subformula(F;F1), F:=neg(F1), nomodel(F1,I).
32 ismodel(F,I) ← subformula(F), F:=and(F1,F2), ismodel(F1;F2,I).
33 ismodel(F,I) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F1,I).
34 ismodel(F,I) ← subformula(F;F1;F2), F:=or(F1,F2), ismodel(F2,I).
35 ismodel(F,I) ← subformula(F), F:=xor(F1,F2), ismodel(F1,I), nomodel(F2,I).
36 ismodel(F,I) ← subformula(F), F:=xor(F1,F2), ismodel(F2,I), nomodel(F1,I).
37 ismodel(F,I) ← subformula(F;F1;F2), F:=imp(F1,F2), nomodel(F1,I).
38 ismodel(F,I) ← subformula(F), F:=imp(F1,F2), ismodel(F1;F2,I).
39 ismodel(F,I) ← subformula(F), F:=iff(F1,F2), ismodel(F1;F2,I).
40 ismodel(F,I) ← subformula(F), F:=iff(F1,F2), nomodel(F1;F2,I).
41
42 nomodel(X,I) ← atom(X), out(X,I).
43 nomodel(X,I) ← atom(X), X:=c(f), iteration(I).
44 nomodel(F,I) ← subformula(F;F1), F:=neg(F1), ismodel(F1,I).
45 nomodel(F,I) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F1,I).
46 nomodel(F,I) ← subformula(F;F1;F2), F:=and(F1,F2), nomodel(F2,I).
47 nomodel(F,I) ← subformula(F), F:=or(F1,F2), nomodel(F1,I), nomodel(F2,I).
48 nomodel(F,I) ← subformula(F), F:=xor(F1,F2), ismodel(F1,I), ismodel(F2,I).
49 nomodel(F,I) ← subformula(F), F:=xor(F1,F2), nomodel(F1,I), nomodel(F2,I).
50 nomodel(F,I) ← subformula(F), F:=imp(F1,F2), ismodel(F1,I), nomodel(F2,I).
51 nomodel(F,I) ← subformula(F), F:=iff(F1,F2), nomodel(F1,I), ismodel(F2,I).
52 nomodel(F,I) ← subformula(F), F:=iff(F1,F2), nomodel(F2,I), ismodel(F1,I).
53
54 % get the number of statements and create an ordering
55 snum(I) ← I:={statement(Y)}.
56 iteration(I) ← snum(J), I:=J−1.
57 iteration(I) ← iteration(J), I:=J−1, I>=0.
58
59 % create undecided set of variables at the starting point of the function
60 undec(X,I) ← snum(I), statement(X).
61
62 % iterate the function one step further, and guess an additional element for A or R
63 inA(X,I) ← inA(X,J), J:=I+1, iteration(I).
64 inR(X,I) ← inR(X,J), J:=I+1, iteration(I).
65 select(X,I) ← not deselect(X,I), statement(X), iteration(I), undec(X,J), J:=I+1.
66 deselect(X,I) ← not select(X,I), statement(X), iteration(I), undec(X,J), J:=I+1.
67 ← A:={select(_,I)}, iteration(I), A>1.
68 undec(X,I) ← iteration(I), undec(X,J), J:=I+1, deselect(X,I).
69 % check whether the selected element is in A or not.
70 in(X,I) ∨ out(X,I) ← undec(X,J), J:=I+1, iteration(I).
```

```
71   in(X,I) ← iteration(I), J=I+1, inA(X,J).
72   out(X,I) ← iteration(I), J=I+1, inR(X,J).
73
74   okA(I) ← select(X,I), ac(X,F), ismodel(F,I).
75   okA(I) ← A:={select(_,I)}, iteration(I), A=0.
76   inA(X,I) ← okA(I), select(X,I).
77
78   in(X,I) ← okA(I), undec(X,J), J:=I+1, iteration(I).
79   out(X,I) ← okA(I), undec(X,J), J:=I+1, iteration(I).
80
81
82   % check whether the selected element is in R or not.
83   okR(I) ← select(X,I), ac(X,F), nomodel(F,I), not okA(I).
84   in(X,I) ← okR(I), undec(X,J), J:=I+1, iteration(I).
85   out(X,I) ← okR(I), undec(X,J), J:=I+1, iteration(I).
86   inR(X,I) ← okR(I), select(X,I).
87
88   ok(I) ← okA(I).
89   ok(I) ← okR(I).
90
91   ← not ok(I), iteration(I).
