Imperial College London
Department of Computing

Developments in Abstract and Assumption-Based Argumentation and their Application in Logic Programming

Claudia Schulz

supervised by
Prof. Francesca Toni
and
Prof. Marek Sergot

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College London

June 2017

Copyright Declaration

Abstract

Logic Programming (LP) and Argumentation are two paradigms for knowledge representation and reasoning under incomplete information. Even though the two paradigms share common features, they constitute mostly separate areas of research. In this thesis, we present novel developments in Argumentation, in particular in *Assumption-Based Argumentation* (ABA) and *Abstract Argumentation* (AA), and show how they can 1) extend the understanding of the relationship between the two paradigms and 2) provide solutions to problematic reasoning outcomes in LP.

More precisely, we introduce *assumption labellings* as a novel way to express the semantics of ABA and prove a more straightforward relationship with LP semantics than found in previous work. Building upon these correspondence results, we apply methods for argument construction and conflict detection from ABA, and for conflict resolution from AA, to construct *justifications* of unexpected or unexplained LP solutions under the answer set semantics. We furthermore characterise reasons for the *non-existence of stable semantics* in AA and apply these findings to characterise different scenarios in which the computation of meaningful solutions in LP under the answer set semantics *fails*.

Acknowledgements

There is a seemingly endless list of people I would like to acknowledge for their support, guidance, help, or simply their company throughout the past four and a half years. Thank you so much to all of you! In addition, various people had a great impact on my academic development as well as on this thesis, and I would like to thank these people in more detail.

Above all, I would like to express my gratitude to Francesca Toni for being an outstanding advisor. I here choose the term "advisor" on purpose (rather than the official term "supervisor"), since I always felt "advised" rather than "supervised" by Francesca – something I immensely appreciate and that promoted my academic development. I am also grateful to Francesca for writing more reference letters than for any other of her PhD students (her words), for genuinely caring about my academic development, and for giving me every career opportunity imaginable.

I would like to thank Kristijonas Čyras for proof-reading this thesis and making useful suggestions for improvements. We also had great technical discussions throughout my time as a PhD student, which helped shaping this work. I am furthermore thankful to Oana Cocarascu for her advice on implementations and for persuading me that everything is always amazing.

Ken Satoh has greatly inspired me with his passion for both research and long-distance running. I am grateful to him for the productive meetings we had during my internship at the NII as well as for introducing me to the Japanese culture. I would also like to express my gratitude to Randy Goebel for his continuous career support. Furthermore, I would like to thank my second supervisor Marek Sergot, who provided guidance during my first year as a PhD student and gave me some great tips for travelling.

A special word of thanks goes to everyone who shared an office with me during my time as a PhD student, especially (in no particular order) Kyriacos Nikiforou, Marta Garnelo, Zafeirios Fountas, Nat Dilokthanakul, Christos Kaplanis, Pedro Martinez Mediano, and Filipe Pinto Teixeira. They shared my ups and downs and made me have a great time at Imperial College. Thank you so much guys (and girls), I will miss your daily company!

I would like to thank Eric Eaton for his support and for being a great friend, and Paolo Turrini for all his advice, the many coffee-breaks, and for being my friend. I am also extremely thankful to my amazing friend and fellow PhD student Silvia Vinyes Mora for our great lunches, which gave me the energy and motivation to work hard in the afternoon, and for listening to both my worries and successes.

Last but by no means least, I am grateful to Dan Wong for supporting me throughout the last four and a half years, for putting up with me during stressful times, and for ensuring that I keep a good work-life balance. I would also like to thank my parents as well as my sister Andrea Schulz and my best friend Dominique Corneillie for all their support and advice.

On a final note, my thanks goes to my examiners Krysia Broda and David Pearce, who read this thesis in all its detail and asked great questions during an enjoyable viva.

Contents

Chapter 1

Introduction

1.1 Motivation

Answer Set Programming (ASP) is one of the most widely used non-monotonic reasoning paradigms, allowing for the efficient computation of solutions to complex problems that require reasoning with defaults and exceptions [Gel08]. It has aided developments in a variety of areas in computer science and has been used as a problem solving paradigm in many different domains. For example, Luitel et al. use ASP to evaluate software models [LSI16], Gagnon and Esfandiari apply ASP for operating system discovery [GE09], Delgrande et al. show how ASP can aid cryptography [DGH09], and Smith and Bryson review the application of ASP for content generation in video games [SB14]. In other domains, ASP has been applied for solving problems such as resource allocation [RGA$^+$12], handling biological information [BCT$^+$04, Erd11, GSTV11, KOJS15], identifying inconsistencies in medical databases [TML13], and refining psychological theories [BG10, Inc15].

A problem to be solved with ASP is represented in terms of a *logic program*, which consists of if-then clauses whose literals (i.e. the statements in the if-then clauses) can be negated in two different ways: using negation-as-failure (NAF) or explicit negation. NAF literals only occur in the if-part of clauses and express exceptions on the applicability of a clause, whereas explicitly negated literals express the opposite (or classical negation) of a literal and can occur both in the if- and the then-part of a clause. The solutions to a problem represented as a logic program are then given by the declarative *answer set semantics* [GL91]. A logic program can have various different answer sets, each representing a different "acceptable" set of literals, which together satisfy the problem encoding. Answer set solvers like clingo [GKK$^+$11, GKKS14], smodels [SN01], DLV [ELM$^+$97, LPF$^+$02, LPF$^+$06], WASP [ADF$^+$13, ADLR15], and ME-ASP [MPR14, MPR15] provide efficient tools for the computation of the answer set semantics, thus facilitating the application of ASP for real-world problem solving.

The solutions of a problem to be solved using ASP heavily rely on the exact encoding of this problem as a logic program. Different or erroneous encodings can therefore result in different or unintended solutions. Furthermore, it is not always obvious why an answer set is the solution of the encoded problem. To illustrate this issue, consider the following (simple) problem to be solved using ASP.

An ophthalmologist has to decide whether his short-sighted patient should get corrective lenses, i.e. glasses or contact lenses, or have laser surgery. To encode this decision making problem in ASP, the doctor considers exception conditions under which either corrective lenses or laser surgery are not a good choice. Since laser surgery is rather expensive, the doctor adds an exception condition to the clause representing the laser surgery choice, expressing that laser surgery is an option as long as there is no evidence that the patient is tight on money. Furthermore, the doctor adds a clause representing his common-sense knowledge that students are usually tight on money, as well as factual knowledge he has about his patient, namely that the patient is short-sighted and a student. This results in the following encoding as a logic program:[1]

$$\begin{aligned}
\{\ correctiveLenses &\leftarrow shortSighted;\\
laserSurgery &\leftarrow shortSighted, \texttt{not}\ tightOnMoney;\\
tightOnMoney &\leftarrow student;\\
shortSighted &\leftarrow\ ;\\
student &\leftarrow\ \}
\end{aligned}$$

This logic program has a single answer set: $\{student, shortSighted, tightOnMoney, correctiveLenses\}$. Thus, according to ASP, the solution to the decision problem is that the patient should get some form of corrective lenses. The doctor was unsure whether corrective lenses or laser surgery would be more suitable, so is now faced with the problem of whether or not to trust the decision made by ASP.

If ASP is to be used for solving real-world problems such as aiding decision making, it is thus often important that the user *understands* how the solution came about in order to trust the solution. This is also important, if the user expected a different solution.

In addition to unintended, unexpected, or non-understandable answer sets, the computation of answer sets sometimes *fails* altogether. Such failure occurs in two different ways: on the one hand, no answer sets may be computed at all; on the other hand, a single answer set may be computed that consists of all literals occurring in the logic program (which is not generally a meaningful solution as it expresses that everything is "acceptable", including conflicting information). Such ASP failure is caused by encodings that cannot be rationally satisfied, which may be due to mistakes in the way a problem is encoded as a logic program.

Especially when non-ASP-experts use ASP for problem solving, non-understandable solutions or failure of an ASP solver is problematic. In this thesis, we deal with both types of problems.

1. Concerning unexpected or non-understandable answer sets, we propose a method for *explaining why* a literal is contained in an answer set (e.g. if the user expects that the literal is not part of the solution) or why it is not contained in an answer set (e.g. if the user expects that the literal is part of the solution).

2. Concerning ASP failure, we characterise four different *failure scenarios* and *culprit literals*, which are responsible for the failure in each scenario.

[1]The right-hand side of the arrow constitutes the if-part of a clause and the left-hand side the then-part. Clauses with an empty right-hand side represent facts and `not` denotes NAF. Note that this simple example does not comprise any explicit negation.

1.2 Approach

In order to deal with non-understandable ASP solutions and ASP failure, we apply methods from the field of *computational argumentation*.

1.2.1 Argumentation

The study of computational argumentation is concerned with the development of frameworks and computational tools for representing arguments and interactions between them and determining sets of accepted arguments. Many different argumentation frameworks have been proposed; they can be divided into *abstract* and *structured* frameworks.

The former (e.g. [Dun95b, Mod09, CLS13]) assume that a set of arguments, which are abstract entities, and interactions between them are given. The most prominent (and most simple) abstract framework was introduced by Dung, whose *Abstract Argumentation* (AA) framework [Dun95b] comprises attacks between arguments as the only interactions. Dung defined different semantics for determining sets of accepted arguments in AA, so-called *argument extensions*. These semantics were later reformulated in terms of *argument labellings*, which assign to each argument one of the labels "accepted", "rejected", or "undecided" [CG09].

In contrast to abstract frameworks, structured frameworks (e.g. [BDKT97, BH01, GS04, MP13], see [BGH$^+$14] for an overview) assume that domain and problem-specific knowledge is given in some underlying logical language, e.g. in terms of inference rules, facts, and information that is true by default. Based on this knowledge, structured frameworks provide mechanisms for constructing arguments. Importantly, the logical language of a structured framework must also include a notion of *contrary*, for example classical negation in propositional logic, which allows to determine conflicts between the constructed arguments.

In this thesis, we make use of a structured framework called *Assumption-Based Argumentation* (ABA) [BDKT97, DKT09, Ton14], which is inspired by logic programming, default logic and other non-monotonic reasoning paradigms closely related to ASP. Structured knowledge in an ABA framework is given in terms of inference rules made of sentences in some underlying logical language. A subset of sentences is defined as *assumptions*, representing information assumed to hold by default. For each assumption, a *contrary* sentence in the language is defined. In contrast to AA frameworks, where semantics are expressed in terms of sets of accepted arguments, ABA semantics are typically expressed in terms of sets of accepted assumptions, called *assumption extensions*.

Given a *flat* ABA framework (where assumptions cannot be deduced from other assumptions) arguments and attacks between them can be constructed. A flat ABA framework thus instantiates an AA framework comprising all arguments and attacks constructable in the flat ABA framework [DMT07]. We call such an instantiated AA framework the *corresponding AA framework* of the underlying ABA framework. Then the semantics of AA frameworks can be applied to a flat ABA framework by means of its corresponding AA framework [DMT07].

1.2.2 Argumentation versus Logic Programs

Since ABA frameworks operate on the basis of inference rules just like ASP, an ABA framework can express the same information as a logic program [BDKT97]. We call an ABA framework representing a logic program the *translated ABA framework* of the underlying logic program. Furthermore, since a translated ABA framework is guaranteed to be flat, it instantiates a corresponding AA framework, so a logic program can also be encoded in terms of the corresponding AA framework of the translated ABA framework. We call such an AA framework the *translated AA framework* of the underlying logic program.

One of the semantics of ABA and AA frameworks (in terms of assumption and argument extensions, respectively) is the stable semantics [BDKT97, Dun95b], which has its roots in the *stable model semantics* for logic programs [GL88]. On the other hand, the answer set semantics of logic programs is an extension of the stable model semantics for logic programs [GL91]. It is thus unsurprising that answer sets of a logic program and stable assumption/argument extensions of the translated ABA/AA framework correspond [BDKT97, Dun95b]. In this thesis, we make use of this semantic connection between logic programs and ABA/AA frameworks as it allows to apply methods developed for ABA and AA to logic programs.

To study the semantic connection between ABA frameworks and logic programs in more detail, we introduce a labelling-based semantics for ABA, inspired by the labelling semantics for AA frameworks. Using these new *assumption labellings*, we are able to extend existing correspondence results between the semantics of a logic program and the translated ABA framework by showing a more detailed correspondence. In addition to investigating the stable ABA semantics and answer set semantics for logic programs, which is needed to apply ABA concepts to logic programs under the answer set semantics, we also consider and relate other semantics of ABA frameworks and logic programs. Furthermore, to provide a full picture of semantic correspondence, we also investigate semantic correspondence between logic programs and AA frameworks, extending existing correspondence results by using our correspondence results between logic programs and translated ABA frameworks.

Note that even though we will more frequently refer to the translated AA framework than to the translated ABA framework of a logic program when using argumentation methods to solve problems in ASP, the translated AA framework is built from the translated ABA framework. Thus, the semantic correspondence between a logic program and its translated AA framework relies on the correspondence between the logic program and the translated ABA framework together with the semantic correspondence between the translated ABA framework and the translated AA framework. The translated ABA framework consequently plays an important (albeit implicit) role in our investigations.

1.2.3 Explaining Answer Sets using Argumentation

If a solution computed by an ASP solver is unexpected or the user simply wants to know why a literal is or is not part of an answer set, an explanation is desirable. We make use of the correspondence results between the answer set semantics of a logic program and the stable semantics of the translated ABA and AA frameworks to *explain why* literals are (not) contained in an answer set in terms of arguments (not) contained in a stable argument extension of the translated AA framework. We propose two types of argumentative explanations, which can both be interpreted as a dialogue between two adversaries arguing about the "truth" of the literal in question. When justifying a literal contained in an answer set, the proponent, who is trying to explain why the literal in question should be regarded as "true", is able to refute all evidence against the "truth" of the literal given by the opponent. On the other hand, when justifying a literal not contained in an answer set, the proponent is not able to refute all evidence given by the opponent.

The first justification approach, an *Attack Tree*, expresses how to construct an argument for the literal in question (the supporting argument given by the proponent) as well as which arguments attack the argument for the literal in question (the attacking arguments given by the opponent). The same information is provided for all arguments attacking the attacking arguments (given by the proponent), and so on.

The second justification approach, an *ABA-Based Answer Set (ABAS) Justification* of a literal, represents similar information to an Attack Tree, but expressed in terms of literals rather than arguments. An ABAS Justification comprises facts and NAF literals necessary to derive the literal in question (the "supporting literals") as well as information about literals that are in conflict with the literal in question (the "attacking literals"). The same information is provided for all supporting and attacking literals of the literal in question, for all their supporting and attacking literals, and so on.

Attack Trees may be more suitable for non-ASP experts since they provide explanations in terms of arguments, whereas ABAS Justifications may be more suitable for ASP experts as they provide explanations in terms of literals.

1.2.4 Characterising and Explaining ASP Failure using Argumentation

If an ASP solver is unable to compute answer sets at all or yields the set of all literals occurring in the logic program as the only answer set, the logic program is *inconsistent*. In such a case, it is useful to know what caused the inconsistency, in particular, which part of the logic program is responsible for the inconsistency.

We again aim at applying argumentation methods for solving the problem of ASP failure. Therefore, we first investigate inconsistency in argumentation, i.e. the non-existence of stable labellings. Rather than basing this investigation on translated ABA frameworks, we abstract away from the structure of arguments constructed from a logic program and instead give more general results, which apply to any AA framework. In particular, we introduce the first characterisation of parts of an AA framework that are responsible for the non-existence of stable argument labellings. Additionally, we introduce a method for obtaining a stable argument labelling by revising the responsible parts. Based on our semantic correspondence result between a logic program and the translated AA framework, we then transfer these inconsistency results from AA frameworks to logic programs. This yields a characterisation of parts of a logic program without explicit negation that are responsible for the logic program being inconsistent.

We then propose a method for identifying the reason of inconsistency in *any* logic program based on the well-founded [VRS91] and M-stable [ELS97] model semantics. These semantics are “weaker” than the answer set semantics in that they are 3-valued rather than 2-valued. We prove that the two ways in which a logic program may be inconsistent (i.e. no answer sets or all literals as the only answer set) can in fact be divided into four inconsistency cases, which provide different reasons for the inconsistency: one where only explicit negation is responsible, one where only NAF is responsible, and two where the interplay of explicit negation and NAF is responsible. We show how in each of these inconsistency cases the reason of the inconsistency can be refined to a characteristic set of *culprit literals*. In the case where only NAF is responsible, these culprit literals are characterised in the same way as responsible parts of a logic program without explicit negation. We thus apply our characterisation of the non-existence of stable argument labellings of AA frameworks to logic programs. Finally, we show how culprit literals can be used to explain why the inconsistency arises by constructing explanations trees which are similar to our Attack Trees for consistent logic programs.

1.2.5 Approach Summary

Both issues investigated in this thesis, namely non-understandable ASP solutions and ASP failure, pose problems for the user if he or she is unable to understand the ASP behaviour. Therefore, human-understandable explanations of the ASP behaviour is an important component of both issues, which we here address through argumentation.

Argumentation has been used as a tool for constructing dialectical explanations in a variety of domains, e.g. linked open data [ACP^{+}16], decision making [ZFTL14], and belief revision [FKIS02]. We show in this thesis that argumentation is also an ideal formalism for the explanation of ASP behaviour, in particular the explanation of literals with respect to answer sets and the explanation of ASP failure scenarios.

In addition to its explanatory capabilities, developments in argumentation can also be beneficial for investigating other issues of ASP. Here, we show how novel findings on the non-existence of stable semantics in AA frameworks can be applied to draw conclusions about the non-existence of answer sets.

1.3 Contributions and Thesis Structure

We give the necessary background on AA and ABA frameworks as well as on logic programs and ASP in Chapter 2 and conclude in Chapter 8. The main contributions of the rest of this thesis are as follows.

- Chapter 3: We introduce a new way of defining ABA semantics, namely in terms of *assumption labellings*, which represent a more refined interpretation than assumption extensions. We prove that there is a one-to-one correspondence between the new assumption labellings and assumption extensions, and investigate the correspondence with argument labellings of the corresponding AA framework of a flat ABA framework. In addition, we define assumption labellings for *non-flat* ABA frameworks and prove correspondence with assumption extensions of non-flat ABA frameworks.
- Chapter 4: We review and extend existing *correspondence results* between the semantics of a logic program and its translated ABA framework, as well as its translated AA framework, thus improving the understanding of the relationship between the three formalisms. These results are partly based on our new semantic formalisations for ABA frameworks in Chapter 3.
- Chapter 5: Based on the correspondence results from Chapter 4, we propose *argumentative explanations* for literals (not) contained in answer sets of a logic program. This is our first approach that applies methods from Argumentation to aid ASP. We furthermore present a web-platform, which implements our argumentative explanations.
- Chapter 6: We characterise the parts of an AA framework that are *responsible* for the non-existence of stable argument labellings and propose a methodology for turning a preferred argument labelling into a stable one.
- Chapter 7: Using our characterisations from Chapter 6 and our correspondence results from Chapter 4, we characterise the parts of an inconsistent logic program without explicit negation that are responsible for the inconsistency. We then study inconsistent logic programs in general and characterise four *failure scenarios*. For each scenario, we characterise culprit literals responsible for the failure and propose argumentative explanations as to why the failure arises.

1.4 Publications

This thesis combines and builds upon work that has been published or is under review for publication:

- Chapter 3: C. Schulz and F. Toni. Complete Assumption Labellings. In *Proceedings of the 5th International Conference on Computational Models of Argument (COMMA)*, pages 405–412, 2014 [ST14].

- Chapter 3: C. Schulz and F. Toni. Labellings for Assumption-Based and Abstract Argumentation. *International Journal of Approximate Reasoning*, 84, pages 110–149, 2017 [ST17a].

- Chapter 4: C. Schulz and F. Toni. Logic Programming in Assumption-Based Argumentation Revisited – Semantics and Graphical Representation. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1569–1575, 2015 [ST15].

- Chapter 4: M. Caminada and C. Schulz. On the Equivalence between Assumption-Based Argumentation and Logic Programming. In *Proceedings of the 1st International Workshop on Argumentation and Logic Programming (ArgLP)*, 2015 [CS15].

- Chapters 4 and 5: C. Schulz and F. Toni. Justifying Answer Sets using Argumentation. *Theory and Practice of Logic Programming*, 16(01), pages 59–110, 2016 [ST16].

- Chapter 6: C. Schulz and F. Toni. On the Non-Existence and Restoration of Stable Labellings in Abstract Argumentation Frameworks. *Under Review* [ST17b]

- Chapter 7: C. Schulz, K. Satoh and F. Toni. Characterising and Explaining Inconsistency in Logic Programs. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 467–479, 2015 [SST15].

Note that the work published in [DS14] and [SD16] was also performed as part of the PhD studies of the author, but is not included in this thesis as it is only loosely related to the rest of the presented material.

1.5 Statement of Originality

I declare that this thesis was composed by myself and that the work it presents is my own, except where otherwise stated.

Chapter 2

Background

2.1 Introduction

As outlined in the previous chapter, in this thesis we present novel concepts and results regarding argumentation frameworks and apply them for the development of solutions to issues concerning logic programs.

In this chapter, we give some background on the two argumentation frameworks and on ASP used throughout the whole thesis. In addition to the concepts introduced in this chapter, some of the following chapters provide further background specific to the respective chapters.

The chapter is organised as follows. In Section 2.2, we introduce AA and ABA frameworks and their respective notions of semantics, and recall how to construct an AA framework from an ABA framework. In Section 2.3, we introduce logic programs and present the answer set semantics as well as various notions of 3-valued models. We summarise the given background in Section 2.4.

2.2 Argumentation

For the past twenty years, argumentation has been an active field of research in Artificial Intelligence (AI), which incorporates ideas from logic, computer science, philosophy, psychology, and linguistics. Argumentation has been applied to many different domains of AI, such as decision making, multi-agent communication, legal reasoning and explanation (see e.g. [RS09] for an overview).

Two kinds of approaches can be distinguished in argumentation: *abstract* and *structured* approaches. The former consider arguments as abstract entities, which can be instantiated with anything desired by the user, whereas the latter consider arguments to have a specific internal structure, which is based on some underlying structured knowledge.

In this thesis, we consider one abstract and one structured approach, namely Abstract Argumentation frameworks (AA) and Assumption-Based Argumentation (ABA) frameworks, respectively. We chose these two frameworks since they can be considered "lightweight" argumentation frameworks in the sense that they are made of very few components, compared to related frameworks. As we will see throughout this thesis, these few components are sufficient for our purposes, so choosing frameworks with more components would result in unused, and thus for our purposes unnecessary, components.

For example, AA frameworks, as introduced in Section 2.2.1, are made of a set of arguments and a set of attacks between these arguments. In contrast, other *abstract* frameworks comprise components such as a set of support relations between the arguments (e.g [CLS05, ON08, NR10, CLS13, Gab16a]), a set of attack relations from arguments to attacks [Mod09], or a set of values or preferences associated with the arguments (e.g. [BC03, KvdT08, LM11]).

ABA on the other hand, introduced in Section 2.2.2, is a *structured* argumentation framework made of one type of inference rule, a contrariness mapping, and a set of defeasible elements. In contrast, other structured argumentation frameworks comprise additional components. For example, ASPIC+ [MP10, MP13] has two types of inference rules as well as an additional set of facts. Similarly, DeLP (Defeasible Logic Programming) [GS04, GS14] comprises two different types of inference rules.

2.2.1 Abstract Argumentation (AA)

An *Abstract Argumentation (AA) framework* [Dun95b] is a pair $\langle Ar, Att \rangle$, where Ar is a set of arguments and $Att \subseteq Ar \times Ar$ is a binary attack relation between arguments. A pair $(A, B) \in Att$ expresses that argument A *attacks* argument B, or equivalently that B is attacked by A. A set of arguments $Args \subseteq Ar$ attacks an argument $B \in Ar$ if and only if there is $A \in Args$ such that A attacks B. $Args^+ = \{A \in Ar \mid Args \textit{ attacks } A\}$ denotes the set of all arguments attacked by $Args$ [BCG11]. $Args$ attacks a set of arguments $Args'$ if and only if $Args$ attacks some $B \in Args'$.
Let $Args \subseteq Ar$ be a set of arguments.

- $Args$ is *conflict-free* if and only if $Args \cap Args^+ = \emptyset$.
- $Args$ *defends* $A \in Ar$ if and only if $Args$ attacks every $B \in Ar$ attacking A.

The semantics of an AA framework are defined in terms of *argument extensions*, i.e. sets of accepted arguments [Dun95b, DMT07, Cam06b]. A set of arguments $Args \subseteq Ar$ is

- an *admissible argument extension* if and only if $Args$ is conflict-free and defends all arguments $A \in Args$;
- a *complete argument extension* if and only if $Args$ is conflict-free and consists of all arguments it defends;
- a *grounded argument extension* if and only if $Args$ is a minimal (w.r.t. [1] $\subseteq$) complete argument extension;
- a *preferred argument extension* if and only if $Args$ is a maximal (w.r.t. $\subseteq$) complete argument extension;
- an *ideal argument extension* if and only if $Args$ is a maximal (w.r.t. $\subseteq$) admissible argument extension satisfying that for all preferred argument extensions $Args'$, $Args \subseteq Args'$;
- a *semi-stable argument extension* if and only if $Args$ is a complete argument extension and for all complete argument extensions $Args'$, $Args \cup Args^+ \not\subset Args' \cup Args'^+$;
- a *stable argument extension* if and only if $Args$ is a complete argument extension and $Args \cup Args^+ = Ar$.

[1] Throughout this thesis, we often abbreviate "with respect to" as "w.r.t.".

Note that some of these definitions of argument extensions are not the original ones introduced in [Dun95b] but are equivalent formulations [BCG11].

Example 2.1. Let $\mathcal{AA}_1 = \langle \{a, b, c\}, \{(a, b), (b, a), (b, c), (c, c)\} \rangle$ be an AA framework. As any AA framework, $\mathcal{AA}_1$ can be represented by a graph where nodes are arguments and directed edges are attacks between the arguments, as illustrated in Figure 2.1.
The singleton sets $\{a\}$ and $\{b\}$ are conflict-free, whereas $\{c\}$ is not. Furthermore, no set with more than one argument is conflict-free.
Both $\{a\}$ and $\{b\}$ are admissible and complete argument extensions, and so is the empty set. The empty set is furthermore the unique grounded argument extension, and both $\{a\}$ and $\{b\}$ are preferred argument extensions. It then follows, that the only ideal argument extension is the empty set since it is the intersection of $\{a\}$ and $\{b\}$, and is furthermore an admissible argument extension.
The set of arguments attacked by $Args_1 = \{a\}$ is ${Args_1}^+ = \{b\}$, whereas for $Args_2 = \{b\}$ it is ${Args_2}^+ = \{a, c\}$, and for $Args_3 = \{\}$ it is ${Args_3}^+ = \{\}$.
Therefore, $Args_2$ is the only semi-stable argument extension and also the only stable argument extension.

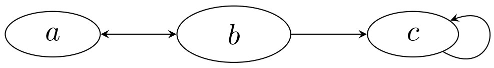


Figure 2.1: The AA framework $\mathcal{AA}_1$ from Example 2.1.

Another way of expressing the semantics of an AA framework is in terms of argument labellings [Cam06a, CG09]. An *argument labelling* is a total function $LabArg : Ar \rightarrow \{\mathtt{in}, \mathtt{out}, \mathtt{undec}\}$. The set of arguments labelled in by $LabArg$ is $\mathtt{in}(LabArg) = \{A \in Ar \mid LabArg(A) = \mathtt{in}\}$; the sets of arguments labelled out and undec are denoted $\mathtt{out}(LabArg)$ and $\mathtt{undec}(LabArg)$, respectively.

An argument labelling $LabArg$ is an *admissible argument labelling* if and only if for each argument $A \in Ar$ it holds that:

- if $LabArg(A) = \mathtt{in}$, then for each $B \in Ar$ attacking A, $LabArg(B) = \mathtt{out}$;
- if $LabArg(A) = \mathtt{out}$, then there exists some $B \in Ar$ attacking A such that $LabArg(B) = \mathtt{in}$.

An argument labelling $LabArg$ is a *complete argument labelling* if and only if it is an admissible argument labelling and for each argument $A \in Ar$ it holds that:

- if $LabArg(A) = \mathtt{undec}$, then there exists some $B \in Ar$ attacking A such that $LabArg(B) = \mathtt{undec}$ and there exists no $C \in Ar$ attacking A such that $LabArg(C) = \mathtt{in}$.

Equivalently, a complete argument labelling can be defined by reversing the conditions[2]. That is, an argument labelling is a complete argument labelling if and only if for each argument $A \in Ar$ it holds that:

- if for each $B \in Ar$ attacking A, $LabArg(B) = \texttt{out}$, then $LabArg(A) = \texttt{in}$;
- if there exists some $B \in Ar$ attacking A such that $LabArg(B) = \texttt{in}$, then $LabArg(A) = \texttt{out}$;
- if there exists some $B \in Ar$ attacking A such that $LabArg(B) = \texttt{undec}$ and there exists no $C \in Ar$ attacking A such that $LabArg(C) = \texttt{in}$, then $LabArg(A) = \texttt{undec}$.

In order to define argument labellings according to other semantics, we first recall how to compare the *commitment* of argument labellings [BCG11].

> Let $LabArg_1$ and $LabArg_2$ be argument labellings. $LabArg_2$ is *more or equally committed* than $LabArg_1$, denoted $LabArg_1 \sqsubseteq LabArg_2$, if and only if $\texttt{in}(LabArg_1) \subseteq \texttt{in}(LabArg_2)$ and $\texttt{out}(LabArg_1) \subseteq \texttt{out}(LabArg_2)$.

A complete argument labelling $LabArg$ is [CG09, Cam11]

- a *grounded argument labelling* if and only if $\texttt{in}(LabArg)$ is minimal (w.r.t. $\subseteq$) among all complete argument labellings;
- a *preferred argument labelling* if and only if $\texttt{in}(LabArg)$ is maximal (w.r.t. $\subseteq$) among all complete argument labellings;
- an *ideal argument labelling* if and only if $LabArg$ is a maximal (w.r.t. $\sqsubseteq$) admissible argument labelling which satisfies that for all preferred argument labellings $LabArg'$, $LabArg \sqsubseteq LabArg'$;
- a *semi-stable argument labelling* if and only if $\texttt{undec}(LabArg)$ is minimal (w.r.t. $\subseteq$) among all complete argument labellings;
- a *stable argument labelling* if and only if $\texttt{undec}(LabArg) = \emptyset$.

Complete, grounded, preferred, ideal, semi-stable, and stable argument extensions correspond one-to-one to the sets of arguments labelled in by the complete, grounded, preferred, ideal, semi-stable, and stable argument labellings, respectively [CG09, Cam11]. In contrast, an admissible argument extension may correspond to various admissible argument labellings [CG09].

Example 2.2. Consider again $\mathcal{AA}_1$ from Example 2.1. It has three complete argument labellings:

- $LabArg_1 = \{(a, \texttt{in}), (b, \texttt{out}), (c, \texttt{undec})\}$,

[2]This follows from Proposition 5 in [CG09].

- $LabArg_2 = \{(a, \texttt{out}), (b, \texttt{in}), (c, \texttt{out})\}$, and
- $LabArg_3 = \{(a, \texttt{undec}), (b, \texttt{undec}), (c, \texttt{undec})\}$.

$LabArg_1$ and $LabArg_2$ are both preferred argument labellings, $LabArg_3$ is the only grounded and only ideal argument labelling, and $LabArg_2$ is the only semi-stable and only stable argument labelling. We note that $LabArg_1$ corresponds to the complete argument extension $Args_1$, $LabArg_2$ to $Args_2$, and $LabArg_3$ to $Args_3$ (see Example 2.1).
All three complete argument labellings are also admissible argument labellings. In addition, there exists an admissible argument labelling that is not a complete argument labelling, namely $LabArg_4 = \{(a, \texttt{out}), (b, \texttt{in}), (c, \texttt{undec})\}$. $LabArg_4$ corresponds to the admissible argument extension $Args_2$, illustrating the one-to-many correspondence between admissible argument extensions and labellings.

Admissible extensions can also be defined using trees of attacking arguments.

An *abstract dispute tree* [DKT06] for an argument $A \in Ar$ is a (possibly infinite) tree such that:

1. every node in the tree is labelled by an argument and is assigned the status of *proponent* or *opponent* node, but not both;
2. the root is a proponent node labelled by A;
3. for every proponent node N labelled by an argument $B \in Ar$ and for every argument $C \in Ar$ attacking B, there exists a child of N that is an opponent node labelled by C;
4. for every opponent node N labelled by an argument $B \in Ar$, there exists exactly one child of N, which is a proponent node labelled by an argument $C \in Ar$ that attacks B;
5. there are no other nodes in the tree except those given by 1-4 above.

An abstract dispute tree is *admissible* [DKT09] if and only if no argument labels both a proponent and an opponent node. It was shown that the set of all arguments labelling proponent nodes in an admissible abstract dispute tree is an admissible argument extension [DMT07].

2.2.2 Assumption-Based Argumentation (ABA)

An *Assumption-Based Argumentation (ABA) framework* [BDKT97, DKT09, Ton14] is a tuple $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ } \rangle$ where:

- $(\mathcal{L}, \mathcal{R})$ is a deductive system, with $\mathcal{L}$ a language of countably many sentences and $\mathcal{R}$ a set of inference rules of the form $s_0 \leftarrow s_1, \ldots, s_n$ $(n \geq 0)$ with $s_0, \ldots, s_n \in \mathcal{L}$;
  s_0 is the *head* of the inference rule and $s_1, \ldots, s_n$ is the *body*;

- $\mathcal{A} \subseteq \mathcal{L}$ is a non-empty set of *assumptions*;
- $^{-}$ is a total mapping from $\mathcal{A}$ into $\mathcal{L}$ defining the *contrary* of assumptions, where $\overline{\alpha}$ denotes the contrary of $\alpha \in \mathcal{A}$.

An ABA framework is *flat* if assumptions occur only in the body of inference rules [DKT06]. For the rest of this section, we assume as given a flat ABA framework $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^{-} \rangle$.

An *argument* [DKT09] for (the conclusion) $s \in \mathcal{L}$ supported by the set of *premises* $Asms \subseteq \mathcal{A}$, denoted $Asms \vdash s$, is a finite tree where every node holds a sentence in $\mathcal{L}$ or the sentence τ (where $\tau \notin \mathcal{L}$ stands for "true") such that:

- the root node holds s;
- for every node N
  - if N is a leaf, then N holds either an assumption or τ;
  - if N is not a leaf and N holds the sentence s_0, then there is an inference rule $s_0 \leftarrow s_1, \ldots, s_m \in \mathcal{R}$ and either $m = 0$ and the only child node of N holds τ or $m > 0$ and N has m children holding $s_1, \ldots, s_m$;
- $Asms$ is the set of all assumptions held by leaf nodes.

We sometimes name arguments with capital letters, e.g. $A\!: Asms \vdash s$ is an argument with name A. With an abuse of notation, the name of an argument is also used to refer to the whole argument. Note that for every assumption $\alpha \in \mathcal{A}$ there exists an *assumption-argument* $\{\alpha\} \vdash \alpha$.

Let $Asms, Asms_1, Asms_2 \subseteq \mathcal{A}$ be sets of assumptions and let $\alpha \in \mathcal{A}$ be an assumption.

- *Asms attacks* α if and only if there exists an argument $Asms' \vdash \overline{\alpha}$ such that $Asms' \subseteq Asms$. Equivalently, we say that α is attacked by $Asms$.
- $Asms_1$ *attacks* $Asms_2$ if and only if $Asms_1$ attacks some $\alpha \in Asms_2$.
- $Asms^+ = \{\alpha \in \mathcal{A} \mid Asms \text{ attacks } \alpha\}$.
- $Asms$ is *conflict-free* if and only if $Asms \cap Asms^+ = \emptyset$.
- *Asms defends* α if and only if $Asms$ attacks all sets of assumptions attacking α.

Example 2.3. Let ABA_1 be the following (flat) ABA framework:

$$\mathcal{L} = \{p, q, x, \psi, \chi\},$$
$$\mathcal{R} = \{q \leftarrow\ ;\ p \leftarrow q, \chi\ ;\ x \leftarrow p, \psi\},$$
$$\mathcal{A} = \{\psi, \chi\},$$
$$\overline{\psi} = p,\ \overline{\chi} = x.$$

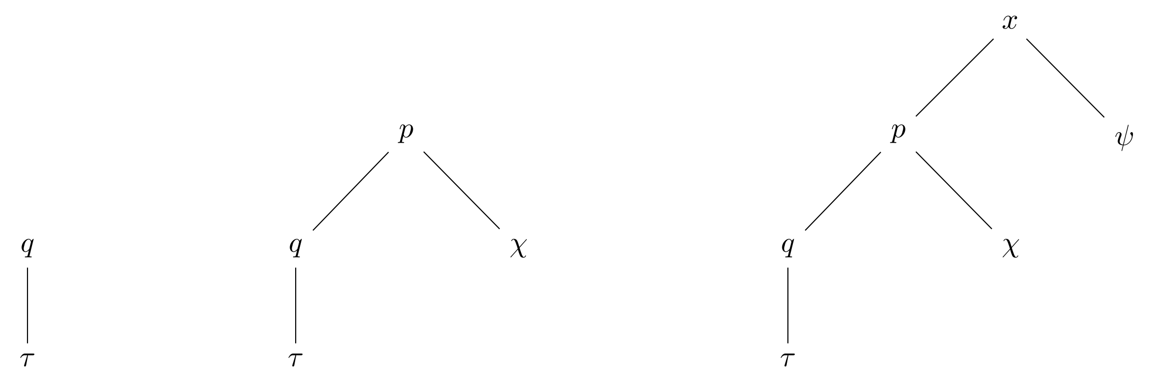


Figure 2.2: The arguments $\{\} \vdash q$, $\{\chi\} \vdash p$, and $\{\chi, \psi\} \vdash x$ (left to right) constructible in ABA_1 (see Example 2.3).

The non-assumption-arguments constructible in ABA_1 are illustrated in Figure 2.2. In addition, there are two assumption-arguments, $\{\chi\} \vdash \chi$ and $\{\psi\} \vdash \psi$, which both consist of only a single node, namely χ and ψ, respectively.
The set of assumptions $\{\chi\}$ attacks assumption ψ since there exists an argument $\{\chi\} \vdash p$, and $\{\chi, \psi\}$ attacks both ψ and χ. The sets of assumptions $\{\psi\}$ and $\{\}$ do not attack any assumption.

The semantics of an ABA framework are defined as *assumption extensions*, i.e. sets of accepted assumptions [BDKT97, DMT07, CSAD15a]. A set of assumptions $Asms \subseteq \mathcal{A}$ is

- an *admissible assumption extension* if and only if $Asms$ is conflict-free and defends every $\alpha \in Asms$;
- a *complete assumption extension* if and only if $Asms$ is conflict-free and consists of all assumptions it defends;
- a *grounded assumption extension* if and only if $Asms$ is a minimal (w.r.t. $\subseteq$) complete assumption extension;
- a *preferred assumption extension* if and only if $Asms$ is a maximal (w.r.t. $\subseteq$) complete assumption extension;
- an *ideal assumption extension* if and only if $Asms$ is a maximal (w.r.t. $\subseteq$) complete assumption extension satisfying that for all preferred assumption extensions $Asms'$, $Asms \subseteq Asms'$;
- a *semi-stable assumption extension* if and only if $Asms$ is a complete assumption extension and for all complete assumption extensions $Asms'$, $Asms \cup Asms^+ \not\subset Asms' \cup Asms'^+$;
- a *stable assumption extension* if and only if $Asms$ is a complete assumption extension and $Asms \cup Asms^+ = \mathcal{A}$.

Note that some of these definitions are not the original ones introduced in [BDKT97, DMT07] but are equivalent formulations as proven in [CSAD15a].

Example 2.4. Consider again ABA_1 from Example 2.3. The only admissible assumption extensions of ABA_1 are $\{\chi\}$, as it defends χ against the attacking set of assumptions $\{\chi, \psi\}$, and $\{\}$, which trivially defends all its assumptions against attackers (as there are none). Both $\{\}$ and $\{\chi\}$ are complete assumption extensions, $\{\chi\}$ is the only preferred, ideal, semi-stable, and stable assumption extension, and $\{\}$ is the unique grounded assumption extension.

2.2.3 Correspondence between ABA and AA

A flat ABA framework $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^- \rangle$ can be mapped into a *corresponding AA framework* $\langle Ar_{ABA}, Att_{ABA} \rangle$ [DMT07] where:

- Ar_{ABA} is the set of all arguments $Asms \vdash s$;
- $(Asms_1 \vdash s_1,\ Asms_2 \vdash s_2) \in Att_{ABA}$ if and only if $\exists \alpha \in Asms_2$ such that $s_1 = \overline{\alpha}$.

Given an admissible / complete / grounded / preferred / ideal / stable assumption extension $Asms$ of $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^- \rangle$, the set of all arguments whose premises are a subset of $Asms$ is an admissible / complete / grounded / preferred / ideal / stable argument extension of $\langle Ar_{ABA}, Att_{ABA} \rangle$ [DMT07, Ton12, CSAD15a].

Conversely, given an admissible / complete / grounded / preferred / ideal / stable argument extension $Args$ of $\langle Ar_{ABA}, Att_{ABA} \rangle$, the union of all premises of arguments in $Args$ is an admissible / complete / grounded / preferred / ideal / stable assumption extension of $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^- \rangle$ [DMT07, Ton12, CSAD15a].

Note that this correspondence does not hold for semi-stable assumption and argument extensions [CSAD15a].

2.3 Logic Programming

Logic programming is a large field of research in AI, comprising various sub-fields such as inductive logic programming, constraint logic programming, and answer set programming. For an overview of the development of the field and its sub-fields, see for example see [DP10].

Many different semantics for logic programs have been proposed and studied, and the language of logic programs has been extended in various ways to incorporate, for example, preferences and aggregation (see [BET11, Fab13] for an overview). Here, we restrict ourselves to logic programs without language extensions, which may however comprise two different types of negation, namely negation-as-failure (NAF) and explicit negation. Concerning the semantics, we focus on the answer set semantics as well as the 3-valued model semantics, as outlined in the following.

2.3.1 Logic Programs

A *logic program* $\mathcal{P}$ is a (finite) set of ground clauses[3] of the form

$$l_0 \leftarrow l_1, \ldots, l_m, \texttt{not } l_{m+1}, \ldots, \texttt{not } l_{m+n}$$

with $m, n \geq 0$, where all l_i $(1 \leq i \leq m)$ and all l_j $(m+1 \leq j \leq m+n)$ are classical literals, i.e. atoms a or *explicitly negated atoms* $\neg a$, and $\texttt{not } l_j$ are *negation-as-failure* (NAF) literals. The classical literal l_0 on the left-hand side of the arrow is referred to as the clause's *head*, all literals on the right of the arrow form the *body* of the clause. If the body of a clause is empty, the clause is called a *fact*.

We will use the following notion of dependency inspired by [YY94]:

- l_0 is *positively dependent* on l_i and
- l_0 is *negatively dependent* on l_j.

A *dependency path* is a chain of positively or negatively dependent literals. A *negative dependency path* is obtained from a dependency path by deleting all literals l in the path such that some k in the path is positively dependent on l, e.g. if p, q, r is a dependency path where p is positively dependent on q and q is negatively dependent on r, then p, r is a negative dependency path. A *negative dependency cycle* is a negative dependency path $l_0, \ldots, l_n$ with $l_0 = l_n$. It is an *odd-length* cycle if n is odd, and an *even-length* cycle otherwise.

Example 2.5. Let $\mathcal{P}_1$ be the following logic program:

$$\begin{aligned} \{\ p &\leftarrow \neg q, \texttt{not } x; \\ x &\leftarrow \texttt{not } p; \\ \neg q &\leftarrow \texttt{not } p\ \} \end{aligned}$$

There exists both an odd- and an even-length negative dependency cycle: p, p is odd, and p, x, p is even.

$\mathcal{HB}_\mathcal{P}$ denotes the *Herbrand Base* of $\mathcal{P}$, i.e. the set of all ground atoms of $\mathcal{P}$, and $Lit_\mathcal{P} = \mathcal{HB}_\mathcal{P} \cup \{\neg a \mid a \in \mathcal{HB}_\mathcal{P}\}$ consists of all classical literals of $\mathcal{P}$. $NAF_{\mathcal{HB}_\mathcal{P}} = \{\texttt{not } a \mid a \in \mathcal{HB}_\mathcal{P}\}$ consists of all NAF literals of atoms of $\mathcal{P}$ and $NAF_{Lit_\mathcal{P}} = \{\texttt{not } l \mid l \in Lit_\mathcal{P}\}$ of all NAF literals of classical literals of $\mathcal{P}$.

An atom a and the explicitly negated atom $\neg a$ are called *complementary literals*. l is the *corresponding classical literal* of a NAF literal $\texttt{not } l$, and conversely $\texttt{not } l$ is the *corresponding NAF literal* of the classical literal l. We will use the letter k for a literal in general, i.e. a classical literal l or a NAF literal $\texttt{not } l$. $\sim k$ denotes the *corresponding literal* of k, i.e. if k is a classical literal l, then $\sim k = \texttt{not } l$, and if k is a NAF literal $\texttt{not } l$, then $\sim k = l$. For a set of literals S, $\sim S = \{\sim k \mid k \in S\}$.

[3]Clauses containing variables are used as shorthand for all their ground instances over the Herbrand Universe of the logic program.

$\vdash_{MP}$ denotes derivability using *modus ponens* on $\leftarrow$ as the only inference rule, treating $l \leftarrow$ as $l \leftarrow true$, where $\mathcal{P} \vdash_{MP} true$ for any $\mathcal{P}$. For a logic program $\mathcal{P}$ and $\Delta \subseteq NAF_{Lit_{\mathcal{P}}}$, $\mathcal{P} \cup \Delta$ denotes $\mathcal{P} \cup \{\texttt{not}\ l \leftarrow \mid \texttt{not}\ l \in \Delta\}$. When used on such $\mathcal{P} \cup \Delta$, $\vdash_{MP}$ treats NAF literals syntactically as in [EK89].

A classical literal $l \in Lit_{\mathcal{P}}$ is *strictly derivable* from $\mathcal{P}$ if and only if $\mathcal{P} \vdash_{MP} l$, and *defeasibly derivable* from $\mathcal{P}$ if and only if $\mathcal{P} \nvdash_{MP} l$ and $\exists \Delta \subseteq NAF_{Lit_{\mathcal{P}}}$ such that $\mathcal{P} \cup \Delta \vdash_{MP} l$. l is *derivable* from $\mathcal{P}$ if and only if l is strictly or defeasibly derivable from $\mathcal{P}$.

2.3.2 Answer Set Semantics

In the following, we recall the concept of answer sets as introduced by Gelfond and Lifschitz [GL91]. Let $\mathcal{P}$ be a logic program without NAF literals. The *answer set* of $\mathcal{P}$, denoted $\mathcal{AS}(\mathcal{P})$, is the smallest set $S \subseteq Lit_{\mathcal{P}}$ such that:

1. for any clause $l_0 \leftarrow l_1, \ldots, l_m$ in $\mathcal{P}$ it holds that if $l_1, \ldots, l_m \in S$, then $l_0 \in S$;

2. $S = Lit_{\mathcal{P}}$ if S contains complementary literals.

For a logic program $\mathcal{P}$, possibly containing NAF literals, and any $S \subseteq Lit_{\mathcal{P}}$, the *reduct* $\mathcal{P}^S$ is obtained from $\mathcal{P}$ by deleting

1. all clauses containing $\texttt{not}\ l$ where $l \in S$, and

2. all NAF literals in the remaining clauses.

Then S is an *answer set* of $\mathcal{P}$ if and only if it is the answer set of the reduct $\mathcal{P}^S$, i.e. $S = \mathcal{AS}(\mathcal{P}^S)$. $\mathcal{P}$ is *inconsistent* if it has no answer sets or if its only answer set is $Lit_{\mathcal{P}}$, else it is *consistent*.

Example 2.6. The logic program $\mathcal{P}_1$ from Example 2.5 has a single answer set, namely $\{\neg q, x\}$.

2.3.3 3-Valued Semantics for Logic Programs without Explicit Negation

We now recall the definition of 3-valued models for logic programs without explicit negation [Prz90, Prz91b]. Let $\mathcal{P}$ be a logic program with no explicitly negated atoms. A *3-valued interpretation* of $\mathcal{P}$ is a pair $\langle \mathcal{T}, \mathcal{F} \rangle$, where $\mathcal{T}, \mathcal{F} \subseteq \mathcal{HB}_{\mathcal{P}}$, $\mathcal{T} \cap \mathcal{F} = \emptyset$, and $\mathcal{U} = \mathcal{HB}_{\mathcal{P}} \setminus (\mathcal{T} \cup \mathcal{F})$. The *truth value* of $a \in \mathcal{HB}_{\mathcal{P}}$ and $\texttt{not}\ a \in NAF_{\mathcal{HB}_{\mathcal{P}}}$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ is:

- $val(a) = \texttt{T}$ if $a \in \mathcal{T}$; $\quad val(\texttt{not}\ a) = \texttt{T}$ if $a \in \mathcal{F}$;

- $val(a) = \texttt{F}$ if $a \in \mathcal{F}$; $\quad val(\texttt{not}\ a) = \texttt{F}$ if $a \in \mathcal{T}$;

- $val(a) = \texttt{U}$ if $a \in \mathcal{U}$; $\quad val(\texttt{not}\ a) = \texttt{U}$ if $a \in \mathcal{U}$.

The truth values are ordered by $\texttt{T} > \texttt{U} > \texttt{F}$ and naturally $val(\texttt{T}) = \texttt{T}$, $val(\texttt{F}) = \texttt{F}$, and $val(\texttt{U}) = \texttt{U}$.

A 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ *satisfies* a clause $a_0 \leftarrow a_1, \ldots, a_m, \texttt{not}\ a_{m+1}, \ldots, \texttt{not}\ a_{m+n}$ if and only if $val(a_0) \geq min\{val(a_1), \ldots, val(\texttt{not}\ a_{m+n})\}$. $\langle \mathcal{T}, \mathcal{F} \rangle$ *satisfies* $a_0 \leftarrow$ if and only if $val(a_0) = \texttt{T}$.

The *partial reduct* $\frac{\mathcal{P}}{\langle \mathcal{T}, \mathcal{F} \rangle}$ of $\mathcal{P}$ w.r.t. a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ is obtained by replacing each NAF literal in every clause of $\mathcal{P}$ by its respective truth value.

- A 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is a *3-valued model* of $\mathcal{P}$ if and only if $\langle \mathcal{T}, \mathcal{F} \rangle$ satisfies every clause in $\mathcal{P}$.

- A 3-valued model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is a *3-valued stable model* of $\mathcal{P}$ if and only if it is a 3-valued model of $\frac{\mathcal{P}}{\langle \mathcal{T}, \mathcal{F} \rangle}$ and $\nexists \langle \mathcal{T}_1, \mathcal{F}_1 \rangle$ that is a 3-valued model of $\frac{\mathcal{P}}{\langle \mathcal{T}, \mathcal{F} \rangle}$ such that $\mathcal{T}_1 \subseteq \mathcal{T}$ and $\mathcal{F}_1 \supseteq \mathcal{F}$ and $\mathcal{T} \neq \mathcal{T}_1$ or $\mathcal{F} \neq \mathcal{F}_1$.

- A 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is the *well-founded model* of $\mathcal{P}$ if and only if $\nexists \langle \mathcal{T}_1, \mathcal{F}_1 \rangle$ that is a 3-valued stable model of $\mathcal{P}$ such that $\mathcal{U} \subseteq \mathcal{U}_1$.[4]

- A 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is a *3-valued M-stable model* (Maximal stable) of $\mathcal{P}$ if and only if $\nexists \langle \mathcal{T}_1, \mathcal{F}_1 \rangle$ that is a 3-valued stable model of $\mathcal{P}$ such that $\mathcal{T} \subseteq \mathcal{T}_1$ and $\mathcal{F} \subseteq \mathcal{F}_1$ and $\mathcal{T} \neq \mathcal{T}_1$ or $\mathcal{F} \neq \mathcal{F}_1$.

- A 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is a *3-valued L-stable model* (Least-undefined stable) of $\mathcal{P}$ if and only if $\nexists \langle \mathcal{T}_1, \mathcal{F}_1 \rangle$ that is a 3-valued stable model of $\mathcal{P}$ such that $\mathcal{U}_1 \subset \mathcal{U}$.

- A 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is a *(2-valued) stable model* of $\mathcal{P}$ if and only if $\mathcal{U} = \emptyset$.

- A 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ is an *ideal model* of $\mathcal{P}$ if and only if $\mathcal{T}$ is maximal (w.r.t. $\subseteq$) among all 3-valued stable models satisfying that for all 3-valued M-stable models $\langle \mathcal{T}_M, \mathcal{F}_M \rangle$, $\mathcal{T} \subseteq \mathcal{T}_M$.

Note that 3-valued stable models as defined here are sometimes called "partial stable" models [Prz91b]. Furthermore, this definition of 3-valued stable models coincides with the definition of *partial stable models* based on unfounded sets [SZ91]. Note that Saccà and Zaniolo [SZ90] call *maximal* partial stable models based on unfounded sets "partial stable" models, but later rename them to "M-stable models" [Sac95] (maximal partial stable models). Saccà [Sac95] also introduces "L-stable" models in terms of partial stable models based on unfounded sets. These coincide with our notions of "3-valued M-stable models" and "3-valued L-stable models" as used by Eiter et al. [ELS97]. 3-valued M-stable models have furthermore been shown [KM92, YY95] to coincide with preferred extensions [Dun91], regular models [YY90], and maximal stable classes [BS92]. The notion of well-founded model used here furthermore coincides with the original definition of well-founded

[4]Przymusinski [Prz90] defines a 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ to be the well-founded model of $\mathcal{P}$ if and only if $\nexists \langle \mathcal{T}_1, \mathcal{F}_1 \rangle$ that is a 3-valued stable model of $\mathcal{P}$ such that $\mathcal{T}_1 \subseteq \mathcal{T}$ and $\mathcal{F}_1 \subseteq \mathcal{F}$, but notes that this is equivalent to $\mathcal{U} \subseteq \mathcal{U}_1$.

model by Van Gelder et al. [VRS88, VRS91] as proven in [Prz90]. In addition, the definition of 2-valued stable models in terms of 3-valued stable models [Prz90] coincides with the original definition of 2-valued stable models [GL88].

Some authors express 3-valued interpretations as a single set $\mathcal{M}$ containing both atoms and NAF literals (see e.g. [ELS97]). Such sets correspond to the tuple-notation of 3-valued interpretations as follows: $\mathcal{M}$ corresponds to $\langle \mathcal{T}, \mathcal{F} \rangle$ if and only if $\mathcal{T} = \mathcal{M} \cap \mathcal{HB}_{\mathcal{P}}$, $\mathcal{F} = \sim(\mathcal{M} \cap NAF_{\mathcal{HB}_{\mathcal{P}}})$, and $\mathcal{M} = \mathcal{T} \cup \mathcal{F}$.

2.3.4 3-Valued Semantics for Logic Programs with Explicit Negation

The *translated logic program* $\mathcal{P}'$ of a logic program $\mathcal{P}$, possibly containing explicitly negated atoms, is obtained by substituting every explicitly negated atom $\neg a$ in $\mathcal{P}$ with a new atom $a' \notin \mathcal{HB}_{\mathcal{P}}$ [GL91, Prz90]. We call a' the *translated literal* of the *original literal* $\neg a$. With an abuse of terminology, we sometimes refer to a as the translated literal of atom a. For a 3-valued stable model $\langle \mathcal{T}', \mathcal{F}' \rangle$ of $\mathcal{P}'$, the *corresponding model* $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\langle \mathcal{T}', \mathcal{F}' \rangle$ is obtained by replacing each translated literal in $\langle \mathcal{T}', \mathcal{F}' \rangle$ by its original literal.

Then $\langle \mathcal{T}, \mathcal{F} \rangle$ is a *3-valued stable model* of $\mathcal{P}$ if and only if $\langle \mathcal{T}', \mathcal{F}' \rangle$ is a 3-valued stable model of $\mathcal{P}'$, $\langle \mathcal{T}, \mathcal{F} \rangle$ is the corresponding model of $\langle \mathcal{T}', \mathcal{F}' \rangle$, and $\mathcal{T}$ does not contain complementary literals [Prz90].

The well-founded / 3-valued M-stable / 3-valued L-stable / (2-valued) stable / ideal models of $\mathcal{P}$ are defined analogously, i.e. they are those corresponding models of well-founded / 3-valued M-stable / 3-valued L-stable / (2-valued) stable / ideal models of $\mathcal{P}'$ where $\mathcal{T}$ does not contain complementary literals.

Example 2.7. In the translated logic program $\mathcal{P}_1'$ of $\mathcal{P}_1$, the literal $\neg q$ is replaced by q'. $\mathcal{P}_1'$ has two 3-valued stable models: $\langle \{q', x\}, \{p\} \rangle$ and $\langle \{\}, \{\} \rangle$. Since $\mathcal{T}$ of the corresponding models does not contain complementary literals, $\mathcal{P}_1$ has two 3-valued stable models: $\langle \{\neg q, x\}, \{p\} \rangle$ and $\langle \{\}, \{\} \rangle$.

Note that $\mathcal{P}'$ always has a 3-valued stable, and thus a well-founded, 3-valued M-stable, and 3-valued L-stable, model but $\mathcal{P}$ might not. Furthermore, $\langle \mathcal{T}, \mathcal{F} \rangle$ is a (2-valued) stable model of $\mathcal{P}$ if and only if $\mathcal{T} \neq Lit_{\mathcal{P}}$ is an answer set of $\mathcal{P}$ [GL91, Prz90].

2.4 Summary

In this chapter, we presented the background on AA and ABA frameworks as well as on logic programs used throughout this thesis.

Chapter 6 relies only on the background on AA frameworks from Section 2.2.1 and Chapter 3 only makes use of concepts regarding ABA and AA frameworks, presented in Section 2.2. All other chapters apply both concepts from argumentation and logic programming presented in this chapter.

Chapter 3

Labellings for Assumption-Based and Abstract Argumentation

3.1 Introduction

As introduced in Section 2.2.1, the semantics of AA frameworks can be expressed in terms of either argument extensions or labellings. Argument labellings have the advantage over argument extensions that they do not only distinguish between accepted and non-accepted arguments, but further divide the non-accepted arguments into rejected and undecided ones. Since argument labellings and extensions correspond [CG09, BCG11], argument labellings can also be used to characterise the semantics of a flat ABA framework in terms of its corresponding AA framework. In this chapter, we transfer the idea of argument labellings to assumptions, yielding a new characterisation of the semantics of ABA frameworks. In contrast to argument labellings, which label whole arguments, *assumption labellings* label each assumption as IN (accepted), OUT (rejected), or UNDEC (undecided). Assumption labellings have the advantage over assumption extensions that rejected (OUT) assumptions and assumptions that are neither accepted nor rejected (UNDEC) are distinguished. This distinction can be important in applications such as decision making. Undecided assumptions can for example provide an indication that further information from an expert is required in order to make a definite decision about their acceptability.

We propose assumption labellings for all semantics defined for flat ABA frameworks, i.e. admissible, grounded, complete, preferred, ideal, semi-stable, and stable semantics, and prove that there is a one-to-one correspondence between the respective assumption labellings and extensions. We also investigate the relation between assumption labellings of flat ABA frameworks and argument labellings of the corresponding AA frameworks, showing a one-to-one correspondence for the grounded, complete, preferred, ideal, and stable semantics. These results extend existing work on the correspondence between the semantics of flat ABA frameworks and AA frameworks, as illustrated in Figure 3.1. Since semi-stable argument and assumption extensions do not correspond [CSAD15a], it is unsurprising that the respective labellings do not correspond either, as shown in Figure 3.2. Concerning the admissible semantics we prove a one-to-many correspondence between assumption and argument labellings. Based on this dissimilarity, we introduce a variant of admissible argument labellings for AA frameworks, called *committed admissible argument labellings*, which correspond more closely to admissible assumption labellings than the original admissible argument labellings, as illustrated in Figure 3.3. We furthermore introduce labellings for possibly non-flat ABA frameworks, and prove correspondence with the extension semantics for possibly non-flat ABA frameworks, as shown in Figure 3.4.

The chapter is organised as follows. In Section 3.2, we introduce assumption labellings for the different semantics of flat ABA frameworks and prove their correspondence with assumption extensions of flat ABA frameworks. In Section 3.3, we simplify the definition of assumption labellings for flat ABA frameworks by considering only certain sets of assumptions as attackers of assumptions. We furthermore introduce a graphical representation of flat ABA frameworks and illustrate how assumption labellings can be easily determined and represented using these graphs. In Section 3.4, we investigate the correspondence

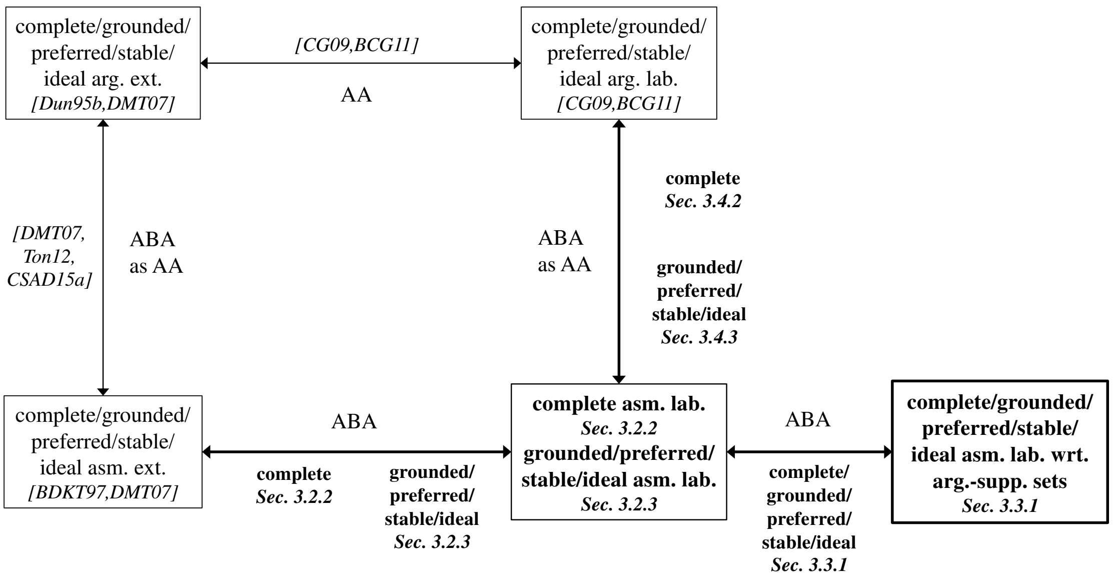

complete/grounded/ preferred/stable/ ideal arg. ext. [Dun95b,DMT07]
[CG09,BCG11]
AA
complete/grounded/ preferred/stable/ ideal arg. lab. [CG09,BCG11]
[DMT07, Ton12, CSAD15a]
ABA as AA
ABA as AA
complete Sec. 3.4.2
grounded/ preferred/ stable/ideal Sec. 3.4.3
complete/grounded/ preferred/stable/ ideal asm. ext. [BDKT97,DMT07]
ABA
complete Sec. 3.2.2
grounded/ preferred/ stable/ideal Sec. 3.2.3
complete asm. lab. Sec. 3.2.2 grounded/preferred/ stable/ideal asm. lab. Sec. 3.2.3
ABA
complete/ grounded/ preferred/ stable/ideal Sec. 3.3.1
complete/grounded/ preferred/stable/ ideal asm. lab. wrt. arg.-supp. sets Sec. 3.3.1

Figure 3.1: A summary of results concerning the complete, grounded, preferred, stable, and ideal semantics in this chapter, where applicable, in the context of previous work. Bidirectional arrows indicate semantic correspondence, and bold indicates novel work presented in this chapter.

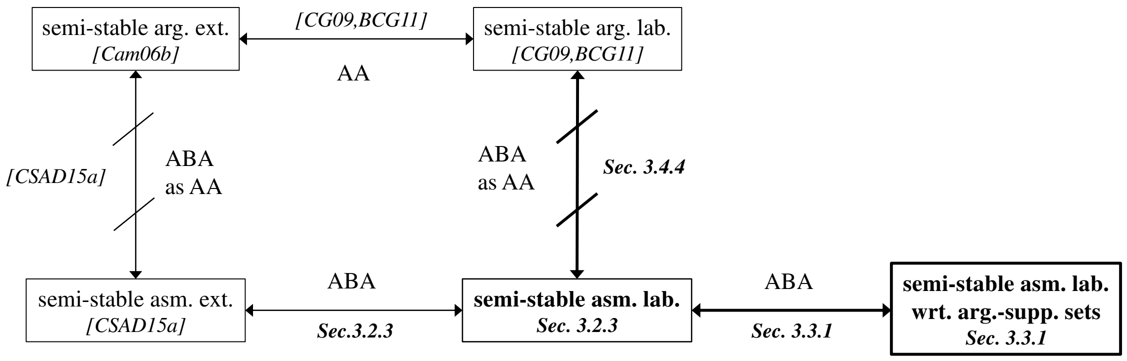

semi-stable arg. ext. [Cam06b]
[CG09,BCG11]
AA
semi-stable arg. lab. [CG09,BCG11]
[CSAD15a]
ABA as AA
ABA as AA
Sec. 3.4.4
semi-stable asm. ext. [CSAD15a]
ABA
Sec.3.2.3
semi-stable asm. lab. Sec. 3.2.3
ABA
Sec. 3.3.1
semi-stable asm. lab. wrt. arg.-supp. sets Sec. 3.3.1

Figure 3.2: Results concerning the semi-stable semantics in this chapter, where applicable, in the context of previous work. Bidirectional arrows indicate semantic correspondence, crossed out arrows denote non-correspondence, and bold indicates novel work in this chapter.

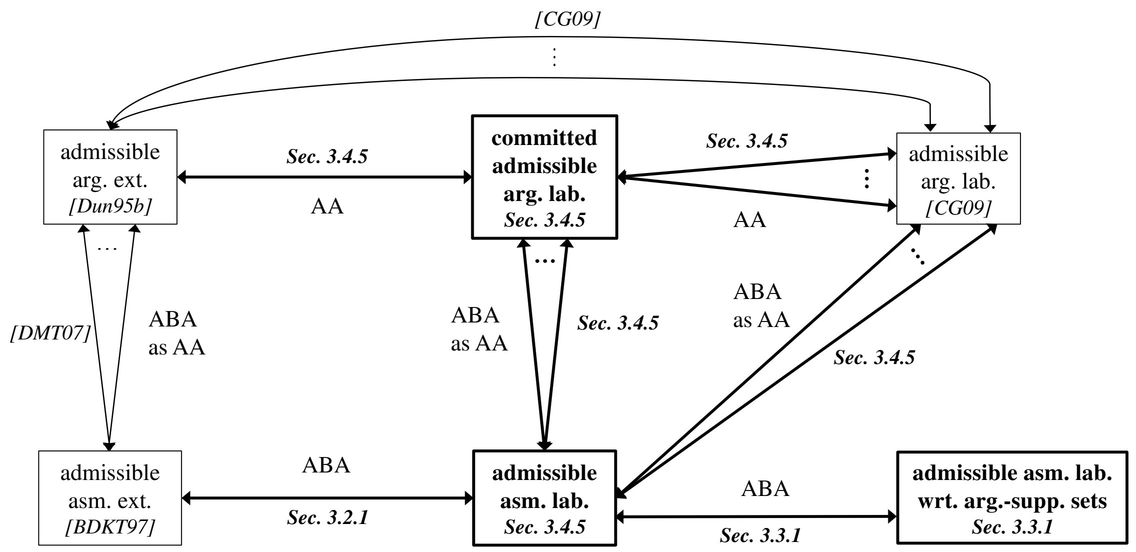

[CG09]
admissible arg. ext. [Dun95b]
Sec. 3.4.5
AA
committed admissible arg. lab. Sec. 3.4.5
Sec. 3.4.5
AA
admissible arg. lab. [CG09]
[DMT07]
ABA as AA
ABA as AA
Sec. 3.4.5
ABA as AA
Sec. 3.4.5
admissible asm. ext. [BDKT97]
ABA
Sec. 3.2.1
admissible asm. lab. Sec. 3.4.5
ABA
Sec. 3.3.1
admissible asm. lab. wrt. arg.-supp. sets Sec. 3.3.1

Figure 3.3: Results concerning the admissible semantics in this chapter, where applicable, in the context of previous work. Bidirectional arrows indicate semantic correspondence (arrows with the same starting point but different end points indicate one-to-many correspondence), and bold indicates novel work in this chapter.

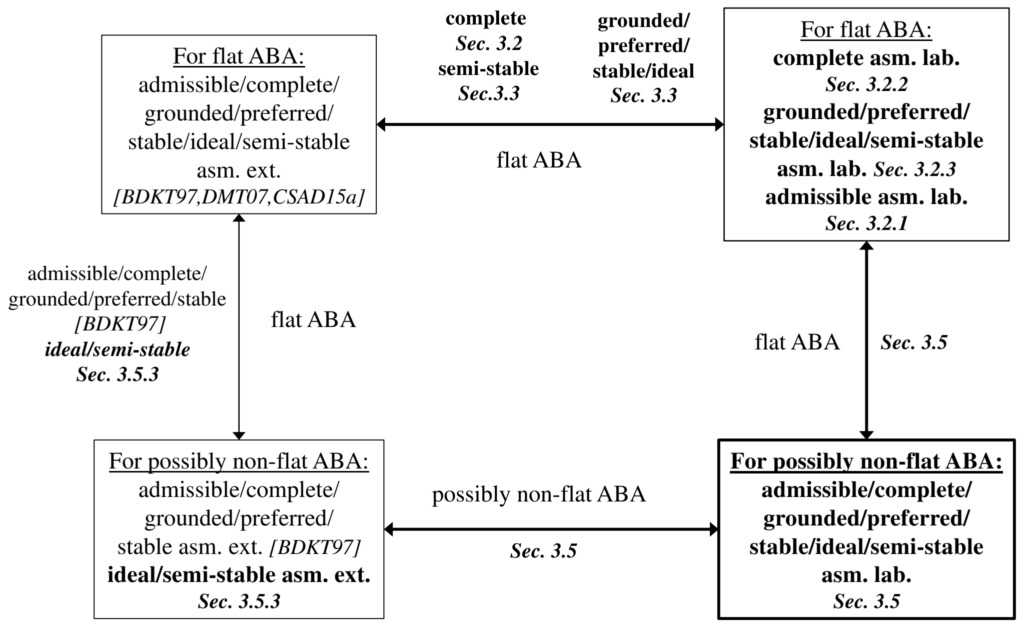

For flat ABA: admissible/complete/ grounded/preferred/ stable/ideal/semi-stable asm. ext. [BDKT97,DMT07,CSAD15a]
complete Sec. 3.2 semi-stable Sec.3.3
grounded/ preferred/ stable/ideal Sec. 3.3
flat ABA
For flat ABA: complete asm. lab. Sec. 3.2.2 grounded/preferred/ stable/ideal/semi-stable asm. lab. Sec. 3.2.3 admissible asm. lab. Sec. 3.2.1
admissible/complete/ grounded/preferred/stable [BDKT97] ideal/semi-stable Sec. 3.5.3
flat ABA
flat ABA
Sec. 3.5
For possibly non-flat ABA: admissible/complete/ grounded/preferred/ stable asm. ext. [BDKT97] ideal/semi-stable asm. ext. Sec. 3.5.3
possibly non-flat ABA
Sec. 3.5
For possibly non-flat ABA: admissible/complete/ grounded/preferred/ stable/ideal/semi-stable asm. lab. Sec. 3.5

Figure 3.4: Results for any (possibly non-flat) ABA framework in this chapter, where applicable, in the context of previous work. Bidirectional arrows indicate semantic correspondence, and bold indicates novel work in this chapter.

between assumption labellings of flat ABA frameworks and argument labellings of their corresponding AA frameworks, and introduce committed admissible argument labellings as a variant of admissible argument labellings for AA frameworks. In Section 3.5, we extend the definition of assumption labellings to possibly non-flat ABA frameworks. We discuss related work in Section 3.6 and summarise the contributions of this chapter in Section 3.7.

3.2 Assumption Labellings

From here onwards, and if not stated otherwise, we assume as given a flat ABA framework $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^- \rangle$. We first introduce labellings for ABA frameworks, which assign a label to each assumption. The three labels used throughout this chapter are $\textsc{in}$, indicating that an assumption is accepted, $\textsc{out}$, indicating that an assumption is rejected, and $\textsc{undec}$, indicating that an assumption is neither accepted nor rejected and thus undecided.

Definition 3.1 (Assumption Labelling)**.** An *assumption labelling* is a total function $\mathit{LabAsm} : \mathcal{A} \rightarrow \{\textsc{in}, \textsc{out}, \textsc{undec}\}$.

If $\mathit{LabAsm}(\alpha) = \textsc{in}$, we say that α *is labelled* $\textsc{in}$ by LabAsm, or equivalently that LabAsm labels α (as) $\textsc{in}$. Analogous terminology is used for assumptions labelled $\textsc{out}$ and $\textsc{undec}$. The set of all assumptions labelled $\textsc{in}$ by LabAsm is $\textsc{in}(\mathit{LabAsm}) = \{\alpha \in \mathcal{A} \mid \mathit{LabAsm}(\alpha) = \textsc{in}\}$, and the sets of all assumptions labelled $\textsc{out}$ and $\textsc{undec}$ are denoted $\textsc{out}(\mathit{LabAsm})$ and $\textsc{undec}(\mathit{LabAsm})$, respectively.

3.2.1 Admissible Semantics

An admissible assumption *extension* is a set of accepted assumptions which is able to defend itself. In other words, if an assumption α is contained in an admissible assumption extension, then all sets of assumptions attacking α contain some assumption attacked by this admissible assumption extension. In an *admissible assumption labelling* the concept of defence is mirrored by requiring that if an assumption α is accepted (labelled $\textsc{in}$), then all sets of assumptions attacking α contain a rejected assumption (labelled $\textsc{out}$), which in turn is attacked by a set of accepted assumptions (all labelled $\textsc{in}$). In addition, we require that an undecided assumption (labelled $\textsc{undec}$) is not attacked by a set of accepted assumptions (all labelled $\textsc{in}$), since an assumption attacked by accepted assumptions can clearly not be accepted (due to the conflict-freeness property of the admissible semantics) and should thus be rejected rather than undecided.

Definition 3.2 (Admissible Assumption Labelling)**.** Let LabAsm be an assumption labelling. LabAsm is an *admissible assumption labelling* if and only if for each assumption $\alpha \in \mathcal{A}$ it holds that:

- if $\mathit{LabAsm}(\alpha) = \textsc{in}$, then for each set of assumptions Asms attacking α there exists some $\beta \in \mathit{Asms}$ such that $\mathit{LabAsm}(\beta) = \textsc{out}$;

- if $LabAsm(\alpha) = \textsc{out}$, then there exists a set of assumptions $Asms$ attacking α such that for all $\beta \in Asms$, $LabAsm(\beta) = \textsc{in}$;
- if $LabAsm(\alpha) = \textsc{undec}$, then for each set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) \neq \textsc{in}$.

Example 3.1. Consider the following ABA framework, which we call ABA_2:

$$\begin{aligned}
&\mathcal{L} = \{r, p, x, \rho, \psi, \chi\},\\
&\mathcal{R} = \{p \leftarrow \rho; x \leftarrow \psi\},\\
&\mathcal{A} = \{\rho, \psi, \chi\},\\
&\overline{\rho} = r \ , \ \overline{\psi} = p \ , \ \overline{\chi} = x.
\end{aligned}$$

ABA_2 has three admissible assumption labellings:

- $LabAsm_1 = \{(\rho, \textsc{undec}), (\psi, \textsc{undec}), (\chi, \textsc{undec})\}$,
- $LabAsm_2 = \{(\rho, \textsc{in}), (\psi, \textsc{out}), (\chi, \textsc{undec})\}$, and
- $LabAsm_3 = \{(\rho, \textsc{in}), (\psi, \textsc{out}), (\chi, \textsc{in})\}$.

These assumption labellings demonstrate two important points: first, an assumption that is not attacked by any set of assumptions (ρ in ABA_2) cannot be labelled OUT; and second, an assumption attacked by a set of assumptions containing only IN-labelled assumptions (ψ in $LabAsm_2$ and $LabAsm_3$) must be labelled OUT.

It is important to note that the *empty set* of assumptions has a special role as an attacking set of assumptions: any assumption attacked by the empty set is labelled OUT by all admissible assumption labellings since an argument supported by the empty set stands for a (non-refutable) fact, so the attacked assumption clearly has to be rejected, as illustrated in Example 3.2.

Example 3.2. Let ABA_3 be ABA_2 from Example 3.1 with the additional sentences ϕ and f in $\mathcal{L}$, where ϕ is an assumption with $\overline{\phi} = f$, and with the additional inference rule $f \leftarrow$.

Since $\{\} \vdash f$ is an argument, ϕ is attacked by the empty set of assumptions as well as by all other sets of assumptions. Thus, ϕ cannot be labelled IN since the attacking empty set does not contain an assumption labelled OUT, and ϕ cannot be labelled UNDEC since the attacking empty set does not contain an assumption not labelled IN. Consequently, ϕ is labelled OUT by all admissible assumption labellings.

ABA_3 has thus three admissible assumption labellings:

- $LabAsm_1 = \{(\phi, \textsc{out}), (\rho, \textsc{undec}), (\psi, \textsc{undec}), (\chi, \textsc{undec})\}$,
- $LabAsm_2 = \{(\phi, \textsc{out}), (\rho, \textsc{in}), (\psi, \textsc{out}), (\chi, \textsc{undec})\}$, and
- $LabAsm_3 = \{(\phi, \textsc{out}), (\rho, \textsc{in}), (\psi, \textsc{out}), (\chi, \textsc{in})\}$.

Note that these are the same admissible assumption labellings as for ABA_2, but with the additional assumption ϕ, which is always labelled $\textsc{out}$. The number of admissible assumption labellings is thus not influenced by assumptions attacked by the empty set since these assumptions do not have alternative labels in different admissible assumption labellings.

The following theorem shows that there is a one-to-one correspondence between the admissible semantics in terms of assumption labellings and extensions.

Theorem 3.1.

1. *Let Asms be an admissible assumption extension. Then LabAsm with* $\textsc{in}(LabAsm) = Asms$, $\textsc{out}(LabAsm) = Asms^+$, *and* $\textsc{undec}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is an admissible assumption labelling.*

2. *Let LabAsm be an admissible assumption labelling. Then* $Asms = \textsc{in}(LabAsm)$ *is an admissible assumption extension with* $Asms^+ = \textsc{out}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \textsc{undec}(LabAsm)$.

Proof.

1. First note that $Asms \cap Asms^+ = \emptyset$ since $Asms$ does not attack itself. Thus each $\alpha \in \mathcal{A}$ is either contained in $\textsc{in}(LabAsm)$, in $\textsc{out}(LabAsm)$, or in $\textsc{undec}(LabAsm)$.

   - Let $LabAsm(\alpha) = \textsc{in}$. Then $\alpha \in Asms$, so $Asms$ defends α, i.e. for all sets of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $Asms$ attacks β. Thus, $\beta \in Asms^+$ and consequently $LabAsm(\beta) = \textsc{out}$.
   - Let $LabAsm(\alpha) = \textsc{out}$. Then $\alpha \in Asms^+$, so $Asms$ attacks α. Since $Asms = \textsc{in}(LabAsm)$, there exists a set of assumptions $Asms_1$ attacking α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$.
   - Let $LabAsm(\alpha) = \textsc{undec}$. Then $\alpha \notin Asms$ and $\alpha \notin Asms^+$, so α is not attacked and not defended by $Asms$. Since α is not attacked by $Asms$, for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $\beta \notin Asms$, and thus $LabAsm(\beta) \neq \textsc{in}$.

2. We first prove that $\textsc{in}(LabAsm)$ is an admissible assumption extension.

   - $\textsc{in}(LabAsm)$ is conflict-free: Assume $\textsc{in}(LabAsm)$ is not conflict-free. Then $\textsc{in}(LabAsm)$ attacks some $\alpha \in \textsc{in}(LabAsm)$. By Definition 3.2, for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) = \textsc{out}$. Hence, $\textsc{in}(LabAsm)$ contains some β such that $LabAsm(\beta) = \textsc{out}$. Contradiction.
   - $\textsc{in}(LabAsm)$ defends all $\alpha \in \textsc{in}(LabAsm)$: Let $\alpha \in \textsc{in}(LabAsm)$. Then by Definition 3.2, for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) = \textsc{out}$. Furthermore, for each such β there

exists a set of assumptions $Asms_2$ attacking β such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) = \textsc{in}$ so $Asms_2 \subseteq \textsc{in}(LabAsm)$. Hence, $\textsc{in}(LabAsm)$ attacks all sets of assumptions attacking α.

- $Asms^+ = \{\alpha \in \mathcal{A} \mid Asms \textit{ attacks } \alpha\} = \{\alpha \in \mathcal{A} \mid \textsc{in}(LabAsm) \textit{ attacks } \alpha\}$ $= \{\alpha \in \mathcal{A} \mid \alpha \in \textsc{out}(LabAsm)\} = \textsc{out}(LabAsm)$.
- $\mathcal{A} \setminus (Asms \cup Asms^+) = \{\alpha \in \mathcal{A} \mid \alpha \notin \textsc{in}(LabAsm), \alpha \notin \textsc{out}(LabAsm)\}$ $= \{\alpha \in \mathcal{A} \mid \alpha \in \textsc{undec}(LabAsm)\} = \textsc{undec}(LabAsm)$.

□

Example 3.3. ABA_3 from Example 3.2 has three admissible assumption extensions: $Asms_1 = \{\}$, $Asms_2 = \{\rho\}$, and $Asms_3 = \{\rho, \chi\}$, corresponding to the three admissible assumption labellings $LabAsm_1$, $LabAsm_2$, and $LabAsm_3$, respectively.

Note that without the third condition in Definition 3.2, the second item in Theorem 3.1 would not hold. For example, $LabAsm_4 = \{(\phi, \textsc{out}), (\rho, \textsc{in}), (\psi, \textsc{undec}), (\chi, \textsc{undec})\}$ would be an admissible assumption labelling of ABA_3 (see Example 3.2), but even though $Asms_4 = \textsc{in}(LabAsm_4) = \{\rho\}$ is an admissible assumption extension of ABA_3, it does not hold that $Asms_4^+ = \textsc{out}(LabAsm_4)$ as stated in the second item of Theorem 3.1 since $\psi \in Asms_4^+$ but $\psi \notin \textsc{out}(LabAsm_4)$.

If an assumption is defended by an admissible assumption extension, then adding this assumption to the extension yields another admissible assumption extension [BDKT97] (similar to the Fundamental Lemma for AA frameworks [Dun95b]). Due to the one-to-one correspondence between admissible assumption labellings and extensions, an analogous property holds for admissible assumption labellings. The following lemma states that if an assumption α is defended by an admissible assumption labelling, i.e. all sets of assumptions $Asms$ attacking α contain an assumption β labelled OUT, then changing the label of α to IN and changing the label of all assumptions γ that now need to be rejected (due to the change of label of α) to OUT yields another admissible assumption labelling.

Lemma 3.2. *Let LabAsm be an admissible assumption labelling and let $\alpha \in \mathcal{A}$ be such that for each set of assumptions Asms attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \textsc{out}$. Let $\alpha^\star = \{\gamma \in \mathcal{A} \mid \exists Asms \subseteq \mathcal{A}$ such that $\alpha \in Asms, Asms$ attacks γ, $\forall \delta \in Asms : \delta \neq \alpha \rightarrow LabAsm(\delta) = \textsc{in}\}$. Then $LabAsm'$ with*

$$\begin{aligned} \textsc{in}(LabAsm') &= \textsc{in}(LabAsm) \cup \{\alpha\}, \\ \textsc{out}(LabAsm') &= \textsc{out}(LabAsm) \cup \alpha^\star, \text{ and} \\ \textsc{undec}(LabAsm') &= \textsc{undec}(LabAsm) \setminus (\{\alpha\} \cup \alpha^\star) \end{aligned}$$

is an admissible assumption labelling.

Proof. Since each set of assumptions attacking α contains some β such that $LabAsm(\beta) = \textsc{out}$, $LabAsm(\alpha) \neq \textsc{out}$. If $LabAsm(\alpha) = \textsc{in}$, then $\forall \gamma \in \alpha^\star : LabAsm(\gamma) = \textsc{out}$ and therefore $LabAsm' = LabAsm$, so trivially $LabAsm'$ is an admissible assumption labelling. If $LabAsm(\alpha) = \text{UNDEC}$, then $\forall\gamma \in \alpha^\star : LabAsm(\gamma) = \text{UNDEC}$ or $LabAsm(\gamma) = \text{OUT}$. Furthermore, $\alpha \notin \alpha^\star$ since each set of assumptions attacking α contains some β such that $LabAsm(\beta) = \text{OUT}$, so if $\alpha \in \alpha^\star$, then $\exists Asms$ attacking α such that $LabAsm(\alpha) = \text{OUT}$ (since all $\forall\delta \in Asms : \delta \neq \alpha \rightarrow LabAsm(\delta) = \text{IN}$), which is a contradiction. Therefore, $LabAsm'$ is an assumption labelling.

- Let $LabAsm'(\epsilon) = \text{IN}$. If $LabAsm(\epsilon) = \text{IN}$, then for each set of assumptions $Asms_1$ attacking ϵ there exists some $\eta \in Asms_1$ such that $LabAsm(\eta) = \text{OUT}$ and therefore $LabAsm'(\eta) = \text{OUT}$. If $LabAsm(\epsilon) \neq \text{IN}$, then $\epsilon = \alpha$, so for each set of assumptions $Asms_2$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \text{OUT}$ and thus $LabAsm'(\beta) = \text{OUT}$.

- Let $LabAsm'(\epsilon) = \text{OUT}$. If $LabAsm(\epsilon) = \text{OUT}$, then there exists a set of assumptions $Asms_1$ attacking ϵ such that for all $\eta \in Asms_1$, $LabAsm(\eta) = \text{IN}$ and therefore $LabAsm'(\eta) = \text{IN}$. If $LabAsm(\epsilon) \neq \text{OUT}$, then $\epsilon \in \alpha^\star$, so there exists a set of assumptions $Asms_2$ attacking ϵ such that $\forall\delta \in Asms_2$ with $\delta \neq \alpha$ it holds that $LabAsm(\delta) = \text{IN}$ and thus $LabAsm'(\delta) = \text{IN}$. Since $LabAsm'(\alpha) = \text{IN}$ the set of assumptions $Asms_2$ attacking ϵ is such that for all $\eta \in Asms_3$, $LabAsm'(\eta) = \text{IN}$.

- Let $LabAsm'(\epsilon) = \text{UNDEC}$. Then $LabAsm(\epsilon) = \text{UNDEC}$. Thus, for each set of assumptions $Asms_1$ attacking ϵ there exists some $\eta \in Asms_1$ such that $LabAsm(\eta) \neq \text{IN}$. Since $\epsilon \notin \alpha^\star$, for each such set of assumptions $Asms_1$ attacking ϵ either $\alpha \notin Asms_1$ or there exists some $\kappa \in Asms_1$ such that $\kappa \neq \alpha$ and $LabAsm(\kappa) \neq \text{IN}$. In the first case $\eta \neq \alpha$, so $LabAsm'(\eta) \neq \text{IN}$. In the second case, $LabAsm'(\kappa) \neq \text{IN}$. Thus, for each set of assumptions $Asms_1$ attacking ϵ there exists some $\lambda \in Asms_1$ such that $LabAsm'(\lambda) \neq \text{IN}$.

□

Example 3.4. Let ABA_4 be the following ABA framework with:

$\mathcal{L} = \{f, p, r, x, \phi, \psi, \rho, \chi\}$,
$\mathcal{R} = \{r \leftarrow \phi, \chi;\ p \leftarrow \rho\}$,
$\mathcal{A} = \{\phi, \psi, \rho, \chi\}$,
$\overline{\phi} = f$, $\overline{\psi} = p$, $\overline{\rho} = r$, $\overline{\chi} = x$.

$LabAsm_1 = \{(\phi, \text{UNDEC}), (\psi, \text{UNDEC}), (\rho, \text{UNDEC}), (\chi, \text{IN})\}$ is an admissible assumption labelling of ABA_4. Since ϕ is not attacked by any set of assumptions, it holds that each set of assumptions attacking ϕ contains an assumption labelled OUT, and $\phi^\star = \{\rho\}$. As stated in Lemma 3.2, $LabAsm_2$ with $\text{IN}(LabAsm_2) = \{\chi, \phi\}$, $\text{OUT}(LabAsm_2) = \{\rho\}$, and $\text{UNDEC}(LabAsm_2) = \{\psi\}$ is an admissible assumption labelling of ABA_4. Since with respect to $LabAsm_2$ it holds that each set of assumptions attacking ψ contains an assumption labelled OUT, $LabAsm_3$ with $\text{IN}(LabAsm_3) = \{\chi, \phi, \psi\}$, $\text{OUT}(LabAsm_3) = \{\rho\}$, and UNDEC($\mathit{LabAsm}_3$) $= \{\}$ is also an admissible assumption labelling of ABA_4 (where $\psi^\star = \{\}$).

3.2.2 Complete Semantics

In addition to defending each of its elements against attackers, a complete assumption extension contains every assumption it defends. This additional condition is mirrored in *complete assumption labellings* by requiring that an assumption that is defended has to be labelled IN. This can be achieved by modifying the definition of admissible assumption labellings in various ways.

In an admissible assumption labelling a defended assumption may be labelled IN or UNDEC. Thus, one way of modifying the definition of admissible assumption labellings is to prohibit labelling defended assumptions as UNDEC. In other words, an assumption labelled UNDEC has to be attacked by at least one set of assumptions that does not contain any assumption labelled OUT.

Definition 3.3 (Complete Assumption Labelling)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is a *complete assumption labelling* if and only if *LabAsm* is an admissible assumption labelling and for each assumption $\alpha \in \mathcal{A}$ it holds that:

- if $\mathit{LabAsm}(\alpha) =$ UNDEC, then there exists a set of assumptions *Asms* attacking α such that for all $\gamma \in \mathit{Asms}$, $\mathit{LabAsm}(\gamma) \neq$ OUT.

Note that the new condition for UNDEC assumptions implies that there exists a set of assumptions *Asms* attacking α such that for some $\gamma \in \mathit{Asms}$, $\mathit{LabAsm}(\gamma) =$ UNDEC, since by the definition of admissible assumption labellings some $\beta \in \mathit{Asms}$ is not labelled IN and by the new condition no $\gamma \in \mathit{Asms}$ is labelled OUT.

Example 3.5. Consider again ABA_2 from Example 3.1 and its three admissible assumption labellings. In LabAsm_1, ρ does not satisfy the new condition for UNDEC assumptions, and in LabAsm_2, χ does not satisfy the new condition. The only admissible assumption labelling satisfying the new condition is LabAsm_3, which is thus the only complete assumption labelling of ABA_2.

The second way to modify the definition of admissible assumption labellings in order to express the complete semantics is to add a condition that explicitly states that if an assumption α is defended, i.e. if all sets of assumptions attacking α contain some assumption labelled OUT, then α has to be labelled IN. This condition adds the “opposite direction” of the first condition of an admissible assumption labelling. To make this way of defining complete assumption labellings more uniform, the “opposite direction” of the second condition of an admissible assumption labelling is added, too. This renders the third condition of an admissible assumption labelling superfluous and thus leaves two “if and only if” conditions to be satisfied by each $\alpha \in \mathcal{A}$:

- *LabAsm*$(\alpha) = $ IN if and only if for each set of assumptions *Asms* attacking α there exists some $\beta \in$ *Asms* such that *LabAsm*$(\beta) = $ OUT;
- *LabAsm*$(\alpha) = $ OUT if and only if there exists a set of assumptions *Asms* attacking α such that for all $\beta \in$ *Asms*, *LabAsm*$(\beta) = $ IN.

Since *LabAsm* is an assumption labelling and thus labels each assumption in an ABA framework, assumptions that do not satisfy the right hand side of either of the above conditions are "automatically" labelled UNDEC by *LabAsm*.

A third way to define complete assumption labellings reverses all three conditions of Definition 3.3, thus specifying which label an assumption satisfying a certain condition should have.

Theorem 3.3. *Let LabAsm be an assumption labelling. The following statements are equivalent:*

1. *LabAsm is a complete assumption labelling.*
2. *LabAsm is such that for each $\alpha \in \mathcal{A}$ it holds that:*
   - *LabAsm$(\alpha) = $* IN *if and only if for each set of assumptions Asms attacking α there exists some $\beta \in$ Asms such that LabAsm$(\beta) = $* OUT*;*
   - *LabAsm$(\alpha) = $* OUT *if and only if there exists a set of assumptions Asms attacking α such that for all $\beta \in$ Asms, LabAsm$(\beta) = $* IN.
3. *LabAsm is such that for each $\alpha \in \mathcal{A}$ it holds that:*
   - *if for each set of assumptions Asms attacking α there exists some $\beta \in$ Asms such that LabAsm$(\beta) = $* OUT*, then LabAsm$(\alpha) = $* IN*;*
   - *if there exists a set of assumptions Asms attacking α such that for all $\beta \in$ Asms, LabAsm$(\beta) = $* IN*, then LabAsm$(\alpha) = $* OUT*;*
   - *if for each set of assumptions Asms$_1$ attacking α there exists some $\beta \in$ Asms$_1$ such that LabAsm$(\beta) \neq$* IN*, and there exists a set of assumptions Asms$_2$ attacking α such that for all $\gamma \in$ Asms$_2$, LabAsm$(\gamma) \neq$* OUT*, then LabAsm$(\alpha) = $* UNDEC.

Proof. Equivalence of first and second item:

- First item implies second item: Let *LabAsm* be a complete assumption labelling. Then clearly the "only if" part of both conditions of the second item are satisfied since they are the same as the conditions in Definition 3.3. To prove that the "if" part of the conditions in the second item holds:
  - Let α be an assumption such that for each set of assumptions *Asms* attacking α there exists some $\beta \in$ *Asms* such that *LabAsm*$(\beta) = $ OUT. Then *LabAsm*$(\alpha) \neq$

OUT because there exists no set of assumptions $Asms_1$ attacking α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$. Furthermore, $LabAsm(\alpha) \neq \textsc{undec}$ because there exists no set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) \neq \textsc{out}$. Hence, $LabAsm(\alpha) = \textsc{in}$.

  – Let α be an assumption such that there exists a set of assumptions $Asms$ attacking α such that for all $\beta \in Asms$, $LabAsm(\beta) = \textsc{in}$. Then $LabAsm(\alpha) \neq \textsc{in}$ because not for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) = \textsc{out}$. Furthermore, $LabAsm(\alpha) \neq \textsc{undec}$ because not for each set of assumptions $Asms_2$ attacking α there exists some $\gamma \in Asms_2$ such that $LabAsm(\gamma) \neq \textsc{in}$. Hence, $LabAsm(\alpha) = \textsc{out}$.

- Second item implies first item: Let $LabAsm$ be such that the second item holds. We prove that $LabAsm$ is a complete assumption labelling. Clearly the first two conditions of complete assumption labellings are satisfied since they are the same as the "only if" part of the conditions in the second item. To prove the third condition of complete assumption labellings, let $LabAsm(\alpha) = \textsc{undec}$. From the first condition of the second item we know that not for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) = \textsc{out}$, so there exists a set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) \neq \textsc{out}$. From the second condition of the second item we know that there exists no set of assumptions $Asms_3$ attacking α such that for all $\delta \in Asms_3$, $LabAsm(\delta) = \textsc{in}$, so for each set of assumptions $Asms_4$ attacking α there exists some $\epsilon \in Asms_4$ such that $LabAsm(\epsilon) \neq \textsc{in}$.

Equivalence of second and third item:

- Second item implies third item: Let $LabAsm$ be such that the second item holds. Then clearly the first two conditions of the third item are satisfied since they are the same as the "if" part of the second item. To prove the third condition of the third item, let α be such that for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) \neq \textsc{in}$, and there exists a set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) \neq \textsc{out}$. Then $LabAsm(\alpha) \neq \textsc{in}$ because not for each set of assumptions $Asms_3$ attacking α there exists some $\delta \in Asms_3$ such that $LabAsm(\delta) = \textsc{out}$, and $LabAsm(\alpha) \neq \textsc{out}$ because there exists no set of assumptions $Asms_4$ attacking α such that for all $\epsilon \in Asms_4$, $LabAsm(\epsilon) = \textsc{in}$. Hence, $LabAsm(\alpha) = \textsc{undec}$.

- Third item implies second item: Assume that $LabAsm$ is such that the third item holds. Then clearly the "if" part of both conditions in the second item are satisfied since they the same as the conditions in the third item. To prove that the "only if" parts of the conditions in the second item are satisfied, first note that for every $\alpha \in \mathcal{A}$ exactly one of the "if" parts of the three conditions in the third item is satisfied. Thus, if $LabAsm(\alpha) = \textsc{in}$, the "if" part of the second and third condition in the third item are not satisfied. It follows that the "if" part of the first condition is satisfied, so for each set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) = \textsc{out}$. Analogously, if $LabAsm(\alpha) = \textsc{out}$, only the "if" part of the second condition in the third item applies, so there exists a set of assumptions $Asms_2$ attacking α such that for all $\beta \in Asms_2$, $LabAsm(\beta) = \textsc{in}$.

□

Example 3.6. Consider again ABA_2 from Example 3.1 and its three admissible assumption labellings. $LabAsm_1$ does not satisfy the second item in Theorem 3.3 since ρ violates the first condition. Similarly, $LabAsm_1$ does not satisfy the third item in Theorem 3.3 since ρ violates the first condition. $LabAsm_2$ does not satisfy the second or third item in Theorem 3.3 since χ violates the first condition of both items. Only $LabAsm_3$ satisfies the second as well as the third item in Theorem 3.3, and is thus the only complete assumption labelling of ABA_2.

All three ways of defining complete assumption labellings are useful in their own rights. Definition 3.3 is particularly suitable to verify whether a given assumption labelling is indeed a complete assumption labelling. In contrast, the third item in Theorem 3.3 is more suitable for determining which assumptions should have which label. Since the second item in Theorem 3.3 can be considered as the "union" of the two other definitions, it lends itself to either of the two tasks.

Note that the definition of admissible assumption labellings cannot be equivalently expressed by reversing the conditions in Definition 3.2 since they are not mutually exclusive. In particular, an unattacked assumption would satisfy both the condition to be labelled IN and to be labelled UNDEC, so not matter which of the two labels was assigned to the assumption, one of the two conditions would be violated.

The following theorem proves that there is a one-to-one correspondence between complete assumption labellings and extensions, just as between admissible assumption labellings and extensions.

Theorem 3.4.

1. *Let Asms be a complete assumption extension. Then LabAsm with* $\textsc{in}(LabAsm) = Asms$*,* $\textsc{out}(LabAsm) = Asms^+$*, and* $\textsc{undec}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is a complete assumption labelling.*

2. *Let LabAsm be a complete assumption labelling. Then* $Asms = \textsc{in}(LabAsm)$ *is a complete assumption extension with* $Asms^+ = \textsc{out}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \textsc{undec}(LabAsm)$*.*

Proof.

1. Since *Asms* is a complete assumption extension it is by definition also an admissible assumption extension. By Theorem 3.1, *LabAsm* is an admissible assumption

labelling. It remains to prove that the additional condition of complete assumption labellings is satisfied. Let $\mathit{LabAsm}(\alpha) = \textsc{undec}$. Then $\alpha \notin \mathit{Asms}$ and $\alpha \notin \mathit{Asms}^+$, so α is not attacked and not defended by *Asms*. Since α is not defended by *Asms*, there exists a set of assumptions Asms_1 attacking α such that Asms_1 is not attacked by *Asms*. Thus, for all $\gamma \in \mathit{Asms}_1$ it holds that $\gamma \notin \mathit{Asms}^+$. Consequently, $\mathit{LabAsm}(\gamma) \neq \textsc{out}$.

2. Since *LabAsm* is a complete assumption labelling it is by Definition 3.3 also an admissible assumption labelling. Thus, by Theorem 3.1 *Asms* is an admissible assumption extension with $\mathit{Asms}^+ = \textsc{out}(\mathit{LabAsm})$ and $\mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+) = \textsc{undec}(\mathit{LabAsm})$. It remains to prove that all assumptions defended by *Asms* are contained in *Asms*. Let α be defended by *Asms* and thus by $\textsc{in}(\mathit{LabAsm})$. Then for each set of assumptions Asms_1 attacking α, $\textsc{in}(\mathit{LabAsm})$ attacks Asms_1. Thus, for each such Asms_1 there exists some $\beta \in \mathit{Asms}_1$ which is attacked by $\textsc{in}(\mathit{LabAsm})$, and therefore $\mathit{LabAsm}(\beta) = \textsc{out}$. Since this holds for each Asms_1 attacking α, $\mathit{LabAsm}(\alpha) = \textsc{in}$.

□

3.2.3 Grounded, Preferred, Ideal, Semi-Stable, and Stable Semantics

Based on the notion of complete assumption labellings, the grounded, preferred, ideal, semi-stable, and stable semantics can be defined in terms of assumption labellings.

Definition 3.4 (Grounded, Preferred, Ideal, Semi-Stable, Stable Assumption Labelling)**.** A complete assumption labelling *LabAsm* is

- a *grounded assumption labelling* if and only if $\textsc{in}(\mathit{LabAsm})$ is minimal (w.r.t. $\subseteq$) among all complete assumption labellings;
- a *preferred assumption labelling* if and only if $\textsc{in}(\mathit{LabAsm})$ is maximal (w.r.t. $\subseteq$) among all complete assumption labellings;
- an *ideal assumption labelling* if and only if $\textsc{in}(\mathit{LabAsm})$ is maximal (w.r.t. $\subseteq$) among all complete assumption labellings satisfying that for all preferred assumption labellings LabAsm', $\textsc{in}(\mathit{LabAsm}) \subseteq \textsc{in}(\mathit{LabAsm}')$;
- a *semi-stable assumption labelling* if and only if $\textsc{undec}(\mathit{LabAsm})$ is minimal (w.r.t. $\subseteq$) among all complete assumption labellings;
- a *stable assumption labelling* if and only if $\textsc{undec}(\mathit{LabAsm}) = \emptyset$.

Example 3.7. Let ABA_5 be the following ABA framework:

$\mathcal{L} = \{r, p, x, \rho, \psi, \chi\}$,
$\mathcal{R} = \{r \leftarrow \psi\ ;\ p \leftarrow \rho\ ;\ p \leftarrow \chi\ ;\ x \leftarrow \psi\ ;\ x \leftarrow \chi\}$,
$\mathcal{A} = \{\rho, \psi, \chi\}$,
$\overline{\rho} = r\ ,\ \overline{\psi} = p\ ,\ \overline{\chi} = x$.

ABA_5 has three complete assumption labellings:

- $LabAsm_1 = \{(\rho, \text{UNDEC}), (\psi, \text{UNDEC}), (\chi, \text{UNDEC})\}$,
- $LabAsm_2 = \{(\rho, \text{OUT}), (\psi, \text{IN}), (\chi, \text{OUT})\}$, and
- $LabAsm_3 = \{(\rho, \text{IN}), (\psi, \text{OUT}), (\chi, \text{UNDEC})\}$.

$LabAsm_1$ is the grounded assumption labelling, $LabAsm_2$ and $LabAsm_3$ are both preferred assumption labellings, $LabAsm_1$ is the ideal assumption labelling, and $LabAsm_2$ is the only stable as well as the only semi-stable assumption labelling.

The following theorem proves that the grounded, preferred, ideal, semi-stable, and stable assumption labellings correspond one-to-one to the respective assumption extensions.

Theorem 3.5.

1. *Let Asms be a grounded / preferred / ideal / semi-stable / stable assumption extension. Then LabAsm with* $\text{IN}(LabAsm) = Asms$, $\text{OUT}(LabAsm) = Asms^+$, *and* $\text{UNDEC}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is a grounded / preferred / ideal / semi-stable / stable assumption labelling.*

2. *Let LabAsm be a grounded / preferred / ideal / semi-stable / stable assumption labelling. Then* $Asms = \text{IN}(LabAsm)$ *is a grounded / preferred / ideal / semi-stable / stable assumption extension with* $Asms^+ = \text{OUT}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \text{UNDEC}(LabAsm)$.

Proof.

1. Let *Asms* be a 1) grounded 2) preferred 3) ideal 4) semi-stable 5) stable assumption extension. By definition *Asms* is a complete assumption extension. Furthermore, for all complete assumption extensions $Asms'$ it holds that 1) $Asms' \not\subset Asms$ 2) $Asms' \not\supset Asms$ 3) if for all preferred assumption extensions $Asms''$ it holds that $Asms' \subseteq Asms''$, then $Asms' \not\supset Asms$ 4) $Asms' \cup Asms'^+ \not\supset Asms \cup Asms^+$ 5) $Asms \cup Asms^+ = \mathcal{A}$. By Theorem 3.4 *LabAsm* is a complete assumption labelling. Furthermore, from the above and Theorem 3.4, for all complete assumption labellings $LabAsm'$ it holds that 1) $\text{IN}(LabAsm') \not\subset \text{IN}(LabAsm)$ 2) $\text{IN}(LabAsm') \not\supset \text{IN}(LabAsm)$ 3) if for all preferred assumption labellings $LabAsm''$ it holds that $\text{IN}(LabAsm') \subseteq \text{IN}(LabAsm'')$, then $\text{IN}(LabAsm') \not\supset \text{IN}(LabAsm)$ 4) $\text{IN}(LabAsm') \cup \text{OUT}(LabAsm') \not\supset \text{IN}(LabAsm) \cup \text{OUT}(LabAsm)$, and consequently $\text{UNDEC}(LabAsm') \not\subset \text{UNDEC}(LabAsm)$ 5) $\text{IN}(LabAsm) \cup \text{OUT}(LabAsm) = \mathcal{A}$, and consequently $\text{UNDEC}(LabAsm) = \emptyset$. Therefore, *LabAsm* is a 1) grounded 2) preferred 3) ideal 4) semi-stable 5) stable assumption labelling.

2. Let *LabAsm* be a 1) grounded 2) preferred 3) ideal 4) semi-stable 5) stable assumption labelling. By definition *LabAsm* is a complete assumption labelling. Furthermore, for all complete assumption labellings $LabAsm'$ it holds that 1) $\text{IN}(LabAsm') \not\subset$

$\text{IN}(LabAsm)$ 2) $\text{IN}(LabAsm') \not\supset \text{IN}(LabAsm)$ 3) if for all preferred assumption labellings $LabAsm''$ it holds that $\text{IN}(LabAsm') \subseteq \text{IN}(LabAsm'')$, then $\text{IN}(LabAsm') \not\supset \text{IN}(LabAsm)$ 4) $\text{UNDEC}(LabAsm') \not\subset \text{UNDEC}(LabAsm)$, or equivalently $\text{IN}(LabAsm') \cup \text{OUT}(LabAsm') \not\supset \text{IN}(LabAsm) \cup \text{OUT}(LabAsm)$ 5) $\text{UNDEC}(LabAsm) = \emptyset$, or equivalently $\text{IN}(LabAsm) \cup \text{OUT}(LabAsm) = \mathcal{A}$. By Theorem 3.4 $Asms = \text{IN}(LabAsm)$ is a complete assumption extension with $Asms^+ = \text{OUT}(LabAsm)$ and $\mathcal{A} \setminus (Asms \cup Asms^+) = \text{UNDEC}(LabAsm)$. Furthermore, from the above and by Theorem 3.4, for all complete assumption extensions $Asms'$ it holds that 1) $Asms' \not\subset Asms$ 2) $Asms' \not\supset Asms$ 3) if for all preferred assumption extensions $Asms''$ it holds that $Asms' \subseteq Asms''$, then $Asms' \not\supset Asms$ 4) $Asms' \cup Asms'^+ \not\supset Asms \cup Asms^+$ 5) $Asms \cup Asms^+ = \mathcal{A}$. Therefore, $Asms$ is a 1) grounded 2) preferred 3) ideal 4) semi-stable 5) stable assumption extension.

□

Corollary 3.6 follows straightaway from the correspondence of grounded and ideal assumption labellings and extensions and the uniqueness of grounded and ideal assumption extensions [BDKT97, DMT07].

Corollary 3.6. *The grounded and ideal assumption labellings are both unique.*

We now show that preferred, ideal, semi-stable, and stable assumption labellings can be redefined in terms of admissible (rather than complete) assumption labellings.

Proposition 3.7. *Let LabAsm be an admissible assumption labelling.*

- *LabAsm is a preferred assumption labelling if and only if* $\text{IN}(LabAsm)$ *is maximal (w.r.t.* $\subseteq$*) among all admissible assumption labellings.*
- *LabAsm is an ideal assumption labelling if and only if* $\text{IN}(LabAsm)$ *is maximal (w.r.t.* $\subseteq$*) among all admissible assumption labellings satisfying that for all preferred assumption labellings* $LabAsm'$, $\text{IN}(LabAsm) \subseteq \text{IN}(LabAsm')$.
- *LabAsm is a semi-stable assumption labelling if and only if* $\text{UNDEC}(LabAsm)$ *is minimal (w.r.t.* $\subseteq$*) among all admissible assumption labellings.*
- *LabAsm is a stable assumption labelling if and only if* $\text{UNDEC}(LabAsm) = \emptyset$.

Proof.

- Preferred: Follows from the one-to-one correspondence between complete assumption labellings and extensions (Theorem 3.4) and between admissible assumption labellings and extensions (Theorem 3.1) together with Theorem 8 in [CSAD15a].
- Ideal: Follows from the one-to-one correspondence between complete assumption labellings and extensions (Theorem 3.4) and between admissible assumption labellings and extensions (Theorem 3.1) together with Theorem 10 in [CSAD15a].

- Semi-stable: Left to right: Let *LabAsm* be a semi-stable assumption labelling, i.e. a complete assumption labelling such that $\textsc{undec}(\mathit{LabAsm})$ is minimal among all complete assumption labellings. By definition, *LabAsm* is an admissible assumption labelling. Assume $\textsc{undec}(\mathit{LabAsm})$ is not minimal among all admissible assumptions labellings, i.e. $\exists \mathit{LabAsm}'$ with $\textsc{undec}(\mathit{LabAsm}') \subset \textsc{undec}(\mathit{LabAsm})$ and LabAsm' is an admissible assumption labelling but not a complete assumption labelling. Thus LabAsm' satisfies Definition 3.2 but not Definition 3.3, so $\exists \alpha \in \textsc{undec}(\mathit{LabAsm}')$ such that for all sets of assumptions *Asms* attacking α there exists some $\beta \in \mathit{Asms}$ such that $\mathit{LabAsm}'(\beta) = \textsc{out}$. By Lemma 3.2, LabAsm'' with $\textsc{in}(\mathit{LabAsm}'') = \textsc{in}(\mathit{LabAsm}') \cup \{\alpha\}$, $\textsc{out}(\mathit{LabAsm}'') = \textsc{out}(\mathit{LabAsm}') \cup \alpha^\star$, and $\textsc{undec}(\mathit{LabAsm}'') = \textsc{undec}(\mathit{LabAsm}') \setminus (\{\alpha\} \cup \alpha^\star)$ is an admissible assumption labelling. Clearly, $\textsc{undec}(\mathit{LabAsm}'') \subset \textsc{undec}(\mathit{LabAsm}')$, so $\textsc{undec}(\mathit{LabAsm}')$ is not minimal among all admissible assumption labellings. Contradiction.
  Right to left: Let *LabAsm* be an admissible assumption labelling such that $\textsc{undec}(\mathit{LabAsm})$ is minimal (w.r.t. $\subseteq$) among all admissible assumption labellings. Assume that *LabAsm* is not a complete assumption labelling. By the same reasoning as above, $\exists \alpha \in \textsc{undec}(\mathit{LabAsm})$ such that for all sets of assumptions *Asms* attacking α there exists some $\beta \in \mathit{Asms}$ such that $\mathit{LabAsm}(\beta) = \textsc{out}$. It follows that there exists an admissible assumption labelling LabAsm'' with $\textsc{undec}(\mathit{LabAsm}'') \subset \textsc{undec}(\mathit{LabAsm})$. Contradiction. Thus, *LabAsm* is a complete assumption labelling. Furthermore, since for all admissible assumption labellings LabAsm' it holds that $\textsc{undec}(\mathit{LabAsm}') \not\subset \textsc{undec}(\mathit{LabAsm})$ and since every complete assumption labelling is an admissible assumption labelling, it follows that for all complete assumption labellings LabAsm', $\textsc{undec}(\mathit{LabAsm}') \not\subset \textsc{undec}(\mathit{LabAsm})$. Thus, $\textsc{undec}(\mathit{LabAsm})$ is minimal (w.r.t. $\subseteq$) among all complete assumption labellings.

- Stable: Left to right: Let *LabAsm* be a complete assumption labelling such that $\textsc{undec}(\mathit{LabAsm}) = \emptyset$. By definition, *LabAsm* is admissible.
  Right to left: Let *LabAsm* be an admissible assumption labelling such that $\textsc{undec}(\mathit{LabAsm}) = \emptyset$. Then *LabAsm* is also a complete assumption labelling since the conditions for $\textsc{in}$ and $\textsc{out}$ assumptions are the same for admissible and complete assumption labellings (see Definitions 3.2 and 3.3).

□

Example 3.8. The results from Proposition 3.7 are illustrated by ABA_2 (see Examples 3.1 and 3.5). For instance, the only maximal admissible assumption labelling of ABA_2 is LabAsm_3, which is also the only maximal complete, and thus preferred, assumption labelling of ABA_2.

3.3 Argument-Supporting Sets of Assumptions

Determining admissible or complete assumption labellings of an ABA framework as well as checking whether an assumption labelling is admissible or complete can be cumbersome since some conditions specifying the label of an assumption α require to consider *every* set of assumptions attacking α. In particular, not only the set of premises of an argument with conclusion $\overline{\alpha}$ attacks α, but also every *superset* thereof.

Example 3.9. To verify whether χ is correctly labelled in the admissible assumption labelling $LabAsm_3 = \{(\rho, \textsc{in}), (\psi, \textsc{out}), (\chi, \textsc{in})\}$ of ABA_2 (see Example 3.1), not only the set of assumptions $\{\psi\}$, which forms the premises of an argument with conclusion x (the contrary of χ), but also every superset thereof, i.e. $\{\rho, \psi\}$, $\{\psi, \chi\}$, and $\{\rho, \psi, \chi\}$, has to be checked.

In this section, we show that considering only sets of assumptions that form the premises of some argument, which we call *argument-supporting* sets of assumptions, when determining or checking assumption labellings is equivalent to considering all sets of assumptions. This is inspired by the fact that assumption extensions can be determined and checked by considering either all or only argument-supporting sets of assumptions [DKT06].

3.3.1 Assumption Labellings with respect to Argument-Supporting Sets of Assumptions

A set of assumptions is *argument-supporting* if it forms the premises of some argument.

Definition 3.5 (Argument-Supporting set of Assumptions)**.** Let $Asms \subseteq \mathcal{A}$ be a set of assumptions. $Asms$ is an *argument-supporting* set of assumptions if and only if there exists some $s \in \mathcal{L}$ such that $Asms \vdash s$.

Note that all singleton sets of assumptions are argument-supporting, i.e. for every assumption $\alpha \in \mathcal{A}$, $\{\alpha\}$ is an argument-supporting set of assumptions, since $\{\alpha\} \vdash \alpha$.

Notation 3.6. The set of all argument-supporting sets of assumptions is $\mathcal{S}_{arg} = \{Asms \subseteq \mathcal{A} \mid Asms \textit{ is an argument-supporting set of assumptions}\}$.

We define a variant of admissible assumption labellings where only argument-supporting, rather than all, sets of assumptions attacking an assumption are taken into account.

Definition 3.7 (Admissible Assumption Labelling w.r.t. Argument-Supporting Sets)**.** Let $LabAsm$ be an assumption labelling. $LabAsm$ is an *admissible assumption labelling w.r.t. argument-supporting sets* if and only if for each assumption $\alpha \in \mathcal{A}$ it holds that:

- if $LabAsm(\alpha) = \textsc{in}$, then for each argument-supporting set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \textsc{out}$;

- if $LabAsm(\alpha) = \text{OUT}$, then there exists an argument-supporting set of assumptions $Asms$ attacking α such that for all $\beta \in Asms$, $LabAsm(\beta) = \text{IN}$;

- if $LabAsm(\alpha) = \text{UNDEC}$, then for each argument-supporting set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) \neq \text{IN}$.

To check whether χ in ABA_2 is correctly labelled according to admissible assumption labellings w.r.t. argument-supporting sets, only the set $\{\psi\}$ has to be taken into account (compare Example 3.9).

The following Lemma shows that our definition of admissible assumption labellings (Definition 3.2) and the new definition of admissible assumption labellings w.r.t. argument-supporting sets can be used interchangeably. This extends the result of Dung et al. [DKT06] that admissible assumption extensions can be equivalently defined in terms of all sets of assumptions or argument-supporting sets of assumptions.

Lemma 3.8. *Let LabAsm be an assumption labelling. LabAsm is an admissible assumption labelling if and only if LabAsm is an admissible assumption labelling w.r.t. argument-supporting sets.*

Proof. Left to right: Let $LabAsm$ be an admissible assumption labelling.

- Let $LabAsm(\alpha) = \text{IN}$. Then for each set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \text{OUT}$.

- Let $LabAsm(\alpha) = \text{OUT}$. Then there exists a set of assumptions $Asms$ attacking α such that for all $\beta \in Asms$, $LabAsm(\beta) = \text{IN}$. Thus, there exists an argument $Asms' \vdash \overline{\alpha}$ such that $Asms' \subseteq Asms$. Therefore, $Asms'$ is an argument-supporting set of assumptions attacking α such that for all $\beta \in Asms'$, $LabAsm(\beta) = \text{IN}$.

- Let $LabAsm(\alpha) = \text{UNDEC}$. Then for each set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) \neq \text{IN}$.

Right to left: Let $LabAsm$ be an admissible assumption labelling w.r.t. argument-supporting sets.

- Let $LabAsm(\alpha) = \text{IN}$. Then for each argument-supporting set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \text{OUT}$. Since each set of assumptions $Asms'$ attacking α is a superset of some argument-supporting set of assumptions attacking α, it follows that for each set of assumptions $Asms'$ attacking α there exists some $\beta \in Asms'$ such that $LabAsm(\beta) = \text{OUT}$.

- Let $LabAsm(\alpha) = \text{OUT}$. Then there exists an argument-supporting set of assumptions $Asms$ attacking α such that for all $\beta \in Asms$, $LabAsm(\beta) = \text{IN}$.

- Let $LabAsm(\alpha) = \text{UNDEC}$. Then for each argument-supporting set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) \neq \text{IN}$. Since each set of assumptions $Asms'$ attacking α is a superset of some argument-supporting set of assumptions, it follows that for each set of assumptions $Asms'$ attacking α there exists some $\beta \in Asms'$ such that $LabAsm(\beta) \neq \text{IN}$.

□

Analogously to admissible assumption labellings, we define a variant of complete assumption labellings where only argument-supporting sets of assumptions attacking an assumption in question are taken into account.

Definition 3.8 (Complete Assumption Labelling w.r.t. Argument-Supporting Sets)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is a *complete assumption labelling w.r.t. argument-supporting sets* if and only if *LabAsm* is an admissible assumption labelling w.r.t. argument-supporting sets and for each assumption $\alpha \in \mathcal{A}$ it holds that:

- if $LabAsm(\alpha) = \text{UNDEC}$, then there exists an argument-supporting set of assumptions $Asms$ attacking α such that for all $\gamma \in Asms$, $LabAsm(\gamma) \neq \text{OUT}$.

As for admissible assumption labellings, the notions of complete assumption labellings and complete assumption labellings w.r.t. argument-supporting sets are equivalent.

Proposition 3.9. *Let LabAsm be an assumption labelling. LabAsm is a complete assumption labelling if and only if LabAsm is a complete assumption labelling w.r.t. argument-supporting sets.*

Proof.

- Left to right: Let *LabAsm* be a complete assumption labelling. By definition, *LabAsm* is an admissible assumption labelling and by Lemma 3.8 an admissible assumption labelling w.r.t. argument-supporting sets. It remains to prove that the additional condition of complete assumption labellings w.r.t. argument-supporting sets is satisfied. Let $LabAsm(\alpha) = \text{UNDEC}$. Then there exists a set of assumptions $Asms$ attacking α such that for all $\beta \in Asms$, $LabAsm(\beta) \neq \text{OUT}$. Thus, there exists an argument $Asms' \vdash \overline{\alpha}$ such that $Asms' \subseteq Asms$. Therefore, $Asms'$ is an argument-supporting set of assumptions attacking α such that for all $\gamma \in Asms'$, $LabAsm(\gamma) \neq \text{OUT}$.

- Right to left: Let *LabAsm* be a complete assumption labelling w.r.t. argument-supporting sets. By definition, *LabAsm* is an admissible assumption labelling w.r.t. argument-supporting sets and by Lemma 3.8 an admissible assumption labelling. It remains to prove that the additional condition of complete assumption labellings is satisfied. Let $LabAsm(\alpha) = \text{UNDEC}$. Then there exists an argument-supporting set of assumptions $Asms$ attacking α such that for all $\gamma \in Asms$, $LabAsm(\gamma) \neq \text{OUT}$.

□

Since the grounded, preferred, ideal, semi-stable, and stable assumption labellings are based on complete/admissible assumption labellings, it follows that they can be equivalently defined in terms of complete/admissible assumption labellings w.r.t. argument-supporting sets.

Depending on the ABA framework, and in particular on the set of inference rules $\mathcal{R}$, the set of all argument-supporting sets of assumptions $\mathcal{S}_{arg}$ may be equal to the set of all sets of assumptions $\wp(\mathcal{A})$ or a subset thereof with much lower cardinality. For example, in ABA_2 from Example 3.1, the set of all argument-supporting sets of assumptions consists only of the singleton sets, i.e. $\{\{\rho\}, \{\psi\}, \{\chi\}\}$, whereas the set of all sets of assumptions is $\wp(\{\rho, \psi, \chi\}) = \{\{\}, \{\rho\}, \{\psi\}, \{\chi\}, \{\rho, \psi\}, \{\rho, \chi\}, \{\psi, \chi\}, \{\rho, \psi, \chi\}\}$. Therefore, considering only argument-supporting sets of assumptions may in the best case require to check only a fraction of all sets of assumptions, but in the worst case it is exactly the same.

Observation 3.10. *Let $\mathcal{S}_{all} = \wp(\mathcal{A})$ be the set of all sets of assumptions, so $|\mathcal{S}_{all}| = 2^{|\mathcal{A}|}$.*

- *In the* best case*, $|\mathcal{S}_{arg}| = |\mathcal{A}|$. This is for example the case if $\mathcal{R} = \emptyset$, since the only argument-supporting sets of assumptions are the singleton sets.*
- *In the* worst case*, $|\mathcal{S}_{arg}| = |\mathcal{S}_{all}| = 2^{|\mathcal{A}|}$. This is for example the case if $\mathcal{R}$ is such that for each $Asms \in \mathcal{S}_{all}$ there exists some inference rule $s_0 \leftarrow s_1, \ldots, s_n \in \mathcal{R}$ such that $Asms = \{s_1, \ldots, s_n\}$.*

Example 3.10. Let ABA_6 be the following ABA framework:

$$\begin{aligned}
&\mathcal{L} = \{p, r, \psi, \rho\},\\
&\mathcal{R} = \{p \leftarrow\ ;\ p \leftarrow \rho;\ r \leftarrow \psi;\ r \leftarrow \psi, \rho\},\\
&\mathcal{A} = \{\psi, \rho\},\\
&\overline{\psi} = p\ ,\ \overline{\rho} = r.
\end{aligned}$$

Here, the set of all argument-supporting sets of assumptions is $\mathcal{S}_{arg} = \{\{\}, \{\psi\}, \{\rho\}, \{\psi, \rho\}\}$, which coincides with the set of all sets of assumptions.

3.3.2 ABA Graphs

In most of the ABA literature (e.g. [DKT06, DMT07, MM09, Ton13, Ton14, HS16]), ABA frameworks are not displayed graphically; they are simply given as tuples, as done in the Examples presented so far. We introduce *ABA graphs*, where nodes are argument-supporting sets of assumptions and edges are attacks between these argument-supporting sets of assumptions.

Definition 3.9 (ABA Graph)**.** The *ABA graph* $\mathcal{G} = (V, E)$ is a directed graph with $V = \mathcal{S}_{arg}$ and $E = \{(Asms_1, Asms_2) \mid Asms_1, Asms_2 \in V$ *and* $Asms_1$ *attacks* $Asms_2\}$.

The ABA graph of ABA_2 from Example 3.1 has only three nodes, namely the singleton sets of assumptions, as shown on the left of Figure 3.5. As a comparison, the right of Figure 3.5 illustrates the graph of all sets of assumptions and attacks between them.

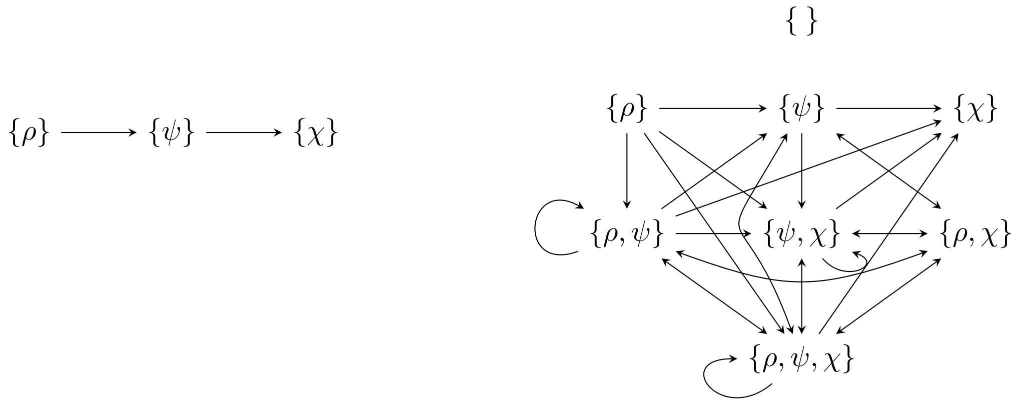


Figure 3.5: Left – the ABA graph of ABA_2. Right – the graph illustrating all sets of assumptions of ABA_2 and all attacks between them.

Since an ABA graph illustrates all argument-supporting sets of assumptions and attacks between them, an ABA graph can be used to determine the semantics of an ABA framework.

Example 3.11. The ABA graph of ABA_5 (see Example 3.7) is displayed on the left of Figure 3.6. It illustrates which (argument-supporting) sets of assumptions have to be taken into account when determining complete or admissible assumption labellings (w.r.t. argument-supporting sets). For example, for ρ to be labelled IN by a complete assumption labelling (w.r.t. argument-supporting sets), all (argument-supporting) sets of assumptions attacking ρ have to contain an assumption labelled OUT. Since the only set of assumptions attacking ρ in the ABA graph is $\{\psi\}$, we deduce that ψ has to be labelled OUT by any complete assumption labelling that labels ρ as IN. It is then easy to verify, based on the two sets of assumptions attacking χ, that with ψ labelled OUT, χ can only be labelled UNDEC. This complete assumption labelling of ABA_5 is illustrated in the ABA graph on the right of Figure 3.6.

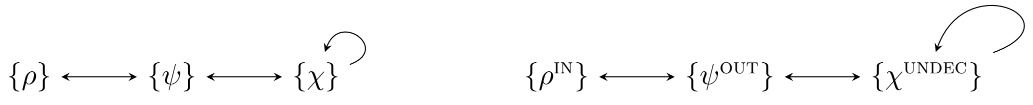


Figure 3.6: The ABA graph of ABA_5 (see Example 3.11). The right version also indicates one of the complete assumption labellings of ABA_5.

Another graphical representation of an ABA framework is the *attack relationship graph* [BDKT97], which was introduced to characterise different types of ABA frameworks. The question thus arises whether attack relationship graphs can also be used to determine the semantics, in particular the assumption labellings, of an ABA framework.

The *attack relationship graph* $\mathcal{G}_{att} = (V, E)$ is a directed graph with $V = \mathcal{A}$ and $E = \{(\alpha, \beta) \mid \alpha, \beta \in V$ *and* $\alpha \in$ *Asms such that Asms attacks* β

and $\nexists Asms' \subset Asms$ such that $Asms'$ attacks $\beta\}$.

The main difference between an ABA graph and an attack relationship graph is that the vertices of an ABA graph are *sets* of assumptions, including all the singleton sets, whereas the vertices of an attack relationship graph are single assumptions. The following example demonstrates that attack relationship graphs do not capture enough information to determine the semantics of an ABA framework.

Example 3.12. Let ABA_7 be the following ABA framework:

$$\begin{aligned}
&\mathcal{L} = \{\rho, \psi, \chi, \phi, \omega, r, p, x\},\\
&\mathcal{R} = \{r \leftarrow \phi \,;\, r \leftarrow \omega \,;\, r \leftarrow \psi, \chi\},\\
&\mathcal{A} = \{\rho, \psi, \chi, \phi, \omega\},\\
&\overline{\rho} = r,\ \overline{\psi} = p,\ \overline{\chi} = x,\ \overline{\phi} = \psi,\ \overline{\omega} = \psi.
\end{aligned}$$

Furthermore, let ABA_8 have the same $\mathcal{L}$, $\mathcal{A}$, and contraries as ABA_7, but with $\mathcal{R} = \{r \leftarrow \phi \,;\, r \leftarrow \psi, \omega \,;\, r \leftarrow \chi, \omega\}$. The ABA graphs of ABA_7 and ABA_8, which are structurally different, are displayed in Figure 3.7. In contrast, the attack relationship graphs of ABA_7 and ABA_8 are the same, as illustrated in Figure 3.8. Thus, it is impossible to distinguish ABA_7 and ABA_8 based on the attack relationship graphs. However, the two ABA frameworks have different complete labellings, as indicated in Figure 3.7. It is therefore not possible to determine the complete or admissible assumption labellings of ABA_7 and ABA_8 based on their attack relationship graphs. Furthermore, it is in general not the case that complete or admissible assumption labellings of an ABA framework can be determined based on its attack relationship graph.

$$\begin{tikzcd}
\{\rho^{\text{OUT}}\} & \{\psi^{\text{IN}}, \chi^{\text{IN}}\} \arrow[l] \arrow[ddl] & \{\chi^{\text{IN}}\} \\
 & & \{\omega^{\text{OUT}}\} \arrow[ull] \\
\{\phi^{\text{OUT}}\} \arrow[uu] & & \{\psi^{\text{IN}}\} \arrow[u] \arrow[ll]
\end{tikzcd}$$

$$\begin{tikzcd}
\{\rho^{\text{IN}}\} & \{\chi^{\text{IN}}, \omega^{\text{OUT}}\} \arrow[l] & \{\chi^{\text{IN}}\} \\
 & \{\psi^{\text{IN}}, \omega^{\text{OUT}}\} \arrow[loop left] \arrow[ul] \arrow[u] \arrow[r] \arrow[dl] & \{\omega^{\text{OUT}}\} \\
\{\phi^{\text{OUT}}\} \arrow[uu] & & \{\psi^{\text{IN}}\} \arrow[u] \arrow[ul] \arrow[uul] \arrow[ll]
\end{tikzcd}$$

Figure 3.7: The ABA graphs of ABA_7 (left) and ABA_8 (right) from Example 3.12, each with its only complete assumption labelling.

Conversely, ABA graphs cannot be (straightforwardly) used to characterise different types of ABA frameworks. For example, an ABA framework is *stratified* if and only if its attack relationship graph does not have an infinite sequence of edges [BDKT97]. However, ABA graphs may have an infinite sequence of edges even if the attack relationship graph does not, as demonstrated in Example 3.13.

$$\begin{tikzcd}
\rho & & \chi \arrow[ll] \\
& & \omega \arrow[llu] \\
\phi \arrow[uu] & & \psi \arrow[lluu] \arrow[u] \arrow[ll]
\end{tikzcd}$$

Figure 3.8: The attack relationship graph of both ABA_7 and ABA_8 from Example 3.12.

Example 3.13. Let ABA_9 be the following ABA framework:

$$\begin{aligned}
&\mathcal{L} = \{p, r, x, \psi, \rho, \chi\},\\
&\mathcal{R} = \{r \leftarrow \psi\ ;\ x \leftarrow \rho\ ;\ r \leftarrow \psi, \chi\},\\
&\mathcal{A} = \{\psi, \rho, \chi\},\\
&\overline{\psi} = p\ ,\ \overline{\rho} = r\ ,\ \overline{\chi} = x.
\end{aligned}$$

The attack relationship graph and the ABA graph of ABA_9 are displayed in Figure 3.9. Since the attack relationship graph does not have any infinite sequence of edges, ABA_9 is stratified. However, the ABA graph does have an infinite sequence of edges since it comprises a cycle.

$$\begin{tikzcd}
\psi \arrow[r] & \rho \arrow[r] & \chi
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\{\psi\} \arrow[r] & \{\rho\} \arrow[r] \arrow[d, bend right] & \{\chi\} \\
& \{\psi, \chi\} \arrow[u, bend right] &
\end{tikzcd}$$

Figure 3.9: The attack relationship graph (left) and the ABA graph (right) of ABA_9 from Example 3.13.

The example illustrates that an infinite sequence of edges in an ABA graph does not indicate that the ABA framework is not stratified.

Proposition 3.11. *Let $\mathcal{G}$ be the ABA graph and $\mathcal{G}_{att}$ the attack relationship graph of $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \overline{\ \ } \rangle$. If $\mathcal{G}_{att}$ has an infinite sequence of edges, then $\mathcal{G}$ has an infinite sequence of edges, but not vice versa.*

Proof. Let there be an infinite sequence of edges $(\alpha_1, \alpha_2), (\alpha_2, \alpha_3), \ldots$ in $\mathcal{G}_{att}$. Then there exists a set of assumptions $Asms_{\alpha_1}$ attacking α_2 such that $\alpha_1 \in Asms_{\alpha_1}$, a set of assumptions $Asms_{\alpha_2}$ attacking α_3 such that $\alpha_2 \in Asms_{\alpha_2}$, and so on. Thus, in $\mathcal{G}$ there exists an edge from $Asms_{\alpha_1}$ to $\{\alpha_2\}$ as well as to every other set of assumptions containing α_2, in particular an edge to $Asms_{\alpha_2}$. Furthermore, there is an edge from $Asms_{\alpha_2}$ to $\{\alpha_3\}$ as well as every set of assumptions containing α_3, and so on. Thus, there is an infinite sequence of edges $(Asms_{\alpha_1}, Asms_{\alpha_2}), (Asms_{\alpha_2}, Asms_{\alpha_3}), \ldots$ in $\mathcal{G}$.
Example 3.13 proves that the converse does not hold. □

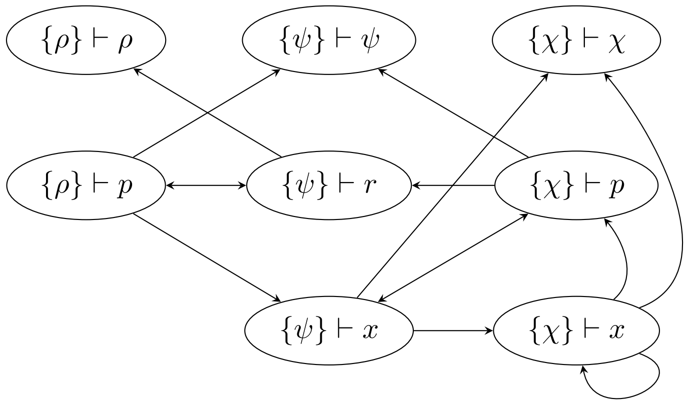


Figure 3.10: AA graph of the corresponding AA framework of ABA_5.

Another way to graphically represent an ABA framework is in terms of the *AA graph* of its corresponding AA framework, as for example done in [CSAD15a, ST16]. Interestingly, even though nodes in an ABA graph are *argument*-supporting sets of assumptions, an ABA graph does not generally have the same number of nodes as the AA graph of the corresponding AA framework, where nodes are *arguments*. In particular, an AA graph may have more nodes than an ABA graph since the same set of assumptions may form the set of premises of various arguments. As an example, compare the ABA graph of ABA_5 shown in Figure 3.6 with the AA graph of its corresponding AA framework illustrated in Figure 3.10.

Recently a new way to represent arguments of an ABA framework has been introduced with the purpose of eliminating redundancies in arguments [CT16a], namely as *argument graphs* rather than trees. Various argument graphs can furthermore be combined to form a larger argument graph, which represents a set of arguments without redundancies. Since the semantics of ABA frameworks in terms of argument graphs is slightly different from the semantics in terms of assumption and argument extensions [CT16a], a detailed comparison between argument graphs and ABA graphs is beyond the scope of this thesis.

3.4 Assumption Labellings versus Argument Labellings

In this section, we examine the relationship between assumption labellings of an ABA framework and argument labellings of its corresponding AA framework. In the remainder, and if clear from the context, we assume as given a flat ABA framework $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^- \rangle$ and its corresponding AA framework $\langle Ar_{ABA}, Att_{ABA} \rangle$.

3.4.1 Translating between Assumption and Argument Labellings

Before going into detail about the (non-) correspondence between assumption and argument labellings according to the various semantics, we show that there is a correspondence between attacks in the ABA framework and attacks in its corresponding AA framework.

Lemma 3.12 states that given a set of assumptions $Asms$, the set of arguments constructable from these assumptions attacks exactly those arguments supported by some assumption attacked by $Asms$. Conversely, Lemma 3.13 states that given a set of arguments $Args$, the set of assumptions supporting these arguments attacks exactly those assumptions whose assumption-arguments are attacked by $Args$.

Lemma 3.12. *Let $Asms \subseteq \mathcal{A}$ and $Args = \{Asms' \vdash s \in Ar_{ABA} \mid Asms' \subseteq Asms\}$. Then*

- $Args^+ = \{Asms' \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms' : \alpha \in Asms^+\}$;
- $Ar_{ABA} \setminus (Args \cup Args^+) = \{Asms' \vdash s \in Ar_{ABA} \mid Asms' \nsubseteq Asms, \nexists \alpha \in Asms' : \alpha \in Asms^+\}$.

Proof. We prove both statements:

- $Args^+ = \{Asms' \vdash s \in Ar_{ABA} \mid Args \text{ attacks } Asms' \vdash s\}$
  $= \{Asms' \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms' : \exists Asms'' \vdash \overline{\alpha} \in Args\}$
  $= \{Asms' \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms' : \exists Asms'' \vdash \overline{\alpha} \text{ and } Asms'' \subseteq Asms\}$
  $= \{Asms' \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms' : Asms \text{ attacks } \alpha\}$
  $= \{Asms' \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms' : \alpha \in Asms^+\}$
- $Ar_{ABA} \setminus (Args \cup Args^+) = \{Asms' \vdash s \in Ar_{ABA} \mid Asms' \vdash s \notin Args, Asms' \vdash s \notin Args^+\} = \{Asms' \vdash s \in Ar_{ABA} \mid Asms' \nsubseteq Asms, \nexists \alpha \in Asms' : \alpha \in Asms^+\}$

□

Lemma 3.13. *Let $Args \subseteq Ar_{ABA}$ and let $Asms = \{\alpha \in \mathcal{A} \mid \exists Asms' : \alpha \in Asms' \text{ and } Asms' \vdash s \in Args\}$. Then*

- $Asms^+ = \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \in Args^+\}$;
- $\mathcal{A} \setminus (Asms \cup Asms^+) = \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \notin Args, \{\alpha\} \vdash \alpha \notin Args^+\}$.

Proof. We prove both statements:

- $Asms^+ = \{\alpha \in \mathcal{A} \mid Asms \text{ attacks } \alpha\} = \{\alpha \in \mathcal{A} \mid \exists Asms' \vdash \overline{\alpha} : Asms' \subseteq Asms\}$
  $= \{\alpha \in \mathcal{A} \mid \exists Asms' \vdash \overline{\alpha} \in Args\} = \{\alpha \in \mathcal{A} \mid Args \text{ attacks } \{\alpha\} \vdash \alpha\}$
  $= \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \in Args^+\}$
- $\mathcal{A} \setminus (Asms \cup Asms^+) = \{\alpha \in \mathcal{A} \mid \alpha \notin Asms, \alpha \notin Asms^+\}$
  $= \{\alpha \in \mathcal{A} \mid \nexists Asms' : \alpha \in Asms' \text{ and } Asms' \vdash s \in Args, \{\alpha\} \vdash \alpha \notin Args^+\}$
  $= \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \notin Args, \{\alpha\} \vdash \alpha \notin Args^+\}$

□

Next, we examine the number of assumption labellings of an ABA framework as compared to the number of argument labellings of its corresponding AA framework.

Notation 3.10. $\mathscr{L}_{Asm}$ denotes the set of all assumption labellings of $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \overline{\ } \rangle$ and $\mathscr{L}_{Arg}$ the set of all argument labellings of $\langle Ar_{ABA}, Att_{ABA} \rangle$.

First, we observe that the number of all possible assumption labellings of an ABA framework is smaller than or equal to the number of all possible argument labellings of its corresponding AA framework since an assumption labelling labels only assumptions, i.e. $|\mathcal{A}|$ elements, whereas an argument labelling labels every assumption-argument as well as every argument constructed using inference rules in $\mathcal{R}$.

Observation 3.14. *Since assumption labellings assign one of three labels to each assumption, $|\mathscr{L}_{Asm}| = 3^{|\mathcal{A}|}$. Since argument labellings assign one of three labels to each argument, $|\mathscr{L}_{Arg}| = 3^{|Ar_{ABA}|}$.*

- *In the* best case $|\mathscr{L}_{Arg}| = |\mathscr{L}_{Asm}| = 3^{|\mathcal{A}|}$. *This is the case if the only arguments are assumption-arguments, so $|Ar_{ABA}| = |\mathcal{A}|$, for example if $\mathcal{R} = \emptyset$.*
- *In* all other cases $|\mathscr{L}_{Arg}| > |\mathscr{L}_{Asm}|$. *This is the case if there exists at least one argument that is not an assumption-argument, so $|Ar_{ABA}| > |\mathcal{A}|$, for example if there exists an inference rule $s \leftarrow\ \in \mathcal{R}$.*

As an example, ABA_5 from Example 3.7 has three assumptions, so there are $|\mathscr{L}_{Asm}| = 3^3 = 27$ possible assumption labellings. In contrast, the corresponding AA framework of ABA_5 has eight arguments (see Figure 3.10), so there are $|\mathscr{L}_{Arg}| = 3^8 = 6561$ possible argument labellings. We will see in the following sections that even though the number of possible assumption labellings of an ABA framework may be less than the number of possible argument labellings of its corresponding AA framework, the number of assumption and argument labellings according to various semantics is the same.

In order to compare assumption and argument labellings, we define two functions for translating between the two types of labellings. The first translation, `LabAsm2LabArg`, determines the labels of arguments based on the given labels of premises of these arguments.

Definition 3.11 (Mapping an Assumption Labelling into an Argument Labelling). `LabAsm2LabArg` : $\mathscr{L}_{Asm} \rightarrow \mathscr{L}_{Arg}$ maps an assumption labelling *LabAsm* into an argument labelling *LabArg* such that:

- $\texttt{in}(LabArg) = \{Asms \vdash s \in Ar_{ABA} \mid Asms \subseteq \textsc{in}(LabAsm)\}$;
- $\texttt{out}(LabArg) = \{Asms \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms : \alpha \in \textsc{out}(LabAsm)\}$;
- $\texttt{undec}(LabArg) = \{Asms \vdash s \in Ar_{ABA} \mid \exists \alpha \in Asms : \alpha \in \textsc{undec}(LabAsm), Asms \cap \textsc{out}(LabAsm) = \emptyset\}$.

`LabAsm2LabArg` mirrors the correspondence between assumption and argument extensions (see Section 2.2.3) through the mapping from IN-labelled assumption into `in`-labelled arguments. In addition, `LabAsm2LabArg` maps assumptions labelled OUT and UNDEC into arguments labelled `out` and `undec`, respectively. An argument is labelled `out` if one of its premises is labelled OUT, independently of the labels of its other premises. The intuition of this translation is that an assumption α which is labelled OUT is attacked by a set of assumptions labelled IN. Since this set gives rise to an `in`-labelled argument, any argument that has α as its premise is attacked by an `in`-labelled argument and should thus be labelled `out`. Arguments labelled `undec` are simply those whose premises fulfil neither the condition for `in`- nor for `out`-labelled arguments.

Lemma 3.15. `LabAsm2LabArg` *is an injective function but not generally a surjective function.*

Proof. Note first that `LabAsm2LabArg` is clearly a function.

- Injective: We prove that no two different assumption labellings $LabAsm_1$ and $LabAsm_2$ are mapped to the same argument labelling by `LabAsm2LabArg`. Let $LabAsm_1 \neq LabAsm_2$. Thus, $\exists \alpha \in \mathcal{A}$ such that $LabAsm_1(\alpha) \neq LabAsm_2(\alpha)$. If $\alpha \in \text{IN}(LabAsm_1)$, then $\alpha \notin \text{IN}(LabAsm_2)$, so $\{\alpha\} \vdash \alpha \in \texttt{in}(\texttt{LabAsm2LabArg}(LabAsm_1))$ but $\{\alpha\} \vdash \alpha \notin \texttt{in}(\texttt{LabAsm2LabArg}(LabAsm_2))$. Analogous results are reached assuming that $\alpha \in \text{OUT}(LabAsm_1)$ and that $\alpha \in \text{UNDEC}(LabAsm_1)$.
  Thus, $\texttt{LabAsm2LabArg}(LabAsm_1) \neq \texttt{LabAsm2LabArg}(LabAsm_2)$.

- Not generally surjective: The following ABA framework illustrates that there may be some $LabArg \in \mathscr{L}_{Arg}$ such that there exists no $LabAsm \in \mathscr{L}_{Asm}$ with $LabArg = \texttt{LabAsm2LabArg}(LabAsm)$: $\mathcal{L} = \{r, \rho\}$, $\mathcal{R} = \{r \leftarrow\}$, $\mathcal{A} = \{\rho\}$, $\overline{\rho} = r$. There are three possible assumption labellings: $LabAsm_1 = \{(\rho, \text{IN})\}$, $LabAsm_2 = \{(\rho, \text{OUT})\}$, and $LabAsm_3 = \{(\rho, \text{UNDEC})\}$. The corresponding AA framework has two arguments: $Ar_{ABA} = \{A_1 : \{\rho\} \vdash \rho, A_2 : \{\} \vdash r\}$. In the translations of all three assumption labellings in terms of `LabAsm2LabArg`, A_2 is labelled `in`. Thus, for instance for the argument labelling $\{(A_1, \texttt{in}), (A_2, \texttt{out})\}$ there exists no $LabAsm$ such that $\texttt{LabAsm2LabArg}(LabAsm) = \{(A_1, \texttt{in}), (A_2, \texttt{out})\}$.

□

The second translation, `LabArg2LabAsm`, determines the labels of assumptions based on the given labels of *assumption-arguments*.

Definition 3.12 (Mapping an Argument Labelling into an Assumption Labelling)**.** $\texttt{LabArg2LabAsm} : \mathscr{L}_{Arg} \rightarrow \mathscr{L}_{Asm}$ maps an argument labelling $LabArg$ into an assumption labelling $LabAsm$ such that:

- $\text{IN}(LabAsm) = \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \in \texttt{in}(LabArg)\}$;

- $\text{OUT}(LabAsm) = \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \in \texttt{out}(LabArg)\}$;

- $\text{UNDEC}(LabAsm) = \{\alpha \in \mathcal{A} \mid \{\alpha\} \vdash \alpha \in \texttt{undec}(LabArg)\}$.

In contrast to `LabAsm2LabArg`, the translation from `in`-labelled arguments into IN-labelled assumptions in terms of `LabArg2LabAsm` does not mirror the correspondence between argument and assumption extensions (see Section 2.2.3). In particular, the set of IN-labelled assumptions consists of all assumptions whose *assumption-arguments* are labelled `in`, rather than of all assumptions occurring as the premise of *some argument* labelled `in` (which would mirror the correspondence between argument and assumption extensions). This is to ensure that the translation of *any* argument labelling results in a well-defined assumption labelling.

Example 3.14. Let ABA_{10} be the following ABA framework:

$\mathcal{L} = \{\rho, \psi, r, p\}$,
$\mathcal{R} = \{r \leftarrow \rho\}$,
$\mathcal{A} = \{\rho, \psi\}$,
$\overline{\rho} = \psi$, $\overline{\psi} = p$.

The corresponding AA framework of ABA_{10} has three arguments, $A_1 : \{\rho\} \vdash \rho$, $A_2 : \{\psi\} \vdash \psi$, and $A_3 : \{\rho\} \vdash r$. Let *LabArg* be the argument labelling $\{(A_1, \mathtt{out}), (A_2, \mathtt{out}), (A_3, \mathtt{in})\}$. Then $\mathtt{LabArg2LabAsm}(\mathit{LabArg}) = \{(\psi, \textsc{in}), (\rho, \textsc{out})\}$ is a well-defined assumption labelling. However, if the set of assumptions labelled IN was defined in such a way that it mirrors the correspondence between argument and assumption extensions, i.e. $\textsc{in}(\mathit{LabAsm}) = \{\alpha \in \mathcal{A} \mid \exists \mathit{Asms} \vdash s \in \mathtt{in}(\mathit{LabArg}), \alpha \in \mathit{Asms}\}$, then $\rho \in \textsc{in}$ because $A_3 \in \mathtt{in}(\mathit{LabArg})$ but also $\rho \in \textsc{out}$ because $A_1 \in \mathtt{out}(\mathit{LabArg})$.

Note that if `LabArg2LabAsm` was restricted to admissible or complete, rather than arbitrary, argument and assumption labellings, the translation from `in`-labelled arguments into IN-labelled assumptions could mirror the correspondence between argument and assumption extensions [ST14].

Lemma 3.16. `LabArg2LabAsm` *is a surjective function but not generally an injective function.*

Proof. Note first that `LabArg2LabAsm` is clearly a function.

- Surjective: We prove that for every $\mathit{LabAsm} \in \mathscr{L}_{\mathit{Asm}}$ there exists some $\mathit{LabArg} \in \mathscr{L}_{\mathit{Arg}}$ such that $\mathtt{LabArg2LabAsm}(\mathit{LabArg}) = \mathit{LabAsm}$. Let $\mathit{LabAsm} \in \mathscr{L}_{\mathit{Asm}}$. Furthermore, let *LabArg* be an argument labelling that satisfies that for all $\alpha \in \mathcal{A}$, $\{\alpha\} \vdash \alpha \in \mathtt{in}(\mathit{LabArg})$ if $\alpha \in \textsc{in}(\mathit{LabAsm})$, $\{\alpha\} \vdash \alpha \in \mathtt{out}(\mathit{LabArg})$ if $\alpha \in \textsc{out}(\mathit{LabAsm})$, and $\{\alpha\} \vdash \alpha \in \mathtt{undec}(\mathit{LabArg})$ if $\alpha \in \textsc{undec}(\mathit{LabAsm})$. Then $\mathtt{LabArg2LabAsm}(\mathit{LabArg}) = \mathit{LabAsm}$. Clearly $\mathit{LabArg} \in \mathscr{L}_{\mathit{Arg}}$.

- Not generally injective: Consider the ABA framework from the proof of Lemma 3.15 and the two argument labellings $\mathit{LabArg}_1 = \{(A_1, \mathtt{in}), (A_2, \mathtt{out})\}$ and $\mathit{LabArg}_2 = \{(A_1, \mathtt{in}), (A_2, \mathtt{in})\}$. Then $\mathtt{LabArg2LabAsm}(\mathit{LabArg}_1) = \mathtt{LabArg2LabAsm}(\mathit{LabArg}_2) = \{(\rho, \textsc{in})\}$.

□

3.4.2 Complete Semantics

Due to the one-to-one correspondence between complete assumption labellings and extensions (Theorem 3.4), between complete assumption and argument extensions [Ton12, CSAD15a], and between complete argument extensions and labellings [CG09], there is also a one-to-one correspondence between complete assumption labellings and complete argument labellings. Theorem 3.17 below characterises the complete argument labelling corresponding to a given complete assumption labelling in terms of the mapping `LabAsm2LabArg`.

Theorem 3.17. *Let LabAsm be an assumption labelling. LabAsm is a complete assumption labelling if and only if* `LabAsm2LabArg`$(LabAsm)$ *is a complete argument labelling.*

Proof. Note that we could simply prove that the conditions of a complete argument labelling (left to right) and a complete assumption labelling (right to left) are satisfied. Instead, we use existing results about the correspondence of assumption and argument semantics.

- Left to right: Let $LabAsm$ be a complete assumption labelling. Firstly note that for all $Asms \vdash s \in Ar_{ABA}$ exactly one of the three conditions in the definition of `LabAsm2LabArg` applies, so all $Asms \vdash s$ are in exactly one of $\texttt{in}(LabArg)$, $\texttt{out}(LabArg)$, or $\texttt{undec}(LabArg)$.

  By Theorem 3.4: $Asms = \textsc{in}(LabAsm)$ is a complete assumption extension with $Asms^+ = \textsc{out}(LabAsm)$ and $\mathcal{A} \setminus (Asms \cup Asms^+) = \textsc{undec}(LabAsm)$.

  By Theorem 6.1 in [CSAD15a]: $Args = \{Asms' \vdash s \mid Asms' \subseteq \textsc{in}(LabAsm)\}$ is a complete argument extension.

  By Lemma 3.12: $Args^+ = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in \textsc{out}(LabAsm)\}$ and $Ar_{ABA} \setminus (Args \cup Args^+) = \{Asms' \vdash s \mid Asms' \nsubseteq \textsc{in}(LabAsm), \nexists \alpha \in Asms' : \alpha \in \textsc{out}(LabAsm)\} = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in \textsc{undec}(LabAsm), Asms' \cap \textsc{out}(LabAsm) = \emptyset\}$.

  By Theorem 10 in [CG09]: $LabArg$ with $\texttt{in}(LabArg) = Args$, $\texttt{out}(LabArg) = Args^+$, $\texttt{undec}(LabArg) = Ar_{ABA} \setminus (Args \cup Args^+)$ is a complete argument labelling.

- Right to left: Let $LabArg = \texttt{LabAsm2LabArg}(LabAsm)$ be a complete argument labelling where $LabAsm$ is an assumption labelling. Since `LabAsm2LabArg` is injective by Lemma 3.15, $LabAsm$ is unique.

  By Theorems 9 and 11 in [CG09]: $Args = \texttt{in}(LabArg) = \{Asms' \vdash s \mid Asms' \subseteq \textsc{in}(LabAsm)\}$ is a complete argument extension with $Args^+ = \texttt{out}(LabArg) = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in \textsc{out}(LabAsm)\}$ and $Ar_{ABA} \setminus (Args \cup Args^+) = \texttt{undec}(LabArg) = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in \textsc{undec}(LabAsm), Asms' \cap \textsc{out}(LabAsm) = \emptyset\}$.

  By Theorem 6.1 in [CSAD15a]: $Asms = \{\alpha \in \mathcal{A} \mid \exists Asms' : \alpha \in Asms' \textit{ and } Asms' \vdash s \in Args\} = \textsc{in}(LabAsm)$ is a complete assumption extension.

By Lemma 3.13: $Asms^+ = \{\alpha \mid \{\alpha\} \vdash \alpha \in Args^+\} = \{\alpha \mid \alpha \in \textsc{out}(LabAsm)\} = \textsc{out}(LabAsm)$ and $\mathcal{A} \setminus (Asms \cup Asms^+) = \{\alpha \mid \{\alpha\} \vdash \alpha \notin Args, \{\alpha\} \vdash \alpha \notin Args^+\} = \{\alpha \mid \{\alpha\} \vdash \alpha \in \texttt{undec}(LabArg)\} = \{\alpha \mid \alpha \in \textsc{undec}(LabAsm)\} = \textsc{undec}(LabAsm)$.

By Theorem 3.4: $LabAsm$ is a complete assumption labelling.

□

Note that in addition to proving that every complete assumption labelling is translated into a corresponding complete argument labelling by `LabAsm2LabArg`, analogous to the correspondence between complete assumption and argument extensions [CSAD15a], Theorem 3.17 also proves that for every complete argument labelling that is the translation of some assumption labelling $LabAsm$ in terms of `LabAsm2LabArg`, $LabAsm$ is a complete assumption labelling.

Since `LabAsm2LabArg` is injective but not generally surjective (see Lemma 3.15), there may be an argument labelling $LabArg$ that is not the translation of any assumption labelling in terms of `LabAsm2LabArg`, so a natural question is whether $LabArg$ may be a complete argument labelling. The following Proposition shows that this is not the case, i.e. every complete argument labelling is the translation of some assumption labelling in terms of `LabAsm2LabArg`.

Proposition 3.18. *Let LabArg be a complete argument labelling. Then there exists a unique assumption labelling LabAsm such that* `LabAsm2LabArg`$(LabAsm) = LabArg$.

Proof. By Theorem 9 in [CG09], $Args = \texttt{in}(LabArg)$ is a complete argument extension.
By Theorem 11 in [CG09], $Args^+ = \texttt{out}(LabArg)$ and $Ar_{ABA} \setminus (Args \cup Args^+) = \texttt{undec}(LabArg)$.
By Theorem 6.1 in [CSAD15a], $Asms = \{\alpha \mid \exists Asms' : \alpha \in Asms', Asms' \vdash s \in Args\}$ is a complete assumption extension.
From Theorem 6.1 and Proposition 1 in [CSAD15a] it also follows that $Args = \{Asms' \vdash s \mid Asms' \subseteq Asms\}$. By Lemma 3.12, $Args^+ = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in Asms^+\}$, and $Ar_{ABA} \setminus (Args \cup Args^+) = \{Asms' \vdash s \mid Asms' \nsubseteq Asms, \nexists \alpha \in Asms' : \alpha \in Asms^+\}$.
By Theorem 3.4, $LabAsm$ with $\textsc{in}(LabAsm) = Asms$, $\textsc{out}(LabAsm) = Asms^+$, and $\textsc{undec}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ is a complete assumption labelling.
It follows that, $Args = \{Asms' \vdash s \mid Asms' \subseteq \textsc{in}(LabAsm)\} = \texttt{in}(LabArg)$. Furthermore, $Args^+ = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in \textsc{out}(LabAsm)\} = \texttt{out}(LabArg)$, and $Ar_{ABA} \setminus (Args \cup Args^+) = \{Asms' \vdash s \mid Asms' \nsubseteq \textsc{in}(LabAsm), \nexists \alpha \in Asms' : \alpha \in \textsc{out}(LabAsm)\} = \{Asms' \vdash s \mid \exists \alpha \in Asms' : \alpha \in \textsc{undec}(LabAsm), Asms' \cap \textsc{out}(LabAsm) = \emptyset\} = \texttt{undec}(LabArg)$.
Thus, `LabAsm2LabArg`$(LabAsm) = LabArg$.
Since `LabAsm2LabArg` is injective by Lemma 3.15, $LabAsm$ is unique. □

It follows directly from Theorem 3.17 that $LabAsm$ is a complete assumption labelling.

We now examine the translation from argument into assumption labellings in terms of $\texttt{LabArg2LabAsm}$. Theorem 3.19 below shows that the translation of a complete argument labelling yields a complete assumption labelling.

Theorem 3.19. *Let LabArg be an argument labelling. If LabArg is a complete argument labelling, then* $\texttt{LabArg2LabAsm}(\mathit{LabArg})$ *is a complete assumption labelling.*

Proof. By Theorems 9 and 11 in [CG09], $\mathit{Args} = \texttt{in}(\mathit{LabArg})$ is a complete argument extension with $\mathit{Args}^+ = \texttt{out}(\mathit{LabArg})$ and $\mathit{Ar}_{ABA} \setminus (\mathit{Args} \cup \mathit{Args}^+) = \texttt{undec}(\mathit{LabArg})$.
By Theorem 6.1 in [CSAD15a], $\mathit{Asms} = \{\alpha \mid \exists \mathit{Asms}' : \alpha \in \mathit{Asms}' \text{ and } \mathit{Asms}' \vdash s \in \texttt{in}(\mathit{LabArg})\}$ is a complete assumption extension.
By Lemma 3.13, $\mathit{Asms}^+ = \{\alpha \mid \{\alpha\} \vdash \alpha \in \mathit{Args}^+\} = \{\alpha \mid \{\alpha\} \vdash \alpha \in \texttt{out}(\mathit{LabArg})\}$ and $\mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+) = \{\alpha \mid \{\alpha\} \vdash \alpha \notin \mathit{Args}, \{\alpha\} \vdash \alpha \notin \mathit{Args}^+\} = \{\alpha \mid \{\alpha\} \vdash \alpha \in \texttt{undec}(\mathit{LabArg})\}$.
Since for an argument $\mathit{Asms}' \vdash s \in \texttt{in}(\mathit{LabArg})$ it holds that all attackers are labelled $\texttt{out}$, it follows that $\forall \alpha \in \mathit{Asms}'$: all attackers of $\{\alpha\} \vdash \alpha$ are labelled $\texttt{out}$, so by the definition of complete argument labellings $\{\alpha\} \vdash \alpha \in \texttt{in}(\mathit{LabArg})$. Thus, $\mathit{Asms} = \{\alpha \mid \{\alpha\} \vdash \alpha \in \texttt{in}(\mathit{LabArg})\}$.
By Theorem 3.4, LabAsm with $\textsc{in}(\mathit{LabAsm}) = \mathit{Asms}$, $\textsc{out}(\mathit{LabAsm}) = \mathit{Asms}^+$ and $\textsc{undec}(\mathit{LabAsm}) = \mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+)$ is a complete assumption labelling. □

Note that since $\texttt{LabArg2LabAsm}$ is surjective (see Lemma 3.16) the converse of Theorem 3.19 does not hold, i.e. a complete assumption labelling LabAsm may be the translation of some argument labelling in terms of $\texttt{LabArg2LabAsm}$ that is not a complete argument labelling, as illustrated by the following example.

Example 3.15. ABA_{10} from Example 3.14 has only one complete assumption labelling $\mathit{LabAsm}_1 = \{(\rho, \textsc{out}), (\psi, \textsc{in})\}$. The corresponding AA framework of ABA_{10} has one complete argument labelling, $\mathit{LabArg}_1 = \{(A_1, \texttt{out}), (A_2, \texttt{in}), (A_3, \texttt{out})\}$. It holds that $\texttt{LabArg2LabAsm}(\mathit{LabArg}_1) = \mathit{LabAsm}_1$, but also that for $\mathit{LabArg}_2 = \{(A_1, \texttt{out}), (A_2, \texttt{in}), (A_3, \texttt{undec})\}$, $\texttt{LabArg2LabAsm}(\mathit{LabArg}_2) = \mathit{LabAsm}_1$, where LabArg_2 is not a complete argument labelling.

However, a weaker version of the converse of Theorem 3.19 holds: every complete assumption labelling is the translation of some complete argument labelling in terms of $\texttt{LabArg2LabAsm}$.

Lemma 3.20. *Let LabAsm be a complete assumption labelling. Then there exists a complete argument labelling LabArg such that* $\texttt{LabArg2LabAsm}(\mathit{LabArg}) = \mathit{LabAsm}$.

Proof. Let $\mathit{LabArg} = \texttt{LabAsm2LabArg}(\mathit{LabAsm})$, so by Theorem 3.17 LabArg is a complete argument labelling. Now let $\mathit{LabAsm}' = \texttt{LabArg2LabAsm}(\mathit{LabArg})$, so $\textsc{in}(\mathit{LabAsm}') = \{\alpha \mid \{\alpha\} \subseteq \textsc{in}(\mathit{LabAsm})\} = \textsc{in}(\mathit{LabAsm})$, $\textsc{out}(\mathit{LabAsm}') = \{\alpha \mid \alpha \in \textsc{out}(\mathit{LabAsm})\} = \textsc{out}(\mathit{LabAsm})$, $\textsc{undec}(\mathit{LabAsm}') = \{\alpha \mid \alpha \in \textsc{undec}(\mathit{LabAsm}), \{\alpha\} \cap \textsc{out}(\mathit{LabAsm}) =$

$\emptyset\} = \textsc{undec}(LabAsm)$. Thus, $LabAsm = LabAsm'$, so there exists a complete argument labelling $LabArg$ such that $\texttt{LabArg2LabAsm}(LabArg) = LabAsm$. □

Even though there may be multiple argument labellings that are translated into the same complete assumption labelling in terms of `LabArg2LabAsm`, there are no two *complete* argument labellings that are translated into the same assumption labelling.

Lemma 3.21. *Let $LabArg_1 \neq LabArg_2$ be two complete argument labellings. Then $\texttt{LabArg2LabAsm}(LabArg_1) \neq \texttt{LabArg2LabAsm}(LabArg_2)$.*

Proof. Let $\texttt{LabArg2LabAsm}(LabArg_1) = LabAsm_1$, $\texttt{LabArg2LabAsm}(LabArg_2) = LabAsm_2$. Assume that $LabAsm_1 = LabAsm_2$. Since $LabArg_1 \neq LabArg_2$, $\exists Asms_1 \vdash s_1 \in Ar_{ABA}$ such that $LabArg_1(Asms_1 \vdash s_1) \neq LabArg_2(Asms_1 \vdash s_1)$.

- Let $Asms_1 \vdash s_1 \in \texttt{in}(LabArg_1)$, so $Asms_1 \vdash s_1 \notin \texttt{in}(LabArg_2)$. Then there exists some $Asms_2 \vdash \overline{\alpha}$ attacking $Asms_1 \vdash s_1$ where $\alpha \in Asms_1$ and $Asms_2 \vdash \overline{\alpha} \notin \texttt{out}(LabArg_2)$. However, $Asms_2 \vdash \overline{\alpha} \in \texttt{out}(LabArg_1)$ since all attackers of $Asms_1 \vdash s_1$ are labelled `out` by $LabArg_1$. Thus, $\{\alpha\} \vdash \alpha \notin \texttt{in}(LabArg_2)$ but $\{\alpha\} \vdash \alpha \in \texttt{in}(LabArg_1)$, so $\alpha \in \texttt{in}(LabAsm_1)$ but $\alpha \notin \texttt{in}(LabAsm_2)$. Contradiction.

- Let $Asms_1 \vdash s_1 \in \texttt{out}(LabArg_1)$, so $Asms_1 \vdash s_1 \notin \texttt{out}(LabArg_2)$. Then there exists some $Asms_2 \vdash \overline{\alpha}$ attacking $Asms_1 \vdash s_1$ where $\alpha \in Asms_1$ and $Asms_2 \vdash \overline{\alpha} \in \texttt{in}(LabArg_1)$. However, $Asms_2 \vdash \overline{\alpha} \notin \texttt{in}(LabArg_2)$ since no attacker of $Asms_1 \vdash s_1$ is labelled `in` by $LabArg_2$. Thus, $\{\alpha\} \vdash \alpha \in \texttt{out}(LabArg_1)$ but $\{\alpha\} \vdash \alpha \notin \texttt{out}(LabArg_2)$, so `LabArg2LabAsm` that $\alpha \in \textsc{out}(LabAsm_1)$ but $\alpha \notin \textsc{out}(LabAsm_2)$. Contradiction.

- Let $Asms_1 \vdash s_1 \in \texttt{undec}(LabArg_1)$, so $Asms_1 \vdash s_1 \notin \texttt{undec}(LabArg_2)$. Then either for all $Asms_2 \vdash \overline{\alpha}$ attacking $Asms_1 \vdash s_1$ where $\alpha \in Asms_1$ it holds that $Asms_2 \vdash \overline{\alpha} \in \texttt{out}(LabArg_2)$ or there exists some $Asms_3 \vdash \overline{\beta}$ attacking $Asms_1 \vdash s_1$ where $\beta \in Asms_1$ and $Asms_3 \vdash \overline{\beta} \in \texttt{in}(LabArg_2)$. In the first case for all $\{\alpha\} \vdash \alpha$, $\{\alpha\} \vdash \alpha \in \texttt{in}(LabArg_2)$ but some $\{\alpha\} \vdash \alpha \in \texttt{undec}(LabArg_1)$ since there exists an attacker $Asms_2 \vdash \overline{\alpha}$ of $Asms_1 \vdash s_1$ such that $Asms_2 \vdash \overline{\alpha} \notin \texttt{out}(LabArg_1)$. It follows that $\alpha \in \textsc{undec}(LabAsm_1)$ but $\alpha \notin \textsc{undec}(LabAsm_2)$. Contradiction. In the second case, $\{\beta\} \vdash \beta \in \texttt{out}(LabArg_2)$ but $\{\beta\} \vdash \beta \notin \texttt{out}(LabArg_1)$ since no attacker of $Asms_1 \vdash s_1$ is labelled `in` by $LabArg_1$. Thus, $\beta \in \textsc{out}(LabAsm_2)$ but $\beta \notin \textsc{out}(LabAsm_1)$. Contradiction.

□

Lemmas 3.20 and 3.21 imply that every complete assumption labelling is the translation of a *unique* complete argument labelling in terms of `LabArg2LabAsm`.

Corollary 3.22. *Let $LabAsm$ be a complete assumption labelling. Then there exists a unique complete argument labelling $LabArg$ such that $\texttt{LabArg2LabAsm}(LabArg) = LabAsm$.*

From Theorems 3.17 and 3.19, Proposition 3.18, and Lemmas 3.20 and 3.21, it follows that there is a one-to-one correspondence between complete argument and assumption labellings in terms of both `LabArg2LabAsm` and `LabAsm2LabArg`. Thus, when restricting `LabArg2LabAsm` and `LabAsm2LabArg` to complete argument and assumption labellings, they are bijective functions as well as the inverse of one another (see [CSAD15a] for the analogous result about complete argument and assumption extensions).

Corollary 3.23. *Let $\mathscr{L}_{AsmComp}$ be the set of all complete assumption labellings of $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ } \rangle$ and $\mathscr{L}_{ArgComp}$ the set of all complete argument labellings of $\langle Ar_{ABA}, Att_{ABA} \rangle$. Let*

- $\texttt{LabArg2LabAsm}' : \mathscr{L}_{ArgComp} \to \mathscr{L}_{AsmComp}$ *such that* $\forall LabArg \in \mathscr{L}_{ArgComp}$: $\texttt{LabArg2LabAsm}'(LabArg) = \texttt{LabArg2LabAsm}(LabArg)$*, and*
- $\texttt{LabAsm2LabArg}' : \mathscr{L}_{AsmComp} \to \mathscr{L}_{ArgComp}$ *such that* $\forall LabAsm \in \mathscr{L}_{AsmComp}$: $\texttt{LabAsm2LabArg}'(LabAsm) = \texttt{LabAsm2LabArg}(LabAsm)$.

$\texttt{LabArg2LabAsm}'$ *and* $\texttt{LabAsm2LabArg}'$ *are bijective functions and each other's inverses.*

3.4.3 Grounded, Preferred, Ideal, and Stable Semantics

Due to existing correspondence results between grounded, preferred, ideal, and stable argument labellings and extensions [CG09, Cam11], argument and assumption extensions [DMT07, Ton12, CSAD15a], and assumption extensions and labellings (see Section 3.2.3), the one-to-one correspondence between grounded, preferred, ideal, and stable assumption and argument labellings can be proven in a similar way as for complete assumption and argument labellings.

Theorem 3.24 states the relationship between a given grounded, preferred, ideal, and stable assumption labelling and the respective argument labelling in terms of `LabAsm2LabArg`.

Theorem 3.24. *Let LabAsm be an assumption labelling. LabAsm is a grounded / preferred / ideal / stable assumption labelling if and only if* $\texttt{LabAsm2LabArg}(LabAsm)$ *is a grounded / preferred / ideal / stable argument labelling.*

Proof. Analogous to the proof of Theorem 3.17 but using Theorem 3.5 instead of Theorem 3.4, Theorem 6.2 / 6.3 / 6.4 / 6.5 in [CSAD15a] instead of Theorem 6.1 in [CSAD15a], the analogues of Theorems 10 and 11 in [CG09] for the grounded / preferred / stable semantics (only informally given in [CG09]) instead of Theorems 9, 10, and 11 in [CG09], and Theorem 3.7 in [Cam11] for the ideal semantics instead of Theorems 9, 10, and 11 in [CG09]. □

Theorem 3.25 states the relationship between a given grounded, preferred, ideal, and stable argument labelling and the respective assumption labelling in terms of `LabArg2LabAsm`.

Theorem 3.25. *Let LabArg be an argument labelling. If LabArg is a grounded / preferred / ideal / stable argument labelling, then* $\texttt{LabArg2LabAsm}(LabArg)$ *is a grounded / preferred / ideal / stable assumption labelling.*

Proof. Analogous to the proof of Theorem 3.19 but using Theorem 3.5 instead of Theorem 3.4, Theorem 6.2 / 6.3 / 6.4 / 6.5 in [CSAD15a] instead of Theorem 6.1 in [CSAD15a], the analogues of Theorems 9 and 11 in [CG09] for the grounded / preferred / stable semantics (only informally given in [CG09]) instead of Theorems 9 and 11 in [CG09], and Theorem 3.7 in [Cam11] for the ideal semantics instead of Theorems 9 and 11 in [CG09]. □

Note that, analogous to the complete semantics (see Theorem 3.19), the converse of Theorem 3.25 does not hold. A counter-example is ABA_{10} from Example 3.15 whose only grounded, preferred, ideal, and stable assumption labelling is $LabAsm_1$, which is the translation of the argument labelling $LabArg_2$ in terms of `LabArg2LabAsm`, but $LabArg_2$ is not a grounded, preferred, ideal, or stable argument labelling.

Due to the one-to-one correspondence between complete assumption and argument labellings (see Corollary 3.23), it is straightforward that there is also a one-to-one correspondence between grounded, preferred, ideal, and stable argument and assumption labellings in terms of `LabAsm2LabArg` and `LabArg2LabAsm`.

3.4.4 Semi-Stable Semantics

In contrast to the grounded, preferred, ideal, and stable semantics, semi-stable assumption and argument extensions are not in a one-to-one correspondence [CSAD15a]. Since semi-stable assumption labellings correspond to semi-stable assumption extensions (Theorem 3.5) and semi-stable argument labellings to semi-stable argument extensions [CG09], it follows that there is no one-to-one correspondence between semi-stable assumption and argument labellings in terms of `LabAsm2LabArg` and `LabArg2LabAsm`. However, since semi-stable assumption and argument labellings are complete labellings, the translation of a semi-stable assumption labelling in terms of `LabAsm2LabArg` is of course a complete argument labelling and the translation of a semi-stable argument labelling in terms of `LabArg2LabAsm` is a complete assumption labelling.

The following example illustrates an ABA framework where all semi-stable argument labellings are translated into semi-stable assumption labellings by `LabArg2LabAsm`, but not all semi-stable assumption labellings are translated into semi-stable argument labellings by `LabAsm2LabArg`.

Example 3.16. Let ABA_{11} be the following ABA framework:

$$\begin{aligned}
&\mathcal{L} = \{\rho, \psi, \chi, x\},\\
&\mathcal{R} = \{x \leftarrow \psi, \chi\},\\
&\mathcal{A} = \{\rho, \psi, \chi\},\\
&\overline{\rho} = \psi,\ \overline{\psi} = \rho,\ \overline{\chi} = \chi.
\end{aligned}$$

ABA_{11} has three complete assumption labellings: $LabAsm_1$ labels all assumptions as UNDEC, and $LabAsm_2$ and $LabAsm_3$ are as illustrated in the ABA graphs in Figure 3.11. Both $LabAsm_2$ and $LabAsm_3$ are semi-stable assumption labellings of ABA_{11}.

The corresponding AA framework of ABA_{11} is shown in Figure 3.12, along with two of its complete argument labellings $LabArg_2$ and $LabArg_3$. The third complete argument labelling $LabArg_1$ labels all arguments as `undec`. Only $LabArg_2$ is a semi-stable argument labelling. Thus, `LabArg2LabAsm` translates all semi-stable argument labellings into semi-stable assumption labellings, namely `LabArg2LabAsm`$(LabArg_2) = LabAsm_2$, but `LabAsm2LabArg` does not translate all semi-stable assumption labellings into semi-stable argument labellings since `LabAsm2LabArg`$(LabAsm_3) = LabArg_3$.

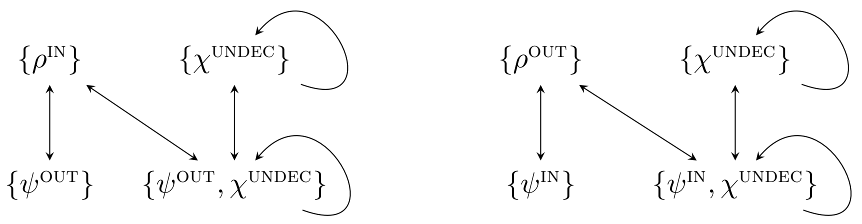


Figure 3.11: The ABA graph of ABA_{11} with two of its complete assumption labellings $LabAsm_2$ (left) and $LabAsm_3$ (right), which are both semi-stable assumption labellings (see Example 3.16).

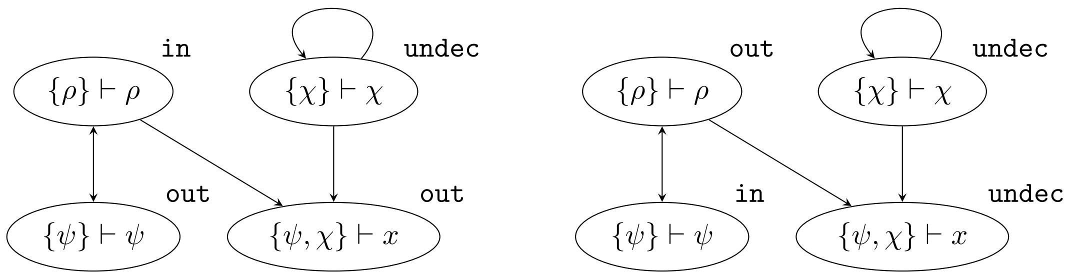


Figure 3.12: The AA graph of the corresponding AA framework of ABA_{11} with two of its complete argument labellings $LabArg_2$ (left) and $LabArg_3$ (right). Only $LabArg_2$ is a semi-stable argument labelling (see Example 3.16).

The next example illustrates an ABA framework where all semi-stable assumption labellings are translated into semi-stable argument labellings by `LabAsm2LabArg`, but not all semi-stable argument labellings are translated into semi-stable assumption labellings by `LabArg2LabAsm`.

Example 3.17. Let ABA_{12} be the following ABA framework:

$\mathcal{L} = \{\rho, \psi, \chi, \omega, x, w\}$,
$\mathcal{R} = \{x \leftarrow \psi, \chi; w \leftarrow \omega; w \leftarrow \psi\}$,
$\mathcal{A} = \{\rho, \psi, \chi, \omega\}$,
$\overline{\rho} = \psi$, $\overline{\psi} = \rho$, $\overline{\chi} = \chi$, $\overline{\omega} = w$.

ABA_{12} has three complete assumption labellings: $LabAsm_1$ labels all assumptions as UNDEC, and $LabAsm_2$ and $LabAsm_3$ are as illustrated in the ABA graphs in Figure 3.13. Only $LabAsm_3$ is a semi-stable assumption labelling.

The corresponding AA framework of ABA_{12} is shown in Figure 3.14, along with two of its complete argument labellings $LabArg_2$ and $LabArg_3$. The third complete argument labelling $LabArg_1$ labels all arguments as `undec`. Both $LabArg_2$ and $LabArg_3$ are semi-stable argument labellings. Thus, `LabAsm2LabArg` translates all semi-stable assumption labellings into semi-stable argument labellings, namely `LabAsm2LabArg`$(LabAsm_3) = LabArg_3$, but `LabArg2LabAsm` does not translate all semi-stable argument labellings into semi-stable assumption labellings since `LabArg2LabAsm`$(LabArg_2) = LabAsm_2$.

$$\begin{tikzcd}
\{\rho^{\text{IN}}\} \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \{\chi^{\text{UNDEC}}\} \arrow[loop right] \arrow[d, leftrightarrow] \\
\{\psi^{\text{OUT}}\} \arrow[d] & \{\psi^{\text{OUT}}, \chi^{\text{UNDEC}}\} \arrow[loop right] \\
\{\omega^{\text{UNDEC}}\} \arrow[loop right] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\{\rho^{\text{OUT}}\} \arrow[d, leftrightarrow] \arrow[dr, leftrightarrow] & \{\chi^{\text{UNDEC}}\} \arrow[loop right] \arrow[d, leftrightarrow] \\
\{\psi^{\text{IN}}\} \arrow[d] & \{\psi^{\text{IN}}, \chi^{\text{UNDEC}}\} \arrow[loop right] \\
\{\omega^{\text{OUT}}\} \arrow[loop right] &
\end{tikzcd}$$

Figure 3.13: The ABA graph of ABA_{12} with two of its complete assumption labellings $LabAsm_2$ (left) and $LabAsm_3$ (right). Only $LabAsm_3$ is a semi-stable assumption labelling (see Example 3.17).

Note that ABA_{11} and ABA_{12} are special cases illustrating that semi-stable assumption and argument labellings do not correspond in general. However, there are also cases where semi-stable argument and assumption labellings correspond, as demonstrated by the following example.

Example 3.18. Let ABA_{13} be the same as ABA_{12} but with $\overline{\chi} = x$. Then $LabAsm_1$ and $LabAsm_3$ are complete assumption labellings as before, but in $LabAsm_2$, χ is labelled IN rather than UNDEC, so both $LabAsm_2$ and $LabAsm_3$ are semi-stable assumption labellings. The corresponding AA framework of ABA_{13} has the same complete argument labellings $LabArg_1$ and $LabArg_3$ as the corresponding AA framework of ABA_{12}, but in $LabArg_2$ the argument $\{\chi\} \vdash \chi$ is labelled `in` rather than `undec`. Thus, $LabArg_2$ and $LabArg_3$ are semi-stable argument labellings, corresponding to the two semi-stable assumption labellings of ABA_{13} in terms of `LabAsm2LabArg` and `LabArg2LabAsm`.

3.4.5 Admissible Semantics

We have shown in Theorem 3.1 that admissible assumption extensions and labellings are in a one-to-one correspondence. Furthermore, we know that admissible assumption and

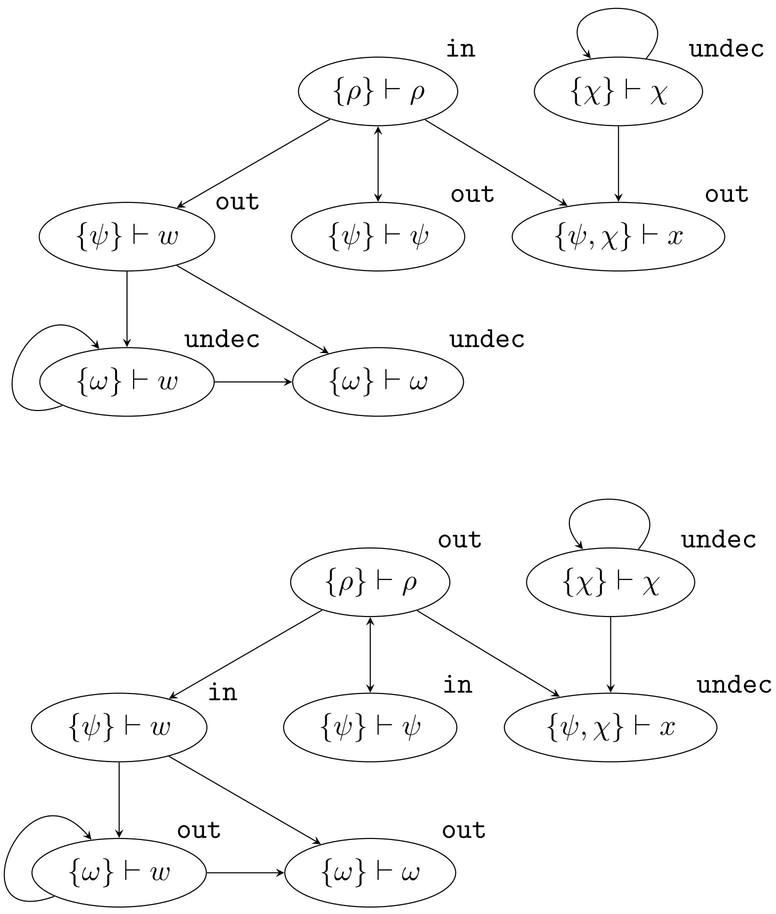


Figure 3.14: The AA graph of the corresponding AA framework of ABA_{12} with two of its complete argument labellings $LabArg_2$ (top) and $LabArg_3$ (bottom), which are both semi-stable argument labellings (see Example 3.17).

argument extensions correspond [DMT07], but this correspondence is not one-to-one as for the complete, grounded, preferred, ideal, and stable semantics, but one-to-many, as illustrated by the following example.

Example 3.19. Let ABA_{14} be the following ABA framework, illustrated as an ABA graph on the left of Figure 3.15:

$$\mathcal{L} = \{\rho, \psi, p\},$$
$$\mathcal{R} = \{p \leftarrow \psi\},$$
$$\mathcal{A} = \{\rho, \psi\},$$
$$\overline{\rho} = \psi, \overline{\psi} = \rho.$$

The admissible assumption extensions of ABA_{14} are $Asms_1 = \{\}$, $Asms_2 = \{\rho\}$ and $Asms_3 = \{\psi\}$. The corresponding AA framework, illustrated on the right of Figure 3.15, has three arguments, $A_1 : \{\rho\} \vdash \rho$, $A_2 : \{\psi\} \vdash \psi$, and $A_3 : \{\psi\} \vdash p$, and four admissible argument extensions, $Args_1 = \{\}$, $Args_2 = \{A_1\}$, $Args_3 = \{A_2\}$, and $Args_4 = \{A_2, A_3\}$. $Args_3$ and $Args_4$ both correspond to the admissible assumption extension $Asms_3$ in the

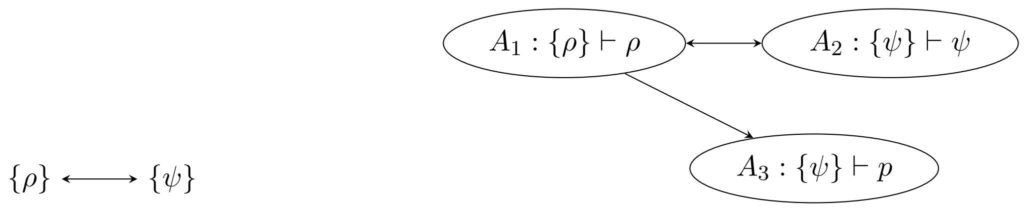

$\{\rho\}$
$\{\psi\}$
$A_1 : \{\rho\} \vdash \rho$
$A_2 : \{\psi\} \vdash \psi$
$A_3 : \{\psi\} \vdash p$

Figure 3.15: The ABA graph (left) and the AA graph (right) of ABA_{14} (see Example 3.19).

sense that $Asms_3$ is the set of all assumptions occurring in the premises of arguments in both $Args_3$ and $Args_4$ (see Section 2.2.3). Conversely, only $Args_4$ corresponds to $Asms_3$ in the sense that it is the set of all arguments whose premises are contained in $Asms_3$.

In addition, the correspondence between admissible argument extensions and labellings is one-to-many rather than one-to-one [CG09]. This implies that the correspondence between admissible assumption and argument labellings is one-to-many rather than one-to-one. Thus, only some of the correspondence results analogous to those for complete semantics hold for admissible semantics.

Theorem 3.26. *Let LabAsm be an assumption labelling. If LabAsm is an admissible assumption labelling, then* `LabAsm2LabArg`(*LabAsm*) *is an admissible argument labelling.*

Proof. Analogous to the "left to right" part of the proof of Theorem 3.17 but using Theorem 3.1 instead of Theorem 3.4, Theorem 2.2 in [DMT07] instead of Theorem 6.1 in [CSAD15a], and Theorem 21 in [CG09] instead of Theorem 10 in [CG09]. □

Example 3.20. Consider again ABA_{14} from Example 3.19 (see Figure 3.15). ABA_{14} has the same number of admissible assumption labellings and extensions, which correspond one-to-one:

- $LabAsm_1 = \{(\rho, \text{UNDEC}), (\psi, \text{UNDEC})\}$ corresponds to $Asms_1$;
- $LabAsm_2 = \{(\rho, \text{IN}), (\psi, \text{OUT})\}$ corresponds to $Asms_2$;
- $LabAsm_3 = \{(\rho, \text{OUT}), (\psi, \text{IN})\}$ corresponds to $Asms_3$.