92
93
94   wf(X) ← inA(X,0).
95   r(X) ← inR(X,0).
96
97   #maximize [wf(X)@2].
98   #maximize [r(X)@1].
99
100  #hide.
101  #show wf/1.
102  #show r/1.
```

Bibliography

Amgoud, L., Belabbes, S., and Prade, H. (2005). Towards a formal framework for the search of a consensus between autonomous agents. In Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M. P., and Wooldridge, M., editors, *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 537–543. ACM.

Amgoud, L. and Cayrol, C. (2002). Inferring from inconsistency in preference-based argumentation frameworks. *J. Autom. Reasoning*, 29(2):125–169.

Amgoud, L., Cayrol, C., Lagasquie-Schiex, M.-C., and Livet, P. (2008). On bipolarity in argumentation frameworks. *Int. J. Intell. Syst.*, 23(10):1062–1093.

Apt, K. R. (1997). *From Logic Programming to Prolog*, volume 368. Prentice Hall.

Baroni, P., Caminada, M., and Giacomin, M. (2011a). An introduction to argumentation semantics. *Knowledge Eng. Review*, 26(4):365–410.

Baroni, P., Cerutti, F., Giacomin, M., and Guida, G. (2011b). AFRA: Argumentation framework with recursive attacks. *Int. J. Approx. Reasoning*, 52(1):19–37.

Barth, E. M. and Krabbe, E. C. (1982). *From Axiom to Dialogue: A Philosophical Study of Logics and Argumentation*. W. de Gruyter.

Bench-Capon, T. J. M. (2002). Representation of case law as an argumentation framework. In Bench-Capon, T. J. M., Daskalopoulu, A., and Winkels, R., editors, *Proceedings of the Fifteenth Annual Conference on Legal Knowledge and Information Systems (JURIX 2002)*, pages 103–112. IOS Press.

Bench-Capon, T. J. M. and Dunne, P. E. (2005). Argumentation in AI and law: Editors' introduction. *Artif. Intell. Law*, 13(1):1–8.

Bench-Capon, T. J. M. and Dunne, P. E. (2007). Argumentation in Artificial Intelligence. *Artif. Intell.*, 171(10-15):619–641.

Bench-Capon, T. J. M., Prakken, H., and Sartor, G. (2009). Argumentation in legal reasoning. In Simari, G. and Rahwan, I., editors, *Argumentation in Artificial Intelligence*, pages 363–382. Springer US.

Besnard, P. and Hunter, A. (2005). Practical first-order argumentation. In Veloso, M. M. and Kambhampati, S., editors, *The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI)*, pages 590–595. AAAI Press / The MIT Press.

Besnard, P. and Hunter, A. (2008). *Elements of Argumentation*, volume 47. MIT Press Cambridge.

Boella, G., Gabbay, D. M., van der Torre, L. W. N., and Villata, S. (2009). Meta-argumentation modelling I: Methodology and techniques. *Studia Logica*, 93(2-3):297–355.

Bondarenko, A., Dung, P. M., Kowalski, R. A., and Toni, F. (1997). An abstract, argumentation-theoretic approach to default reasoning. *Artif. Intell.*, 93(1–2):63 – 101.

Brewka, G., Dunne, P. E., and Woltran, S. (2011a). Relating the semantics of abstract dialectical frameworks and standard AFs. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 780–785. AAAI Press.

Brewka, G. and Eiter, T. (2007). Equilibria in heterogeneous nonmonotonic multi-context systems. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 385–390. AAAI Press.

Brewka, G. and Eiter, T. (2009). Argumentation context systems: A framework for abstract group argumentation. In Erdem, E., Lin, F., and Schaub, T., editors, *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 5753 of *Lecture Notes in Computer Science*, pages 44–57. Springer.

Brewka, G., Eiter, T., Fink, M., and Weinzierl, A. (2011b). Managed multi-context systems. In Walsh, T., editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 786–791. AAAI Press.

Brewka, G., Eiter, T., and Truszczyński, M. (2011c). Answer set programming at a glance. *Commun. ACM*, 54(12):92–103.

Brewka, G. and Gordon, T. F. (2010). Carneades and abstract dialectical frameworks: A reconstruction. In Baroni, P., Cerutti, F., Giacomin, M., and Simari, G. R., editors, *Computational Models of Argument: Proceedings of COMMA 2010*, volume 216 of *Frontiers in Artificial Intelligence and Applications*, pages 3–12. IOS Press.