In contrast, the corresponding AA framework of ABA_{14} has eight admissible argument labellings, even though it has only four admissible argument extensions:

- $LabArg_{11} = \{(A_1, \texttt{undec}), (A_2, \texttt{undec}), (A_3, \texttt{undec})\}$ corresponds to $Args_1$;
- $LabArg_{21} = \{(A_1, \texttt{in}), (A_2, \texttt{undec}), (A_3, \texttt{undec})\}$,
  $LabArg_{22} = \{(A_1, \texttt{in}), (A_2, \texttt{undec}), (A_3, \texttt{out})\}$,
  $LabArg_{23} = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{undec})\}$, and
  $LabArg_{24} = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{out})\}$ all correspond to $Args_2$;
- $LabArg_{31} = \{(A_1, \texttt{undec}), (A_2, \texttt{in}), (A_3, \texttt{undec})\}$, and
  $LabArg_{32} = \{(A_1, \texttt{out}), (A_2, \texttt{in}), (A_3, \texttt{undec})\}$ both correspond to $Args_3$;

- $LabArg_{41} = \{(A_1, \texttt{out}), (A_2, \texttt{in}), (A_3, \texttt{in})\}$ corresponds to $Args_4$.

The translation of each admissible assumption labelling in terms of `LabAsm2LabArg` is an admissible argument labelling, i.e.
$\texttt{LabAsm2LabArg}(LabAsm_1) = LabArg_{11}$,
$\texttt{LabAsm2LabArg}(LabAsm_2) = LabArg_{24}$, and
$\texttt{LabAsm2LabArg}(LabAsm_3) = LabArg_{41}$.

The following example shows that the converse of Theorem 3.26 does not hold.

Example 3.21. In ABA_{14} from Example 3.20, $\texttt{LabAsm2LabArg}(\{(\rho, \textsc{in}), (\psi, \textsc{undec})\}) = LabArg_{21}$, which is an admissible argument labelling, but $\{(\rho, \textsc{in}), (\psi, \textsc{undec})\}$ is not an admissible assumption labelling.

It is furthermore not the case that every admissible argument labelling is the translation of some admissible assumption labelling in terms of `LabAsm2LabArg` (i.e. the analogous result of Proposition 3.18 for the admissible semantics does not hold).

Example 3.22. Consider the admissible argument labelling $LabArg_{22}$ of ABA_{14} (see Example 3.20). There exists no admissible assumption labelling such that $LabArg_{22}$ is the translation in terms of `LabAsm2LabArg` since the arguments A_2 and A_3 have different labels even though their premises are the same.

Concerning `LabArg2LabAsm`, it is surprisingly not the case that the translation of every admissible argument labelling in terms of `LabArg2LabAsm` is an admissible assumption labelling (i.e. the analogous result of Theorem 3.19 for admissible semantics does not hold), as illustrated by the following example.

Example 3.23. Consider the admissible argument labelling $LabArg_{31}$ of ABA_{14} (see Example 3.20). $\texttt{LabArg2LabAsm}(LabArg_{31}) = \{(\rho, \textsc{undec}), (\psi, \textsc{in})\}$, which is not an admissible assumption labelling.

However, it holds that every admissible assumption labelling is the translation of some admissible argument labelling in terms of `LabArg2LabAsm`.

Proposition 3.27. *Let LabAsm be an admissible assumption labelling. Then there exists an admissible argument labelling LabArg such that* $\texttt{LabArg2LabAsm}(LabArg) = LabAsm$.

Proof. Analogous to the proof of Lemma 3.20 but using Theorem 3.26 instead of Theorem 3.17. □

As in the case of complete assumption and argument labellings, an admissible assumption labelling may also be the translation of some argument labelling in terms of `LabArg2LabAsm` that is not an admissible argument labelling.

For example, $\texttt{LabArg2LabAsm}(\{(A_1, \texttt{undec}), (A_2, \texttt{undec}), (A_3, \texttt{in})\}) = LabAsm_2$, where $LabAsm_2$ is an admissible assumption labelling, but $\{(A_1, \texttt{undec}), (A_2, \texttt{undec}), (A_3, \texttt{in})\}$ is not an admissible argument labelling (see Example 3.20).

Committed Admissible Argument Labellings

One of the reasons for the one-to-many correspondence between admissible assumption and argument labellings is the one-to-many correspondence between admissible argument extensions and labellings. This arises since admissible argument labellings pose no restriction on arguments labelled `undec`, so any argument can be labelled `undec` in an admissible argument labelling. In contrast, admissible assumption labellings and extensions are in a one-to-one correspondence since admissible assumption labellings pose restrictions on assumptions labelled UNDEC (see Section 3.2.1). We now introduce a variant of admissible argument labellings, which follows the spirit of admissible assumption labellings by restricting `undec` arguments to arguments that are not attacked by any `in`-labelled arguments.

Definition 3.13 (Committed Admissible Argument Labelling)**.** Let $\langle Ar, Att\rangle$ be an AA framework and let *LabArg* be an argument labelling of $\langle Ar, Att\rangle$. *LabArg* is a *committed admissible argument labelling* of $\langle Ar, Att\rangle$ if and only if for each argument $A \in Ar$ it holds that:

- if $LabArg(A) = \mathtt{in}$, then for each $B \in Ar$ attacking A, $LabArg(B) = \mathtt{out}$;
- if $LabArg(A) = \mathtt{out}$, then there exists some $B \in Ar$ attacking A such that $LabArg(B) = \mathtt{in}$;
- if $LabArg(A) = \mathtt{undec}$, then there exists no $B \in Ar$ attacking A such that $LabArg(B) = \mathtt{in}$.

From Definition 3.13 it follows directly that each committed admissible argument labelling is an admissible argument labelling.

Corollary 3.28. *Let $\langle Ar, Att\rangle$ be an AA framework and let LabArg be an argument labelling of $\langle Ar, Att\rangle$. If LabArg is a committed admissible argument labelling of $\langle Ar, Att\rangle$, then it is an admissible argument labelling of $\langle Ar, Att\rangle$, but not vice versa.*

Example 3.24. The AA framework $\langle Ar_{ABA_{14}}, Att_{ABA_{14}}\rangle$ (see Examples 3.19 and 3.20) has four committed admissible argument labellings, namely $LabArg_{11}$, $LabArg_{24}$, $LabArg_{32}$, and $LabArg_{41}$. The other admissible argument labellings are not committed admissible since they violate the third condition in Definition 3.13. For example, $LabArg_{21}$ is not a committed admissible argument labelling since argument A_2 is labelled `undec`, but there exists an argument attacking A_2 which is labelled `in`, namely A_1.

Differently from admissible argument labellings, committed admissible argument labellings are in a one-to-one correspondence with admissible argument extensions.

Theorem 3.29. *Let $\langle Ar, Att\rangle$ be an AA framework.*

1. *Let $Args \subseteq Ar$ be an admissible argument extension of $\langle Ar, Att\rangle$. Then LabArg with $\mathtt{in}(LabArg) = Args$, $\mathtt{out}(LabArg) = Args^+$, and $\mathtt{undec}(LabArg) = Ar \setminus (Args \cup Args^+)$ is a committed admissible argument labelling of $\langle Ar, Att\rangle$.*

2. *Let $LabArg$ be a committed admissible argument labelling of $\langle Ar, Att\rangle$. Then $Args =$ $\texttt{in}(LabArg)$ is an admissible argument extension of $\langle Ar, Att\rangle$ with $Args^+ =$ $\texttt{out}(LabArg)$, and $Ar \setminus (Args \cup Args^+) = \texttt{undec}(LabArg)$.*

Proof.

1. First note that $Args \cap Args^+ = \emptyset$ since $Args$ does not attack itself. Thus each $A \in Ar$ is either contained in $\texttt{in}(LabArg)$, $\texttt{out}(LabArg)$, or $\texttt{undec}(LabArg)$, so $LabArg$ is an argument labelling. We prove that $LabArg$ satisfies Definition 3.13.

   - Let $LabArg(A) = \texttt{in}$. Then $A \in Args$. Thus, all attackers B of A are attacked by some $C \in Args$, so $B \in Args^+$. Consequently, for each attacker B of A, $LabArg(B) = \texttt{out}$.
   - Let $LabArg(A) = \texttt{out}$. Then $A \in Args^+$. Thus, A is attacked by some $B \in Args$, and therefore there exists some B attacking A such that $LabArg(B) = \texttt{in}$.
   - Let $LabArg(A) = \texttt{undec}$. Then $A \notin Args^+$. Thus, A is not attacked by any $B \in Args$ and consequently there exists no B attacking A such that $LabArg(B) = \texttt{in}$.

2. We prove that $\texttt{in}(LabArg)$ is an admissible argument extension.

   - $\texttt{in}(LabArg)$ is conflict-free: Assume $\texttt{in}(LabArg)$ is not conflict-free. Then there exist $A, B \in \texttt{in}(LabArg)$ such that A attacks B, so B is attacked by an argument that is not labelled $\texttt{out}$. Contradiction.
   - All arguments in $\texttt{in}(LabArg)$ are defended by $\texttt{in}(LabArg)$: Let $A \in \texttt{in}(LabArg)$. Then for each attacker B of A, $LabArg(B) = \texttt{out}$ and therefore for each B there exists an attacker C such that $LabArg(C) = \texttt{in}$. Thus, each attacker of A is attacked by $\texttt{in}(LabArg)$, i.e. $\texttt{in}(LabArg)$ defends A.
   - $Args^+ = \{A \mid Args \textit{ attacks } A\} = \{A \mid \texttt{in}(LabArg) \textit{ attacks } A\}$
     $= \{A \mid A \in \texttt{out}(LabArg)\} = \texttt{out}(LabArg)$
   - $Ar \setminus (Args \cup Args^+) = \{A \mid A \notin Args, A \notin Args^+\}$
     $= \{A \mid A \notin \texttt{in}(LabArg), A \notin \texttt{out}(LabArg)\} = \{A \mid A \in \texttt{undec}(LabArg)\} =$ $\texttt{undec}(LabArg)$

□

Note that the way the sets of arguments labelled $\texttt{in}$, $\texttt{out}$, and $\texttt{undec}$ are defined in the first item of Theorem 3.29 mirrors the *Ext*2*Lab* operator of Caminada and Gabbay [CG09]. On the other hand, the second item of Theorem 3.29 extends the *Lab*2*Ext* operator in [CG09], as it not only defines an argument extension based on an argument labelling, but also the set of arguments attacked by the argument extension and the set of arguments that are neither contained in nor attacked by the argument extension.

Note also that committed admissible argument labellings are different from other variations of the admissible semantics, such as strongly admissible argument labellings (and extensions) [BG07, Cam14], which require that an accepted argument is defended by accepted arguments other than itself, and related admissible argument extensions [FT14], which require that all accepted arguments are "relevant" for defending some accepted argument.

Given this one-to-one correspondence between committed admissible argument labellings and admissible argument labellings, we now show that there is a "more refined" one-to-many correspondence between admissible assumption labellings and committed admissible argument labellings as compared to admissible argument labellings, i.e. some additional correspondence results hold. Firstly, the converse of Theorem 3.26 is satisfied for committed admissible argument labellings.

Theorem 3.30. *Let LabAsm be an assumption labelling. LabAsm is an admissible assumption labelling if and only if* `LabAsm2LabArg`(*LabAsm*) *is a committed admissible argument labelling.*

Proof. Analogous to the proof of Theorem 3.17, but using Theorem 3.1 instead of Theorem 3.4, Theorem 3.29 instead of Theorems 10 and 11 in [CG09], and Theorem 2.2 in [DMT07] instead of Theorem 6.1 in [CSAD15a]. □

Secondly, the translation of a committed admissible argument labelling in terms of `LabArg2LabAsm` is an admissible assumption labelling.

Theorem 3.31. *Let LabArg be an argument labelling. If LabArg is a committed admissible argument labelling, then* `LabArg2LabAsm`(*LabArg*) *is an admissible assumption labelling.*

Proof. Analogous to the proof of Theorem 3.19 but using Theorem 3.29 instead of Theorems 9 and 11 in [CG09], Theorem 2.2 in [DMT07] instead of Theorem 6.1 in [CSAD15a], and Theorem 3.1 instead of Theorem 3.4. □

Furthermore, Proposition 3.27 also holds for committed admissible argument labellings.

Proposition 3.32. *Let LabAsm be an admissible assumption labelling. Then there exists a committed admissible argument labelling LabArg such that* `LabArg2LabAsm`(*LabArg*) = *LabAsm.*

Proof. Analogous to the proof of Lemma 3.20 but using Theorem 3.30 instead of Theorem 3.17. □

The following example illustrates that due to the additional correspondence results, the one-to-many correspondence of admissible assumption labellings with committed admissible argument labellings is "more refined" than with admissible argument labellings.

Example 3.25. Consider again ABA_{14} from Examples 3.19, 3.20, and 3.24.
$LabAsm_2$ is the translation of only one committed admissible argument labelling in terms of `LabArg2LabAsm`, namely $LabArg_{24}$, rather than of two different admissible argument

labellings $LabArg_{23}$ and $LabArg_{24}$. Furthermore, the translations of all committed admissible argument labellings in terms of `LabArg2LabAsm` are admissible assumption labellings. In contrast, the translations of the three admissible argument labellings $LabArg_{21}$, $LabArg_{22}$, and $LabArg_{31}$ in terms of `LabArg2LabAsm` are not admissible assumption labellings.

The reason that despite the additional correspondence results there is no one-to-one correspondence between admissible assumption labellings and committed admissible argument labellings is that a committed admissible argument labelling may not be the translation of any admissible assumption labelling in terms of `LabAsm2LabArg`. For example, the committed admissible argument labelling $LabArg_{32}$ of ABA_{14} is not the translation of any admissible assumption labelling in terms of `LabAsm2LabArg` (see Examples 3.19, 3.20, and 3.24).

Note that it would also be straightforward to define a new notion of admissible assumption labellings, which corresponds more closely to admissible argument labellings. This can be achieved by deleting the restriction on assumptions labelled UNDEC from the definition of admissible assumption labellings. However, we do not examine this possible variation further since we believe that the restriction on assumptions labelled UNDEC is intuitive: it seems reasonable that any assumption attacked by accepted assumptions cannot be accepted and should thus be rejected (OUT) rather than neither accepted nor rejected (UNDEC).

3.5 Non-Flat ABA Frameworks

So far, we only considered flat ABA frameworks. In general however, ABA frameworks may not be flat, for example the instance of ABA corresponding to auto-epistemic logic [Moo85] is never flat [BDKT97]. For possibly non-flat ABA frameworks assumption extensions are defined in a slightly different way than for flat ABA frameworks: they are *closed* sets of assumptions and they are based on a more general notion of defence [BDKT97]. A set of assumptions $Asms \subseteq \mathcal{A}$

- is *closed* if and only if $Asms = \{\alpha \in \mathcal{A} \mid \exists Asms' \subseteq Asms : Asms' \vdash \alpha\}$;
- defends $\alpha \in \mathcal{A}$ if and only if $Asms$ attacks all closed sets of assumptions attacking α.

Note that in flat ABA frameworks every set of assumptions is closed since in these frameworks assumptions do not occur as the head of inference rules and therefore, the more general notion of defence coincides with the notion of defence introduced in Section 2.2.2. For flat ABA frameworks, the more general definition of assumption extensions for possibly non-flat ABA frameworks (introduced in the following sections) thus coincides with the definitions given in Section 2.2.2.

In the remainder of this chapter, and if not specified otherwise, we assume as given a possibly non-flat ABA framework $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^{-} \rangle$. Furthermore, "defence" refers to the more general notion introduced above. Note that the definition of assumption labellings (Definition 3.1) can be straightforwardly extended to non-flat ABA frameworks.

3.5.1 Admissible Semantics

We recall the definition of admissible assumption extensions for possibly non-flat ABA frameworks.

> A set of assumptions $Asms \subseteq \mathcal{A}$ is an *admissible assumption extension* if and only if $Asms$ is closed, conflict-free, and defends every $\alpha \in Asms$.

We first illustrate that admissible assumption labellings as introduced for flat ABA frameworks (Definition 3.2) do not correctly express the semantics of non-flat ABA frameworks.

Example 3.26. Let ABA_{15} be the following non-flat ABA framework:

$\mathcal{L} = \{\rho, \psi, \chi, p, x\}$,
$\mathcal{R} = \{\rho \leftarrow \chi\}$,
$\mathcal{A} = \{\rho, \psi, \chi\}$,
$\overline{\rho} = \psi$, $\overline{\psi} = p$, $\overline{\chi} = x$.

According to Definition 3.2, ABA_{15} has four admissible assumption labellings:

- $LabAsm_1 = \{(\rho, \text{UNDEC}), (\psi, \text{UNDEC}), (\chi, \text{UNDEC})\}$,
- $LabAsm_2 = \{(\rho, \text{OUT}), (\psi, \text{IN}), (\chi, \text{UNDEC})\}$,
- $LabAsm_3 = \{(\rho, \text{UNDEC}), (\psi, \text{UNDEC}), (\chi, \text{IN})\}$, and
- $LabAsm_4 = \{(\rho, \text{OUT}), (\psi, \text{IN}), (\chi, \text{IN})\}$.

However, ABA_{15} has only two admissible assumption extensions (according to the definition for possibly non-flat ABA frameworks): $Asms_1 = \{\}$, and $Asms_2 = \{\psi\}$. $Asms_1$ corresponds to $LabAsm_1$ and $Asms_2$ to $LabAsm_2$ (in terms of Theorem 3.1). The corresponding sets of assumptions (in terms of Theorem 3.1) of $LabAsm_3$ and $LabAsm_4$ are $Asms_3 = \{\chi\}$ and $Asms_4 = \{\psi, \chi\}$, respectively. Neither of them is an admissible assumption extension of ABA_{15}, since neither of them is a closed set of assumptions. Thus, $LabAsm_3$ and $LabAsm_4$ should not be admissible assumption labellings of the non-flat ABA framework ABA_{15}.

As illustrated in Example 3.26, a reason that the definition of admissible assumption labellings of flat ABA frameworks does not correctly express the semantics of non-flat ABA frameworks is that the set of IN-labelled assumptions may not be closed. A straightforward way of revising the definition of admissible assumption labellings is thus to explicitly add the condition "IN($LabAsm$) is a closed set of assumptions". However, this condition expresses a restriction on the whole set of IN-labelled assumptions, rather than on the label of a single assumption, as done by the three conditions of admissible assumption labellings.

To adhere to the structure of the conditions of admissible assumption labellings, we instead add an additional restriction to the conditions of UNDEC- and OUT-labelled assumptions, which ensures that an assumption can only be labelled UNDEC or OUT if it is not derivable from the set of IN-labelled assumptions using the inference rules. To express this new restriction, we introduce the notion of a set of assumptions *supporting* an assumption.

Definition 3.14 (Support in Non-Flat ABA)**.** Let $Asms \subseteq \mathcal{A}$ and $\alpha \in \mathcal{A}$. *$Asms$ supports α* if and only if there exists an argument $Asms' \vdash \alpha$ and $Asms' \subseteq Asms$. Equivalently, we say that α is supported by $Asms$.

The following definition extends Definition 3.2 to admissible assumption labellings of possibly non-flat ABA frameworks.

Definition 3.15 (Admissible Assumption Labelling in Non-Flat ABA)**.** Let $LabAsm$ be an assumption labelling. $LabAsm$ is an *admissible assumption labelling* if and only if for each assumption $\alpha \in \mathcal{A}$ it holds that:

- if $LabAsm(\alpha) = \text{IN}$, then for each closed set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \text{OUT}$;
- if $LabAsm(\alpha) = \text{OUT}$, then there exists a closed set of assumptions $Asms_1$ attacking α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \text{IN}$, and there exists no set of assumptions $Asms_2$ supporting α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) = \text{IN}$;
- if $LabAsm(\alpha) = \text{UNDEC}$, then for each closed set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) \neq \text{IN}$, and there exists no set of assumptions $Asms_2$ supporting α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) = \text{IN}$.

According to the revised definition, only $LabAsm_1$ and $LabAsm_2$ of ABA_{15} from Example 3.26 are admissible assumption labellings. $LabAsm_3$ and $LabAsm_4$ are not admissible assumption labellings since ρ violates the new restriction on UNDEC/OUT-labelled assumptions as ρ is supported by $\{\chi\}$ and χ is labelled IN.

Note that we also incorporated the more general notion of defence into Definition 3.15, by only considering closed sets of assumptions attacking the assumption in question.

Observation 3.33. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is an admissible assumption labelling according to Definition 3.2 if and only if it is an admissible assumption labelling according to Definition 3.15.*

The following theorem states that Definition 3.15 correctly expresses the admissible semantics of possibly non-flat ABA frameworks, i.e. that there is a one-to-one correspondence between admissible assumption extensions and labellings of possibly non-flat ABA frameworks.

Theorem 3.34.

1. *Let Asms be an admissible assumption extension. Then LabAsm with* $\text{IN}(LabAsm) = Asms$*,* $\text{OUT}(LabAsm) = Asms^+$ *and* $\text{UNDEC}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is an admissible assumption labelling.*

2. *Let LabAsm be an admissible assumption labelling. Then* $Asms = \text{IN}(LabAsm)$ *is an admissible assumption extension with* $Asms^+ = \text{OUT}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \text{UNDEC}(LabAsm)$*.*

Proof.

1. First note that $Asms \cap Asms^+ = \emptyset$ since *Asms* does not attack itself. Thus each $\alpha \in \mathcal{A}$ is either contained in $\text{IN}(LabAsm)$, in $\text{OUT}(LabAsm)$, or in $\text{UNDEC}(LabAsm)$. We prove that *LabAsm* satisfies Definition 3.15.

   - Let $LabAsm(\alpha) = \text{IN}$. Then $\alpha \in Asms$, so *Asms* defends α, i.e. for all closed sets of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that *Asms* attacks β. Thus, $\beta \in Asms^+$ and consequently $LabAsm(\beta) = \text{OUT}$. Therefore, for each closed set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $LabAsm(\beta) = \text{OUT}$.
   - Let $LabAsm(\alpha) = \text{OUT}$. Then $\alpha \in Asms^+$, so *Asms* attacks α. Since $Asms = \text{IN}(LabAsm)$ and since *Asms* is a closed set of assumptions, there exists a closed set of assumptions $Asms_1$ attacking α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \text{IN}$. Furthermore, since *Asms* is a closed set of assumptions, for all δ supported by *Asms* it holds that $\delta \in Asms$. Since $\alpha \in Asms^+$ and since $Asms \cap Asms^+ = \emptyset$, it follows that $\alpha \notin Asms$ and therefore α is not supported by *Asms*. Thus there exists no set of assumptions $Asms_2$ supporting α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) = \text{IN}$.
   - Let $LabAsm(\alpha) = \text{UNDEC}$. Then $\alpha \notin Asms$ and $\alpha \notin Asms^+$, so α is not attacked and not defended by *Asms*. Since α is not attacked by *Asms*, for each closed set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that $\beta \notin Asms$, and thus $LabAsm(\beta) \neq \text{IN}$. Furthermore, since *Asms* is a closed set of assumptions, it follows from the same reasoning as in the previous item that there exists no set of assumptions $Asms_2$ supporting α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) = \text{IN}$.

2. We first prove that $\text{IN}(LabAsm)$ is an admissible assumption extension.

   - $\text{IN}(LabAsm)$ is closed: Assume $\text{IN}(LabAsm)$ is not closed. Then There exists $\alpha \notin \text{IN}(LabAsm)$ such that $\text{IN}(LabAsm)$ supports α. Thus, $LabAsm(\alpha) = \text{OUT}$ or $LabAsm(\alpha) = \text{UNDEC}$. Contradiction since in either case there exists no set of assumptions $Asms_1$ supporting α such that for all $\gamma \in Asms_1$, $LabAsm(\gamma) = \text{IN}$.

- IN(LabAsm) is conflict-free: Assume IN(LabAsm) is not conflict-free. Then IN(LabAsm) attacks some $\alpha \in$ IN(LabAsm). By Definition 3.15, for each closed set of assumptions Asms_1 attacking α there exists some $\beta \in \mathit{Asms}_1$ such that $\mathit{LabAsm}(\beta) =$ OUT. Since IN(LabAsm) is a closed set of assumptions, there exists some $\beta \in$ IN(LabAsm) such that $\mathit{LabAsm}(\beta) =$ OUT. Contradiction.
- IN(LabAsm) defends all $\alpha \in$ IN(LabAsm): Let $\alpha \in$ IN(LabAsm). Then by Definition 3.15, for each closed set of assumptions Asms_1 attacking α there exists some $\beta \in \mathit{Asms}_1$ such that $\mathit{LabAsm}(\beta) =$ OUT. Furthermore, for each such β there exists a closed set of assumptions Asms_2 attacking β such that for all $\gamma \in \mathit{Asms}_2$, $\mathit{LabAsm}(\gamma) =$ IN so $\mathit{Asms}_2 \subseteq$ IN(LabAsm). Hence, IN(LabAsm) attacks all closed sets of assumptions attacking α.

$\mathit{Asms}^+ =$ OUT(LabAsm) and $\mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+) =$ UNDEC(LabAsm) as in the proof of Theorem 3.1.

□

3.5.2 Complete Semantics

We recall the definition of complete assumption extensions for possibly non-flat ABA frameworks.

> A set of assumptions $\mathit{Asms} \subseteq \mathcal{A}$ is a *complete assumption extension* if and only if Asms is closed, conflict-free, and consists of all assumptions it defends.

For flat ABA frameworks, complete assumption labellings are defined as admissible assumption labellings satisfying an additional condition. Analogously, we define complete assumption labellings of possibly non-flat ABA frameworks.

Definition 3.16 (Complete Assumption Labelling in Non-Flat ABA)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is a *complete assumption labelling* if and only if *LabAsm* is an admissible assumption labelling and for each assumption $\alpha \in \mathcal{A}$ it holds that:

- if $\mathit{LabAsm}(\alpha) =$ UNDEC, then there exists a closed set of assumptions Asms_3 attacking α such that for all $\delta \in \mathit{Asms}_3$, $\mathit{LabAsm}(\delta) \neq$ OUT.

Analogous to the definition of admissible assumption labellings of possibly non-flat ABA frameworks, the additional condition of complete assumption labellings only takes into account attacking sets of assumptions that are closed. Without this restriction, the definition would yield different assumption labellings.

Observation 3.35. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is a complete assumption labelling according to Definition 3.3 if and only if it is a complete assumption labelling according to Definition 3.16.*

As intended, complete assumption labellings and extensions of possibly non-flat ABA frameworks are in one-to-one correspondence.

Theorem 3.36.

1. *Let Asms be a complete assumption extension. Then LabAsm with* $\textsc{in}(LabAsm) = Asms$*,* $\textsc{out}(LabAsm) = Asms^+$ *and* $\textsc{undec}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is a complete assumption labelling.*

2. *Let LabAsm be a complete assumption labelling. Then* $Asms = \textsc{in}(LabAsm)$ *is a complete assumption extension with* $Asms^+ = \textsc{out}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \textsc{undec}(LabAsm)$*.*

Proof.

1. Since *Asms* is a complete assumption extension it is by definition also an admissible assumption extension [BDKT97]. By Theorem 3.34, *LabAsm* is an admissible assumption labelling. It remains to prove that the additional condition of complete assumption labellings is satisfied. Let $LabAsm(\alpha) = \textsc{undec}$. Then $\alpha \notin Asms$ and $\alpha \notin Asms^+$, so α is not attacked and not defended by *Asms*. Since α is not defended by *Asms*, there exists a closed set of assumptions $Asms_1$ attacking α such that $Asms_1$ is not attacked by *Asms*. Thus, for all $\gamma \in Asms_1$ it holds that $\gamma \notin Asms^+$. Consequently, $LabAsm(\gamma) \neq \textsc{out}$.

2. Since *LabAsm* is a complete assumption labelling it is by Definition 3.16 also an admissible assumption labelling. Thus, by Theorem 3.34 *Asms* is an admissible assumption extension with $Asms^+ = \textsc{out}(LabAsm)$ and $\mathcal{A} \setminus (Asms \cup Asms^+) = \textsc{undec}(LabAsm)$. It remains to prove that all assumptions defended by *Asms* are contained in *Asms*. Let α be defended by *Asms* and thus by $\textsc{in}(LabAsm)$. Then for each closed set of assumptions $Asms_1$ attacking α, $\textsc{in}(LabAsm)$ attacks $Asms_1$. Thus, for each such $Asms_1$ there exists some $\beta \in Asms_1$ which is attacked by $\textsc{in}(LabAsm)$, and therefore $LabAsm(\beta) = \textsc{out}$. Since this holds for each $Asms_1$ attacking α, $LabAsm(\alpha) = \textsc{in}$.

□

For flat ABA frameworks, we identified two equivalent variations to the definition of complete assumption labellings. One of them used the converse of each condition in the definition of a complete assumption labellings (see Lemma 3.3). Extending this alternative definition of complete assumption labellings of flat ABA frameworks with an additional condition ensuring that the set of IN-labelled assumptions is closed and considering only attacking sets of assumptions that are closed, makes it equivalent to the definition of complete assumption labellings for possibly non-flat ABA frameworks.

Proposition 3.37. *Let LabAsm be an assumption labelling. The following statements are equivalent:*

1. *LabAsm is a complete assumption labelling.*

2. LabAsm is such that for each $\alpha \in \mathcal{A}$ it holds that:

- *if there exists a set of assumptions Asms supporting α such that for all $\beta \in$ Asms, LabAsm$(\beta) = $* IN*, then LabAsm$(\alpha) = $* IN*;*
- *if for each closed set of assumptions Asms attacking α there exists some $\beta \in$ Asms such that LabAsm$(\beta) = $* OUT*, then LabAsm$(\alpha) = $* IN*;*
- *if there exists a closed set of assumptions Asms attacking α such that for all $\beta \in$ Asms, LabAsm$(\beta) = $* IN*, then LabAsm$(\alpha) = $* OUT*;*
- *if for each closed set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that LabAsm$(\beta) \neq$* IN*, and there exists a closed set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, LabAsm$(\gamma) \neq$* OUT*, then LabAsm$(\alpha) = $* UNDEC*.*

Proof. First item implies second item:

- Let α be such that there exists a set of assumptions *Asms* supporting α such that for all $\beta \in$ *Asms*, *LabAsm*$(\beta) = $ IN. If *LabAsm*$(\alpha) = $ OUT or *LabAsm*$(\alpha) = $ UNDEC, then the second or third, respectively, condition of complete assumption labellings is violated. Thus *LabAsm*$(\alpha) = $ IN since it satisfies the first condition.

- Let α be such that for each closed set of assumptions *Asms* attacking α there exists some $\beta \in$ *Asms* such that *LabAsm*$(\beta) = $ OUT. If *LabAsm*$(\alpha) = $ OUT or *LabAsm*$(\alpha) = $ UNDEC, then the second or third, respectively, condition of complete assumption labellings is violated. Thus *LabAsm*$(\alpha) = $ IN since it satisfies the first condition.

- Let α be such that there exists a closed set of assumptions *Asms* attacking α such that for all $\beta \in$ *Asms*, *LabAsm*$(\beta) = $ IN. If *LabAsm*$(\alpha) = $ IN or *LabAsm*$(\alpha) = $ UNDEC, then the first or third, respectively, condition of complete assumption labellings is violated. Thus *LabAsm*$(\alpha) = $ OUT since it satisfies the second condition.

- Let α be such that for each closed set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that *LabAsm*$(\beta) \neq$ IN, and there exists a closed set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, *LabAsm*$(\gamma) \neq$ OUT. If *LabAsm*$(\alpha) = $ IN or *LabAsm*$(\alpha) = $ OUT, then the first or second, respectively, condition of complete assumption labellings is violated. Thus *LabAsm*$(\alpha) = $ UNDEC since it satisfies the third condition.

Second item implies first item.

- Let *LabAsm*$(\alpha) = $ IN. Then for each closed set of assumptions $Asms_1$ attacking α there exists some $\beta \in Asms_1$ such that *LabAsm*$(\beta) \neq$ IN. Furthermore, it either holds that there exists a closed set of assumptions $Asms_2$ attacking α such that for all

$\gamma \in Asms_2$, $LabAsm(\gamma) = \textsc{in}$, (contradiction) or that for each closed set of assumptions $Asms_3$ attacking α there exists some $\delta \in Asms_3$ such that $LabAsm(\delta) = \textsc{out}$. Thus, the second part of the or-statement applies.

- Let $LabAsm(\alpha) = \textsc{out}$. Then there exists no set of assumptions $Asms_1$ supporting α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$. Furthermore, there exists a closed set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) \neq \textsc{out}$. Furthermore, it either holds that there exists a closed set of assumptions $Asms_3$ attacking α such that for all $\delta \in Asms_3$, $LabAsm(\delta) = \textsc{in}$, or that for each closed set of assumptions $Asms_4$ attacking α there exists some $\epsilon \in Asms_4$ such that $LabAsm(\epsilon) = \textsc{out}$ (contradiction). Thus, the first part of the or-statement applies.
- Let $LabAsm(\alpha) = \textsc{undec}$. Then there exists no set of assumptions $Asms_1$ supporting α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$. Furthermore, there exists a closed set of assumptions $Asms_2$ attacking α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) \neq \textsc{out}$. Furthermore, for each closed set of assumptions $Asms_3$ attacking α there exists some $\delta \in Asms_3$ such that $LabAsm(\delta) \neq \textsc{in}$.

□

Note that reversing the conditions in Definition 3.16 does not result in an equivalent definition of complete assumption labellings for possibly non-flat ABA frameworks. For example, the assumption labelling $LabAsm = \{(\rho, \textsc{undec}), (\psi, \textsc{in}), (\chi, \textsc{in})\}$ of ABA_{15} (see Example 3.26) satisfies the converse of each condition in Definition 3.16: for both ψ and χ the converse of the first condition applies and is satisfied, and for ρ none of the converses of the three conditions applies, so ρ trivially satisfies the converse conditions. However, $LabAsm$ is not a complete assumption labelling of ABA_{15} since ABA_{15} has no complete assumption labellings.

The other equivalent definition of complete assumption labellings for flat ABA frameworks we identified was the "if and only if" version of the first and second conditions of a complete assumption labelling of flat ABA frameworks (see Lemma 3.3). The analogue in terms of complete assumption labellings of possibly non-flat ABA frameworks does however not result in an equivalent definition. That is, an assumption labelling satisfying the following conditions

- $LabAsm(\alpha) = \textsc{in}$ if and only if for each closed set of assumptions $Asms$ attacking α there exists some $\beta \in Asms$ such that $LabAsm(\beta) = \textsc{out}$;
- $LabAsm(\alpha) = \textsc{out}$ if and only if there exists a closed set of assumptions $Asms_1$ attacking α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$, and there exists no set of assumptions $Asms_2$ supporting α such that for all $\gamma \in Asms_2$, $LabAsm(\gamma) = \textsc{in}$;

is not generally a complete assumption labelling of a possibly non-flat ABA framework, since for instance $LabAsm = \{(\rho, \textsc{undec}), (\psi, \textsc{in}), (\chi, \textsc{in})\}$ of ABA_{15} (see Example 3.26) satisfies both conditions, but $LabAsm$ is not a complete assumption labelling of ABA_{15}.

3.5.3 Grounded, Preferred, Ideal, Semi-Stable, and Stable Semantics

Originally, the grounded, preferred, and stable assumption extensions of possibly non-flat ABA frameworks were defined as specific admissible rather than complete assumption extensions. For flat ABA frameworks these two definitions are equivalent, but, as we will show in this section, for non-flat ABA frameworks they are not.

We first recall the definitions of grounded, preferred, and stable assumption extensions for possibly non-flat ABA frameworks [BDKT97]. A set of assumptions $Asms \subseteq \mathcal{A}$ is

- a *grounded assumption extension* if and only if $Asms$ is the intersection of all complete assumption extensions;[1]
- a *preferred assumption extension* if and only if $Asms$ is a maximal (w.r.t. $\subseteq$) admissible assumption extension;
- a *stable assumption extension* if and only if $Asms$ is closed, conflict-free, and for all $\alpha \in \mathcal{A}$ it holds that if $\alpha \notin Asms$, then $Asms$ attacks α.

Since ideal and semi-stable semantics have only been defined for flat ABA frameworks so far, we will investigate these semantics after dealing with the grounded, preferred, and stable semantics.

Grounded Semantics

The following example illustrates that for possibly non-flat ABA frameworks, the minimally complete assumption extensions do not generally coincide with the grounded assumption extensions.

Example 3.27. Let ABA_{16} be the following non-flat ABA framework:

$$\begin{aligned}
&\mathcal{L} = \{\rho, \psi, \chi, \omega, x\},\\
&\mathcal{R} = \{x \leftarrow \rho;\ x \leftarrow \psi;\ \chi \leftarrow\},\\
&\mathcal{A} = \{\rho, \psi, \chi, \omega\},\\
&\overline{\rho} = \psi,\ \overline{\psi} = \rho,\ \overline{\chi} = \omega,\ \overline{\omega} = x.
\end{aligned}$$

ABA_{16} has two complete assumption extensions: $Asms_1 = \{\rho, \chi\}$ and $Asms_2 = \{\psi, \chi\}$. $Asms_1$ and $Asms_2$ are both minimally complete, but the grounded assumption extension is $Asms_3 = \{\chi\}$.

In order to express the grounded semantics of possibly non-flat ABA frameworks in terms of assumption labellings, the set of IN-labelled assumptions has to be the intersection of the sets of IN-labelled assumptions of all complete assumption labellings.

Definition 3.17 (Grounded Assumption Labelling in Non-Flat ABA)**.** Let $LabAsm$ be an assumption labelling. $LabAsm$ is a *grounded assumption labelling* if and only if for all $\alpha \in \mathcal{A}$ it holds that:

[1] Note that Bondarenko et al. [BDKT97] use the term “well-founded” instead of “grounded”.

- $\mathit{LabAsm}(\alpha) = \text{IN}$ if and only if for all complete assumption labellings LabAsm', $\mathit{LabAsm}'(\alpha) = \text{IN}$;

- $\mathit{LabAsm}(\alpha) = \text{OUT}$ if and only if there exists a closed set of assumptions Asms attacking α such that for all $\beta \in \mathit{Asms}$, $\mathit{LabAsm}(\beta) = \text{IN}$.

The second condition ensures the one-to-one correspondence between grounded assumption labellings and extensions of possibly non-flat ABA frameworks.

Theorem 3.38.

1. *Let Asms be a grounded assumption extension. Then LabAsm with* $\text{IN}(\mathit{LabAsm}) = \mathit{Asms}$, $\text{OUT}(\mathit{LabAsm}) = \mathit{Asms}^+$ *and* $\text{UNDEC}(\mathit{LabAsm}) = \mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+)$ *is a grounded assumption labelling.*

2. *Let LabAsm be a grounded assumption labelling. Then* $\mathit{Asms} = \text{IN}(\mathit{LabAsm})$ *is a grounded assumption extension with* $\mathit{Asms}^+ = \text{OUT}(\mathit{LabAsm})$ *and* $\mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+) = \text{UNDEC}(\mathit{LabAsm})$.

Proof.

1. First note that since Asms is the intersection of all complete assumption labellings, which are all conflict-free, it follows that $\mathit{Asms} \cap \mathit{Asms}^+ = \emptyset$ and thus each $\alpha \in \mathcal{A}$ is either contained in $\text{IN}(\mathit{LabAsm})$, $\text{OUT}(\mathit{LabAsm})$, or $\text{UNDEC}(\mathit{LabAsm})$, so LabAsm is an assumption labelling. Furthermore, note that grounded assumption extensions are always closed, even though this is not explicitly required in their definition. Since the grounded assumption extension is a subset of every complete assumption extension, any assumption α supported by the grounded assumption extension is also supported by each complete assumption extension. Since complete assumption extensions are closed, α is thus in each complete assumption extension and consequently part of the grounded assumption extension. We prove that LabAsm satisfies Definition 3.17.

   - Left to right: Let $\mathit{LabAsm}(\alpha) = \text{IN}$. Then $\alpha \in \mathit{Asms}$. Therefore, for all complete assumption extensions Asms', $\alpha \in \mathit{Asms}'$. By Theorem 3.36, for each Asms' it holds that LabAsm' with $\text{IN}(\mathit{LabAsm}') = \mathit{Asms}'$, $\text{OUT}(\mathit{LabAsm}') = \mathit{Asms}'^+$, and $\text{UNDEC}(\mathit{LabAsm}') = \mathcal{A} \setminus (\mathit{Asms}' \cup \mathit{Asms}'^+)$ is a complete assumption labelling and there are no other complete assumption labellings. Thus, for all complete assumption labellings LabAsm', $\mathit{LabAsm}'(\alpha) = \text{IN}$.
     Right to left: Let α be such that for all complete assumption labellings LabAsm', $\mathit{LabAsm}'(\alpha) = \text{IN}$. Then by Theorem 3.36, for each LabAsm' it holds that $\mathit{Asms}' = \text{IN}(\mathit{LabAsm}')$ is a complete assumption extension and there are no other complete assumption extensions. Thus, for all complete assumption extensions Asms', $\alpha \in \mathit{Asms}'$. Therefore, $\alpha \in \mathit{Asms}$ and thus $\mathit{LabAsm}(\alpha) = \text{IN}$.

- Left to right: Let $LabAsm(\alpha) = \textsc{out}$. Then $\alpha \in Asms^+$. Thus, α is attacked by $Asms$ and thus by a set of assumptions $Asms_1$ such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$.
  Right to left: Let α be such that there exists a set of assumptions $Asms_1$ attacking α such that for all $\beta \in Asms_1$, $LabAsm(\beta) = \textsc{in}$. Then $Asms_1 \subseteq Asms$, so α is attacked by $Asms$. Therefore, $\alpha \in Asms^+$ and thus $LabAsm(\alpha) = \textsc{out}$.

2. Since $LabAsm$ is a grounded assumption labelling, it holds that for all $\alpha \in \mathcal{A}$: $LabAsm(\alpha) = \textsc{in}$ if and only if for all complete assumption labellings $LabAsm'$, $LabAsm'(\alpha) = \textsc{in}$. By Theorem 3.36, for each $LabAsm'$ it holds that $Asms' = \textsc{in}(LabAsm')$ with $Asms'^+ = \textsc{out}(LabAsm')$ and $\mathcal{A} \setminus (Asms' \cup Asms'^+) = \textsc{undec}(LabAsm')$ is a complete assumption extension and there are no other complete assumption extensions. Thus, for all $\alpha \in \mathcal{A}$: $\alpha \in Asms$ if and only if for all complete assumption extensions $Asms'$, $\alpha \in Asms'$. Therefore, $Asms$ is the intersection of all complete assumption extensions.
   $Asms^+ = \{\alpha \in \mathcal{A} \mid Asms \textit{ attacks } \alpha\} = \{\alpha \in \mathcal{A} \mid \textsc{in}(LabAsm) \textit{ attacks } \alpha\}$
   $= \{\alpha \in \mathcal{A} \mid \alpha \in \textsc{out}(LabAsm)\} = \textsc{out}(LabAsm)$
   $\mathcal{A} \setminus (Asms \cup Asms^+) = \{\alpha \in \mathcal{A} \mid \alpha \notin \textsc{in}(LabAsm), \alpha \notin \textsc{out}(LabAsm)\}$
   $= \{\alpha \in \mathcal{A} \mid \alpha \in \textsc{undec}(LabAsm)\} = \textsc{undec}(LabAsm)$

□

Based on the correspondence between grounded assumption labellings and extensions of possibly non-flat ABA frameworks and results of Bondarenko et al. [BDKT97], we prove that for flat ABA frameworks Definition 3.17 is equivalent to the definition of grounded assumption labellings for flat ABA frameworks.

Proposition 3.39. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is a grounded assumption labelling according to Definition 3.4 if and only if it is a grounded assumption labelling according to Definition 3.17.*

Proof.

- Right to left: Let $LabAsm$ be a grounded assumption labelling according to Definition 3.17. By Theorem 3.38 $Asms = \textsc{in}(LabAsm)$ is a grounded assumption extension of possibly non-flat ABA frameworks with $Asms^+ = \textsc{out}(LabAsm)$ and $\mathcal{A} \setminus (Asms \cup Asms^+) = \textsc{undec}(LabAsm)$. By Theorem 6.2 in [BDKT97], for flat ABA frameworks $Asms$ is a minimal (w.r.t. $\subseteq$) complete assumption extension, and thus $Asms$ is a grounded assumption extension as defined for flat ABA frameworks. By Theorem 3.5, $LabAsm$ is a grounded assumption labelling according to Definition 3.4.

- Left to right: Let $LabAsm$ be a grounded assumption labelling according to Definition 3.4. By Theorem 3.5 $Asms = \textsc{in}(LabAsm)$ is a grounded assumption extension as defined for flat ABA frameworks with $Asms^+ = \textsc{out}(LabAsm)$ and

$\mathcal{A} \setminus (Asms \cup Asms^+) = \text{UNDEC}(LabAsm)$, i.e. *Asms* is a minimal (w.r.t. $\subseteq$) complete assumption extension. Let $Asms'$ be the intersection of all complete assumption extensions, i.e. a grounded assumption extension of possibly non-flat ABA frameworks. Since the grounded extension of a flat ABA framework is unique, *Asms* is unique and so is $Asms'$. Thus, by Theorem 6.2 in [BDKT97] $Asms' = Asms$. Then by Theorem 3.38 *LabAsm* is a grounded assumption labelling according to Definition 3.17.

□

Preferred Semantics

The non-flat ABA framework ABA_{15} from Example 3.26 illustrates that maximally complete assumption extensions do not generally coincide with preferred assumption extensions: ABA_{15} has no complete assumption extensions, but $\{\psi\}$ is its preferred assumption extension as it is maximally admissible. We thus define preferred assumption labellings of possibly non-flat ABA frameworks as admissible, rather than complete, assumption labellings with a maximal set of IN-labelled assumptions.

Definition 3.18 (Preferred Assumption Labelling in Non-Flat ABA)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is a *preferred assumption labelling* if and only if *LabAsm* is an admissible assumption labelling and $\text{IN}(LabAsm)$ is maximal (w.r.t. $\subseteq$) among all admissible assumption labellings.

Since preferred assumption labellings of flat ABA frameworks can be equivalently defined as *admissible* assumption labellings with a maximal set of IN labelled assumptions (see Proposition 3.7) and since for flat ABA frameworks Definition 3.15 coincides with Definition 3.2 (see Observation 3.33), it follows that for flat ABA frameworks Definition 3.18 coincides with the definition of preferred assumption labellings of flat ABA frameworks.

Proposition 3.40. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is a preferred assumption labelling according to Definition 3.4 if and only if it is a preferred assumption labelling according to Definition 3.18.*

As desired, preferred assumption labellings correctly express the preferred semantics of possibly non-flat ABA frameworks.

Theorem 3.41.

1. *Let Asms be a preferred assumption extension. Then LabAsm with* $\text{IN}(LabAsm) = Asms$, $\text{OUT}(LabAsm) = Asms^+$ *and* $\text{UNDEC}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is a preferred assumption labelling.*

2. *Let LabAsm be a preferred assumption labelling. Then* $Asms = \text{IN}(LabAsm)$ *is a preferred assumption extension with* $Asms^+ = \text{OUT}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \text{UNDEC}(LabAsm)$.

Proof. Analogous to the proof of Theorem 3.5, but using the definition of admissible assumption extensions and labellings of possibly non-flat ABA frameworks instead of complete assumption extensions and labellings of flat ABA frameworks, as well as Theorem 3.34 instead of Theorem 3.4. □

Stable Semantics

Even though stable assumption extensions of possibly non-flat ABA frameworks are not defined as specific admissible or complete assumption extensions, it was shown by Bondarenko et al. [BDKT97] that stable assumption extensions are always complete assumption extensions. Therefore, we define stable assumption labellings of possibly non-flat ABA frameworks in the same way as for flat ABA frameworks, i.e. as complete assumption labellings that label no assumption as UNDEC.

Definition 3.19 (Stable Assumption Labelling in Non-Flat ABA)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is a *stable assumption labelling* if and only if *LabAsm* is a complete assumption labelling and $\text{UNDEC}(LabAsm) = \emptyset$.

From Observation 3.35 and Definition 3.19 it follows straightaway that for flat ABA frameworks Definition 3.19 is equivalent to the definition of stable assumption labellings of flat ABA frameworks.

Observation 3.42. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is a stable assumption labelling according to Definition 3.4 if and only if it is a stable assumption labelling according to Definition 3.19.*

Furthermore, there is a one-to-one correspondence between stable assumption labellings and extensions of possibly non-flat ABA frameworks.

Theorem 3.43.

1. *Let Asms be a stable assumption extension. Then LabAsm with* $\text{IN}(LabAsm) = Asms$, $\text{OUT}(LabAsm) = Asms^+$ *and* $\text{UNDEC}(LabAsm) = \mathcal{A} \setminus (Asms \cup Asms^+)$ *is a stable assumption labelling.*

2. *Let LabAsm be a stable assumption labelling. Then* $Asms = \text{IN}(LabAsm)$ *is a stable assumption extension with* $Asms^+ = \text{OUT}(LabAsm)$ *and* $\mathcal{A} \setminus (Asms \cup Asms^+) = \text{UNDEC}(LabAsm)$.

Proof.

1. By Theorem 5.5 in [BDKT97], *Asms* is a complete assumption extension. By Theorem 3.36, *LabAsm* is a complete assumption labelling. Furthermore, since for all $\alpha \in \mathcal{A}$ it holds that if $\alpha \notin Asms$, then *Asms* attacks α, it follows that $Asms \cup Asms^+ = \mathcal{A}$. Then $\text{IN}(LabAsm) \cup \text{OUT}(LabAsm) = \mathcal{A}$, so $\text{UNDEC}(LabAsm) = \emptyset$. Thus, *LabAsm* is a stable assumption labelling.

2. By definition *LabAsm* is a complete assumption labelling. By Theorem 3.36, *Asms* is a complete assumption extension. Since $\text{UNDEC}(LabAsm) = \emptyset$ it follows that for all $\alpha \in \mathcal{A}$, $\alpha \in \text{IN}(LabAsm)$ of $\alpha \in \text{OUT}(LabAsm)$. And thus $\alpha \in Asms$ or $\alpha \in Asms^+$. Thus, if $\alpha \notin Asms$, then *Asms* attacks α.

□

Ideal Semantics

Since the ideal semantics has so far only been defined in the context of flat ABA frameworks, we define ideal assumption extensions of possibly non-flat ABA frameworks. We follow the spirit of the original definition for flat ABA frameworks, where ideal assumption extensions are defined as specific admissible rather than complete assumption extensions [DMT07].

Definition 3.20 (Ideal Assumption Extension in Non-Flat ABA)**.** A set of assumptions $Asms \subseteq \mathcal{A}$ is an *ideal assumption extension* if and only if *Asms* is a maximal (w.r.t. $\subseteq$) admissible assumption extension satisfying that for all preferred assumption extensions $Asms'$, $Asms \subseteq Asms'$.

Just as for preferred assumption extensions, ideal assumption extensions of possibly non-flat ABA frameworks do not generally coincide with maximally complete assumption extensions that are a subset of each preferred assumption extension. We thus define ideal assumption labellings of possibly non-flat ABA frameworks in terms of admissible rather than complete assumption labellings.

Definition 3.21 (Ideal Assumption Labelling in Non-Flat ABA)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is an *ideal assumption labelling* if and only if *LabAsm* is an admissible assumption labelling and $\text{IN}(LabAsm)$ is maximal (w.r.t. $\subseteq$) among all admissible assumption labellings satisfying that for all preferred assumption labellings $LabAsm'$, $\text{IN}(LabAsm) \subseteq \text{IN}(LabAsm')$.

From Proposition 3.7 and Observation 3.33 it follows that for flat ABA frameworks Definition 3.21 coincides with the definition of ideal assumption labellings of flat ABA frameworks.

Proposition 3.44. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is an ideal assumption labelling according to Definition 3.4 if and only if it is an ideal assumption labelling according to Definition 3.21.*

Furthermore, as desired there is a one-to-one correspondence between ideal assumption extensions and labellings of possibly non-flat ABA frameworks.

Theorem 3.45.

1. *Let Asms be an ideal assumption extension. Then LabAsm with* $\textsc{in}(\mathit{LabAsm}) = \mathit{Asms}$*,* $\textsc{out}(\mathit{LabAsm}) = \mathit{Asms}^+$ *and* $\textsc{undec}(\mathit{LabAsm}) = \mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+)$ *is an ideal assumption labelling.*

2. *Let LabAsm be an ideal assumption labelling. Then* $\mathit{Asms} = \textsc{in}(\mathit{LabAsm})$ *is an ideal assumption extension with* $\mathit{Asms}^+ = \textsc{out}(\mathit{LabAsm})$ *and* $\mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+) = \textsc{undec}(\mathit{LabAsm})$*.*

Proof. Analogous to the proof of Theorem 3.5, but using the definition of admissible assumption extensions and labellings of possibly non-flat ABA frameworks instead of complete assumption extensions of flat ABA frameworks, as well as the definition of preferred assumption extensions and labellings of possibly non-flat ABA frameworks instead of preferred assumption extensions and labellings of flat ABA frameworks, and Theorem 3.34 instead of Theorem 3.4. □

Semi-Stable Semantics

Just like the ideal semantics, the semi-stable semantics has so far only been defined for flat ABA frameworks. Semi-stable assumption extensions are originally defined as specific complete assumption extensions, but for flat ABA frameworks they can be equivalently defined as specific admissible assumption extensions. For non-flat ABA frameworks this is not the case.

Example 3.28. Let ABA_{17} be the following non-flat ABA framework:

$$\mathcal{L} = \{\rho, \psi, \chi, \omega, p\},$$
$$\mathcal{R} = \{p \leftarrow \rho;\ p \leftarrow \chi;\ p \leftarrow \psi;\ \psi \leftarrow \rho, \chi\},$$
$$\mathcal{A} = \{\rho, \psi, \chi, \omega\},$$
$$\overline{\rho} = \psi,\ \overline{\psi} = p,\ \overline{\chi} = \psi,\ \overline{\omega} = \chi.$$

The only complete assumption extension of ABA_{17} is $\mathit{Asms}_1 = \{\}$, and thus $\mathit{Asms}_1 \cup \mathit{Asms}_1^+$ is maximal among all complete assumption extensions. In contrast, there are three admissible assumption extensions: Asms_1, $\mathit{Asms}_2 = \{\rho\}$, and $\mathit{Asms}_3 = \{\chi\}$. Among these, $\mathit{Asms}_3 \cup \mathit{Asms}_3^+$ is maximal.

One of the defining properties of semi-stable assumption extensions of flat ABA frameworks is that they are preferred assumption extensions [CSAD15a]. To retain this property, we define semi-stable assumption extensions and labellings of possibly non-flat ABA frameworks in terms of admissible rather than complete assumption extensions and labellings.

Definition 3.22 (Semi-Stable Assumption Extension in Non-Flat ABA)**.** A set of assumptions $\mathit{Asms} \subseteq \mathcal{A}$ is a *semi-stable assumption extension* if and only if Asms is an admissible assumption extension and for all admissible assumption extensions Asms', $\mathit{Asms} \cup \mathit{Asms}^+ \not\subset \mathit{Asms}' \cup \mathit{Asms}'^+$.

Definition 3.23 (Semi-Stable Assumption Labelling in Non-Flat ABA)**.** Let *LabAsm* be an assumption labelling. *LabAsm* is a *semi-stable assumption labelling* if and only if *LabAsm* is an admissible assumption labelling and $\text{UNDEC}(\mathit{LabAsm})$ is minimal (w.r.t. $\subseteq$) among all admissible assumption labellings.

By Proposition 3.7, for flat ABA frameworks Definition 3.23 coincides with the definition of semi-stable assumption labellings of flat ABA frameworks.

Proposition 3.46. *Let LabAsm be an assumption labelling of a flat ABA framework. Then LabAsm is a semi-stable assumption labelling according to Definition 3.4 if and only if it is a semi-stable assumption labelling according to Definition 3.23.*

As desired, there is a one-to-one correspondence between semi-stable assumption extensions and labellings of possibly non-flat ABA frameworks.

Theorem 3.47.

1. *Let Asms be a semi-stable assumption extension. Then LabAsm with* $\text{IN}(\mathit{LabAsm}) = \mathit{Asms}$, $\text{OUT}(\mathit{LabAsm}) = \mathit{Asms}^+$ *and* $\text{UNDEC}(\mathit{LabAsm}) = \mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+)$ *is a semi-stable assumption labelling.*

2. *Let LabAsm be a semi-stable assumption labelling. Then* $\mathit{Asms} = \text{IN}(\mathit{LabAsm})$ *is a semi-stable assumption extension with* $\mathit{Asms}^+ = \text{OUT}(\mathit{LabAsm})$ *and* $\mathcal{A} \setminus (\mathit{Asms} \cup \mathit{Asms}^+) = \text{UNDEC}(\mathit{LabAsm})$.

Proof. Analogous to the proof of Theorem 3.5, but using the definition of admissible assumption extensions and labellings of possibly non-flat ABA frameworks instead of complete assumption extensions and labellings of flat ABA frameworks, and Theorem 3.34 instead of Theorem 3.4. □

Finally, we prove that semi-stable assumption labellings of possibly non-flat ABA frameworks satisfy the property we desired, namely that they are preferred assumption labellings.

Proposition 3.48. *Let LabAsm be a semi-stable assumption labelling. Then LabAsm is a preferred assumption labelling.*

Proof. Since $\text{UNDEC}(\mathit{LabAsm})$ is minimal it follows that $\text{IN}(\mathit{LabAsm}) \cup \text{OUT}(\mathit{LabAsm})$ is maximal among all admissible assumption labellings. Assume by contradiction that there exists an admissible assumption labelling LabAsm' such that $\text{IN}(\mathit{LabAsm}) \subset \text{IN}(\mathit{LabAsm}')$. Then for all $\alpha \in \mathcal{A}$ such that $\text{IN}(\mathit{LabAsm})$ attacks α, $\text{IN}(\mathit{LabAsm}')$ also attacks α. Thus, $\text{OUT}(\mathit{LabAsm}) \subseteq \text{OUT}(\mathit{LabAsm}')$. It follows that $\text{IN}(\mathit{LabAsm}) \cup \text{OUT}(\mathit{LabAsm}) \subset \text{IN}(\mathit{LabAsm}') \cup \text{OUT}(\mathit{LabAsm}')$. Contradiction. □

3.6 Related Work

As discussed in more detail in Section 3.4, assumption labellings are closely related to argument labellings for AA frameworks [CG09]. Both assumption and argument labellings use three different labels, one indicating acceptance (IN and `in`), one indicating rejection (OUT and `out`), and one indicating neither acceptance nor rejection (UNDEC and `undec`). Using the semantics of AA frameworks in terms of argument labellings has proven useful for example for the computation of semantics [LLD13, CGVZ14, CVG15, CDG$^+$15], for studying decomposability of semantics [BBC$^+$14], for judgement aggregation [CP11], as well as for teaching the semantics of AA frameworks to novices [DS14, SD16].

In addition to AA frameworks – and now ABA frameworks – labellings have also been introduced for argumentation frameworks with necessities (AFNs). Nouioua [Nou13] shows how the semantics of AFNs can be defined in terms of labellings and how to apply the new definitions for the computation of semantics of AFNs.

Besides argument labellings for AA frameworks, which correspond to the semantics of AA frameworks in terms of argument extensions, new semantics have been defined in terms of labellings. Thimm and Kern-Isberner [TKI14] introduce *stratified labellings*, which rank arguments according to their controversiality and which are determined by combining various (traditional) argument labellings. Baroni et al. [BGL15] review some further labelling semantics defined in the literature, focussing on different interpretations and meanings of the *undecided* label.

We will see in Chapter 4, that assumption and argument labellings are furthermore related 3-valued interpretations of logic programs (see Section 2.3 for the definition). Similarly to argument and assumption labellings, 3-valued interpretation assign one of three "labels" to each literal: one indicating acceptance (`T`), one indicating rejection (`F`), and one indicating neither acceptance nor rejection (`U`).

3.7 Summary

In this chapter, we defined and studied assumption labellings of flat as well as possibly non-flat ABA frameworks for the admissible, grounded, complete, preferred, ideal, semi-stable, and stable semantics and proved that there is a one-to-one correspondence with the respective assumption extensions. We also investigated the relationship of assumption labellings of flat ABA frameworks and argument labellings of their corresponding AA frameworks, and found that grounded, complete, preferred, ideal, and stable assumption and argument labellings are in a one-to-one correspondence, whereas semi-stable assumption and argument labellings do not generally correspond. Furthermore, admissible assumption and argument labellings are in a one-to-many correspondence.

In the next chapters, we use assumption labellings to investigate the correspondence between the semantics of logic programs and ABA frameworks representing the same knowledge. Assumption labellings lend themselves for the formulation of such correspondence results since the three labels IN, OUT, and UNDEC of assumptions can be seen as duals of the three truth values `T`, `F`, and `U` in the semantics of logic programs.

Chapter 4

Logic Programs as ABA and AA Frameworks

4.1 Introduction

One of the main contributions of this thesis is to use ideas from ABA and AA for Answer Set Programming (ASP). In order to apply methods defined for ABA frameworks to logic programs, a problem encoded as a logic program must first be represented as an ABA framework. In this chapter, we recall how the *translated ABA framework* can be obtained from a logic program, which can then be used to instantiate the *translated AA framework*. We then extend existing correspondence results between the semantics of logic programs, translated ABA frameworks, and translated AA frameworks by showing a more fine-grained correspondence and dealing with logic programs that may comprise explicit negation in addition to NAF. This is aided by the novel definitions of assumption labellings from Chapter 3.

The chapter is organised as follows. In Section 4.2, we recall how to obtain a translated ABA framework from a logic program. In Section 4.3, we review existing results on the correspondence between the semantics of a logic program and the assumption extensions of the translated ABA framework, and extend and refine these results in terms of assumption labellings of the translated ABA framework. In Section 4.4, we review and extend existing results on the correspondence between the semantics of a logic program and its translated AA framework, using the correspondence results between assumption labellings of an ABA framework and argument labellings of its corresponding AA framework from Chapter 3 in combination with the results from Section 4.3. In Section 4.5, we discuss related work and in Section 4.6, we summarise the contributions of this chapter.

4.2 Existing Translations

Even though the semantics of a logic program and an ABA framework are determined in completely different ways, the two formalisms share structural features. Both represent knowledge in terms of inference rules comprising defeasible elements, i.e. elements that are true by default, as long as no contrary information can be proven to hold: NAF literals in logic programs and assumptions in ABA frameworks. Every assumption α has a contrary $\overline{\alpha} = x$, where x may also be the contrary of other assumptions. A NAF literal `not` a has a complement a, but in contrast to contraries in ABA, a is the complement of only one NAF literal (namely of `not` a). Therefore, a logic program can be seen as a special instance of an ABA framework, which means that every logic program can be encoded in an ABA framework.

We use the approach of Bondarenko et al. [BDKT97] for translating a logic program into an ABA framework, where the clauses of a logic program form the set of ABA rules and NAF literals are used as assumptions in ABA.

Definition 4.1 (Translated ABA Framework)**.** Let $\mathcal{P}$ be a logic program. $ABA_{\mathcal{P}} = \langle \mathcal{L}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}}, \mathcal{A}_{\mathcal{P}}, \overline{} \rangle$ is the *translated ABA framework* of $\mathcal{P}$ where:

- $\mathcal{R}_{\mathcal{P}} = \mathcal{P}$;

- $\mathcal{A}_{\mathcal{P}} = NAF_{Lit_{\mathcal{P}}}$;
- for every $\texttt{not}\ l \in \mathcal{A}_{\mathcal{P}}$: $\overline{\texttt{not}\ l} = l$;
- $\mathcal{L}_{\mathcal{P}} = Lit_{\mathcal{P}}\ \cup\ NAF_{Lit_{\mathcal{P}}}$.

Note that translated ABA frameworks are always flat since NAF literals do not occur in the head of clauses of a logic program. Thus, every translated ABA framework has a corresponding AA framework, which represents the same information as the underlying logic program.

Definition 4.2 (Translated AA Framework)**.** Let $\mathcal{P}$ be a logic program and let $ABA_{\mathcal{P}}$ be the translated ABA framework of $\mathcal{P}$. The *translated AA framework* of $\mathcal{P}$, denoted $AA_{\mathcal{P}} = \langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$, is the corresponding AA framework of $ABA_{\mathcal{P}}$.

4.3 Semantics of Logic Programs and ABA Frameworks

Since a problem encoded as a logic program can also be expressed in terms of the translated ABA framework, we now investigate the relation between the semantics of logic programs and their translated ABA frameworks.

4.3.1 Existing Results

In early work on ABA frameworks, various correspondence results between assumption extensions and the semantics of logic programs without explicit negation were proven [BTK93, BDKT97]. These correspondence results can be summarised as follows[1].

> Let $\mathcal{P}$ be a logic program with no explicitly negated atoms, $ABA_{\mathcal{P}}$ the translated ABA framework of $\mathcal{P}$, and $Asms \subseteq \mathcal{A}_{\mathcal{P}}$.
>
> 1. $Asms$ is a complete assumption extension if and only if $\mathcal{P} \cup Asms$ is a stationary expansion [Prz91a] of $\mathcal{P}$.
> 2. $Asms$ is a complete assumption extension if and only if $\mathcal{P} \cup Asms$ is a complete scenario [Dun91] of $\mathcal{P}$.
> 3. $Asms$ is a grounded assumption extension if and only if $\{k \mid Asms' \vdash k, Asms' \subseteq Asms\}$ is a well-founded model of $\mathcal{P}$.
> 4. $Asms$ is a preferred assumption extension if and only if $\mathcal{P} \cup Asms$ is a preferred extension [Dun91] of $\mathcal{P}$.
> 5. $Asms$ is a preferred assumption extension if and only if $\{k \mid Asms' \vdash k, Asms' \subseteq Asms\}$ is a 3-valued M-stable model[2] of $\mathcal{P}$.

[1]Note that these results use the notation of 3-valued models of a logic program as a single set, as explained in Section 2.3.3.

[2]The original result is in terms of partial stable models of [SZ90], which were later called (3-valued) M-stable models [Sac95].

6. $S \subseteq \mathcal{HB}_{\mathcal{P}}$ is a stable model of $\mathcal{P}$ if and only if there exists a stable assumption extension *Asms* and $S = \{a \in \mathcal{HB}_{\mathcal{P}} \mid Asms' \vdash a, Asms' \subseteq Asms\}$.

Example 4.1. Let $\mathcal{P}_2$ be the logic program $\{r \leftarrow \texttt{not}\ r;\ q \leftarrow \texttt{not}\ p\}$. The translated ABA framework $ABA_{\mathcal{P}_2}$ has three assumptions: $\texttt{not}\ p$, $\texttt{not}\ q$, and $\texttt{not}\ r$. The grounded extension of $ABA_{\mathcal{P}_2}$ is $Asms = \{\texttt{not}\ p\}$. Then by the third point above, $\mathcal{M} = \{q, \texttt{not}\ p\}$ is the well-founded model of $\mathcal{P}_2$.

Note that the three correspondence results regarding model-theoretic semantics of logic programs (points 3., 5., and 6.) are stated in terms of the *conclusions* of arguments constructable from an assumption extension, rather than in terms of the *assumptions* in the assumption extension. In the following sections, we give *direct* correspondence results between models of a logic program and assumptions of the translated ABA framework.

Furthermore, the results regarding model-theoretic semantics of logic programs only show how a corresponding model can be derived from a given assumption extension. Even though this can be used to reconstruct an assumption extension from a given model of a logic program, it is less straightforward. We will provide an explicit mapping from models of a logic program into assumption labellings of the translated ABA framework.

4.3.2 Translating between Assumption Labellings and 3-Valued Interpretations

We will see in the following sections that when expressing the semantics of an ABA framework in terms of assumption *labellings*, there is a straightforward correspondence between atoms with truth values `T`, `F`, and `U` in a model of a logic program and assumptions labelled IN, OUT, and UNDEC in a complete assumption labelling of the translated ABA framework.

For this purpose, we first define a translations `LabAsm2Mod` from assumption labellings into 3-valued interpretations and a translation `Mod2LabAsm` from 3-valued interpretations into assumption labellings. Throughout this section, and if not stated otherwise, we assume as given a logic program $\mathcal{P}$ and its translated ABA framework $ABA_{\mathcal{P}} = \langle \mathcal{L}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}}, \mathcal{A}_{\mathcal{P}}, ^{-} \rangle$.

Definition 4.3 (Mapping an Assumption Labelling into a 3-Valued Interpretation). `LabAsm2Mod` maps an assumption labelling *LabAsm* of $ABA_{\mathcal{P}}$ into a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ such that:

- $\mathcal{T} = \sim \textsc{out}(LabAsm)$;
- $\mathcal{F} = \sim \textsc{in}(LabAsm)$;
- $\mathcal{U} = \sim \textsc{undec}(LabAsm)$.

Instead of defining the interpretation of a logic program in terms of the conclusions of arguments whose premises are labelled IN, as in the previously reviewed existing works (see points 3., 5., 6. in Section 4.3.1), we use a direct mapping from labels of assumptions into truth values of literals.

We also define a mapping for the opposite direction, i.e. from 3-valued interpretations into assumption labellings.

Definition 4.4 (Mapping a 3-Valued Interpretation into an Assumption Labelling)**.** `Mod2LabAsm` maps a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ into an assumption labelling $LabAsm$ of $ABA_{\mathcal{P}}$ such that:

- $\text{IN}(LabAsm) = \sim\mathcal{F}$;
- $\text{OUT}(LabAsm) = \sim\mathcal{T}$;
- $\text{UNDEC}(LabAsm) = \sim\mathcal{U}$.

It is easy to see that `LabAsm2Mod` and `Mod2LabAsm` are bijective functions and each other's inverses since
1) $\texttt{LabAsm2Mod}(\texttt{Mod2LabAsm}(\langle \mathcal{T}, \mathcal{F} \rangle)) = \langle \mathcal{T}, \mathcal{F} \rangle$ and
2) $\texttt{Mod2LabAsm}(\texttt{LabAsm2Mod}(LabAsm)) = LabAsm$.

Example 4.2. Let $LabAsm = \{(\texttt{not}\ p, \text{UNDEC}), (\texttt{not}\ q, \text{OUT}), (\texttt{not}\ r, \text{IN})\}$ be an assumption labelling (which is not a complete assumption labelling) of the translated ABA framework $ABA_{\mathcal{P}_2}$ from Example 4.1. $\texttt{LabAsm2Mod}(LabAsm)$ yields the 3-valued interpretation $\langle \{q\}, \{r\} \rangle$ of $\mathcal{P}_2$. Furthermore, $\texttt{Mod2LabAsm}(\texttt{LabAsm2Mod}(LabAsm)) = \{(\texttt{not}\ p, \text{UNDEC}), (\texttt{not}\ q, \text{OUT}), (\texttt{not}\ r, \text{IN})\} = LabAsm$.

Combining `LabAsm2Mod` with the conditions of *complete* assumption labellings (see Definitions 3.3 and 3.2), we can characterise the 3-valued interpretation obtained by `LabAsm2Mod` in terms of the *conclusions* of arguments whose premises have certain labels. For example, as stated in Definition 4.3, `LabAsm2Mod` defines the set $\mathcal{T}$ as consisting of the corresponding literals of assumptions labelled OUT. According to the conditions of complete assumption labellings, an assumption $\texttt{not}\ l$ is labelled OUT if and only if some argument attacking this assumption is such that all its premises are labelled IN, where the conclusion of the attacking argument is l. Thus, given a complete assumption labelling, the set $\mathcal{T}$ consists of all classical literals l that are conclusions of arguments whose premises are all labelled IN. Similar considerations apply to $\mathcal{F}$ and $\mathcal{U}$.

Proposition 4.1. *Let LabAsm be a complete assumption labelling of $ABA_{\mathcal{P}}$. Then* $\texttt{LabAsm2Mod}(LabAsm)$ *is equivalent to $\langle \mathcal{T}, \mathcal{F} \rangle$ with:*

- $\mathcal{T} = \{l \in Lit_{\mathcal{P}} \mid \exists Asms \vdash l : Asms \subseteq \text{IN}(LabAsm)\}$;
- $\mathcal{F} = \{l \in Lit_{\mathcal{P}} \mid \forall Asms \vdash l : Asms \cap \text{OUT}(LabAsm) \neq \emptyset\}$;

- $\mathcal{U} = \{l \in Lit_{\mathcal{P}} \mid \exists Asms \vdash l : Asms \cap \text{OUT}(LabAsm) = \emptyset, \forall Asms \vdash l : Asms \nsubseteq \text{IN}(LabAsm)\}$.

Proof. Let $\mathcal{T} = \{l \in Lit_{\mathcal{P}} \mid \exists Asms \vdash l : Asms \subseteq \text{IN}(LabAsm)\}$. For every $l \in \mathcal{T}$ it holds that since $Asms \vdash l$ attacks $\texttt{not}\ l$, by the third item of Theorem 3.3 it follows that $\texttt{not}\ l \in \text{OUT}(LabAsm)$. Thus, $\mathcal{T} = \{l \in Lit_{\mathcal{P}} \mid \texttt{not}\ l \in \text{OUT}(LabAsm)\} =\sim \text{OUT}(LabAsm)$. Using the same reasoning, we can show that $\mathcal{F} = \{l \in Lit_{\mathcal{P}} \mid \forall Asms \vdash l : Asms \cap \text{OUT}(LabAsm) \neq \emptyset\} =\sim \text{IN}(LabAsm)$ and $\mathcal{U} = \{l \in Lit_{\mathcal{P}} \mid \exists Asms \vdash l : Asms \cap \text{OUT}(LabAsm) = \emptyset, \forall Asms \vdash l : Asms \nsubseteq \text{IN}(LabAsm)\} =\sim \text{UNDEC}(LabAsm)$. □

This characterisation refines the translation used in existing results (see Section 4.3.1), where the 3-valued interpretation is given in terms of the conclusions of arguments whose premises are all contained in the corresponding assumption extension.

Note that even though the mapping given in Proposition 4.1 may not be the same as `LabAsm2Mod` if *LabAsm* is not a complete assumption labelling, the mapping is well-defined for any assumption labelling as it ensures that every literal has exactly one truth value.

Example 4.3. Let $LabAsm = \{(\texttt{not}\ p, \text{UNDEC}), (\texttt{not}\ q, \text{OUT}), (\texttt{not}\ r, \text{IN})\}$ be the assumption labelling of $ABA_{\mathcal{P}_2}$ from Example 4.2, so $\texttt{LabAsm2Mod}(LabAsm) = \langle\{q\}, \{r\}\rangle$. In contrast, the mapping from Proposition 4.1 yields the completely different 3-valued interpretation $\langle\{r\}, \{p\}\rangle$.

4.3.3 Semantic Correspondence between Assumption Labellings and 3-Valued Interpretations

Having defined mappings between 3-valued interpretations of a logic program and assumption labellings of the translated ABA framework, we now prove that these translations preserve the semantics. We start by proving that there is a one-to-one correspondence between 3-valued stable models and complete assumption labellings in terms of `LabAsm2Mod` and `Mod2LabAsm` when considering logic programs without explicitly negated atoms.

Theorem 4.2. *Let $\mathcal{P}$ be a logic program without explicitly negated atoms, $ABA_{\mathcal{P}}$ the translated ABA framework of $\mathcal{P}$, and LabAsm an assumption labelling of $ABA_{\mathcal{P}}$. If LabAsm is a complete assumption labelling of $ABA_{\mathcal{P}}$, then $\langle\mathcal{T}, \mathcal{F}\rangle = \texttt{LabAsm2Mod}(LabAsm)$ is a 3-valued stable model of $\mathcal{P}$.*

Proof.

- By Theorem 3.4: $\text{IN}(LabAsm)$ is a complete assumption extension.
- By Theorem 5.9 in [BDKT97]: $\mathcal{P} \cup \text{IN}(LabAsm)$ is a complete scenario of $\mathcal{P}$ as defined by [Dun91].
- By Corollary 4.16(i) in [BLMM92]: $E = \mathcal{P} \cup \text{IN}(LabAsm) \cup \{\neg\texttt{not}\ a \mid a \in \mathcal{HB}_{\mathcal{P}}, \mathcal{P} \cup \text{IN}(LabAsm) \vdash_{MP} a\}$ is a stationary expansion of $\mathcal{P}$ as defined by [Prz91a].

- By Theorem 3.1 in [Prz91a]: $M = \{a \mid E \vdash_{MP} a\} \cup \{\texttt{not}\ a \mid E \vdash_{MP} \texttt{not}\ a\}$ is a partial stable model of $\mathcal{P}$ as defined in [Prz91b].
- $\{a \mid E \vdash_{MP} a\}$ is equivalent to $\{a \mid \mathcal{P} \cup \textsc{in}(\mathit{LabAsm}) \vdash_{MP} a\}$ and $\{\texttt{not}\ a \mid E \vdash_{MP} \texttt{not}\ a\}$ to $\{\texttt{not}\ a \mid \mathcal{P} \cup \textsc{in}(\mathit{LabAsm}) \vdash_{MP} \texttt{not}\ a\}$. Thus, $M = \{a \mid \mathcal{P} \cup \textsc{in}(\mathit{LabAsm}) \vdash_{MP} a\} \cup \textsc{in}(\mathit{LabAsm})$.
- By Proposition 3.2 in [Prz91b]: $M = \langle \mathcal{T}, \mathcal{F} \rangle$ with $\mathcal{T} = \{a \mid \mathcal{P} \cup \textsc{in}(\mathit{LabAsm}) \vdash_{MP} a\}$ and $\mathcal{F} = \sim \textsc{in}(\mathit{LabAsm})$ is a 3-valued stable model of $\mathcal{P}$.
- By definition of arguments and complete assumption labellings: $\mathcal{T} = \{a \mid AP \vdash a, AP \subseteq \textsc{in}(\mathit{LabAsm})\} = \sim \{\texttt{not}\ a \mid AP \vdash a, AP \subseteq \textsc{in}(\mathit{LabAsm})\} = \sim \textsc{out}(\mathit{LabAsm})$.
- By definition of 3-valued model and Definition 4.1: $\mathcal{U} = \mathcal{HB_P} \setminus (\mathcal{T} \cup \mathcal{F}) = \mathcal{HB_P} \setminus (\sim \textsc{out}(\mathit{LabAsm}) \cup \sim \textsc{in}(\mathit{LabAsm})) = \sim \mathcal{A_P} \setminus \sim (\textsc{out}(\mathit{LabAsm}) \cup \textsc{in}(\mathit{LabAsm})) = \sim \textsc{undec}(\mathit{LabAsm})$.

□

Theorem 4.3. *Let $\mathcal{P}$ be a logic program without explicitly negated atoms, $ABA_\mathcal{P}$ the translated ABA framework of $\mathcal{P}$, and $\langle \mathcal{T}, \mathcal{F} \rangle$ a 3-valued interpretation of $\mathcal{P}$. If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued stable model of $\mathcal{P}$, then $\mathit{LabAsm} = \texttt{Mod2LabAsm}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is a complete assumption labelling of $ABA_\mathcal{P}$.*

Proof.

- By definition of 3-valued stable model: $M = \mathcal{T} \cup \sim \mathcal{F}$ is a partial stable model of $\mathcal{P}$ as defined in [Prz91b].
- By Theorem 3.1 in [Prz91a] $E = \mathcal{P} \cup \{\texttt{not}\ a \mid \texttt{not}\ a \in M\} \cup \{\neg\texttt{not}\ a \mid a \in M\}$ is a stationary expansion of $\mathcal{P}$.
- By Corollary 4.16(ii) in [BLMM92]: $\mathcal{P} \cup (E \cap \sim \mathcal{HB_P})$ is a complete scenario of $\mathcal{P}$.
- By Theorem 5.9 in [BDKT97]: $E \cap \sim \mathcal{HB_P}$ is a complete assumption extension.
- This can be simplified to $\{\texttt{not}\ a \mid \texttt{not}\ a \in M\}$ is a complete assumption extension, and further to $\sim \mathcal{F}$ is a complete assumption extension.
- By Theorem 3.4: $\textsc{in}(\mathit{LabAsm}) = \sim \mathcal{F}$.
- By Theorem 3.4: $\textsc{out}(\mathit{LabAsm}) = \{\texttt{not}\ a \mid AP \vdash a, AP \subseteq \sim \mathcal{F}\} = \{\texttt{not}\ a \mid \mathcal{P} \cup \sim \mathcal{F} \vdash_{MP} a\} = \{\texttt{not}\ a \mid \mathcal{P} \cup \{\texttt{not}\ b \mid \texttt{not}\ b \in M\} \vdash_{MP} a\} = \{\texttt{not}\ a \mid a \in \mathcal{T}\} = \sim \mathcal{T}$.
- By Theorem 3.4: $\textsc{undec}(\mathit{LabAsm}) = \mathcal{A_P} \setminus (\textsc{in}(\mathit{LabAsm}) \cup \textsc{out}(\mathit{LabAsm})) = \sim \mathcal{HB_P} \setminus (\sim \mathcal{F} \cup \sim \mathcal{T}) = \sim \mathcal{U}$.

□

Example 4.4. The only 3-valued stable model of $\mathcal{P}_2$ (see Example 4.1) is $\langle\{q\},\{p\}\rangle$, so $\mathcal{U} = \{r\}$. Applying $\texttt{Mod2LabAsm}$ yields the only complete assumption labelling of $ABA_{\mathcal{P}_2}$, namely $\{(\texttt{not}\ p, \textsc{in}), (\texttt{not}\ q, \textsc{out}), (\texttt{not}\ r, \textsc{undec})\}$. Conversely, applying $\texttt{LabAsm2Mod}$ to this complete assumption labelling yields the 3-valued stable model.

Since $\texttt{LabAsm2Mod}$ and $\texttt{Mod2LabAsm}$ are each other's inverses, Theorems 4.2 and 4.3 can be combined to yield the following results.

Corollary 4.4. *Let $\mathcal{P}$ be a logic program without explicitly negated atoms, $ABA_{\mathcal{P}}$ the translated ABA framework of $\mathcal{P}$, and LabAsm an assumption labelling of $ABA_{\mathcal{P}}$. LabAsm is a complete assumption labelling of $ABA_{\mathcal{P}}$ if and only if $\langle\mathcal{T},\mathcal{F}\rangle = \texttt{LabAsm2Mod}(LabAsm)$ is a 3-valued stable model of $\mathcal{P}$.*

Corollary 4.5. *Let $\mathcal{P}$ be a logic program without explicitly negated atoms, $ABA_{\mathcal{P}}$ the translated ABA framework of $\mathcal{P}$, and $\langle\mathcal{T},\mathcal{F}\rangle$ a 3-valued interpretation of $\mathcal{P}$. $\langle\mathcal{T},\mathcal{F}\rangle$ is a 3-valued stable model of $\mathcal{P}$ if and only if $LabAsm = \texttt{Mod2LabAsm}(\langle\mathcal{T},\mathcal{F}\rangle)$ is a complete assumption labelling of $ABA_{\mathcal{P}}$.*

For logic programs that may comprise explicitly negated atoms, the correspondence between 3-valued stable models and complete assumption labellings is not in general one-to-one. That is, every 3-valued stable model corresponds to a complete assumption labelling of the translated ABA framework, but not vice versa. More precisely, 3-valued stable models correspond to complete assumption labellings where the set of $\textsc{out}$-labelled assumptions does not comprise assumptions of the form $\texttt{not}\ a$ and $\texttt{not}\ \neg a$ (this follows directly from Corollaries 4.4 and 4.5 and the definition of 3-valued stable models of logic programs with explicitly negated atoms as reviewed in Section 2.3.4).

Corollary 4.6. *Let LabAsm be an assumption labelling of $ABA_{\mathcal{P}}$. LabAsm is a complete assumption labelling of $ABA_{\mathcal{P}}$ such that*

$$\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$$

if and only if $\langle\mathcal{T},\mathcal{F}\rangle = \texttt{LabAsm2Mod}(LabAsm)$ is a 3-valued stable model of $\mathcal{P}$.

Corollary 4.7. *Let $\langle\mathcal{T},\mathcal{F}\rangle$ be a 3-valued interpretation of $\mathcal{P}$. $\langle\mathcal{T},\mathcal{F}\rangle$ is a 3-valued stable model of $\mathcal{P}$ if and only if $LabAsm = \texttt{Mod2LabAsm}(\langle\mathcal{T},\mathcal{F}\rangle)$ is a complete assumption labelling of $ABA_{\mathcal{P}}$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.*

Example 4.5. Let $\mathcal{P}_3$ be the following logic program:

$$\begin{aligned} \{\ p &\leftarrow \texttt{not}\ q; \\ \neg p &\leftarrow \texttt{not}\ q; \\ q &\leftarrow \texttt{not}\ p\ \} \end{aligned}$$

The translated ABA framework $ABA_{\mathcal{P}_3}$ has three complete assumption labellings:

- $LabAsm_1 = \{(\texttt{not}\ p, \textsc{undec}), (\texttt{not}\ \neg p, \textsc{undec}), (\texttt{not}\ q, \textsc{undec}), (\texttt{not}\ \neg q, \textsc{in})\}$,
- $LabAsm_2 = \{(\texttt{not}\ p, \textsc{in}), (\texttt{not}\ \neg p, \textsc{in}), (\texttt{not}\ q, \textsc{out}), (\texttt{not}\ \neg q, \textsc{in})\}$, and
- $LabAsm_3 = \{(\texttt{not}\ p, \textsc{out}), (\texttt{not}\ \neg p, \textsc{out}), (\texttt{not}\ q, \textsc{in}), (\texttt{not}\ \neg q, \textsc{in})\}$.

The translated logic program $\mathcal{P}_3'$ has three 3-valued stable models: $\langle\{\}, \{q'\}\rangle$, $\langle\{q\}, \{p, p', q'\}\rangle$, and $\langle\{p, p'\}, \{q, q'\}\rangle$. In contrast, $\mathcal{P}_3$ has only *two* 3-valued stable models, namely $\langle\{\}, \{\neg q\}\rangle$ and $\langle\{q\}, \{p, \neg p, \neg q\}\rangle$ since the corresponding model $\langle\{p, \neg p\}, \{q, \neg q\}\rangle$ of $\langle\{p, p'\}, \{q, q'\}\rangle$ comprises p and $\neg p$ in $\mathcal{T}$. As stated in the two corollaries, only $LabAsm_1$ and $LabAsm_2$ correspond to the 3-valued stable models of $\mathcal{P}_3$, since in $LabAsm_3$ both $\texttt{not}\ p$ and $\texttt{not}\ \neg p$ are labelled $\textsc{out}$.

Based on the correspondence results between 3-valued stable models and complete assumption labellings, we move on to prove correspondence between well-founded, 3-valued M-stable, ideal, 3-valued L-stable, and stable models and grounded, preferred, ideal, semi-stable and stable assumption labellings, respectively. The proof requires the following lemma (and corollary), stating that if the set of assumptions labelled $\textsc{in}$ by some complete assumption labelling is a subset of the set of assumptions labelled $\textsc{in}$ by some other complete assumption labelling, then the set of assumption labelled $\textsc{out}$ by the former is also a subset of the set of assumptions labelled $\textsc{out}$ by the latter.

Lemma 4.8. *Let $\langle\mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ }\rangle$ be an ABA framework and let $LabAsm_1$ and $LabAsm_2$ be complete assumption labellings of $\langle\mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ }\rangle$. Then $\textsc{in}(LabAsm_1) \subseteq \textsc{in}(LabAsm_2)$ if and only if $\textsc{out}(LabAsm_1) \subseteq \textsc{out}(LabAsm_2)$.*

Proof. Left to right: Assume that $\textsc{in}(LabAsm_1) \subseteq \textsc{in}(LabAsm_2)$. Let $\alpha \in \textsc{out}(LabAsm_1)$. Then, by the definition of a complete assumption labelling (Definition 3.3) there exists an ABA argument $Asms \vdash \overline{\alpha}$ with $Asms \subseteq \textsc{in}(LabAsm_1)$. Since $\textsc{in}(LabAsm_1) \subseteq \textsc{in}(LabAsm_2)$ it follows that $Asms \subseteq \textsc{in}(LabAsm_2)$. So by Theorem 3.3 (point 3, item 2), $\alpha \in \textsc{out}(LabAsm_2)$.
Right to left: Assume that $\textsc{out}(LabAsm_1) \subseteq \textsc{out}(LabAsm_2)$. Let $\alpha \in \textsc{in}(LabAsm_1)$. Then, by the definition of a complete assumption labelling (Definition 3.3) it holds that each ABA argument $Asms \vdash \overline{\alpha}$ has $Asms \cap \textsc{out}(LabAsm_1) \neq \emptyset$. Since $\textsc{out}(LabAsm_1) \subseteq \textsc{out}(LabAsm_2)$ it follows that $Asms \cap \textsc{out}(LabAsm_2) \neq \emptyset$. So by Theorem 3.3 (point 3, item 1), $\alpha \in \textsc{in}(LabAsm_2)$. □

Since $\textsc{in}(LabAsm_1) \subset \textsc{in}(LabAsm_2)$ if and only if $\textsc{in}(LabAsm_1) \subseteq \textsc{in}(LabAsm_2)$ and $\textsc{in}(LabAsm_2) \nsubseteq \textsc{in}(LabAsm_1)$, the next corollary follows straightaway from Lemma 4.8.

Corollary 4.9. *Let $\langle\mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ }\rangle$ be an ABA framework and let $LabAsm_1$ and $LabAsm_2$ be complete assumption labellings of $\langle\mathcal{L}, \mathcal{R}, \mathcal{A}, \bar{\ }\rangle$. It holds that $\textsc{in}(LabAsm_1) \subset \textsc{in}(LabAsm_2)$ if and only if $\textsc{out}(LabAsm_1) \subset \textsc{out}(LabAsm_2)$.*

Using these results, we now prove the correspondence of the other semantics of a logic program and its translated ABA framework in terms of `LabAsm2Mod` and `Mod2LabAsm`. We straightaway consider logic programs that may comprise explicitly negated atoms.

Theorem 4.10. *Let LabAsm be an assumption labelling of $ABA_{\mathcal{P}}$. LabAsm is a grounded / preferred / ideal / semi-stable / stable assumption labelling of $ABA_{\mathcal{P}}$ such that*

$$\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$$

if and only if $\langle \mathcal{T}, \mathcal{F} \rangle = \texttt{LabAsm2Mod}(LabAsm)$ is a well-founded / 3-valued M-stable / ideal / 3-valued L-stable / (2-valued) stable model of $\mathcal{P}$.

Proof.

- Grounded and well-founded:
  Left to right: If $LabAsm$ is a grounded assumption labelling such that $\forall a \in \mathcal{HB}_{\mathcal{P}}$: $\texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$, then $\textsc{in}(LabAsm)$ is minimal among all complete assumption labellings. By Corollary 4.9, $\textsc{out}(LabAsm)$ is minimal among all complete assumption labellings. Since by Corollary 3.6 the grounded assumption labelling is unique, $\textsc{in}(LabAsm) \cup \textsc{out}(LabAsm)$ is minimal among all complete assumption labellings. By the Definition of `LabAsm2Mod` and Corollary 4.6 $\mathcal{T} \cup \mathcal{F}$ is minimal among all 3-valued stable models, or equivalently $\mathcal{U}$ is maximal among all 3-valued stable models, so $\langle \mathcal{T}, \mathcal{F} \rangle$ is a well-founded model.
  Right to left: If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a well-founded stable model, then $\mathcal{T} \cup \mathcal{F}$ is minimal among all 3-valued stable models, so by the Definition of `Mod2LabAsm` and Corollary 4.6 $\textsc{in}(LabAsm) \cup \textsc{out}(LabAsm)$ is minimal among all complete assumption labellings. If $\textsc{in}(LabAsm)$ is not minimal among all complete assumption labellings, i.e. there exists $LabAsm_1$ with $\textsc{in}(LabAsm_1) \subset \textsc{in}(LabAsm)$, then by Corollary 4.8 $\textsc{out}(LabAsm_1) \subset \textsc{out}(LabAsm)$, so $\textsc{in}(LabAsm) \cup \textsc{out}(LabAsm)$ is not minimal among all complete assumption labellings. Contradiction. Thus, $\textsc{in}(LabAsm)$ is minimal among all complete assumption labellings, so $LabAsm$ is a grounded assumption labelling. By Corollary 4.6, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.

- Preferred and 3-valued M-stable:
  If $LabAsm$ is a preferred assumption labelling such that $\forall a \in \mathcal{HB}_{\mathcal{P}}$: $\texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$, then $\textsc{in}(LabAsm)$ is maximal among all complete assumption labellings. By Corollary 4.9, $\textsc{out}(LabAsm)$ is maximal among all complete assumption labellings. By the Definition of `LabAsm2Mod` and Corollary 4.6 both $\mathcal{T}$ and $\mathcal{F}$ are maximal among all 3-valued stable models, so $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued M-stable model.
  Right to left: If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued M-stable model, then $\mathcal{T}$ and $\mathcal{F}$ are maximal among all 3-valued stable models, so by the Definition of `Mod2LabAsm` and Corollary 4.6 $\textsc{in}(LabAsm)$ and $\textsc{out}(LabAsm)$ are maximal among all complete assumption labellings. Thus, $LabAsm$ is a preferred assumption labelling. By Corollary 4.6, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.

- Ideal:
  Left to right: If $LabAsm$ is an ideal assumption labelling such that $\forall a \in \mathcal{HB}_{\mathcal{P}}$: $\texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$, then $\textsc{in}(LabAsm)$ is maximal among all complete assumption labellings satisfying that for all preferred assumption labellings $LabAsm'$, $\textsc{in}(LabAsm) \subseteq \textsc{in}(LabAsm')$. By Corollary 4.9, $\textsc{out}(LabAsm)$ is maximal among all complete assumption labellings satisfying that for all preferred assumption labellings $LabAsm'$, $\textsc{out}(LabAsm) \subseteq \textsc{out}(LabAsm')$. By the Definition of `LabAsm2Mod`, Corollary 4.6, and the second item of this proof, $\mathcal{T}$ is maximal among all 3-valued stable models satisfying that for all 3-valued M-stable models $\langle \mathcal{T}_M, \mathcal{F}_M \rangle$, $\mathcal{T} \subseteq \mathcal{T}_M$. Thus, $\langle \mathcal{T}, \mathcal{F} \rangle$ is an ideal model.
  Right to left: If $\langle \mathcal{T}, \mathcal{F} \rangle$ is an ideal model, then $\mathcal{T}$ is maximal among all 3-valued stable models satisfying that for all 3-valued M-stable models $\langle \mathcal{T}_M, \mathcal{F}_M \rangle$, $\mathcal{T} \subseteq \mathcal{T}_M$. By the Definition of `Mod2LabAsm`, Corollary 4.6, and the second item of this proof, $\textsc{out}(LabAsm)$ is maximal among all complete assumption labellings satisfying that for all preferred assumption labellings $LabAsm'$, $\textsc{out}(LabAsm) \subseteq \textsc{out}(LabAsm')$, and by Corollary 4.9,$\textsc{in}(LabAsm)$ is maximal among all complete assumption labellings satisfying that for all preferred assumption labellings $LabAsm'$, $\textsc{in}(LabAsm) \subseteq \textsc{in}(LabAsm')$. Thus, $LabAsm$ is an ideal assumption labelling. By Corollary 4.6, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.

- Semi-stable and 3-valued L-stable:
  Left to right: If $LabAsm$ is a semi-stable assumption labelling such that $\forall a \in \mathcal{HB}_{\mathcal{P}}$: $\texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$, then $\textsc{undec}(LabAsm)$ is minimal among all complete assumption labellings. By the Definition of `LabAsm2Mod` and Corollary 4.6, $\mathcal{U}$ is minimal among all 3-valued stable models, so $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued L-stable model.
  Right to left: If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued L-stable model, then $\nexists \langle \mathcal{T}_1, \mathcal{F}_1 \rangle$ which is a 3-valued stable model such that $\mathcal{U}_1 \subset \mathcal{U}$. By the Definition of `Mod2LabAsm` and Corollary 4.6, there exists no $LabAsm'$ which is a complete assumption labelling such that $\textsc{undec}(LabAsm') \subset \textsc{undec}(LabAsm)$. Thus, $LabAsm$ is a semi-stable assumption labelling. By Corollary 4.6, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.

- Stable:
  Left to right: If $LabAsm$ is a stable assumption labelling such that $\forall a \in \mathcal{HB}_{\mathcal{P}}$: $\texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$, then $\textsc{undec}(LabAsm) = \emptyset$. By the Definition of `LabAsm2Mod` and Corollary 4.6, $\mathcal{U} = \emptyset$ and $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued stable model, so $\langle \mathcal{T}, \mathcal{F} \rangle$ is a (2-valued) stable model.
  Right to left: If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a (2-valued) stable model, then $\mathcal{U} = \emptyset$. By the Definition of `Mod2LabAsm` and Corollary 4.6, $\textsc{undec}(LabAsm) = \emptyset$ and $LabAsm$ is a complete assumption labelling. Thus, $LabAsm$ is a stable assumption labelling and by Corollary 4.6, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.

□

Theorem 4.11. *Let $\langle \mathcal{T}, \mathcal{F} \rangle$ be a 3-valued interpretation of $\mathcal{P}$. $\langle \mathcal{T}, \mathcal{F} \rangle$ is a well-founded / 3-valued M-stable / ideal / 3-valued L-stable / (2-valued) stable model of $\mathcal{P}$ if and only if $LabAsm = \texttt{Mod2LabAsm}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is a grounded / preferred / ideal / semi-stable / stable assumption labelling of $ABA_{\mathcal{P}}$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$.*

Proof. Follows from Theorem 4.10 and the fact that `LabAsm2Mod` and `Mod2LabAsm` are each other's inverses. □

Example 4.6. Let $\mathcal{P}_4$ be the following logic program:

$$\begin{aligned} \{\ p &\leftarrow \texttt{not}\ q; \\ \neg p &\leftarrow \texttt{not}\ q; \\ q &\leftarrow \texttt{not}\ p; \\ r &\leftarrow \texttt{not}\ r; \\ r &\leftarrow \texttt{not}\ \neg p\ \} \end{aligned}$$

The translated ABA framework $ABA_{\mathcal{P}_4}$ has three complete assumption labellings:

- $LabAsm_1 = \{(\texttt{not}\ p, \textsc{undec}), (\texttt{not}\ \neg p, \textsc{undec}), (\texttt{not}\ q, \textsc{undec}), (\texttt{not}\ \neg q, \textsc{in}), (\texttt{not}\ r, \textsc{undec}), (\texttt{not}\ \neg r, \textsc{in})\}$,
- $LabAsm_2 = \{(\texttt{not}\ p, \textsc{out}), (\texttt{not}\ \neg p, \textsc{out}), (\texttt{not}\ q, \textsc{in}), (\texttt{not}\ \neg q, \textsc{in}), (\texttt{not}\ r, \textsc{undec}), (\texttt{not}\ \neg r, \textsc{in})\}$, and
- $LabAsm_3 = \{(\texttt{not}\ p, \textsc{in}), (\texttt{not}\ \neg p, \textsc{in}), (\texttt{not}\ q, \textsc{out}), (\texttt{not}\ \neg q, \textsc{in}), (\texttt{not}\ r, \textsc{out}), (\texttt{not}\ \neg r, \textsc{in})\}$.

$LabAsm_1$ is the grounded assumption labelling, $LabAsm_2$ and $LabAsm_3$ are preferred assumption labellings, and only $LabAsm_3$ is a stable assumption labelling. By Theorems 4.10 and 4.11, we deduce that the well-founded model of $\mathcal{P}_4$ is $\texttt{LabAsm2Mod}(LabAsm_1) = \langle \{\}, \{\neg q, \neg r\} \rangle$, and the only 3-valued M-stable and only 2-valued stable model is $\texttt{LabAsm2Mod}(LabAsm_3) = \langle \{q, r\}, \{p, \neg p, \neg q, \neg r\} \rangle$.

Similar to the results for 3-valued stable models and complete assumption labellings, the condition $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(LabAsm) \vee \texttt{not}\ \neg a \notin \textsc{out}(LabAsm)$ in Theorems 4.10 and 4.11 can be neglected when dealing with a logic program without explicitly negated atoms.

4.4 Semantics of Logic Programs and AA Frameworks

In this section, we review and extend semantic correspondence results between logic programs and their translated AA frameworks (as defined in Section 4.2).

4.4.1 Existing Results

In his seminal work on AA frameworks, Dung [Dun95b] introduces a translation from a logic program without explicitly negated literals into an AA framework, which yields exactly $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ as defined here in terms of the translated ABA framework. Dung then proves the following semantic correspondences.

> Let $\mathcal{P}$ be a logic program without explicitly negated atoms, $AA_{\mathcal{P}} = \langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ the translated AA framework of $\mathcal{P}$, and $\mathcal{M} \subseteq \mathcal{HB}_{\mathcal{P}} \cup NAF_{\mathcal{HB}_{\mathcal{P}}}$.
>
> 1. $\mathcal{M}$ is a (2-valued) stable model of $\mathcal{P}$ if and only if there exists a stable argument extension $Args$ of $AA_{\mathcal{P}}$ such that $\mathcal{M} = \{k \mid \exists Asms \vdash k \in Args\}$.
> 2. If $\mathcal{M}$ is a (2-valued) stable model of $\mathcal{P}$, then $Args = \{Asms \vdash k \in Ar_{\mathcal{P}} \mid Asms \subseteq \mathcal{M}\}$ is a stable argument extension of $AA_{\mathcal{P}}$.
> 3. $\mathcal{M}$ is the well-founded model of $\mathcal{P}$ if and only if $Args$ is the grounded argument extension of $AA_{\mathcal{P}}$ and $\mathcal{M} = \{k \mid \exists Asms \vdash k \in Args\}$.

Example 4.7. Consider again the logic program $\mathcal{P}_2$ from Example 4.2. The translated AA framework $AA_{\mathcal{P}_2}$ has two arguments in addition to the three assumption-arguments: $A_1 : \{\texttt{not}\ p\} \vdash \texttt{not}\ p$, $A_2 : \{\texttt{not}\ q\} \vdash \texttt{not}\ q$, $A_3 : \{\texttt{not}\ r\} \vdash \texttt{not}\ r$, $A_4 : \{\texttt{not}\ r\} \vdash r$, and $A_5 : \{\texttt{not}\ p\} \vdash q$. A_4 attacks itself and A_3, and A_5 attacks A_2. The grounded argument extension of $AA_{\mathcal{P}_2}$ is $Args = \{A_1, A_5\}$. Then the set of conclusions of arguments in $Args$ is $\{\texttt{not}\ p, q\}$, which is the well-founded model of $\mathcal{P}_2$ (see Example 4.2).

Wu et al. [WCG09] investigate the relation between the semantics of logic programs, in particular 3-valued stable models, and complete argument labellings of AA frameworks. They give a translation from a logic program into an AA framework, which amounts to $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$, and define mappings between 3-valued interpretations and argument labellings, which we recall using a simplified but equivalent notation.

> Let $\mathcal{P}$ be a logic program without explicitly negated atoms and $AA_{\mathcal{P}} = \langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ the translated AA framework of $\mathcal{P}$.
>
> 1. Let $LabArg$ be an argument labelling of $AA_{\mathcal{P}}$. $\texttt{LabArg2Mod}_{Wu}$ maps $LabArg$ into a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ such that:
>   - $\mathcal{T} = \{a \in \mathcal{HB}_{\mathcal{P}} \mid \exists Asms \vdash a \in \texttt{in}(LabArg)\}$;
>   - $\mathcal{F} = \{a \in \mathcal{HB}_{\mathcal{P}} \mid \forall Asms \vdash a : Asms \vdash a \in \texttt{out}(LabArg)\}$;
>   - $\mathcal{U} = \{a \in \mathcal{HB}_{\mathcal{P}} \mid \nexists Asms \vdash a \in \texttt{in}(LabArg), \exists Asms' \vdash a \in \texttt{undec}(LabArg)\}$.
> 2. Let $\langle \mathcal{T}, \mathcal{F} \rangle$ be a 3-valued interpretation of $\mathcal{P}$. $\texttt{Mod2LabArg}_{Wu}$ maps $\langle \mathcal{T}, \mathcal{F} \rangle$ into an argument labelling $LabArg$ of $AA_{\mathcal{P}}$ such that:

- $\texttt{in}(LabArg) = \{Asms \vdash k \mid$ *for all attackers* $Asms' \vdash k'$ *of* $Asms \vdash k : k' \in \mathcal{F}\}$;
- $\texttt{out}(LabArg) = \{Asms \vdash k \mid$ *there exists an attacker* $Asms' \vdash k'$ *of* $Asms \vdash k : k' \in \mathcal{T}\}$;
- $\texttt{undec}(LabArg) = \{Asms \vdash k \mid$ *there exists an attacker* $Asms' \vdash k'$ *of* $Asms \vdash k : k' \notin \mathcal{F}$, *for all attackers* $Asms'' \vdash k''$ *of* $Asms \vdash k : k'' \notin \mathcal{T}\}$.

Note that the translation from arguments labelled in into $\mathcal{T}$ in $\texttt{LabArg2Mod}_{Wu}$ mirrors the mapping from stable/grounded argument extensions into stable/well-founded models by Dung.

Wu et al. prove that there is a one-to-one correspondence between 3-valued stable models and complete argument labellings in terms of $\texttt{LabArg2Mod}_{Wu}$ and $\texttt{Mod2LabArg}_{Wu}$.

> Let $\mathcal{P}$ be a logic program without explicitly negated atoms and $AA_\mathcal{P}$ the translated AA framework.
>
> 1. If $LabArg$ is a complete argument labelling of $AA_\mathcal{P}$, then $\texttt{LabArg2Mod}_{Wu}(LabArg)$ is a 3-valued stable model of $\mathcal{P}$.
> 2. If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued stable model of $\mathcal{P}$, then $\texttt{Mod2LabArg}_{Wu}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is a complete argument labelling of $AA_\mathcal{P}$.

Wu et al. also note that for complete argument labellings and 3-valued stable models $\texttt{LabArg2Mod}_{Wu}$ and $\texttt{Mod2LabArg}_{Wu}$ are bijective functions and each other's inverses.

Example 4.8. The only 3-valued stable model of $\mathcal{P}_2$ is $\langle \{q\}, \{p\} \rangle$ with $\mathcal{U} = \{r\}$. Applying $\texttt{Mod2LabArg}_{Wu}$, yields the only complete argument labelling of $AA_{\mathcal{P}_2}$, where A_1 and A_5 are labelled in since they have no attackers, A_2 is labelled out since it is attacked by $A_5 : \{\texttt{not } p\} \vdash q$ and $q \in \mathcal{T}$, and A_3 and A_4 are labelled undec since they are only attacked by $A_4 : \{\texttt{not } r\} \vdash r$ and $r \notin \mathcal{F}$ and $r \notin \mathcal{T}$.
Conversely, when applying $\texttt{LabArg2Mod}_{Wu}$, we consider the two arguments with conclusions in $\mathcal{HB}_\mathcal{P}$, namely A_4 and A_5. Since $A_4 \in \texttt{undec}(LabArg)$, it follows that $r \in \mathcal{U}$; since $A_5 \in \texttt{in}(LabArg)$, it follows that $q \in \mathcal{T}$; and since there exists no argument with conclusion p, it follows that $p \in \texttt{out}(LabArg)$ since it is satisfied that all arguments with conclusion p are labelled out. This yields the 3-valued stable model $\langle \{q\}, \{p\} \rangle$.

Caminada et al. [CSAD15b] introduce a mapping from argument labellings into *conclusion labellings* of a translated AA framework and from conclusion labellings back to argument labellings. They then prove that the conclusion labellings obtained from complete, grounded, preferred, and stable argument labellings of the translated AA framework coincide, respectively, with the 3-valued stable, well-founded, 3-valued M-stable, and (2-valued) stable models of the underlying logic program. Since the mappings between argument labellings and conclusion labellings of arguments mirror $\texttt{LabArg2Mod}_{Wu}$ and

$\texttt{Mod2LabArg}_{Wu}$, the correspondence results by Caminada et al. directly extend the results by Wu et al.

> Let $\mathcal{P}$ be a logic program without explicitly negated atoms and $AA_{\mathcal{P}}$ the translated AA framework.
>
> 1. If $LabArg$ is a grounded / preferred / stable argument labelling of $AA_{\mathcal{P}}$, then $\texttt{LabArg2Mod}_{Wu}(LabArg)$ is a well-founded / 3-valued M-stable / (2-valued) stable model of $\mathcal{P}$.
> 2. If $\langle \mathcal{T}, \mathcal{F} \rangle$ is a well-founded / 3-valued M-stable / (2-valued) stable model of $\mathcal{P}$, then $\texttt{Mod2LabArg}_{Wu}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is a grounded / preferred / stable argument labelling of $AA_{\mathcal{P}}$.

Caminada et al. also point out (in terms of conclusion labellings) that an analogous correspondence does not hold between 3-valued L-stable models of a logic program and semi-stable argument labellings of the translated AA framework.

4.4.2 Deriving Translations between Argument Labellings and 3-Valued Interpretations

In Section 4.3.2, we defined mappings between 3-valued interpretations of a logic program and assumption labellings of the translated ABA framework and in Section 3.4.1 between assumption labellings of an ABA framework and argument labellings of the corresponding AA framework. Since mapping a logic program into an ABA framework and then into its corresponding AA framework yields the translated AA framework of the logic program, we now obtain mappings between 3-valued interpretations of a logic program and argument labellings of the translated AA framework by concatenating the aforementioned mappings. Since our mappings between 3-valued interpretations and assumption labellings are defined for *all* logic programs, the following mappings maintain this property.

From here onwards, and if not specified otherwise, we assume as given an arbitrary logic program $\mathcal{P}$, its translated ABA framework $ABA_{\mathcal{P}}$, and its translated AA framework $AA_{\mathcal{P}} = \langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$.

Definition 4.5 (Mapping an Argument Labelling into a 3-Valued Interpretation and vice versa)**.**

- `LabArg2Mod` maps an argument labelling $LabArg$ of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ into a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ such that
  $\texttt{LabArg2Mod}(LabArg) = \texttt{LabAsm2Mod}(\texttt{LabArg2LabAsm}(LabArg))$.

- `Mod2LabArg` maps a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ into an argument labelling $LabArg$ of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ such that
  $\texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle) = \texttt{LabAsm2LabArg}(\texttt{Mod2LabAsm}(\langle \mathcal{T}, \mathcal{F} \rangle))$.

$\texttt{LabArg2Mod}$ thus maps an argument labelling LabArg into a 3-valued interpretation $\langle\mathcal{T},\mathcal{F}\rangle$ as follows:

- $\mathcal{T} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})\}$;
- $\mathcal{F} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg})\}$;
- $\mathcal{U} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg})\}$.

Conversely, $\texttt{Mod2LabArg}$ maps a 3-valued interpretation $\langle\mathcal{T},\mathcal{F}\rangle$ into an argument labelling LabArg as follows:

- $\texttt{in}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \mathit{Asms} \subseteq \sim\mathcal{F}\}$;
- $\texttt{out}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \exists \texttt{not}\ l \in \mathit{Asms} : \texttt{not}\ l \in \sim\mathcal{T}\}$;
- $\texttt{undec}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \exists \texttt{not}\ l \in \mathit{Asms} : \texttt{not}\ l \in \sim\mathcal{U}, \mathit{Asms} \cap \sim\mathcal{T} = \emptyset\}$.

Note that using the alternative formulation of $\texttt{LabAsm2Mod}$ (from Proposition 4.1) for $\texttt{LabArg2Mod}$ would yield a more involved definition, for example $\mathcal{T}$ would be defined as $\{l \in \mathit{Lit}_{\mathcal{P}} \mid \exists \mathit{Asms} \vdash l : \mathit{Asms} \subseteq \{\texttt{not}\ m \mid \{\texttt{not}\ m\} \vdash \texttt{not}\ m \in \texttt{in}(\mathit{LabArg})\}\}$. Note also that the translation from literals in $\mathcal{F}$ into arguments labelled $\texttt{in}$ using $\texttt{Mod2LabArg}$ mirrors the mapping from stable models into stable argument extensions by Dung (see previous section).

We observe that when mapping *complete* argument labellings into 3-valued interpretation, $\texttt{LabArg2Mod}$ coincides with $\texttt{LabArg2Mod}_{Wu}$ (extended to logic programs that may comprise explicitly negated atoms).

Proposition 4.12. *Let LabArg be a complete argument labelling of $\mathit{AA}_{\mathcal{P}}$. Then $\texttt{LabArg2Mod}(\mathit{LabArg})$ is equivalent to $\texttt{LabArg2Mod}_{Wu}(\mathit{LabArg})$.*

Proof. Let $\mathcal{T} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \exists \mathit{Asms} \vdash l \in \texttt{in}(\mathit{LabArg})\}$. For every l it holds that since $\mathit{Asms} \vdash l$ attacks $\{\texttt{not}\ l\} \vdash \texttt{not}\ l$, by the (reverse) definition of complete argument labellings (see Section 2.2.1) it follows that $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})$. Thus, $\mathcal{T} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})\}$. Using similar reasoning we show that $\mathcal{F} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \forall \mathit{Asms} \vdash l : \mathit{Asms} \vdash l \in \texttt{out}(\mathit{LabArg})\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg})\}$ and $\mathcal{U} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \exists \mathit{Asms} \vdash l \in \texttt{undec}(\mathit{LabArg}), \nexists \mathit{Asms}' \vdash l \in \texttt{in}(\mathit{LabArg})\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg})\}$. □

Example 4.9. Consider again the only complete argument labelling of $\mathit{AA}_{\mathcal{P}_2}$, i.e. $\mathit{LabArg} = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{undec}), (A_4, \texttt{undec}), (A_5, \texttt{in})\}$ (see Example 4.8). According to $\texttt{LabArg2Mod}(\mathit{LabArg})$, $p \in \mathcal{F}$ since $A_1 : \{\texttt{not}\ p\} \vdash \texttt{not}\ p \in \texttt{in}(\mathit{LabArg})$, $q \in \mathcal{T}$ since $A_2 : \{\texttt{not}\ q\} \vdash \texttt{not}\ q \in \texttt{out}(\mathit{LabArg})$, and $r \in \mathcal{U}$ since $A_3 : \{\texttt{not}\ r\} \vdash \texttt{not}\ r \in \texttt{undec}(\mathit{LabArg})$. This coincides with the 3-valued interpretation obtained by $\texttt{LabArg2Mod}_{Wu}$ and is the only 3-valued stable model of $\mathcal{P}_2$, as discussed in Example 4.8.

Note however, that this equivalence does not hold for argument labellings that are not complete.

Example 4.10. Let $\mathit{LabArg} = \{(A_1, \texttt{out}), (A_2, \texttt{out}), (A_3, \texttt{undec}), (A_4, \texttt{out}), (A_5, \texttt{undec})\}$ be an argument labelling of $AA_{\mathcal{P}_2}$ that is not a complete argument labelling. $\texttt{LabArg2Mod}(\mathit{LabArg}) = \langle\{p, q\}, \{\}\rangle$, but $\texttt{LabArg2Mod}_{Wu} = \langle\{r\}, \{p\}\rangle$.

Concerning Mod2LabArg, we obtain a similar result, namely that for 3-valued stable models it coincides with $\texttt{Mod2LabArg}_{Wu}$.

Proposition 4.13. *Let $\langle\mathcal{T}, \mathcal{F}\rangle$ be a 3-valued stable model of $\mathcal{P}$. Then $\texttt{Mod2LabArg}(\langle\mathcal{T}, \mathcal{F}\rangle)$ is equivalent to $\texttt{Mod2LabArg}_{Wu}(\langle\mathcal{T}, \mathcal{F}\rangle)$.*

Proof. Let $\texttt{in}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \textit{ for all attackers } \mathit{Asms}' \vdash k' \textit{ of } \mathit{Asms} \vdash k : k' \in \mathcal{F}\}$. Thus, for all attacked $\texttt{not}\ k' \in \mathit{Asms}$ it holds that $\texttt{not}\ k' \in \sim \mathcal{F}$. Furthermore, for all unattacked $\texttt{not}\ k'$ it holds that there exists no argument $\mathit{Asms}' \vdash k' \in Ar_{\mathcal{P}}$. Thus, by the definition of 3-valued stable model, $k' \in \mathcal{F}$, so $\texttt{not}\ k' \in \sim \mathcal{F}$. Then $\texttt{in}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \mathit{Asms} \subseteq \sim \mathcal{F}\}$.
Using similar reasoning we show that $\texttt{out}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \textit{ there exists an attacker } \mathit{Asms}' \vdash k' \textit{ of } \mathit{Asms} \vdash k : k' \in \mathcal{T}\} = \{\mathit{Asms} \vdash k \mid \exists \texttt{not}\ l \in \mathit{Asms} : \texttt{not}\ l \in \sim \mathcal{T}\}$ and $\texttt{undec}(\mathit{LabArg}) = \{\mathit{Asms} \vdash k \mid \textit{ there exists an attacker } \mathit{Asms}' \vdash k' \textit{ of } \mathit{Asms} \vdash k : k' \notin \mathcal{F}, \textit{ for all attackers } \mathit{Asms}'' \vdash k'' \textit{ of } \mathit{Asms} \vdash k : k'' \notin \mathcal{T}\} = \{\mathit{Asms} \vdash k \mid \exists \texttt{not}\ l \in \mathit{Asms} : \texttt{not}\ l \in \sim \mathcal{U}, \mathit{Asms} \cap \sim \mathcal{T} = \emptyset\}$. □

This equivalence does not hold for 3-valued interpretations in general.

Example 4.11. Let $\mathcal{P}_5 = \{p \leftarrow \texttt{not}\ q, \texttt{not}\ u;\ q \leftarrow \texttt{not}\ p\}$ be a logic program and let $\langle\{p, u\}, \{q\}\rangle$ be a 3-valued interpretation. In addition to the three assumption-arguments A_1, A_2, and A_3 for assumptions $\texttt{not}\ p$, $\texttt{not}\ q$, and $\texttt{not}\ u$, the translated AA framework has arguments $A_4 : \{\texttt{not}\ q, \texttt{not}\ u\} \vdash p$ and $A_5 : \{\texttt{not}\ p\} \vdash q$, where A_4 and A_5 attack each other. Then $\texttt{Mod2LabArg}(\langle\{p, u\}, \{q\}\rangle)$ and $\texttt{Mod2LabArg}_{Wu}(\langle\{p, u\}, \{q\}\rangle)$ differ in the labels of A_4: $\texttt{Mod2LabArg}(A_4) = \texttt{out}$, but $\texttt{Mod2LabArg}_{Wu}(A_4) = \texttt{in}$.

In contrast to the mappings between assumption labellings and 3-valued interpretations, LabArg2Mod and Mod2LabArg are in general neither bijections nor the inverses of one another, since they apply the mappings LabArg2LabAsm and LabAsm2LabArg, which are not bijections or each other's inverses (see Section 3.4.1). However, we observe the following relation between LabArg2Mod and Mod2LabArg. Our first result states that translating an argument labelling into a 3-valued interpretation and then back into an argument labelling preserves the labels of assumption-arguments.

Proposition 4.14. *Let LabArg be an argument labelling of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$ and $\mathit{LabArg}' = \texttt{Mod2LabArg}(\texttt{LabArg2Mod}(\mathit{LabArg}))$. Then for all $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in Ar_{\mathcal{P}}$ it holds that $\mathit{LabArg}(\{\texttt{not}\ l\} \vdash \texttt{not}\ l) = \mathit{LabArg}'(\{\texttt{not}\ l\} \vdash \texttt{not}\ l)$.*

Proof. Let $\langle \mathcal{T}, \mathcal{F} \rangle = \texttt{LabArg2Mod}(\mathit{LabArg})$.
If $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg})$, then $l \in \mathcal{F}$, so $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg}')$.
If $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})$, then $l \in \mathcal{T}$, so $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg}')$.
If $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg})$, then $l \in \mathcal{U}$, so $\{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg}')$. □

However, non-assumption-arguments may have different labels in *LabArg* and *LabArg′*.

Example 4.12. Let $\mathcal{P}_6 = \{p \leftarrow \texttt{not}\ q\}$ be a logic program. $AA_{\mathcal{P}_6}$ has three arguments: $A_1 : \{\texttt{not}\ p\} \vdash \texttt{not}\ p$, $A_2 : \{\texttt{not}\ q\} \vdash \texttt{not}\ q$, and $A_3 : \{\texttt{not}\ q\} \vdash p$. Let $\mathit{LabArg} = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{undec})\}$ be an argument labelling of $AA_{\mathcal{P}_6}$. Then $\langle \mathcal{T}, \mathcal{F} \rangle = \texttt{LabArg2Mod}(\mathit{LabArg}) = \langle \{q\}, \{p\} \rangle$, and $\mathit{LabArg}' = \texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle) = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{out})\}$, so the assumption-arguments A_1 and A_2 have the same labels in *LabArg* and *LabArg′*, but the non-assumption-argument A_3 has a different label in *LabArg* and *LabArg′*.

Conversely, translating a 3-valued interpretation into an argument labelling and then back into a 3-valued interpretation preserves the initial interpretation.

Proposition 4.15. *Let* $\langle \mathcal{T}, \mathcal{F} \rangle$ *be a 3-valued interpretation of* $\mathcal{P}$ *and let* $\langle \mathcal{T}', \mathcal{F}' \rangle = \texttt{LabArg2Mod}(\texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle))$. *Then* $\langle \mathcal{T}, \mathcal{F} \rangle = \langle \mathcal{T}', \mathcal{F}' \rangle$.

Proof. Let $\mathit{LabArg} = \texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle)$. Then

- $\mathcal{T}' = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \texttt{not}\ l \in \sim\mathcal{T}\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid l \in \mathcal{T}\} = \mathcal{T}$,
- $\mathcal{F}' = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg})\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \subseteq \sim\mathcal{F}\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid l \in \mathcal{F}\} = \mathcal{F}$,
- $\mathcal{U}' = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg})\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \texttt{not}\ l \in \sim\mathcal{U}, \{\texttt{not}\ l\} \cap \sim\mathcal{T} = \emptyset\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid \texttt{not}\ l \in \sim\mathcal{U}\} = \{l \in \mathit{Lit}_{\mathcal{P}} \mid l \in \mathcal{U}\} = \mathcal{U}$.

□

Note that the relationship from Propositions 4.14 and 4.15 does not generally hold for $\texttt{LabArg2Mod}_{Wu}$ and $\texttt{Mod2LabArg}_{Wu}$. We thus argue, that our mappings `LabArg2Mod` and `Mod2LabArg` are preferable, as they map an argument labelling "more accurately" into a 3-valued interpretation.

Example 4.13. Let $\mathit{LabArg} = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{undec})\}$ be an argument labelling of $AA_{\mathcal{P}_6}$ (see Example 4.12). Let $\langle \mathcal{T}, \mathcal{F} \rangle = \texttt{LabArg2Mod}_{Wu}(\mathit{LabArg}) = \langle \emptyset, \{q\} \rangle$, and let us then translate $\langle \mathcal{T}, \mathcal{F} \rangle$ back into an argument labelling, which yields $\mathit{LabArg}' = \texttt{Mod2LabArg}_{Wu}(\langle \mathcal{T}, \mathcal{F} \rangle) = \{(A_1, \texttt{undec}), (A_2, \texttt{in}), (A_3, \texttt{in})\}$. This argument labelling is completely different from the original argument labelling. Furthermore, starting from a 3-valued interpretation, e.g. $\langle \mathcal{T}, \mathcal{F} \rangle$, we observe that $\texttt{LabArg2Mod}_{Wu}(\texttt{Mod2LabArg}_{Wu}(\langle \mathcal{T}, \mathcal{F} \rangle)) = \langle \{p\}, \{q\} \rangle \neq \langle \mathcal{T}, \mathcal{F} \rangle$.

4.4.3 Semantic Correspondence between Argument Labellings and 3-Valued Models

Since by Propositions 4.12 and 4.13 for complete argument labellings $\texttt{LabArg2Mod}$ coincides with $\texttt{LabArg2Mod}_{Wu}$ and $\texttt{Mod2LabArg}$ with $\texttt{Mod2LabArg}_{Wu}$, the correspondence results between the semantics of a logic program and its translated AA framework by Wu et al. [WCG09] and Caminada et al. [CSAD15b] also hold for our mappings. However, their results are restricted to logic programs without explicit negation. We extend these results to logic programs that may comprise explicitly negated atoms, and show that this correspondence furthermore straightforwardly follows from the correspondence between argument and assumption labellings (see Section 3.4) and assumption labellings and 3-valued models (see Section 4.3.3), thus considerably simplifying the proofs of Wu et al. [WCG09] and Caminada et al. [CSAD15b].

Theorem 4.16. *Let LabArg be an argument labelling of $AA_{\mathcal{P}}$. If LabArg is a complete argument labelling of $AA_{\mathcal{P}}$ such that*

$$\forall a \in \mathcal{HB}_{\mathcal{P}} : \{\texttt{not}\ a\} \vdash \texttt{not}\ a \notin \texttt{out}(\mathit{LabArg}) \lor \{\texttt{not}\ \neg a\} \vdash \texttt{not}\ \neg a \notin \texttt{out}(\mathit{LabArg}),$$

then $\langle \mathcal{T}, \mathcal{F}\rangle = \texttt{LabArg2Mod}(\mathit{LabArg})$ is a 3-valued stable model of $\mathcal{P}$.

Proof. Let LabArg be a complete argument labelling of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}}$: $\{\texttt{not}\ a\} \vdash \texttt{not}\ a \notin \texttt{out}(\mathit{LabArg}) \lor \{\texttt{not}\ \neg a\} \vdash \texttt{not}\ \neg a \notin \texttt{out}(\mathit{LabArg})$. Then by Theorem 3.19 $\mathit{LabAsm} = \texttt{LabArg2LabAsm}(\mathit{LabArg})$ is a complete assumption labelling of $ABA_{\mathcal{P}}$, where $\textsc{in}(\mathit{LabAsm}) = \{\texttt{not}\ l \in \mathcal{A}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg})\}$, $\textsc{out}(\mathit{LabAsm}) = \{\texttt{not}\ l \in \mathcal{A}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})\}$, $\textsc{undec}(\mathit{LabAsm}) = \{\texttt{not}\ l \in \mathcal{A}_{\mathcal{P}} \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg})\}$. Thus, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not}\ a \notin \textsc{out}(\mathit{LabAsm}) \lor \texttt{not}\ \neg a \notin \textsc{out}(\mathit{LabAsm})$. Then by Corollary 4.6, $\langle \mathcal{T}, \mathcal{F}\rangle$ with $\mathcal{T} =\sim \textsc{out}(\mathit{LabAsm}) = \{l \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{out}(\mathit{LabArg})\}$ and $\mathcal{F} =\sim \textsc{in}(\mathit{LabAsm}) = \{l \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{in}(\mathit{LabArg})\}$ is a 3-valued stable model of $\mathcal{P}$, where $\mathcal{U} =\sim \textsc{undec}(\mathit{LabAsm}) = \{l \mid \{\texttt{not}\ l\} \vdash \texttt{not}\ l \in \texttt{undec}(\mathit{LabArg})\}$. □

Given a complete argument labelling LabArg, we say that $\texttt{LabArg2Mod}(\mathit{LabArg})$ is the *corresponding 3-valued stable model* of LabArg.

Note that the correspondence holds one way only. More precisely, it is not the case that any argument labelling LabArg that is mapped into a 3-valued stable model by $\texttt{LabArg2Mod}$ is a complete argument labelling. This is because the translation only takes the labels of assumption-arguments into account. The labels of all other arguments may thus not satisfy the conditions of a complete argument labelling.

Regarding the mapping from 3-valued stable models into complete argument labellings, we not only extend the correspondence results of Wu et al. [WCG09] and Caminada et al. [CSAD15b] to logic programs that may contain explicitly negated atoms, but also prove the opposite direction of the correspondence. That is, any 3-valued interpretation that is mapped into a complete argument labelling by $\texttt{Mod2LabArg}$ is a 3-valued stable model.

Theorem 4.17. *Let $\langle \mathcal{T}, \mathcal{F} \rangle$ be a 3-valued interpretation of $\mathcal{P}$. $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued stable model of $\mathcal{P}$ if and only if $LabArg = \texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is a complete argument labelling of $AA_{\mathcal{P}}$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}} : \{\texttt{not } a\} \vdash \texttt{not } a \notin \texttt{out}(LabArg) \vee \{\texttt{not } \neg a\} \vdash \texttt{not } \neg a \notin \texttt{out}(LabArg)$.*

Proof. By Corollary 4.7 $\langle \mathcal{T}, \mathcal{F} \rangle$ is a 3-valued stable model of $\mathcal{P}$ if and only if $LabAsm$ with $\text{IN}(LabAsm) = \sim\mathcal{F}$, $\text{OUT}(LabAsm) = \sim\mathcal{T}$, and $\text{UNDEC}(LabAsm) = \sim\mathcal{U}$ is a complete assumption labelling of $ABA_{\mathcal{P}}$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}} : \texttt{not } a \notin \text{OUT}(LabAsm) \vee \texttt{not } \neg a \notin \text{OUT}(LabAsm)$. By Theorem 3.17, $LabAsm$ is a complete assumption labelling of $ABA_{\mathcal{P}}$ if and only if $\texttt{LabAsm2LabArg}(LabAsm)$ is a complete argument labelling of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$, where $\texttt{in}(LabArg) = \{Asms \vdash k \mid Asms \subseteq \sim\mathcal{F}\}$, $\texttt{out}(LabArg) = \{Asms \vdash k \mid \exists \texttt{not } l \in Asms : \texttt{not } l \in \sim\mathcal{T}\}$, $\texttt{undec}(LabArg) = \{Asms \vdash k \mid \exists \texttt{not } l \in Asms : \texttt{not } l \in \sim\mathcal{U}, Asms \cap \sim\mathcal{T} = \emptyset\}$. Thus, $\forall a \in \mathcal{HB}_{\mathcal{P}} : \{\texttt{not } a\} \vdash \texttt{not } a \notin \texttt{out}(LabArg) \vee \{\texttt{not } \neg a\} \vdash \texttt{not } \neg a \notin \texttt{out}(LabArg)$. □

Example 4.14. Consider again the logic program $\mathcal{P}_4$ from Example 4.6. The translated AA framework $AA_{\mathcal{P}_4}$ is illustrated in Figure 4.1. It has three complete argument labellings:

- $LabArg_1 = \{(A_1, \texttt{undec}), (A_2, \texttt{undec}), (A_3, \texttt{undec}), (A_4, \texttt{in}), (A_5, \texttt{undec}), (A_6, \texttt{in}), (A_7, \texttt{undec}), (A_8, \texttt{undec}), (A_9, \texttt{undec}), (A_{10}, \texttt{undec}), (A_{11}, \texttt{undec})\}$,
- $LabArg_2 = \{(A_1, \texttt{out}), (A_2, \texttt{out}), (A_3, \texttt{in}), (A_4, \texttt{in}), (A_5, \texttt{undec}), (A_6, \texttt{in}), (A_7, \texttt{in}), (A_8, \texttt{in}), (A_9, \texttt{out}), (A_{10}, \texttt{undec}), (A_{11}, \texttt{out})\}$, and
- $LabArg_3 = \{(A_1, \texttt{in}), (A_2, \texttt{in}), (A_3, \texttt{out}), (A_4, \texttt{in}), (A_5, \texttt{out}), (A_6, \texttt{in}), (A_7, \texttt{out}), (A_8, \texttt{out}), (A_9, \texttt{in}), (A_{10}, \texttt{out}), (A_{11}, \texttt{in})\}$.

$LabArg_2$ does not correspond to a 3-valued stable model of $\mathcal{P}_4$ since both A_1 and A_2 are labelled $\texttt{out}$. $LabArg_1$ and $LabArg_3$ correspond to the two 3-valued stable models $\langle \{\}, \{\neg q, \neg r\} \rangle$ and $\langle \{q, r\}, \{p, \neg p, \neg q, \neg r\} \rangle$, respectively.

Given a 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$, we say that $\texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is the *corresponding complete argument labelling* of $\langle \mathcal{T}, \mathcal{F} \rangle$.

Correspondence between other semantics of a logic program and its translated AA framework then follows straightaway.

Theorem 4.18. *Let $LabArg$ be an argument labelling of $AA_{\mathcal{P}}$. If $LabArg$ is a grounded / preferred / ideal / stable argument labelling of $AA_{\mathcal{P}}$ such that*

$$\forall a \in \mathcal{HB}_{\mathcal{P}} : \{\texttt{not } a\} \vdash \texttt{not } a \notin \texttt{out}(LabArg) \vee \{\texttt{not } \neg a\} \vdash \texttt{not } \neg a \notin \texttt{out}(LabArg),$$

then $\langle \mathcal{T}, \mathcal{F} \rangle = \texttt{LabArg2Mod}(LabArg)$ is a well-founded / 3-valued M-stable / ideal / (2-valued) stable model of $\mathcal{P}$.

Proof. Analogous to the proof of Theorem 4.16 but using Theorem 3.25 instead of Theorem 3.19 and Theorem 4.10 instead of Corollary 4.6. □

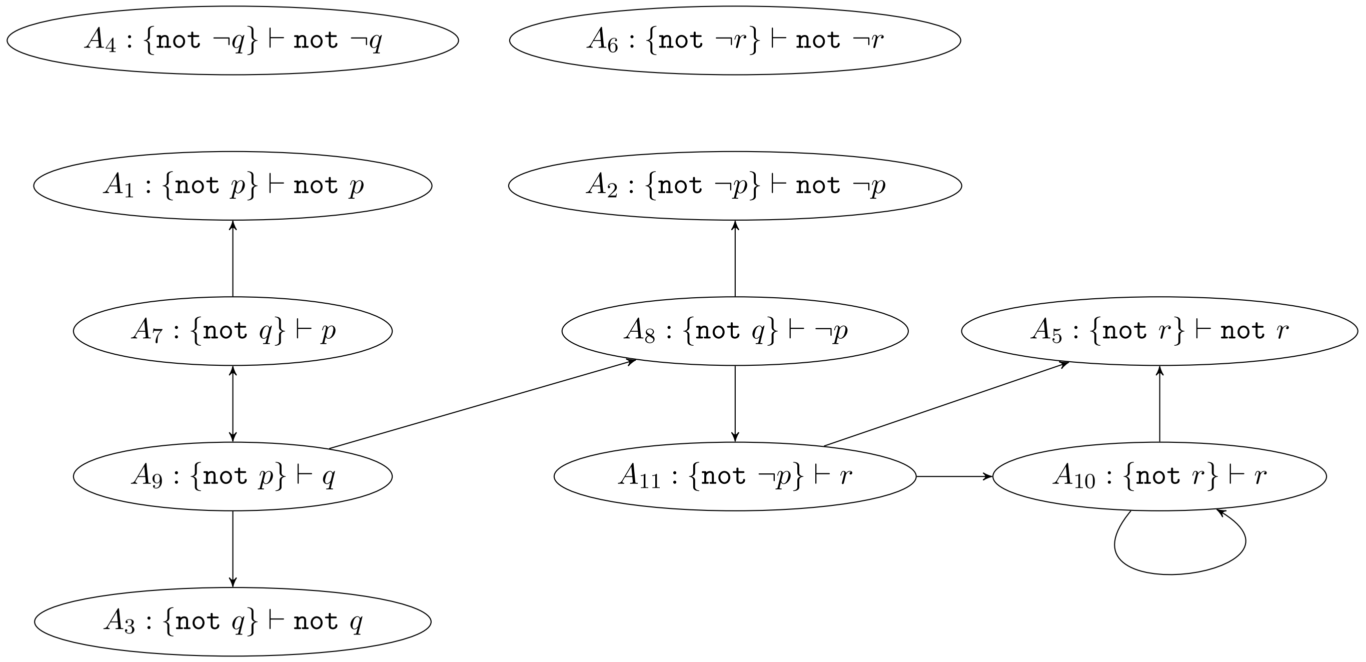

$A_4 : \{\texttt{not } \neg q\} \vdash \texttt{not } \neg q$
$A_6 : \{\texttt{not } \neg r\} \vdash \texttt{not } \neg r$
$A_1 : \{\texttt{not } p\} \vdash \texttt{not } p$
$A_2 : \{\texttt{not } \neg p\} \vdash \texttt{not } \neg p$
$A_7 : \{\texttt{not } q\} \vdash p$
$A_8 : \{\texttt{not } q\} \vdash \neg p$
$A_5 : \{\texttt{not } r\} \vdash \texttt{not } r$
$A_9 : \{\texttt{not } p\} \vdash q$
$A_{11} : \{\texttt{not } \neg p\} \vdash r$
$A_{10} : \{\texttt{not } r\} \vdash r$
$A_3 : \{\texttt{not } q\} \vdash \texttt{not } q$

Figure 4.1: The translated AA framework $AA_{\mathcal{P}_4}$ of $\mathcal{P}_4$ (see Example 4.14).

Theorem 4.19. *Let $\langle \mathcal{T}, \mathcal{F} \rangle$ be a 3-valued interpretation of $\mathcal{P}$. $\langle \mathcal{T}, \mathcal{F} \rangle$ is a well-founded / 3-valued M-stable / ideal / (2-valued) stable model of $\mathcal{P}$ if and only if $LabArg = \texttt{Mod2LabArg}(\langle \mathcal{T}, \mathcal{F} \rangle)$ is a grounded / preferred / ideal / stable argument labelling of $AA_{\mathcal{P}}$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}} : \{\texttt{not}\ a\} \vdash \texttt{not}\ a \notin \texttt{out}(LabArg) \vee \{\texttt{not}\ \neg a\} \vdash \texttt{not}\ \neg a \notin \texttt{out}(LabArg)$.*

Proof. Analogous to the proof of Theorem 4.17 but using Theorem 4.11 instead of Corollary 4.7 and Theorem 3.24 instead of Theorem 3.17. □

The only semantics that do not correspond in general are semi-stable argument labellings and 3-valued L-stable models (also pointed out by Caminada et al. [CSAD15b]) since semi-stable argument and assumption labellings do not correspond (see Section 3.4.4).

Example 4.15. Let $\mathcal{P}_7$ be the following logic program (without explicit negation):

$$\begin{aligned} \{\ & p \leftarrow \texttt{not}\ q; \\ & q \leftarrow \texttt{not}\ p; \\ & r \leftarrow \texttt{not}\ r; \\ & r \leftarrow \texttt{not}\ p, \texttt{not}\ r\ \} \end{aligned}$$

The translated AA framework $AA_{\mathcal{P}_7}$ is illustrated in Figure 4.2. $\mathcal{P}_7$ has three 3-valued stable models, namely $\langle \{\}, \{\} \rangle$, $\langle \{q\}, \{p\} \rangle$, and $\langle \{p\}, \{q\} \rangle$. Using Mod2LabArg, we obtain the three complete argument labellings of $AA_{\mathcal{P}_7}$:

- $LabArg_1 = \{(A_1, \texttt{undec}), (A_2, \texttt{undec}), (A_3, \texttt{undec}), (A_4, \texttt{undec}), (A_5, \texttt{undec}), (A_6, \texttt{undec}), (A_7, \texttt{undec})\}$,
- $LabArg_2 = \{(A_1, \texttt{in}), (A_2, \texttt{out}), (A_3, \texttt{undec}), (A_4, \texttt{out}), (A_5, \texttt{in}), (A_6, \texttt{undec}), (A_7, \texttt{undec})\}$, and
- $LabArg_3 = \{(A_1, \texttt{out}), (A_2, \texttt{in}), (A_3, \texttt{undec}), (A_4, \texttt{in}), (A_5, \texttt{out}), (A_6, \texttt{undec}), (A_7, \texttt{out})\}$.

Both $\langle \{q\}, \{p\} \rangle$, and $\langle \{p\}, \{q\} \rangle$ are 3-valued L-stable models of $\mathcal{P}_7$, but only the complete argument labelling corresponding to the latter is a semi-stable argument labelling of $AA_{\mathcal{P}_7}$, i.e. $LabArg_3$.

4.4.4 Semantic Correspondence between Stable Argument Extensions and Answer Sets

In Chapter 5, we present a justification approach for literals with respect to an *answer set* of a logic program. This approach relies on the correspondence between answer sets and the stable semantics of the translated AA framework. Importantly, we will base these justifications on stable argument *extensions* instead of labellings, since for stable

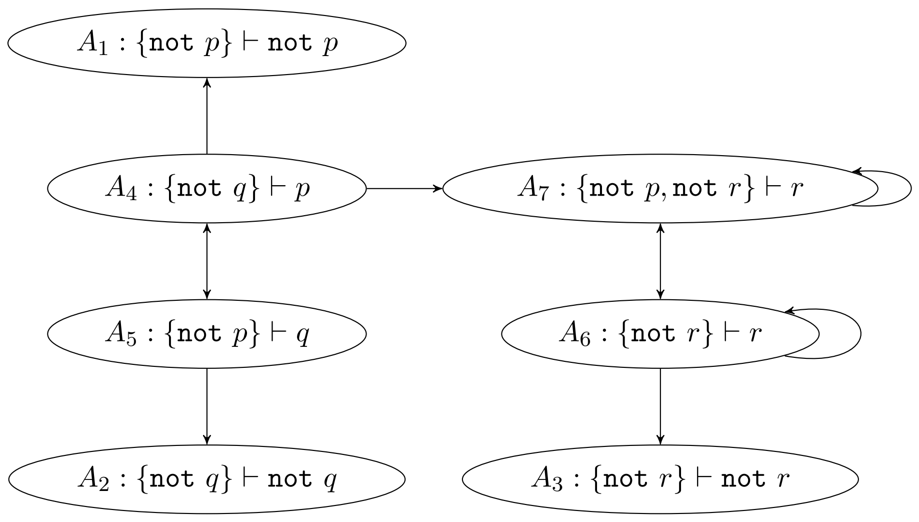

$A_1 : \{\texttt{not}\ p\} \vdash \texttt{not}\ p$
$A_4 : \{\texttt{not}\ q\} \vdash p$
$A_7 : \{\texttt{not}\ p, \texttt{not}\ r\} \vdash r$
$A_5 : \{\texttt{not}\ p\} \vdash q$
$A_6 : \{\texttt{not}\ r\} \vdash r$
$A_2 : \{\texttt{not}\ q\} \vdash \texttt{not}\ q$
$A_3 : \{\texttt{not}\ r\} \vdash \texttt{not}\ r$

Figure 4.2: The translated AA framework $AA_{\mathcal{P}_7}$ from Example 4.15.

argument labellings the labels of all arguments are directly characterised by the respective stable argument extension, i.e. all arguments contained in the stable argument extension are labelled `in` and all arguments not in the stable argument extension are labelled `out`[3]. Therefore, we reformulate the correspondence results between (2-valued) stable models and stable argument labellings from the previous section to state correspondence between answer sets and stable argument *extensions*.

Usually, answer sets only contain classical literals. However, if $l \notin S$ for an answer set S of $\mathcal{P}$ and some classical literal $l \in Lit_{\mathcal{P}}$, then $\texttt{not}\ l$ is considered satisfied with respect to S. Thus, we introduce the notion of *Answer Sets with NAF literals*, i.e. answer sets which also comprise all true NAF literals.

Definition 4.6 (Answer Set with NAF Literals)**.** Let $S \subseteq Lit_{\mathcal{P}}$ be a set of classical literals. $\Delta_S = \{\texttt{not}\ l \in NAF_{Lit_{\mathcal{P}}} \mid l \notin S\}$ consists of all NAF literals $\texttt{not}\ l$ whose corresponding classical literal l is not contained in S. If S is an answer set of $\mathcal{P}$, then $S_{NAF} = S \cup \Delta_S$ is an *answer set with NAF literals* of $\mathcal{P}$.

Since for *consistent* logic programs the answer set semantics coincides with the (2-valued) stable semantics [GL91, Prz90], the correspondence results from Theorems 4.18 and 4.19 concerning (2-valued) stable models also hold between answer sets and stable argument labellings and can be reformulated in terms of stable argument extensions.

We reformulate the "consistency" condition for argument labellings, which is stated in terms of assumption-arguments labelled `out`, namely $\forall a \in \mathcal{HB}_{\mathcal{P}} : \{\texttt{not}\ a\} \vdash \texttt{not}\ a \notin \texttt{out}(LabArg) \vee \{\texttt{not}\ \neg a\} \vdash \texttt{not}\ \neg a \notin \texttt{out}(LabArg)$, as a condition on arguments in the stable extension, namely $\forall a \in \mathcal{HB}_{\mathcal{P}} : \nexists Asms \vdash a \in \mathcal{E} \vee \nexists Asms \vdash \neg a \in \mathcal{E}$, where $\mathcal{E}$ is a stable extension.

[3]This is in general not the case for, e.g., complete extensions. In order to determine the labels of all arguments according to a complete extension, the attacks between arguments need to be taken into account.

Corollary 4.20. *Let $\mathcal{P}$ be a consistent logic program and $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$ the translated AA framework of $\mathcal{P}$. If $\mathcal{E} \subseteq Ar_{\mathcal{P}}$ is a stable argument extension of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$ such that*

$$\forall a \in \mathcal{HB}_{\mathcal{P}} : \nexists Asms \vdash a \in \mathcal{E} \vee \nexists Asms \vdash \neg a \in \mathcal{E},$$

then $S_{NAF} = \{k \mid \exists Asms \vdash k \in \mathcal{E}\}$ is an answer set with NAF literals of $\mathcal{P}$.

This follows from Theorem 4.18 and the correspondence between `LabArg2Mod` and $\texttt{LabArg2Mod}_{Wu}$ from Proposition 4.12. Note that this Corollary extends the correspondence result of Dung [Dun95b] (see Section 4.4) to answer sets, i.e. to logic programs that may comprise explicitly negated atoms.

Corollary 4.21. *Let $\mathcal{P}$ be a consistent logic program and $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$ the translated AA framework of $\mathcal{P}$. $S \subseteq Lit_{\mathcal{P}}$ is an answer set of $\mathcal{P}$ if and only if $\mathcal{E} = \{Asms \vdash k \mid Asms \subseteq \Delta_S\}$ is a stable argument extension of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$ such that $\forall a \in \mathcal{HB}_{\mathcal{P}} : \nexists Asms \vdash a \in \mathcal{E} \vee \nexists Asms \vdash \neg a \in \mathcal{E}$.*

This follows from Theorem 4.19. Note that this Corollary also extends the correspondence result of Dung [Dun95b] (see Section 4.4) to answer sets, i.e. to logic programs that may comprise explicitly negated atoms.

Given an answer set S, we call $\mathcal{E} = \{Asms \vdash k \mid Asms \subseteq \Delta_S\}$ the *corresponding stable argument extension* of S.

4.5 Related Work

Throughout this chapter, we mentioned various closely related works: Bondarenko et al. [BTK93, BDKT97] present some correspondences between the semantics of logic programs and their translated ABA frameworks (see Section 4.3.1), which we extended using our new assumption labellings. Concerning the correspondence between the semantics of logic programs and AA frameworks, both Dung [Dun95b] and Wu et al. [WCG09] present various results, which we extended too.

We focussed on the translation of logic programs into ABA and AA frameworks, whereas other authors have investigated the opposite direction. Dung [Dun95b] gives a translation of AA frameworks into logic programs and proves some semantic correspondence, which is (among others) extended by Osorio et al. [OZNC05] and Wu et al. [WCG09]. Furthermore, Caminada and Schulz [CS15] present a translation of ABA frameworks into logic programs and show semantic correspondence.

More generally, the question whether different non-monotonic reasoning formalism can be translated into one another and how their semantics relate has received considerable attention. Early work focussed on formalisms such as default logic, circumscription, and autoepistemic logic [Imi87, Got95, Jan99] and the formulation of the answer set semantics in other logical formalisms, such as equilibrium logic [Pea96]. More recently, work has been done regarding translations between argumentation frameworks and other non-monotonic logics.

Thimm and Kern-Isberner [TKI08] investigate the correspondence between defeasible logic programming (DeLP), which is commonly classified as an argumentation framework, and answer set programming. Lam et al. [LGR16] study the relation between the ASPIC+ argumentation framework and defeasible logic, and Young et al. [YMR16] between ASPIC+ and prioritised default logic. Heyninck and Straßer [HS16] give translations and correspondence results between ASPIC+, ABA frameworks, and adaptive logics. Furthermore, Bochman [Boc16] studies a translation from abstract dialectical frameworks (ADFs) into causal calculus.

Furthermore, various authors investigate mappings between different argumentation frameworks. Oren et al. [ORL10] present a mapping between AA frameworks and evidential argumentation frameworks (EAFs) and prove semantic correspondence. Polberg and Oren [PO14] investigate mappings between EAFs and argumentation frameworks with necessities and show that there exists no natural translation between the two which preserves the semantics. Polberg [Pol17] extends that work and additionally investigates mappings with ADFs.

4.6 Summary

In this chapter, we reviewed and extended existing correspondence results between the semantics of a logic program and its translated ABA and AA frameworks.

Concerning the translated ABA framework, existing results only showed how to derive a corresponding 3-valued interpretation from an assumption extension, but not vice versa. Furthermore, the corresponding 3-valued interpretations were defined based on the arguments supported by the assumption extension, rather than on the assumption extension itself. We introduced direct mappings between 3-valued interpretations of a logic program and assumption labellings of the translated ABA framework, which do not require to construct arguments. We then proved that the mapping of complete assumption labellings yields 3-valued stable models (analogous to existing results), and that the mapping of 3-valued stable models yields complete assumption labellings. These results can be extended to the correspondence between grounded, preferred, ideal, semi-stable, and stable assumption labellings and, respectively, well-founded, 3-valued M-stable, ideal, 3-valued L-stable, and (2-valued) stable models.

With regards to the translated AA framework of a logic program, various mappings and correspondence results exist, both regarding argument extensions and labellings. We compared these mappings with new mappings obtained by concatenating our mappings between 3-valued interpretations and assumption labellings and between assumption labellings and argument labellings. We showed that for complete assumption labellings and 3-valued stable models, our mappings yield the same outcome as existing mappings. Thus, existing correspondence results between complete, grounded, preferred, ideal, and stable argument labellings and, respectively, 3-valued stable, well-founded, 3-valued M-stable, ideal, and (2-valued) stable models also hold for our mappings. However, in the general case the outcome of our mappings and existing mappings may not be the same. We also show that in contrast to existing mappings, our mappings always preserve the labels/truth values of certain assumptions/literals when translating back and forth between 3-valued interpretations and assumption labellings.

In the next chapter, we introduce a justification approach for logic programs under the answer set semantics, which is based upon the correspondence results presented in this chapter.

Chapter 5

Justifying Answer Sets using Argumentation

5.1 Introduction

If ASP is used for applications in real-world scenarios involving non-experts, it is useful to have an explanation as to why a literal does or does not belong to an answer set. Answer set justification has thus been identified as an important but not yet sufficiently studied research area [LD04, BD08]. In this chapter, we present two methods for justifying literals with respect to an answer set of a consistent logic program by applying the notions of arguments and attacks of the translated ABA and AA framework of a logic program. Our approach is based upon the semantic correspondence results between logic programs and their translated ABA and AA frameworks presented in Chapter 4. Of particular importance for this chapter is the result that every answer set of a logic program corresponds to a stable argument extension of the translated AA framework (Corollaries 4.20 and 4.21).

Our first justification approach, an *Attack Tree*, expresses how to construct an argument for a literal in question (the supporting argument) as well as which arguments attack the argument for the literal in question (the attacking arguments); the same information is provided for all arguments attacking the attacking arguments, and so on. The second justification approach, an *ABA-Based Answer Set (ABAS) Justification* of a literal, represents the same information as an Attack Tree, but expressed in terms of literals rather than arguments. An ABAS Justification comprises facts and NAF literals necessary to derive the literal in question (the "supporting literals") as well as information about literals that are in conflict with the literal in question (the "attacking literals"). The same information is provided for all supporting and attacking literals of the literal in question, for all their supporting and attacking literals, and so on.

The chapter is organised as follows. In Section 5.2, we give some definitions specific to this chapter and in Section 5.3 we introduce a motivating (medical) example and a technical example, which will serve as the running examples throughout this chapter. In Section 5.4, we introduce Attack Trees as our first justification method, show their relationship with abstract dispute trees of the translated AA framework, and characterise the explanations they provide as admissible fragments of the answer set in question. Based on Attack Trees, we define two forms of ABAS Justifications: Basic ABA-Based Answer Set Justifications, introduced in Section 5.5, illustrate how to flatten Attack Trees, yielding a justification in terms of literals and their relations. Labelled ABA-Based Answer Set Justifications, introduced in Section 5.6, constitute a more elaborate version of Basic ABA-Based Answer Set Justifications, following the same flattening strategy, but additionally using labels to solve some deficiencies of the basic variant. In Section 5.7, we present a web-platform implementing Attack Trees and Labelled ABA-Based Answer Set Justifications. In Section 5.8, we compare ABAS Justifications to related work and in Section 5.9 we summarise the contributions of this chapter.

5.2 Preliminaries

Throughout this chapter we use a slightly modified notion of arguments constructible from an ABA framework. In addition to the set of assumptions supporting an argument, the modified version also explicitly comprises the set of facts used in the construction of the argument.

Definition 5.1 (Argument)**.** Let $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^{-} \rangle$ be an ABA framework. An argument for (the conclusion) $s \in \mathcal{L}$ supported by the set of *assumption-premises* $AP \subseteq \mathcal{A}$ and the set of *fact-premises* $FP \subseteq \{t \mid t \leftarrow \in \mathcal{R}\}$ is a finite tree, where every node holds a sentence in $\mathcal{L}$, such that:

- the root node holds s;
- for every node N
  - if N is a leaf, then N holds either an assumption or a fact;
  - if N is not a leaf and N holds the sentence s_0, then there is an inference rule $s_0 \leftarrow s_1, \ldots, s_m$ $(m > 0)$ and N has m children, holding $s_1, \ldots, s_m$ respectively;
- AP is the set of all assumptions held by leaves;
- FP is the set of all facts held by leaves.

We use an analogous notation as for the definition of arguments from Section 2.2.2.

Notation 5.2. Let $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^{-} \rangle$ be an ABA framework. An argument for s supported by AP and FP is denoted $(AP, FP) \vdash s$. We often use a unique name to denote an argument, e.g. $A : (AP, FP) \vdash s$ is an argument with name A. With an abuse of notation, the name of an argument sometimes stands for the whole argument. An argument of the form $(\{\alpha\}, \emptyset) \vdash \alpha$ is called *assumption-argument*, and similarly an argument of the form $(\emptyset, \{t\}) \vdash t$ is called *fact-argument*. Given some argument $A : (AP, FP) \vdash s$ with $\alpha \in AP$ and $t \in FP$, we say that $(\{\alpha\}, \emptyset) \vdash \alpha$ is the *assumption-argument of the assumption-premise* α of argument A and that $(\emptyset, \{t\}) \vdash t$ is the *fact-argument of the fact-premise* t of A.

We note that Definition 5.1 generates the notion of argument as introduced in Section 2.2.2: If $(AP, FP) \vdash s$ is an argument according to Definition 5.1, then $AP \vdash s$ is an argument as defined in Section 2.2.2. Conversely, if $Asms \vdash s$ is an argument as defined in Section 2.2.2, then there exists some $FP \subseteq \{t \mid t \leftarrow \in \mathcal{R}\}$ such that $(Asms, FP) \vdash s$ is an argument according to Definition 5.1.

Based on the modified notion of arguments, we also reformulate attacks between arguments.

Definition 5.3 (Attacks)**.** Let $\langle \mathcal{L}, \mathcal{R}, \mathcal{A}, ^{-} \rangle$ be an ABA framework.
An argument $(AP_1, FP_1) \vdash s_1$ *attacks* an argument $(AP_2, FP_2) \vdash s_2$ if and only if

$\exists \alpha \in AP_2$ such that $s_1 = \overline{\alpha}$. Equivalently, we say that $(AP_2, FP_2) \vdash s_2$ *is attacked by* $(AP_1, FP_1) \vdash s_1$ or that $(AP_1, FP_1) \vdash s_1$ is an *attacker of* $(AP_2, FP_2) \vdash s_2$.

Attacks between sets of arguments are then defined as for AA frameworks in Section 2.2.1

Clearly, attacks between the modified notion of arguments and arguments as defined in Section 2.2.3 correspond. That is, if an argument $(AP_1, FP_1) \vdash s_1$ attacks an argument $(AP_2, FP_2) \vdash s_2$ according to Definition 5.3, then $AP_1 \vdash s_1$ attacks $AP_2 \vdash s_2$ as defined in Section 2.2.3. Conversely, if $Asms_1 \vdash s_1$ attacks $Asms_2 \vdash s_2$ as defined in Section 2.2.3, then there exist $FP_1, FP_2 \subseteq \{t \mid t \leftarrow \in \mathcal{R}\}$ such that $(Asms_1, FP_1) \vdash s_1$ attacks $(Asms_2, FP_2) \vdash s_2$ according to Definition 5.3.

Based on the correspondence results from Section 4.4.4, we show that for every literal k in an answer set with NAF literals there is at least one argument with conclusion k in the corresponding stable argument extension. Conversely, if a literal k is not contained in an answer set with NAF literals, then no argument with conclusion k is part of the corresponding stable argument extension.

Proposition 5.1. *Let $\mathcal{P}$ be a logic program, S an answer set of $\mathcal{P}$, and $\mathcal{E}$ the corresponding stable argument extension of S in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$. Let $k \in Lit_{\mathcal{P}} \cup NAF_{Lit_{\mathcal{P}}}$.*

1. *If $k \in S_{NAF}$, then there exists an argument $A \in \mathcal{E}$ such that $A : (AP, FP) \vdash k$ with $AP \subseteq \Delta_S$ and $FP \subseteq S$.*
2. *If $k \notin S_{NAF}$, then there exists no $A : (AP, FP) \vdash k$ in $Ar_{\mathcal{P}}$ such that $A \in \mathcal{E}$.*

Proof.

1. By Corollary 4.20, $S_{NAF} = \{k_1 \mid \exists (AP, FP) \vdash k_1 \in \mathcal{E}\}$, so if $k \in S_{NAF}$, then there exists at least one argument $A : (AP, FP) \vdash k \in \mathcal{E}$. By Corollary 4.21, $\mathcal{E} = \{(AP_1, FP_1) \vdash k_1 \mid AP_1 \subseteq \Delta_S\}$, so it follows that for argument A, $AP \subseteq \Delta_S$. Furthermore, $FP \subseteq S$ because $FP \subseteq \{t \mid t \leftarrow \in \mathcal{P}\}$ and for consistent logic programs it trivially holds that $\{ t \mid t \leftarrow \in \mathcal{P}\} \subseteq S$.
2. Assume that there exists $A : (AP, FP) \vdash k$ in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ such that $A \in \mathcal{E}$. Then according to Corollary 4.20, $k \in S_{NAF}$. Contradiction.

□

Given an answer set S of $\mathcal{P}$, the corresponding stable argument extension $\mathcal{E}$ of $AA_{\mathcal{P}}$, and a literal $k \in S_{NAF}$, an argument $A \in \mathcal{E}$ with conclusion k is called a *corresponding argument* of k.

Note that the first part of Proposition 5.1 only states that for a literal k in the answer set with NAF literals there *exists* a corresponding argument in the corresponding stable argument extension. However, there might be further arguments $(AP, FP) \vdash k$ that are not part of the corresponding stable argument extension, where $AP \nsubseteq \Delta_S$. Note also that the second part of Proposition 5.1 does not exclude the existence of arguments with conclusion k. It merely states that no such argument is contained in the corresponding stable argument extension.

5.3 Running Examples

We now introduce two running examples used throughout this chapter. The first one is an intuitive medical example, which extends the example from Chapter 1, whereas the second one is more technical and is used to illustrate some details of our approach.

5.3.1 Intuitive Medical Example

Let Dr. Smith be an ophtalmologist (an eye doctor) and let one of his patients be Peter, who is diagnosed by Dr. Smith as being short-sighted. Based on this diagnosis, Dr. Smith has to decide on the most suitable treatment for Peter, taking into account all information he has about his patient, namely that Peter is afraid to touch his own eyes, that he is a student, and that he likes to do sports. Based on this information and his specialist knowledge, Dr. Smith decides that the most appropriate treatment for Peter's short-sightedness is laser surgery. Dr. Smith now checks whether this decision is in line with the recommendation of his decision support system, which is implemented in ASP.

The following logic program $\mathcal{P}_{doctor}$ represents the decision support system used by Dr. Smith. It encodes some general world knowledge as well as an ophtalmologist's specialist knowledge about the possible treatments of short-sightedness. $\mathcal{P}_{doctor}$ also captures the additional information that Dr. Smith has about his short-sighted patient Peter.

$$\begin{aligned}
\{\ tightOnMoney &\leftarrow student, \texttt{not}\ richParents;\\
caresAboutPracticality &\leftarrow likesSports;\\
correctiveLenses &\leftarrow shortSighted, \texttt{not}\ laserSurgery;\\
laserSurgery &\leftarrow shortSighted, \texttt{not}\ tightOnMoney, \texttt{not}\ correctiveLenses;\\
glasses &\leftarrow correctiveLenses, \texttt{not}\ caresAboutPracticality,\\
&\ \texttt{not}\ contactLenses;\\
contactLenses &\leftarrow correctiveLenses, \texttt{not}\ afraidToTouchEyes,\\
&\ \texttt{not}\ longSighted, \texttt{not}\ glasses;\\
intraocularLenses &\leftarrow correctiveLenses, \texttt{not}\ glasses, \texttt{not}\ contactLenses;\\
shortSighted &\leftarrow\ ;\\
afraidToTouchEyes &\leftarrow\ ;\\
student &\leftarrow\ ;\\
likesSports &\leftarrow\ \}
\end{aligned}$$

$\mathcal{P}_{doctor}$ has only one answer set, namely $S_{doctor} = \{shortSighted,\ afraidToTouchEyes,$

student, *likesSports*, *tightOnMoney*, *correctiveLenses*, *caresAboutPracticality*, *intraocularLenses*}.

To Dr. Smith's surprise, the answer set computed by the decision support system contains the literal *intraocularLenses* but not *laserSurgery*, suggesting that Peter should get intraocular lenses instead of having laser surgery. Dr. Smith now finds himself in the difficult situation to determine whether to trust his own decision or adopt the system's suggestion. Providing Dr. Smith with an explanation of the system's suggestion or with an explanation as to why his own intended decision might be wrong would make it considerably easier for Dr. Smith to decide whether to trust himself or the decision support system.

We will use this example of Dr. Smith and his patient Peter to demonstrate our justification approaches and to show how they can be applied to explain the solutions of a decision support system that is based on ASP.

5.3.2 Technical Example

Let $\mathcal{P}_8$ be the following logic program, where $Lit_{\mathcal{P}_8} = \{p, \neg p, q, \neg q, u, \neg u, w, \neg w\}$:

$$\begin{aligned}
\{\ p &\leftarrow \texttt{not}\ \neg p;\\
p &\leftarrow \neg p, \texttt{not}\ q, \texttt{not}\ w;\\
\neg p &\leftarrow \texttt{not}\ q, \texttt{not}\ u;\\
q &\leftarrow \texttt{not}\ w;\\
u &\leftarrow \texttt{not}\ \neg p;\\
w &\leftarrow\ \}
\end{aligned}$$

$\mathcal{P}_8$ has two answer sets: $S_1 = \{w, u, p\}$ and $S_2 = \{w, \neg p\}$. The respective sets of satisfied NAF literals are
$\Delta_{S_1} = \{\texttt{not}\ \neg p, \texttt{not}\ q, \texttt{not}\ \neg q, \texttt{not}\ \neg u, \texttt{not}\ \neg w\}$ and
$\Delta_{S_2} = \{\texttt{not}\ p, \texttt{not}\ q, \texttt{not}\ \neg q, \texttt{not}\ u, \texttt{not}\ \neg u, \texttt{not}\ \neg w\}$.

In order to use ABA and AA for the justification of literals with respect to an answer set of a logic program, we construct the translated ABA and AA frameworks of the logic program. The translated ABA framework of $\mathcal{P}_8$ is $ABA_{\mathcal{P}_8} = \langle \mathcal{L}_{\mathcal{P}_8}, \mathcal{R}_{\mathcal{P}_8}, \mathcal{A}_{\mathcal{P}_8}, \bar{\ } \rangle$ with:

- $\mathcal{R}_{\mathcal{P}_8} = \mathcal{P}_8$,
- $\mathcal{A}_{\mathcal{P}_8} = NAF_{\mathcal{P}_8} = \{\texttt{not}\ p, \texttt{not}\ \neg p, \texttt{not}\ q, \texttt{not}\ \neg q, \texttt{not}\ u, \texttt{not}\ \neg u, \texttt{not}\ w, \texttt{not}\ \neg w\}$,
- $\overline{\texttt{not}\ p} = p$; $\overline{\texttt{not}\ \neg p} = \neg p$; $\overline{\texttt{not}\ q} = q$; $\overline{\texttt{not}\ \neg q} = \neg q$; $\overline{\texttt{not}\ u} = u$; $\overline{\texttt{not}\ \neg u} = \neg u$; $\overline{\texttt{not}\ w} = w$; $\overline{\texttt{not}\ \neg w} = \neg w$,
- $\mathcal{L}_{\mathcal{P}_8} = Lit_{\mathcal{P}_8} \cup NAF_{\mathcal{P}_8}$.

Fourteen arguments can be constructed in $ABA_{\mathcal{P}_8}$, including eight assumption-arguments (A_1 - A_8) and one fact-argument (A_{14}):

$A_1 : (\{\texttt{not}\ p\}, \emptyset) \vdash \texttt{not}\ p$
$A_2 : (\{\texttt{not}\ \neg p\}, \emptyset) \vdash \texttt{not}\ \neg p$
$A_3 : (\{\texttt{not}\ q\}, \emptyset) \vdash \texttt{not}\ q$
$A_4 : (\{\texttt{not}\ \neg q\}, \emptyset) \vdash \texttt{not}\ \neg q$
$A_5 : (\{\texttt{not}\ u\}, \emptyset) \vdash \texttt{not}\ u$
$A_6 : (\{\texttt{not}\ \neg u\}, \emptyset) \vdash \texttt{not}\ \neg u$
$A_7 : (\{\texttt{not}\ w\}, \emptyset) \vdash \texttt{not}\ w$
$A_8 : (\{\texttt{not}\ \neg w\}, \emptyset) \vdash \texttt{not}\ \neg w$
$A_9 : (\{\texttt{not}\ \neg p\}, \emptyset) \vdash p$
$A_{10} : (\{\texttt{not}\ q, \texttt{not}\ u, \texttt{not}\ w\}, \emptyset) \vdash p$
$A_{11} : (\{\texttt{not}\ q, \texttt{not}\ u\}, \emptyset) \vdash \neg p$
$A_{12} : (\{\texttt{not}\ w\}, \emptyset) \vdash q$
$A_{13} : (\{\texttt{not}\ \neg p\}, \emptyset) \vdash u$
$A_{14} : (\emptyset, \{w\}) \vdash w$
The translated AA framework $\langle Ar_{\mathcal{P}_8}, Att_{\mathcal{P}_8}\rangle$ of $\mathcal{P}_8$ is given in Figure 5.1.

Two stable argument extensions can be determined for $\langle Ar_{\mathcal{P}_8}, Att_{\mathcal{P}_8}\rangle$:
$\mathcal{E}_1 = \{A_2, A_3, A_4, A_6, A_8, A_9, A_{13}, A_{14}\}$ and
$\mathcal{E}_2 = \{A_1, A_3, A_4, A_5, A_6, A_8, A_{11}, A_{14}\}$.
As expected, the conclusions of arguments in the stable argument extensions coincide with $S1_{NAF}$ and $S2_{NAF}$, as stated in Corollary 4.20, where the conclusions are:
$\{\texttt{not}\ \neg p, \texttt{not}\ q, \texttt{not}\ \neg q, \texttt{not}\ \neg u, \texttt{not}\ \neg w, p, u, w\}$ of $\mathcal{E}_1$ and
$\{\texttt{not}\ p, \texttt{not}\ q, \texttt{not}\ \neg q, \texttt{not}\ u, \texttt{not}\ \neg u, \texttt{not}\ \neg w, \neg p, w\}$ of $\mathcal{E}_2$.
Conversely, the two sets of arguments whose assumption-premises are subsets of Δ_{S_1} and Δ_{S_2}, respectively, coincide with the two stable argument extensions $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, as stated in Corollary 4.21.

When taking a closer look at $S1_{NAF}$, we can verify that every literal has a corresponding argument in $\mathcal{E}_1$: w has A_{14}, u has A_{13}, p has A_9, $\texttt{not}\ \neg p$ has A_2, $\texttt{not}\ q$ has A_3, and so on. Furthermore, for all literals not contained in $S1_{NAF}$, there exists no argument with this conclusion in the stable argument extension $\mathcal{E}_1$, e.g. $\neg p \notin S1_{NAF}$ and $A_{11} \notin \mathcal{E}_1$. The same holds for S_2 and $\mathcal{E}_2$.

5.4 Attack Trees

Proposition 5.1, part 1, provides the starting point for our justification approaches as it allows us to explain why a literal is in an answer set based on the reasons for a corresponding argument to be in the corresponding stable argument extension. Similarly, Proposition 5.1, part 2, is a starting point for justifying why a literal is not contained in an answer set based on arguments for that literal, all of which are not contained in the corresponding stable argument extension. In AA it is easy to explain why an argument is or is not contained in a stable argument extension: an argument is part of a stable argument extension if it is not attacked by it. Since the stable argument extension attacks

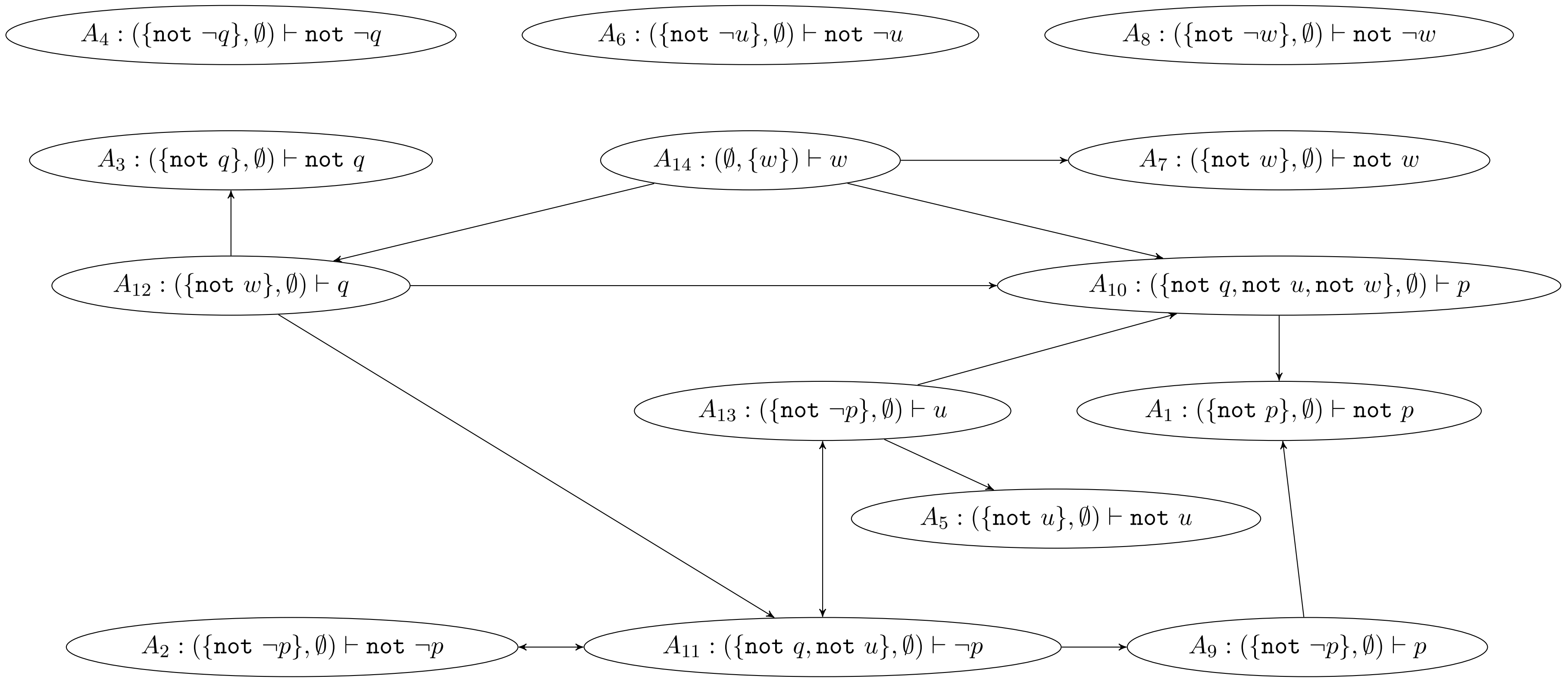

$A_4 : (\{\texttt{not } \neg q\}, \emptyset) \vdash \texttt{not } \neg q$
$A_6 : (\{\texttt{not } \neg u\}, \emptyset) \vdash \texttt{not } \neg u$
$A_8 : (\{\texttt{not } \neg w\}, \emptyset) \vdash \texttt{not } \neg w$
$A_3 : (\{\texttt{not } q\}, \emptyset) \vdash \texttt{not } q$
$A_{14} : (\emptyset, \{w\}) \vdash w$
$A_7 : (\{\texttt{not } w\}, \emptyset) \vdash \texttt{not } w$
$A_{12} : (\{\texttt{not } w\}, \emptyset) \vdash q$
$A_{10} : (\{\texttt{not } q, \texttt{not } u, \texttt{not } w\}, \emptyset) \vdash p$
$A_{13} : (\{\texttt{not } \neg p\}, \emptyset) \vdash u$
$A_1 : (\{\texttt{not } p\}, \emptyset) \vdash \texttt{not } p$
$A_5 : (\{\texttt{not } u\}, \emptyset) \vdash \texttt{not } u$
$A_2 : (\{\texttt{not } \neg p\}, \emptyset) \vdash \texttt{not } \neg p$
$A_{11} : (\{\texttt{not } q, \texttt{not } u\}, \emptyset) \vdash \neg p$
$A_9 : (\{\texttt{not } \neg p\}, \emptyset) \vdash p$

Figure 5.1: The translated AA framework of $\mathcal{P}_8$.

all arguments that are not part of it, this entails that an argument in the stable argument extension is defended by the stable argument extension, i.e. the stable argument extension attacks all attackers of this argument. Conversely, an argument is not part of a stable argument extension if it is attacked by this stable argument extension. In this section, we will make use of these results in order to develop a justification method that provides explanations in terms of arguments and attacks between them.

Our first justification approach explains why arguments are or are not contained in a stable argument extension by constructing an *Attack Tree* of this argument with respect to the stable argument extension. This tree of attacking arguments is later used to construct a justification in terms of literals. Due to the correspondence between answer sets and stable argument extensions, a justification of a literal k with respect to an answer set can be obtained from an Attack Tree of an argument with conclusion k constructed with respect to the corresponding stable argument extension. In this section we define the notion of Attack Trees and show their relationship with abstract dispute trees, characterising the explanations they provide as admissible fragments of the stable argument extension as well as of the answer set.

From here onwards, and if not specified otherwise, we assume as given a consistent logic program $\mathcal{P}$ and its translated AA framework $AA_{\mathcal{P}} = \langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$.

5.4.1 Constructing Attack Trees

Nodes in an Attack Tree hold arguments which are labelled either '+' or '−'. An Attack Tree of an argument A has A itself in the root node, where either one or all attackers of A form(s) the child node(s) of this root. In the same way, each of these child nodes holding some argument B have either all or one of B's attackers as children, and so on. Whether only one or all attackers of an argument are considered as child nodes depends on the argument's label in the Attack Tree, which is determined with respect to a given set of arguments (typically a stable argument extension of the translated AA framework). If an argument is part of given set, it is labelled '+' and has all its attackers as child nodes. If the argument is not contained in the set, it is labelled '−' and has exactly one of its attackers as a child node.

Definition 5.4 (Attack Tree)**.** Let $Args \subseteq Ar_{\mathcal{P}}$ and $A \in Ar_{\mathcal{P}}$. An *Attack Tree* of A (constructed) w.r.t. $Args$, denoted $attTree_{Args}(A)$, is a (possibly infinite) tree such that:

1. every node in $attTree_{Args}(A)$ holds an argument in $Ar_{\mathcal{P}}$, labelled '+' or '−';

2. the root node is A^+ if $A \in Args$ or A^- if $A \notin Args$;

3. for every node A_N^+ and for every argument A_i attacking A_N in $AA_{\mathcal{P}}$, there exists a child node A_i^- of A_N^+;

4. every node A_N^-

(i) has no child node if A_N is not attacked in $AA_{\mathcal{P}}$ or if for all attackers A_i of A_N: $A_i \notin Args$; or else

(ii) has exactly one child node A_i^+ for some $A_i \in Args$ attacking A_N;

5. there are no other nodes in $attTree_{Args}(A)$ except those given in 1-4.

If $attTree_{Args}(A)$ is an Attack Tree of A w.r.t. $Args$, we also say that *A has the Attack Tree $attTree_{Args}(A)$*. Note that due to condition 4(ii), where only one of possibly many arguments A_i is chosen, an argument can have more than one Attack Tree. Furthermore, note the difference between 3, where A_i is any argument attacking A_N, and 4(ii), where A_i has to be an attacking argument contained in $Args$.

Notation 5.5. If $A \in Args$, and thus the root node of $attTree_{Args}(A)$ is A^+, we denote the Attack Tree as $attTree^+_{Args}(A)$ and call it a *positive Attack Tree.* If $A \notin Args$, and thus the root node of $attTree_{Args}(A)$ is A^-, we denote the Attack Tree as $attTree^-_{Args}(A)$ and call it a *negative Attack Tree.*

The following example illustrates the notion of Attack Trees w.r.t. a set of arguments which is a stable argument extension.

Example 5.1. We consider the logic program $\mathcal{P}_8$ and its translated AA framework $AA_{\mathcal{P}_8}$ from Section 5.3. Figure 5.2 shows the two negative Attack Trees of argument A_{10} w.r.t. the stable argument extension $\mathcal{E}_1 = \{A_2, A_3, A_4, A_6, A_8, A_9, A_{13}, A_{14}\}$, i.e. $attTree^-_{\mathcal{E}_1}(A_{10})_1$ and $attTree^-_{\mathcal{E}_1}(A_{10})_2$. Since $A_{10} \notin \mathcal{E}_1$, the root node of all Attack Trees of A_{10} holds A^-_{10}, and consequently has exactly one or no attacker of A_{10} as a child node. A_{10} is attacked by the three arguments A_{12}, A_{13}, and A_{14} (see Figure 5.1), so these are the candidates for being a child node of A^-_{10}. However, A^+_{12} cannot serve as a child node of A^-_{10} as $A_{12} \notin \mathcal{E}_1$ (see condition 4(ii) in Definition 5.4). Since both A_{13} and A_{14} are contained in $\mathcal{E}_1$, either of them can be used as a child node of A^-_{10}, leading to two possible Attack Trees of A_{10}. The left of Figure 5.2 depicts the negative Attack Tree $attTree^-_{\mathcal{E}_1}(A_{10})_1$ where A^+_{14} is chosen as the child node of A^-_{10}, whereas the right of Figure 5.2 illustrates $attTree^-_{\mathcal{E}_1}(A_{10})_2$ where A^+_{13} is chosen. $attTree^-_{\mathcal{E}_1}(A_{10})_1$ ends with A^+_{14} since A_{14} is not attacked in $AA_{\mathcal{P}_8}$. In contrast, choosing A^+_{13} as the child node of A^-_{10} leads to an infinite negative Attack Tree $attTree^-_{\mathcal{E}_1}(A_{10})_2$: A^+_{13} has a single child A^-_{11} since A_{11} is the only argument attacking A_{13}; A_{11} is attacked by both A_{12} and A_{13} in $\mathcal{P}_8$, but only A^+_{13} can serve as a child node of A^-_{11} as $A_{12} \notin \mathcal{E}_1$; at this point, the Attack Tree starts to repeat itself, since the only possible child node of A^-_{11} is A^+_{13}, whose only child node is A^-_{11}, and so on.
With respect to the stable argument extension $\mathcal{E}_2 = \{A_1, A_3, A_4, A_5, A_6, A_8, A_{11}, A_{14}\}$ of $AA_{\mathcal{P}_8}$, A_{10} has a unique negative Attack Tree $attTree^-_{\mathcal{E}_2}(A_{10})$, which is exactly the same as $attTree^-_{\mathcal{E}_1}(A_{10})_1$. The reason is that only A^+_{14} can serve as a child node of A^-_{10} since both $A_{12} \notin \mathcal{E}_2$ and $A_{13} \notin \mathcal{E}_2$.

Figure 5.2 illustrates that an argument might have more than one Attack Tree, as well as that Attack Trees can be infinite. Figure 5.3 depicts another negative Attack

$$\begin{tikzcd}
A_{10}^{-} : (\{\mathtt{not}\ q, \mathtt{not}\ u, \mathtt{not}\ w\}, \emptyset) \vdash p \\
A_{14}^{+} : (\emptyset, \{w\}) \vdash w \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
A_{10}^{-} : (\{\mathtt{not}\ q, \mathtt{not}\ u, \mathtt{not}\ w\}, \emptyset) \vdash p \\
A_{13}^{+} : (\{\mathtt{not}\ \neg p\}, \emptyset) \vdash u \arrow[u] \\
A_{11}^{-} : (\{\mathtt{not}\ q, \mathtt{not}\ u\}, \emptyset) \vdash \neg p \arrow[u] \\
A_{13}^{+} : (\{\mathtt{not}\ \neg p\}, \emptyset) \vdash u \arrow[u] \\
\vdots
\end{tikzcd}$$

Figure 5.2: The two negative Attack Trees $attTree^{-}_{\mathcal{E}_1}(A_{10})_1$ (left) and $attTree^{-}_{\mathcal{E}_1}(A_{10})_2$ (right) of A_{10} w.r.t. $\mathcal{E}_1$, as described in Example 5.1. The left Attack Tree is also the unique negative Attack Tree $attTree^{-}_{\mathcal{E}_2}(A_{10})$ of A_{10} w.r.t. $\mathcal{E}_2$.

$$\begin{tikzcd}
 & A_{9}^{-} : (\{\mathtt{not}\ \neg p\}, \emptyset) \vdash p & & & \\
 & A_{11}^{+} : (\{\mathtt{not}\ q, \mathtt{not}\ u\}, \emptyset) \vdash \neg p \arrow[u] & & & \\
A_{12}^{-} : (\{\mathtt{not}\ w\}, \emptyset) \vdash q \arrow[ur] & & A_{13}^{-} : (\{\mathtt{not}\ \neg p\}, \emptyset) \vdash u \arrow[ul] & & \\
A_{14}^{+} : (\emptyset, \{w\}) \vdash w \arrow[u] & & A_{11}^{+} : (\{\mathtt{not}\ q, \mathtt{not}\ u\}, \emptyset) \vdash \neg p \arrow[u] & & \\
 & A_{12}^{-} : (\{\mathtt{not}\ w\}, \emptyset) \vdash q \arrow[ur] & & A_{13}^{-} : (\{\mathtt{not}\ \neg p\}, \emptyset) \vdash u \arrow[ul] & \\
 & A_{14}^{+} : (\emptyset, \{w\}) \vdash w \arrow[u] & & \vdots &
\end{tikzcd}$$

Figure 5.3: The unique negative Attack Tree $attTree^{-}_{\mathcal{E}_2}(A_9)$ of A_9 w.r.t. the stable argument extension $\mathcal{E}_2$ of $AA_{\mathcal{P}_8}$ (see Section 5.3).

Tree, illustrating the case where a node labelled '+' has more than one child node. Note that every argument in an AA framework has at least one Attack Tree. However, an Attack Tree may solely consist of the root, for example the unique positive Attack Tree $attTree^{+}_{\mathcal{E}_1}(A_{14})$ of A_{14} w.r.t. the stable argument extension $\mathcal{E}_1$ consists of only one node, namely the root node A_{14}^{+} as this argument has no attackers.

From the definition of Attack Trees it follows that the Attack Trees of an argument are either all positive or all negative.

Lemma 5.2. *Let $Args \subseteq Ar_{\mathcal{P}}$ be a set of arguments.*

1. *If $A \in Args$, then all Attack Trees of A w.r.t. $Args$ are positive Attack Trees $attTree^{+}_{Args}(A)$.*

2. *If $A \notin Args$, then all Attack Trees of A w.r.t. $Args$ are negative Attack Trees*

$attTree^{-}_{Args}(A)$.

Proof. This follows directly from Definition 5.4 and Notation 5.5. □

Intuitively, an Attack Tree of an argument w.r.t. a set of arguments explains why the argument is or is not contained in the set by showing either that the argument is defended by the set, i.e. the set attacks all attackers of the argument, or that the argument is attacked by the set and cannot defend itself against it.

5.4.2 Attack Trees with respect to Stable Extensions

For justification purposes we construct Attack Trees w.r.t. stable argument extensions rather than an arbitrary set of arguments. This enables us to later extract a justification of a literal w.r.t. an answer set from an Attack Tree constructed w.r.t. the corresponding stable argument extension. In this section we show some characteristics of Attack Trees when constructed w.r.t. a stable argument extension, which hold for both positive and negative Attack Trees.

One of these characteristics is that we can deduce whether or not an argument held by a node in an Attack Tree constructed w.r.t. a stable argument extension is contained in this stable argument extension: all arguments labelled '+' in the Attack Tree are contained in the stable argument extension, whereas all arguments labelled '−' are not in the stable argument extension.

Lemma 5.3. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$ and let $\Upsilon = attTree_{\mathcal{E}}(A)$ be an Attack Tree of $A \in Ar_{\mathcal{P}}$ w.r.t. $\mathcal{E}$. Then*

1. *for each node A_i^+ in Υ: $A_i \in \mathcal{E}$;*
2. *for each node A_i^- in Υ: $A_i \notin \mathcal{E}$.*

Proof.

1. A_i^+ is either the root node, then by definition $A_i \in \mathcal{E}$, or it is the only child node of some A_N^-, meaning that by definition $A_i \in \mathcal{E}$.
2. A_i^- is either the root node, then by definition $A_i \notin \mathcal{E}$, or A_i^- is a child node of some A_N^+, and A_i attacks A_N. From part 1 we know that $A_N \in \mathcal{E}$, hence $A_i \notin \mathcal{E}$ because $\mathcal{E}$ does not attack itself.

□

Another interesting characteristic of an Attack Tree constructed w.r.t. a stable argument extension is that all nodes holding arguments labelled '−' have exactly one child node, rather than none. Furthermore, all leaf nodes hold arguments labelled '+'.

Lemma 5.4. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$ and let $\Upsilon = attTree_{\mathcal{E}}(A)$ be an Attack Tree of $A \in Ar_{\mathcal{P}}$ w.r.t. $\mathcal{E}$. Then*

1. *every node A_N^- in Υ has exactly one child node;*

2. *all leaf nodes in Υ hold arguments labelled '$+$'.*

Proof.

1. By condition 4 in Definition 5.4, any node A_N^- in an Attack Tree has either no or exactly one child node. By Lemma 5.3 $A_N \notin \mathcal{E}$. Assume that A_N^- has no child node. Then A_N is not attacked in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$. But by definition of stable argument extension all arguments not contained in a stable argument extension are attacked by the stable argument extension. Contradiction.

2. This follows directly from part 1 as nodes holding an argument labelled '$-$' always have a child node and thus cannot be a leaf node.

$\square$

Note that infinite branches of Attack Trees do not have leaf nodes, in which case the second part of Lemma 5.4 is trivially satisfied.