Brewka, G. and Woltran, S. (2010). Abstract dialectical frameworks. In Lin, F., Sattler, U., and Truszczyński, M., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010*, pages 102–111. AAAI Press.

Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M., Sirianni, M., Terracina, G., and Veltri, P. (2011). The third answer set programming competition: Preliminary report of the system competition track. In Delgrande, J. and Faber, W.,

editors, *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*, pages 388–403. Springer Berlin / Heidelberg.

Caminada, M. (2006). On the issue of reinstatement in argumentation. In Fisher, M., van der Hoek, W., Konev, B., and Lisitsa, A., editors, *10th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 4160 of *Lecture Notes in Computer Science*, pages 111–123. Springer.

Caminada, M. and Amgoud, L. (2007). On the evaluation of argumentation formalisms. *Artif. Intell.*, 171(5-6):286–310.

Caminada, M. and Wu, Y. (2011). On the limitations of abstract argumentation. In Causmaecker, P. D., Maervoet, J., Messelis, T., Verbeeck, K., and Vermeulen, T., editors, *Proceedings of the 23rd Benelux Conference on Artificial Intelligence (BNAIC 2011)*, pages 51–58.

Chesñevar, C. I., Maguitman, A. G., and Loui, R. P. (2000). Logical models of argument. *ACM Comput. Surv.*, 32(4):337–383.

Church, A. (1996). *Introduction to Mathematical Logic, pt. 1 Reprint*. Princeton Univeristy Press.

Coste-Marquis, S., Devred, C., and Marquis, P. (2006). Constrained argumentation frameworks. In Doherty, P., Mylopoulos, J., and Welty, C. A., editors, *Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 112–122. AAAI Press.

Denecker, M., Vennekens, J., Bond, S., Gebser, M., and Truszczyński, M. (2009). The second answer set programming competition. In Erdem, E., Lin, F., and Schaub, T., editors, *10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 5753 of *Lecture Notes in Computer Science*, pages 637–654. Springer.

Dimopoulos, Y., Moraitis, P., and Amgoud, L. (2009). Extending argumentation to make good decisions. In Rossi, F. and Tsoukiàs, A., editors, *Algorithmic Decision Theory, First International Conference (ADT)*, volume 5783 of *Lecture Notes in Computer Science*, pages 225–236. Springer.

Dimopoulos, Y. and Torres, A. (1996). Graph theoretical structures in logic programs and default theories. *Theor. Comput. Sci.*, 170(1-2):209–244.

Dung, P. M. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358.

Dvořák, W. (2011). On the complexity of computing the justification status of an argument. In *First International Workshop on the Theory and Applications of Formal Argumentation (TAFA-11)*, Barcelona, Catalonia, Spain,.

Dvořák, W., Järvisalo, M., Wallner, J. P., and Woltran, S. (2012). Complexity-sensitive decision procedures for abstract argumentation. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, pages 54–64. AAAI Press.

Dvořák, W., Morak, M., Nopp, C., and Woltran, S. (2011). dynpartix - a dynamic programming reasoner for abstract argumentation. *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011).*

Dvořák, W., Pichler, R., and Woltran, S. (2010). Towards fixed-parameter tractable algorithms for argumentation. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR'10)*, pages 112–122.

Egly, U., Gaggl, S. A., and Woltran, S. (2008). Aspartix: Implementing argumentation frameworks using answer-set programming. In de la Banda, M. G. and Pontelli, E., editors, *Logic Programming, 24th International Conference, ICLP 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 734–738. Springer.

Egly, U., Gaggl, S. A., and Woltran, S. (2010). Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177.

Eiter, T. and Gottlob, G. (1995). On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323.

Ellmauthaler, S. and Wallner, J. P. (2012). Evaluating Abstract Dialectical Frameworks with ASP. In Verheij, B., Szeider, S., and Woltran, S., editors, *Proceedings of the 4th Conference on Computational Models of Argument (COMMA 2012)*, volume 245, pages 505–506. IOS Press.

Gabbay, D. M. (2009). Fibring argumentation frames. *Studia Logica*, 93(2-3):231–295.

Gebser, M., Kaminski, R., and Schaub, T. (2011a). Complex optimization in answer set programming. *Computing Research Repository (CoRR)*, abs/1107.5742.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. T. (2011b). Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. (2007a). clasp: A Conflict-Driven Answer Set Solver. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer.

Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., and Truszczyński, M. (2007b). The first answer set programming system competition. In Baral, C., Brewka, G., and Schlipf, J. S., editors, *9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 4483 of *Lecture Notes in Computer Science*, pages 3–17. Springer.

Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming (ICLP/SLP)*, pages 1070–1080.

Giunchiglia, F. and Serafini, L. (1994). Multilanguage hierarchical logics or: How we can do without modal logics. *Artif. Intell.*, 65(1):29–70.

Gordon, T. F., Prakken, H., and Walton, D. N. (2007). The Carneades model of argument and burden of proof. *Artif. Intell.*, 171:875–896.

Gordon, T. F. and Walton, D. (2009). Proof burdens and standards. In Simari, G. and Rahwan, I., editors, *Argumentation in Artificial Intelligence*, pages 239–258. Springer US.

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). The evolution of lua. In *Proceedings of the third Association for Computing Machinery (ACM) Special Interest Group on Programming Languages (SIGPLAN) conference on History of Programming Languages (HOPL)*, pages (2–1)–(2–26). ACM.

Johnson, D. S. (1992). *Handbook of Theoretical Computer Science: Algorithms and Complexity*, chapter 2: A Catalog of Complexity Classes, pages 68–162. MIT Press Cambridge.

Kakas, A. C. and Moraitis, P. (2006). Adaptive agent negotiation via argumentation. In Nakashima, H., Wellman, M. P., Weiss, G., and Stone, P., editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, pages 384–391. ACM.

Leitsch, A. (1997). *The Resolution Calculus*. Springer-Verlag.

Lundström, J. E. (2009). *On the Formal Modeling of Games of Language and Adversarial Argumentation: A Logic-Based Artificial Intelligence Approach*. Universitetsbiblioteket, Uppsala University. Ph.D. Thesis.

Lundström, J. E., Aceto, G., and Hamfelt, A. (2011a). A dynamic metalogic argumentation framework implementation. In Bassiliades, N., Governatori, G., and Paschke, A., editors, *Rule-Based Reasoning, Programming, and Applications - 5th International Symposium (RuleML Europe)*, volume 6826 of *Lecture Notes in Computer Science*, pages 83–98. Springer.

Lundström, J. E., Aceto, G., and Hamfelt, A. (2011b). Towards a dynamic metalogic implementation of legal argumentation. In Ashley, K. D. and van Engers, T. M., editors, *The 13th International Conference on Artificial Intelligence and Law (ICAIL)*, pages 91–95. ACM.

Modgil, S. (2009). Reasoning about preferences in argumentation frameworks. *Artif. Intell.*, 173(9–10):901 – 934.

Modgil, S. and Bench-Capon, T. J. M. (2011). Metalevel argumentation. *J. Log. Comput.*, 21(6):959–1003.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Prakken, H. and Sartor, G. (1997). Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, 7(1).

Prakken, H. and Vreeswijk, G. (2002). Logics for defeasible argumentation. *Handbook of Philosophical Logic*, 4:219–318.

Rahwan, I. and Simari, G., editors (2009). *Argumentation in Artificial Intelligence*. Springer US.

Rothmaler, P. (2000). *Introduction to Model Theory*. Algebra, Logic, and Applications. Gordon & Breach.

South, M., Vreeswijk, G., and Fox, J. (2008). Dungine: A java dung reasoner. In Besnard, P., Doutre, S., and Hunter, A., editors, *Proceedings of the 2nd Conference on Computational Models of Argument (COMMA 208)*, volume 172 of *Frontiers in Artificial Intelligence and Applications*, pages 360–368. IOS Press.

Toulmin, S. E. (2003). *The uses of Argument*. Press Syndicate of the University of Cambridge, 2nd edition.

Verheij, B. (2007). A labeling approach to the computation of credulous acceptance in argumentation. In Veloso, M. M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 623–628.

Villata, S., Boella, G., and van der Torre, L. W. N. (2011). Attack semantics for abstract argumentation. In Walsh, T., editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 406–413. IJCAI/AAAI.

Walton, D. N. (1996). *Argumentation Schemes for Presumptive Reasoning*. Lawrence Erlbaum.

Weydert, E. (2011). Semi-stable extensions for infinite frameworks. In Causmaecker, P. D., Maervoet, J., Messelis, T., Verbeeck, K., and Vermeulen, T., editors, *Proceedings of the 23rd Benelux Conference on Artificial Intelligence*, pages 336–343.