Lemma 5.4 highlights how an Attack Tree justifies an argument A w.r.t. a stable argument extension. If the argument A is part of the stable argument extension, the Attack Tree shows that the reason is that A is defended by the stable argument extension. This means that any attackers of A are counter-attacked by an argument in the stable argument extension, defending A against the attacker, and even if the defending argument is further attacked, there will be another argument in the stable argument extension defending this defender, until eventually the defending arguments from the stable argument extension are not further attacked, forming the leaf nodes of the Attack Tree. If an argument A is not part of the stable argument extension, the leaf nodes of the Attack Tree again hold arguments from the stable argument extension, but this time these leaf nodes defend the argument attacking A, meaning that this attacker is contained in the stable argument extension. Thus, A is attacked by the stable argument extension and consequently A is not part of the stable argument extension.

Lemma 5.4 also emphasises the idea that to justify an argument that is not in the stable argument extension, it is enough to show that one of its attackers is contained in the stable argument extension, even if there might be more than one such attacker. This follows the general concept of proof by counter-example. Thus, an Attack Tree disproves that the argument held by the root node is in the stable argument extension by showing one way in which the argument is attacked by the stable argument extension.

From these considerations is follows directly that the subtree of any negative Attack Tree obtained by removing the root node is a positive Attack Tree of the argument attacking the root node.

Lemma 5.5. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$. Let $\Upsilon = attTree_{\mathcal{E}}^-(A)$ be an Attack Tree of $A \in Ar_{\mathcal{P}}$ such that $A \notin \mathcal{E}$ and let A_i^+ be*

$$\begin{tikzcd}
A_1^- : (\{\texttt{not}\ tightOnMoney, \texttt{not}\ correctiveLenses\}, \{shortSighted\}) \vdash laserSurgery \\
A_2^+ : (\{\texttt{not}\ richParents\}, \{student\}) \vdash tightOnMoney \arrow[u]
\end{tikzcd}$$

Figure 5.4: A negative Attack Tree of the argument A_1 w.r.t. the corresponding stable argument extension of the answer set S_{doctor} of the logic program $\mathcal{P}_{doctor}$ (see Example 5.2).

the (only) child node of the root node A^- in $attTree^-_{\mathcal{E}}(A)$. Let Υ' be the subtree of Υ with root node A_i^+ obtained from Υ by removing its root node A^-. Then Υ' is a positive Attack Tree of A_i.

Proof. This follows directly from Definition 5.4 and Notation 5.5. □

This observation will be useful when comparing Attack Trees to abstract dispute trees in the next section. Example 5.2 demonstrates how an Attack Tree can be used to explain why a literal is or is not contained in an answer set in terms of an argument for this literal.

Example 5.2. Consider Dr. Smith, his patient Peter, and the decision support system introduced in Section 5.3. In order to explain to Dr. Smith why *laserSurgery* is not a suggested treatment of the decision support system, an Attack Tree for an argument with conclusion *laserSurgery* w.r.t. the corresponding stable argument extension of the answer set S_{doctor} can be constructed. Figure 5.4 displays such an Attack Tree, which expresses that Peter should not have laser surgery as the decision to use laser surgery is based on the assumption that the patient is not tight on money; however there is evidence that Peter is tight on money as he is known to be a student and there is no evidence against the assumption that his parents are not rich. Note that this is not the only Attack Tree for A_1 and therefore not the only possible explanation why Peter should not have laser surgery. A second Attack Tree can be constructed using an argument with conclusion *correctiveLenses* as an attacker of A_1.

On the other hand, Dr. Smith might want to know why the treatment recommended by the decision support system is *intraocularLenses*. The respective Attack Tree is illustrated in Figure 5.5. It expresses that Peter should get intraocular lenses because for every possible evidence against intraocular lenses (A_1, A_4, A_6) there is counter-evidence (A_2, A_5, and A_7 respectively): for example, receiving intraocular lenses is based on the assumption that it has not been decided that the patient should have glasses. Even though there is some evidence that Peter could have glasses, this evidence is based on the assumption that he does not care about the practicality of his treatment. However, it is known that Peter cares about practicality since he likes to do sports.

5.4.3 Relationship between Attack Trees and Abstract Dispute Tress

In order to further characterise Attack Trees, we prove that Attack Trees constructed w.r.t. stable argument extensions are special cases of abstract dispute trees (see Sec-

$$\begin{tikzcd}
 & {A_3^+ : (\{\texttt{not}\ \mathit{laserSurgery}, \texttt{not}\ \mathit{glasses}, \texttt{not}\ \mathit{contactLenses}\}, \{\mathit{shortSighted}\}) \vdash \mathit{intraocularLenses}} & \\
{A_1^- : (\ldots) \vdash \mathit{laserSurgery}} \arrow[ur] & & {A_6^- : (\{\texttt{not}\ \mathit{laserSurgery}, \texttt{not}\ \mathit{caresAboutPracticality}, \texttt{not}\ \mathit{contactLenses}\}, \{\mathit{shortSighted}\}) \vdash \mathit{glasses}} \arrow[ul] \\
{A_2^+ : (\ldots) \vdash \mathit{tightOnMoney}} \arrow[u] & & {A_7^+ : (\emptyset, \{\mathit{likesSports}\}) \vdash \mathit{caresAboutPracticality}} \arrow[u] \\
{A_4^- : (\{\texttt{not}\ \mathit{laserSurgery}, \texttt{not}\ \mathit{afraidToTouchEyes}, \texttt{not}\ \mathit{longSighted}, \texttt{not}\ \mathit{glasses}\}, \{\mathit{shortSighted}\}) \vdash \mathit{contactLenses}} \arrow[uuur] & & \\
{A_5^+ : (\emptyset, \{\mathit{afraidToTouchEyes}\}) \vdash \mathit{afraidToTouchEyes}} \arrow[u] & &
\end{tikzcd}$$

Figure 5.5: A positive Attack Tree of the argument A_3 w.r.t. the corresponding stable argument extension of the answer set S_{doctor} of the logic program $\mathcal{P}_{doctor}$ (see Example 5.2). The nodes holding A_1^- and A_2^+ are abbreviated as they are the same as in Figure 5.4.

tion 2.2.1). Using this correspondence, we show that Attack Trees provide explanations of an argument in terms of an admissible fragment of the stable argument extension. This result is then extended, proving that given a literal k and an answer set, an Attack Tree of an argument with conclusion k w.r.t. the corresponding stable argument extension provides a justification in terms of an admissible fragment of the answer set.

We first define a translation of the nodes holding arguments labelled '+' and '−' in Attack Trees into the status of proponent and opponent nodes in abstract dispute trees.

Definition 5.6 (Translated Abstract Dispute Tree)**.** Let $Args \subseteq Ar_{\mathcal{P}}$ be a set of arguments and let $attTree_{Args}(A)$ be an Attack Tree of $A \in Ar_{\mathcal{P}}$ w.r.t. $Args$. The *translated abstract dispute tree* $\mathscr{T}_{Args}(A)$ is obtained from $attTree_{Args}(A)$ by assigning the status of proponent to all nodes holding an argument labelled '+', the status of opponent to all nodes holding an argument labelled '−', and dropping the labels '+' and '−' of all arguments in the tree.

If Attack Trees are constructed w.r.t. a stable argument extension, they correspond to abstract dispute trees in the following way.

Lemma 5.6. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$. Let $attTree_{\mathcal{E}}(A)$ be an Attack Tree of $A \in Ar_{\mathcal{P}}$ w.r.t. $\mathcal{E}$ and let $\mathscr{T}_{\mathcal{E}}(A)$ be the translated abstract dispute tree. Then*

1. *if $A \in \mathcal{E}$, then $\mathscr{T}_{\mathcal{E}}(A)$ is an abstract dispute tree for A;*

2. *if $A \notin \mathcal{E}$, then the subtree of $\mathscr{T}_{\mathcal{E}}(A)$ with root node A_i, where A_i^+ is the only child of the root A^- in $attTree_{\mathcal{E}}(A)$, is an abstract dispute tree for A_i.*

Proof. This follows directly from the definition of abstract dispute trees and Lemma 5.4. □

Note that the converse of the first item in Lemma 5.6 does not hold, i.e. it is not the case that every abstract dispute tree for an argument A corresponds to an Attack Tree $attTree_{\mathcal{E}}(A)$.

Example 5.3. Let $\mathcal{P}_9$ be the following logic program:

$$\begin{aligned} \{\ p &\leftarrow \texttt{not}\ p, \texttt{not}\ q;\\ q &\leftarrow \texttt{not}\ p, \texttt{not}\ u;\\ u &\leftarrow \texttt{not}\ q\ \} \end{aligned}$$

The translated AA framework $AA_{\mathcal{P}_9}$ has six arguments:

$A_1 : (\{\texttt{not}\ p\}, \emptyset) \vdash \texttt{not}\ p$	$A_4 : (\{\texttt{not}\ p, \texttt{not}\ q\}, \emptyset) \vdash p$
$A_2 : (\{\texttt{not}\ q\}, \emptyset) \vdash \texttt{not}\ q$	$A_5 : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q$
$A_3 : (\{\texttt{not}\ u\}, \emptyset) \vdash \texttt{not}\ u$	$A_6 : (\{\texttt{not}\ q\}, \emptyset) \vdash u$

The only stable argument extension of $AA_{\mathcal{P}_9}$, corresponding to the only answer set of $\mathcal{P}_9$, is $\mathcal{E} = \{A_1, A_3, A_5\}$. Figure 5.6 illustrates the unique negative Attack Tree $attTree^-_{\mathcal{E}}(A_4)$ of A_4 w.r.t. $\mathcal{E}$. Constructing the translated abstract dispute tree of $attTree^-_{\mathcal{E}}(A_4)$ results in the tree shown in Figure 5.7. As stated by the second item in Lemma 5.6, deleting the opponent root node of the translated abstract dispute tree $\mathscr{T}_{\mathcal{E}}(A_4)$ yields an abstract dispute tree for A_5. Figure 5.8 gives an example of an abstract dispute tree that does not correspond to an Attack Tree, showing that the converse of Lemma 5.6 does not hold. The abstract dispute tree for A_6 starts with a proponent node, which corresponds to the label '+' in an Attack Tree. However, any Attack Tree of A_6 is negative since $A_6 \notin \mathcal{E}$, so the root node is always A_6^-. Thus, there is no Attack Tree that corresponds to the abstract dispute tree for A_6.

Using the correspondence with abstract dispute trees, we can further characterise Attack Trees constructed w.r.t. a stable argument extension as representing admissible fragments of this stable argument extension. Starting with positive Attack Trees, we show that translated abstract dispute trees of positive Attack Trees w.r.t. a stable argument extension are admissible.

Lemma 5.7. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$ and let $A \in \mathcal{E}$. For every positive Attack Tree $attTree^+_{\mathcal{E}}(A)$ of A w.r.t. $\mathcal{E}$, $\mathscr{T}_{\mathcal{E}}(A)$ is an admissible abstract dispute tree.*

$$\begin{tikzcd}
& A_4^- : (\{\texttt{not}\ p, \texttt{not}\ q\}, \emptyset) \vdash p & \\
& A_5^+ : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q \arrow[u] & \\
A_4^- : (\{\texttt{not}\ p, \texttt{not}\ q\}, \emptyset) \vdash p \arrow[ur] & & A_6^- : (\{\texttt{not}\ q\}, \emptyset) \vdash u \arrow[ul] \\
\vdots & & A_5^+ : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q \arrow[u] \\
& & \vdots
\end{tikzcd}$$

Figure 5.6: The unique negative Attack Tree $attTree^-_{\mathcal{E}}(A_4)$ of A_4 w.r.t. the stable argument extension $\mathcal{E}$ of $\langle Ar_{\mathcal{P}_9}, Att_{\mathcal{P}_9} \rangle$ (see Example 5.3).

$$\begin{tikzcd}
& \text{opponent: } A_4 : (\{\texttt{not}\ p, \texttt{not}\ q\}, \emptyset) \vdash p \arrow[d, dash] & \\
& \text{proponent: } A_5 : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q \arrow[dl, dash] \arrow[dr, dash] & \\
\text{opponent: } A_4 : (\{\texttt{not}\ p, \texttt{not}\ q\}, \emptyset) \vdash p \arrow[d, dash] & & \text{opponent: } A_6 : (\{\texttt{not}\ q\}, \emptyset) \vdash u \arrow[d, dash] \\
\text{proponent: } A_5 : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q & & \text{proponent: } A_5 : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q \\
\vdots & & \vdots
\end{tikzcd}$$

Figure 5.7: The translated abstract dispute tree $\mathscr{T}_{\mathcal{E}}(A_4)$ of $attTree^-_{\mathcal{E}}(A_4)$ (see Example 5.3 and Figure 5.6). As the root of $\mathscr{T}_{\mathcal{E}}(A_4)$ is an opponent node, it is not an abstract dispute tree. However, the subtree with root node A_5 is an abstract dispute tree A_5.

Proof. According to Lemma 5.3, for each A_i^+ in $attTree^+_{\mathcal{E}}(A)$, $A_i \in \mathcal{E}$, and for each A_j^- in $attTree^+_{\mathcal{E}}(A)$, $A_j \notin \mathcal{E}$. By definition of stable argument extension, for all arguments B in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ either $B \in \mathcal{E}$ or $B \notin \mathcal{E}$. Thus, $A_i \neq A_j$ for all i, j, and therefore by Definition 5.6 no argument labels both a proponent and an opponent node in $\mathscr{T}_{\mathcal{E}}(A)$, satisfying the condition for admissibility. By Lemma 5.6, $\mathscr{T}_{\mathcal{E}}(A)$ is an abstract dispute tree. □

Since a positive Attack Tree constructed w.r.t. a stable argument extension corresponds to an admissible abstract dispute tree, the set of all arguments labelled '+' in the Attack Tree forms an admissible argument extension, in particular one that is a subset of this stable argument extension.

Theorem 5.8. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$, and $attTree^+_{\mathcal{E}}(A)$ a positive Attack Tree of $A \in \mathcal{E}$. Then the set Args of all arguments labelled '+' in $attTree^+_{\mathcal{E}}(A)$ is an admissible argument extension of $AA_{\mathcal{P}}$ and $Args \subseteq \mathcal{E}$.*

Proof. Let $Args$ denote the set of all arguments labelled '+' in $attTree^+_{\mathcal{E}}(A)$. Then $Args$ is the set of arguments held by proponent nodes in the translated abstract dispute tree

$$\begin{tikzcd}[row sep=small]
\text{proponent:} \arrow[d, dash] & A_6 : (\{\texttt{not}\ q\}, \emptyset) \vdash u \\
\text{opponent:} \arrow[d, dash] & A_5 : (\{\texttt{not}\ p, \texttt{not}\ u\}, \emptyset) \vdash q \\
\text{proponent:} & A_6 : (\{\texttt{not}\ q\}, \emptyset) \vdash u \\
\vdots &
\end{tikzcd}$$

Figure 5.8: An abstract dispute tree for A_6 in $AA_{\mathcal{P}_9}$ (see Example 5.3).

$\mathscr{T}_{\mathcal{E}}(A)$ of $attTree^{+}_{\mathcal{E}}(A)$. By Lemma 5.7, $\mathscr{T}_{\mathcal{E}}(A)$ is an admissible abstract dispute tree. By Theorem 3.2(i) in [DMT07], *Args* is an admissible argument extension, and by Lemma 5.3, $Args \subseteq \mathcal{E}$. □

This result characterises Attack Trees as a way of justifying an argument by means of an admissible fragment of the stable argument extension. In other words, an Attack Tree does not use the whole stable argument extension to explain that an argument is in the stable argument extension, but only provides an admissible subset sufficient to show that it defends the argument in question. Furthermore, we can express this result in logic programming terms: given a literal and an answer set, an Attack Tree of an argument for this literal constructed w.r.t. the corresponding stable argument extension justifies the argument using an admissible fragment of the answer set.

Theorem 5.9. *Let S be an answer set of $\mathcal{P}$, $k \in S_{NAF}$, and $\mathcal{E}$ the corresponding stable argument extension of S in $AA_{\mathcal{P}}$. Let $A \in \mathcal{E}$ be a corresponding argument of k, $attTree^{+}_{\mathcal{E}}(A)$ a positive Attack Tree of A, and $Asms = \{\alpha \mid \alpha \in AP, A_1^{+} : (AP, FP) \vdash k_1$ in $attTree^{+}_{\mathcal{E}}(A)\}$. Then*

1. *$\mathcal{P} \cup Asms$ is an admissible scenario of $\mathcal{P}$ in the sense of [DR91];*
2. *$\{k_1 \mid A_1^{+} : (AP, FP) \vdash k_1$ in $attTree^{+}_{\mathcal{E}}(A)\} \subseteq S_{NAF}$.*

Proof.

1. By Theorem 5.8 and Theorem 2.2(ii) in [DMT07], *Asms* is an admissible set of assumptions. Then by Theorem 4.5 in [BDKT97], $\mathcal{P} \cup Asms$ is an admissible scenario of $\mathcal{P}$ in the sense of [DR91].[1]
2. By Theorem 5.8 and Corollary 4.20.

□

The following example illustrates the characteristics of positive Attack Trees and how they can be used for justifying an argument for a literal in an answer set.

[1]Theorem 4.5 refers to [Dun95a] where admissible scenarios are defined for logic programs without classical negation. This result can be easily extended to the definition of admissible scenarios of logic programs with both classical negation and NAF as we are only concerned with consistent logic programs.

Example 5.4. Consider again the logic program $\mathcal{P}_8$ and its answer set $S_1 = \{w, u, p\}$ with the corresponding stable argument extension $\mathcal{E}_1 = \{A_2, A_3, A_4, A_6, A_8, A_9, A_{13}, A_{14}\}$ (see Section 5.3). To justify that $\mathtt{not}\ q \in S_{1_{NAF}}$, we can construct an Attack Tree of an argument for $\mathtt{not}\ q$, i.e. of A_3, w.r.t. $\mathcal{E}_1$. The resulting positive Attack Tree $attTree^+_{\mathcal{E}_1}(A_3)$ is depicted on the left of Figure 5.9. Translating this Attack Tree into an abstract dispute tree as given in Definition 5.6, yields the translated abstract dispute tree $\mathscr{T}_{\mathcal{E}_1}(A_3)$ illustrated on the right of Figure 5.9. This abstract dispute tree is admissible as stated in Lemma 5.7. The set of arguments labelled '+' in $attTree^+_{\mathcal{E}_1}(A_3)$ is $\{A_3, A_{14}\} \subseteq \mathcal{E}_1$, which is an admissible argument extension of $AA_{\mathcal{P}_8}$, and the set of conclusions of these arguments is $\{\mathtt{not}\ q, w\} \subseteq S_{1_{NAF}}$, as stated by Theorems 5.8 and 5.9. The Attack Tree $attTree^+_{\mathcal{E}_1}(A_3)$ explains that the literal $\mathtt{not}\ q$ is in the answer set S_1 because it is supported and defended by an admissible subset of S_1, namely by $\{\mathtt{not}\ q, w\}$. In terms of literals, the Attack Tree expresses that $\mathtt{not}\ q$ is "attacked" by the literal q, which is "counter-attacked" by w, thereby "defending" $\mathtt{not}\ q$.

$$\begin{tikzcd}
A_3^+ : (\{\mathtt{not}\ q\}, \emptyset) \vdash \mathtt{not}\ q \\
A_{12}^- : (\{\mathtt{not}\ w\}, \emptyset) \vdash q \arrow[u] \\
A_{14}^+ : (\emptyset, \{w\}) \vdash w \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{proponent:} \arrow[d, dash] & A_3 : (\{\mathtt{not}\ q\}, \emptyset) \vdash \mathtt{not}\ q \\
\text{opponent:} \arrow[d, dash] & A_{12} : (\{\mathtt{not}\ w\}, \emptyset) \vdash q \\
\text{proponent:} & A_{14} : (\emptyset, \{w\}) \vdash w
\end{tikzcd}$$

Figure 5.9: The positive Attack Tree $attTree^+_{\mathcal{E}_1}(A_3)$ of A_3 w.r.t. the corresponding stable argument extension $\mathcal{E}_1$ of S_1 (left) and the translated abstract dispute tree $\mathscr{T}_{\mathcal{E}_1}(A_3)$ of $attTree^+_{\mathcal{E}_1}(A_3)$ (right) (see Example 5.4).

Similarly to positive Attack Trees, we can characterise the explanations given by negative Attack Trees using the correspondence between the subtree of a negative Attack Tree and an abstract dispute tree: negative Attack Trees justify that an argument is not in a stable argument extension because it is attacked by an admissible fragment of this stable argument extension. We first prove that when deleting the opponent root node of the translated abstract dispute tree of a negative Attack Tree constructed w.r.t. a stable argument extension, the resulting abstract dispute tree is admissible.

Lemma 5.10. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$, and $A \in Ar_{\mathcal{P}}$ such that $A \notin \mathcal{E}$. For every negative Attack Tree $attTree^-_{\mathcal{E}}(A)$ of A w.r.t. $\mathcal{E}$, the subtree of $\mathscr{T}_{\mathcal{E}}(A)$ with root node A_i, where A_i^+ is the only child of the root A^- in $attTree^-_{\mathcal{E}}(A)$, is an admissible abstract dispute tree.*

Proof. By Lemma 5.5, the subtree of Υ' of $attTree^-_{\mathcal{E}}(A)$ with root node A_i is a positive Attack Tree of A_i. By Lemma 5.7, Υ' is an admissible abstract dispute tree. Trivially, the subtree of $\mathscr{T}_{\mathcal{E}}(A)$ with root node A_i coincides with the translated abstract dispute tree of Υ'. □

We now prove that a negative Attack Tree constructed w.r.t. a stable argument extension justifies the root by showing that it is attacked by an admissible argument extension of $AA_{\mathcal{P}}$, and in particular by an admissible argument extension that is a subset of the stable argument extension.

Theorem 5.11. *Let $\mathcal{E}$ be a corresponding stable argument extension of some answer set of $\mathcal{P}$, and $attTree^{-}_{\mathcal{E}}(A)$ a negative Attack Tree of $A \in Ar_{\mathcal{P}}$. Then the set Args of all arguments labelled '+' in $attTree^{-}_{\mathcal{E}}(A)$ is an admissible argument extension of $AA_{\mathcal{P}}$ and $Args \subseteq \mathcal{E}$.*

Proof. Let $Args$ denote the set of all arguments labelled '+' in $attTree^{-}_{\mathcal{E}}(A)$. Then $Args$ is the set of arguments held by proponent nodes in the translated abstract dispute tree $\mathscr{T}_{\mathcal{E}}(A)$ of $attTree^{-}_{\mathcal{E}}(A)$. By Lemma 5.10, the subtree of $\mathscr{T}_{\mathcal{E}}(A)$ with root node A_i, where A_i^{+} is the only child of the root A^{-} in $attTree^{-}_{\mathcal{E}}(A)$, is an admissible abstract dispute tree. By Theorem 3.2(i) in [DMT07], $Args$ is an admissible argument extension. By Lemma 5.3, $Args \subseteq \mathcal{E}$. □

It follows, that a negative Attack Tree justifies an argument for a literal that is not in the answer set in question in terms of an admissible fragment of the answer set "attacking" the literal.

Theorem 5.12. *Let S be an answer set of $\mathcal{P}$, $k \notin S_{NAF}$, and $\mathcal{E}$ the corresponding stable argument extension of S in $AA_{\mathcal{P}}$. Let A be some argument for k, $attTree^{-}_{\mathcal{E}}(A)$ an Attack Tree of A, and $Asms = \{\alpha \mid \alpha \in AP, A_1^{+} : (AP, FP) \vdash k_1 \text{ in } attTree^{-}_{\mathcal{E}}(A)\}$. Then*

1. *$\mathcal{P} \cup Asms$ is an admissible scenario of $\mathcal{P}$ in the sense of [DR91];*
2. *$\{k_1 \mid A_1^{+} : (AP, FP) \vdash k_1 \text{ in } attTree^{-}_{\mathcal{E}}(A)\} \subseteq S_{NAF}$.*

Proof.

1. By Theorem 5.11 and Theorem 2.2(ii) in [DMT07], $Asms$ is an admissible set of assumptions. Then by Theorem 4.5 in [BDKT97], $\mathcal{P} \cup Asms$ is an admissible scenario of $\mathcal{P}$ in the sense of [DR91].
2. By Theorem 5.11 and Corollary 4.20.

□

This result provides the basis for the construction of a justification of a literal not contained in an answer set, which provides a meaningful explanation in terms of an admissible subset of the answer set.

Example 5.5. Consider the logic program $\mathcal{P}_9$ and its only answer set $S = \{q\}$ with the corresponding stable argument extension $\mathcal{E} = \{A_1, A_3, A_5\}$ (see Example 5.3). To justify why $p \notin S$ we can construct an Attack Tree of an argument with conclusion p, i.e. of A_4, w.r.t. $\mathcal{E}$. The resulting negative Attack Tree $attTree^{-}_{\mathcal{E}}(A_4)$ is depicted in Figure 5.6

and the translated abstract dispute tree $\mathscr{T}_{\mathcal{E}}(A_4)$ in Figure 5.7. When deleting the root opponent node A_4 of $\mathscr{T}_{\mathcal{E}}(A_4)$, the resulting abstract dispute tree is admissible as observed in Lemma 5.10. Furthermore, the set of arguments labelled '+' in $attTree^-_{\mathcal{E}}(A_4)$ is $\{A_5\}$, which is a subset of the corresponding stable argument extension $\mathcal{E}$ and an admissible argument extension of $AA_{\mathcal{P}_9}$ (by Theorem 5.11). Moreover, the set of conclusions of arguments in this admissible argument extension is $\{q\} \subseteq S$, which is an admissible scenario of $\mathcal{P}$ as stated in Theorem 5.12. Therefore, the negative Attack Tree $attTree^-_{\mathcal{E}}(A_4)$ explains that the argument A_4 is not in the corresponding stable argument extension because it is attacked by an admissible fragment of this stable argument extension, namely by $\{A_5\}$. Even though A_4 together with A_6 counter-attacks this attack, A_5 defends itself against this counter-attack. This explanation can also be interpreted in terms of literals: p is not in the answer set S because its derivation is "attacked" by a derivation of q, which is an admissible fragment of S. Even though the derivation of p and the derivation of u both "counter-attack" the derivation of q, attempting to defend p, the derivation of q can attack both counter-attacks and thus the derivation of q defends itself. Consequently, the attack of the derivation of q on the derivation of p "succeeds", which is the reason that p is not part of the answer set.

Using argumentation-theoretic concepts for the explanation of literals w.r.t. an answer set, may seem unintuitive to ASP-experts. Thus, we now define a second type of justification, which provides explanations in terms of literals and relations between them, rather than in terms of arguments as used in Attack Trees. The new type of justification is constructed from Attack Trees by flattening the structure of arguments occurring in an Attack Tree as well as of the attack relation between these arguments. In addition to reflecting logic programming concepts, an advantage of the new justifications is that they are finite even if constructed from infinite Attack Trees.

5.5 Basic ABA-Based Answer Set Justifications

In this section we define the basic concepts for constructing justifications of a literal k in terms of literals and their relations, based on Attack Trees of arguments with conclusion k. The idea is to extract the assumption- and fact-premises of each argument in the Attack Tree to express a support-relation between each of the premise-literals and the literal forming the conclusion of the argument. Furthermore, the attacks between arguments in an Attack Tree are translated into attack-relations between the literals forming the conclusions of these arguments. We first introduce some terminology to refer to the structure of an Attack Tree.

Notation 5.7. Let Υ be an Attack Tree and let N be a node in Υ. $arg(N)$ denotes the argument held by node N. If $arg(N)$ is $A : (AP, FP) \vdash k$, then $name(N) = A$, $conc(N) = k$, $AP(N) = AP$, $FP(N) = FP$, and $label(N)$ is either '+' or '$-$', depending on the label of A in Υ. The set of all child nodes of N in Υ is denoted $children(N)$.

5.5.1 Basic Justifications

We now define how to express the structure of an Attack Tree as a set of relations between literals.

Definition 5.8 (Basic Justification)**.** Let $Args \subseteq Ar_{\mathcal{P}}$, $A \in Ar_{\mathcal{P}}$, and $\Upsilon = attTree_{Args}(A)$ an Attack Tree of A w.r.t. $Args$. The *Basic Justification* of A w.r.t. Υ, denoted $justB_{\Upsilon}(A)$, is obtained as follows:
$justB_{\Upsilon}(A) = \bigcup_{N \textit{ in } \Upsilon}$
$\{supp\_rel(k, conc(N)) \mid k \in AP(N) \,\cup\, FP(N) \setminus \{conc(N)\}\} \,\cup$
$\{att\_rel(conc(M), k) \mid M \in children(N), conc(M) = \overline{k}\}$

Example 5.6. Consider the logic program $\mathcal{P}_8$ from Section 5.3 and the Attack Trees discussed in Example 5.1. Since $\Upsilon_1 = attTree^{+}_{\mathcal{E}_1}(A_{14})$ comprises only the node A^{+}_{14}, the Basic Justification of A_{14} w.r.t. Υ_1 is $justB_{\Upsilon_1}(A_{14}) = \emptyset$.

Now consider the negative Attack Tree $\Upsilon_2 = attTree^{-}_{\mathcal{E}_2}(A_{10})$ of A_{10} w.r.t. $\mathcal{E}_2$ depicted on the left of Figure 5.2. The Basic Justification of A_{10} w.r.t. Υ_2 is:

$$\begin{aligned} justB_{\Upsilon_2}(A_{10}) &= \{supp\_rel(\texttt{not } q, p),\ supp\_rel(\texttt{not } u, p),\ supp\_rel(\texttt{not } w, p)\} \cup \\ &\qquad \{att\_rel(w, \texttt{not } w)\} \\ &= \{supp\_rel(\texttt{not } q, p),\ supp\_rel(\texttt{not } u, p),\ supp\_rel(\texttt{not } w, p), \\ &\qquad att\_rel(w, \texttt{not } w)\} \end{aligned}$$

The following Basic Justification is obtained from the negative Attack Tree $\Upsilon_3 = attTree^{-}_{\mathcal{E}_2}(A_9)$ of A_9 w.r.t. the stable argument extension $\mathcal{E}_2$ (see Figure 5.3):

$$\begin{aligned} justB_{\Upsilon_3}(A_9) = \{&supp\_rel(\texttt{not } \neg p, p),\ att\_rel(\neg p, \texttt{not } \neg p),\ supp\_rel(\texttt{not } q, \neg p), \\ &supp\_rel(\texttt{not } u, \neg p),\ att\_rel(q, \texttt{not } q),\ att\_rel(u, \texttt{not } u), \\ &supp\_rel(\texttt{not } w, q),\ att\_rel(w, \texttt{not } w),\ supp\_rel(\texttt{not } \neg p, u)\} \end{aligned}$$

Note that even though Υ_3 is an infinite Attack Tree, the Basic Justification of A_9 w.r.t. Υ_3 is finite. In particular, when A_{11} reoccurs in the Attack Tree as an attacker of A_{13}, no new *att_rel* or *supp_rel* pairs are added to the Basic Justification: even though A_{11} attacks A_9 with conclusion p at its first occurrence and A_{13} with conclusion d at its second occurrence, no new *att_rel* pair is added since the attacked assumption is in both cases $\texttt{not } \neg p$.

In Basic Justifications, attacks between arguments are translated into “attacks” between literals, and supports of premises into “supports” of literals. In other words, a Basic Justification is the flattened version of an Attack Tree. Even though it provides an explanation in terms of literals rather than arguments, it is not sufficient to justify a literal w.r.t. an answer set for two reasons, as explained below.

Firstly, a Basic Justification does not contain the literal being justified, which is for example a problem when justifying a fact. When justifying a fact k, we construct an Attack

Tree of the fact-argument for k, which consists of only the root node $A^+ : (\emptyset, \{k\}) \vdash k$, leading to an empty Basic Justification. An empty set is not meaningful, so it would be useful if the literal in question was contained in the justification. Furthermore, a problem arises when trying to justify a literal for which no argument exists in the translated AA framework, i.e. a literal that cannot be derived in any way from the logic program. For such a literal, which is trivially not part of any answer set, it is not possible to construct an Attack Tree as no argument for this literal exists in the translated AA framework. Since a Basic Justification is constructed from an Attack Tree, there is no Basic Justification for such a literal. This is unsatisfying, so we would like to have some kind of justification, rather than to fail.

The second shortcoming of a Basic Justification is that it only provides one reason why a literal is not in an answer set as it is constructed from a single negative Attack Tree, which provides one explanation how the root argument is attacked by the set of arguments in question. However, it is more meaningful to capture all different explanations of how a literal "failed" to be in the answer set in question. Thus, we want the justification of a literal not in the answer set to consist of all possible Basic Justifications of this literal.

In order to overcome these two deficiencies, we introduce BABAS Justifications, which add the literal being justified to the Basic Justification set and provide a collection of all Basic Justifications for a literal that is not contained in an answer set.

5.5.2 BABAS Justifications

We now define the *Basic ABA-Based Answer Set (BABAS) Justification* of a literal w.r.t. an answer set, which is based on the Basic Justifications of an argument w.r.t. an Attack Tree. If a literal k is contained in an answer set, its BABAS Justification is constructed from one Basic Justification of one of the corresponding arguments of k. This is inspired by the result in Proposition 5.1 that if a literal k is part of an answer set, there exists some argument with conclusion k in the corresponding stable argument extension. Conversely, if k is not contained in an answer set, its BABAS Justification is constructed from all Basic Justifications of all arguments with conclusion k, expressing all reasons why k is not part of this answer set. Again, this choice is based on Proposition 5.1, stating that if a literal k is not part of an answer set, all arguments with conclusion k are not contained in the corresponding stable argument extension.

Definition 5.9 (Basic ABA-Based Answer Set Justification)**.** Let S be an answer set of $\mathcal{P}$ and let $\mathcal{E}$ be the corresponding stable argument extension of S in $AA_{\mathcal{P}}$.

1. Let $k \in S_{NAF}$, $A \in \mathcal{E}$ a corresponding argument of k, and $\Upsilon = attTree^+_{\mathcal{E}}(A)$ some positive Attack Tree of A w.r.t. $\mathcal{E}$. A *Positive BABAS Justification* of k w.r.t. S is: $justB^+_S(k) = \{k\} \cup justB_{\Upsilon}(A)$.

2. Let $k \notin S_{NAF}$, $A_1, \dots, A_n$ $(n \geq 0)$ all arguments with conclusion k in $Ar_{\mathcal{P}}$, and $\Upsilon_{11}, \dots, \Upsilon_{1m_1}, \dots, \Upsilon_{n1}, \dots, \Upsilon_{nm_n}$ $(m_1, \dots, m_n \geq 0)$ all negative Attack Trees of

$A_1, \ldots, A_n$ w.r.t. $\mathcal{E}$.

(a) If $n = 0$, then the *Negative BABAS Justification* of k w.r.t. S is: $justB^-_S(k) = \emptyset$.

(b) If $n > 0$, then the *Negative BABAS Justification* of k w.r.t. S is: $justB^-_S(k) = \{\{k\} \cup justB_{\Upsilon_{11}}(A_1), \ldots, \{k\} \cup justB_{\Upsilon_{1m_1}}(A_1), \ldots, \{k\} \cup justB_{\Upsilon_{nm_n}}(A_n)\}$.

Note that there can be more than one Positive BABAS Justification of a literal contained in an answer set, but only one Negative BABAS Justification of a literal not contained in an answer set. Note also that the Positive BABAS Justification is a set of *supp_rel* and *att_rel* pairs (plus the literal that is justified), whereas the Negative BABAS Justification is a set of sets containing these pairs (where each set also contains the literal that is justified).

A BABAS Justification can be represented as a graph, where all literals occurring in a *supp_rel* or *att_rel* pair form nodes, and the *supp_rel* and *att_rel* relations are edges between these nodes. For Negative BABAS Justifications, a separate graph for each set in the justification is given. In contrast, Positive BABAS Justifications are illustrated as a single graph.

Example 5.7. Based on the Basic Justifications in Example 5.6, we illustrate the construction of BABAS Justifications. Consider $w \in S_1$, where the corresponding stable argument extension of S_1 is $\mathcal{E}_1$ (see Section 5.3). There is only one corresponding argument of w in $\mathcal{E}_1$, namely $A_{14} : (\emptyset, \{w\}) \vdash w$, which has a unique Basic Justification $justB_{\Upsilon_1}(A_{14}) = \emptyset$. Therefore, a unique Positive BABAS Justification of w w.r.t. S_1 is $justB^+_{S_1}(w) = \{w\}$. This justification expresses that w is in the answer set S_1 because it is supported only by itself, in other words, it is a fact.

We now consider the BABAS Justification of $p \notin S_2$, where the corresponding stable argument extension of S_2 is $\mathcal{E}_2$. Since $p \notin S_2$, we examine all arguments with conclusion p in $Ar_{\mathcal{P}_8}$, that is A_9 and A_{10}. Both A_9 and A_{10} have a unique negative Attack Tree w.r.t. $\mathcal{E}_2$, $\Upsilon_3 = attTree^-_{\mathcal{E}_2}(A_9)$ (see Figure 5.3) and $\Upsilon_2 = attTree^-_{\mathcal{E}_2}(A_{10})$ (see left of Figure 5.2). From the Basic Justifications $justB_{\Upsilon_3}(A_9)$ and $justB_{\Upsilon_2}(A_{10})$ explained in Example 5.6, the BABAS Justification of p w.r.t. S_2 is obtained as follows:

$$\begin{aligned} justB^-_{S_2}(p) = \{\{&p, supp\_rel(\texttt{not}\ \neg p, p), att\_rel(\neg p, \texttt{not}\ \neg p), supp\_rel(\texttt{not}\ q, \neg p),\\ &supp\_rel(\texttt{not}\ u, \neg p), att\_rel(q, \texttt{not}\ q), att\_rel(u, \texttt{not}\ u),\\ &supp\_rel(\texttt{not}\ w, q), att\_rel(w, \texttt{not}\ w), supp\_rel(\texttt{not}\ \neg p, u)\},\\ &\{p, supp\_rel(\texttt{not}\ q, p), supp\_rel(\texttt{not}\ u, p), supp\_rel(\texttt{not}\ w, p),\\ &att\_rel(w, \texttt{not}\ w)\}\} \end{aligned}$$

Figure 5.10 depicts the graphical representation of the Negative BABAS Justification $justB^-_{S_2}(p)$, where the left of the figure represents the first set in $justB^-_{S_2}(p)$, and the

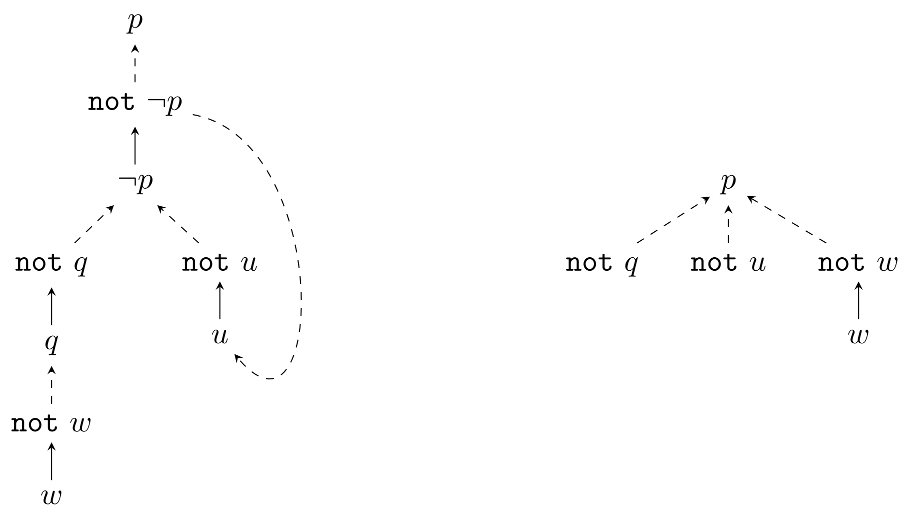


Figure 5.10: Graphical representation of the Negative BABAS Justification $justB^{-}_{S_2}(p)$ in Example 5.7. Dashed lines stand for *supp_rel* pairs in the BABAS Justification, whereas solid lines represent *att_rel* pairs.

right of the figure the second set.

So far, we only illustrated BABAS Justifications of literals k for which at least one argument exists. In general, the BABAS Justification of literals that do not have such an argument is the empty set.

Example 5.8. Consider the literal $\neg q \notin S_1$ in the logic program $\mathcal{P}_8$ (see Section 5.3). There is no clause with head $\neg q$ in $\mathcal{P}_8$, and consequently $Ar_{\mathcal{P}_8}$ does not comprise an argument with conclusion $\neg q$. Thus, there is no Attack Tree of an argument for $\neg q$ and no Basic Justification of an argument for $\neg q$. As a consequence, the Negative BABAS Justification of $\neg q$ w.r.t. S_1 is $justB^{-}_{S_1}(\neg q) = \emptyset$.

5.5.3 Shortcomings of BABAS Justifications

A BABAS Justification is a flat structure, which loses some information as compared to the underlying Attack Trees. Attack Trees label arguments w.r.t. a stable argument extension, expressing whether or not an argument is part of the stable argument extension. However, a BABAS Justification does not provide any information about whether or not a literal is contained in the answer set in question. Whether or not a literal is part of an answer set is important to know, since attacks and supports by literals contained in the answer set "succeed", whereas attacks and supports by literals not in the answer set do not "succeed".

Example 5.9. Consider the Negative BABAS Justification $justB^{-}_{S_2}(p)$ from Example 5.7 (see Figure 5.10). $justB^{-}_{S_2}(p)$ does not express whether or not the "attacking" literal $\neg p$ is part of S_2, neither in set notation nor in the graphical representation. In contrast, the underlying Attack Tree $attTree^{-}_{\mathcal{E}_2}(A_9)$ in Figure 5.3 specifies that the argument A_{11} for $\neg p$ is in the corresponding stable argument extension $\mathcal{E}_2$, by labelling A_{11} as '+'.

The next example illustrates another shortcoming of BABAS Justifications, which arises if the underlying Attack Tree contains different arguments that have the same conclusion and occur as child nodes of the same parent node.

Example 5.10. Consider the two logic programs $\mathcal{P}_{10}$ (left) and $\mathcal{P}_{11}$ (right):

$$\begin{aligned} \{\; & p \leftarrow \texttt{not}\ u; \\ & p \leftarrow \texttt{not}\ w; \\ & q \leftarrow \texttt{not}\ p; \\ & u \leftarrow\ ; \\ & w \leftarrow \;\} \end{aligned}$$

$$\begin{aligned} \{\; & p \leftarrow \texttt{not}\ u, \texttt{not}\ w; \\ & q \leftarrow \texttt{not}\ p; \\ & u \leftarrow\ ; \\ & w \leftarrow \;\} \end{aligned}$$

Both logic programs have only one answer set, $S_{\mathcal{P}_{10}} = S_{\mathcal{P}_{11}} = \{u, w, q\}$. The translated AA frameworks $AA_{\mathcal{P}_{10}}$ (left) and $AA_{\mathcal{P}_{11}}$ (right) have the following arguments:

$A_1 : (\{\texttt{not}\ u\}, \emptyset) \vdash \texttt{not}\ u$
$A_2 : (\{\texttt{not}\ \neg u\}, \emptyset) \vdash \texttt{not}\ \neg u$
$A_3 : (\{\texttt{not}\ w\}, \emptyset) \vdash \texttt{not}\ w$
$A_4 : (\{\texttt{not}\ \neg w\}, \emptyset) \vdash \texttt{not}\ \neg w$
$A_5 : (\{\texttt{not}\ p\}, \emptyset) \vdash \texttt{not}\ p$
$A_6 : (\{\texttt{not}\ \neg p\}, \emptyset) \vdash \texttt{not}\ \neg p$
$A_7 : (\{\texttt{not}\ q\}, \emptyset) \vdash \texttt{not}\ q$
$A_8 : (\{\texttt{not}\ \neg q\}, \emptyset) \vdash \texttt{not}\ \neg q$
$A_9 : (\{\texttt{not}\ p\}, \emptyset) \vdash q$
$A_{10} : (\emptyset, \{u\}) \vdash u$
$A_{11} : (\emptyset, \{w\}) \vdash w$
$A_{12} : (\{\texttt{not}\ u\}, \emptyset) \vdash p$
$A_{13} : (\{\texttt{not}\ w\}, \emptyset) \vdash p$

$A_1 : (\{\texttt{not}\ u\}, \emptyset) \vdash \texttt{not}\ u$
$A_2 : (\{\texttt{not}\ \neg u\}, \emptyset) \vdash \texttt{not}\ \neg u$
$A_3 : (\{\texttt{not}\ w\}, \emptyset) \vdash \texttt{not}\ w$
$A_4 : (\{\texttt{not}\ \neg w\}, \emptyset) \vdash \texttt{not}\ \neg w$
$A_5 : (\{\texttt{not}\ p\}, \emptyset) \vdash \texttt{not}\ p$
$A_6 : (\{\texttt{not}\ \neg p\}, \emptyset) \vdash \texttt{not}\ \neg p$
$A_7 : (\{\texttt{not}\ q\}, \emptyset) \vdash \texttt{not}\ q$
$A_8 : (\{\texttt{not}\ \neg q\}, \emptyset) \vdash \texttt{not}\ \neg q$
$A_9 : (\{\texttt{not}\ p\}, \emptyset) \vdash q$
$A_{10} : (\emptyset, \{u\}) \vdash u$
$A_{11} : (\emptyset, \{w\}) \vdash w$
$A_{14} : (\{\texttt{not}\ u, \texttt{not}\ w\}, \emptyset) \vdash p$

$AA_{\mathcal{P}_{10}}$ and $AA_{\mathcal{P}_{11}}$ share arguments A_1 to A_{11}. In addition, $AA_{\mathcal{P}_{10}}$ has arguments A_{12} and A_{13}, whereas $AA_{\mathcal{P}_{11}}$ has only one additional argument A_{14}. Both AA frameworks have a unique stable argument extension, $\mathcal{E}_{\mathcal{P}_{10}} = \mathcal{E}_{\mathcal{P}_{11}} = \{A_2, A_4, A_5, A_6, A_8, A_9, A_{10}, A_{11}\}$. $\mathcal{E}_{\mathcal{P}_{10}}$ is the corresponding stable argument extension of $S_{\mathcal{P}_{10}}$ and $\mathcal{E}_{\mathcal{P}_{11}}$ the corresponding stable argument extension of $S_{\mathcal{P}_{11}}$. We now examine the BABAS Justifications of q w.r.t. $S_{\mathcal{P}_{10}}$ and $S_{\mathcal{P}_{11}}$ by constructing Attack Trees of the corresponding arguments of q w.r.t. $\mathcal{E}_{\mathcal{P}_{10}}$ and $\mathcal{E}_{\mathcal{P}_{11}}$, respectively. In both $AA_{\mathcal{P}_{10}}$ and $AA_{\mathcal{P}_{11}}$, the only corresponding argument of q is A_9, which has a unique positive Attack Tree w.r.t. $\mathcal{E}_{\mathcal{P}_{10}}$ ($attTree^{+}_{\mathcal{E}_{\mathcal{P}_{10}}}(A_9)$), depicted in Figure 5.11, and two positive Attack Trees w.r.t. $\mathcal{E}_{\mathcal{P}_{11}}$ ($attTree^{+}_{\mathcal{E}_{\mathcal{P}_{11}}}(A_9)_1$ and $attTree^{+}_{\mathcal{E}_{\mathcal{P}_{11}}}(A_9)_2$), depicted in Figure 5.12. The unique Positive BABAS Justification of q w.r.t. $S_{\mathcal{P}_{10}}$ constructed from $attTree^{+}_{\mathcal{E}_{\mathcal{P}_{10}}}(A_9)$ and the two possible Positive BABAS Justifications of q w.r.t. $S_{\mathcal{P}_{11}}$ constructed from $attTree^{+}_{\mathcal{E}_{\mathcal{P}_{11}}}(A_9)_1$ and $attTree^{+}_{\mathcal{E}_{\mathcal{P}_{11}}}(A_9)_2$,

$$\begin{tikzcd}
 & A_9^+ : (\{\texttt{not}\ p\}, \emptyset) \vdash q & \\
A_{12}^- : (\{\texttt{not}\ u\}, \emptyset) \vdash p \arrow[ur] & & A_{13}^- : (\{\texttt{not}\ w\}, \emptyset) \vdash p \arrow[ul] \\
A_{10}^+ : (\emptyset, \{u\}) \vdash u \arrow[u] & & A_{11}^+ : (\emptyset, \{w\}) \vdash w \arrow[u]
\end{tikzcd}$$

Figure 5.11: The unique positive Attack Tree $attTree^+_{\mathcal{E}_{\mathcal{P}_{10}}}(A_9)$ of A_9 w.r.t. $\mathcal{E}_{\mathcal{P}_{10}}$ (see Example 5.10).

$$\begin{tikzcd}
A_9^+ : (\{\texttt{not}\ p\}, \emptyset) \vdash q & A_9^+ : (\{\texttt{not}\ p\}, \emptyset) \vdash q \\
A_{14}^- : (\{\texttt{not}\ u, \texttt{not}\ w\}, \emptyset) \vdash p \arrow[u] & A_{14}^- : (\{\texttt{not}\ u, \texttt{not}\ w\}, \emptyset) \vdash p \arrow[u] \\
A_{10}^+ : (\emptyset, \{u\}) \vdash u \arrow[u] & A_{11}^+ : (\emptyset, \{w\}) \vdash w \arrow[u]
\end{tikzcd}$$

Figure 5.12: The two positive Attack Trees $attTree^+_{\mathcal{E}_{\mathcal{P}_{11}}}(A_9)_1$ (left) and $attTree^+_{\mathcal{E}_{\mathcal{P}_{11}}}(A_9)_2$ (right) of A_9 w.r.t. $\mathcal{E}_{\mathcal{P}_{11}}$ (see Example 5.10).

respectively, are:

$$\begin{aligned} justB^+_{S_{\mathcal{P}_{10}}}(q) = \{&q,\ supp\_rel(\texttt{not}\ p, q),\ att\_rel(p, \texttt{not}\ p),\ supp\_rel(\texttt{not}\ u, p), \\ &att\_rel(u, \texttt{not}\ u),\ supp\_rel(\texttt{not}\ w, p),\ att\_rel(w, \texttt{not}\ w)\}\end{aligned}$$

$$\begin{aligned} justB^+_{S_{\mathcal{P}_{11}}}(q) = \{&q,\ supp\_rel(\texttt{not}\ p, q),\ att\_rel(p, \texttt{not}\ p),\ supp\_rel(\texttt{not}\ u, p), \\ &supp\_rel(\texttt{not}\ w, p),\ att\_rel(u, \texttt{not}\ u)\}\end{aligned}$$

$$\begin{aligned} justB^+_{S_{\mathcal{P}_{11}}}(q) = \{&q,\ supp\_rel(\texttt{not}\ p, q),\ att\_rel(p, \texttt{not}\ p),\ supp\_rel(\texttt{not}\ u, p), \\ &supp\_rel(\texttt{not}\ w, p),\ att\_rel(w, \texttt{not}\ w)\}\end{aligned}$$

The graphical representations of these BABAS Justifications are depicted in Figure 5.13. All of them give the impression that p is supported by $\texttt{not}\ u$ and $\texttt{not}\ w$ together, which is only correct in the case of $\mathcal{P}_{11}$. In $\mathcal{P}_{10}$, there are two different ways of concluding p, one supported by the NAF literal $\texttt{not}\ u$, and the other one by $\texttt{not}\ w$, which is not clear from $justB^+_{S_{\mathcal{P}_{10}}}(q)$.

Example 5.10 suggests that if a node in an Attack Tree has various children holding arguments with the same conclusion, these child nodes should be distinguished in a justification. We address this problem in the next section by defining a more elaborate version of ABA-Based Answer Set Justifications.

$$\begin{tikzcd}
 & q & \\
 & \texttt{not}\ p \arrow[u, dashed] & \\
 & p \arrow[u] & \\
\texttt{not}\ u \arrow[ur, dashed] & & \texttt{not}\ w \arrow[ul, dashed] \\
u \arrow[u] & & w \arrow[u]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & q & \\
 & \texttt{not}\ p \arrow[u, dashed] & \\
 & p \arrow[u] & \\
\texttt{not}\ u \arrow[ur, dashed] & & \texttt{not}\ w \arrow[ul, dashed] \\
u \arrow[u] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & q & \\
 & \texttt{not}\ p \arrow[u, dashed] & \\
 & p \arrow[u] & \\
\texttt{not}\ u \arrow[ur, dashed] & & \texttt{not}\ w \arrow[ul, dashed] \\
 & & w \arrow[u]
\end{tikzcd}$$

Figure 5.13: The unique Positive BABAS Justification $justB^{+}_{S_{\mathcal{P}_{10}}}(q)$ (left) and the two possible Positive BABAS Justifications $justB^{+}_{S_{\mathcal{P}_{11}}}(q)$ (middle and right) from Example 5.10.

5.6 Labelled ABA-Based Answer Set Justifications

We now introduce *Labelled ABA-Based Answer Set (LABAS) Justifications*, which address the shortcomings of BABAS Justifications by labelling the relations and literals in the justification as either '+' or '−', depending on the labels of arguments in the underlying Attack Trees. In addition, literals can have an *asm* or *fact* tag, indicating that they are used as assumptions or facts, respectively. Non-assumption and non-fact literals are tagged with their argument's name in order to distinguish between different arguments with the same conclusion occurring in an Attack Tree. We refer to the structure of nodes in an Attack Tree as introduced in Notation 5.7. Similarly to BABAS Justifications, LABAS Justifications are defined in terms of *Labelled Justifications*, which are a flattened version of Attack Trees. In contrast to Basic Justifications, Labelled Justifications label the literals and relations extracted from an Attack Tree, and extract only relevant support relations.

5.6.1 Labelled Justifications

A Labelled Justification assigns the label '+' to all facts and NAF literals occurring as premises of an argument labelled '+' in the Attack Tree, as well as to this argument's conclusion. A Labelled Justification assigns the label '−' to the conclusion of an argument labelled '−' in the Attack Tree as well as to some NAF literals supporting this argument, namely to those NAF literals whose contrary is the conclusion of a child node of this argument in the Attack Tree. Attack and support relations are labelled '+' if the first literal in the relation is labelled '+', and labelled '−' if the first literal in the relation is labelled '−'. Since the labels in a Labelled Justification depend on the labels of arguments in an Attack Tree, the definition is split into two cases: one for nodes holding arguments labelled '+' in the Attack Tree, and the other for nodes holding arguments labelled '−' in the Attack Tree.

Definition 5.10 (Labelled Justification)**.** Let $Args \subseteq Ar_{\mathcal{P}}$, $A \in Ar_{\mathcal{P}}$, and let $\Upsilon = attTree_{Args}(A)$ be an Attack Tree of A w.r.t. $Args$. The *Labelled Justification* of A

w.r.t. Υ, denoted $justL_{\Upsilon}(A)$, is obtained as follows:
$justL_{\Upsilon}(A) =$
$\bigcup_{N \text{ } in \text{ } \Upsilon, \, label(N)=+}$

$$\begin{array}{ll}
\{supp\_rel^{+}(k^{+}_{asm}, conc(N)^{+}_{A_N}) & \mid k \in AP(N) \backslash conc(N), name(N) = A_N\} \cup \\
\{supp\_rel^{+}(k^{+}_{fact}, conc(N)^{+}_{A_N}) & \mid k \in FP(N) \backslash conc(N), name(N) = A_N\} \cup \\
\{att\_rel^{-}(conc(M)^{-}_{A_M}, k^{+}_{asm}) & \mid M \in children(N), conc(M) = \overline{k}, \\
 & \; name(M) = A_M\} \cup
\end{array}$$

$\bigcup_{N \text{ } in \text{ } \Upsilon, \, label(N)=-}$

$$\begin{array}{ll}
\{supp\_rel^{-}(k^{-}_{asm}, conc(N)^{-}_{A_N}) & \mid k \in AP(N) \backslash conc(N), children(N) = \{M\}, \\
 & \; conc(M) = \overline{k}, name(N) = A_N\} \cup \\
\{att\_rel^{+}(conc(M)^{+}_{fact}, k^{-}_{asm}) & \mid children(N) = \{M\}, conc(M) = \overline{k}, \\
 & \; FP(M) = \{conc(M)\}, AP(M) = \emptyset\} \cup \\
\{att\_rel^{+}(conc(M)^{+}_{A_M}, k^{-}_{asm}) & \mid children(N) = \{M\}, conc(M) = \overline{k}, AP(M) \neq \emptyset \\
 & \quad or \; FP(M) \neq \{conc(M)\}, name(M) = A_M\}
\end{array}$$

To illustrate Labelled Justifications and the differences with Basic Justification, we construct the Labelled Justifications for some of the arguments we used for Basic Justifications in Example 5.6.

Example 5.11. The Labelled Justification of $A_{14} : (\emptyset, \{w\}) \vdash w$ w.r.t. the positive Attack Tree $\Upsilon_1 = attTree^{+}_{\mathcal{E}_1}(A_{14})$ is the empty set, exactly as for the Basic Justification: $justL_{\Upsilon_1}(A_{14}) = justB_{\Upsilon_1}(A_{14}) = \emptyset$. The reason is that A_{14} is labelled '+' in Υ_1, but none of the three conditions for nodes with label '+' in Definition 5.10 is satisfied.

Now consider the Labelled Justification of A_{10} w.r.t. the negative Attack Tree $\Upsilon_2 = attTree^{-}_{\mathcal{E}_2}(A_{10})$:

$$\begin{aligned}
justL_{\Upsilon_2}(A_{10}) &= \{supp\_rel^{-}(\texttt{not } w^{-}_{asm}, p^{-}_{A_{10}})\} \cup \{att\_rel^{+}(w^{+}_{fact}, \texttt{not } w^{-}_{asm})\} \\
&= \{supp\_rel^{-}(\texttt{not } w^{-}_{asm}, p^{-}_{A_{10}}),\ att\_rel^{+}(w^{+}_{fact}, \texttt{not } w^{-}_{asm})\}
\end{aligned}$$

This Labelled Justification contains fewer literal-pairs than the Basic Justification of A_{10} w.r.t. Υ_2 (see Example 5.6), which additionally comprises supports of `not` q and `not` u for p. Since these two supports are not necessary to explain why p is not in S_2 (the explanation is that the supporting literal `not` w is attacked by the fact w), they are omitted in the Labelled Justification.

The procedure of extracting attack and support relations from an Attack Tree in the construction of a Labelled Justification is similar to the method of Basic Justifications, where the relations are extracted step by step for every node in the Attack Tree. The main difference of Labelled Justifications is that nodes holding arguments labelled '$+$' and nodes holding arguments labelled '$-$' in an Attack Tree are handled separately in order to obtain the correct labelling of literals and relations in the justification. Furthermore, the extraction of the support relation is divided into two cases: one for assumption-premises, and one for fact-premises. Similarly, there are two cases for the extraction of the attack relation: the attacker can be a fact or another (non-fact and non-assumption) literal. Note that not all supporting literals of an argument with label '$-$' are extracted for a Labelled Justification, but only the "attacked" ones.

5.6.2 LABAS Justifications

In this section, we define the *Labelled ABA-Based Answer Set (LABAS) Justification* of a literal w.r.t. an answer set, which is based on the Labelled Justifications of an argument for this literal w.r.t. an Attack Tree. We also prove that a LABAS Justification provides an explanation for a literal using an admissible fragment of the answer set in question.

Just as for BABAS Justifications, if a literal k is contained in an answer set, its LABAS Justification is constructed from one Labelled Justification of one of the corresponding arguments of k. Conversely, if k is not in an answer set, its LABAS Justification is constructed from all Labelled Justifications of all arguments with conclusion k. The only difference in the construction is that the literal being justified is labelled before it is added to the justification.

Recall that the translated ABA framework of $\mathcal{P}$ is $ABA_{\mathcal{P}}$.

Definition 5.11 (Labelled ABA-Based Answer Set Justification)**.** Let S be an answer set of $\mathcal{P}$ and $\mathcal{E}$ the corresponding stable argument extension of S in $AA_{\mathcal{P}}$.

1. Let $k \in S_{NAF}$, $A \in \mathcal{E}$ a corresponding argument of k, and $\Upsilon = attTree^{+}_{\mathcal{E}}(A)$ some positive Attack Tree of A w.r.t. $\mathcal{E}$. Let $lab(k) = k^{+}_{asm}$ if $k \in \mathcal{A}_{\mathcal{P}}$, $lab(k) = k^{+}_{fact}$ if $k \leftarrow \in \mathcal{R}_{\mathcal{P}}$, and $lab(k) = k^{+}_{A}$ else. A *Positive LABAS Justification* of k w.r.t. S is: $justL^{+}_{S}(k) = \{lab(k)\} \cup justL_{\Upsilon}(A)$.

2. Let $k \notin S_{NAF}$, $A_1, \dots, A_n$ $(n \geq 0)$ all arguments with conclusion k in $Ar_{\mathcal{P}}$, and $\Upsilon_{11}, \dots, \Upsilon_{1m_1}, \dots, \Upsilon_{n1}, \dots, \Upsilon_{nm_n}$ $(m_1, \dots, m_n \geq 0)$ all negative Attack Trees of $A_1, \dots, A_n$ w.r.t. $\mathcal{E}$.

   (a) If $n = 0$, then the *Negative LABAS Justification* of k w.r.t. S is: $justL^{-}_{S}(k) = \emptyset$.

   (b) If $n > 0$, then let $lab(k_1) = \dots = lab(k_n) = k^{-}_{asm}$ if $k \in \mathcal{A}_{\mathcal{P}}$ and $lab(k_1) = k^{-}_{A_1}, \dots, lab(k_n) = k^{-}_{A_n}$ else. The *Negative LABAS Justification* of k w.r.t. S is: $justL^{-}_{S}(k) = \{\{lab(k_1)\} \cup justL_{\Upsilon_{11}}(A_1), \dots, \{lab(k_n)\} \cup justL_{\Upsilon_{nm_n}}(A_n)\}$.

Example 5.12. We illustrate the advantages of LABAS Justifications as compared to BABAS Justifications by justifying the same literal as in Example 5.10, i.e. $q \in S_{\mathcal{P}_{10}}$ and $q \in S_{\mathcal{P}_{11}}$ of the logic programs $\mathcal{P}_{10}$ and $\mathcal{P}_{11}$. The LABAS Justifications are constructed from the same Attack Trees as the BABAS Justifications (see Figures 5.11 and 5.12). The unique Positive LABAS Justification of q w.r.t. $S_{\mathcal{P}_{10}}$ and the two possible Positive LABAS Justifications of q w.r.t. $S_{\mathcal{P}_{11}}$ are:

$$\begin{aligned} justL^{+}_{S_{\mathcal{P}_{10}}}(q) = \{ & q^{+}_{A_9},\ supp\_rel^{+}(\texttt{not}\ p^{+}_{asm}, q^{+}_{A_9}),\ att\_rel^{-}(p^{-}_{A_{12}}, \texttt{not}\ p^{+}_{asm}), \\ & att\_rel^{-}(p^{-}_{A_{13}}, \texttt{not}\ p^{+}_{asm}),\ supp\_rel^{-}(\texttt{not}\ u^{-}_{asm}, p^{-}_{A_{12}}), \\ & att\_rel^{+}(u^{+}_{fact}, \texttt{not}\ u^{-}_{asm}),\ supp\_rel^{-}(\texttt{not}\ w^{-}_{asm}, p^{-}_{A_{13}}), \\ & att\_rel^{+}(w^{+}_{fact}, \texttt{not}\ w^{-}_{asm})\} \end{aligned}$$

$$\begin{aligned} justL^{+}_{S_{\mathcal{P}_{11}}}(q) = \{ & q^{+}_{A_9},\ supp\_rel^{+}(\texttt{not}\ p^{+}_{asm}, q^{+}_{A_9}),\ att\_rel^{-}(p^{-}_{A_{14}}, \texttt{not}\ p^{+}_{asm}), \\ & supp\_rel^{-}(\texttt{not}\ u^{-}_{asm}, p^{-}_{A_{14}}), att\_rel^{+}(u^{+}_{fact}, \texttt{not}\ u^{-}_{asm})\} \end{aligned}$$

$$\begin{aligned} justL^{+}_{S_{\mathcal{P}_{11}}}(q) = \{ & q^{+}_{A_9},\ supp\_rel^{+}(\texttt{not}\ p^{+}_{asm}, q^{+}_{A_9}),\ att\_rel^{-}(p^{-}_{A_{14}}, \texttt{not}\ p^{+}_{asm}), \\ & supp\_rel^{-}(\texttt{not}\ w^{-}_{asm}, p^{-}_{A_{14}}), att\_rel^{+}(w^{+}_{fact}, \texttt{not}\ w^{-}_{asm})\} \end{aligned}$$

The graphical representations of these LABAS Justifications are depicted in Figure 5.14. The differences between BABAS and LABAS Justifications can be easily spotted when comparing the BABAS Justification graphs in Figure 5.13 with the LABAS Justification graphs in Figure 5.14, both of which explain why q is part of $S_{\mathcal{P}_{10}}$ and $S_{\mathcal{P}_{11}}$. In contrast to the BABAS Justifications, the LABAS Justifications express that in $\mathcal{P}_{10}$ there are two different ways of deriving p, one supported by `not` u (yielding A_{12}) and the other one by `not` w (yielding A_{13}), but in $\mathcal{P}_{11}$ there is only one way of deriving p, supported by both `not` u and `not` w (yielding A_{14}). The reason that neither of the two LABAS Justifications of q w.r.t. $S_{\mathcal{P}_{11}}$ comprises both of these supporting NAF literals is that LABAS Justifications only contain the supporting NAF literals that are "attacked"; in the first case `not` u is "attacked" by u, in the second case `not` w is "attacked" by w.

As illustrated by Example 5.12, LABAS Justifications solve the shortcomings of BABAS Justifications: They indicate whether or not support and attack relations "succeed", as well as which literals are facts or assumptions. Furthermore, tagging literals with argument-names makes it possible to distinguish between different ways of deriving the same literal. In addition, a LABAS Justification is sometimes shorter than the respective BABAS Justification, only comprising relevant supporting literals of a literal not in the answer set in question.

Example 5.13. Recall Dr. Smith who has to determine whether to follow his own decision to treat the short-sightedness of his patient Peter with laser surgery or whether to act according to the suggestion of his decision support system and treat Peter with intraocular lenses (see Section 5.3). In Example 5.2, we illustrated how Attack Trees can be used to explain the suggestion of the decision support system as well as why Dr. Smith's decision

$$\begin{tikzcd}
 & q^{+}_{A_9} & \\
 & \texttt{not}\ p^{+}_{asm} \arrow[u, dashed, "+"] & \\
p^{-}_{A_{12}} \arrow[ur, "-"] & & p^{-}_{A_{13}} \arrow[ul, "-"'] \\
\texttt{not}\ u^{-}_{asm} \arrow[u, dashed, "-"] & & \texttt{not}\ w^{-}_{asm} \arrow[u, dashed, "-"] \\
u^{+}_{fact} \arrow[u, "+"] & & w^{+}_{fact} \arrow[u, "+"]
\end{tikzcd}
\qquad
\begin{tikzcd}
q^{+}_{A_9} \\
\texttt{not}\ p^{+}_{asm} \arrow[u, dashed, "+"] \\
p^{-}_{A_{14}} \arrow[u, "-"] \\
\texttt{not}\ u^{-}_{asm} \arrow[u, dashed, "-"] \\
u^{+}_{fact} \arrow[u, "+"]
\end{tikzcd}
\qquad
\begin{tikzcd}
q^{+}_{A_9} \\
\texttt{not}\ p^{+}_{asm} \arrow[u, dashed, "+"] \\
p^{-}_{A_{14}} \arrow[u, "-"] \\
\texttt{not}\ w^{-}_{asm} \arrow[u, dashed, "-"] \\
w^{+}_{fact} \arrow[u, "+"]
\end{tikzcd}$$

Figure 5.14: The unique Positive LABAS Justification $justL^{+}_{S_{\mathcal{P}_{10}}}(q)$ (left) and the two Positive LABAS Justifications $justL^{+}_{S_{\mathcal{P}_{11}}}(q)$ (middle and right) from Example 5.12.

is wrong. Here, we demonstrate the LABAS Justifications explaining this.

Figure 5.15 displays the Negative LABAS Justification of the literal *laserSurgery*, which is not contained in the answer set S_{doctor} of the logic program $\mathcal{P}_{doctor}$ (see Section 5.3). This LABAS Justification is constructed from all Labelled Justifications of all arguments with conclusion *laserSurgery*, i.e. from all Attack Trees for arguments with conclusion *laserSurgery*. There is only one argument with conclusion *laserSurgery*, but there are two different negative Attack Trees for this argument (see Example 5.2). The negative Attack Tree underlying the left part of the LABAS Justification in Figure 5.15 was illustrated in Figure 5.4. The Negative LABAS Justification of *laserSurgery* expresses that Peter should not have laser surgery for two reasons: first (left part), because laser surgery should only be used if the patient is not tight on money, but Peter is tight on money as he is a student and as there is no evidence that his parents are rich; and second (right part), because laser surgery should only be used if it has not been decided that the patient should have corrective lenses, but there is evidence that Peter should have corrective lenses since he is short-sighted and since there is evidence against having laser surgery (and assuming that the patient does not have laser surgery is a prerequisite for having corrective lenses).

A Positive LABAS Justification explaining why Peter should get intraocular lenses is displayed in Figure 5.16. This LABAS Justification expresses that all supporting assumptions needed to draw the conclusion that Peter should have intraocular lenses are satisfied, namely Peter is short-sighted, he should not have laser surgery, he should not have glasses, and he should not have contact lenses. The explanation also illustrates why these other treatments are not applicable.

Using the LABAS Justifications, Dr. Smith can now understand why the decision support system suggested intraocular lenses as the best treatment for Peter and why Peter should not have laser surgery. Dr. Smith can therefore easily revise his original decision

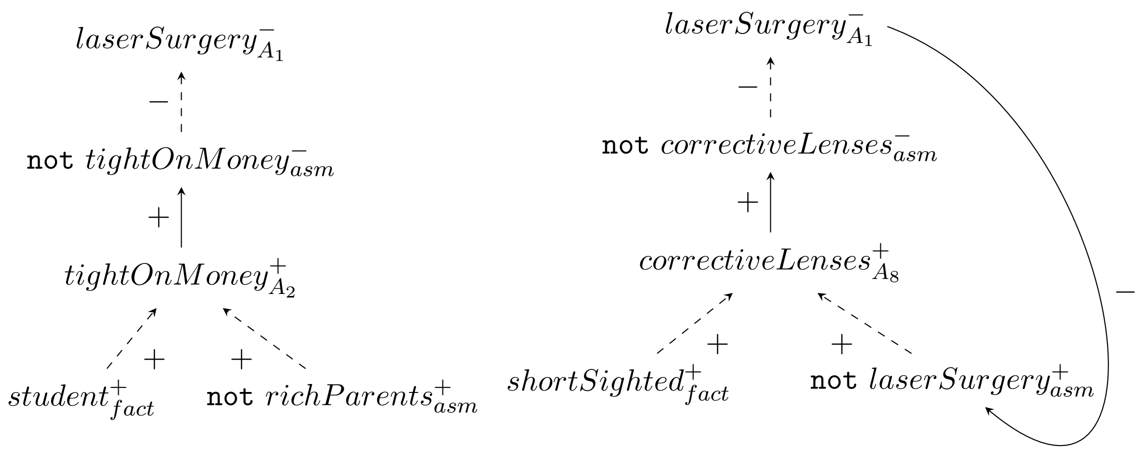


Figure 5.15: The Negative LABAS Justification of *laserSurgery* w.r.t. S_{doctor} of the logic program $\mathcal{P}_{doctor}$ as explained in Example 5.13.

that Peter should have laser surgery, realising that he forgot to consider that Peter is a student and that consequently Peter has not enough money to pay for laser surgery.

In the following, we show that LABAS Justifications explain a literal w.r.t. an answer set in terms of an admissible fragment of this answer set. We first introduce some terminology to refer to the literals in a LABAS Justification.

Notation 5.12. Let $justL^+_S(k)$ be a Positive LABAS Justification. We say that a literal k_1 *occurs in* $justL^+_S(k)$ if and only if $k_1 = k$ or k_1 is one of the literals in a support- or attack-pair in $justL^+_S(k)$. We say that k_1 *occurs positively* in $justL^+_S(k)$ if and only if it occurs as $k^+_{1_{asm}}$, $k^+_{1_{fact}}$, or $k^+_{1_A}$ (where A is some argument with conclusion k_1).

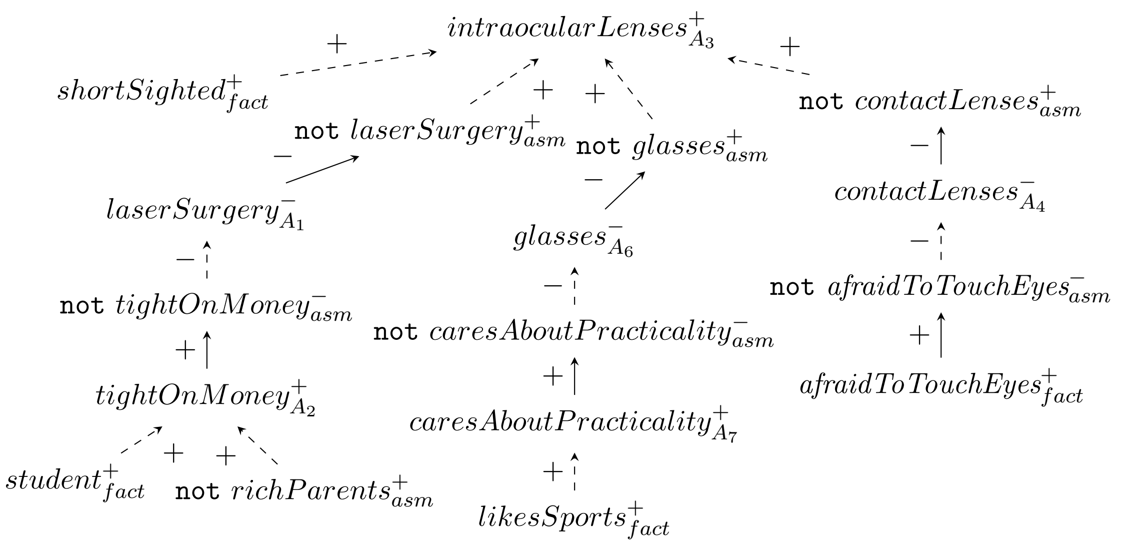
We use analogous terminology for Negative LABAS Justifications.

The following theorem characterises the explanations given by Positive LABAS Justifications.

Theorem 5.13. *Let $justL^+_S(k_1)$ be a Positive LABAS Justification of some literal k_1 w.r.t. an answer set S of $\mathcal{P}$. Let $NAF^+ = \{k \mid k^+_{asm}$ occurs in $justL^+_S(k_1)\}$ be the set of all NAF literals occurring positively in $justL^+_S(k_1)$. Then*

- *$\mathcal{P} \cup NAF^+$ is an admissible scenario of $\mathcal{P}$ in the sense of [DR91];*
- *$NAF^+ \subseteq S_{NAF}$.*

Proof. By Definitions 5.10 and 5.11 and Notation 5.12, NAF^+ is the union of all assumptions supporting arguments labelled '+' in the Attack Tree $attTree^+_{\mathcal{E}}(A)$ used for the construction of $justL^+_S(k_1)$, where $\mathcal{E}$ is the corresponding stable argument extension of S and $A \in \mathcal{E}$ is a corresponding argument of k_1. So $NAF^+ = Asms$ as defined in Theorem 5.12. □



$intraocularLenses^{+}_{A_3}$
$shortSighted^{+}_{fact}$
not $laserSurgery^{+}_{asm}$
not $glasses^{+}_{asm}$
not $contactLenses^{+}_{asm}$
$laserSurgery^{-}_{A_1}$
$glasses^{-}_{A_6}$
$contactLenses^{-}_{A_4}$
not $tightOnMoney^{-}_{asm}$
not $caresAboutPracticality^{-}_{asm}$
not $afraidToTouchEyes^{-}_{asm}$
$tightOnMoney^{+}_{A_2}$
$caresAboutPracticality^{+}_{A_7}$
$afraidToTouchEyes^{+}_{fact}$
$student^{+}_{fact}$
not $richParents^{+}_{asm}$
$likesSports^{+}_{fact}$
+
−

Figure 5.16: A Positive LABAS Explanation of *intraocularLenses* w.r.t. S_{doctor} of the logic program $\mathcal{P}_{doctor}$ as explained in Example 5.13.

This result expresses that LABAS Justifications explain that a literal is contained in an answer set because this literal is supported and defended by the answer set. However, LABAS Justifications do not simply provide the whole answer set as an explanation, but instead use an admissible fragment of it. A similar result can be formulated for Negative LABAS Justifications.

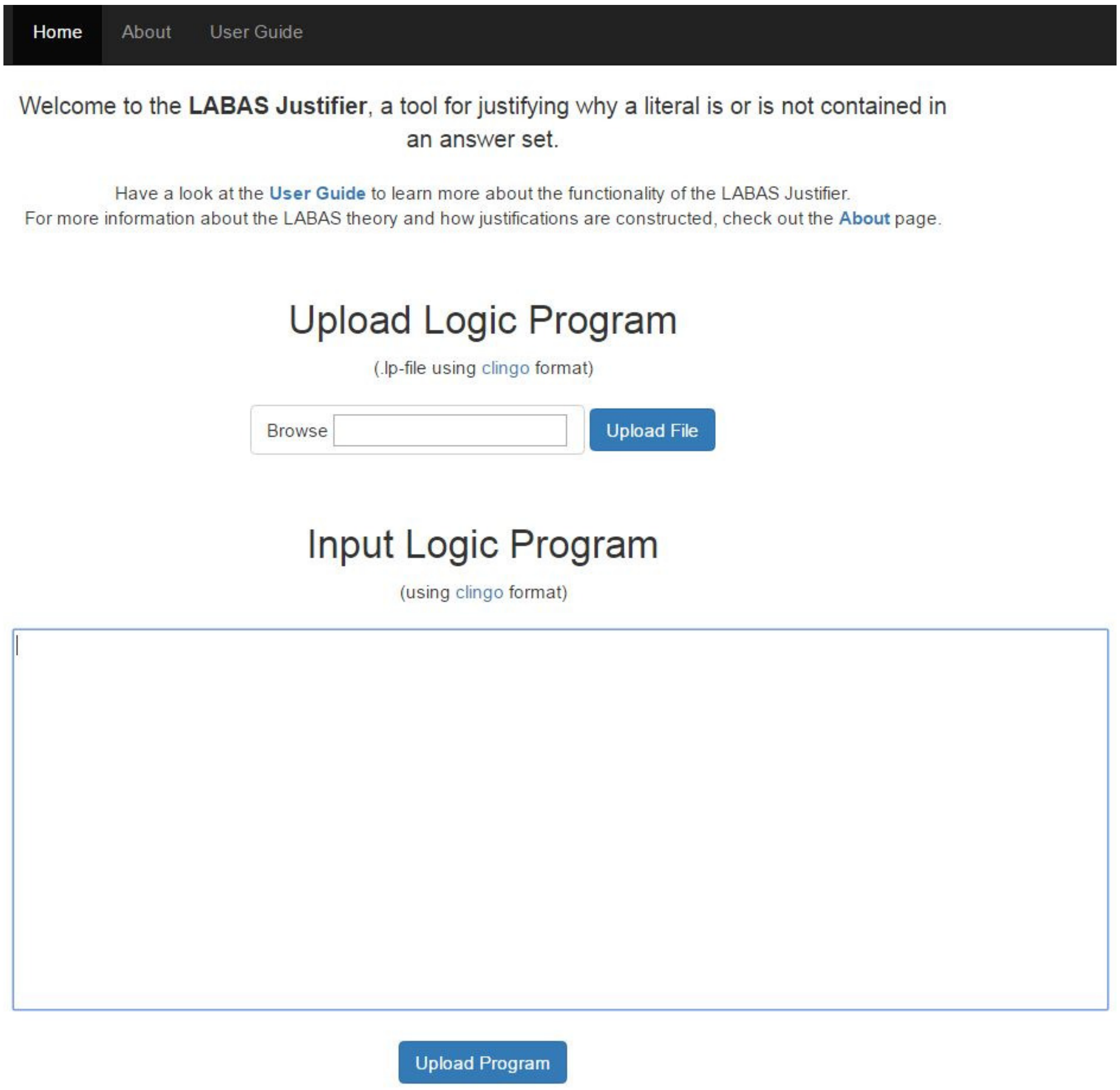
Theorem 5.14. *Let* $justL^{-}_{S}(k_1)$ *be a Negative LABAS Justification of a literal* k_1 *w.r.t. an answer set* S *of* $\mathcal{P}$. *Let* $NAF^{+}_{11}, \ldots, NAF^{+}_{1m_1}, \ldots, NAF^{+}_{n1}, \ldots, NAF^{+}_{nm_n}$ *be the sets of all NAF literals occurring positively in the subsets of* $justL^{-}_{S}(k_1)$, *i.e.* $NAF^{+}_{ij} = \{k \mid k^{+}_{asm}$ *occurs in* $lab(k_{1_i}) \cup justL_{\Upsilon_{ij}}(A_i)\}$ *where* $0 \leq i \leq n$ *and* $0 \leq j \leq m_n$. *Then for each* NAF^{+}_{ij}

- $\mathcal{P} \cup NAF^{+}_{ij}$ *is an admissible scenario of* $\mathcal{P}$ *in the sense of [DR91];*
- $NAF^{+}_{ij} \subseteq S_{NAF}$.

Proof. Analogous to the proof of Theorem 5.13. □

This means that the LABAS Justification of a literal that is not part of an answer set explains all different ways in which this literal is "attacked" by an admissible fragment of the answer set.

In summary, LABAS Justifications use the same information for an explanation as Attack Trees, namely an admissible fragment of an answer set, but expressing these information in terms of literals and the support and "attack" relations between them rather than in terms of arguments and attacks. Thus, LABAS Justifications are more suitable explanations if logic programming concepts are desired.



Welcome to the **LABAS Justifier**, a tool for justifying why a literal is or is not contained in an answer set.

Have a look at the **User Guide** to learn more about the functionality of the LABAS Justifier.
For more information about the LABAS theory and how justifications are constructed, check out the **About** page.

Upload Logic Program

(.lp-file using clingo format)

Input Logic Program

(using clingo format)

Figure 5.17: LABAS Justifier home page.

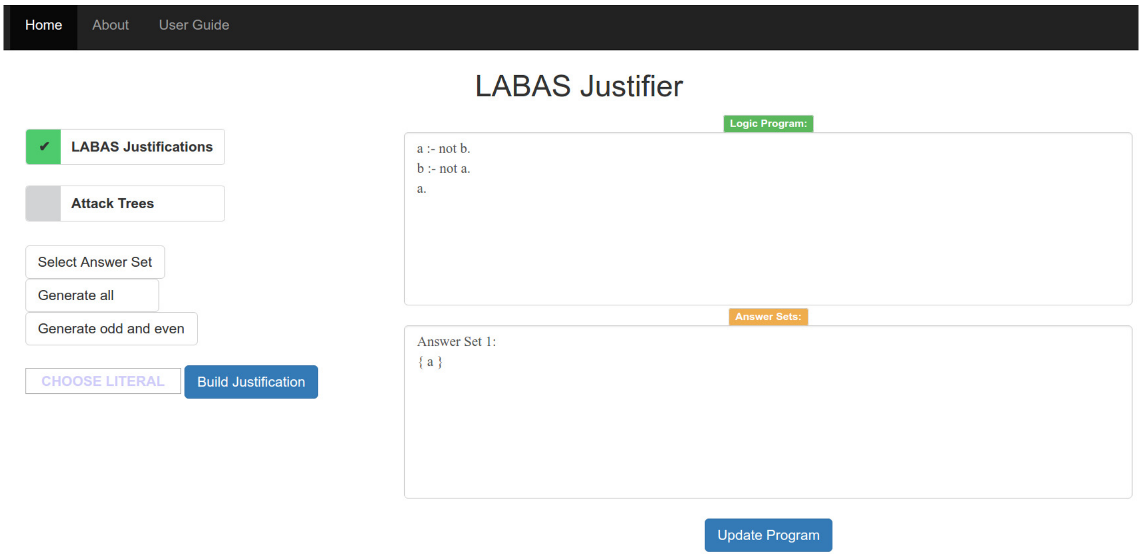
5.7 The LABAS Justifier

We implemented LABAS Justifications and Attack Trees in a web platform, called the *LABAS Justifier*[2], as a proof of concept and to allow users to gain a better understanding of our justification methods. The LABAS Justifier is hosted on the Heroku cloud platform [MS13, Han14], making it easily accessible and independent of the user's operating system.

5.7.1 Functionality

On the home page (see Figure 5.17), the user can either input a logic program manually in a text box or upload a logic program as a ".lp" file, as required by the ASP solver clingo [GKK+11, GKKS14].

[2]http://labas-justification.herokuapp.com/



```
a :- not b.
b :- not a.
a.
```

```
Answer Set 1:
{ a }
```

Figure 5.18: Main justification interface of the LABAS Justifier.

As is standard for ASP solvers, the implication ($\leftarrow$) in clauses is written as ":-" and every clause has to be followed by a full stop. Furthermore, facts such as $a \leftarrow$ are written without implication symbol as "a.", explicit negation $\neg$ is denoted by "-" and NAF `not` by "not". Note that the LABAS Justifier currently does not support logic programs with variables.

After uploading a logic program, the user is taken to the main justification interface, shown in Figure 5.18. It displays the answer sets of the given logic program and allows the user to specify the desired justifications to be constructed:

- **LABAS Justifications versus Attack Trees**: By default, LABAS Justifications are constructed. The user can choose Attack Trees instead by clicking the respective button.

- **Answer Set**: By default, justifications are constructed w.r.t. "Answer Set 1". A different answer set can be chosen in the "Select Answer Set" drop-down menu.

- **Number of justifications**: By default, *all* justifications for a chosen literal w.r.t. the selected answer set are constructed, no matter if the literal is or is not contained in the answer set. In the "Generate all" drop-down menu, the user can choose a maximum of 1, 5, 10, or 20 justifications to be constructed instead.

- **Similarity of justifications**: If various justifications are constructed for a literal, justifications with successive indices are often more similar than justifications with non-successive indices, as will be discussed in Section 5.7.4. The user can thus choose to only view justifications with odd or even indices in the "Generate odd and even" drop-down menu (by default both odd and even justifications are displayed).

- **Justified literal**: The user has to type in the literal to be justified in the "Choose

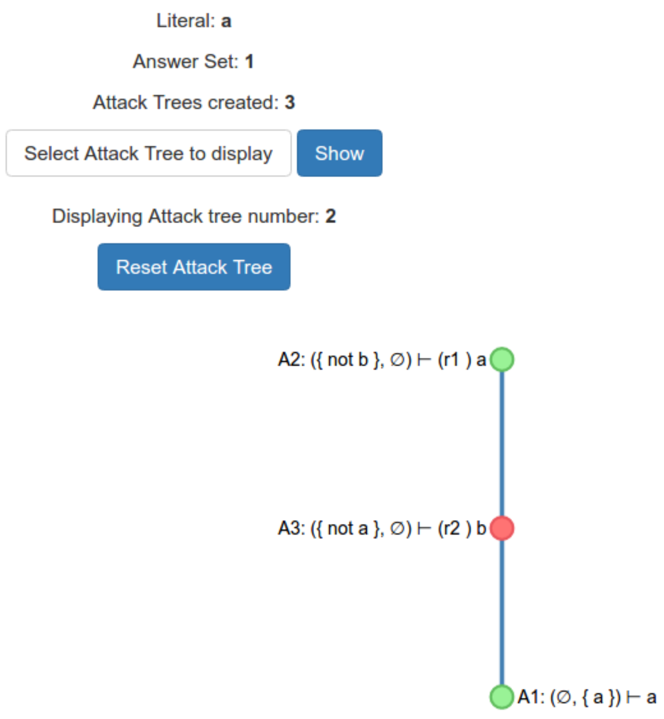

Literal: **a**
Answer Set: **1**
Attack Trees created: **3**
Select Attack Tree to display
Show
Displaying Attack tree number: **2**
Reset Attack Tree
A2: ({ not b }, ∅) ⊢ (r1) a
A3: ({ not a }, ∅) ⊢ (r2) b
A1: (∅, { a }) ⊢ a

Figure 5.19: Attack Tree constructed by the LABAS Justifier.

Literal" text box. The format used is the same as previously described for logic programs.

The only input required from the user is thus the literal to be justified. All other options have a default value and do therefore not require any action from the user.

After clicking the "Build Justification" button, Attack Trees or LABAS Justifications are constructed as specified by the user and the first justification is displayed below the main justification interface. The top part of the justification (see top of Figure 5.19) provides some basic information, in particular, which literal is being justified w.r.t. which answer set, how many justifications were computed, and which justification is currently displayed (by default, the first justification is displayed). A drop-down menu is used to select which of the computed justifications to display. Note that in contrast to our theory of justifications, we construct *all* justifications (unless a different number is chosen) even for a literal contained in the chosen answer set, since the user may be interested to investigate alternative justifications.

Attack Trees

An example Attack Tree created by the LABAS Justifier is displayed in Figure 5.19. The LABAS Justifier uses the colours green and red, respectively, rather than the labels '+' and '−' for nodes in Attack Trees. To provide the user with some additional information about the structure of arguments, the clauses used to construct an argument are given in addition to its premises. For example, "r2" in argument "A3" in Figure 5.19 refers to the second clause in the given logic program shown in Figure 5.18, where facts are not counted when indexing the clauses[3].

To simplify large Attack Trees and focus on important parts, nodes in an Attack Tree can be collapsed by clicking on them. Collapsed nodes can be expanded by clicking on them again. The "Reset Attack Tree" button is used to re-create the original state of the Attack Tree, where no nodes are collapsed.

Infinite Attack Trees are indicated by dots labelling the last node of an Attack Tree, as shown on the right of Figure 5.20. We chose to always indicate infinite Attack Trees after a node labelled '+' (green) has been repeated.

Note that in theory there may be infinitely many Attack Trees for a chosen literal. However, for simplicity the LABAS Justifier does not construct infinitely many Attack Trees if these trees follow the same repetitive pattern, as illustrated by Example 5.14.

Example 5.14. Let $\mathcal{P}_{12}$ be the following logic program, whose only answer set is $S = \{a\}$:

$$\begin{aligned} \{\ a &\leftarrow \texttt{not}\ b; \\ b &\leftarrow \texttt{not}\ a; \\ a &\leftarrow\ \} \end{aligned}$$

The LABAS Justifier constructs three Attack Trees for a w.r.t. S, illustrated in Figure 5.20. However, in theory there are infinitely many Attack Trees, which repeat arguments A2 and A3 different numbers of times before ending with argument A1, for example A2 – A3 – A2 – A3 – A1, A2 – A3 – A2 – A3 – A2 – A3 – A1, A2 – A3 – A2 – A3 – A2 – A3 – A2 – A3 – A1, and so on (where each argument has one child node, namely the succeeding argument, similar to the Attack Trees in Figure 5.20).

LABAS Justifications

Due to the advantages of LABAS Justifications over BABAS Justifications discussed in Section 5.6.2, the LABAS Justifier only constructs LABAS Justifications. Similarly to Attack Trees, the colours red and green are used to respectively indicate the + and − labels of literals and relations in LABAS Justifications.

As given by our theory, the LABAS Justifier constructs LABAS Justifications from Attack Trees. It may thus seem surprising that the LABAS Justifier is able to construct *all* LABAS Justifications, even though not all Attack Trees are created. This is due to the fact that each Attack Tree that is not constructed by the LABAS Justifier repeats parts of itself, parts which are also present in an Attack Tree constructed by the LABAS Justifier not comprising these repetitions. Since LABAS Justifications are constructed by extracting information from arguments in Attack Trees, the LABAS Justifications of a repetitive

[3] "r" stands for "rule" since the clauses of a logic program are referred to as "rules" in the translated ABA framework.

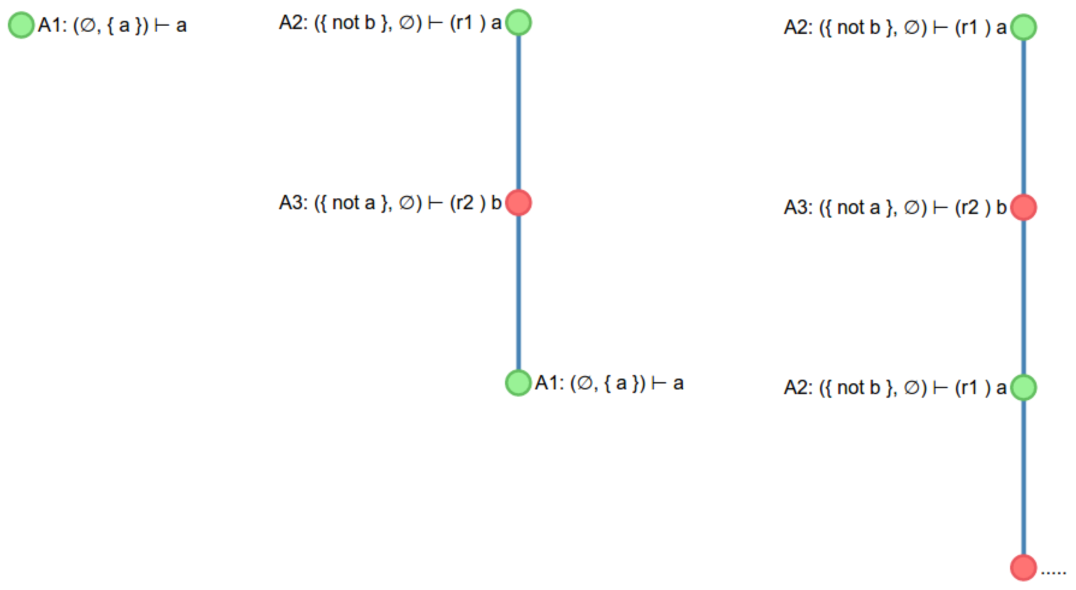

A1: (∅, { a }) ⊢ a
A2: ({ not b }, ∅) ⊢ (r1) a
A3: ({ not a }, ∅) ⊢ (r2) b
A1: (∅, { a }) ⊢ a
A2: ({ not b }, ∅) ⊢ (r1) a
A3: ({ not a }, ∅) ⊢ (r2) b
A2: ({ not b }, ∅) ⊢ (r1) a
.....

Figure 5.20: The three Attack Trees constructed by the LABAS Justifier for literal a in Example 5.14.

Attack Tree not constructed by the LABAS Justifier is the same as the LABAS Justification of some non-repetitive Attack Tree constructed by the LABAS Justifier. For example, the LABAS Justifications shown in Figure 5.21 are extracted from the three Attack Trees in Figure 5.20. These are the only LABAS Justifications of literal a w.r.t. the answer set S (see Example 5.14) since the additional Attack Trees mentioned in Example 5.14 all yield the third LABAS Justification.

The literal nodes of a LABAS Justification can be dragged horizontally, enabling the user to customise the layout of the justification. Vertical dragging is not allowed to ensure that the hierarchical structure of the LABAS Justification is preserved, with the top node being the justified literal. Similarly to Attack Trees, the "Reset LABAS Justification" button is used to recreate the initial layout of the LABAS Justification.

5.7.2 Architecture

The overall architecture of the LABAS Justifier is displayed in Figure 5.22. As standard for web-applications, we distinguish between server and client side.

We use Node.js[4], a JavaScript runtime environment, to build the server, which communicates with the client side, and combine it with Express.js[5], a flexible Node.js web application framework equipped with a robust set of features. The computation of answer sets is done using the clingo answer set solver provided by Potassco[6] (Potsdam Answer Set

[4]http://nodejs.org/
[5]http://expressjs.com/
[6]http://potassco.org/

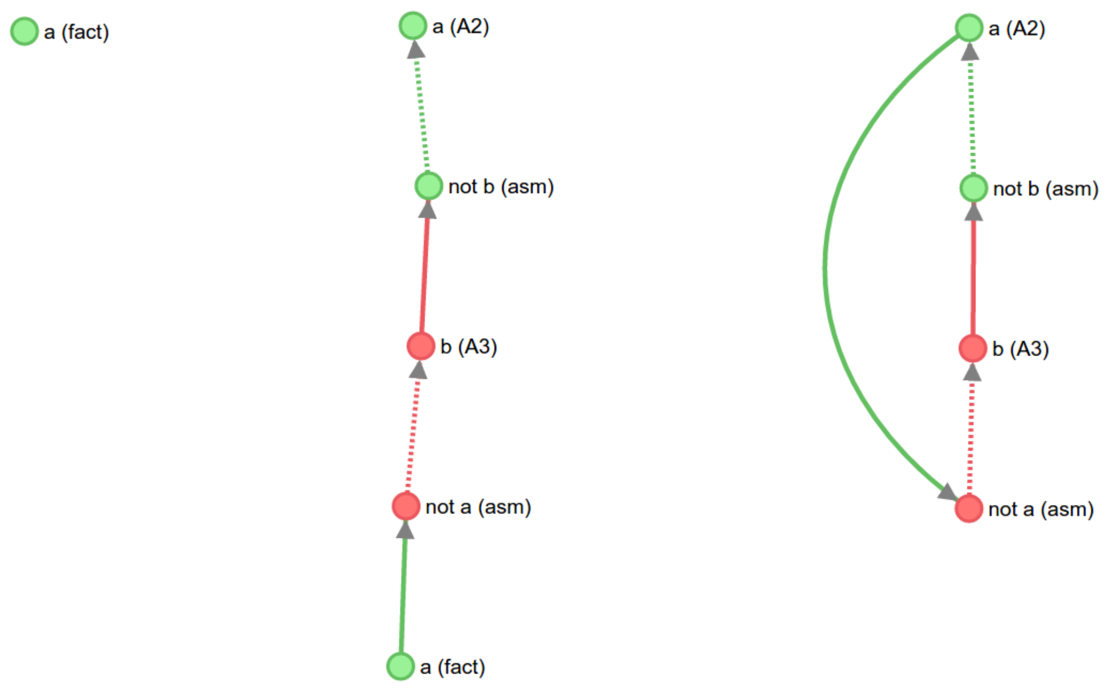


Figure 5.21: The three LABAS Justification constructed by the LABAS Justifier from the three Attack Trees in Figure 5.20.

Solving Collection), whereas the construction of justifications is implemented in Python[7].

In addition to the standard HTML, CSS, and JavaScript[8], the client side also makes use of the D3 JavaScript library[9] for graphically displaying Attack Trees and LABAS Justifications. We furthermore use Bootstrap[10] to improve the design of the LABAS Justifier.

[7]`http://www.python.org/`
[8]`http://www.javascript.com/`
[9]`http://d3js.org/`
[10]`http://getbootstrap.com/`

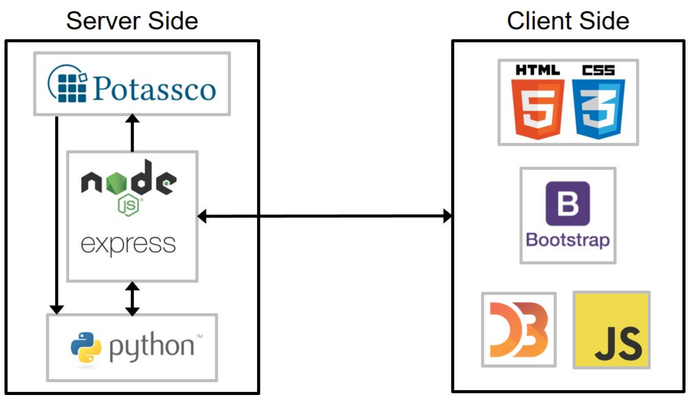


Figure 5.22: Architecture of the LABAS Justifier.

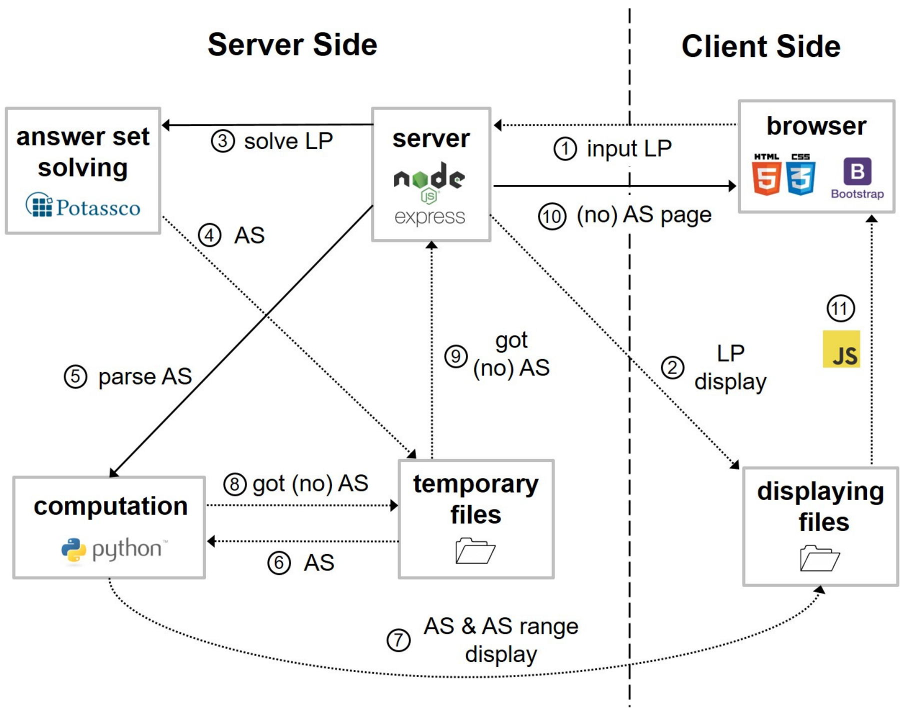


Figure 5.23: Computing, parsing, and displaying answer sets (AS) of a logic program (LP) as performed by the LABAS Justifier, where solid arrows indicate communication/calls between the components and dotted arrows denote information transfer.

5.7.3 Work Flow

The LABAS Justifier performs three tasks: 1) parsing a logic program provided by the user and computing its answer sets, 2) constructing justifications for the literal and answer set specified by the user, and 3) displaying justifications as requested by the user. In the following, we describe the work flow of each task in more detail, with a particular focus on the interaction between the different components of the LABAS Justifier.

When a user enters the LABAS Justifier homepage, the Node.js server creates a cookie[11], which is used throughout all tasks and component communications to ensure that the user is provided with the justification requested, rather than with those requested by another user.

Parsing a Logic Program and Computing Answer Sets

Figure 5.23 illustrates the interaction between the client and the server when the user provides a logic program.

After the user inputs or uploads a logic program on the home page of the LABAS Justifier (see Figure 5.17), the Node.js server receives the plain text or .lp file (1), respectively, and saves the logic program to a new text file (2) to be parsed by JavaScript later in order to display the logic program on the main justification page (see Figure 5.18).

[11]`http://www.npmjs.com/package/cookie-parser`

The server then calls the answer set solver clingo with the given logic program (3), which saves its output to a temporary file (4). Subsequently, the server calls a Python script (5), which takes the temporary clingo output file (6) and extracts the answer sets from irrelevant information such as warnings and solving time. The Python script saves the extracted answer sets in a text file (7), formatted such that they can be easily displayed in the web browser on the main justification page. If no answer sets could be extracted, i.e. if the given logic program has no answer sets, the whole clingo output is saved (7) to be displayed to the user later. The Python script also saves the range (i.e. the total number) of answer sets to a text file (7), which is later used to populate the drop-down menu for selecting which answer set to use for the justification on the main justification page. Finally, the Python script creates a temporary file containing the number 1 or 0 (8), the former indicating that answer sets were computed, the latter that clingo was unable to compute answer sets for the given logic program. This temporary file is then read by the Node.js server (9) to either direct the browser to the main justification page or to an error page (10) displaying the clingo output previously saved. JavaScript is used to display the previously saved logic program and answer sets in the browser, and populate the answer set drop-down menu (11).

Constructing and Displaying Justifications

Figure 5.24 illustrates the interaction between the server and the client of the LABAS Justifier after the user specifies a literal and answer set to be justified.

After the user has input a literal to be justified and provided the justification parameters (the default values if the user does not change the parameters, as explained in Section 5.7.1), the Node.js server receives the literal as well as the parameters (1) and saves the literal and the index of the answer set to be justified in files (2), which will be parsed by JavaScript later in order to display justification information in the browser. The Node.js server then calls the main Python justification script (3), which reads the files containing the answer sets and logic program (4) that were created during the parsing and answer set computation stage, and computes Attack Trees. Depending on the justification parameters chosen by the user, the Python script may subsequently create LABAS Justifications. It then saves the respective justifications in a JSON file (5). In the next step the Node.js server directs the browser to the LABAS Justifications or the Attack Trees page (6), based on the user's choice of justification parameters. The respective browser page loads the graphical representations of the (first) justification using the D3.js library (7), which reads the necessary information from the previously created JSON files (8). The graphs as well as the previously saved literal and chosen answer set are then displayed in the browser using JavaScript (9).

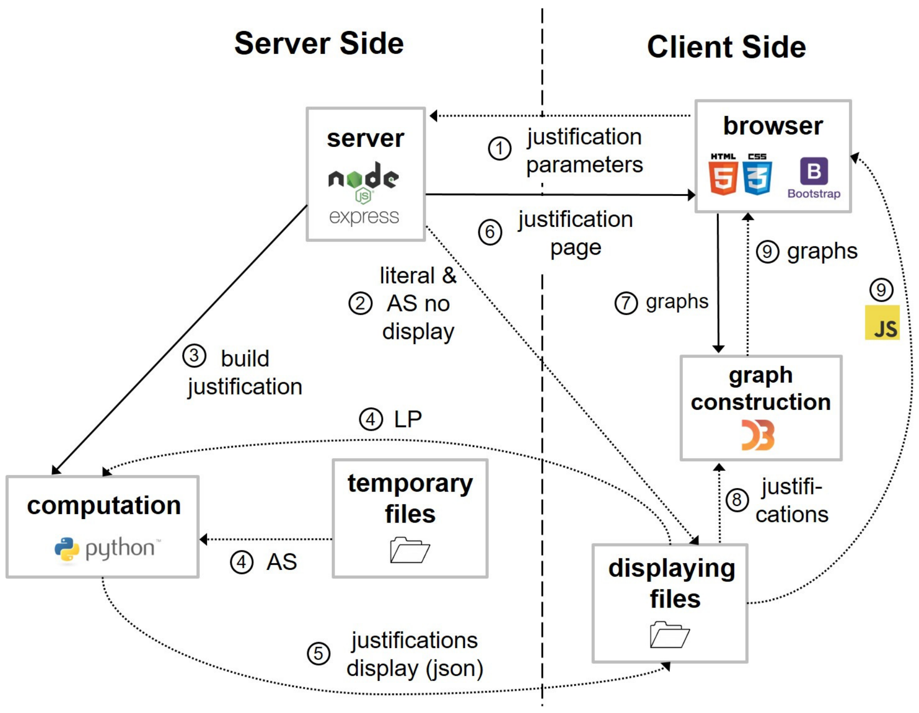


Figure 5.24: Computing and displaying justifications as performed by the LABAS Justifier, where solid arrows indicate communication/calls between the components and dotted arrows denote information transfer.

Displaying a Particular Justification

The third task of the LABAS Justifier, i.e. displaying justifications other than the first one, chosen by the user from the drop-down menu, involves only the client side, in particular the last steps from the "constructing and displaying justifications" task are repeated: The browser displays the graphical representation of the chosen justification using the D3.js library, which reads the respective JSON file (that has the respective index chosen by the user) and constructs the Attack Tree graph. JavaScript then displays the graphs in the browser along with the respective information, such as the updated index of the displayed justification.

5.7.4 Construction of Attack Trees and LABAS Justifications

We now describe the algorithm for constructing Attack Trees and LABAS Justifications in more detail and point out various design choices as well as deviations from the theory.

The Attack Tree Algorithm and Argument Construction

Algorithm 1 outlines the method for constructing the set *attackTrees* consisting of all Attack Trees for a given literal k w.r.t. answer set S.

According to the theory (see Section 5.4), Attack Trees for k w.r.t. S are constructed by computing the translated AA framework of the given logic program and identifying the corresponding stable argument extension of S. Arguments labelled '+' and '−' are then determined based on their (non-) membership in the stable argument extension. Computing all arguments and attacks in the translated AA framework and subsequently determining stable argument extensions is computationally expensive. We thus neither construct all arguments nor compute stable argument extensions. Instead, we make use of the correspondence between arguments in stable argument extensions and literals in answer sets (see Section 4.4.4).

The *constructAttackTrees* algorithm (see Algorithm 1) first determines whether the Attack Trees to be constructed are positive or negative, i.e. whether the argument held by the root node should be labelled '+' or '−'. Diverting from the theory, this is done by checking if k is contained in the answer set S, since we know from Corollaries 4.20 and 4.21 that the conclusion of an argument that is in the corresponding stable argument extension of S is contained in S. In the next step, all arguments for k with the correct label are constructed to serve as root nodes of Attack Trees.

The *argConstruction* method constructs all arguments with conclusion k by applying the clauses in the logic program until a finite argument tree is obtained. In addition, the desired label of the argument is taken into account to ensure that no arguments with the wrong label are constructed. In particular, if the label of an argument for k should be '+', i.e. the argument is contained in the corresponding stable argument extension, there may exist other arguments with conclusion k that are not contained in the stable argument extension, so the *argConstruction* method needs to ensure that only arguments contained in the stable argument extension are constructed. This is achieved by using the following result.

Lemma 5.15. *Let S be an answer set of $\mathcal{P}$, $\mathcal{E}$ the corresponding stable argument extension of S in $AA_{\mathcal{P}}$, and $k \in S_{NAF}$. If an argument for k is in $\mathcal{E}$, then every child node of the root holding k holds a literal k_N such that $k_N \in S_{NAF}$.*

Proof. Let A be an argument for k contained in $\mathcal{E}$ and let $k_1, \dots k_n$ $(n \geq 0)$ be all literals held by child nodes of the root node holding k in A. Then every subtree of A with root node $k_1, \dots k_n$ is an argument $A_1, \dots A_n$ for $k_1, \dots k_n$, respectively. Assume that some $k_i \notin S_{NAF}$. Then by the second item in Proposition 5.1, there exists no argument with conclusion k_i in $\mathcal{E}$, so argument $A_i \notin \mathcal{E}$. Thus, by Corollary 4.21 there exists `not` l in the assumption premises of A_i such that $l \in S$, so `not` $l \notin \Delta_S$. Since A_i is a sub-tree of argument A, `not` l is in the assumption premises of A, so by Corollary 4.21, $A \notin \mathcal{E}$. Contradiction, so all $k_i \in S_{NAF}$. □

Thus, when constructing an argument tree for k which should be contained in the corresponding stable argument extension of S, the *argConstruction* method checks after each application of a clause from the logic program if all body literals are contained in S_{NAF}. If not, the construction of the respective argument is stopped and an argument not applying the problematic clause is constructed, again checking the body literals of all clauses applied.

Algorithm 1 *constructAttackTrees*($k, S, justificationMode, no\_of\_justifications$): construct Attack Trees for literal k w.r.t. answer set S, with justification parameters *justificationMode* (*even* or *odd*) and *no_of_justifications*

```
 1: $attackTrees = [\,]$
 2: if $k \in S$ then
 3:     $label = +$
 4: else
 5:     $label = -$
 6: end if
 7: $arguments = argConstruction(k, label)$
 8: for $arg \in arguments$ do
 9:     $root = AttackTree(arg, label)$
10:     if $label == +$ then
11:         $trees = extendTreeNode(root, [arg])$
12:     else
13:         $trees = extendTreeNode(root, [\,])$
14:     end if
15:     $attackTrees = attackTrees + trees$
16:     if $justificationMode == even$ then
17:         delete odd indices from $attackTrees$
18:     end if
19:     if $justificationMode == odd$ then
20:         delete even indices from $attackTrees$
21:     end if
22:     if $length(attackTrees) \geq no\_of\_justifications$ then
23:         break
24:     end if
25: end for
26: if $length(attackTrees) > no\_of\_justifications$ then
27:     $attackTrees = attackTrees[0 : no\_of\_justifications]$
28: end if
29: return $attackTrees$
```

If the label of arguments for k should be '$-$', no checks are required since by Proposition 5.1 no argument for k is contained in the corresponding stable argument extension of k. The *argConstruction* method thus constructs all arguments for k.

Deviating from the theory, the *argConstruction* method does not construct "looping" arguments, i.e. arguments where the same rule is applied more than once in a branch of the argument tree. This prevents the construction of infinitely many arguments that repeat parts of themselves, thus providing no additional information compared to the same argument without the repetition. However, various arguments with the same sets of assumptions, facts, and applied rules may be constructed if their tree structure is different. Note that, except for their names, these arguments look exactly the same in Attack Trees constructed by the LABAS Justifier since arguments are displayed in terms of their abbreviation rather than as argument trees.

Example 5.15. Let $\mathcal{P}_{13}$ be the following logic program, whose only answer set is $S =$

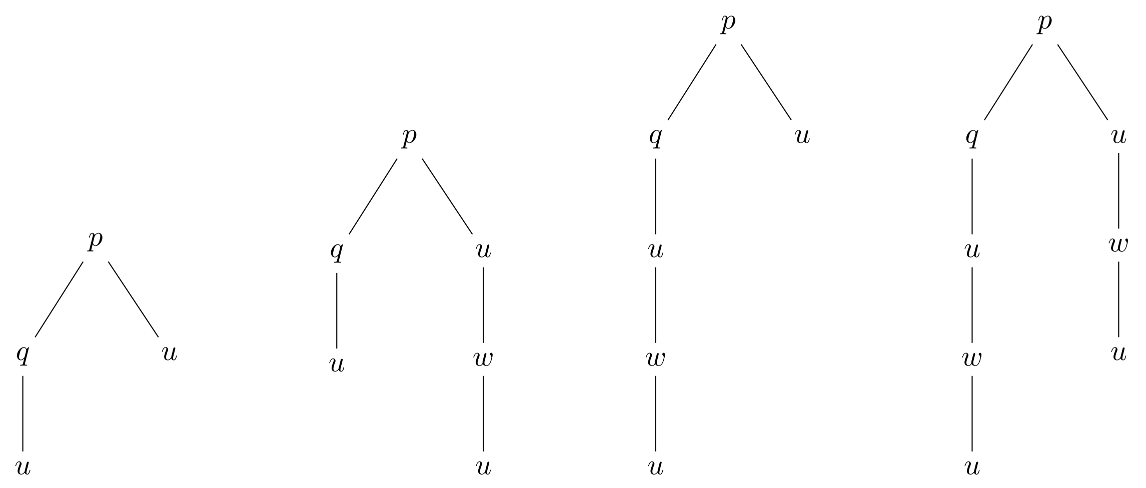


Figure 5.25: The four arguments for p constructed by the *argConstruction* method (see Example 5.15).

$\{p, q, u\}$:

$$\{\ p \leftarrow q, u;$$
$$q \leftarrow u;$$
$$u \leftarrow w;$$
$$w \leftarrow u;$$
$$u \leftarrow \}$$

The LABAS Justifier constructs four arguments for p w.r.t. S, illustrated in Figure 5.25. However, in theory there are infinitely many arguments since the third and fourth clause can be applied repeatedly to extend the leaf nodes, before ending both branches of the argument tree with fact u. Note that except the first argument (leftmost in Figure 5.25), all arguments have the same abbreviation in the LABAS Justifier when disregarding their names, namely $(\emptyset, \{u\}) \vdash (r_1, r_2, r_3, r_4)\ p$. Thus, in an Attack Tree these arguments will only be distinguishable by their names.

After constructing all arguments for k that are (not) contained in the corresponding stable argument extension of S, an Attack Tree node is created for each of the arguments (see line 8 in Algorithm 1), which is then extended into full Attack Trees in all possible ways using the *extendTreeNode* method outlined in Algorithm 2. The second argument taken by the *extendTreeNode* method is the set of all arguments held by nodes labelled '+' in the Attack Tree so far. This is used to ensure that if an argument is repeated, the Attack Tree is not further extended but marked as being infinite. Thus, if the root node of an Attack Tree holds an argument for k that is labelled '+', this argument is passed as the set of already used arguments, whereas if the root node holds an argument labelled '−', an empty set of already used arguments is passed.

After extending the root node holding an argument for k into all possible full Attack

Trees, these *trees* are added to the final set of all *attackTrees* for *k* (see line 15 in Algorithm 1). The *constructAttackTrees* algorithm then checks the parameters specified for the justifications. If the user indicated that only justifications with odd (even) indices are to be constructed, all Attack Trees with even (respectively odd) indices constructed so far are deleted. Furthermore, the algorithm checks whether the user restricted the total number of justifications to be constructed and, if so, whether the maximal number of Attack Trees has already been reached, in which case we stop the construction of additional trees.

In the last step, the algorithm ensures that at most the number of Attack Trees specified by the user is returned, by pruning the set of *attackTrees* to the maximum number if required.

Extending a Root Node into Full Attack Trees

Algorithm 2 outlines the recursive *extendTreeNode* method for extending a given root node of an Attack Tree into all possible full Attack Trees. The idea is to find all attackers of the argument held by the given *root* node and to then appropriately extend the root node with child nodes holding the attackers, where the way of extending the Attack Tree depends on the label of the root argument. If the root argument is labelled '+', all attacking arguments are added as child nodes of the root node. In contrast, if the root argument is labelled '$-$', it should have exactly one child node holding an attacking argument (see Definition 5.4). Thus, for each argument attacking the root argument, a new Attack Tree is created whose root node holds the given root argument and has only one child node, holding the respective attacking argument. In both cases, each branch of every extended Attack Tree is then further extended in the same way.

The *extendTreeNode* method starts by identifying the desired label of the root node's children and initialising the set of all *extendedTrees* to be returned (see lines 1-7 in Algorithm 2). In case the root node is labelled '+', the set is initialised to consist of the root node rather than being the empty set as one may expect. The reason for this will become apparent when we explain how the algorithm extends a root node labelled '+' with child nodes for all attacking arguments.

In the next step (see Lines 8-17), the *extendTreeNode* method finds all arguments attacking the given root argument. Since we do not construct all arguments and attacks in the translated AA framework upfront, the *extendTreeNode* method has to construct these attackers. This is done by determining the conclusions of all potential attackers, i.e. the corresponding literals of all NAF literals occurring as assumption premises of the root argument, and then finding all arguments with this conclusion and the correct label. Since the *argConstruction* method saves arguments that have been constructed for a literal, the *extendTreeNode* method first checks for each identified literal if arguments for this literal have previously been constructed, and if so adds them to the set *attackingArgs* of all arguments attacking the root argument. If no arguments for a literal have been

Algorithm 2 $extendTreeNode(root, usedArguments)$: extend a single $root$ node into full Attack Trees without using arguments from the set of $usedArguments$

```
1: if $getLabel(root) == +$ then
2:    $attackerLabel = -$
3:    $extendedTrees = [root]$
4: else
5:    $attackerLabel = +$
6:    $extendedTrees = [\,]$
7: end if
8: $attackingArgs = [\,]$
9: for $assumption \in getAssumptions(root)$ do
10:    $literal = \sim assumption$
11:    if arguments for $literal$ have already been constructed then
12:       retrieve those $arguments$
13:    else
14:       $arguments = argConstruction(literal, attackerLabel)$
15:    end if
16:    $attackingArgs = attackingArgs + arguments$
17: end for
18: for $attacker \in attackingArgs$ do
19:    $attackerNode = AttackTree(attacker, attackerLabel)$
20:    if $attacker \in usedArguments$ then
21:       $treesOfOneAttacker = [attackerNode]$
22:    else
23:       if $attackerLabel == +$ then
24:          $usedArgumentsNew = usedArguments + [attacker]$
25:       else
26:          $usedArgumentsNew = usedArguments$
27:       end if
28:       $treesOfOneAttacker = extendTreeNode(attackerNode, usedArgumentsNew)$
29:    end if
30:    if $getLabel(root) == +$ then
31:       $tmpTrees = [\,]$
32:       for $attackerTree \in treesOfOneAttacker$ do
33:          for $extendedTree \in extendedTrees$ do
34:             $tree = addAsChild(attackerTree, extendedTree)$
35:             $tmpTrees = tmpTrees + [tree]$
36:          end for
37:       end for
38:       $extendedTrees = tmpTrees$
39:    else
40:       for $attackerTree \in treesOfOneAttacker$ do
41:          $tree = addAsChild(attackerTree, root)$
42:          $extendedTrees = extendedTrees + [tree]$
43:       end for
44:    end if
45: end for
46: return $extendedTrees$
```

constructed so far, arguments with the correct label are constructed using the previously described *argConstruction* method and added to the *attackingArgs* set.

Having found all attacking arguments (see line 16 in Algorithm 2), the main part of the Attack Tree extension starts, which handles every *attacker* separately by creating the set *treesOfOneAttacker*, consisting of of all possible Attack Trees holding the *attacker* in the root node, and then appropriately appending these Attack Trees to the *root* node. For this purpose, a new Attack Tree node holding the *attacker* is created, which we call *attackerNode*. Then, the *extendTreeNode* method checks if the *attacker* occurs in the set of *usedArguments*, i.e. if the argument already occurs "higher up" in the Attack Tree. If so, the newly created *attackerNode* will be the last node displayed in an infinite Attack Tree, so it does not have to be further extended. Therefore, the set of all possible Attack Trees holding the *attacker* in the root node contains only one Attack Tree made of one node, namely the the newly created *attackerNode* (see Line 21 in Algorithm 2). If the *attacker* is not part of the *usedArguments*, the newly created *attackerNode* is further extended into all possible Attack Trees having this node as the root, by recursively applying the *extendTreeNode* method. When calling the *extendTreeNode* method for the *attackerNode*, the set of *usedArguments* is passed, potentially adding the *attacker* in case it is labelled '+', to ensure that if the *attacker* re-occurs in an extended Attack Tree, the Attack Tree is marked as "infinite" (see Line 24 in Algorithm 2).

When adding the *attackerNode* as a child of the *root* node to construct a full Attack Tree, the *treesOfOneAttacker* set, containing all possible ways of extending the *attackerNode*, provides alternatives for extending the *attackerNode* branch of the Attack Tree. If the *root* argument is labelled '+', it has a child node for each attacking argument (see Line 30 in Algorithm 2). These children can then be extended with all possible combinations of extensions from the sets *treesOfOneAttacker* of each attacker. Thus, for the first *attacker*, each Attack Tree from its *treesOfOneAttacker* set is added to a copy of the *root* node (the only node contained in the set *extendedTrees*). These Attack Trees are then saved as the new set of *extendedTrees*. For the second *attacker*, each Attack Tree from its *treesOfOneAttacker* set is then added to a copy of each of the previously created trees (stored in the set of *extendedTrees*), thus creating Attack Trees with every combination of extending the *root* node when it has only the first and the second attacker as child nodes. This procedure is repeated for all attackers, creating Attack Trees for all combinations of extending each child node of the *root* holding an attacking argument. In contrast, if the root node is labelled '−', the extension of Attack Trees is much simpler since each Attack Tree has only one child node holding one attacker (see line 39 in Algorithm 2). Thus, for each *attacker* and each way of extending the *attackerNode* as given by the Attack Trees in the *treesOfOneAttacker* set, a new Attack Tree with the *root* node is created, which is extended with one of the Attack Trees from the *treesOfOneAttacker*. Each of these new Attack Trees is a full Attack Tree, thus added to the set of final *extendedTrees*, which is returned after processing every attacker.

5.8 Related Work

According to Pontelli et al. [PSEK09], a justification should "provide only the information that are relevant to the item being explained", making it easier to understand. We incorporate this in ABAS Justifications[12] by not using the whole derivation of a literal, but only the underlying facts and NAF literals necessary to derive the literal in question.

The two approaches for justifying why a literal is or is not part of an answer set that are most related to ABAS Justifications are Argumentation-Based Answer Set Justifications and off-line justifications. *Argumentation-Based Answer Set Justifications* [SST13] are a "predecessor" of ABAS Justifications, which uses the ASPIC+ argumentation framework [Pra10] instead of ABA. In contrast, *off-line justifications* [PSEK09] explain why a literal is or is not part of an answer set by making use of the well-founded model semantics for logic programs.

5.8.1 Off-line Justifications

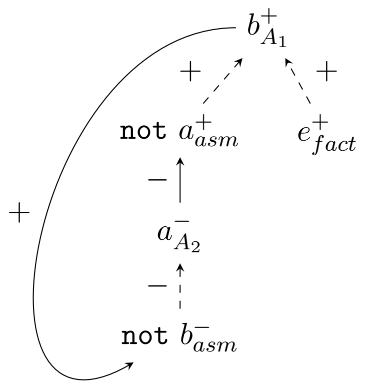
The off-line justification for a classical literal l is a graph of classical literals with root node l. The child nodes of l are the literals on which l is positively or negatively dependent. In other words, the justified literal l has the body literals of an applicable clause in the logic program as its child nodes, and the justifications of these body literals as subgraphs.

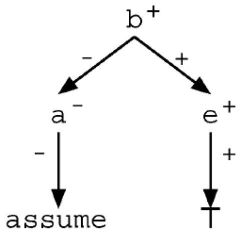
Example 5.16. Consider the following logic program $\mathcal{P}_{14}$ (taken from [PSEK09]), which has two answer sets $S_1 = \{b, e, f\}$ and $S_2 = \{a, e, f\}$:

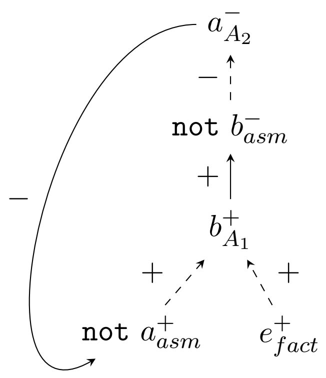
$$\begin{aligned}\{\ & a \leftarrow f, \texttt{not}\ b;\\ & b \leftarrow e, \texttt{not}\ a;\\ & f \leftarrow e;\\ & d \leftarrow c, e;\\ & c \leftarrow d, f;\\ & e \leftarrow\ \}\end{aligned}$$

The off-line justification for $b \in S_1$ is depicted on the top right of Figure 5.26. It is constructed using the second clause in $\mathcal{P}_{14}$, yielding a positive dependency of b on e, and a negative dependency of b on a. This expresses that b is in the answer set because it depends on e being part of the answer set and on a not being part of it. Whether or not a classical literal l occurring in the off-line justification is part of the answer set in question is indicated by the labels '$+$' (if l is in the answer set) or '$-$' (if l is not in the answer set). The dependency conditions of b on e and a are satisfied, since e is labelled '$+$' and a is labelled '$-$'. The off-line justification graph also expresses that e is known to be true since it is a fact (indicated by $\top$ in the graph) and that a is assumed to be false (indicated by *assume* in the graph). It is important to note that NAF literals are represented indirectly

[12]We will use the term *ABAS Justification* as shorthand for both BABAS and LABAS Justifications.







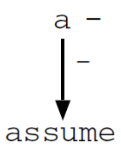


Figure 5.26: The two graphs at the top illustrate the LABAS Justification (left) and the Off-line Justification (right) of $b \in S_1$ of $\mathcal{P}_{14}$, whereas the graphs at the bottom represent the justifications of $a \notin S_1$ of $\mathcal{P}_{14}$.

in an off-line justification by means of their corresponding classical literal. For example in the off-line justification of b (top right of Figure 5.26), the classical literal a is used to represent the dependency of b on the NAF literal `not` a.

Off-line justifications treat the relationship between literals in a proof-oriented way, that is as top-down dependencies, whereas ABAS Justifications (and Attack Trees) provide explanations in a bottom-up manner in terms of assumptions and underlying knowledge supporting the conclusion. We argue that our bottom-up approach might be clearer for non-experts, as human decision making seems to involve starting from what is known along with some kind of assumptions, and then drawing conclusions from that. Instead of saying that b is dependent on e in $\mathcal{P}_{14}$ as done by an off-line justification, a LABAS Justification expresses that e supports b, as shown on the top left of Figure 5.26. Especially with respect to NAF literals, we believe that a bottom-up support relation is more intuitive than a top-down dependency relation: instead of saying that b negatively depends on a not being in the answer set as done by an off-line justification, the LABAS Justification states that `not` a supports b (compare the two graphs at the top of Figure 5.26).

The well-founded model semantics is used in the construction of off-line justifications to determine literals that are "assumed" to be false with respect to an answer set, as opposed to literals that are always false. These assumed literals are not further justified, i.e. they are leaf nodes in an off-line justification graph. In contrast, LABAS Justifications further justify these "assumed" literals. They are usually true NAF literals that are part of a dependency cycle. An example is the literal a in the logic program $\mathcal{P}_{14}$, which is assumed to be false in the off-line justification of b w.r.t. S_1 (bottom right of Figure 5.26). In contrast, the LABAS Justification further explains that a is not in the answer set because the support by `not` b does not "succeed" since the attack by b on `not` b "succeeds" (bottom left of Figure 5.26).

An off-line justification graph includes all intermediate literals in the derivation of the literal in question. However, following Brain and De Vos [BD08] we argue that it is sufficient for a justification to include the most basic relevant literals, without considering intermediate steps. Especially in the case of large logic programs, where derivations include many steps, an off-line justification will be a large graph with many positive and negative dependency relations, which is hard to understand for humans. In contrast, an ABAS Justification only contains the basic underlying literals, i.e. facts and NAF literals necessary to derive the literal in question, making the justification clearer. However, if the intermediate steps were required, they could be easily extracted from the arguments in the Attack Trees underlying an ABAS Justification.

In contrast to off-line justifications, where in addition to answer sets the well-founded model has to be computed, for the construction of ABAS Justifications the computation of answer sets is sufficient. Even though the definitions of ABAS Justifications refer to the corresponding stable argument extensions of the translated AA framework, it is not necessary to compute these stable argument extensions, as explained in Section 5.7.

5.8.2 Argumentation-Based Answer Set Justifications

Argumentation-Based Answer Set Justifications [SST13] constitute the first approach that applies argumentation theory to answer set programming in order to justify answer sets. There, the ASPIC+ argumentation framework [Pra10] is used instead of ABA.

Similarly to ABAS Justifications, in Argumentation-Based Answer Set Justifications literals are justified with respect to an answer set by means of ASPIC+ arguments w.r.t. the stable argument extension corresponding to the answer set in question. For the translation of a logic program into an ASPIC+ framework only a fraction of ASPIC+ features are needed; defeasible rules, issues, and preference orders are redundant. This is to say that the ASPIC+ framework is too complex for the purpose of a justification and a more lightweight framework like ABA is more suitable.

The method for constructing a justification in Argumentation-Based Answer Set Justification is slightly different from the ABAS Justification approach. Instead of extracting support- and attack-pairs from Attack Trees, Argumentation-Based Answer Set Justifica-

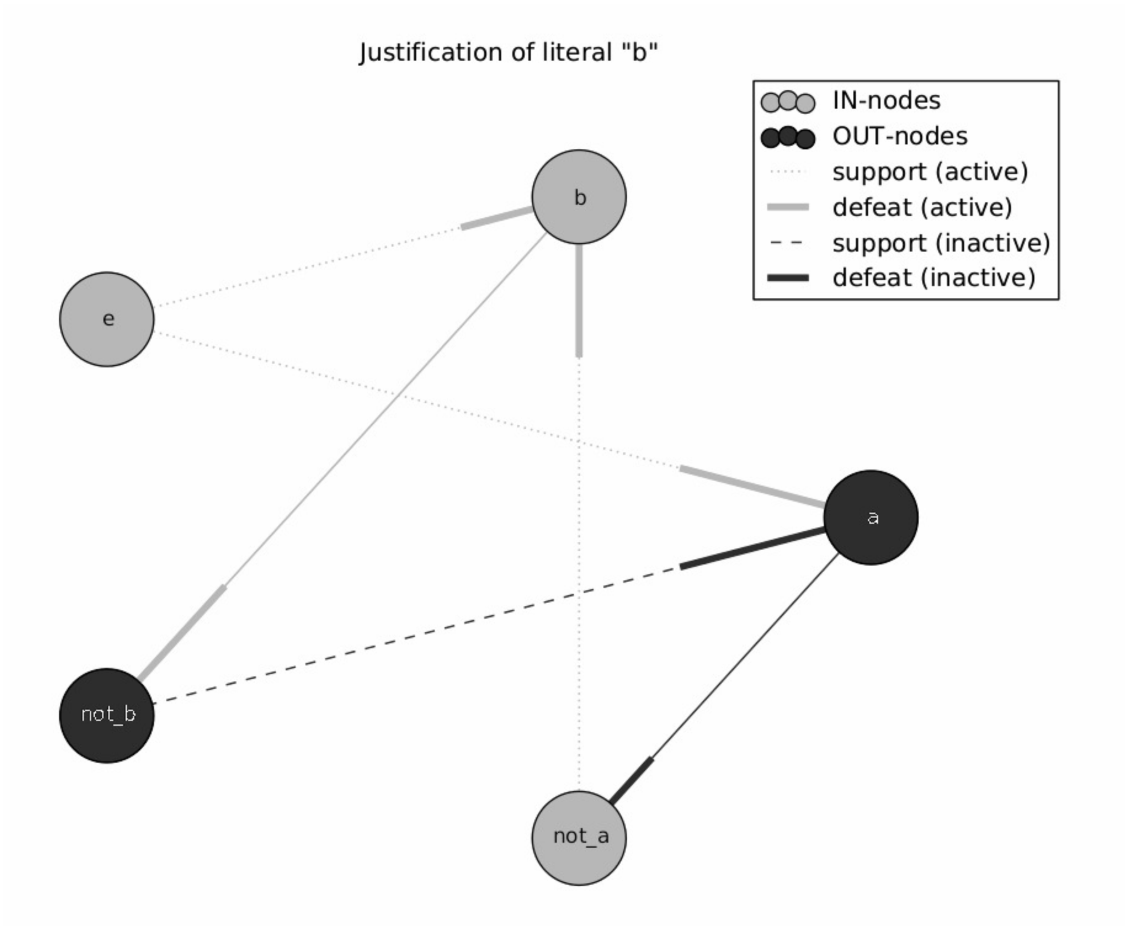


Figure 5.27: Argumentation-Based Answer Set Justification of $b \in S_1$ of $\mathcal{P}_{14}$ from Example 5.16.

tions are defined recursively: For an assumption-argument its attackers are investigated, whereas for non-assumption- and non-fact-arguments supports by assumption- and fact-arguments are examined. The recursion terminates when fact-arguments or non-attacked assumption-arguments are encountered.

Argumentation-Based Answer Set Justifications have the same deficiencies as BABAS Justifications; it is not clear which literals are facts or assumptions, and whether or not support and attack relations "succeed". The implementation of Argumentation-Based Answer Set Justification colours the relations and literals similarly to the labels '+' and '−' on relations and literals in LABAS Justifications, where green corresponds to '+' and red to '−'. However, facts and assumptions cannot be distinguished from other literals, as depicted in Figure 5.27.

In summary, ABAS Justifications are an improvement of Argumentation-Based Answer Set Justifications, both with respect to the elegance of the justification definition and the appropriateness of the argumentation framework used. LABAS Justifications also solve the deficiencies of Argumentation-Based Answer Set Justifications by providing more information about the literals in the explanation as well as about their relationship. Furthermore, Argumentation-Based Answer Set Justifications were introduced without any characterisation. In contrast, here we prove that ABAS Justifications provide an explanation in terms of an admissible fragment of the answer set in question, and show their relationship with abstract dispute trees.

5.8.3 Other Related Explanation Approaches

In addition to the two explanations approaches for answer sets discussed in the previous sections, Erdem and Öztok [EÖ15] introduce a formalism for explaining biomedical queries expressed in ASP. Similarly to ABAS Justifications, they construct trees for the explanation, but in contrast to our justifications these trees carry rules in the nodes rather than literals. Another difference is that their explanation trees comprise every step in the derivation of a literal (similar to off-line justifications explained in Section 5.8.1) rather than abstracting away from intermediate derivation steps between the literal in question and the underlying facts and NAF literals.

Brain and De Vos [BD05] try to answer a similar question as the one we address with ABAS Justifications, i.e. why a set of literals is or is not a subset of an answer. Their explanations are presented in text form, but they point out that it might be possible to use a tree representation instead. Just like [EÖ15], all intermediate steps in a derivation are considered in the explanation, thus differing from ABAS Justifications.

Further justification approaches for logic programs include the causal justifications of Cabalar and Fandinno [CFF14, CF17], the why-provenance of Damásio et al. [DAA13], the justifications of Denecker et al. [DBS15], and the rule-based justifications of Béatrix et al. [BLGS16].

Related to the explanation of ASP is the visualisation of the structure of logic programs in general. ASPIDE [FRR11] is an Integrated Development Environment for ASP, which, among other features, displays the dependency graph of a logic program, i.e. it visualises all positive and negative dependencies between literals. It is thus similar to the previously mentioned approaches in that it illustrates every step in a derivation.

The problem of constructing explanations has been addressed for logic programs without NAF by Arora et al. [ARR^{+}93] and Ferrand et al. [FLT06]. In the early work by Arora et al. [ARR^{+}93] explanations of atoms in a logic program are constructed as simple derivations of these atoms. Thus, this approach is closer to [EÖ15] and [BD05] than to ABAS Justifications, as it provides all intermediate derivation steps. Similarly, Ferrrand et al. [FLT06] show how to use proof trees as explanations for least fixpoint operators, such as the semantics of constraint logic programs, where proof trees are derivations.

The comparison with these existing approaches demonstrates the novelty of ABAS Justifications as they only provide the facts and NAF literals necessary for the derivation of a literal in question rather than the whole derivation with all its intermediate steps.

Explanations have also received attention in other areas in the field of knowledge representation and reasoning, and it has been emphasised that any expert system should provide explanations for its solutions (see [LD04] for an overview of explanations in heuristic expert systems). Furthermore, it has been pointed out that even though argumentation and other knowledge-based systems have been studied mostly separately in the past, argumentation could serve as a useful tool for the explanation of other knowledge-based systems [MIBD02]. In fact, Bench-Capon et al. [BCLM91] provide an early account of explanations for logic programs in terms of arguments, where Toulmin's argument scheme is applied. However, a meta-program encoding the argument scheme has to be created by hand for any logic program that needs explanation, making it infeasible for automatic computation.

Related to argumentation as an explanation method, García et al. [GCRS13] introduce explanations in argumentative terms for argumentation-based reasoning methods, such as Defeasible Logic Programming [GS04], explaining why an argument with a certain conclusion is or is not deemed to be "winning". Similar to ABAS Justifications and Attack Trees, the motivation behind their approach is to provide explanations in terms of attacking and defending relations between arguments. Explanations are given in terms of argument trees similar to Attack Trees, where arguments held by child nodes in the tree attack the argument held by the parent node. In contrast to Attack Trees, however, every node in the tree is extended with all its attackers and the tree is labelled with respect to the grounded argument extension, instead of stable argument extensions. Another difference to our justifications is that García et al. explain why a literal l is not a winning conclusion in terms of an explanation why the contrary literal $\neg l$ is a winning conclusion. In contrast, ABAS Justifications explain why a literal l is not a winning conclusion by pointing out why it cannot possibly be winning. Arioua et al. [ATC15, ACP$^+$16] also use the dialectical structure of argumentation frameworks for explanation. Their application area is ontologies. More recently, argumentation has been used for explanations in Bayesian Networks [VPRV16, TMP$^+$17]

5.9 Summary

In this chapter, we presented two approaches for justifying why a literal is or is not contained in an answer set of a consistent logic program by translating the logic program into an AA framework and using the structure of arguments and attacks for the explanation. Attack Trees, our first justification approach, provide an explanation for a literal in argumentation-theoretic terms, i.e. in terms of arguments and attacks between them. ABA-Based Answer Set Justifications, our second justification approach, flatten the structure of Attack Trees, yielding a set of literal-pairs in a support or attack relation. This justification approach is more aligned with logic programming concepts as it uses literals rather than arguments as an explanation. Both justification approaches are based on the correspondence between answer sets of a logic program and stable argument extensions of the translated AA framework presented in Chapter 4.

Importantly, both Attack Trees and ABAS Justifications explain why a literal is or is not in an answer set in terms of an admissible fragment of this answer set. The justification that a literal is in an answer set is that a derivation of this literal is supported by an admissible fragment of this answer set. In contrast, the justification that a literal is not contained in an answer set is that all derivations of this literal are "attacked" by an admissible fragment of this answer set. In comparison to existing explanation methods for answer sets, ABAS Justifications take an argumentative premise-conclusion approach, i.e. a literal is explained in terms of the facts and NAF literals necessary for its derivations, rather than in terms of the whole derivation.

In this chapter, we only dealt with explanations of *consistent* logic programs, i.e. logic programs with meaningful answer sets. In Chapter 7, we investigate *inconsistent* logic programs that have no answer set or whose only answer set is the set of all literals, and introduce explanations of the inconsistency using concepts similar to Attack Trees.

Chapter 6

On the Non-Existence and Restoration of Stable Labellings in AA

6.1 Introduction

In the previous chapter, we investigated answer sets of consistent logic programs, which correspond to stable argument extensions, or equivalently stable argument labellings, of the translated AA framework. Stable argument labellings (and equivalently stable argument extensions) are not guaranteed to exist for an AA framework, a problem which has mostly been addressed through the usage of semantics that are "as decisive as possible" and guaranteed to exist, such as as the preferred and semi-stable semantics. In contrast, in this chapter we aim to characterise *reasons* for the non-existence of stable argument labellings.

Dung [Dun95b] gives a characterisation of AA frameworks without stable argument extensions, proving that an AA framework that comprises no odd-length cycle of attacking arguments has at least one stable argument extension (and thus at least one stable argument labelling). Consequently, any AA framework that has no stable argument labellings must comprise an odd-length cycle of attacking arguments. However, an AA framework may comprise many odd-length cycles and, as we will show in this chapter, it may be that not all of them should be deemed *responsible* for the non-existence of stable argument labellings.

We investigate the non-existence of stable argument labellings by characterising parts of an AA framework that are responsible for a *preferred argument labelling* not being a stable argument labelling. We propose two different approaches: a labelling-based approach and a structural approach. In the labelling-based approach, we give two characterisations of responsible parts in terms sets of arguments that are labelled `undec` by a preferred argument labelling and that are *illegally* labelled if their labels are changed to `in` or `out`. In contrast, in the structural approach we characterise responsible parts as initial strongly connected components (SCCs) of the AA framework restricted to arguments labelled `undec` by a preferred argument labelling. We call such parts *strongly connected* `undec` *parts* (SCUPs) and prove that they always comprise an odd-length cycle of attacking arguments.

In addition to proposing characterisations of responsible parts of an AA framework, we take our investigations of the non-existence problem of stable argument labellings further by showing how to turn a preferred argument labelling into a stable argument labelling. We propose to re-label certain arguments labelled `undec` by a preferred argument labelling as `in` or `out` and enforcing[1] these new labels to be legal. Since our labelling-based approach characterises responsible arguments as illegally labelled `in` or `out`, we propose to enforce these illegal labels of responsible arguments, i.e. to structurally revise the AA framework such that the (illegal) labels of responsible arguments become legal. We show that this method results in a stable argument labelling. Note that we are here not interested in the exact structural change of an AA framework as long as it ensures that arguments are

[1]Baumann and Brewka [BB10] introduce the term "enforcement" as a structural change of an AA framework that makes a desired set of arguments an argument extension. We here use the term differently, to refer to a structural change that makes desired *labels* of arguments *legal*.

legally labelled according to the desired labels. With respect to our structural approach, we propose to enforce the label `in` or `out` onto all arguments in SCUPs. Again, we are not concerned with the exact structural revision of SCUPs as long as it results in all arguments in a SCUP being legally labelled `in` or `out`. In general, enforcing the labels `in` and `out` onto arguments in SCUPs may *not* result in a stable argument labelling of the revised AA framework. Nevertheless, we prove that *iteratively* enforcing the labels `in` and `out` onto arguments in SCUPs results in a stable argument labelling.

The chapter is organised as follows. We provide some additional background on AA frameworks used throughout this chapter in Section 6.2 and introduce an intuitive running example and some preliminary definitions in Section 6.3. In Section 6.4, we define three labelling-based characterisations of parts of an AA framework responsible that no stable argument labelling exists and prove that two of them provide necessary and sufficient conditions for the (non-)existence of stable argument labellings. In Section 6.5, we introduce three structural characterisations of responsible parts: a basic characterisation, odd-length cycles of attacking arguments, and SCUPs. We furthermore propose an iterative method for revising SCUPs, which guarantees to result in a stable argument labelling. We then investigate the relation between our labelling-based and structural characterisations in Section 6.6. In Section 6.7, we discuss some of the design choices underlying our approach and compare our approach to related work. In Section 6.8, we summarise the contributions of this chapter.

6.2 Background

Since this chapter is solely about AA frameworks and there is no risk of confusion, we will call argument labellings simply "labellings" and argument extensions simply "extensions". Furthermore, we call a labelling $LabArg$ with $\texttt{undec}(LabArg) = \emptyset$ an `in`-`out` *labelling.*

Throughout this chapter, we identify complete labellings based on the *legality* of arguments' labels, which is equivalent to the conditions given in Section 2.2.1. Given a labelling $LabArg$ of $\mathcal{AA}$ and an argument $A \in Ar$:

- A is *legally labelled* `in` by $LabArg$ (in $\mathcal{AA}$) if and only if $A \in \texttt{in}(LabArg)$ and $\forall B \in Ar$ attacking A it holds that $B \in \texttt{out}(LabArg)$;
- A is *legally labelled* `out` by $LabArg$ (in $\mathcal{AA}$) if and only if $A \in \texttt{out}(LabArg)$ and $\exists B \in Ar$ attacking A such that $B \in \texttt{in}(LabArg)$;
- A is *legally labelled* `undec` by $LabArg$ (in $\mathcal{AA}$) if and only if $A \in \texttt{undec}(LabArg)$ and $\exists B \in Ar$ attacking A such that $B \in \texttt{undec}(LabArg)$, and $\forall C \in Ar$ attacking A it holds that $C \notin \texttt{in}(LabArg)$.

A is *legally labelled* by $LabArg$ (in $\mathcal{AA}$) if and only if it is legally labelled `in`, `out`, or `undec` by $LabArg$ (in $\mathcal{AA}$); otherwise A is *illegally labelled* by $LabArg$ (in $\mathcal{AA}$). Equivalently we say that a label *is legal/illegal* w.r.t. $LabArg$ (in $\mathcal{AA}$).

A labelling *LabArg* of $\mathcal{AA}$ is a *complete labelling* of $\mathcal{AA}$ if and only if all arguments $A \in Ar$ are legally labelled by *LabArg* (in $\mathcal{AA}$). Preferred and stable labellings are defined based on complete labellings as in Section 2.2.1.

Given a set of arguments $Args \subseteq Ar$, $\mathcal{AA}{\downarrow}_{Args} = \langle Args, Att_{Args} \rangle$ denotes the *restriction of* $\mathcal{AA}$ *to Args*, where $Att_{Args} = Att \cap (Args \times Args)$. Furthermore, given a labelling *LabArg* of $\mathcal{AA}$, $LabArg{\downarrow}_{Args} = LabArg \cap (Args \times \{\texttt{in}, \texttt{out}, \texttt{undec}\})$ denotes the *restriction of LabArg to Args* [BBC$^+$14].

Example 6.1. Let $\mathcal{AA}_2$ be the AA framework on the left of Figure 6.1, which has only one complete labelling, also illustrated on the left of the figure. Given the set of arguments $\{a, b\}$, $\mathcal{AA}_2{\downarrow}_{\{a,b\}}$ is depicted on the right of Figure 6.1 along with the labelling $LabArg{\downarrow}_{\{a,b\}}$.

$$\begin{tikzcd}
a \arrow[r] & b \arrow[loop above] \arrow[r, leftrightarrow] & c & & a \arrow[r] & b \arrow[loop above] \\
\texttt{in} & \texttt{out} & \texttt{in} & & \texttt{in} & \texttt{out}
\end{tikzcd}$$

Figure 6.1: Left – $\mathcal{AA}_2$ and its only complete labelling *LabArg*. Right – $\mathcal{AA}_2{\downarrow}_{\{a,b\}}$ and the labelling $LabArg{\downarrow}_{\{a,b\}}$.

Given a set of arguments *Args*, we denote by *parents*(*Args*) the set of all arguments that are not contained in *Args* and attack *Args*, i.e. $parents(Args) = \{A \in Ar \mid (A, B) \in Att, A \notin Args, B \in Args\}$.

A *path* from argument $A \in Ar$ to argument $B \in Ar$ is a sequence of arguments $A_0, A_1, \ldots, A_n$ ($n > 0$, $\forall i \in \{0, \ldots, n\} : A_i \in Ar$) with $A_0 = A$ and $A_n = B$ such that $\forall i \in \{0, \ldots, n-1\} : A_i$ attacks A_{i+1}. A *cycle* is a path $A_0, A_1, \ldots, A_n$ where $A_n = A_0$. It is an *odd-length* cycle if n is odd. With an abuse of notation, we denote a cycle as a set of arguments $\mathscr{C}$, where $A_i \in \mathscr{C}$ means that argument A_i occurs in cycle $\mathscr{C}$.

Path-equivalence between two arguments $A \in Ar$ and $B \in Ar$ holds if and only if $A = B$ or there exists a path both from A to B and from B to A. The equivalence classes of arguments under the relation of path-equivalence are called *strongly connected components* (SCCs) of $\mathcal{AA}$ [BGG05]. Since SCCs are sets of arguments, the notion of attacks between sets of arguments can be straightforwardly lifted to a notion of attacks between SCCs. Given an SCC $Args \subseteq Ar$, the set of *parent SCCs* is $parentSCCs(Args) = \{Args' \subseteq Ar \mid Args' \textit{ is an SCC of } \mathcal{AA}, Args' \cap parents(Args) \neq \emptyset\}$. If $parentSCCs(Args) = \emptyset$, then *Args* is an *initial SCC*. Furthermore, the set of *ancestor SCCs* of *Args* is $ancestorSCCs(Args) = parentSCCs(Args) \cup \bigcup_{Args' \in parentSCCs(Args)} ancestorSCCs(Args')$.

Example 6.2. $\mathcal{AA}_2$ (see left of Figure 6.1) has one odd-length cycle, namely $\{b\}$, and two SCCs, namely $\{a\}$ and $\{b, c\}$, where the former attacks the latter. $parentSCCs(\{a\}) = ancestorSCCs(\{a\}) = \emptyset$ and $parentSCCs(\{b, c\}) = ancestorSCCs(\{b, c\}) = \{a\}$, so $\{a\}$ is an initial SCC.

An *AA framework with input* [BBC$^+$14] is a tuple $\mathcal{AA}_I = (\mathcal{AA}, I, \mathit{LabArg}_I, \mathit{Att}_I)$ where I is a set of *input* arguments such that $I \cap Ar = \emptyset$, LabArg_I is the *input labelling* of I (i.e. a labelling of I), and Att_I is an attack relation between I and Ar, i.e. $\mathit{Att}_I \subseteq (I \times Ar)$. We say that argument $A \in I$ *attacks* argument $B \in Ar$ if $(A, B) \in \mathit{Att}_I$.

The semantics of an AA framework with input is defined as follows. A labelling LabArg of $\mathcal{AA}$ is a *complete labelling w.r.t.* $\mathcal{AA}_I$ if and only if for all $A \in Ar$ it holds that[2]:

- if $A \in \texttt{in}(\mathit{LabArg})$, then $\forall B \in Ar$ attacking A it holds that $B \in \texttt{out}(\mathit{LabArg})$ and $\forall B \in I$ attacking A it holds that $B \in \texttt{out}(\mathit{LabArg}_I)$;
- if $A \in \texttt{out}(\mathit{LabArg})$, then $\exists B \in Ar$ attacking A such that $B \in \texttt{in}(\mathit{LabArg})$ or $\exists B \in I$ attacking A such that $B \in \texttt{in}(\mathit{LabArg}_I)$;
- if $A \in \texttt{undec}(\mathit{LabArg})$, then $\exists B \in Ar$ attacking A such that $B \in \texttt{undec}(\mathit{LabArg})$ or $\exists B \in I$ attacking A such that $B \in \texttt{undec}(\mathit{LabArg}_I)$, and $\forall B \in Ar$ attacking A it holds that $B \notin \texttt{in}(\mathit{LabArg})$ and $\forall B \in I$ attacking A it holds that $B \notin \texttt{in}(\mathit{LabArg}_I)$.

A labelling LabArg of $\mathcal{AA}$ is a *stable labelling w.r.t.* $\mathcal{AA}_I$ if and only if LabArg is a complete labelling w.r.t. $\mathcal{AA}_I$ and $\texttt{undec}(\mathit{LabArg}) = \emptyset$. We sometimes say that LabArg is a complete/stable labelling *of $\mathcal{AA}$ w.r.t. its input I.*

Example 6.3. An AA framework with input $(\mathcal{AA}_2, I, \mathit{LabArg}_I, \mathit{Att}_I)$ is depicted in Figure 6.2, where the set of input arguments is $I = \{a', b'\}$, the labelling of input arguments is $\mathit{LabArg}_I = \{(a', \texttt{in}), (b', \texttt{undec})\}$, and $\mathit{Att}_I = \{(a', a)\}$. There are two complete labellings w.r.t. $(\mathcal{AA}_2, I, \mathit{LabArg}_I, \mathit{Att}_I)$, namely $\{(a, \texttt{out}), (b, \texttt{undec}), (c, \texttt{undec})\}$ and $\{(a, \texttt{out}), (b, \texttt{out}), (c, \texttt{in})\}$, where the latter is a stable labelling w.r.t. $(\mathcal{AA}_2, I, \mathit{LabArg}_I, \mathit{Att}_I)$.

$$\begin{tikzcd}
b' \arrow[loop above] & a' \arrow[r, leftrightarrow] & a \arrow[r] & b \arrow[loop above] \arrow[r, leftrightarrow] & c \\
\texttt{undec} & \texttt{in} & & &
\end{tikzcd}$$

Figure 6.2: The AA framework with input from Example 6.3.

6.3 Preliminaries

6.3.1 Running Example

Throughout this chapter, we will use an intuitive medical example, which illustrates why the non-existence of stable labellings is problematic in situations which require to make a definite decision. Consider a physician who needs to decide which of five possible therapies to recommend to her patient. She first reads a study praising therapy A and concluding that therapy A is way more effective than therapy B. This study thus provides an argument

[2]Baroni et al. [BBC$^+$14] call this the “canonical local function” of the complete semantics.

for the effectiveness of therapy A and positions it as a counterargument against any argument stating that therapy B is effective. A second article recommends therapy B, showing that it is more reliable than therapy C and much more effective than therapy D. The physician reviews a third study, which describes the enormous success of therapy C and the poor performance of therapy A compared to C. Another article advocates therapy D, but also reveals that therapy D is controversial, sometimes scoring high effectiveness and sometimes poor performance. Finally, a fifth article discusses therapy E, recommending not to apply this therapy. The AA framework representing the physician's reasoning on the effectiveness of the five therapies, which we denote $\mathcal{AA}_{therapy}$ and which is illustrated in Figure 6.3), has no stable labelling, so no conclusion about any of the therapies can be drawn.

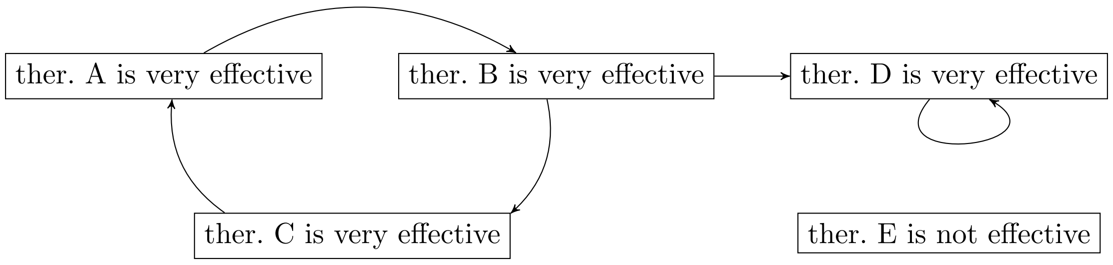

ther. A is very effective
ther. B is very effective
ther. D is very effective
ther. C is very effective
ther. E is not effective

Figure 6.3: $\mathcal{AA}_{therapy}$ representing the physician's reasoning about therapies according to information from scientific articles.

The only preferred labelling (and also the only semi-stable labelling) of $\mathcal{AA}_{therapy}$ labels all arguments as `undec` except the argument "therapy E is not effective", which is labelled `in`. Thus, even using a semantics that is "as decisive as possible", the physician cannot make any decision as to which therapy to prescribe. The only conclusion she can draw is that therapy E is definitely not effective. The non-existence of stable labellings thus poses a problem.

From here onwards, we use a shorthand notation for each argument according to the letter of the respective therapy, e.g. A denotes the argument "therapy A is very effective".

6.3.2 Preliminary Definitions and Results

The aim of this chapter is to give characterisations of parts of an AA framework that are responsible for the non-existence of stable labellings and to provide methods for obtaining a stable labelling. We start with the observation that if an AA framework has no stable labellings, then none of its preferred labellings is a stable labelling.[3] This observation is used for our characterisations by defining responsibility that no stable labelling exists in terms of responsibility that a *preferred* labelling is not a stable labelling. Similarly, we define methods for obtaining a stable labelling in terms of turning a *preferred* labelling into a stable labelling. This is achieved by re-labelling arguments labelled `undec` by a

[3]This follows from the fact that every stable labelling is a preferred labelling [CG09].

preferred labelling as `in` or `out`, in particular arguments identified as responsible that this preferred labelling is not a stable labelling.

It is however not sufficient to simply re-label arguments to obtain a stable labelling; in addition, we have to ensure that the new labels are *legal*. This will be achieved by "enforcing" the new labels, i.e. by structurally revising the AA framework in such a way that the new labels become legal. Since we are only interested in enforcing the labels of *certain* arguments (usually those with new labels, which have been identified as responsible), we restrict structural revisions to these arguments.

We therefore introduce *set-driven* revisions, which ensure that labels (according to some desired labelling) of arguments in a given set become legal, while not making any structural changes affecting arguments not in the set.

Definition 6.1 (Set-Driven Revision and Revision Labelling)**.** Let $LabArg$ be a labelling of $\mathcal{AA}$ and let $Args \subseteq Ar$. A *(set-driven) revision* of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ is $\mathcal{AA}^{\circledast} = \langle Ar^{\circledast}, Att^{\circledast} \rangle$ such that:

- $Ar \subseteq Ar^{\circledast}$;
- $\{(A, B) \in Att \mid B \in Ar \setminus Args\} = \{(A, B) \in Att^{\circledast} \mid B \in Ar \setminus Args\}$;
- $\exists LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ satisfying that:
  - $\forall C \in Ar$: $LabArg^{\circledast}(C) = LabArg(C)$;
  - $\forall D \in Ar^{\circledast} \setminus Ar$: D is legally labelled `in` or `out` by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$.
  - $\forall E \in Args$: E is legally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$;

Any such $LabArg^{\circledast}$ is called a *revision labelling* of $\mathcal{AA}^{\circledast}$.

Since a set-driven revision enforces desired labels onto arguments in the given set $Args$, we do not allow the deletion of arguments in $Args$. A set-driven revision may thus only include the *addition* of new arguments (specified by the first bullet in Definition 6.1). Furthermore, structural changes which may affect (the legality of labels of) arguments *not* in $Args$ are not allowed. Thus, all attacks on arguments not in $Args$ have to remain the same in the revision (specified by the second bullet). Since $LabArg$ specifies the desired labels of all arguments, a revision labelling is a simple "enlargement" of $LabArg$ to include (legal) labels of new arguments; the labels of all other arguments remain unchanged (specified by the first and second item of the third bullet). Furthermore, and most importantly, a revision labelling ensures that all arguments in $Args$ are *legally* labelled in the revision (specified by the third item of the third bullet).

From here onwards, we will refer to set-driven revisions simply as *revisions*.

Example 6.4. Let $LabArg$ be the labelling of $\mathcal{AA}_{therapy}$ illustrated in Figure 6.4. Figure 6.5 depicts a revision of $\mathcal{AA}_{therapy}$ w.r.t. $\{A\}$ by $LabArg$, which we denote $\mathcal{AA}^{\circledast}_{therapy}$, and the labelling in Figure 6.5 is a revision labelling of $\mathcal{AA}^{\circledast}_{therapy}$. Note that $\mathcal{AA}^{\circledast}_{therapy}$ is also a revision of $\mathcal{AA}_{therapy}$ w.r.t. any superset of $\{A\}$ by $LabArg$.

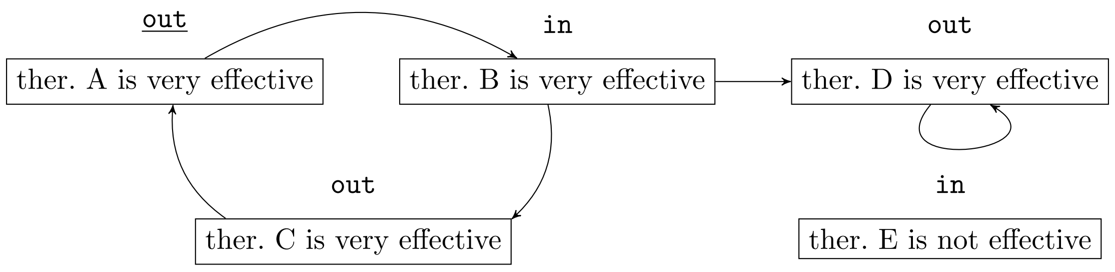

out
in
out
ther. A is very effective
ther. B is very effective
ther. D is very effective
out
in
ther. C is very effective
ther. E is not effective

Figure 6.4: $\mathcal{AA}_{therapy}$ and a labelling *LabArg* (underlined labels are illegal).

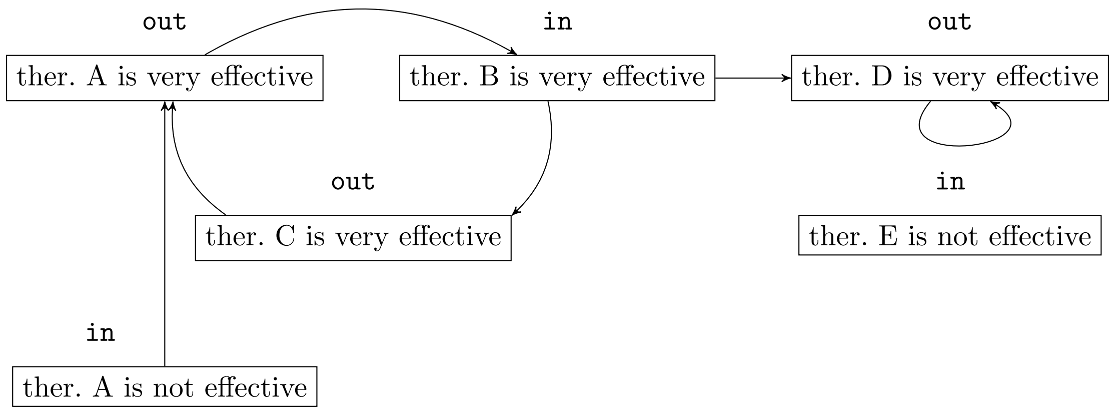

out
in
out
ther. A is very effective
ther. B is very effective
ther. D is very effective
out
in
ther. C is very effective
ther. E is not effective
in
ther. A is not effective

Figure 6.5: A revision of $\mathcal{AA}_{therapy}$ w.r.t. $\{A\}$ by *LabArg*.

Example 6.4 anticipates how we will use revisions in the context of turning a preferred labelling into a stable labelling. As previously explained, the only preferred labelling of $\mathcal{AA}_{therapy}$ labels all arguments as `undec` except argument E, which is labelled `in`. In order to turn this preferred labelling into a stable labelling, we may thus change all `undec` labels to `in` or `out` labels. One such option is the labelling *LabArg* illustrated in Figure 6.4. However, since in this labelling not all arguments are legally labelled, in particular argument A is illegally labelled, we perform a revision w.r.t. $\{A\}$ by the desired labelling *LabArg*, obtaining an AA framework where A is legally labelled, as illustrated in Figure 6.5. The desired labelling *LabArg* (plus the label `in` of the newly added argument) is now a stable labelling of the structurally revised AA framework. Throughout this chapter, we will characterise different sets of arguments that are good choices for revisions, in particular sets of arguments that are *responsible* that the preferred labelling in question is not a stable labelling.

In the following lemma, we show that a revision exists for any given set of arguments and labelling. This means that any labelling can be "enforced" onto a set of arguments through a structural change as given by Definition 6.1.

Lemma 6.1. *Let LabArg be a labelling of $\mathcal{AA}$ and let $Args \subseteq Ar$. Then there exists a revision of $\mathcal{AA}$ w.r.t. $Args$ by LabArg.*

Proof. Let $\mathcal{AA}^{\circledast} = \langle Ar^{\circledast}, Att^{\circledast} \rangle$ be such that $Ar^{\circledast} = Ar \cup \{X\}$ where $X \notin Ar$ and $Att^{\circledast} =$

$(Att \setminus \{(B,A) \in Att \mid A \in Args, A \in \mathtt{in}(LabArg) \cup \mathtt{undec}(LabArg)\}) \cup (\{(X,A) \mid A \in Args, A \in \mathtt{out}(LabArg)\} \cup \{(A,A) \mid A \in Args, A \in \mathtt{undec}(LabArg)\})$. Let $LabArg^{\circledast} = LabArg \cup \{(X,\mathtt{in})\}$. Then clearly $Ar \subseteq Ar^{\circledast}$ and $\{(A,B) \in Att \mid B \in Ar \setminus Args\} = \{(A,B) \in Att^{\circledast} \mid B \in Ar \setminus Args\}$, and $\forall C \in Ar$: $LabArg^{\circledast}(C) = LabArg(C)$.

Let $A \in Args$. If $A \in \mathtt{in}(LabArg^{\circledast})$, then A is not attacked by any argument B in $\mathcal{AA}^{\circledast}$, so trivially for all attackers B of A in $\mathcal{AA}^{\circledast}$, $B \in \mathtt{out}(LabArg^{\circledast})$. Thus, A is legally labelled $\mathtt{in}$ by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. If $A \in \mathtt{out}(LabArg^{\circledast})$, then A is attacked by X in $\mathcal{AA}^{\circledast}$ and $X \in \mathtt{in}(LabArg^{\circledast})$, so A is legally labelled $\mathtt{out}$ by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. If $A \in \mathtt{undec}(LabArg^{\circledast})$, then A is only attacked by itself in $\mathcal{AA}^{\circledast}$. Thus, there exists an attacker of A in $\mathcal{AA}^{\circledast}$ labelled $\mathtt{undec}$ by $LabArg^{\circledast}$ and there exists no attacker of A in $\mathcal{AA}^{\circledast}$ labelled $\mathtt{in}$ by $LabArg^{\circledast}$, so A is legally labelled $\mathtt{undec}$ by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$.

Since furthermore $X \in Ar^{\circledast} \setminus Ar$ is legally labelled $\mathtt{in}$ by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$, $\mathcal{AA}^{\circledast}$ and $LabArg^{\circledast}$ satisfy the conditions in Definition 6.1, so $\mathcal{AA}^{\circledast}$ is a revision of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$. □

Note that we are not concerned with the *exact* structural change of a revision compared to the original AA framework. We simply use the structural change of an AA framework as a tool to ensure that labels of arguments are legal. As a result, there may be various revisions of an AA framework w.r.t. a given set of arguments and labelling. Furthermore, a revision may have various different revision labellings. It is in general up to the preference of users to decide which of these revisions and revision labellings to use. For example, a user may be interested in revisions with "minimal" structural changes as in [Bau12, CMKMM14b].

Example 6.5. Let $\mathcal{AA}_3$ be the AA framework depicted on the left of Figure 6.6 and $LabArg$ the labelling of $\mathcal{AA}_3$ illustrated on the left of Figure 6.6, which is the labelling we desire. Argument a is illegally labelled by $LabArg$, so a revision can be used to enforce the desired label onto argument a. A possible revision of $\mathcal{AA}_3$ w.r.t. $\{a\}$ by $LabArg$ is illustrated on the right of Figure 6.6 alongside a revision labelling. Another revision of $\mathcal{AA}_3$ w.r.t. $\{a\}$ by $LabArg$ is illustrated in Figure 6.7 alongside two different revision labellings.

Next, we extend the comparison notion of *commitment* of two labellings of an AA framework [CG09] to the comparison of labellings of potentially different AA frameworks, where the arguments of one AA framework form a superset of the arguments of the other.

Definition 6.2 (Commitment of Labellings)**.** Let $LabArg$ be a labelling of $\mathcal{AA}$ and let $LabArg'$ be a labelling of $\mathcal{AA}' = \langle Ar', Att' \rangle$, where $Ar \subseteq Ar'$.

- $LabArg'$ is *more or equally committed* than $LabArg$, denoted $LabArg \sqsubseteq LabArg'$, if and only if $\mathtt{in}(LabArg) \subseteq \mathtt{in}(LabArg')$, $\mathtt{out}(LabArg) \subseteq \mathtt{out}(LabArg')$ and $\mathtt{undec}(LabArg') \subseteq \mathtt{undec}(LabArg)$.
- $LabArg'$ is *more committed* than $LabArg$, denoted $LabArg \sqsubset LabArg'$, if and only if $LabArg \sqsubseteq LabArg'$ and $\mathtt{undec}(LabArg') \subset \mathtt{undec}(LabArg)$.

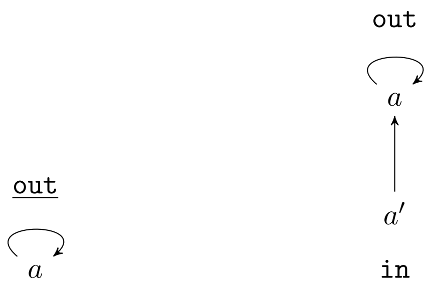


Figure 6.6: Left – $\mathcal{AA}_3$ and a labelling *LabArg*, where the underline indicates that the argument is illegally labelled. Right – A revision of $\mathcal{AA}_3$ and its only revision labelling (see Example 6.5).

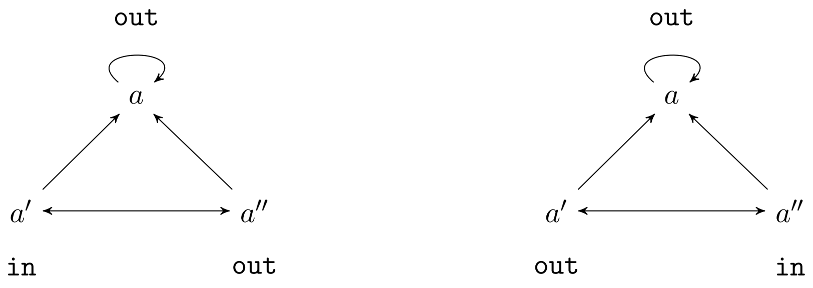


Figure 6.7: A revision of $\mathcal{AA}_3$, which has two different revision labellings (see Example 6.5).

Since the set of arguments of a revision is a subset of or equal to the set of arguments of the original AA framework, and since a revision labelling of the revision labels all arguments of the original AA framework the same as the original labelling used to obtain the revision and new arguments as `in` or `out`, a revision labelling is more or equally committed than the original labelling.

Observation 6.2. *Let LabArg be a labelling of* $\mathcal{AA}$ *and* $Args \subseteq Ar$. *Then, for all revisions* $\mathcal{AA}^{\circledast}$ *of* $\mathcal{AA}$ *w.r.t. Args by LabArg and all revision labellings* $LabArg^{\circledast}$ *of* $\mathcal{AA}^{\circledast}$ *it holds that* $LabArg \sqsubseteq LabArg^{\circledast}$.

For instance, the two revision labellings of the revision of $\mathcal{AA}_3$ illustrated in Figure 6.7 (see Example 6.5) are more committed than the original labelling of $\mathcal{AA}_3$, depicted on the left of Figure 6.6.

In the remainder, and if not stated otherwise, we assume that $\mathcal{AA} = \langle Ar, Att \rangle$ has no stable labelling and that $LabArg_{pref}$ is a preferred labelling[4] of $\mathcal{AA}$. When talking about *the* preferred labelling, we therefore do not suggest that $\mathcal{AA}$ has only one preferred labelling, but rather we refer to the preferred labelling $LabArg_{pref}$ in question.

[4]By Corollary 12 in [Dun95b] $\mathcal{AA}$ has at least one preferred extension, and therefore, by the correspondence between extensions and labellings [CG09], $\mathcal{AA}$ has at least one preferred labelling.

6.4 Labelling-Based Characterisations

As previously explained, we aim to 1) characterise sets of arguments responsible for the non-existence of stable labellings in terms of sets of arguments responsible that a preferred labelling is not a stable labelling, and 2) use these responsible sets to turn the preferred labelling in question into a stable labelling. In this section, we give three declarative characterisations of sets of arguments that are responsible for LabArg_{pref} not to be a stable labelling. These characterisations are *labelling-based*, which means that they identify responsible sets based on labellings that are more or equally committed than LabArg_{pref}. In other words, the characterisations rely purely on changing `undec` labels in the preferred labelling to `in` or `out` labels and checking which of the new labels are illegal. The structure of the AA framework is not explicitly taken into account. We also investigate how our characterisations relate to revisions of the AA framework which (do not) have a stable labelling that is more committed than LabArg_{pref}. In particular, we show that our two non-naive characterisations, which we introduce in Sections 6.4.2 and 6.4.3, define *necessary and sufficient* conditions for the existence and non-existence of a stable labelling of a revision.

6.4.1 The Basic Approach

A naive way to characterise arguments responsible for LabArg_{pref} not being a stable labelling is in terms of *all* arguments labelled `undec` by LabArg_{pref}, since these are the arguments violating the definition of stable labelling.

Definition 6.3 (Labelling-Based Characterisation 1). $\mathtt{undec}(\mathit{LabArg}_{pref})$ is the *labelling-based responsible set* w.r.t. LabArg_{pref}.

It is straightforward to use this characterisation of a set of arguments responsible for the non-existence of stable labellings to obtain a stable labelling. The following proposition proves that re-labelling all arguments in the labelling-based responsible set as `in` or `out` and ensuring that these new labels are legal by constructing a revision, results in a *stable labelling* of the revision. Thus, the labelling-based responsible set provides a *sufficient* condition for obtaining a stable labelling through a revision.

Proposition 6.3. *Let Args be the labelling-based responsible set w.r.t. LabArg_{pref} and let LabArg be a labelling such that $\mathit{LabArg}_{pref} \sqsubset \mathit{LabArg}$ and $\mathtt{undec}(\mathit{LabArg}) = \emptyset$. Then, for all revisions $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by LabArg and all revision labellings $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$, $\mathit{LabArg}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

Proof. Since $\mathtt{undec}(\mathit{LabArg}) = \emptyset$, it follows from Observation 6.2 that $\mathtt{undec}(\mathit{LabArg}^{\circledast}) = \emptyset$. Furthermore, by Definition 6.1 all $A \in \mathit{Ar}^{\circledast} \setminus \mathit{Ar}$ are legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Let $B \in \mathit{Ar}$. If $B \in \mathit{Args}$, then by Definition 6.1 B is legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. If $B \notin \mathit{Args}$, then $B \in \mathtt{in}(\mathit{LabArg}_{pref}) \cup \mathtt{out}(\mathit{LabArg}_{pref})$, so B is legally labelled by LabArg_{pref} in $\mathcal{AA}$. By Lemma A.2 in Appendix A, B is legally labelled by LabArg

in $\mathcal{AA}$, and by Lemma A.1 in Appendix A, B is legally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Since all arguments in $\mathcal{AA}^{\circledast}$ are legally labelled by $LabArg^{\circledast}$ and $\texttt{undec}(LabArg^{\circledast}) = \emptyset$, $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. $\square$

Example 6.6. Consider again $\mathcal{AA}_{therapy}$ from Example 6.4 and its only preferred labelling $LabArg_{pref}$, which labels all arguments `undec` except for argument E, which is labelled `in`. Thus, the labelling-based responsible set w.r.t. $LabArg_{pref}$ is $\{A, B, C, D\}$. Let $LabArg$ be the labelling of $\mathcal{AA}_{therapy}$ illustrated in Figure 6.4. The revision labelling of the revision $\mathcal{AA}^{\circledast}_{therapy}$ of $\mathcal{AA}_{therapy}$ w.r.t. $\{A, B, C, D\}$ by $LabArg$ (see Figure 6.5) is a stable labelling of $\mathcal{AA}^{\circledast}_{therapy}$.

Since by Lemma 6.1 a revision exists w.r.t. any set of arguments and any labelling, it follows that there exists a revision w.r.t. the labelling-based responsible set, and in particular (by Proposition 6.3) a revision that has a stable labelling.

Corollary 6.4. *Let Args be the labelling-based responsible set w.r.t. $LabArg_{pref}$ and let LabArg be labelling such that $LabArg_{pref} \sqsubset LabArg$ and* $\texttt{undec}(LabArg) = \emptyset$*. Then, there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by LabArg and and a revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

Note that, by Observation 6.2, a stable labelling obtained through such a revision is more committed than $LabArg_{pref}$. Thus, as desired, the labelling-based responsible set can be used to turn a preferred labelling into a stable labelling.

6.4.2 Enforcement Sets

The definition of labelling-based responsible set is a rather naive characterisation of arguments responsible for the preferred labelling not to be a stable labelling, since it is often possible to legally label some of its arguments `in` or `out`. For example, considering the arguments A, B, C, and D labelled `undec` by the preferred labelling of $\mathcal{AA}_{therapy}$ (see Figure 6.3), we observe that three out of these four arguments can in fact be legally labelled `in` or `out`, as illustrated in Figure 6.4 (only argument A is illegally labelled).

Our next characterisation takes this observation into account, characterising specific subsets of the labelling-based responsible set as responsible. In particular, arguments that are legally labelled by an `in-out` labelling that is more committed than $LabArg_{pref}$ will not be deemed responsible. More precisely, a set of responsible arguments according to our second labelling-based characterisation is defined as a *minimal* subset of arguments labelled `undec` by $LabArg_{pref}$ satisfying that some `in-out` labelling that is more committed than $LabArg_{pref}$ legally labels all non-responsible arguments (i.e. all arguments not contained in this set).

Definition 6.4 (Labelling-Based Characterisation 2). *Args* is an *enforcement set* w.r.t. $LabArg_{pref}$ if and only if it is a minimal set of arguments (w.r.t. $\subseteq$) such that

$Args \subseteq \texttt{undec}(LabArg_{pref})$ and

$\exists$*LabArg* of $\mathcal{AA}$ with $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\texttt{undec}(\mathit{LabArg}) = \emptyset$ such that
$\forall A \in \texttt{undec}(\mathit{LabArg}_{\mathit{pref}}) \setminus \mathit{Args}$: A is legally labelled by *LabArg*.

Any such *LabArg* is an *enforcement labelling* w.r.t. *Args*.

Example 6.7. Consider again $\mathcal{AA}_{\mathit{therapy}}$ and its only preferred labelling $\mathit{LabArg}_{\mathit{pref}}$ (see Example 6.6). Then $\{A\}$ is an enforcement set w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, where the labelling shown in Figure 6.4 is an enforcement labelling as it is an `in-out` labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$ and it legally labels all arguments labelled `undec` by $\mathit{LabArg}_{\mathit{pref}}$ except for argument A (i.e. arguments B, C, and D). Furthermore, $\{A\}$ is a minimal set satisfying this condition, since for its only subset $\{\}$ there exists no `in-out` labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$ and that legally labels *all* arguments labelled `undec` by $\mathit{LabArg}_{\mathit{pref}}$. There are two more enforcement sets w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, namely $\{B\}$, and $\{C\}$. Note that $\{D\}$ is not an enforcement set since there exists no `in-out` labelling that legally labels A, B, and C.

In Example 6.7, all enforcement sets are disjoint. The following example illustrates that different enforcement sets may contain the same arguments and that an enforcement set may have various different enforcement labellings.

Example 6.8. Let $\mathcal{AA}_4$ be the AA framework on the left of Figure 6.8, whose only preferred labelling $\mathit{LabArg}_{\mathit{pref}}$ labels all arguments as `undec`. There are three enforcement sets w.r.t. $\mathit{LabArg}_{\mathit{pref}}$: $\{a, e\}$, $\{b, e\}$, and $\{c, e\}$. Note that for all of them various enforcement labellings exist, e.g. the labelling illustrated on the left of Figure 6.8 is an enforcement labelling of $\{b, e\}$, and so is $\{(a, \texttt{out}), (b, \texttt{out}), (c, \texttt{in}), (d, \texttt{in}), (e, \texttt{in})\}$ (among others).

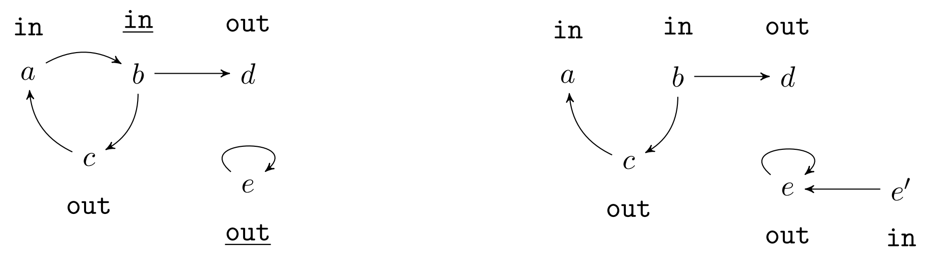


Figure 6.8: Left – $\mathcal{AA}_4$ and labelling *LabArg*, where underlined labels are illegal. Right – A revision $\mathcal{AA}_4^{\circledast}$ of $\mathcal{AA}_4$ by *LabArg* and a revision labelling that is a stable labelling of $\mathcal{AA}_4$ (see Examples 6.8 and 6.9).

It follows from Definition 6.4 that all arguments in an enforcement set are illegally labelled by an enforcement labelling. For example, arguments b and e are illegally labelled by both enforcement labellings discussed in Example 6.8. It is important to note that, nevertheless, enforcement labellings cannot be equivalently defined as minimal sets of arguments that are illegally labelled by an enforcement labelling, as this would always yield the empty set as the only enforcement set. Rather, an enforcement set is a minimal set of arguments consisting of *all* the illegally labelled arguments w.r.t. an enforcement labelling.

In the following lemma, we show that at least one enforcement set exists and that enforcement sets are always non-empty. Both are important properties for sets of arguments characterising parts of an AA framework responsible for the non-existence of stable labellings.

Lemma 6.5.

1. *There exists an enforcement set w.r.t. $LabArg_{pref}$.*

2. *If $Args$ is an enforcement set w.r.t. $LabArg_{pref}$, then $Args \neq \emptyset$.*

Proof.

1. Let $Args = \texttt{undec}(LabArg_{pref})$. Clearly there exists some $LabArg$ with $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$. Then trivially, $\forall A \in \texttt{undec}(LabArg_{pref}) \backslash Args = \emptyset$ it holds that A is legally labelled by $LabArg$. Thus, $Args$ and $LabArg$ satisfy the conditions in Definition 6.4, but $Args$ may not be a minimal set satisfying the conditions. If for all $Args_1 \subset Args$ and for all $LabArg'$ of $\mathcal{AA}$ with $LabArg_{pref} \sqsubset LabArg'$ and $\texttt{undec}(LabArg') = \emptyset$ there exists some $A \in \texttt{undec}(LabArg_{pref}) \setminus Args_1$ which is illegally labelled by $LabArg'$, then $Args$ is a minimal set satisfying the conditions in Definition 6.4, so it is an enforcement set (and $LabArg$ an enforcement labelling w.r.t. $Args$). Else, there is a smallest $Args_1 \subset Args$ satisfying that $\exists LabArg_1$ with $LabArg_{pref} \sqsubset LabArg_1$ and $\texttt{undec}(LabArg_1) = \emptyset$ such that $\forall A \in \texttt{undec}(LabArg_{pref}) \setminus Args_1$: A is legally labelled by $LabArg_1$. Thus, $Args_1$ is an enforcement set (and $LabArg_1$ an enforcement labelling).

2. Let $LabArg$ be an enforcement labelling w.r.t. $Args$. If $Args = \emptyset$, then by Lemma A.3 in Appendix A it holds that all arguments in Ar are legally labelled by $LabArg$, so since $\texttt{undec}(LabArg) = \emptyset$, $LabArg$ is a stable labelling. Contradiction since $\mathcal{AA}$ has no stable labellings.

□

Responsibility of Enforcement Sets

The reason for naming our second labelling-based characterisation "enforcement sets" is illustrated by Theorem 6.6: "enforcing" the labels of an enforcement labelling onto arguments in an enforcement set in terms of a revision, results in a stable labelling. An enforcement set is thus a *sufficient* condition for obtaining a stable labelling through a revision, which is more refined than the condition given by the labelling-based responsible set (since every enforcement set is a subset of the labelling-based responsible set).

Theorem 6.6. *Let $Args \supseteq Args_{enf}$ where $Args_{enf}$ is an enforcement set w.r.t. $LabArg_{pref}$ and let $LabArg$ be an enforcement labelling w.r.t. $Args_{enf}$. Then, for all revisions $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and all revision labellings $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$, $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

Proof. Since by Definition 6.4, $\texttt{undec}(\mathit{LabArg}) = \emptyset$, it follows from Observation 6.2 that $\texttt{undec}(\mathit{LabArg}^{\circledast}) = \emptyset$. By Definition 6.1, all $A \in Ar^{\circledast} \setminus Ar$ are legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Let $B \in Ar$. If $B \in \mathit{Args}$, then by Definition 6.1 B is legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. If $B \notin \mathit{Args}$, and thus B $\notin \mathit{Args}_{\mathit{enf}}$, then by Lemma A.3 in Appendix A, B is legally labelled by LabArg in $\mathcal{AA}$, so by Lemma A.1 in Appendix A, B is legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Since all arguments are legally labelled by $\mathit{LabArg}^{\circledast}$ and $\texttt{undec}(\mathit{LabArg}^{\circledast}) = \emptyset$, $\mathit{LabArg}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. □

Example 6.9. Consider the enforcement set $\{b, e\}$ and the enforcement labelling LabArg of $\mathcal{AA}_4$ illustrated on the left of Figure 6.8. The AA framework on the right of Figure 6.8 is a revision $\mathcal{AA}_4^{\circledast}$ of $\mathcal{AA}_4$ w.r.t. $\{b, e\}$ by LabArg and the revision labelling $\mathit{LabArg}^{\circledast}$ illustrated in the figure is a stable labelling of $\mathcal{AA}_4^{\circledast}$.

Since by Lemma 6.1 a revision exists w.r.t. any set of arguments and labelling, it follows that there exists a revision w.r.t. an enforcement set by an enforcement labelling and that the revision has a stable labelling which is more committed than $\mathit{LabArg}_{\mathit{pref}}$.

Corollary 6.7. *Let $\mathit{Args} \supseteq \mathit{Args}_{\mathit{enf}}$ where $\mathit{Args}_{\mathit{enf}}$ is an enforcement set w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ and let LabArg be an enforcement labelling w.r.t. $\mathit{Args}_{\mathit{enf}}$. Then there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by LabArg and a revision labelling $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $\mathit{LabArg}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

6.4.3 Preventing Sets

Enforcement sets characterise a responsible set of arguments with respect to a *specific* more committed labelling, which labels *all* arguments in this set illegally. Our second non-naive characterisation instead defines a responsible set of arguments as containing at least *one* illegally labelled argument with respect to *every* `in-out` labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$.

Definition 6.5 (Labelling-Based Characterisation 3)**.** *Args is a preventing set* w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ if and only if it is a minimal set of arguments (w.r.t. $\subseteq$) such that
$\mathit{Args} \subseteq \texttt{undec}(\mathit{LabArg}_{\mathit{pref}})$ and
$\forall \mathit{LabArg}$ of $\mathcal{AA}$ with $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\texttt{undec}(\mathit{LabArg}) = \emptyset$ it holds that
$\exists A \in \mathit{Args}$ such that A is illegally labelled by LabArg.

Example 6.10. Consider again $\mathcal{AA}_{\mathit{therapy}}$ and its only preferred labelling $\mathit{LabArg}_{\mathit{pref}}$ (see Example 6.6). The only preventing set w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ is $\{A, B, C\}$, since no matter how the labels `in` and `out` are assigned to this set of arguments, at least one argument is illegally labelled. In contrast, for all subsets there exists some `in-out` labelling that labels all arguments legally. For instance, for the set $\{A, B\}$, an `in-out` labelling that labels A as `in` and B and C as `out` legally labels both A and B.

Similarly to enforcement sets, at least one preventing set exists w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ and preventing sets are always non-empty.

Lemma 6.8.

1. *There exists a preventing set w.r.t. $LabArg_{pref}$.*

2. *If $Args$ is a preventing set w.r.t. $LabArg_{pref}$, then $Args \neq \emptyset$.*

Proof.

1. Let $Args = \texttt{undec}(LabArg_{pref})$ and let $LabArg$ be such that $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$. Since $LabArg_{pref}$ is a maximal complete labelling, $\exists A \in Args$ such that A is illegally labelled by $LabArg$. Since this holds for all such labellings $LabArg$, $Args$ satisfies the conditions in Definition 6.5. However, $Args$ may not be a minimal set satisfying these conditions. If for all $Args_1 \subset Args$ there exists $LabArg_1$ with $LabArg_{pref} \sqsubset LabArg_1$ and $\texttt{undec}(LabArg_1) = \emptyset$ such that all $A \in Args_1$ are legally labelled by $LabArg_1$, then $Args$ is a minimal set satisfying the conditions in Definition 6.5, so it is a preventing set w.r.t. $LabArg_{pref}$. Else, there is a smallest $Args_1 \subset Args$ satisfying that $\forall LabArg'$ with $LabArg_{pref} \sqsubset LabArg'$ and $\texttt{undec}(LabArg') = \emptyset$ it holds that $\exists A \in Args_1$ such that A is illegally labelled by $LabArg'$. Then $Args_1$ is a preventing set w.r.t. $LabArg_{pref}$.

2. Assume $Args = \emptyset$ is a preventing set w.r.t. $LabArg_{pref}$. By Definition 6.5, $\exists A \in Args$ such that A is illegally labelled. Contradiction since $\nexists A \in Args$.

□

Responsibility of Preventing Sets

Theorem 6.9 illustrates the reason for naming our third labelling-based characterisation "preventing sets": any revision w.r.t. a set of arguments not comprising any argument from some preventing set has no stable labelling that is more committed than $LabArg_{pref}$. Thus, preventing sets define a *sufficient* condition for "preventing" the existence of a stable labelling that is more committed than $LabArg_{pref}$.

Theorem 6.9. *Let $Args \subseteq Ar \backslash Args_{prev}$ where $Args_{prev}$ is a preventing set w.r.t. $LabArg_{pref}$. Then for all labellings $LabArg$ of $\mathcal{AA}$ such that $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$, there exists no revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ such that some revision labelling $LabArg^{\circledast}$ of $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

Proof. Assume there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and a revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. By Definition 6.5, $\exists A \in Args_{prev}$ such that A is illegally labelled by $LabArg$ in $\mathcal{AA}$. Since $A \in Ar \setminus Args$, it follows from Lemma A.1 in Appendix A that A is illegally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Contradiction. □

Example 6.11. Recall $\mathcal{AA}_4$, depicted on the left of Figure 6.8, and its only preferred labelling $LabArg_{pref}$, which labels all arguments as $\texttt{undec}$. There are two preventing sets w.r.t. $LabArg_{pref}$, namely $\{a, b, c\}$ and $\{e\}$. Consider the preventing set $\{e\}$ and some `in-out` labelling of $\mathcal{AA}_4$, e.g. $LabArg$ illustrated on the left of Figure 6.8. In order to ensure that e is legally labelled by $LabArg$, an attack on e from an argument labelled `in` has to be added (e.g. as on the right of Figure 6.8). Conversely, if e was labelled `in` by an `in-out` labelling, the self-attack of e would have to be deleted in order to ensure that e was legally labelled. Thus, no revision w.r.t. a set of arguments not containing e can result in e being legally labelled.

6.4.4 Enforcement versus Preventing Sets

Theorems 6.6 and 6.9 hint at a connection between enforcement and preventing sets: one provides a sufficient condition for the *existence* of a stable labelling after revision, the other a sufficient condition for the *non-existence*. In this section, we investigate the relationship between enforcement and preventing sets in more detail.

We first show that a preventing set is a minimal set containing exactly one argument from each enforcement set.

Theorem 6.10. *Let S_{enf} be the set of all enforcement sets w.r.t. $LabArg_{pref}$. Then $S = \{Args \subseteq Ar \mid Args$ is a minimal set satisfying that $\forall Args_{enf} \in S_{enf} : Args \cap Args_{enf} \neq \emptyset\}$ is the set of all preventing sets w.r.t. $LabArg_{pref}$.*

Proof. We prove that all $Args \in S$ are preventing sets and that all preventing sets are contained in S. We note that, by Lemma 6.5, $S_{enf} \neq \emptyset$ and $\forall Args_{enf} \in S_{enf}$: $Args_{enf} \neq \emptyset$.

- Let $Args \in S$ and assume that $Args$ is not a preventing set. Then either $Args$ is not a minimal set satisfying the conditions in Definition 6.5 or $Args$ does not satisfy the conditions at all.
  - In the first case, $\exists Args_{prev} \subset Args$ such that $Args_{prev}$ is a preventing set. Since $Args$ is a minimal set satisfying that $\forall Args_{enf} \in S_{enf} : Args \cap Args_{enf} \neq \emptyset$, it follows that $\exists Args'_{enf} \in S_{enf}$ such that $Args_{prev} \cap Args'_{enf} = \emptyset$. Since $Args'_{enf}$ is an enforcement set there exists an enforcement labelling $LabArg$. By Lemma A.3 in Appendix A it holds that $\forall B \in Ar \setminus Args'_{enf}$, B is legally labelled by $LabArg$. Since $Args_{prev}$ is a preventing set it holds that $\exists C \in Args_{prev}$ such that C is illegally labelled by $LabArg$. Contradiction since $C \in Ar \setminus Args'_{enf}$.
  - In the second case, we note that $Args \subseteq \texttt{undec}(LabArg_{pref})$ since $\forall A \in Args$: $\exists Args_{enf}$ such that $A \in Args_{enf}$ and $Args_{enf} \subseteq \texttt{undec}(LabArg_{pref})$ by Definition 6.4. Thus, $Args$ violates Definition 6.5 because $\exists LabArg$ such that $LabArg_{pref} \sqsubset LabArg$, $\texttt{undec}(LabArg) = \emptyset$, and $\forall A \in Args$ it holds that A is legally labelled by $LabArg$. Let $Args' = Ar \setminus Args$. Then $\forall A \in Ar \setminus Args' = Args$, A is legally labelled by $LabArg$, in particular all $A \in \texttt{undec}(LabArg_{pref}) \setminus Args'$ are legally labelled by $LabArg$. Thus, $Args'$ satisfies the conditions of an enforcement set (disregarding minimality). Since by definition of $Args'$ it holds

that $Args \cap Args' = \emptyset$, $Args'$ is not an enforcement set (by definition of $Args$). Thus, $Args'$ is not a minimal set satisfying the conditions of an enforcement set, i.e. $\exists Args_{enf} \in S_{enf}$ such that $Args_{enf} \subset Args'$. Then, by definition of $Args$ it holds $Args \cap Args_{enf} \neq \emptyset$ and thus $Args \cap Args' \neq \emptyset$. Contradiction.

Thus, $Args$ is a preventing set.

- Let $Args_{prev}$ be a preventing set and assume that $Args_{prev} \notin S$. Then either $\exists Args_{enf} \in S_{enf}$ such that $Args_{prev} \cap Args_{enf} = \emptyset$ or there exists a minimal set $Args \subset Args_{prev}$ satisfying that $Args \cap Args_{enf} \neq \emptyset$ for all $Args_{enf} \in S_{enf}$.
  - In the first case, since $Args_{enf}$ is an enforcement set there exists an enforcement labelling *LabArg*. By Lemma A.3 in Appendix A it holds that $\forall A \in Ar \setminus Args_{enf}$, A is legally labelled by *LabArg*. Since $Args_{prev}$ is a preventing set it holds that $\exists B \in Args_{prev}$ such that B is illegally labelled by *LabArg*. Contradiction since $B \in Ar \setminus Args_{enf}$.
  - In the second case, $Args \in S$, so it follows from the first item of this proof that $Args$ is a preventing set. Contradiction since $Args_{prev}$ is a preventing set (and thus minimal).

Thus, $Args_{prev} \in S$.

□

Example 6.12. From Example 6.8, we know that for $\mathcal{AA}_4$ the set of all enforcement sets is $S_{enf} = \{\{a, e\}, \{b, e\}, \{c, e\}\}$. Then both $\{a, b, c\}$ and $\{e\}$ are minimal sets containing an argument from each enforcement set. Indeed, $\{a, b, c\}$ and $\{e\}$ are the two preventing sets w.r.t. $LabArg_{pref}$ of $\mathcal{AA}_4$ (see Example 6.11).

Conversely, an enforcement set is a minimal set containing exactly one argument from each preventing set.

Theorem 6.11. *Let S_{prev} the set of all preventing sets w.r.t. $LabArg_{pref}$. Then $S = \{Args \subseteq Ar \mid Args$ is a minimal set satisfying that $\forall Args_{prev} \in S_{prev} : Args \cap Args_{prev} \neq \emptyset\}$ is the set of all enforcement sets w.r.t. $LabArg_{pref}$.*

Proof. Analogous to the proof of Theorem 6.10. □

Example 6.13. From Example 6.11, we know that $S_{prev} = \{\{a, b, c\}, \{e\}\}$ for $\mathcal{AA}_4$. Then, $\{a, e\}$, $\{b, e\}$, and $\{c, e\}$ are all the minimal sets containing one argument from each preventing set. Indeed, these three sets are the enforcement sets of $\mathcal{AA}_4$ w.r.t. $LabArg_{pref}$ (see Example 6.8).

These results, together with the results in previous sections, mean that enforcement and preventing sets are two sides of the same coin. The different enforcement sets characterise minimal sets of arguments that, if appropriately revised, yield a stable labelling. Thus, if we take an argument from each enforcement set, then at least one of these arguments needs to be revised in order to obtain a stable labelling; in other words, if none of these arguments is revised then no stable labelling will be obtained. So as stated in Theorem 6.10, these arguments form exactly a preventing set. The same duality holds when considering all preventing sets and selecting one argument from each of them.

6.4.5 Necessary Conditions for the (Non-)Existence of Stable Labellings

Based on the correspondence results between enforcement and preventing sets, we now further investigate their role regarding revisions. We prove that both enforcement and preventing sets define not only sufficient but also *necessary* conditions for the existence and non-existence, respectively, of a stable labelling of a revision.

Firstly, Theorem 6.12 states that any revision whose revision labelling is a stable labelling that is more committed than the preferred labelling was obtained using a superset of some enforcement set. In other words, enforcement sets define a *necessary* condition for obtaining a stable labelling that is more committed than $LabArg_{pref}$.

Theorem 6.12. *Let $Args \subseteq Ar$ and let $LabArg$ be a labelling of $\mathcal{AA}$ such that $LabArg_{pref} \sqsubset LabArg$, $\texttt{undec}(LabArg) = \emptyset$, and there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and a revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. Then there exists an enforcement set $Args_{enf}$ w.r.t. $LabArg_{pref}$ such that $Args_{enf} \subseteq Args$.*

Proof. Let $Args \subseteq Ar$. By (the contrapositive of) Theorem 6.9 it holds that: if there exists a labelling $LabArg$ of $\mathcal{AA}$ such that $LabArg_{pref} \sqsubset LabArg$, $\texttt{undec}(LabArg) = \emptyset$, and there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and a revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$, then $Args \nsubseteq Ar \setminus Args_{prev}$ where $Args_{prev}$ is a preventing set w.r.t. $LabArg_{pref}$. Thus, $\exists A \in Args$ such that $A \notin Ar \setminus Args_{prev}$, and consequently $A \in Args_{prev}$. Furthermore, let $Args'_{prev}$ be another preventing set and assume that $\nexists B \in Args$ such that $B \in Args'_{prev}$, so $Args \subseteq Ar \setminus Args'_{prev}$. Then by Theorem 6.9, $LabArg^{\circledast}$ is not a stable labelling of $\mathcal{AA}^{\circledast}$. Contradiction, so for all $Args'_{prev}$ there exists $B \in Args$ such that $B \in Args'_{prev}$. Let $Args'$ be the set of all such $B \in Args'_{prev}$. By Theorem 6.11, $Args_{enf} \subseteq Args'$, where $Args_{enf}$ is an enforcement set, and since $Args' \subseteq Args$, it follows that $Args_{enf} \subseteq Args$. □

Example 6.14. Consider $\mathcal{AA}_5$ and its only preferred labelling $LabArg_{pref}$, illustrated on the left of Figure 6.9. Let $LabArg$ be the labelling illustrated on the right of Figure 6.9 and let $Args = \{d, g\}$. Then $\mathcal{AA}_5^{\circledast}$ on the left of Figure 6.10 is a revision of $\mathcal{AA}_5$ w.r.t. $Args$ by $LabArg$, where the labelling $LabArg^{\circledast}$ on the left of Figure 6.10 is a revision labelling of $\mathcal{AA}_5^{\circledast}$. We note that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. As stated by Theorem 6.12, $Args$ is a superset of some enforcement set, in fact, it is a superset of both enforcement set $\{d\}$ and enforcement set $\{g\}$.

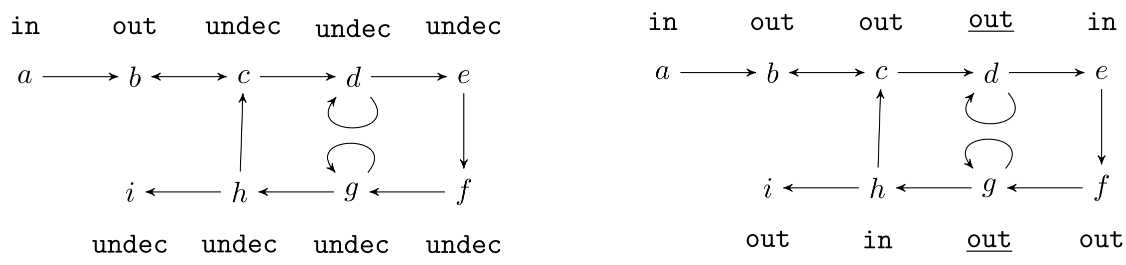


Figure 6.9: $\mathcal{AA}_5$ with its only preferred labelling $LabArg_{pref}$ (left) and with a labelling $LabArg$ that is more committed than $LabArg_{pref}$, where arguments d and g are illegally labelled (right).

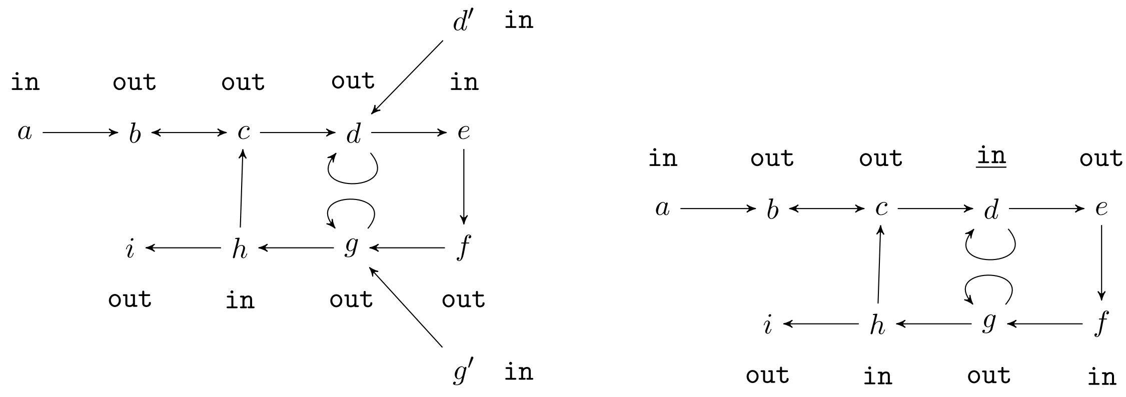


Figure 6.10: Left – A revision $\mathcal{AA}_5^{\circledast}$ of $\mathcal{AA}_5$ and a revision labelling $LabArg^{\circledast}$ (see Example 6.14). Right – The only enforcement labelling of the enforcement set $\{d\}$ of $\mathcal{AA}_5$ w.r.t. $LabArg_{pref}$.

Note that even if a set of arguments used to revise an AA framework is a superset of an enforcement set, the labelling used for the revision may be different from all enforcement labellings of the enforcement set. For example, $LabArg$ from Example 6.14 (see right of Figure 6.9) is used for the revision of $\mathcal{AA}_5$ w.r.t. $Args$, but it is not an enforcement labelling of either of the two enforcement sets that are subsets of $Args$. For instance, the only enforcement labelling of the enforcement set $\{d\}$ is illustrated on the right of Figure 6.10.

The next Corollary follows directly from Theorem 6.12 and states that the converse of Theorem 6.6 holds.

Corollary 6.13. *Let $Args \subseteq Ar$ and let $LabArg$ be a labelling of $\mathcal{AA}$ such that $LabArg_{pref} \sqsubset LabArg$,* $\texttt{undec}(LabArg) = \emptyset$*, and for all revisions $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and all revision labellings $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ it holds that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. Then there exists an enforcement set $Args_{enf}$ w.r.t. $LabArg_{pref}$ such that $Args_{enf} \subseteq Args$.*

Theorem 6.14 proves that the converse of Theorem 6.9 holds. That is, if no revision w.r.t. a set of arguments $Args$ is such that some revision labelling is a stable labelling of the revision, then there exists a preventing set that is disjoint from $Args$. In other words, preventing sets define a *necessary* condition for the non-existence of a stable labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$.

Theorem 6.14. *Let $\mathit{Args} \subseteq \mathit{Ar}$ be such that for all labellings LabArg of $\mathcal{AA}$ with $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\mathtt{undec}(\mathit{LabArg}) = \emptyset$ there exists no revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by LabArg such that some revision labelling $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. Then there exists a preventing set $\mathit{Args}_{\mathit{prev}}$ w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ such that $\mathit{Args} \subseteq \mathit{Ar} \setminus \mathit{Args}_{\mathit{prev}}$.*

Proof. Let $\mathit{Args} \subseteq \mathit{Ar}$. By (the contrapositive of) Corollary 6.7 it holds that: if for all labellings LabArg of $\mathcal{AA}$ such that $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\mathtt{undec}(\mathit{LabArg}) = \emptyset$, there exists no revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by LabArg such that some revision labelling $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$, then $\mathit{Args} \not\supseteq \mathit{Args}_{\mathit{enf}}$ where $\mathit{Args}_{\mathit{enf}}$ is an enforcement set. Thus, $\exists A \in \mathit{Args}_{\mathit{enf}}$ such that $A \notin \mathit{Args}$. Furthermore, assume that for some other enforcement set $\mathit{Args}'_{\mathit{enf}}$ it holds that $\mathit{Args} \supseteq \mathit{Args}'_{\mathit{enf}}$. Then by Corollary 6.7, there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by the enforcement labelling LabArg' of $\mathit{Args}'_{\mathit{enf}}$ such that some revision labelling $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. Contradiction, so for all enforcement sets $\mathit{Args}'_{\mathit{enf}}$, it holds that $\exists A \in \mathit{Args}'_{\mathit{enf}}$ such that $A \notin \mathit{Args}$. Let Args' be the set of all such arguments $A \in \mathit{Args}'_{\mathit{enf}}$ which are not in Args. By Theorem 6.10, $\mathit{Args}' \supseteq \mathit{Args}_{\mathit{prev}}$ where $\mathit{Args}_{\mathit{prev}}$ is a preventing set. Clearly, $\mathit{Args} \subseteq \mathit{Ar} \setminus \mathit{Args}'$, so $\mathit{Args} \subseteq \mathit{Ar} \setminus \mathit{Args}_{\mathit{prev}}$ where $\mathit{Args}_{\mathit{prev}}$ is a preventing set. □

Example 6.15. Consider again $\mathcal{AA}_4 = \langle \mathit{Ar}_4, \mathit{Att}_4 \rangle$ illustrated on the left of Figure 6.8 and the set of arguments $\mathit{Args} = \{c, d\}$. Then for any `in-out` labelling LabArg of $\mathcal{AA}_4$ that is more committed than $\mathit{LabArg}_{\mathit{pref}}$, there exists no revision $\mathcal{AA}_4^{\circledast}$ of $\mathcal{AA}_4$ w.r.t. Args by LabArg such that a revision labelling of $\mathcal{AA}_4^{\circledast}$ is a stable labelling of $\mathcal{AA}_4^{\circledast}$, since any revision labelling will illegally label e (as no attacks can be added to or deleted from e). As stated by Theorem 6.14, it holds that for the preventing set $\{e\}$, $\mathit{Args} \subset \mathit{Ar}_4 \setminus \{e\}$.

Theorems 6.6 and 6.12 as well as Theorems 6.9 and 6.14 show that enforcement and preventing sets indeed characterise sets of arguments that are *responsible* that a preferred labelling is not a stable labelling. Enforcement sets are responsible since they are minimal sets of arguments that all need to be revised in order to obtain a stable labelling, whereas preventing sets are responsible because if no argument from the set is revised, no stable labelling exists.

6.5 Structural Characterisations

Determining responsible sets of arguments according to the declarative labelling-based characterisations from Section 6.4 involves guessing sets of arguments and checking if they satisfy the respective definition by changing `undec` labels to `in` or `out` labels in the preferred labelling. In this section, we instead characterise sets of arguments as responsible that a preferred labelling is not a stable labelling based on the *structure* of the AA framework. We thereby aim at characterisations that allow to *constructively* determine responsible sets of arguments.

6.5.1 Odd-Length Cycles

Our first structural characterisation is inspired by the work of Dung [Dun95b], who proves that if an AA framework has no odd-length cycles, then a stable extension, and thus a stable labelling, exists. Consequently, the non-existence of stable labellings implies the existence of an odd-length cycle.

Building upon this result, we define odd-length cycles of arguments labelled `undec` by $LabArg_{pref}$ as responsible that $LabArg_{pref}$ is not a stable labelling. The reason to exclude odd-length cycles of arguments labelled `in` or `out` is that such cycles do not violate the definition of a stable labelling.

Definition 6.6 (Structural Characterisation 1)**.** $\mathscr{C}$ is a *responsible cycle* w.r.t. $LabArg_{pref}$ if and only if $\mathscr{C}$ is an odd-length cycle of $\mathcal{AA}$ and for all $A \in \mathscr{C}$ it holds that $A \in \texttt{undec}(LabArg_{pref})$.

Example 6.16. Let $\mathcal{AA}_6$ be the AA framework illustrated in Figure 6.11 and $LabArg_{pref}$ its only preferred labelling also depicted in the figure. $\mathcal{AA}_6$ has three odd-length cycles, but only one of them is a responsible cycle w.r.t. $LabArg_{pref}$, namely $\mathscr{C} = \{c\}$.

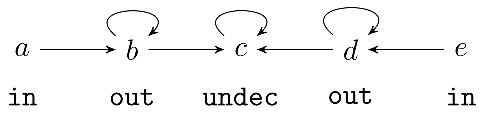


Figure 6.11: $\mathcal{AA}_6$ and its only preferred labelling (see Example 6.16).

As for our labelling-based characterisations, we prove that at least one responsible cycle w.r.t. $LabArg_{pref}$ exists, showing that responsible cycles are well-defined characterisations of parts of an AA framework responsible for $LabArg_{pref}$ not being a stable labelling.

Proposition 6.15. *There exists a responsible cycle w.r.t. $LabArg_{pref}$.*

Proof. Assume there exists no odd-length cycle of arguments labelled `undec` by $LabArg_{pref}$. Then $\mathcal{AA}_u = \mathcal{AA}{\downarrow_{\texttt{undec}(LabArg_{pref})}}$ comprises no odd-length cycle. By Corollary 36 in [Dun95b], $\mathcal{AA}_u$ has a stable labelling $LabArg_u$. We observe that for all arguments $A \in \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})$ which are attacking some argument in $\texttt{undec}(LabArg_{pref})$ it holds that $A \in \texttt{out}(LabArg_{pref})$ and that for all arguments $B \in \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})$ which are attacked by some argument in $\texttt{undec}(LabArg_{pref})$ it holds that $B \in \texttt{out}(LabArg_{pref})$. Let $LabArg = LabArg_{pref}{\downarrow_{\texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})}} \cup LabArg_u$, so $\texttt{undec}(LabArg) = \emptyset$ and $LabArg_{pref} \sqsubset LabArg$. We show that $LabArg$ is a complete labelling of $\mathcal{AA}$:

- Let $A \in \texttt{in}(\mathit{LabArg})$. If $A \in \texttt{in}(\mathit{LabArg}_{\mathit{pref}})$, then by Lemma A.2 in Appendix A A is legally labelled by *LabArg*. If $A \in \texttt{in}(\mathit{LabArg}_u)$, then for all attackers B of A such that $B \in \texttt{in}(\mathit{LabArg}_{\mathit{pref}}) \cup \texttt{out}(\mathit{LabArg}_{\mathit{pref}})$, $B \in \texttt{out}(\mathit{LabArg}_{\mathit{pref}})$ (by the above observation), and thus $B \in \texttt{out}(\mathit{LabArg})$. Furthermore, for all attackers C of A such that $C \in \texttt{undec}(\mathit{LabArg}_{\mathit{pref}})$, $C \in \texttt{out}(\mathit{LabArg}_u)$ since LabArg_u is a stable labelling of $\mathcal{AA}_u$, and thus $C \in \texttt{out}(\mathit{LabArg})$. Thus, A is legally labelled `in` by *LabArg*.

- Let $A \in \texttt{out}(\mathit{LabArg})$. If $A \in \texttt{out}(\mathit{LabArg}_{\mathit{pref}})$, then by Lemma A.2 in Appendix A A is legally labelled by *LabArg*. If $A \in \texttt{out}(\mathit{LabArg}_u)$, then there exists an attacker B of A such that $B \in \texttt{undec}(\mathit{LabArg}_{\mathit{pref}})$ and $B \in \texttt{in}(\mathit{LabArg}_u)$ since LabArg_u is a stable labelling of $\mathcal{AA}_u$, and thus $B \in \texttt{in}(\mathit{LabArg})$. Thus, A is legally labelled `out` by *LabArg*.

Thus, *LabArg* is a stable labelling of $\mathcal{AA}$. Contradiction. It follows that there exists an odd-length cycle of arguments all labelled `undec` by $\mathit{LabArg}_{\mathit{pref}}$. □

We are again interested how our characterisation of responsible arguments can be used to obtain a stable labelling. The following proposition states that a revision w.r.t. the set of all responsible cycles can yield a stable labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$ if the labelling used for the revision is chosen appropriately.

Proposition 6.16. *Let $S = \{A \in Ar \mid \mathscr{C}$ is a responsible cycle w.r.t. $\mathit{LabArg}_{\mathit{pref}}, A \in \mathscr{C}\}$. Then there exists a labelling LabArg of $\mathcal{AA}$ with $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\texttt{undec}(\mathit{LabArg}) = \emptyset$ such that for all revisions $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. S by LabArg and all revision labellings $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$, $\mathit{LabArg}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

Proof. Since $\mathcal{AA}{\downarrow}_{\texttt{undec}(\mathit{LabArg}_{\mathit{pref}})\setminus S}$ comprises no odd-length cycles, by Corollary 36 in [Dun95b] it has a stable labelling $\mathit{LabArg}_{\mathit{stable}}$. Let $\mathit{LabArg}' = \mathit{LabArg}_{\mathit{stable}} \cup \mathit{LabArg}_o$ be a labelling of $\mathcal{AA}{\downarrow}_{\texttt{undec}(\mathit{LabArg}_{\mathit{pref}})}$ where LabArg_o is a labelling of arguments in $\mathcal{AA}{\downarrow}_S$ such that $\texttt{out}(\mathit{LabArg}_o) = S$, and let $\mathit{LabArg} = \mathit{LabArg}' \cup \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\texttt{in}(\mathit{LabArg}_{\mathit{pref}}) \cup \texttt{out}(\mathit{LabArg}_{\mathit{pref}})}$. Clearly *LabArg* is a labelling of $\mathcal{AA}$ such that $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\texttt{undec}(\mathit{LabArg}) = \emptyset$.

- Let $A \in \texttt{in}(\mathit{LabArg}_{\mathit{pref}}) \cup \texttt{out}(\mathit{LabArg}_{\mathit{pref}})$. By Lemma A.2 in Appendix A, A is legally labelled by *LabArg*.

- Let $A \in \texttt{undec}(\mathit{LabArg}_{\mathit{pref}}) \setminus S$ and $\mathit{LabArg}(A) = \texttt{in}$. Then for all attackers B of A such that $B \in \texttt{undec}(\mathit{LabArg}_{\mathit{pref}}) \setminus S$, $\mathit{LabArg}_{\mathit{stable}}(B) = \texttt{out}$ and thus $\mathit{LabArg}(B) = \texttt{out}$. Furthermore, for all attackers C of A such that $C \in S$, $\mathit{LabArg}_o(C) = \texttt{out}$ and thus $\mathit{LabArg}(C) = \texttt{out}$. Additionally, for all attackers D of A such that $D \in \texttt{in}(\mathit{LabArg}_{\mathit{pref}}) \cup \texttt{out}(\mathit{LabArg}_{\mathit{pref}})$, $\mathit{LabArg}_{\mathit{pref}}(D) = \texttt{out}$ and thus $\mathit{LabArg}(D) = \texttt{out}$. Hence, A is legally labelled `in` by *LabArg*.

- Let $A \in \texttt{undec}(\mathit{LabArg}_{pref}) \setminus S$ and $\mathit{LabArg}(A) = \texttt{out}$. Then there exists an attacker B of A such that $B \in \texttt{undec}(\mathit{LabArg}_{pref}) \setminus S$ and $\mathit{LabArg}_{stable}(B) = \texttt{in}$ and thus $\mathit{LabArg}(B) = \texttt{in}$. Hence, A is legally labelled $\texttt{out}$ by LabArg.

Thus, all $A \in Ar \setminus S$ are legally labelled by LabArg. Let $\mathcal{AA}^{\circledast}$ be a revision of $\mathcal{AA}$ w.r.t. S by LabArg and $\mathit{LabArg}^{\circledast}$ a revision labelling of $\mathcal{AA}^{\circledast}$. By Definition 6.1, all $A \in S$ and all $B \in Ar^{\circledast} \setminus Ar$ are legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Furthermore, by Lemma A.1 in Appendix A, all $A \in Ar \setminus S$ are legally labelled by $\mathit{LabArg}^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Therefore, $\mathit{LabArg}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. □

Example 6.17. Consider again $\mathcal{AA}_5$ illustrated on the left of Figure 6.9. The set of arguments occurring in responsible cycles w.r.t. LabArg_{pref} is $S = \{d, g\}$. Consider the labelling LabArg depicted on the right of Figure 6.9, which is more committed than LabArg_{pref} and labels no arguments as $\texttt{undec}$. A revision $\mathcal{AA}_5^{\circledast}$ of $\mathcal{AA}_5$ w.r.t. S by LabArg is shown on the left of Figure 6.10, along with a revision labelling that is a stable labelling of $\mathcal{AA}_5^{\circledast}$.

Since by Lemma 6.1 a revision exists w.r.t. any set of arguments and any labelling, it follows that there exists a revision w.r.t. responsible cycles which has a stable labelling that is more committed than the preferred labelling.

Corollary 6.17. *Let $S = \{A \in Ar \mid \mathscr{C}$ is a responsible cycle w.r.t. $\mathit{LabArg}_{pref}, A \in \mathscr{C}\}$. Then there exists a labelling LabArg of $\mathcal{AA}$ with $\mathit{LabArg}_{pref} \sqsubset \mathit{LabArg}$ and $\texttt{undec}(\mathit{LabArg}) = \emptyset$, and there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. S by LabArg and a revision labelling $\mathit{LabArg}^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $\mathit{LabArg}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.*

6.5.2 Strongly Connected Components

Our second structural characterisation is based upon a result on the composition of stable labellings, namely that stable labellings can be computed along the SCCs [BGG05] of the AA framework. That is, the stable labellings of initial SCCs are computed, and then the stable labellings of the following SCCs are iteratively determined taking the labels of arguments in their parent SCCs into account. It follows that if the AA framework has no stable labelling, some SCC in this iterative computation has no stable labelling (when taking the labels in parent SCCs into account).

Our second structural characterisation of sets of arguments responsible for the non-existence of stable labellings refines this observation. It defines the "first" SCCs that have no stable labelling in the iterative computation of a stable labelling of the whole AA framework as responsible. More precisely, responsible sets are SCCs satisfying that: 1) the SCC has no stable labelling w.r.t. the input from its parent SCCs, i.e. w.r.t. the labels of attackers in parent SCCs according to LabArg_{pref}; and 2) all parent SCCs have a stable labelling w.r.t. the input from their parent SCCs that coincides with the labels assigned by LabArg_{pref}.

Definition 6.7 (Structural Characterisation 2)**.** $Args \subseteq Ar$ is a *responsible SCC* w.r.t. $LabArg_{pref}$ if and only if $Args$ is an SCC of $\mathcal{AA}$ such that

1. there exists no stable labelling w.r.t. $(\mathcal{AA}{\downarrow}_{Args}, parents(Args), LabArg_{pref}{\downarrow}_{parents(Args)}, Att \cap (parents(Args) \times Args))$ that is more or equally committed than $LabArg_{pref}{\downarrow}_{Args}$, and

2. for all $Args' \in parentSCCs(Args)$, $LabArg_{pref}{\downarrow}_{Args'}$ is a stable labelling w.r.t. $(\mathcal{AA}{\downarrow}_{Args'}, parents(Args'), LabArg_{pref}{\downarrow}_{parents(Args')}, Att \cap (parents(Args') \times Args'))$.

Example 6.18. The only responsible SCC of $\mathcal{AA}_{therapy}$ (see Figure 6.3) w.r.t. its only preferred labelling $LabArg_{pref}$ (see Example 6.6) is $\{A, B, C\}$. Since this is an initial SCC, it is trivially satisfied that its parent SCCs have a stable labelling.

The following example illustrates an AA framework where a responsible SCC is not an initial SCC of the AA framework.

Example 6.19. Consider again $\mathcal{AA}_5$ and its only preferred labelling $LabArg_{pref}$, illustrated on the left of Figure 6.9. The only responsible SCC w.r.t. $LabArg_{pref}$ is the SCC $\{b, c, d, e, f, g, h\}$ since: 1) there exists no stable labelling w.r.t. the AA framework with input $(\mathcal{AA}{\downarrow}_{\{b,c,d,e,f,g,h\}}, \{a\}, \{(a, \texttt{in})\}, \{(a, b)\})$, which is depicted in Figure 6.12; and 2) $\{b, c, d, e, f, g, h\}$ only has one parent SCC, namely $\{a\}$, and $LabArg_{pref}$ restricted to $\{a\}$, i.e $\{(a, \texttt{in})\}$, is a stable labelling w.r.t. $(\mathcal{AA}{\downarrow}_{\{a\}}, \emptyset, \emptyset, \emptyset)$.

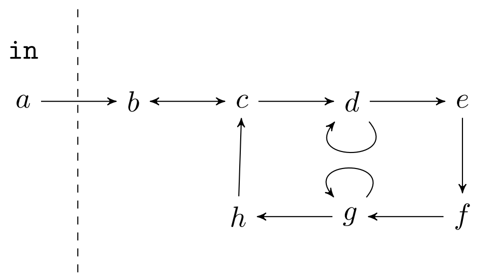

$$\begin{tikzcd}
\texttt{in} & & & & \\
a \arrow[r] & b \arrow[r, leftrightarrow] & c \arrow[r] & d \arrow[r] \arrow[d, bend left] & e \arrow[d] \\
& & h \arrow[u] & g \arrow[l] \arrow[u, bend left] & f \arrow[l]
\end{tikzcd}$$

Figure 6.12: The AA framework with input made of the SCC $\{b, c, d, e, f, g, h\}$ of $\mathcal{AA}_5$ (right of dashed line) and the input arguments from its parent SCCs (left of dashed line) with the input labelling (given by the preferred labelling of $\mathcal{AA}_5$).

Note that Definition 6.7 does not require a responsible SCC to not have a stable labelling at all (w.r.t. its parent SCCs), but rather that it has no stable labelling that is more committed than the labels assigned to the SCC by $LabArg_{pref}$. This is because we aim to define responsibility for the non-existence of stable labellings in terms of responsibility for $LabArg_{pref}$ not being a stable labelling. Therefore, Definition 6.7 characterises the "first" SCCs in which the labels assigned by the preferred labelling do not form a stable labelling of the SCC.

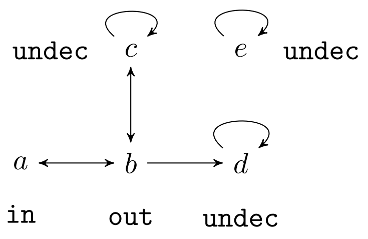


Figure 6.13: $\mathcal{AA}_7$ and a preferred labelling $LabArg_{pref}$.

Example 6.20. Let $\mathcal{AA}_7$ be the AA framework in Figure 6.13, which has no stable labellings, and consider the depicted preferred labelling $LabArg_{pref}$. $\mathcal{AA}_7$ has three SCCs, namely $\{a, b, c\}$, $\{d\}$, and $\{e\}$. The SCC $\{a, b, c\}$ *has* a stable labelling w.r.t. $(\mathcal{AA}{\downarrow}_{\{a,b,c\}}, \emptyset, \emptyset, \emptyset)$, namely $\{(a, \texttt{out}), (b, \texttt{in}), (c, \texttt{out})\}$, but this stable labelling is not more or equally committed than $LabArg_{pref}{\downarrow}_{\{a,b,c\}} = \{(a, \texttt{in}), (b, \texttt{out}), (c, \texttt{undec})\}$ (a would have to be labelled `out` and b would have to be labelled `in` in $LabArg_{pref}{\downarrow}_{\{a,b,c\}}$). This illustrates the importance of the comparison with $LabArg_{pref}$ in the first condition of Definition 6.7: due to the comparison, $\{a, b, c\}$ satisfies the condition; without the comparison, $\{a, b, c\}$ would not satisfy the condition. Thus, without the comparison $\{a, b, c\}$ would not be identified as a responsible SCC. However, $\{a, b, c\}$ should be identified as responsible since it is the "first" SCC that provides a reason why $LabArg_{pref}$ is not a stable labelling.
Of similar importance is the comparison with $LabArg_{pref}$ in the second condition of Definition 6.7. Consider the SCC $\{d\}$ and its parent SCC $\{a, b, c\}$. $\{a, b, c\}$ *has* a stable labelling w.r.t. $(\mathcal{AA}{\downarrow}_{\{a,b,c\}}, \emptyset, \emptyset, \emptyset)$, namely $\{(a, \texttt{out}), (b, \texttt{in}), (c, \texttt{out})\}$, so without the comparison with $LabArg_{pref}$, the SCC $\{d\}$ would be identified as a responsible SCC. However, since the stable labelling w.r.t. $(\mathcal{AA}{\downarrow}_{\{a,b,c\}}, \emptyset, \emptyset, \emptyset)$ does not coincide with $LabArg_{pref}{\downarrow}_{\{a,b,c\}}$, $\{d\}$ is not a responsible SCC.

As for previous characterisations of sets of arguments responsible for the non-existence of stable labellings, we prove that at least one responsible SCC exists w.r.t. $LabArg_{pref}$.

Proposition 6.18. *There exists a responsible SCC w.r.t. $LabArg_{pref}$.*

Proof. Since the attacks between SCCs are by definition unidirectional, there exists a sequence of SCCs $Args_1, \ldots, Args_n$ ($\forall i \neq k : Args_i \neq Args_k$) such that if $Args_i$ is attacked by $Args_k$ ($i \neq k$), then $k < i$. By Corollary A.6 in Appendix A, $LabArg_{pref} = LabArg_1 \cup \ldots \cup LabArg_n$ where $LabArg_i$ is a labelling of $Args_i$, $LabArg_1$ is a complete labelling of $Args_1$, and for all $j \in \{2 \ldots n\}$ it holds that $LabArg_j$ is compatible with $LabArg_1 \cup \ldots \cup LabArg_{j-1}$. If $LabArg_1$ is not a stable labelling of $Args_1$, then $Args_1$ satisfies Definition 6.7, so there exists a responsible SCC w.r.t. $LabArg_{pref}$. Else, there exists $LabArg_i$ such that for all $LabArg_j$ with $j < i$ it holds that $\texttt{undec}(LabArg_j) = \emptyset$ and $\texttt{undec}(LabArg_i) \neq \emptyset$. Since by the construction of our sequence of SCCs, for all $Args' \in parentSCCs(Args_i)$ it holds that $Args' = Args_j$ for some $j < i$, it follows that for all these $Args'$, $LabArg_{pref}{\downarrow}_{Args'}$ is a stable labelling w.r.t.

$(\mathcal{AA}{\downarrow}_{Args'}, parents(Args'), LabArg_{pref}{\downarrow}_{parents(Args')}, Att \cap (parents(Args') \times Args'))$.
Furthermore, since $\texttt{undec}(LabArg_i) \neq \emptyset$, it follows that there exists no stable labelling w.r.t.
$(\mathcal{AA}{\downarrow}_{Args_i}, parents(Args_i), LabArg_{pref}{\downarrow}_{parents(Args_i)}, Att \cap (parents(Args_i) \times Args_i))$
that is more committed than $LabArg_i$. (If there was such a labelling, then $LabArg_{pref}$ would not be a preferred labelling.) □

Differently from our previous characterisations, we do not investigate how to use responsible SCCs to obtain a stable labelling, since our next structural characterisation refines responsible SCCs. We will then study how to obtain a stable labelling using our refined characterisation.

6.5.3 Strongly Connected `undec` Parts (SCUPs)

Our characterisation of responsible SCCs relies on the decomposability of stable labellings with regards to the SCCs of an AA framework. In this section, we refine this notion by using another decomposability result. Baroni et al. [BBC+14] show that the complete labellings of an AA framework can be obtained by splitting the AA framework into *any* partition and then determining complete labellings of the different parts in such a way that they are compatible. We can thus think of $LabArg_{pref}$ as a combination of two compatible labellings: a labelling of the part of the AA framework whose arguments are labelled `in` or `out` by $LabArg_{pref}$, and a labelling of the part of the AA framework whose arguments are labelled `undec` by $LabArg_{pref}$. We call these two parts the `in`/`out`-part and the `undec`-part, respectively.

The fact that all arguments in the `undec`-part are labelled `undec` by $LabArg_{pref}$ implies that this is the only labelling that is compatible with the `in` and `out` labels in the `in`/`out`-part (if there was another labelling, the preferred labelling would not be maximal). Proposition 6.19 proves that, furthermore, labelling all arguments in the `undec`-part as `undec` is the *only complete labelling* of this part on its own (disregarding the `in`/`out`-part). In other words, the labels of arguments in the `in`/`out`-part are not responsible that all arguments in the `undec`-part are labelled `undec`. Rather, the structure of the `undec`-part itself is responsible that the arguments cannot be legally labelled `in` or `out`.

Proposition 6.19. *The only complete labelling of* $\mathcal{AA}{\downarrow}_{\texttt{undec}(LabArg_{pref})}$ *labels all arguments as* `undec`.

Proof. Let $ArgsIO = \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})$ and $ArgsU = \texttt{undec}(LabArg_{pref})$. We observe that since arguments labelled `undec` are not attacked by arguments labelled `in` by a complete labelling, $\forall B \in ArgsIO$ attacking some $A \in ArgsU$, it holds that $B \in \texttt{out}(LabArg_{pref}{\downarrow}_{ArgsIO})$.
We first prove that $LabArg_{pref}{\downarrow}_{ArgsU}$ is a complete labelling of $\mathcal{AA}{\downarrow}_{ArgsU}$. Since by Lemma A.8 in Appendix A, $LabArg_{pref}{\downarrow}_{ArgsU}$ is a complete labelling w.r.t.
$(\mathcal{AA}{\downarrow}_{ArgsU}, ArgsIO, LabArg_{pref}{\downarrow}_{ArgsIO}, Att \cap (ArgsIO \times ArgsU))$, it follows that $\forall A \in$

$ArgsU$ and $\forall B \in ArgsU$ attacking A, $B \notin \texttt{in}(LabArg_{pref}\downarrow_{ArgsU})$, and $\exists C \in ArgsU$ attacking A such that $C \in \texttt{undec}(LabArg_{pref}\downarrow_{ArgsU})$ since by our above observation $\nexists D \in ArgsIO$ attacking A such that $D \in \texttt{undec}(LabArg_{pref}\downarrow_{ArgsIO})$. Thus, all $A \in ArgsU$ are legally labelled by $LabArg_{pref}\downarrow_{ArgsU}$, so $LabArg_{pref}\downarrow_{ArgsU}$ is a complete labelling of $\mathcal{AA}\downarrow_{ArgsU}$.
We now prove that there exists no other complete labelling of $\mathcal{AA}\downarrow_{ArgsU}$. Assume there exists a complete labelling $LabArgU$ of $\mathcal{AA}\downarrow_{ArgsU}$ such that $\texttt{undec}(LabArgU) \neq ArgsU$. Clearly, $LabArg_{pref} \sqsubset LabArg_{pref}\downarrow_{ArgsIO} \cup LabArgU$.

By Lemma A.10 in Appendix A, $LabArg_{pref}\downarrow_{ArgsIO}$ is compatible with $LabArgU$. Furthermore, $LabArgU$ is compatible with $LabArg_{pref}\downarrow_{ArgsIO}$:

- If $A \in \texttt{in}(LabArgU)$, then $\forall B \in ArgsU$ attacking A, $B \in \texttt{out}(LabArgU)$ since $LabArgU$ is a complete labelling of $\mathcal{AA}\downarrow_{ArgsU}$. Furthermore $\forall B \in ArgsIO$ attacking A, $B \in \texttt{out}(LabArg_{pref}\downarrow_{ArgsIO})$ as previously noted.
- If $A \in \texttt{out}(LabArgU)$, then $\exists B \in ArgsU$ attacking A such that $B \in \texttt{in}(LabArgU)$ since $LabArgU$ is a complete labelling of $\mathcal{AA}\downarrow_{ArgsU}$.
- If $A \in \texttt{undec}(LabArgU)$, then $\forall B \in ArgsU$ attacking A, $B \notin \texttt{in}(LabArgU)$, and $\exists B \in ArgsU$ attacking A such that $B \in \texttt{undec}(LabArgU)$ since $LabArgU$ is a complete labelling of $\mathcal{AA}\downarrow_{ArgsU}$. Furthermore, $\forall B \in ArgsIO$ attacking A, $B \notin \texttt{in}(LabArg_{pref}\downarrow_{ArgsIO})$ as previously noted.

It follows by Lemma A.4 in Appendix A, that $LabArg_{pref}\downarrow_{ArgsIO} \cup LabArgU$ is a complete labelling of $\mathcal{AA}$. Contradiction, since $LabArg_{pref} \sqsubset LabArg_{pref}\downarrow_{ArgsIO} \cup LabArgU$ and $LabArg_{pref}$ is a preferred labelling. □

Since the undec-part has only one complete labelling, which labels all arguments as undec, this labelling is also its only preferred labelling. Thus, the question as to why $LabArg_{pref}$ is not a stable labelling can be reduced to the question as to why the preferred labelling of the undec-part is not a stable labelling.

Applying our notion of responsible SCCs, we obtain that the preferred labelling of the undec-part is not a stable labelling because of its “first” SCCs that have no stable labelling. These “first” SCCs are the initial SCCs of the undec-part since *no* SCC in the undec-part has a stable labelling. This observation results in the following new characterisation of sets of arguments responsible for $LabArg_{pref}$ not being a stable labelling: a set of arguments is responsible if it is an initial SCC of the undec-part.

Definition 6.8 (Structural Characterisation 3). *$Args \subseteq Ar$ is a strongly connected* undec *part (SCUP)* w.r.t. $LabArg_{pref}$ if and only if $Args$ is an initial SCC of $\mathcal{AA}\downarrow_{\texttt{undec}(LabArg_{pref})}$.

Example 6.21. $\mathcal{AA}_{therapy}$ from Section 6.1 has only one SCUP w.r.t. its only preferred labelling (see Example 6.6), namely $\{A, B, C\}$.

Importantly, at least one SCUP exists w.r.t. $LabArg_{pref}$, which shows that SCUPs provide a well-defined characterisation of responsible sets of arguments.

Proposition 6.20. *There exists a SCUP w.r.t. $LabArg_{pref}$.*

Proof. Since every AA framework has an initial SCC, $\mathcal{AA}\downarrow_{\texttt{undec}(LabArg_{pref})}$ has an initial SCC, which by Definition 6.8 is a SCUP w.r.t. $LabArg_{pref}$. □

The following example illustrates that an AA framework may have various SCUPs w.r.t. a preferred labelling.

Example 6.22. Let $\mathcal{AA}_8$ and its only preferred labelling $LabArg_{pref}$ be as illustrated in Figure 6.14. There are two SCUPs w.r.t. $LabArg_{pref}$, namely $\{c\}$ and $\{d\}$.

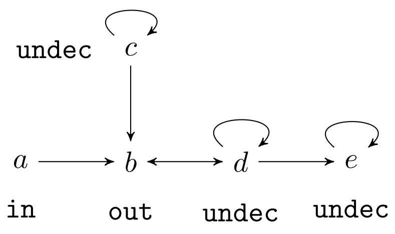


Figure 6.14: $\mathcal{AA}_8$ and its only preferred labelling $LabArg_{pref}$ (see Example 6.22).

We now prove that SCUPs are indeed refinements of responsible SCCs in the sense that every responsible SCC comprises a SCUP.

Proposition 6.21. *Let Args be a responsible SCC w.r.t. $LabArg_{pref}$. Then $\exists Args' \subseteq Args$ such that $Args'$ is a SCUP w.r.t. $LabArg_{pref}$.*

Proof. By Definition 6.7, there exists no stable labelling w.r.t.
$(\mathcal{AA}\downarrow_{Args}, parents(Args), LabArg_{pref}\downarrow_{parents(Args)}, Att \cap (parents(Args) \times Args))$ that is more or equally committed than $LabArg_{pref}\downarrow_{Args}$. Thus, $\texttt{undec}(LabArg_{pref}\downarrow_{Args}) \neq \emptyset$. Let $Args' = \texttt{undec}(LabArg_{pref}\downarrow_{Args})$. Since $Args$ is an SCC of $\mathcal{AA}$, $Args'$ is an SCC of $\mathcal{AA}\downarrow_{\texttt{undec}(LabArg_{pref})}$.
By Definition 6.7, for all $Args_p \in parentSCCs(Args)$ it holds that $LabArg_{pref}\downarrow_{Args_p}$ is a stable labelling w.r.t.
$(\mathcal{AA}\downarrow_{Args_p}, parents(Args_p), LabArg_{pref}\downarrow_{parents(Args_p)}, Att \cap (parents(Args_p) \times Args_p))$.
Thus, $\nexists A \in parents(Args), B \in Args$ such that A attacks B and $A \in \texttt{undec}(LabArg_{pref})$. Since $Args' \subseteq Args$ and since $Args \setminus Args' \subseteq \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})$, it follows that $\nexists A \in parents(Args'), B \in Args'$ such that A attacks B and $A \in \texttt{undec}(LabArg_{pref})$. Thus, in $\mathcal{AA}\downarrow_{\texttt{undec}(LabArg_{pref})}$ it holds that $Args'$ is an SCC and $Args'$ is not attacked by any arguments not contained in $Args'$. Thus, $Args'$ is an initial SCC of $\mathcal{AA}\downarrow_{\texttt{undec}(LabArg_{pref})}$, so it is a SCUP w.r.t. $LabArg_{pref}$. □

Example 6.23. Consider $\mathcal{AA}_5$ and its only preferred labelling $LabArg_{pref}$ illustrated on the left of Figure 6.9. By Example 6.19, the only responsible SCC w.r.t. $LabArg_{pref}$ is $\{b, c, d, e, f, g, h\}$. As expected, there exists a SCUP that is a subset of this responsible SCC, namely $\{c, d, e, f, g, h\}$, which is the only SCUP of $\mathcal{AA}_5$ w.r.t. $LabArg_{pref}$.

Note that the converse of Proposition 6.21 does not hold in general, i.e. it is not the case that every SCUP is a subset of some responsible SCC. For example, $\{d\}$ is a SCUP of $\mathcal{AA}_8$ w.r.t. $LabArg_{pref}$ (see Figure 6.14), but the SCC containing d, i.e. $\{b, d\}$, is not a responsible SCC, since the parent SCC $\{c\}$ has no stable labelling.

Even though SCUPs are defined based on the *structure* of the AA framework rather than based on labellings that are more committed than $LabArg_{pref}$ as our labelling-based characterisations, we prove that SCUPs constitute sets of arguments that cannot all be legally labelled `in` or `out`. More precisely, with respect to all `in`-`out` labellings that are more committed than $LabArg_{pref}$, at least one argument in every SCUP is illegally labelled.

Lemma 6.22. *Let $Args$ be a SCUP w.r.t. $LabArg_{pref}$. Then for all labellings $LabArg$ of $\mathcal{AA}$ with $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$ it holds that there exists $A \in Args$ such that A is illegally labelled by $LabArg$.*

Proof. Assume $\exists LabArg$ of $\mathcal{AA}$ with $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$ such that $\forall A \in Args$, A is legally labelled by $LabArg$ in $\mathcal{AA}$. Let $Args_1 = \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref}) \cup Args$, $Args_2 = Ar \setminus Args_1$, and $LabArg_1 = LabArg{\downarrow}_{Args_1}$. Since $Args$ is a SCUP, it holds that $\forall A \in Args$ and $\forall B$ attacking A, $B \in Args_1$. Thus, A being legally labelled by $LabArg$ only depends on $LabArg_1$. Let $LabArg_2$ be some labelling of $Args_2$. Then $\forall A \in Args$, A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}$. Furthermore, clearly $LabArg_{pref} \sqsubset LabArg_1 \cup LabArg_2$. Then by Lemma A.2 in Appendix A, $\forall A \in \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})$ it holds that A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}$. Thus, $\forall A \in Args_1$, A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}$. Then by Lemma A.7 in Appendix A, $LabArg_1$ is compatible with $LabArg_2$ (for any labelling $LabArg_2$ of $Args_2$). Furthermore, by Lemma A.9 in Appendix A, there exists a labelling $LabArg_2'$ that is compatible with $LabArg_1$. Then by Lemma A.4 in Appendix A, $LabArg_1 \cup LabArg_2'$ is a complete labelling of $\mathcal{AA}$. Contradiction since $LabArg_{pref} \sqsubset LabArg_1 \cup LabArg_2'$. $\square$

Since by Proposition 6.21 every responsible SCC comprises a SCUP, an analogous result to Lemma 6.22 also holds for responsible SCCs. That is, with respect to all `in`-`out` labellings that are more committed than $LabArg_{pref}$, at least one argument in every responsible SCC is illegally labelled.

6.5.4 Revising SCUPs

In this section, we investigate how SCUPs can be used to turn $LabArg_{pref}$ into a stable labelling. We first prove that, similarly to preventing sets, SCUPs provide a *sufficient* condition for "preventing" the existence of a stable labelling that is more committed than $LabArg_{pref}$. That is, any revision w.r.t. a set of arguments not containing any arguments from some SCUP has no stable labelling that is more committed than $LabArg_{pref}$.

Theorem 6.23. *Let $Args \subseteq Ar \setminus Args_{SCUP}$ where $Args_{SCUP}$ is a SCUP w.r.t. $LabArg_{pref}$ and let $LabArg$ be a labelling such that $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$.*

Then there exists no revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. Args by LabArg such that some revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$.

Proof. Assume there exists a revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and a revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. By Lemma 6.22, $\exists A \in Args_{SCUP}$ such that A is illegally labelled by $LabArg$ in $\mathcal{AA}$. Since $A \in Ar \setminus Args$, by Lemma A.1 in Appendix A, A is illegally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Contradiction. □

Example 6.24. Consider again the SCUPs of $\mathcal{AA}_8$ w.r.t. its only preferred labelling $LabArg_{pref}$ (see Example 6.22). Let $LabArg$ be the `in-out` labelling illustrated in Figure 6.15, which is more committed than $LabArg_{pref}$. The set $\{a, b, d, e\}$ does not contain any argument from the SCUP $\{c\}$. It is easy to see that there exists no revision $\mathcal{AA}_8^{\circledast}$ of $\mathcal{AA}_8$ w.r.t. $\{a, b, d, e\}$ by $LabArg$ such that a revision labelling is a stable labelling of $\mathcal{AA}_8^{\circledast}$ since c will always be illegally labelled `out`.

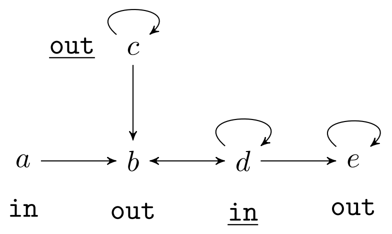


Figure 6.15: $\mathcal{AA}_8$ and a labelling $LabArg$ (see Example 6.24), where illegal labels are underlined.

Therefore, if we are to obtain a stable labelling, a revision has to involve arguments from every SCUP. In what follows, we thus investigate if revising *all* SCUPs yields a stable labelling. For this purpose, we define a *SCUP revision* as a revision w.r.t. the set of all arguments in all SCUPs by a labelling that is more committed than $LabArg_{pref}$ and labels all arguments in all SCUPs as `in` or `out`.

Notation 6.9. Let $Args_1, \ldots, Args_n$ be all SCUPs w.r.t. $LabArg_{pref}$. $\mathcal{SCUPS} = Args_1 \cup \ldots \cup Args_n$ denotes the set of all arguments in SCUPs.

Definition 6.10 (SCUP Revision and SCUP Revision Labelling)**.** Let $LabArg_{\mathcal{SCUPS}}$ be a labelling of $\mathcal{AA}{\downarrow}_{\mathcal{SCUPS}}$ with `undec`$(LabArg_{\mathcal{SCUPS}}) = \emptyset$ and let $LabArg = LabArg_{\mathcal{SCUPS}} \cup LabArg_{pref}{\downarrow}_{Ar\setminus\mathcal{SCUPS}}$. $\mathcal{AA}^{\circledast}$ is a *SCUP revision* of $\mathcal{AA}$ if and only if $\mathcal{AA}^{\circledast}$ is a revision of $\mathcal{AA}$ w.r.t. $\mathcal{SCUPS}$ by $LabArg$. A revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ is called a *SCUP revision labelling* of $\mathcal{AA}^{\circledast}$.

Example 6.25. Consider again $\mathcal{AA}_8$ from Example 6.22 (see Figure 6.14). A SCUP revision of $\mathcal{AA}_8$ along with a SCUP revision labelling is depicted on the left of Figure 6.16. The labelling of arguments in $\mathcal{SCUPS}$ used for the SCUP revision is $LabArg_{\mathcal{SCUPS}} = \{(c, \texttt{out}), (d, \texttt{in})\}$.

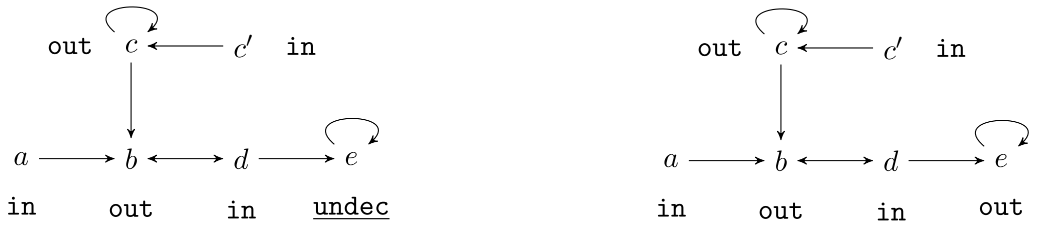


Figure 6.16: Left – $\mathcal{AA}_8^{\circledast}$ and a SCUP revision labelling $LabArg^{\circledast}$ (see Example 6.25), where illegal labels are underlined. Right – $\mathcal{AA}_8^{\circledast}$ and a preferred labelling that is more committed than $LabArg^{\circledast}$ (see Example 6.26).

Since by Lemma 6.1, a revision exists w.r.t. any set of arguments and labelling and since by Proposition 6.20 there exists a SCUP w.r.t. the preferred labelling, a SCUP revision exists.

Corollary 6.24. *There exists a SCUP revision $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$.*

The SCUP revision from Example 6.25 illustrates that a SCUP revision labelling may not be a complete labelling of the SCUP revision (see the left of Figure 6.16). We prove that, nevertheless, there exists a *preferred labelling* of the SCUP revision that is more or equally committed than the SCUP revision labelling.

Theorem 6.25. *Let $\mathcal{AA}^{\circledast}$ be a SCUP revision of $\mathcal{AA}$ and $LabArg^{\circledast}$ a SCUP revision labelling of $\mathcal{AA}^{\circledast}$. Then there exists a preferred labelling $LabArg^{\circledast}_{pref}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg^{\circledast} \sqsubseteq LabArg^{\circledast}_{pref}$.*

Proof. Let $\mathcal{SCUPS}^{\circledast} = \{A \in Ar^{\circledast} \mid A \in \mathcal{SCUPS} \vee A \notin Ar\}$. Let $Args_1 = \mathtt{in}(LabArg_{pref}) \cup \mathtt{out}(LabArg_{pref}) \cup \mathcal{SCUPS}^{\circledast}$, $Args_2 = Ar^{\circledast} \setminus Args_1$, and $LabArg_1 = LabArg^{\circledast}\downarrow_{Args_1}$.
By Definitions 6.10 and 6.1 it holds that $\forall A \in \mathcal{SCUPS}^{\circledast}$, A is legally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Since $\mathcal{SCUPS}$ consists of arguments in SCUPs, it holds that $\forall A \in \mathcal{SCUPS}^{\circledast}$ and $\forall B$ attacking A in $\mathcal{AA}^{\circledast}$, $B \in Args_1$. Thus, A being legally labelled by $LabArg^{\circledast}$ only depends on $LabArg_1$. Let $LabArg_2$ be some labelling of $Args_2$. Then $\forall A \in \mathcal{SCUPS}^{\circledast}$, A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}^{\circledast}$. Note that for any $LabArg_2$ of $Args_2$ it holds that $LabArg^{\circledast}\downarrow_{Args_2} \sqsubseteq LabArg_2$ since $\mathtt{undec}(LabArg^{\circledast}\downarrow_{Args_2}) = Args_2$, because $Args_2 \subseteq \mathtt{undec}(LabArg_{pref}) \setminus \mathcal{SCUPS}$. Then $LabArg^{\circledast} \sqsubseteq LabArg_1 \cup LabArg_2$.
Let $LabArg = LabArg_{\mathcal{SCUPS}} \cup LabArg_{pref}\downarrow_{Ar \setminus \mathcal{SCUPS}}$ be the labelling used for the SCUP revision. By Lemma A.2 in Appendix A, $\forall A \in \mathtt{in}(LabArg_{pref}) \cup \mathtt{out}(LabArg_{pref})$ it holds that A is legally labelled by $LabArg$ in $\mathcal{AA}$ since $LabArg_{pref} \sqsubset LabArg$. Then by Lemma A.1 in Appendix A, $\forall A \in \mathtt{in}(LabArg_{pref}) \cup \mathtt{out}(LabArg_{pref})$ it holds that A is legally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$. Since $LabArg^{\circledast} \sqsubseteq LabArg_1 \cup LabArg_2$, by Lemma A.2 in Appendix A it holds that $\forall A \in \mathtt{in}(LabArg_{pref}) \cup \mathtt{out}(LabArg_{pref})$, A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}^{\circledast}$.
Thus, $\forall A \in Args_1$, A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}^{\circledast}$. Then by Lemma A.7 in Appendix A, $LabArg_1$ is compatible with $LabArg_2$ (for any labelling

$LabArg_2$ of $Args_2$). Furthermore by Lemma A.9 in Appendix A, there exists a labelling $LabArg_2'$ that is compatible with $LabArg_1$. Then by Lemma A.4 in Appendix A, $LabArg_1 \cup LabArg_2'$ is a complete labelling of $\mathcal{AA}^{\circledast}$. Then either $LabArg_1 \cup LabArg_2'$ is a preferred labelling of $\mathcal{AA}^{\circledast}$ or there exists a preferred labelling $LabArg^{\circledast\prime}$ such that $LabArg_1 \cup LabArg_2' \sqsubset LabArg^{\circledast\prime}$ and thus $LabArg^{\circledast} \sqsubset LabArg^{\circledast\prime}$. □

Example 6.26. Given the SCUP revision $\mathcal{AA}_8^{\circledast}$ and the SCUP revision labelling $LabArg^{\circledast}$ from Example 6.25 (see left of Figure 6.16), there exists a preferred labelling of $\mathcal{AA}_8^{\circledast}$ that is more committed than $LabArg^{\circledast}$, as illustrated on the right of Figure 6.16.

Since a SCUP revision labelling is more committed than $LabArg_{pref}$ (because all arguments in SCUPs are labelled `in` or `out` by the SCUP revision labelling, but are labelled `undec` by $LabArg_{pref}$), it follows that there exists a preferred labelling of the SCUP revision that is more committed than $LabArg_{pref}$.

Corollary 6.26. *Let $\mathcal{AA}^{\circledast}$ be a SCUP revision of $\mathcal{AA}$. Then there exists a preferred labelling $LabArg^{\circledast}_{pref}$ of $\mathcal{AA}^{\circledast}$ such that $LabArg_{pref} \sqsubset LabArg^{\circledast}_{pref}$.*

In Example 6.26, there exists a preferred labelling of the SCUP revision that is more committed than the SCUP revision labelling and that is also a *stable* labelling of the SCUP revision. However, in general a SCUP revision may not have a stable labelling that is more committed than the SCUP revision labelling.

Example 6.27. Let $\mathcal{AA}_9$ and its only preferred labelling $LabArg_{pref}$ be as illustrated on the left of Figure 6.17. There are two SCUPs w.r.t. $LabArg_{pref}$, namely $\{a\}$ and $\{e\}$. A SCUP revision $\mathcal{AA}_9^{\circledast}$ of $\mathcal{AA}_9$ is depicted on the right of Figure 6.17 (it coincides with $\mathcal{AA}_6$), along with the SCUP revision labelling. A preferred labelling $LabArg^{\circledast}_{pref}$ of $\mathcal{AA}_9^{\circledast}$ that is more committed than the revision labelling is illustrated in Figure 6.11. However, $LabArg^{\circledast}_{pref}$ is not a stable labelling of $\mathcal{AA}_9^{\circledast}$. Furthermore, in this example there exists no SCUP revision and SCUP revision labelling which result in a stable labelling that is more committed than $LabArg_{pref}$.

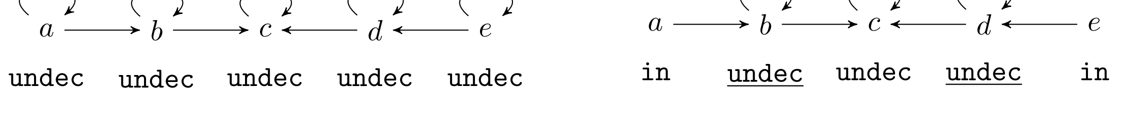


Figure 6.17: Left – $\mathcal{AA}_9$ and its only preferred labelling $LabArg_{pref}$. Right – A SCUP revision $\mathcal{AA}_9^{\circledast}$ of $\mathcal{AA}_9$ (see Example 6.27) and the SCUP revision labelling, where illegal labels are underlined.

To summarise, differently from enforcement sets, revisions w.r.t. SCUPs are not guaranteed to have a stable labelling that is more committed than $LabArg_{pref}$. Nevertheless, they yield a more committed preferred labelling.

If a SCUP revision has a preferred labelling that is not a stable labelling, then by Proposition 6.20 there exists a SCUP w.r.t. this preferred labelling. In order to obtain a stable labelling of the whole AA framework, these "new" SCUPs thus have to be revised. We therefore define an iterative procedure of SCUP revisions w.r.t. preferred labellings.

Definition 6.11 (Iterative SCUP Revision)**.** A sequence $\langle \mathcal{AA}^1, \mathit{LabArg}^1\rangle, \ldots,$ $\langle \mathcal{AA}^n, \mathit{LabArg}^n\rangle$ $(n > 1)$ is an *iterative SCUP revision* of $\mathcal{AA}$ if and only if

- $\mathcal{AA}^1 = \mathcal{AA}$ and $\mathit{LabArg}^1 = \mathit{LabArg}_{\mathit{pref}}$, and
- $\forall i$ $(1 \leq i < n)$ it holds that $\mathcal{AA}^{i+1}$ is a SCUP revision of $\mathcal{AA}^i$ with $\mathit{LabArg}^{\circledast i+1}$ a SCUP revision labelling of $\mathcal{AA}^{i+1}$, and LabArg^{i+1} is a preferred labelling of $\mathcal{AA}^{i+1}$ such that $\mathit{LabArg}^{\circledast i+1} \sqsubseteq \mathit{LabArg}^{i+1}$.

We are, of course, most interested in iterative SCUP revisions that result in a stable labelling.

Definition 6.12 (Stable Iterative SCUP Revision)**.** An iterative SCUP revision $\langle \mathcal{AA}^1, \mathit{LabArg}^1\rangle, \ldots, \langle \mathcal{AA}^n, \mathit{LabArg}^n\rangle$ of $\mathcal{AA}$ is a *stable iterative SCUP revision* of $\mathcal{AA}$ if and only if LabArg^n is a stable labelling of $\mathcal{AA}^n$.

Example 6.28. Consider again $\mathcal{AA}_9$ and its preferred labelling, illustrated on the left of Figure 6.17. An example of a stable iterative SCUP revision of $\mathcal{AA}_9$ is $\langle \mathcal{AA}_9^1, \mathit{LabArg}^1\rangle$, $\langle \mathcal{AA}_9^2, \mathit{LabArg}^2\rangle$, $\langle \mathcal{AA}_9^3, \mathit{LabArg}^3\rangle$, where $\mathcal{AA}_9^2$ and LabArg^2 are depicted in Figure 6.11, and $\mathcal{AA}_9^3$ and LabArg^3 are as illustrated in Figure 6.18.

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r] \arrow[loop above] & c & d \arrow[l] \arrow[loop above] & e \arrow[l] \\
\texttt{in} & \texttt{out} & \texttt{in} & \texttt{out} & \texttt{in}
\end{tikzcd}$$

Figure 6.18: The AA framework obtained from a stable iterative SCUP revision of $\mathcal{AA}_9$ (see Example 6.28).

Since a SCUP revision has a preferred labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$, each iteration in the iterative SCUP revision reduces the set of arguments labelled `undec`. Since there are only finitely many arguments, there exists an iterative SCUP revision that results in a stable labelling.

Theorem 6.27. *There exists a stable iterative SCUP revision of* $\mathcal{AA}$.

Proof. By Proposition 6.20, there exists a SCUP w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ and thus by Corollary 6.26 there exists a preferred labelling LabArg^2 of $\mathcal{AA}^2$ such that $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}^2$. If LabArg^2 is not a stable labelling, then by Proposition 6.20 there exists a SCUP w.r.t. LabArg^2 and thus a SCUP revision $\mathcal{AA}^3$ of $\mathcal{AA}^2$, and by Corollary 6.26 a preferred labelling LabArg^3 of $\mathcal{AA}^3$ such that $\mathit{LabArg}^2 \sqsubset \mathit{LabArg}^3$. The same then applies to $\mathcal{AA}^3$, and so on. Thus, the set of `undec` arguments in LabArg^i monotonically decreases, and since there are only finitely many arguments, the sequence terminates with some $\mathcal{AA}^n$ such that $\texttt{undec}(\mathit{LabArg}^n) = \emptyset$. □

Our results show that SCUPs provide a sufficient condition for the non-existence of a stable labelling that is more committed than $LabArg_{pref}$, and thus characterise parts of an AA framework that necessarily need to be revised in order to obtain a stable labelling. Furthermore, SCUPs can be used for a well-directed revision of the AA framework which leads to a stable labelling that is more committed than $LabArg_{pref}$.

Since by Proposition 6.21 every responsible SCC comprises a SCUP, responsible SCCs also define a sufficient condition for the non-existence of a stable labelling that is more committed than $LabArg_{pref}$, and consequently need to be revised in order to obtain a stable labelling. However, the condition provided by responsible SCCs is less refined than the notion of SCUPs. Therefore, we do not investigate the revision w.r.t. responsible SCCs in more detail.

6.5.5 Responsible Cycles versus Responsible SCCs and SCUPs

The characterisation of responsible arguments in terms of responsible cycles differs considerably from our second and third structural characterisations, which are based on SCCs. Nevertheless, we prove that the three characterisations are connected. In particular, every SCUP comprises a responsible cycle.

Proposition 6.28. *Let $Args$ be a SCUP w.r.t. $LabArg_{pref}$. Then there exists a responsible cycle $\mathscr{C}$ w.r.t. $LabArg_{pref}$ such that $\mathscr{C} \subseteq Args$.*

Proof. By Proposition 6.19 and the SCC recursiveness of complete labellings [BGG05], $\mathcal{AA}\downarrow_{Args}$ has no stable labelling. Then by Corollary 36 in [Dun95b], there exists an odd-length cycle in $Args$. □

Example 6.29. The only SCUP of $\mathcal{AA}_5$ (see left of Figure 6.9) is $\{c, d, e, f, g, h\}$. Here, there are two responsible cycles that form subsets of the SCUP, namely $\{d\}$ and $\{g\}$ (see Example 6.17).

Note that the converse of Proposition 6.28 does not hold, i.e. it is not the case that every responsible cycle is a subset of some SCUP. For instance, in $\mathcal{AA}_9$ (see left of Figure 6.17) each of the five self-attacking arguments is a responsible cycle. However, there are only two SCUPs, namely $\{a\}$ and $\{e\}$, so for instance the responsible cycle $\{b\}$ is not a subset of any SCUP.

Since by Proposition 6.21 every responsible SCC comprises a SCUP, it follows that every responsible SCC contains a responsible cycle.

Corollary 6.29. *Let $Args$ be a responsible SCC w.r.t. $LabArg_{pref}$. Then there exists a responsible cycle $\mathscr{C}$ w.r.t. $LabArg_{pref}$ such that $\mathscr{C} \subseteq Args$.*

Note that Propositions 6.16 and 6.28 imply that rather than defining a SCUP revision w.r.t. all arguments in SCUPs, we could only revise the responsible cycles in the SCUPs. This is illustrated by Example 6.17, where a revision w.r.t. the responsible cycles contained in the only SCUP is illustrated.

On the other hand, the responsible cycles in a SCUP do not *have to* be revised in order to legally label all arguments in the SCUP. Instead, the SCUP may be revised w.r.t. a subset of the SCUP not containing arguments from responsible cycles. For instance, a SCUP revision of $\mathcal{AA}_5$ w.r.t. $LabArg_{pref}$ (see left of Figure 6.9) where no responsible cycles are revised is illustrated in Figure 6.19, along with a preferred labelling that is more committed than the SCUP revision labelling.

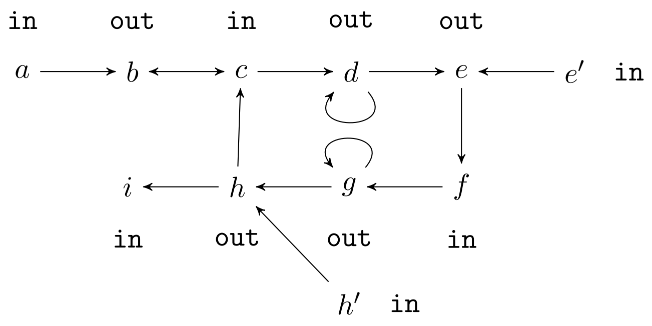


Figure 6.19: A SCUP revision of $\mathcal{AA}_5$ and a preferred labelling that is more committed than the SCUP revision labelling.

It is therefore up to the user to decide what type of SCUP revision is most suitable.

6.6 Labelling-Based versus Structural Characterisations

In the previous sections, we presented two different approaches to characterising sets of arguments responsible for $LabArg_{pref}$ not being a stable labelling: the labelling-based and the structural approach. We proved that the labelling-based characterisations in terms of enforcement and preventing sets define *necessary and sufficient* conditions for the (non-) existence of a stable labelling that is more committed than $LabArg_{pref}$. However, these characterisations are not constructive. On the other hand, our structural characterisations are constructive. They can also be used to guide the revision of an AA framework in such a way that a stable labelling is obtained, but they do not define necessary conditions for the (non-) existence of a stable labelling.

In this section, we examine the connection between our labelling-based and structural characterisations in more detail. Note that we neglect the naive characterisation of labelling-based responsible sets, since both enforcement and preventing sets are refinements of this characterisation. Similarly, we do not include responsible SCCs in our comparison since SCUPs provide a more refined characterisation than responsible SCCs.

6.6.1 SCUPs versus Preventing Sets

SCUPs and preventing sets share the property that if none of their arguments is involved in a revision, then the revision has no stable labelling that is more committed than $LabArg_{pref}$ (see Theorems 6.9 and 6.23). These results hint at a close connection between SCUPs and preventing sets. Indeed, Theorem 6.30 proves that a SCUP comprises a preventing set.

Theorem 6.30. *Let $Args_{SCUP}$ be a SCUP w.r.t. $LabArg_{pref}$. Then there exists a preventing set $Args_{prev}$ w.r.t. $LabArg_{pref}$ such that $Args_{prev} \subseteq Args_{SCUP}$.*

Proof. By Lemma 6.22, for all labellings $LabArg$ of $\mathcal{AA}$ with $LabArg_{pref} \sqsubset LabArg$ and $\mathtt{undec}(LabArg) = \emptyset$ it holds that there exists $A \in Args_{SCUP}$ such that A is illegally labelled by $LabArg$. Then either $Args_{SCUP}$ is a minimal set satisfying this property, and thus $Args_{SCUP}$ is a preventing set, or there exists a minimal set $Args_{prev} \subset Args_{SCUP}$ satisfying this property, so $Args_{prev}$ is a preventing set. □

Example 6.30. Consider again $\mathcal{AA}_5$ illustrated on the left of Figure 6.9. As discussed in Example 6.23, the only SCUP w.r.t. the $LabArg_{pref}$ is $\{c, d, e, f, g, h\}$. Here, two different preventing sets w.r.t. $LabArg_{pref}$ are subsets of the SCUP, namely $\{c, d, g, h\}$ and $\{d, e, f, g\}$.

Note that, conversely, it is not the case that every preventing set is a subset of some SCUP.

Example 6.31. Consider again $\mathcal{AA}_9$, illustrated on the left of Figure 6.17. There are three preventing sets w.r.t. $LabArg_{pref}$: $\{a\}$, $\{e\}$, and $\{b, c, d\}$. The first two coincide with the two SCUPs w.r.t. $LabArg_{pref}$, but the latter is not the subset of any SCUP.

Since SCUPs only characterise the "first" problematic sets of arguments, whereas preventing sets define "all" problematic sets, it is not surprising that some preventing sets are disjoint from SCUPs. However, when considering all SCUPs in a *stable iterative SCUP revision*, every preventing set shares an argument with some SCUP.

Notation 6.13. Let $\langle \mathcal{AA}^1, LabArg^1\rangle, \ldots, \langle \mathcal{AA}^n, LabArg^n\rangle$ be an iterative SCUP revision. $\biguplus \mathcal{SCUPS} = \{\mathcal{SCUPS}^i \mid \mathcal{SCUPS}^i$ *is the set of all arguments in SCUPs w.r.t.* $LabArg^i, 1 \leq i \leq n\}$ consists of the sets of arguments in SCUPs at every step in the iterative SCUP revision.

Theorem 6.31. *Let $\langle \mathcal{AA}^1, LabArg^1\rangle, \ldots, \langle \mathcal{AA}^n, LabArg^n\rangle$ be a stable iterative SCUP revision. Then for all preventing sets $Args_{prev}$ w.r.t. $LabArg_{pref}$ it holds that $\exists \mathcal{SCUPS} \in \biguplus \mathcal{SCUPS}$ such that $\mathcal{SCUPS} \cap Args_{prev} \neq \emptyset$.*

Proof. Let $Args_{prev}$ be a preventing set w.r.t. $LabArg_{pref}$. By (the contrapositive of) Theorem 6.9, it holds that if $\mathcal{AA}^{\circledast}$ is a revision of $\mathcal{AA}$ w.r.t. some $Args \subseteq Ar$ by some $LabArg$ such that some revision labelling $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$, then $Args \cap Args_{prev} \neq \emptyset$. Since $\mathcal{AA}^n$ has a stable labelling $LabArg^n$ and since $\mathcal{AA}^n$ is a revision of $\mathcal{AA}$ w.r.t. $\bigcup_{\mathcal{SCUPS} \in \biguplus \mathcal{SCUPS}} \mathcal{SCUPS}$ by $LabArg^n \cap (Ar \times \{\mathtt{in}, \mathtt{out}, \mathtt{undec}\})$ it holds that $\exists \mathcal{SCUPS} \in \biguplus \mathcal{SCUPS}$ such that $\mathcal{SCUPS} \cap Args_{prev} \neq \emptyset$. □

Example 6.32. Consider again $\mathcal{AA}_9$ illustrated on the left of Figure 6.17 and the stable iterative SCUP revision of $\mathcal{AA}_9$ discussed in Example 6.28. The set of arguments in SCUPs in every step of the stable iterative SCUP revision is $\biguplus \mathcal{SCUPS} = \{\{a, e\}, \{c\}\}$.

For the preventing set $\{b, c, d\}$ w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, which is not a subset of any SCUP w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ (see Example 6.31), there exists the set $\{c\}$ in $\biguplus \mathcal{SCUPS}$ which shares an argument with the preventing set $\{b, c, d\}$. Clearly, the preventing sets $\{a\}$ and $\{e\}$, which are subsets of SCUPs w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, also have a non-empty intersection with a set in $\biguplus \mathcal{SCUPS}$, namely with $\{a, e\}$.

6.6.2 SCUPs versus Enforcement Sets

Next, we investigate the relationship between SCUPs and enforcement sets. We first show that a SCUP contains an argument from each enforcement set.

Theorem 6.32. *Let $\mathit{Args}_{\mathit{SCUP}}$ be a SCUP w.r.t. $\mathit{LabArg}_{\mathit{pref}}$. Then for all enforcement sets $\mathit{Args}_{\mathit{enf}}$ w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ it holds that $\mathit{Args}_{\mathit{SCUP}} \cap \mathit{Args}_{\mathit{enf}} \neq \emptyset$.*

Proof. By Theorem 6.30, there exists a preventing set $\mathit{Args}_{\mathit{prev}}$ w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ such that $\mathit{Args}_{\mathit{prev}} \subseteq \mathit{Args}_{\mathit{SCUP}}$. Since by Theorem 6.10 it holds that for all enforcement sets $\mathit{Args}_{\mathit{enf}}$ w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, $\mathit{Args}_{\mathit{prev}} \cap \mathit{Args}_{\mathit{enf}} \neq \emptyset$, it follows that $\mathit{Args}_{\mathit{SCUP}} \cap \mathit{Args}_{\mathit{enf}} \neq \emptyset$. □

Example 6.33. $\mathcal{AA}_9$, illustrated on the left of Figure 6.17, has two SCUPs w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, namely $\{a\}$ and $\{e\}$ (see Example 6.27). Both SCUPs contain an argument from each of the three enforcement sets w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, i.e. $\{a, b, e\}$, $\{a, c, e\}$, $\{a, d, e\}$. In fact, both *SCUPs are subsets of each enforcement set.*

In contrast, $\mathcal{AA}_5$ illustrated on the left of Figure 6.9 has one SCUP w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, namely $\{c, d, e, f, g, h\}$. Again the SCUP contains an argument from each enforcement set w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, i.e. from $\{d\}$, $\{g\}$, $\{c, e\}$, $\{c, f\}$, $\{e, h\}$, and $\{f, h\}$. In fact, here each *enforcement set is a subset of the SCUP.*

Note that in general, SCUPs are not subsets of enforcement sets or vice versa. For instance, the SCUP $\{a, b, c\}$ of $\mathcal{AA}_4$ (see left of Figure 6.8) is not a subset of any of the enforcement sets $\{a, e\}$, $\{b, e\}$, or $\{c, e\}$, and none of the enforcement sets is a subset of this SCUP.

By Theorem 6.12, we know that if a revision has a stable labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$, the set of arguments used for the revision must be a superset of some enforcement set. Since a stable iterative SCUP revision results in such a stable labelling, it follows that there exists an enforcement set that is a subset of the set of all arguments occurring in SCUPs of the iterative SCUP revision.

Theorem 6.33. *Let $\langle \mathcal{AA}^1, \mathit{LabArg}^1\rangle, \dots, \langle \mathcal{AA}^n, \mathit{LabArg}^n\rangle$ be a stable iterative SCUP revision. Then there exists an enforcement set $\mathit{Args}_{\mathit{enf}}$ w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ such that $\forall A \in \mathit{Args}_{\mathit{enf}} : \exists \mathcal{SCUPS} \in \biguplus \mathcal{SCUPS}$ with $A \in \mathcal{SCUPS}$.*

Proof. By Theorem 6.31, for each preventing set $\mathit{Args}_{\mathit{prev}}$ it holds that $\exists A \in \mathit{Args}_{\mathit{prev}}$ such that $\exists \mathcal{SCUPS} \in \biguplus \mathcal{SCUPS}$ with $A \in \mathcal{SCUPS}$. It then follows from Theorem 6.11 that there exists an enforcement set $Args_{enf}$ such that such that $\forall A \in Args_{enf} : \exists \mathcal{SCUPS} \in \biguplus \mathcal{SCUPS}$ with $A \in \mathcal{SCUPS}$. $\square$

Example 6.34. Consider again $\mathcal{AA}_9$ illustrated in Figure 6.17 and the stable iterative SCUP revision of $\mathcal{AA}_9$ discussed in Example 6.28. $\biguplus \mathcal{SCUPS} = \{\{a, e\}, \{c\}\}$, so there exists an enforcement set whose arguments are all contained in a set in $\biguplus \mathcal{SCUPS}$, namely the enforcement set $\{a, c, e\}$.

The relation between enforcement sets and SCUPs implies that even though a stable iterative SCUP revision is not a *minimal* way of revising the AA framework to obtain a stable labelling, it includes the arguments that definitely have to be revised.

6.6.3 Responsible Cycles versus Enforcement and Preventing Sets

We now turn to the comparison of responsible cycles with enforcement and preventing sets. We first prove that there exists an enforcement set that consists of arguments from responsible cycles.

Theorem 6.34. *Let $S = \{A \in Ar \mid \mathscr{C}$ is a responsible cycle w.r.t. $LabArg_{pref}, A \in \mathscr{C}\}$. Then there exists an enforcement set Args w.r.t. $LabArg_{pref}$ such that $Args \subseteq S$.*

Proof. By Proposition 6.16, there exists a labelling $LabArg$ of $\mathcal{AA}$ with $LabArg_{pref} \sqsubset LabArg$ and $\texttt{undec}(LabArg) = \emptyset$ such that for all revisions $\mathcal{AA}^{\circledast}$ of $\mathcal{AA}$ w.r.t. S by $LabArg$ and all revision labellings $LabArg^{\circledast}$ of $\mathcal{AA}^{\circledast}$, $LabArg^{\circledast}$ is a stable labelling of $\mathcal{AA}^{\circledast}$. It then follows from Theorem 6.12 that there exists an enforcement set $Args$ w.r.t. $LabArg_{pref}$ such that $Args \subseteq S$. $\square$

Example 6.35. Consider again $\mathcal{AA}_5$, illustrated on the left of Figure 6.9. The set of arguments in responsible cycles w.r.t. $LabArg_{pref}$ is $S = \{d, g\}$. There are two different enforcement sets that are subsets of S, namely $\{d\}$ and $\{g\}$. This example also illustrates that not all enforcement sets contain arguments that are part of a responsible cycle, e.g. the enforcement set $\{c, e\}$ is disjoint from S.

Note that not *every* responsible cycle shares arguments with an enforcement set. For instance, the responsible cycle $\{e\}$ w.r.t. the preferred labelling $LabArg_{pref}$ of $\mathcal{AA}_8$, illustrated in Figure 6.14, and the only enforcement set w.r.t. $LabArg_{pref}$, namely $\{c, d\}$, do not have any arguments in common.

Next, we show the connection between responsible cycles and preventing set. In particular, every preventing set comprises a responsible cycle.

Theorem 6.35. *Let Args be a preventing set w.r.t. $LabArg_{pref}$. Then there exists a responsible cycle $\mathscr{C}$ w.r.t. $LabArg_{pref}$ such that $\mathscr{C} \subseteq Args$.*

Proof. Let $ArgsIO = \texttt{in}(LabArg_{pref}) \cup \texttt{out}(LabArg_{pref})$. Assume there exists no responsible cycle $\mathscr{C}$ w.r.t. $LabArg_{pref}$ such that $\mathscr{C} \subseteq Args$. Thus, $\mathcal{AA}{\downarrow}_{Args}$ comprises no odd-length cycles, so by Corollary 36 in [Dun95b] $\mathcal{AA}{\downarrow}_{Args}$ has a stable labelling $LabArg_{Args}$.

By Lemma A.10 in Appendix A, $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}$ is compatible with $\mathit{LabArg}_{\mathit{Args}}$. Furthermore, by the same reasoning as in the proof of Proposition 6.19, $\mathit{LabArg}_{\mathit{Args}}$ is compatible with $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}$. It follows from Lemma A.4 in Appendix A that $\mathit{LabArg}_{\mathit{Args}} \cup \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}$ is a complete labelling of $\mathcal{AA}{\downarrow}_{\mathit{Args} \cup \mathit{ArgsIO}}$. Let LabArg' be a labelling of $\mathit{Args}' = \mathit{Ar} \setminus (\mathit{Args} \cup \mathit{ArgsIO})$ such that $\mathtt{out}(\mathit{LabArg}') = \mathit{Args}'$. Let $\mathit{LabArg} = \mathit{LabArg}_{\mathit{Args}} \cup \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}} \cup \mathit{LabArg}'$. Clearly $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$. Furthermore, $\forall A \in \mathit{Args}$ it holds that A is legally labelled by LabArg. Contradiction, since by Definition 6.5, $\forall \mathit{LabArg}$ with $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}$ and $\mathtt{undec}(\mathit{LabArg}) = \emptyset$ it holds that $\exists A \in \mathit{Args}$ such that A is illegally labelled by LabArg. $\square$

Example 6.36. Consider again $\mathcal{AA}_5$, illustrated on the left of Figure 6.9. The two preventing sets w.r.t. the preferred labelling $\mathit{LabArg}_{\mathit{pref}}$ of $\mathcal{AA}_5$ are $\{c, d, g, h\}$ and $\{d, e, f, g\}$. Both contain a responsible cycle w.r.t. $\mathit{LabArg}_{\mathit{pref}}$, in this case even two responsible cycles, namely $\{d\}$ and $\{g\}$.

These results imply that odd-length cycles of arguments labelled `undec` by $\mathit{LabArg}_{\mathit{pref}}$ are an important characteristic of sets of arguments that prevent $\mathit{LabArg}_{\mathit{pref}}$ from being a stable labelling (Theorem 6.35). Furthermore, it is sufficient to revise (specific) arguments in odd-length cycles to obtain a stable labelling that is more committed than $\mathit{LabArg}_{\mathit{pref}}$ (Theorem 6.34).

6.7 Discussion and Related Work

In this section, we first discuss why we chose to investigate the non-existence of stable labellings in terms of *preferred* labellings not being stable labellings. We then compare our approach to related work.

6.7.1 Preferred versus Semi-Stable Labellings

We investigated the question as to why an AA framework has no stable labellings in terms of why a preferred labelling is not a stable labelling. We also considered to use *semi-stable labellings* instead, as they are even closer to the notion of stable labelling.

However, with regards to SCUPs, semi-stable labellings lead to a problem: even though SCUPs can be defined with respect to a semi-stable instead of a preferred labelling, stable iterative SCUP revisions may *not* exist when defining SCUPs with respect to a semi-stable labelling. The reason is that Theorem 6.25 and Corollary 6.26 are not guaranteed to hold for semi-stable labellings, i.e. a SCUP revision may not have a semi-stable labelling that is *more committed* than the semi-stable labelling of the original AA framework.

Example 6.37. Let $\mathcal{AA}_{10}$ be the AA framework on the left of Figure 6.20, which also illustrates the only preferred and only semi-stable labelling $\mathit{LabArg}_{\mathit{pref}}$ of $\mathcal{AA}_{10}$. The only SCUP w.r.t. $\mathit{LabArg}_{\mathit{pref}}$ is $\{a\}$. A SCUP revision $\mathcal{AA}_{10}^{\circledast}$ of $\mathcal{AA}_{10}$ and its SCUP revision labelling $\mathit{LabArg}^{\circledast}$ are shown on the right of Figure 6.20. The left of Figure 6.21

illustrates the only preferred labelling of $\mathcal{AA}_{10}^{\circledast}$ that is more committed than $LabArg^{\circledast}$ and $LabArg_{pref}$. Note that this preferred labelling is not a semi-stable labelling of $\mathcal{AA}_{10}^{\circledast}$. The only semi-stable labelling of $\mathcal{AA}_{10}^{\circledast}$ is illustrated on the right of Figure 6.21. It is not more (or equally) committed than $LabArg^{\circledast}$. The same problem arises if the SCUP is revised in such a way that a is labelled `out` in the SCUP revision labelling, as illustrated on the left of Figure 6.22. The only semi-stable labelling of the SCUP revision is shown on the right of Figure 6.22, which is not more or equally committed than the SCUP revision labelling or $LabArg_{pref}$.

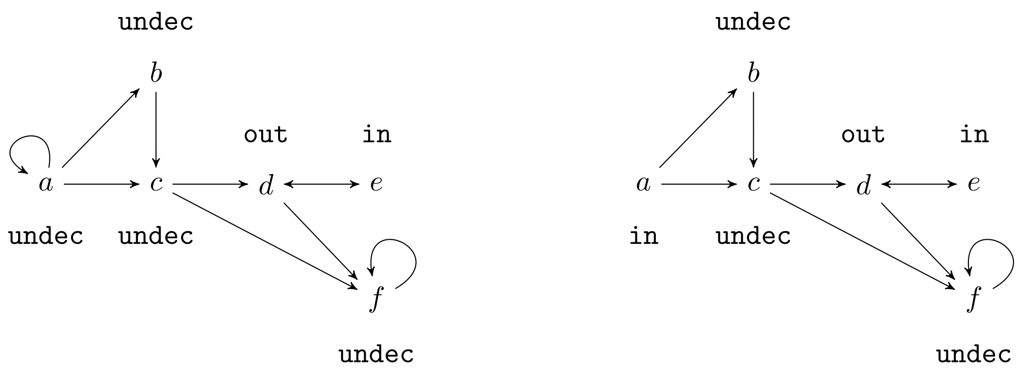


Figure 6.20: Left – The only preferred and semi-stable labelling of $\mathcal{AA}_{10}$. Right – A SCUP revision $\mathcal{AA}_{10}^{\circledast}$ of $\mathcal{AA}_{10}$ and a SCUP revision labelling.

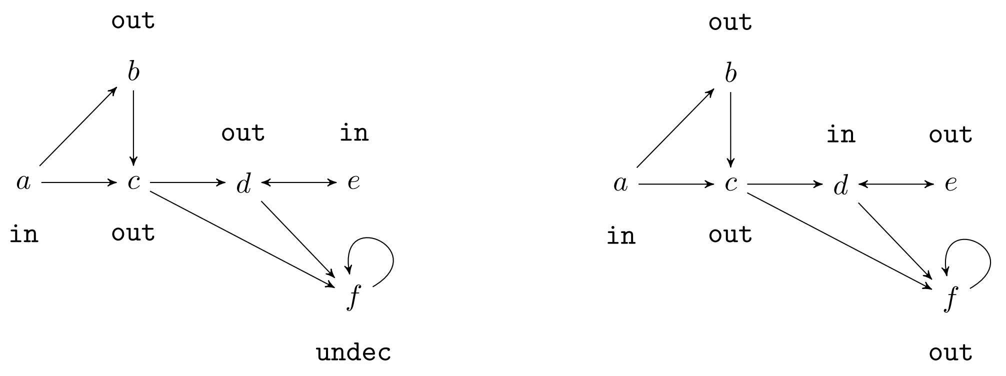
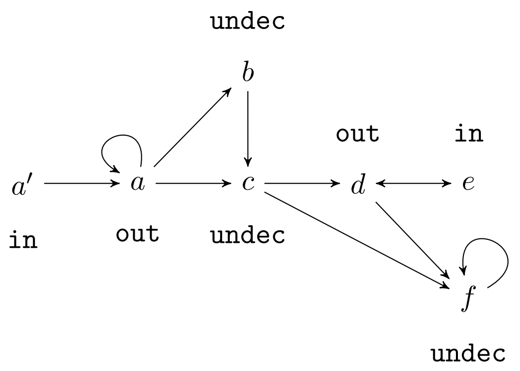


Figure 6.21: A preferred labelling of $\mathcal{AA}_{10}^{\circledast}$ that is more committed than $LabArg_{pref}$ (left) and the only semi-stable labelling of $\mathcal{AA}_{10}^{\circledast}$ (right).

The problem with defining SCUPs with respect to semi-stable rather than preferred labellings is thus that iterative SCUP revisions cannot be applied unless we are prepared to change the labels of arguments already labelled `in` and `out` by the semi-stable labelling of the original AA framework. However, this would defeat the spirit of our work as we are interested in why a *particular* labelling is not a stable labelling and how to turn this *particular* labelling into a stable labelling.



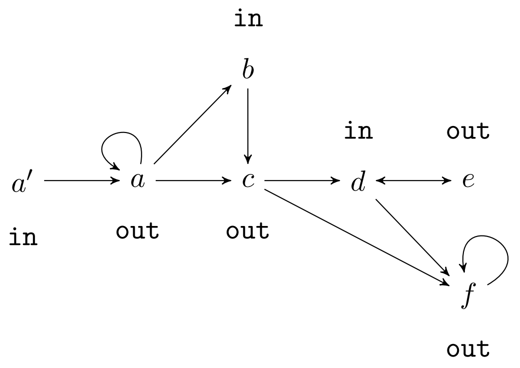


Figure 6.22: Another SCUP revision of $\mathcal{AA}_{10}$ and a SCUP revision labelling (left), and the only semi-stable labelling of this SCUP revision (right).

6.7.2 Related Work

Related to our work on stable semantics, Baumann and Strass [BS13] focus on the question *how many* stable extensions an AA framework has on average and what the maximal number of stable extensions is. Furthermore, Dunne and Bench-Capon [DBC02] investigate AA frameworks whose stable and preferred extensions coincide, so-called coherent AA frameworks, and thus deal with AA frameworks that *always* have a stable extension.

To the best of our knowledge, the only work investigating the non-existence of stable extensions or labellings is by Nouioua and Würbel [NW14], who propose a revision operator which transforms an AA framework without stable extensions into one with a stable extension. Their setting is different from ours as they assume that the AA framework in question, which has no stable extension, was obtained from an addition of arguments and attacks to some original AA framework. Assuming that the added arguments and attacks are "correct", they restrict the structural change performed by the revision operator to the original AA framework. Furthermore, Nouioua and Würbel's approach differs from ours in various ways: Firstly, they revise an AA framework through a particular structural change, namely the deletion of attacks, whereas in our approach the addition of arguments and attacks is allowed, too. Furthermore, their approach is not concerned with preserving a particular preferred, or even the grounded, labelling when performing the structural change. Most importantly, their work does not aim to characterise which part of the AA framework is *responsible* for the non-existence of stable extensions, but simply to find minimal (regarding cardinality) changes that guarantee the existence of a stable extension. In our approach minimality also plays a role, as enforcement sets are minimal (regarding set inclusion) sets of arguments used to obtain a stable labelling.

Like us, Baroni et al. [BGL15] are interested in arguments labelled `undec`. However, rather than investigating how `undec` labels can be turned into definite `in` or `out` labels, they argue that undecidedness is desirable in some situations and review various semantics that include different notions of "undecidedness".

In the following sections, we review some further strands of research sharing particular aspects with our work.

Cycles in Argumentation Frameworks

Recently, cycles (of attacking arguments) in AA frameworks have received considerable attention, including a special issue of the Journal of Logic and Computation [BGG16]. Many authors regard the behaviour of preferred semantics with respect to cycles as "problematic", as it treats odd-length and even-length cycles differently. In particular, arguments in odd-length cycles can often only be labelled `undec`, as is the case for our responsible cycles, whereas arguments in even-length cycles can alternately be labelled `in` and `out`.

Baroni et al. [BGG05] discuss this "problematic" behaviour of preferred semantics and introduce the *CF2* semantics for AA frameworks, which "correctly" handles odd- and even-length cycles. Dvořák and Gaggl [DG16] extend the CF2 semantics to the so-called *stage2* semantics, which fulfils some additional properties. Arieli [Ari16] introduces a new family of *conflict-tolerant* semantics, where the conflict-freeness requirement for extensions is dropped. Therefore, odd- and even-length cycles are treated the same by the new semantics. Gabbay [Gab16b] defines another family of new semantics able to handle the "problematic" behaviour of the preferred semantics concerning cycles. In the new *loop busting* semantics no argument is labelled `undec`. The procedure for computing the new semantics has similarities with ideas used in our approach, since it iteratively applies a specific type of revision of initial SCCs. More precisely, an argument in an initial SCC of the `undec`-part with respect to the grounded extension is chosen and a new attacker is added. Then the grounded labelling of the new AA framework is computed and the same procedure is performed iteratively for the new AA framework restricted to arguments labelled `undec`. The iterative SCUP revision introduced here applies a similar approach since an initial SCC of the `undec`-part (i.e. a SCUP) is revised and the revision is then repeated on the AA framework restricted to arguments still labelled `undec`. However, we allow for any revision and use the *preferred* rather than grounded semantics. Bodanza and Tohmé [BT09] propose two new semantics for handling odd-length cycles: the first one allows to accept arguments attacked by an odd-length cycle, and the second one additionally allows to accept single arguments in an odd-length cycle. Both types of semantics yield labellings which are more committed than preferred labellings.

In contrast to the aforementioned works, Bench-Capon [BC16] argues that the way the preferred semantics handles cycles is not "problematic" by providing an interpretation of even-length cycles as dilemmas and odd-length cycles as paradoxes. He argues that using this point of view, it is reasonable that arguments in odd-length cycles are neither true nor false, and that consequently their justification status cannot be decided.

Note that the motivation of our approach is completely different from the motivations of the works reviewed above. We do not make any claims about whether or not the preferred semantics handles cycles "correctly", and are therefore not concerned with new semantics. Instead, we characterise specific parts of an AA framework comprising odd-length cycles as *responsible* for the non-existence of stable labellings and define a procedure for turning a preferred labelling into a stable labelling by structurally revising the AA framework.

Like us, Baumann and Woltran [BW16] are not concerned with the "correct" or "incorrect" behaviour of semantics regarding odd-length cycles. Instead, they study the role of self-attacking arguments, i.e. cycles of length one, with regards to the equivalence of AA frameworks.

Splitting Argumentation Frameworks

Two of our structural characterisations build upon the idea of SCCs introduced in [BGG05]. We investigate a particular type of SCCs, namely specific *initial* SCCs, and use Baroni et al.'s results [BGG05] that the preferred and stable semantics are SCC-recursive, i.e. that the preferred or stable extensions (or equivalently labellings) of an AA framework can be obtained by computing the respective extensions for initial SCCs and using them recursively for computing the extensions of the following SCCs. Liao [Lia13] shows how the semantics of an AA framework can be computed by the step-wise computation of semantics of SCCs and Baroni et al. [BBC$^+$14] generalise the results about SCCs, showing how complete labellings of an AA framework can be computed by combining complete labellings of *arbitrary* parts of the AA framework. We apply and extend Baroni et al.'s results for a particular partitions of an AA framework into the set of arguments labelled `in` or `out` by a preferred labelling, and (a subset of the) arguments labelled `undec`.

Our results about combining a labelling of a SCUP with the `in` and `out` labels in a preferred labelling are also related to the *splitting* results of Baumann [Bau11] and Baumann et al. [BBDW12]. They show that for the stable semantics, extensions of an AA framework can be obtained by splitting the AA framework into two parts and computing the extensions of the two parts using a method that takes the extensions of the respective other part into account. Another related approach was introduced by Rienstra et al. [RPV$^+$11], who propose *multi-sorted extensions* as a new semantics of an AA framework with respect to a partition of the AA framework. A multi-sorted extension is such that its restriction to a part coincides with a given semantics for this part. This approach is conceptually related to our work, which combines the stable labellings of parts of the AA framework, namely SCUPs, with `in` and `out` labels from a preferred labelling.

Dynamics in Argumentation Frameworks

The study of dynamics in AA frameworks has received considerable attention in recent years. Our work investigates the dynamics of AA frameworks from a special angle since we are not concerned with the exact structural change of an AA framework and its effect (as e.g. in [CdSCLS10]), but rather with effects that may be obtained through *various different* structural changes. Importantly, *which* structural change is chosen is not of importance for our work as long as it results in arguments being legally labelled as desired.

Liao et al. [LJK11] introduce a general approach for computing extensions of an AA framework that has been structurally changed, allowing for any number of additions and deletions of arguments and attacks. The idea is that in order to compute the semantics of the new AA framework only the semantics of the part of the AA framework that is *affected* by the structural change has to be re-computed. The semantics of the *unaffected* part stays the same as before the structural change and only "conditions" the extensions of the affected part. This idea is related to our iterative SCUP revisions, where we do not change the labels of arguments labelled `in` or `out` in the SCUP revision labelling (they are "unaffected"), but only of those labelled `undec`, which are "conditioned" by the `in` and `out` labels of the SCUP revision labelling.

The work of Booth et al. [BKRvdT13] is of similar spirit to our work, but concerned with the complete rather than the stable semantics: they investigate how to turn a non-complete labelling into a complete one through a structural change. In contrast to our work, Booth et al. assume an *intended* complete labelling, whereas for our approach no intended stable labelling is required.

Baumann and Brewka [BB10] were the first to investigate whether certain sets of arguments can be *enforced* as an extension according to a chosen semantics. In contrast to our general revisions, they only allow structural changes called "expansions", where arguments and attacks can be added, and new attacks must involve a new argument. Baumann and Brewka prove that for certain kinds of expansions, all arguments that are part of extensions before the structural change are also part of extensions after the structural change. In line with their work, we show that for any revision w.r.t. an enforcement set by an enforcement labelling, a stable labelling is obtained in which all previously `in`- and `out`-labelled arguments keep their labels. Baumann [Bau12] as well as Coste-Marquis et al. [CMKMM14b] study how to enforce a set of arguments through a *minimal* structural change of adding or deleting attacks. Similarly, we prove that enforcement sets are minimal sets of arguments that, when used for a revision, yield a stable labelling. Coste-Marquis et al. [CMKMM15] introduce a whole family of revision operators which can be used for enforcement, generalising revision operators defined by others, e.g. [KBM^{+}13, BGK^{+}14, CMKMM14a]. Other authors [BCdSCLS13, DHP14, BGP^{+}11] study enforcements as logical formulae to be satisfied through structural change. It is important to note that even though enforcement is a related problem, we do not assume a set of arguments to be "enforced" as an extension of the SCUP. In contrast, we only require that some stable extension exists after the revision.

However, the previously mentioned approaches could be used for enforcing a certain set of arguments as a stable extension of a SCUP.

6.8 Summary

We gave three labelling-based and three structural characterisations of sets of arguments responsible for the non-existence of stable argument labellings. These sets characterise reasons why a *preferred argument labelling* is not a stable argument labelling and are thus defined with respect to a chosen preferred argument labelling. We also investigated revisions of the AA framework using our different notions of responsible sets, and in particular whether or not such revisions can turn the chosen preferred argument labelling into a stable argument labelling.

In the basic labelling-based characterisation, the set of all arguments labelled `undec` by the chosen preferred argument labelling is deemed responsible, since arguments labelled `undec` violate the definition of stable argument labelling. Our two non-naive labelling-based approaches characterise responsible sets of arguments with respect the legality of labels in argument labellings that are more committed than the chosen preferred argument labelling. We also proved that these two characterisations define *necessary and sufficient* conditions for the existence and non-existence, respectively, of a stable argument labelling (that is more committed than the chosen preferred argument labelling) after revising the AA framework with respect to such a responsible set of arguments.

Since the labelling-based characterisations are declarative rather than constructive, we also give constructive characterisations of responsible sets of arguments based on the structure of the AA framework. Two characterisations define special types of SCCs as responsible that the preferred argument labelling in question is not a stable argument labelling. The first one characterises the "first" SCCs that have no stable argument labelling (that is more or equally committed than the preferred argument labelling). Our second structural characterisation refines this notion to initial SCCs of the AA framework restricted to arguments labelled `undec` by the chosen preferred argument labelling. We call the sets of arguments thus characterised as responsible *SCUPs* (Strongly Connected `undec` Parts). We also introduce an iterative procedure for revising SCUPs, which yields a revised AA framework that has a stable argument labelling which is more committed than the chosen preferred argument labelling. Following findings by Dung [Dun95b], our third structural characterisation defines odd-length cycles of arguments labelled `undec` by the chosen preferred argument labelling as responsible. Even though each SCUP contains an odd-length cycle, we show that the cycles may not have to be revised in the iterative revision of SCUPs to obtain a stable argument labelling.

We compared our labelling-based and structural characterisations, proving that SCUPs provide a constructive approximation of our precise labelling-based characterisations. In other words, even though SCUPs do not define necessary conditions for the (non-) existence of a stable argument labelling after revising the AA framework, they are *sufficient* for obtaining a stable argument labelling. Furthermore, our comparison shows that odd-length cycles are an important characteristic of all our characterisations.

In the next chapter, we will transfer our notion of SCUPs to inconsistent logic programs without explicitly negated atoms and show that this characterises parts of a logic program which are responsible for the inconsistency. Whether or not our additional results on obtaining a stable argument labelling using our notions of responsible sets of arguments can also be transferred to inconsistent logic programs in order to restore consistency is left for future work.

Chapter 7

Classifying and Explaining Inconsistency in Answer Set Programming

7.1 Introduction

A logic program may comprise two kinds of negation: *explicit negation* and *negation as failure* (NAF). If no negation of either kind is present, a logic program will always be consistent under the answer set semantics [GL91]. However, if negation is used in a logic program, inconsistency may arise in one of two different ways: either the only answer set of the logic program is the set of all literals, or the logic program has no answer sets at all.

In the case of an inconsistent logic program, answer set solvers do not provide any classification of the inconsistency, or explanation thereof. Especially when dealing with a large inconsistent logic program or if the inconsistency is unexpected, understanding why the inconsistency arises and which part of the logic program is responsible for it is an important first step towards debugging the logic program in order to restore consistency. Various approaches have been developed for finding the source of inconsistency, and even for suggesting ways of debugging the logic program. These approaches assume explicitly or implicitly the existence of an intended answer set.

We propose a new method for identifying the reason of inconsistency in a logic program without the need of an intended answer set, which is based on the well-founded and 3-valued M-stable models of the logic program in question. Based on our results on the non-existence of stable argument labellings in AA frameworks from Chapter 6, we first investigate inconsistency in logic programs without explicitly negated atoms, since their stable models correspond one-to-one to the stable argument labellings of the translated AA framework (see Section 4.4). We show that the concept of SCUPs can be transferred to logic programs. We then prove that the two ways in which a logic program with both NAF literals and explicitly negated atoms may be inconsistent (no answer set or the only answer set is the set of all literals) are further divided into four inconsistency cases, which have different reasons for the inconsistency: one where only explicit negation is responsible and the only answer set is the set of all literals, one where only NAF is responsible and the logic program has no answer sets, and two where an interplay of explicit negation and NAF is responsible and the logic program has no answer sets.

We show how in each of these inconsistency cases the reason of the inconsistency can be refined to a characteristic set of "culprit literals". These "culprit literals" can then be used to construct trees whose nodes hold derivations that explain why the inconsistency arises and which part of the logic program is responsible.

The chapter is organised as follows. In Section 7.2, we define SCUPs of an inconsistent logic program without explicit negation and characterise them as responsible for the inconsistency. In Section 7.3, we characterise inconsistency in logic programs *with* explicit negation, distinguishing three inconsistency cases. We then show in Section 7.4, that the third cases can be further divided into two different sub-cases, and characterise sets of literals that are responsible for the inconsistency in each case. In Section 7.5, we illustrate how to construct explanation trees for the responsible literals and in Section 7.7

we summarise the contributions of this chapter.

7.2 Inconsistency in Logic Programs without Explicit Negation

In this section, we characterise sets of literals that can be deemed *responsible* for the non-existence of answer sets of a logic program without explicitly negated atoms. More precisely, we show that the concept of SCUPs, as defined for AA frameworks in Chapter 6, can be transferred to logic programs without explicitly negated atoms. Throughout this section, we assume as given a logic program $\mathcal{P}$ without explicitly negated atoms that has no (2-valued) stable model (equivalently, no answer set – see Section 2.3.4), and a 3-valued M-stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$.[1]

We first define a special kind of negative dependency graph, whose nodes consist of all atoms in $\mathcal{U}$ with respect to a chosen 3-valued M-stable model and whose edges indicate negative dependencies between these atoms. Importantly, a negative dependency of an atom a_2 on an atom a_1 is excluded if the derivation of a_2 dependent on a_1 is also dependent on an atom $a \in \mathcal{T}$.

Definition 7.1 (Negative Undefined Dependency Graph)**.** The *negative undefined dependency graph* of $\mathcal{P}$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ is (V, E) with

- $V = \mathcal{U}$, and
- $E = \{(a_1, a_2) \in V \times V \mid \exists \mathcal{P} \cup \Delta \vdash_{MP} a_2$ *s.t.* $\texttt{not}\ a_1 \in \Delta$ *and* $\nexists \texttt{not}\ a \in \Delta$ *with* $a \in \mathcal{T}\}$.

Example 7.1. Let $\mathcal{P}_{15}$ be the following logic program:

$$\begin{aligned}
\{\ & u \leftarrow \texttt{not}\ w, \texttt{not}\ z;\\
& u \leftarrow \texttt{not}\ z;\\
& u \leftarrow \texttt{not}\ p, \texttt{not}\ q;\\
& w \leftarrow \texttt{not}\ u;\\
& z \leftarrow \texttt{not}\ w;\\
& p \leftarrow \texttt{not}\ p;\\
& q \leftarrow\ \}
\end{aligned}$$

The only 3-valued stable model of $\mathcal{P}_{15}$ is $\langle \{q\}, \emptyset \rangle$ with $\mathcal{U} = \{u, w, z, p\}$. The negative undefined dependency graph of $\mathcal{P}_{15}$ w.r.t. the 3-valued stable model is displayed in Figure 7.1. Note that there is no negative dependency between u and p since $\mathcal{P}_{15} \cup \{\texttt{not}\ p, \texttt{not}\ q\} \vdash_{MP} u$ is such that $q \in \mathcal{T}$.

[1]Note that $\mathcal{P}$ has at least one 3-valued M-stable model as noted in Section 2.3.4.

$$\begin{tikzcd}
p \arrow[loop below] & u \arrow[rr, bend left, <->] & & w \arrow[dl] \\
& & z \arrow[ul] &
\end{tikzcd}$$

Figure 7.1: The negative undefined dependency graph of $\mathcal{P}_{15}$ w.r.t. its only 3-valued stable model (see Example 7.1).

The reason that derivations of an atom $a_2 \in \mathcal{U}$ that are negatively depended on an atom in $\mathcal{T}$ are not taken into account is that these derivations are not the reason why the truth value of a_2 is $\mathtt{U}$. Consider for instance the clause $u \leftarrow \mathtt{not}\ p, \mathtt{not}\ q$. Since $q \in \mathcal{T}$, $val(\mathtt{not}\ q) = \mathtt{F}$, and therefore the clause is satisfied independently of the truth value of u. If this was the only clause with head u, then the truth value of u in 3-valued stable models would be $\mathtt{F}$ rather than $\mathtt{U}$. However, the truth value of u is $\mathtt{U}$, which is, consequently, due to one of the other clauses with head u.

7.2.1 SCUPs of a Logic Program

We define SCUPs of a logic program based on a negative undefined dependency graph w.r.t. a 3-valued M-stable model. Similarly to AA frameworks, SCUPs of a logic program are initial SCCs.

Definition 7.2 (Strongly Connected Undefined Part)**.** $S \subseteq \mathcal{HB}_{\mathcal{P}}$ is a *strongly connected undefined part* (SCUP) w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ if S is an initial SCC of the negative undefined dependency graph of $\mathcal{P}$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$.

Example 7.2. Consider again $\mathcal{P}_{15}$ from Example 7.1 and the negative undefined dependency graph of $\mathcal{P}_{15}$ w.r.t. its only 3-valued M-stable model $\langle \{q\}, \emptyset \rangle$ shown in Figure 7.1. It is easy to see from the graph that there are two SCUPs w.r.t. $\langle \{q\}, \emptyset \rangle$, namely $\{u, w, z\}$ and $\{p\}$.
If we add $\mathtt{not}\ p$ to the body of the second clause of $\mathcal{P}_{15}$, obtaining the new logic program $\mathcal{P}_{16}$, we obtain the negative undefined dependency graph in Figure 7.2 w.r.t. $\langle \{q\}, \emptyset \rangle$, which is the only 3-valued stable model of $\mathcal{P}_{16}$. Then only SCUP w.r.t. $\langle \{q\}, \emptyset \rangle$ of $\mathcal{P}_{16}$ is $\{p\}$.

$$\begin{tikzcd}
p \arrow[loop below] \arrow[r] & u \arrow[rr, bend left, <->] & & w \arrow[dl] \\
& & z \arrow[ul] &
\end{tikzcd}$$

Figure 7.2: The negative undefined dependency graph of $\mathcal{P}_{16}$ w.r.t. its only 3-valued stable model (see Example 7.2).

Importantly, SCUPs characterise parts of an inconsistent logic program (without explicit negation) that always exist.

Proposition 7.1. *There exists a SCUP w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$.*

Proof. Since $\mathcal{U} \neq \emptyset$, the negative undefined dependency graph of $\mathcal{P}$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ has a non-empty set of vertices. Thus, it has an initial SCC, which is a SCUP w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$. □

7.2.2 SCUPs of a Logic Program versus SCUPs of an AA framework

In this section, we investigate the relationship between SCUPs of logic programs and SCUPs of their translated AA frameworks. This relies on the correspondence between 3-valued M-stable models of the logic program and preferred argument labellings of the translated AA framework discussed in Chapter 4.

We first prove that a SCUP of a logic program consists of the conclusions of the arguments in a corresponding SCUP of the translated AA framework.

Theorem 7.2. *Let $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ be the translated AA framework of $\mathcal{P}$ and LabArg the corresponding preferred argument labelling of $\langle \mathcal{T}, \mathcal{F} \rangle$ in $AA_{\mathcal{P}}$. If Args is SCUP w.r.t. LabArg of $AA_{\mathcal{P}}$, then $S = \{a \in \mathcal{HB}_{\mathcal{P}} \mid Asms \vdash a \in Args\}$ is a SCUP w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$.*

Proof.

- We first show that $S \subseteq \mathcal{U}$: Let $Asms_a \vdash a \in Args$. Then $\exists Asms_b \vdash b \in Args$ such that $Asms_a \vdash a \in Args$ attacks $Asms_b \vdash b \in Args$ since $Args$ is strongly connected. Thus, $\forall Asms' \vdash a \neq Asms_a \vdash a$ it holds that $Asms' \vdash a$ attacks $Asms_b \vdash b \in Args$. Since $Args$ is an initial SCC of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle \downarrow_{\texttt{undec}(LabArg)}$, no argument in $Args$ is attacked by an argument labelled `in` by $LabArg$ or by an argument labelled `undec` by $LabArg$ which is not contained in $Args$. Therefore, $Asms' \vdash a \in \texttt{out}(LabArg)$ or $Asms' \vdash a \in \texttt{undec}(LabArg)$ and $Asms' \vdash a \in Args$. It follows by $\texttt{LabArg2Mod}_{Wu}$ (see Section 4.4) that $a \in \mathcal{U}$. Since this holds for all arguments in $Args$, we conclude that $S \subseteq \mathcal{U}$.

- We now show that all $a \in S$ are strongly connected in the negative undefined dependency graph (V, E) of $\mathcal{P}$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$: Let $Asms_a \vdash a, Asms_b \vdash b \in Args$ and $Asms_a \vdash a$ attacks $Asms_b \vdash b \in Args$. By `Mod2LabArg` (see Section 4.4), $\nexists \texttt{not}\ x \in Asms_a, Asms_b$ with $x \in \mathcal{T}$ since $Args \subseteq \texttt{undec}(LabArg)$. Therefore, $(a, b) \in V$. Since this holds for all arguments in $Args$ and since $Args$ is strongly connected, it follows that S is strongly connected in (V, E).

- Lastly, we show that S is an initial SCC, i.e. that $\nexists a \in \mathcal{U}$ such that $a \notin S$ but S is negatively dependent on a in (V, E): Let $a \in \mathcal{U}$ and $a \notin S$. Assume S negatively depends on a in (V, E), i.e. $\exists s \in S$ such that $(a, s) \in E$. Thus, by Definition 7.1 $\exists Asms_s \vdash s$ with $\texttt{not}\ a \in Asms_s$ and $\nexists \texttt{not}\ x \in Asms_s$ with $x \in \mathcal{T}$. Then by `Mod2LabArg` it holds that $Asms_s \vdash s \notin \texttt{out}(LabArg)$. Since $s \in S$ it holds that $\exists Asms'_s \vdash s \in Args$. By the first item of this proof, it follows that $Asms_s \vdash s \in \texttt{undec}(LabArg)$ and $Asms_s \vdash s \in Args$. Since $a \in \mathcal{U}$ it follows by

$\texttt{LabArg2Mod}_{\mathcal{W}u}$ that $\exists Asms_a \vdash a \in \texttt{undec}(LabArg)$. Then clearly $Asms_a \vdash a$ attacks $Asms_s \vdash s \in Args$, so $Args$ is attacked by an argument labelled $\texttt{undec}(LabArg)$ in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle$. Therefore, $Asms_a \vdash a \in Args$ since $Args$ is an initial SCC of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle\downarrow_{\texttt{undec}(LabArg)}$. Thus, by definition of S, it follows that $a \in S$. Contradiction.

Note that the last item also proves that there exists no $a \in \mathcal{U}$ such that $a \notin S$ but a and S are strongly connected in (V, E). In other words, S comprises all a that are strongly connected with S in (V, E). □

Conversely, we prove that a SCUP of the translated AA framework consists of those undec-labelled arguments whose conclusion is in a corresponding SCUP of the logic program.

Theorem 7.3. *Let $AA_{\mathcal{P}}$ be the translated AA framework of $\mathcal{P}$ and LabArg the corresponding preferred argument labelling of $\langle\mathcal{T}, \mathcal{F}\rangle$ in $AA_{\mathcal{P}}$. If S is SCUP w.r.t. $\langle\mathcal{T}, \mathcal{F}\rangle$ of $\mathcal{P}$, then $Args = \{Asms \vdash a \in \texttt{undec}(LabArg) \mid a \in S\}$ is a SCUP w.r.t. LabArg of $AA_{\mathcal{P}}$.*

Proof. By definition of $Args$ it holds that $Args \subseteq \texttt{undec}(LabArg)$.

- We show that all arguments in $Args$ are attacked by some argument in $Args$: Assume $\exists Asms_a \vdash a \in Args$ such that $\nexists Asms_b \vdash b \in Args$ and $Asms_b \vdash b$ attacks $Asms_a \vdash a$. By Mod2LabArg (see Section 4.4), $\exists \texttt{not}\ c \in Asms$ such that $c \in \mathcal{U}$ and $\nexists \texttt{not}\ x \in Asms$ such that $x \in \mathcal{T}$. Then by Definition 7.1, $(c, a) \in E$. It follows that $c \in S$ since $a \in S$ and S is an initial SCC of (V, E). By $\texttt{LabArg2Mod}_{\mathcal{W}u}$, $\exists Asms_c \vdash c \in \texttt{undec}(LabArg)$, so by the definition of $Args$, $Asms_c \vdash c \in Args$. Furthermore, clearly, $Asms_c \vdash c$ attacks $Asms_a \vdash a$. Contradiction.

- Next, we show that if an argument in $Args$ is attacked by an argument contained in $Args$ other than itself, then it also attacks an in $Args$ other than itself: Let $Asms_a \vdash a, Asms_b \vdash b \in Args$, $Asms_a \vdash a \neq Asms_b \vdash b$, and $Asms_a \vdash a$ attacks $Asms_b \vdash b$. Since $Asms_b \vdash b \in \texttt{undec}(LabArg)$, it follows from Mod2LabArg that $\nexists \texttt{not}\ x \in Asms_b$ with $x \in \mathcal{T}$. Thus, by Definition 7.1, $(a, b) \in E$ and by the definition of $Args$, $a, b \in S$. Since S is strongly connected it follows that there exists some $c \in S$ such that $c \neq b$ and $(b, c) \in E$. Therefore, $\exists Asms_c \vdash c$ with $\texttt{not}\ b \in Asms_c$, so $Asms_b \vdash b$ attacks $Asms_c \vdash c$. Furthermore, since $\nexists \texttt{not}\ x \in Asms_c$ with $x \in \mathcal{T}$, it follows from Mod2LabArg that $Asms_c \vdash c \in \texttt{undec}(LabArg)$. Then by the definition of $Args$, $Asms_c \vdash c \in Args$.

It follows from these two items that $Args$ is strongly connected.
It remains to prove that $Args$ is an initial SCC of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}}\rangle\downarrow_{\texttt{undec}(LabArg)}$: Assume that $\exists Asms_a \vdash a \in \texttt{undec}(LabArg)$ with $Asms_a \vdash a \notin Args$ and $\exists Asms_b \vdash b \in Args$ such that $Asms_a \vdash a$ attacks $Asms_b \vdash b$. Since $Asms_b \vdash b \in Args$ it follows that $\nexists \texttt{not}\ x \in Asms$ such that $x \in \mathcal{T}$. Thus, $a \notin \mathcal{T}$. Since it is not the case that $\forall Asms' \vdash a : Asms' \vdash$

$a \in \texttt{out}(LabArg)$, we conclude by $\texttt{LabArg2Mod}_{Wu}$ that $a \in \mathcal{U}$. Then by Definition 7.1, $(a,b) \in E$. Since by the definition of $Args$, $b \in S$, and since S is an initial SCC, it follows that $a \in S$. Therefore, $Asms_a \vdash a \in Args$. Contradiction. □

Example 7.3. Consider the translated AA framework $AA_{\mathcal{P}_{15}}$ of $\mathcal{P}_{15}$, illustrated in Figure 7.3 with its assumption-arguments omitted, since assumption-arguments are never part of a SCUP as they do not attack any argument. Let $A_1, \dots, A_5$ be the assumption-arguments of $\texttt{not } u, \texttt{not } w, \texttt{not } z, \texttt{not } p, \texttt{not } q$, respectively. The only complete (and thus only preferred) argument labelling of $AA_{\mathcal{P}_{15}}$ is $LabArg$, where $\texttt{in}(LabArg) = \{A_{12}\}$, $\texttt{out}(LabArg) = \{A_5, A_{10}\}$, and $\texttt{undec}(LabArg)$ consists of all other arguments. $AA_{\mathcal{P}_{15}}$ has two SCUP w.r.t. its only preferred argument labelling $LabArg$, namely $\{A_6, A_7, A_8, A_9\}$ and $\{A_{11}\}$. The conclusions of arguments in these SCUPs coincide with the SCUPs of $\mathcal{P}_{15}$, i.e. $\{u, w, z\}$ and $\{p\}$ (see Example 7.2). Furthermore, the SCUP $\{A_6, A_7, A_8, A_9\}$ consists of the arguments with conclusion u, w, and z that are labelled $\texttt{undec}$ by $LabArg$. Note that argument A_{10}, whose conclusion is u, is not part of the SCUP since it is labelled $\texttt{out}$ by $LabArg$.

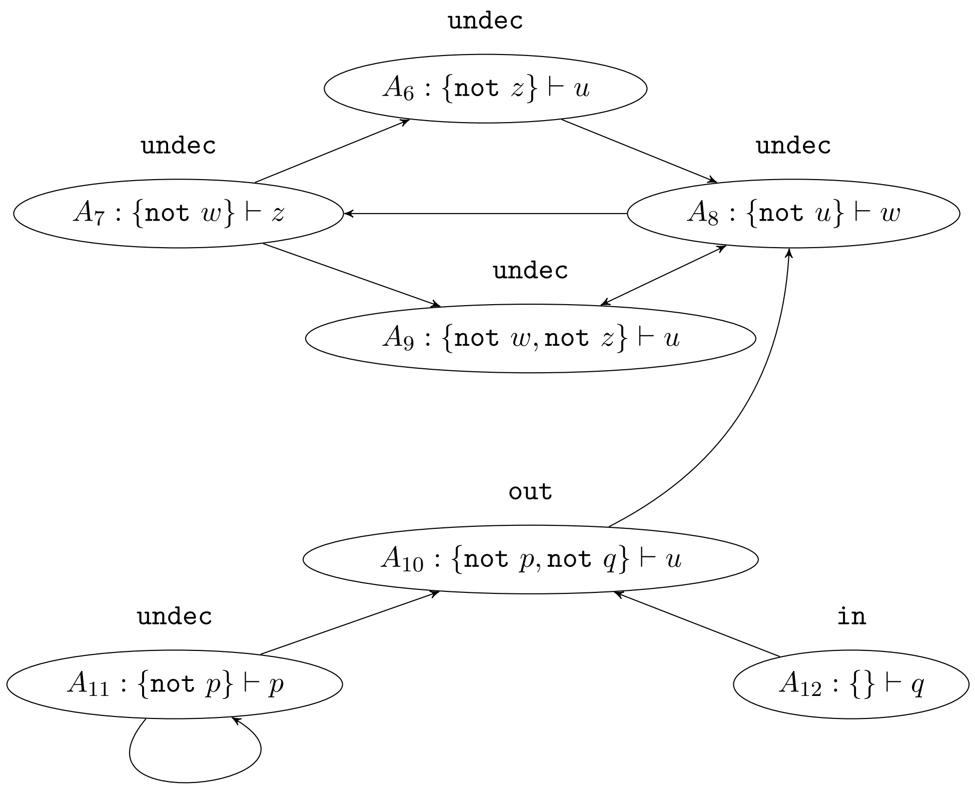


Figure 7.3: The translated AA framework of $\mathcal{P}_{15}$ from Example 7.1 (without assumption-arguments) and its only complete argument labelling.

7.2.3 Properties of SCUPs of a Logic Program

Due to these correspondence results, we can deduce some properties of SCUPs of a logic program from our results about SCUPs of AA frameworks. One such property is that every SCUP of a logic program comprises an odd-length negative dependency cycle.

Theorem 7.4. *Let S be a SCUP w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$. Then there exists an odd-length negative dependency cycle $a_0, \ldots a_n$ such that for all a_i $(0 \leq i \leq n)$ it holds that $a_i \in S$.*

Proof. Let $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ be the translated AA framework of $\mathcal{P}$ and $LabArg$ the corresponding preferred argument labelling of $\langle \mathcal{T}, \mathcal{F} \rangle$ in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$. By Theorem 7.3, $Args = \{Asms \vdash a \in \texttt{undec}(LabArg) \mid a \in S\}$ is a SCUP w.r.t. $LabArg$ of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$. Then by Proposition 6.28, there exists an odd-length cycle $\mathscr{C} \subseteq Args$. That is, there exists a path $Asms_0 \vdash a_0, Asms_1 \vdash a_1, Asms_m \vdash a_m$ of arguments in $Args$ such that $Asms_0 \vdash a_0 = Asms_m \vdash a_m$ with $m \geq 0$ and $Asms_i \vdash a_i$ attacks $Asms_{i+1} \vdash a_{i+1}$ $(0 \leq i \leq m-1)$. By Mod2LabArg from Section 4.4 it holds that $\nexists \texttt{not}\ x \in Asms_i$ such that $x \in \mathcal{T}$. Therefore, $(a_i, a_{i+1}) \in E$ in the negative undefined dependency graph (V, E) of $\mathcal{P}$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$. If for all $i \neq j$ $(0 \leq j \leq m-1)$ it holds that $a_i \neq a_j$, then $a_0, \ldots, a_m$ is a negative dependency path such that m is odd and $a_i \in S$. Else, assume that $\exists j \neq i$ such that $a_i = a_j$ with $i < j$ (i.e. two arguments in the odd-length cycle of arguments have the same conclusion). Then either $j - i$ is odd, so $a_i, \ldots, a_j$ is an odd-length negative dependency path with all atoms in S, or $j - i$ is even, so $a_0, \ldots, a_i, a_{j+1}, \ldots a_m$ is an odd-length negative dependency path with all atoms in S. □

Arguably the most important property of SCUPs of a logic program is that they indeed characterise parts of a logic program that are *responsible* that the logic program has no (2-valued) stable models. To this end, we show that the set of *responsible clauses*, i.e. clauses that are used to derive the atoms in a SCUP (and which are responsible that the truth value of the atom is U as previously explained), has no 2-valued stable model.

Theorem 7.5. *Let S be a SCUP w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ of $\mathcal{P}$ and let $\mathcal{P}_S = \{r \mid r \in \mathcal{P}' : \mathcal{P}' \subseteq \mathcal{P}$ is a minimal set (w.r.t. $\subseteq$) s.t. $\mathcal{P}' \cup \Delta \vdash_{MP} a, a \in S, \nexists \texttt{not}\ x \in \Delta$ with $x \in \mathcal{T}\}$. Then $\mathcal{P}_S$ has no (2-valued) stable model.*

Proof. Let $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$ be the translated AA framework of $\mathcal{P}$ and $LabArg$ the corresponding preferred argument labelling of $\langle \mathcal{T}, \mathcal{F} \rangle$ in $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$. By Theorem 7.3, $Args = \{Asms \vdash a \in \texttt{undec}(LabArg) \mid a \in S\}$ is a SCUP w.r.t. $LabArg$ of $\langle Ar_{\mathcal{P}}, Att_{\mathcal{P}} \rangle$.
We first note that $\mathcal{P}_S$ consists of all clauses necessary to derive the arguments in $Args$ (but there may be more arguments derivable from $\mathcal{P}_S$ which are not in $Args$) since by Mod2LabArg from Section 4.4 for all arguments $Asms \vdash a \in Args$ it holds that $\nexists \texttt{not}\ x \in Asms$ such that $x \in \mathcal{T}$. Thus, for $\langle Ar_{\mathcal{P}_S}, Att_{\mathcal{P}_S} \rangle$ it holds that $Args \subseteq Ar_{\mathcal{P}_S}$.
Let $Asms_a \vdash a \in Ar_{\mathcal{P}_S} \setminus Args$. We show that $Asms \vdash a$ does not attack $Args$: Since all clauses used to construct $Asms_a \vdash a$ are also used to construct some argument in

$Args$, it follows that $\forall \texttt{not } b \in Asms_a$, $\exists Asms_c \vdash c \in Args$ with $\texttt{not } b \in Asms_c$. Assume that $Asms_a \vdash a$ attacks $Args$. Then $Asms_a \vdash a \in \texttt{out}(LabArg)$ since $Args$ is a SCUP w.r.t. $LabArg$. This means that $\exists Asms_x \vdash x \in \texttt{in}(LabArg)$ which attacks $Asms_a \vdash a$, so $\texttt{not } x \in Asms_a$. Thus, $\exists Asms_c \vdash c \in Args$ such that $\texttt{not } x \in Asms_c$, so $Asms_c \vdash c \in \texttt{out}(LabArg)$. Contradiction since $Args \subseteq \texttt{undec}(LabArg)$. Therefore, $Asms_a \vdash a$ does not attacks $Args$.
Assume that $\mathcal{P}_S$ has a 2-valued stable model $\langle \mathcal{T}', \mathcal{F}' \rangle$. By Theorem 4.19, $LabArg' = \texttt{Mod2LabArg}(\langle \mathcal{T}', \mathcal{F}' \rangle)$ is a stable argument labelling of $\langle Ar_{\mathcal{P}_S}, Att_{\mathcal{P}_S} \rangle$. Since no arguments in $Ar_{\mathcal{P}_S} \setminus Args$ attack $Args$, it follows that $LabArg' \downarrow_{Args}$ is a stable argument labelling of $Args$. Contradiction since by Proposition 6.19 the only complete argument labelling of $\texttt{undec}(LabArg)$ labels all arguments as $\texttt{undec}$, so by the SCC-recursiveness of the complete semantics [BGG05] it holds that the only complete argument labelling of $Args$ labels all arguments as $\texttt{undec}$. In other words, $Args$ has no stable argument labelling. Thus, $\mathcal{P}_S$ has no (2-valued) stable model. □

Example 7.4. Consider again the logic program $\mathcal{P}_{15}$ from Example 7.1 and the two SCUPs w.r.t. the 3-valued M-stable model $\langle \{q\}, \emptyset \rangle$, namely $\{u, w, z\}$ and $\{p\}$. For the first of the two SCUPs, the set of responsible clause $\mathcal{P}_{15\{u,w,z\}}$ is:

$$\begin{aligned}
\{u &\leftarrow \texttt{not } w, \texttt{not } z;\\
u &\leftarrow \texttt{not } z;\\
w &\leftarrow \texttt{not } u;\\
z &\leftarrow \texttt{not } w \}
\end{aligned}$$

This logic program has no 2-valued stable models. Similarly, the set of responsible clauses of the second SCUP has no 2-valued stable models.

It follows that the set of responsible clauses with respect to SCUPs definitely have to be revised in order to obtain a 2-valued stable model of the overall logic program.

Note that we here only *characterise* inconsistency of logic programs without explicit negation. Revisions and debugging for transforming an inconsistent logic program into a consistent ones is left for future work.

7.3 Inconsistency in Logic Programs with Explicit Negation

We now investigate inconsistency of logic programs that may comprise explicitly negated atoms. From here onwards, and if not stated otherwise, we assume as given an inconsistent logic program $\mathcal{P}$ and its translated logic program $\mathcal{P}'$, where a', a'_i, a are the translated literals of $\neg a$, $\neg a_i$, and a, respectively (as explained in Section 2.3.4).

We first show how to identify in which way a logic program is inconsistent, i.e. if its only answer set is the set of all literals or if it has no answer sets at all, assuming that we only know what an answer set solver gives us, i.e. that the logic program is inconsistent.

This identification is based on whether or not the logic program has a well-founded model, which can be computed in polynomial time [VRS91]. Our results show that even though a logic program can only be inconsistent in two ways, in fact there are three different inconsistency cases, which arise due to different reasons. The three inconsistency cases are:

- $\mathcal{P}$ has no well-founded model and
   1. the only answer set of $\mathcal{P}$ is $Lit_{\mathcal{P}}$;
   2. $\mathcal{P}$ has no answer sets.
- $\mathcal{P}$ has a well-founded model and
   3. $\mathcal{P}$ has no answer sets.

In the following, we prove that these three cases are the only ones, and characterise them in more detail.

7.3.1 Inconsistency Cases 1 and 2

We start by illustrating the first inconsistency case.

Example 7.5. Let $\mathcal{P}_{17}$ be the following logic program:

$$\begin{aligned} \{\ & p \leftarrow q; \\ & u \leftarrow \texttt{not}\ t; \\ & q \leftarrow r, s; \\ & t \leftarrow \texttt{not}\ u; \\ & r \leftarrow ; \\ \neg & p \leftarrow ; \\ & s \leftarrow \} \end{aligned}$$

$\mathcal{P}_{17}$ has no well-founded model and its only answer set is $Lit_{\mathcal{P}_{17}}$, so $\mathcal{P}_{17}$ falls into inconsistency case 1. The reason that the only answer set is $Lit_{\mathcal{P}_{17}}$ is that for any $S \subseteq Lit_{\mathcal{P}_{17}}$ satisfying the conditions of an answer set, $s, r, \neg p \in S$, so $q, p \in S$, and thus S contains the complementary literals p and $\neg p$. Note that NAF literals do not play any role in the inconsistency of $\mathcal{P}_{17}$; an atom and its explicitly negated atom, both strictly derivable, are responsible for the inconsistency.

The observations in Example 7.5 agree with a well-known result about logic programs whose only answer set it the set of all literals (Proposition 6.7 in [Ino93]).

Lemma 7.6. *The only answer set of $\mathcal{P}$ is $Lit_{\mathcal{P}}$ if and only if $\exists a \in \mathcal{HB}_{\mathcal{P}}$ such that $\mathcal{P} \vdash_{MP} a$ and $\mathcal{P} \vdash_{MP} \neg a$.*

Next, we illustrate the second inconsistency case.

Example 7.6. Let $\mathcal{P}_{18}$ be the following logic program:

$$\begin{aligned}
\{\ q &\leftarrow \texttt{not}\ r;\\
\neg q &\leftarrow \neg s, \texttt{not}\ p;\\
r &\leftarrow \texttt{not}\ \neg t;\\
\neg s &\leftarrow\ ;\\
\neg t &\leftarrow\ \}
\end{aligned}$$

$\mathcal{P}_{18}$ has no well-founded model and no answer sets, so $\mathcal{P}_{18}$ falls into inconsistency case 2. The reason that $\mathcal{P}_{18}$ has no answer sets is an interplay of explicit negation and NAF: for any $S \subseteq Lit_{\mathcal{P}_{18}}$ satisfying the conditions of an answer set, $\neg t, \neg s \in S$, and thus $r \leftarrow \texttt{not}\ \neg t$ is always deleted in ${\mathcal{P}_{18}}^S$ and both $q \leftarrow$ and $\neg q \leftarrow \neg s$ are always part of ${\mathcal{P}_{18}}^S$. Consequently, for any such S it holds that $q, \neg q \in \mathcal{AS}({\mathcal{P}_{18}}^S)$, meaning that the only possible answer set is $Lit_{\mathcal{P}_{18}}$. However, since $r, p, \neg t \in Lit_{\mathcal{P}_{18}}$ the reduct will only consist of $\neg t \leftarrow$ and $\neg s \leftarrow$, so that $\mathcal{AS}({\mathcal{P}_{18}}^{Lit_{\mathcal{P}_{18}}}) = \{\neg t, \neg s\}$, which does not contain complementary literals. Consequently, $\mathcal{P}_{18}$ has no answer sets at all.

Even though both in $\mathcal{P}_{17}$ and in $\mathcal{P}_{18}$ the inconsistency arises due to complementary literals, the difference lies in their derivations: in $\mathcal{P}_{17}$, complementary literals are strictly derivable, whereas in $\mathcal{P}_{18}$, the complementary literals are defeasibly derivable, i.e. not only explicit negation but also NAF is involved in the derivations of literals causing the inconsistency.

The following Theorem characterises inconsistency cases 1 and 2 in terms of derivations of complementary literals.

Theorem 7.7. *If $\mathcal{P}$ has no well-founded model, then*

1. *the only answer set of $\mathcal{P}$ is $Lit_{\mathcal{P}}$ if and only if $\exists a \in \mathcal{HB}_{\mathcal{P}}$ such that $\mathcal{P} \vdash_{MP} a$ and $\mathcal{P} \vdash_{MP} \neg a$;*

2. *$\mathcal{P}$ has no answer sets if and only if $\nexists a \in \mathcal{HB}_{\mathcal{P}}$ such that $\mathcal{P} \vdash_{MP} a$ and $\mathcal{P} \vdash_{MP} \neg a$.*

Proof. From Lemma 7.6. □

7.3.2 Inconsistency Case 3

The following example illustrates the third inconsistency case.

Example 7.7. Let $\mathcal{P}_{19}$ be the following logic program:

$$\begin{aligned}\{\ r &\leftarrow \texttt{not}\ s;\\ s &\leftarrow \texttt{not}\ r;\\ q &\leftarrow \texttt{not}\ s;\\ \neg q &\leftarrow \texttt{not}\ s;\\ p &\leftarrow \texttt{not}\ r;\\ \neg p &\leftarrow \texttt{not}\ r\ \}\end{aligned}$$

The well-founded model of $\mathcal{P}_{19}$ is $\langle\emptyset, \emptyset\rangle$, but $\mathcal{P}_{19}$ has no answer sets. Thus, it falls into inconsistency case 3. The reason that $\mathcal{P}_{19}$ has no answer sets is an interplay of explicit negation and NAF similar to Example 7.6. From the first two clauses, it follows that any potential answer set $S \subseteq Lit_{\mathcal{P}_{19}}$ cannot contain both s and r. If $r \notin S$, then $p, \neg p \in S$; if $s \notin S$, then $q, \neg q \in S$, and thus the only possible answer set is $Lit_{\mathcal{P}_{19}}$. However, ${\mathcal{P}_{19}}^{Lit_{\mathcal{P}_{19}}}$ does not comprise any clauses, so $\mathcal{AS}({\mathcal{P}_{19}}^{Lit_{\mathcal{P}_{19}}}) = \emptyset$, which does not contain complementary literals. Thus, $\mathcal{P}_{19}$ has no answer sets. As in $\mathcal{P}_{18}$ (see Example 7.6), the inconsistency is due to defeasibly derivable complementary literals, but in contrast to $\mathcal{P}_{18}$ here the derivations of complementary literals involve NAF literals that form an even-length negative dependency cycle, namely s and r.

Theorem 7.8 characterises inconsistency case 3.

Theorem 7.8. *If $\mathcal{P}$ has a well-founded model, then $\mathcal{P}$ has no answer sets.*

Proof. Assume that $\exists a \in \mathcal{HB}_{\mathcal{P}}$ s.t. $\mathcal{P} \vdash_{MP} a$ and $\mathcal{P} \vdash_{MP} \neg a$. Then a and a' are in the well-founded model of $\mathcal{P}'$ (by the alternating fixpoint definition of well-founded models [Van93]) and thus a and $\neg a$ are contained in the corresponding well-founded model of $\mathcal{P}$, so $\mathcal{P}$ has no well-founded model (contradiction). Thus, $\nexists a \in \mathcal{HB}_{\mathcal{P}}$ s.t. $\mathcal{P} \vdash_{MP} a$ and $\mathcal{P} \vdash_{MP} \neg a$, so by Lemma 7.6 it is not the case that the only answer set of $\mathcal{P}$ is $Lit_{\mathcal{P}}$. Consequently, $\mathcal{P}$ has no answer sets. □

In summary, if $\mathcal{P}$ has no well-founded model, then its only answer set is $Lit_{\mathcal{P}}$ – caused by explicit negation – or it has no answer sets – caused by the interplay of explicit negation and NAF. If $\mathcal{P}$ has a well-founded model, then it has no answer sets – caused by the interplay of explicit negation and NAF.

7.4 Characterising Culprits

In the examples in the previous section, we already briefly discussed that the reasons for the inconsistency are different in the three inconsistency cases: either only explicit negation or the interplay of explicit negation and NAF. In this section, we show that inconsistency case 3 can, in fact, be further split into two sub-cases: one where the interplay of explicit negation and NAF is responsible as seen in Example 7.7 (case 3a), and one where only NAF is responsible for the inconsistency (case 3b), corresponding to the inconsistency of logic programs without explicit negation. Furthermore, we characterise the different reasons of inconsistency in more detail in terms of "culprit" sets, which are sets of literals included in the well-founded (cases 1,2) or 3-valued M-stable (case 3b) model of $\mathcal{P}$, or in the answer sets of $\mathcal{P}'$ (case 3a). In other words, culprits can be found in "weaker" models.

Definition 7.3 (Culprit Set)**.** Let $\langle \mathcal{T}'_w, \mathcal{F}'_w \rangle$ be the well-founded model of $\mathcal{P}'$, $S'_1, \ldots, S'_n$ ($n \geq 0$) its answer sets, and $\langle \mathcal{T}'_M, \mathcal{F}'_M \rangle$ one of its 3-valued M-stable models with $\mathcal{U}'_M$ the set of undefined atoms.

- If $\mathcal{P}$ has no well-founded model, then
  - $\{a, \neg a\}$ is a *culprit set* of $\mathcal{P}$ if and only if $a, a' \in \mathcal{T}'_w$ and a and a' are strictly derivable from $\mathcal{P}'$ (**case 1**);
  - $\{a, \neg a\}$ is a *culprit set* of $\mathcal{P}$ if and only if $a, a' \in \mathcal{T}'_w$ and one of them is defeasibly derivable from $\mathcal{P}'$ and the other one is derivable from $\mathcal{P}'$ (**case 2**).
- If $\mathcal{P}$ has a well-founded model and
  - $\mathcal{P}'$ has n answer sets ($n \geq 1$), then $\{a_1, \neg a_1, \ldots, a_n, \neg a_n\}$ is a *culprit set* of $\mathcal{P}$ if and only if $\forall a_i, \neg a_i$ ($1 \leq i \leq n$): $a_i, a'_i \in S'_i$ and one of them is defeasibly derivable from $\mathcal{P}'$ and the other one is derivable from $\mathcal{P}'$ (**case 3a**);
  - $\mathcal{P}'$ has no answer sets, then C is a *culprit set* of $\mathcal{P}$ if and only if there exists an odd-length negative dependency cycle $a_0, \ldots, a_n$ in $\mathcal{P}'$ and a SCUP S w.r.t. $\langle \mathcal{T}'_M, \mathcal{F}'_M \rangle$ such that for all a_i ($0 \leq i \leq n$) it holds that $a_i \in S$, and C consists of the original literals of the translated literals $a_0, \ldots, a_n$ (**case 3b**).

We now show that for every inconsistency case at least one culprit set exists.

7.4.1 Inconsistency Case 1

Example 7.8. The well-founded model of the translated logic program $\mathcal{P}_{17}{}'$ (see $\mathcal{P}_{17}$ in Example 7.5) is $\langle \{p, p', q, r, s\}, \emptyset \rangle$. It thus holds that $p, p' \in \mathcal{T}'_w$ and both of them are strictly derivable from $\mathcal{P}'$. Thus, $\{p, \neg p\}$ is a culprit set of $\mathcal{P}_{17}$, which confirms our observation that $Lit_{\mathcal{P}_{17}}$ is the only answer set of $\mathcal{P}_{17}$ because every potential answer set contains both p and $\neg p$ (see Example 7.5). Note that it is not only the literals in the culprit set which characterise this inconsistency case, it is the derivation of the literals, i.e. that both literals are strictly derivable.

Theorem 7.9 states the existence of a culprit set in inconsistency case 1.

Theorem 7.9. *Let $\mathcal{P}$ have no well-founded model and let its only answer set be $Lit_{\mathcal{P}}$. Then $\mathcal{P}$ has a case 1 culprit set $\{a, \neg a\}$.*

Proof. By Lemma 7.6, $\exists a, a' \in \mathcal{HB}_{\mathcal{P}'}$ s.t. $\mathcal{P}' \vdash_{MP} a$ and $\mathcal{P}' \vdash_{MP} a'$. By definition of well-founded model (as an alternating fixpoint [Van93]), $a, a' \in \mathcal{T}'_w$ where $\langle \mathcal{T}'_w, \mathcal{F}'_w \rangle$ is the well-founded model of $\mathcal{P}'$. By Definition 7.3, $\{a, \neg a\}$ is a case 1 culprit set. □

7.4.2 Inconsistency Case 2

Example 7.9. The well founded model of $\mathcal{P}'_{18}$ (see $\mathcal{P}_{18}$ in Example 7.6) is $\langle \{q, q', s', t'\}, \{p, r\} \rangle$. It holds that $q, q' \in \mathcal{T}'_w$ and here even both of them are defeasibly derivable. Thus, $\{q, \neg q\}$ is a culprit set of $\mathcal{P}_{18}$, which confirms our observation that the reason for the inconsistency of $\mathcal{P}_{18}$ is that every potential answer set contains both q and $\neg q$, but $Lit_{\mathcal{P}_{18}}$ is not an answer set due to the NAF literals involved in the derivations of q and $\neg q$. Note that even though the culprit sets of $\mathcal{P}_{17}$ and $\mathcal{P}_{18}$ are very similar – both consist of complementary literals – the difference lies in the derivations of the literals in the culprit set: here, the literals are not both strictly derivable, so the reason for the inconsistency is both that complementary literals are derivable (explicit negation) as well as that their derivations involve NAF literals.

Theorem 7.10 proves the existence of a culprit set in inconsistency case 2.

Theorem 7.10. *Let $\mathcal{P}$ have no well-founded model and no answer sets. Then $\mathcal{P}$ has a case 2 culprit set $\{a, \neg a\}$.*

Proof. Let $\langle \mathcal{T}'_w, \mathcal{F}'_w \rangle$ be the well-founded model of $\mathcal{P}'$. Since $\mathcal{P}$ has no well-founded model, $\mathcal{T}'_w$ must contain some a, a'. Since every answer set is a superset of the well-founded model (Corollary 5.7 in [VRS91]), every potential answer set of $\mathcal{P}$ contains a and $\neg a$, meaning that the only possible answer set is $Lit_{\mathcal{P}}$. From the assumption that $\mathcal{P}$ has no answer sets, we can conclude that $\mathcal{AS}(\mathcal{P}^{Lit_{\mathcal{P}}})$ does not contain a and $\neg a$. Thus, all of the rules needed for the derivation of either a or $\neg a$ are deleted in $\mathcal{P}^{Lit_{\mathcal{P}}}$, meaning that a or $\neg a$ is defeasibly derivable. Trivially, the other literal is also derivable as $a, a' \in \mathcal{T}'_w$. Then by Definition 7.3, $\{a, \neg a\}$ is a case 2 culprit set of $\mathcal{P}$. □

7.4.3 Inconsistency Case 3a

Example 7.10. $\mathcal{P}'_{19}$ (see $\mathcal{P}_{19}$ in Example 7.7) has two answer sets $S'_1 = \{q, q', r\}$ and $S'_2 = \{p, p', s\}$, so $\mathcal{P}_{19}$ falls into inconsistency case 3a. q, q', p, p' are all defeasibly derivable from $\mathcal{P}_{19}'$ and thus $\{q, \neg q, p, \neg p\}$ is a culprit set of $\mathcal{P}_{19}$. This confirms our observation that the reason for the inconsistency of $\mathcal{P}_{19}$ is that the two potential answer sets both contain complementary literals, but that $Lit_{\mathcal{P}_{19}}$ is not an answer set due to the NAF literals involved in the derivations of the complementary literals. Thus, as in Example 7.9, the inconsistency is due to the interplay of explicit negation and NAF with the difference of the even-length cycle described in Example 7.6. Due to this difference in the derivations, here the well-founded model of the translated logic program does not provide any information about culprits (as it is $\langle \emptyset, \emptyset \rangle$), but the answer sets do.

Theorem 7.11 states the existence of a culprit set in inconsistency case 3a.

Theorem 7.11. *Let $\mathcal{P}$ have a well-founded model and let $\mathcal{P}'$ have $n \geq 1$ answer sets. Then, $\mathcal{P}$ has a case 3a culprit set $\{a_1, \neg a_1, \ldots, a_n, \neg a_n\}$.*

Proof. By Theorem 7.8, $\mathcal{P}$ has no answer sets, so all $S_i \subseteq \mathit{Lit}_{\mathcal{P}}$ with $S_i = \mathcal{AS}(\mathcal{P}^{S_i})$ contain complementary literals a_i and $\neg a_i$, but $\mathcal{AS}(\mathcal{P}^{\mathit{Lit}_{\mathcal{P}}})$ does not contain complementary literals. Thus, all S_i' with $S_i' = \mathcal{AS}(\mathcal{P}'^{S_i'})$ contain a_i and a_i', so a_i and a_i' must be derivable from $\mathcal{P}'$. Assume that $\mathcal{P}' \vdash_{MP} a_i$ and $\mathcal{P}' \vdash_{MP} a_i'$. Then by Lemma 7.6 the only answer set of $\mathcal{P}$ is $\mathit{Lit}_{\mathcal{P}}$ (contradiction). Thus, at least one of a_i and a_i' is defeasibly derivable from $\mathcal{P}'$. Then by Definition 7.3, $\{a_1, \neg a_1, \ldots, a_n, \neg a_n\}$ is a case 3a culprit set of $\mathcal{P}$. $\square$

7.4.4 Inconsistency Case 3b

Example 7.11. Let $\mathcal{P}_{20}$ be the following logic program:

$$\begin{aligned}
\{\ s &\leftarrow w;\\
\neg u &\leftarrow \texttt{not}\ v;\\
w &\leftarrow \texttt{not}\ t;\\
v &\leftarrow \texttt{not}\ t, \texttt{not}\ y;\\
t &\leftarrow \neg x;\\
x &\leftarrow\ ;\\
\neg x &\leftarrow \texttt{not}\ \neg u;\\
y &\leftarrow \texttt{not}\ x\ \}
\end{aligned}$$

$\mathcal{P}_{20}$ has a well-founded model and $\mathcal{P}_{20}'$ has no answer sets, so $\mathcal{P}_{20}$ falls into inconsistency case 3b. The only 3-valued M-stable model of $\mathcal{P}_{20}'$ is $\langle\{x\}, \{y\}\rangle$, where $\mathcal{U}_M' = \{s, t, u', v, w, x'\}$. The negative undefined dependency graph of $\mathcal{P}_{20}'$ w.r.t. $\langle\{x\}, \{y\}\rangle$ is illustrated in Figure 7.4. We note that u', v, t, u' is an odd-length negative dependency cycle contained in a SCUP w.r.t. $\langle\{x\}, \{y\}\rangle$, namely the SCUP $\{u', v', t\}$. Thus, $C = \{\neg u, v, t\}$ is a culprit set of $\mathcal{P}_{20}$. This example shows that in inconsistency case 3b the inconsistency is due to NAF on its own; explicit negation plays no role.

$$\begin{tikzcd}
x' & u' \arrow[l] \arrow[rr, bend left] & & v \arrow[dl, bend left] \\
& t & t \arrow[l] \arrow[ul, bend left] \arrow[r] & w
\end{tikzcd}$$

Figure 7.4: The negative undefined dependency graph of $\mathcal{P}_{20}'$ w.r.t. $\langle\{x\}, \{y\}\rangle$ (see Example 7.11).

In inconsistency case 3b, the translated logic program $\mathcal{P}'$ has no answer sets. Since $\mathcal{P}'$ is a logic program without explicitly negated atoms, the reason for the non-existence of answer sets of $\mathcal{P}'$ – and thus of $\mathcal{P}$ – can be characterised as in Section 7.2 for logic programs without explicit negation. Theorem 7.12 not only characterises culprit sets in inconsistency case 3b, but also states how to find a culprit set.

Theorem 7.12. *Let $\mathcal{P}$ have a well-founded model and let $\mathcal{P}'$ have no answer sets. Let $\langle \mathcal{T}'_M, \mathcal{F}'_M \rangle$ be a 3-valued M-stable model of $\mathcal{P}'$ with $\mathcal{U}'_M$ the set of undefined atoms. Then, for any $a_1 \in \mathcal{U}'_M$ there exists a negative dependency path $a_1, \dots, a_n, b_1, \dots, b_m$ such that the set C consisting of the original literals of the translated literals $b_1, \dots, b_m$ is a case 3b culprit set of $\mathcal{P}$.*

Proof. By Proposition 7.1 there exists a SCUP w.r.t. $\langle \mathcal{T}'_M, \mathcal{F}'_M \rangle$ and by Theorem 7.4 there exists an odd-length negative dependency cycle C in the SCUP. Furthermore, for all $a_1 \in \mathcal{U}$ it holds that either 1) a_1 is an initial SCC of the negative undefined dependency graph or 2) a_1 is not an initial SCC of the negative undefined dependency graph. In the first case a_1 is part of a SCUP that comprises an odd-length cycle $b_0, \dots b_m$. Since the SCUP is strongly connected, there exists a path $a_1, \dots, a_n, b_1, \dots b_m$ ($a_n = b_0$) such that the set C consisting of the original literals of the translated literals $b_1, \dots, b_m$ is a case 3b culprit set of $\mathcal{P}$. In the second case, since a_1 is not an initial SCC of the negative undefined dependency graph, it is part of another SCC of the negative undefined dependency graph, which consequently is negatively dependent on some initial SCC over a path of atoms from $\mathcal{U}$. That is, there exists a path $a_1, \dots, a_n, b_1, \dots b_m$ such that $a_n, b_1, \dots, b_m$ is an odd-length cycle in an initial SCC. Thus, he set C consisting of the original literals of the translated literals $b_1, \dots, b_m$ is a case 3b culprit set of $\mathcal{P}$. □

Note that in each of the three inconsistency cases discussed in Section 7.3, the translated logic program $\mathcal{P}'$ might or might not have answer sets. However, regarding culprit sets this distinction only makes a difference in inconsistency case 3.

It follows directly from Theorems 7.9, 7.10, 7.11, and 7.12 that the culprit sets we identified are indeed responsible for the inconsistency, i.e. if no culprit sets exist then the logic program is consistent.

Corollary 7.13. *Let $\mathcal{P}$ be a (possibly consistent) logic program. If there exists no culprit set of inconsistency cases 1, 2, 3a, or 3b of $\mathcal{P}$, then $\mathcal{P}$ is consistent.*

7.5 Explaining Culprits

As pointed out in the previous sections, even though we identify culprits as sets of literals, the reason for the inconsistency is mostly the way in which these literals are derivable from the logic program. In order to make the reason for the inconsistency more understandable to the user, we now show how explanations of the inconsistency can be constructed in terms of trees whose nodes are derivations similar to our Attack Trees from Chapter 5. In contrast to Chapter 5, we do not construct the translated ABA or AA framework, but instead define derivations based on the logic program with respect to a 3-valued interpretation $\langle \mathcal{T}, \mathcal{F} \rangle$.

We call a derivation *true* with respect to $\langle \mathcal{T}, \mathcal{F} \rangle$ if all NAF literals $\texttt{not}\ k$ used in the derivation are true with respect to the interpretation in question, i.e. the literals k are false in the interpretation. We call a derivation *false* with respect to the interpretation if there exists a NAF literal $\texttt{not}\ k$ used in the derivation that is false with respect to the interpretation, i.e. k is true in the interpretation.

Definition 7.4 (True/False Derivation)**.** Let $\langle \mathcal{T}, \mathcal{F} \rangle$ be a 3-valued interpretation of $\mathcal{P}$, $l \in Lit_{\mathcal{P}}$, and $\Delta \subseteq NAF_{Lit_{\mathcal{P}}}$.

1. $\mathcal{P} \cup \Delta \vdash_{MP} l$ is a *true derivation* of l w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ if $\forall \texttt{not}\ k \in \Delta : k \in \mathcal{F}$.

2. $\mathcal{P} \cup \Delta \vdash_{MP} l$ is a *false derivation* of l w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ if $\exists \texttt{not}\ k \in \Delta : k \in \mathcal{T}$.

Example 7.12. Consider $\mathcal{P}_{20}$ from Example 7.11. $\mathcal{P}_{20} \cup \{\texttt{not}\ t, \texttt{not}\ y\} \vdash_{MP} v$ is a true derivation w.r.t. $\langle \{s\}, \{t, y\} \rangle$, a false derivation w.r.t. $\langle \{s, t\}, \{y\} \rangle$, and neither a true nor a false derivation w.r.t. $\langle \{s\}, \{y\} \rangle$.

An explanation of inconsistency cases 1-3a illustrates why the literals in a culprit set are true in the respective 3-valued stable model $\langle \mathcal{T}, \mathcal{F} \rangle$ used to identify this culprit set, which is due to the literals' derivations. Thus, an explanation starts with a true derivation of a literal in the culprit set with respect to $\langle \mathcal{T}, \mathcal{F} \rangle$. The explanation then indicates why this derivation is true, i.e why all NAF literals $\texttt{not}\ k$ are true with respect to $\langle \mathcal{T}, \mathcal{F} \rangle$. The reason why $\texttt{not}\ k$ is true is that some derivation of k is false, i.e. a NAF literal $\texttt{not}\ m$ in a derivation of k is false with respect to $\langle \mathcal{T}, \mathcal{F} \rangle$. This, in turn, is explained in terms of why m is true with respect to $\langle \mathcal{T}, \mathcal{F} \rangle$, and so on.

Definition 7.5 (Explanation of a Literal w.r.t. a Model)**.** Let $\langle \mathcal{T}, \mathcal{F} \rangle$ be a 3-valued stable model of $\mathcal{P}$ and let $l \in Lit_{\mathcal{P}}$. An *explanation* of l w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ is a tree such that:

1. every node holds either a true or a false derivation w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$;

2. the root holds a true derivation of l w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$;

3. for every node N holding a true derivation $\mathcal{P} \cup \Delta \vdash_{MP} k$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ and for every $\texttt{not}\ m \in \Delta$: every false derivation of m w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ is held by a child of N;

4. for every node N holding a false derivation $\mathcal{P} \cup \Delta \vdash_{MP} k$ w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$: N has exactly one child holding a true derivation of some m w.r.t. $\langle \mathcal{T}, \mathcal{F} \rangle$ such that $\texttt{not}\ m \in \Delta$;

5. there are no other nodes except those given in 1-4.

Since culprit sets are determined with respect to different 3-valued stable models in the different inconsistency cases, explanations are constructed with respect to these different models, too.

Definition 7.6 (Inconsistency Explanation - Cases 1 and 2)**.** Let $\mathcal{P}$ have no well-founded model and let $\langle \mathcal{T}'_w, \mathcal{F}'_w \rangle$ be the well-founded model of $\mathcal{P}'$. Let $\{a, \neg a\}$ be a culprit set of $\mathcal{P}$. A *translated inconsistency explanation* of $\mathcal{P}$ consists of an explanation of a w.r.t. $\langle \mathcal{T}'_w, \mathcal{F}'_w \rangle$ and an explanation of a' w.r.t. $\langle \mathcal{T}'_w, \mathcal{F}'_w \rangle$. An *inconsistency explanation* of $\mathcal{P}$ is derived by replacing every translated literal in the translated inconsistency explanation by its respective original literal.

Since explanations are trees, they can be easily visualised, as shown for $\mathcal{P}_{18}$ (see Examples 7.6 and 7.9) in Figure 7.5.

$$\begin{tikzcd}
\mathcal{P}_{18} \cup \{\texttt{not}\ r\} \vdash_{MP} q & \mathcal{P}_{18} \cup \{\texttt{not}\ p\} \vdash_{MP} \neg q \\
\mathcal{P}_{18} \cup \{\texttt{not}\ \neg t\} \vdash_{MP} r \arrow[u] & \\
\mathcal{P}_{18} \cup \emptyset \vdash_{MP} \neg t \arrow[u] &
\end{tikzcd}$$

Figure 7.5: The inconsistency explanation of $\mathcal{P}_{18}$ (see Examples 7.6, 7.9).

Definition 7.7 (Inconsistency Explanation - Case 3a)**.** Let $\mathcal{P}$ have a well-founded model and let $S'_1, \ldots, S'_n$ ($n \geq 1$) be the answer sets of $\mathcal{P}'$. Let $\{a_1, \neg a_1, \ \ldots, a_n, \neg a_n\}$ be a culprit set of $\mathcal{P}$. A *translated inconsistency explanation* of $\mathcal{P}$ consists of an explanation of all a_i and a'_i ($1 \leq i \leq n$) w.r.t. $\langle S'_i, (\mathcal{HB}_{\mathcal{P}'} \backslash S'_i) \rangle$. An *inconsistency explanation* of $\mathcal{P}$ is derived by replacing every translated literal in the translated inconsistency explanation by its respective original literal.

Figure 7.6 shows part of the inconsistency explanation of $\mathcal{P}_{19}$ (see Examples 7.7 and 7.10). It also illustrates the difference between the reason of inconsistency in $\mathcal{P}_{18}$ and $\mathcal{P}_{19}$, namely the negative dependency cycle of s and r in $\mathcal{P}_{19}$, which results in infinite trees.

$$\begin{tikzcd}
\mathcal{P}_{19} \cup \{\texttt{not}\ s\} \vdash_{MP} q & \mathcal{P}_{19} \cup \{\texttt{not}\ s\} \vdash_{MP} \neg q \\
\mathcal{P}_{19} \cup \{\texttt{not}\ r\} \vdash_{MP} s \arrow[u] & \mathcal{P}_{19} \cup \{\texttt{not}\ r\} \vdash_{MP} s \arrow[u] \\
\mathcal{P}_{19} \cup \{\texttt{not}\ s\} \vdash_{MP} r \arrow[u] & \mathcal{P}_{19} \cup \{\texttt{not}\ s\} \vdash_{MP} r \arrow[u] \\
\mathcal{P}_{19} \cup \{\texttt{not}\ r\} \vdash_{MP} s \arrow[u] & \mathcal{P}_{19} \cup \{\texttt{not}\ r\} \vdash_{MP} s \arrow[u] \\
\vdots & \vdots
\end{tikzcd}$$

Figure 7.6: Part of the inconsistency explanation of $\mathcal{P}_{19}$ explaining q and $\neg q$. The full inconsistency explanation also comprises similar explanations for p and $\neg p$.

For inconsistency case 3b, where the literals in a culprit set form an odd-length negative dependency cycle, the inconsistency explanation is constructed with respect to the set $\mathcal{U}$ rather than $\mathcal{T}$ and $\mathcal{F}$, since all literals in a culprit set are contained in $\mathcal{U}$ of a 3-valued M-stable model. The reason that a literal is undefined is that its derivation contains a NAF literal `not` k that is undefined. Then $k \in \mathcal{U}$, which again is due to its derivation containing some undefined NAF literal, and so on. Thus, an explanation of inconsistency case 3b is a tree of negative derivations with respect to $\mathcal{U}$.

Definition 7.8 (Inconsistency Explanation - Case 3b)**.** Let $\mathcal{P}$ have a well-founded model and let $\mathcal{P}'$ have no answer sets. Let $\langle \mathcal{T}'_M, \mathcal{F}'_M \rangle$ be a 3-valued M-stable model of $\mathcal{P}'$ with $\mathcal{U}'_M$ the set of undefined atoms. Let C be a culprit set of $\mathcal{P}$ and $a \in C$. A *translated inconsistency explanation* of $\mathcal{P}$ is a tree such that:

1. every node holds a false derivation w.r.t. $\langle \mathcal{U}'_M, \emptyset \rangle$;
2. the root holds a false derivation of a w.r.t. $\langle \mathcal{U}'_M, \emptyset \rangle$;
3. for every node N holding a false derivation $\mathcal{P} \cup \Delta \vdash_{MP} b$ w.r.t. $\langle \mathcal{U}'_M, \emptyset \rangle$: N has exactly one child node holding a false derivation of some m w.r.t. $\langle \mathcal{U}'_M, \emptyset \rangle$ such that $\texttt{not}\ m \in \Delta$ and $m \in C$;
4. there are no other nodes except those given in 1-3.

An *inconsistency explanation* of $\mathcal{P}$ is derived by replacing every translated literal in the translated inconsistency explanation by its respective original literal.

Figure 7.7 illustrates the inconsistency explanation of $\mathcal{P}_{20}$ (see Example 7.11), showing the odd-length cycle of derivations of literals contained in a SCUP. It also illustrates how the derivations in an inconsistency explanation can be expanded to derivation trees, which can also be done for cases 1-3a.

$$\begin{tikzcd}
\mathcal{P}_{20} \cup \{\texttt{not}\ v\} \vdash_{MP} \neg u \\
\mathcal{P}_{20} \cup \{\texttt{not}\ t, \texttt{not}\ x\} \vdash_{MP} v \arrow[u] \\
\mathcal{P}_{20} \cup \{\texttt{not}\ \neg u\} \vdash_{MP} t \arrow[u] \\
\mathcal{P}_{20} \cup \{\texttt{not}\ v\} \vdash_{MP} \neg u \arrow[u] \\
\vdots
\end{tikzcd}$$

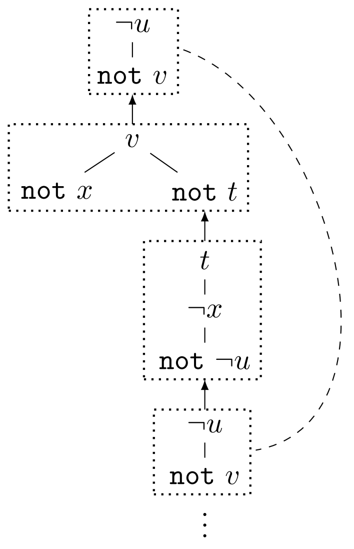


Figure 7.7: The inconsistency explanation of $\mathcal{P}_{20}$ (left) and the version where derivations are expanded to trees (right).

Note that in all our examples, the culprit set is unique. However, in general a logic program may have various culprit sets (from the same inconsistency case) resulting in various inconsistency explanations. Moreover, there may be various inconsistency explanations for a given culprit set.

7.6 Related Work

7.6.1 Logic Programs without Explicit Negation

We first compare our characterisation of SCUPs as parts that are responsible for inconsistency to related work on inconsistency of logic programs without explicit negation.

You and Yuan [YY94] show that for some logic programs (without explicit negation), namely those with a well-founded stratification, it holds that with respect to 3-valued M-stable models where $\mathcal{U} \neq \emptyset$ there exists an odd-length negative dependency cycle containing at least one atom contained in $\mathcal{U}$. Theorem 7.4 extends this result to *arbitrary* logic programs and shows that, in fact, *all* atoms in an odd-length negative dependency cycle are contained in $\mathcal{U}$.

Fages [Fag94] and Dung [Dun92] prove that a logic program that is order- or call-consistent (this implies that the logic program does not comprise an odd-length negative dependency cycle), respectively, has a stable model. It follows that if a logic program has no stable model, it comprises an odd-length negative dependency cycle. We extend these results by localising the *responsible* odd-length cycles of a logic program without stable model, showing that they are made of atoms that are contained in $\mathcal{U}$ of some 3-valued M-stable model (see Theorem 7.4) and that they are contained in an "initial" SCC of the negative undefined dependency graph. For instance, the negative dependency graph of $\mathcal{P}_{16}$ has two odd-length cycles, namely $\{p\}$ and $\{v, w, z\}$ (see Example 7.2). However, only the former is a part of a responsible set of atoms, as shown in Example 7.2.

Caminada and Sakama [CS06] show that a specific class of logic programs, called extended normal logic programs, always has an answer set, namely those where 1) clauses without NAF literals are closed under transposition, transitivity, and antecedent cleaning (body of clause does not contain the classical negation of its head), and 2) clauses with NAF in the body are "normal", i.e. the body contains as only NAF literal the NAF literal of the clause's head. It follows that if a logic program has no answer set, then it is not an extended normal logic program. In most cases, this will not be surprising, since most logic programs are not extended normal logic programs. Therefore, the work of Caminada and Sakama is in general not helpful when identifying why a logic program has no answer sets.

Dimopolous and Torres [DT96] introduce the notion of *minimal attack graph* of a logic program, which is very similar to our ABA graphs from Section 3.3.2 when constructed for the translated ABA framework of a logic program. They show that if the minimal attack graph comprises no odd-length cycles, then the logic program has at least one stable model. They also prove that if each odd-length cycle in the minimal attack graph has at least two symmetric edges or has two cords (edges between non-consecutive nodes in the cycle) whose heads are consecutive nodes of the cycle, then the logic program has at least one stable model. It follows that if a logic program has no stable model, then there exists an odd-length cycle with at most one symmetric edge and no two cords whose heads are consecutive nodes of the cycle. Therefore, Dimopolous and Torres also characterise which odd-length cycles are responsible for the non-existence of stable models. However, their characterisation is purely structural, whereas ours also has a semantic component, since we use a 3-valued M-stable model for our characterisation. Furthermore, Dimopolous and Torres do not distinguish between *responsible* and non-responsible cycles. For instance, $\mathcal{P}_{16}$ from Example 7.2 yields the minimal attack graph illustrated in Figure 7.8. There are two odd-length cycles that do satisfy the conditions stated by Dimopolous and Torres. However, only one of them corresponds to an odd-length cycle contained in a SCUP, namely the odd-length cycle $\{\texttt{not}\ p\}$, which corresponds to the SCUP $\{p\}$ of $\mathcal{P}_{16}$.

$$\begin{tikzcd}
& \{\texttt{not}\ z, \texttt{not}\ p\} \arrow[dr, bend left] & \\
\{\texttt{not}\ w\} \arrow[ur, bend left] & & \{\texttt{not}\ u\} \arrow[ll, bend left] \\
& \{\texttt{not}\ p, \texttt{not}\ q\} \arrow[ur, bend right] & \\
\{\texttt{not}\ p\} \arrow[uuur] \arrow[ur] \arrow[loop below] & & \{\} \arrow[ul]
\end{tikzcd}$$

Figure 7.8: The minimal attack graph [DT96] of $\mathcal{P}_{16}$ from Example 7.2.

Costantini [Cos06] characterises the existence of stable models in terms of appropriate assignments of the truth values T and F to atoms in odd- and even-length negative dependency cycles, and then combining the values into a global model. It follows, that the non-existence of stable models of a logic program occurs since no suitable combination of truth value assignments to cycles can be found. However, this characterisation does not allow to draw a conclusion about *which* of the cycles are responsible that no stable model exists.

Syrjänen [Syr06] considers *all* odd-length negative dependency cycles as erroneous. In addition, he determines dissatisfied *constraints* as responsible for inconsistency, an ASP language construct not considered in our work. Brain et al. [BGP^{+}07b, BGP^{+}07a] and Gebser et al. [GPST08] study reasons why a given set of literals (e.g. an intended answer set) is not an answer set by translating the given logic program into a meta-program. The answer sets of this meta-program contain additional literals, which indicate errors in the underlying logic program. In contrast, we characterise reasons for inconsistency independent of an intended answer set.

7.6.2 Logic Programs with Explicit Negation

Oetsch et al. [OPT10], Polleres et al. [PFSF13], and Frühstück et al. [FPF13] extend the meta-programming approach for debugging in ASP [BGP$^+$07b, BGP$^+$07a, GPST08] to logic programs with explicit negation, but do not (explicitly) deal with inconsistency due to complementary literals. Furthermore, contrary to our work they assume as given an intended answer set.

Shchekotykhin [Shc15] and Dodaro et al. [DGM$^+$15] extend the meta-programming approach for debugging to the identification of *preferred* explanations as to why an intended set of literals is not an answer set. This is achieved by querying the user about literals that should definitely be included in an answer set. In contrast, our work characterises reasons for inconsistency without requiring interaction from the user and without the need for an intended answer set. Oetsch et al. [OPT11] also present an approach for finding errors in a logic program through interaction with the user, namely by "stepping through" the logic program. That is, in each step the user adds a rule to be satisfied, until some kind of inconsistency is reached. The work is different in spirit from ours, since Oetsch et al. are not concerned with *characterising* different reasons for inconsistency. Furthermore, even though they consider logic programs with explicit negation, they do not (explicitly) investigate inconsistency due to complementary literals.

Ulbrecht et al. [UTB16] investigate the *severity* of inconsistency, i.e. they propose quantitative measures for inconsistency and present desirable properties of such measures. Contrary to our work, the inconsistency measures are not concerned with identifying the cause of inconsistency, but with minimal changes that restore consistency.

7.6.3 Alternative Semantic Definitions

We here used the original definition of consistent and inconsistent logic programs under the answer set semantics as introduced by Gelfond and Lifschitz [GL91].

Some of the later definitions of the answer set semantics, e.g. *Answer Set Prolog* [GL02], do not turn answer sets comprising complementary literals into the set of all literals, thus leading to a slightly different notion of inconsistency. For handling inconsistency in Answer Set Prolog, Balduccini and Gelfond [BG03] propose to add special "consistency-restoring" rules, which allow the specification of preferences and can resolve inconsistencies.

Another line of research applies ideas from paraconsistent logics to the answer set semantics to yield new semantics, see e.g. [SI95, EFM10, AEF$^+$16]. As in Answer Set Prolog, complementary literals in these semantics do not cause to infer all literals. This allows to draw conclusions even when parts of a logic program are inconsistent. Furthermore, different kinds of inconsistencies can be distinguished.

7.7 Summary

We showed that the two ways in which a logic program may be inconsistent – it has no answer sets or its only answer set is the set of all literals – can be determined using the well-founded model semantics and further divided into four inconsistency cases: one where only explicit negation is responsible, one where only NAF is responsible, and two where the interplay of explicit negation and NAF is responsible for the inconsistency. Each of these cases is characterised by a different type of culprit set, containing literals that are responsible for the inconsistency due to the way in which they are derivable. These culprit sets can be identified using "weaker" semantics than answer sets and can be used to explain the inconsistency in terms of trees whose nodes are derivations, similar to the Attack Trees in Chapter 5.

The inconsistency where only NAF is responsible arises due to the same reasons as inconsistency in logic programs without explicitly negated atoms. We characterised this inconsistency by transferring the concept of SCUPs to logic programs.

A natural question following the characterisation of inconsistency cases is how to perform debugging based on the culprit sets, as well as how to deal with of multiple culprit sets for a logic program, which will be addressed in the future.

Chapter 8

Conclusion

8.1 Thesis Summary

In this thesis, we illustrated how concepts and methods from the field of computational argumentation can be applied to solve problems in Answer Set Programming (ASP). We focussed on two particular Argumentation formalisms, namely Assumption-Based Argumentation (ABA) and Abstract Argumentation (AA) frameworks.

For our investigations we both used existing concepts from ABA and AA frameworks, such as the notion of arguments and attacks, and developed new methods for these frameworks. In particular, we

- reformulated the semantics of ABA frameworks in terms of *assumption labellings*, and
- characterised sets of arguments responsible for the *non-existence of stable labellings* in AA frameworks.

Concerning problems in ASP, we investigated questions arising with respect to the *answer set semantics*:

1. how to explain why a literal is or is not contained in an answer set, and
2. how to characterise inconsistency and explain why it arises.

We provided answers to both questions by making use of an existing translation of logic programs into ABA and AA frameworks, and new and existing correspondence results between the semantics of logic programs and the translated ABA and AA frameworks.

As an answer to the first question, we introduced argumentative justifications of literals with respect to an answer set, based on arguments and attacks between them in the translated AA framework. We defined *Attack Trees* as explanations in terms of arguments, which may be more suitable for non-ASP experts. Furthermore, we introduced *ABAS Justifications* as explanations in terms of literals, which may be more suitable for ASP experts.

As an answer to the second question, we applied results from our investigation of the non-existence of stable models in AA frameworks to logic programs, yielding a characterisation of inconsistency in a particular class of logic programs (namely those without explicit negation). We then classified inconsistency cases in arbitrary logic programs, and illustrated how to construct argumentative explanations of parts of the logic program responsible for the inconsistency, which are similar to our Attack Trees.

Overall, we showed how techniques from one areas of research can aid finding solutions to the problems in another area of research. More precisely, we illustrated how techniques from computational argumentation can be applied to provide solutions to the problems of non-understandable answer sets and inconsistency of logic programs.

8.2 Future Work

We only considered the connection between *flat* ABA frameworks, their corresponding AA frameworks, and logic programs that may comprise both *negation-as-failure* (NAF) and *explicit negation*. Especially with regards to logic programs, there are many language extensions (see e.g. [BET11, Fab13]). An interesting direction of future research is therefore whether argumentation in general, and ABA and AA frameworks in particular, can also be used to explain answer sets of logic programs that make use of such language extensions. We will discuss this strand of future work in more detail in Section 8.2.1. Furthermore, ABA frameworks have been extended in various ways, for example to incorporate preferences and to construct arguments as graphs rather than trees. We discuss future directions of research concerning these extensions in Section 8.2.2.

In addition to the above lines of future work, there are two main topics left for future investigations. Firstly, we here only *characterised* inconsistency in logic programs under the answer set semantics. How to restore consistency based on our characterisations is left for future work, as discussed in more detail in Section 8.2.3. Secondly, we here mainly focussed on *theoretical* results and left the development of algorithms for future work, as discussed in Section 8.2.4.

8.2.1 Language Extensions of Logic Programs

A frequently used extension of logic programs is to allow the head of a clause to be a *disjunction* of literals. Since disjunctive heads are not defined for ABA rules, the direct translation of clauses into ABA rules as given in Section 4.2, is not applicable for such logic programs. Bochman [Boc03] introduces an extension of AA frameworks called *Collective Argumentation*, which is able to model the way disjunction is handled in logic programs. Future work will show whether a similar extension can be used for ABA frameworks to model disjunction in the head of rules. Furthermore, You et al. [YYG00] give an *abductive* interpretation of logic programs with disjunction. Since abductive interpretations of logic programs are instances of ABA [BDKT97], the abductive interpretation of disjunction in logic programs may also be an instance of ABA.

Another language extension of logic programs concerning the head of clauses are *constraints*. Constraints are clauses whose head is empty. The head of a constraint can equivalently be thought of as being the truth value $\mathtt{F}$. This means, if the body of a constraint is satisfied (i.e. all literals have truth value $\mathtt{T}$), then $\mathtt{F}$ is implied, and therefore the clause is not satisfied. Consequently, a constraint expresses conditions that should never be satisfied together. How to translate a constraint into an ABA rule is an open question since ABA rules cannot have an empty head. Due to the previously mentioned relationship between ABA and the abductive interpretation of logic programs, a useful starting point may be the work of Toni [Ton95], where an argumentation semantics is given to abductive logic programs with constraints.

Constraints are often combined with another language extension of logic programs, namely *aggregates*. Aggregates are special constructs able to express, for example, that a maximum or minimum number of given literals need to be satisfied. Different approaches have been introduced to translate logic program with aggregates into a logic program without aggregates which preserve the semantics [PDB03, SPE06]. Future work will show whether such translations can be used to model logic programs with aggregates in ABA and use our ABAS Justifications to provide explanations.

8.2.2 Extensions of ABA

Another interesting line of research is to consider extensions of ABA in the light of the developments presented in this thesis.

One such extension is the incorporation *preferences* into an ABA framework. Čyras and Toni [ČT16b] introduce an extension of ABA called ABA+, where some attacks in ABA are reversed to incorporate preferences, and Wakaki [Wak14] presents p_ABA, where assumption extensions are chosen among all (traditional) assumption extensions to account for the given preferences. One line of future research regarding ABA with preferences is to investigate if our new notions of assumption labellings can be extended to express the semantics of ABA+ and p_ABA frameworks. Concerning logic programming, various semantics have been proposed to handle preferences defined over the clauses or literals in a logic program, e.g. [SI96, ZF97, BE99, GTZ07]. Another interesting line of future research is thus the comparison of preference-handling in ABA frameworks and logic programs, to see if, for example, our justification methods can be adapted to logic programs with preferences by applying methods from ABA+ or p_ABA. This comparison will also involve extensions of AA frameworks that take preferences into account, e.g. [BC03, KvdT08, Mod09, BCGG11].

Another recent development of ABA frameworks was presented by Craven and Toni [CT16a], who introduce a new extension semantics for ABA, based on the interpretation of arguments as graphs rather than trees. Future work will show if it is possible to find a labelling semantics that corresponds to the new extension semantics.

As previously mentioned, we here focused on *flat* ABA frameworks (except for Section 3.5). This is because the head of a logic program cannot be a NAF literal, and consequently no rule in the translated ABA framework has an assumption as its head. There is however some work on logic programs that allow NAF literals in the head of clauses, e.g. [IS98, SBL14, Ji15]. Whether the semantics of such logic programs correspond to the semantics of non-flat ABA frameworks is another line of future research.

8.2.3 Inconsistency and Debugging

In this thesis, we focused on *characterising* and *explaining* inconsistency in logic programs. How to use this knowledge for *debugging* an inconsistent logic program so as to obtain meaningful answer sets is left for future work. It will be particularly interesting to see if our results on iterative SCUP revisions for obtaining a stable argument labelling from

Chapter 6 can be transferred to logic programs in order to obtain an answer set. For AA frameworks, we used results about the decomposability of complete argument labellings to prove that revising SCUPs allows to turn a preferred argument labelling into a stable argument labelling. Concerning logic programs, the decomposability of answer sets has also been investigated by various authors, e.g. [LT94, ELS97, FLLP09], which may be a useful starting point for proving that changes to SCUPs in logic programs allow to turn a 3-valued M-stable model into an answer set.

Furthermore, characterising the non-existence of stable assumption labellings in ABA is left for future work. Due to the semantic correspondence between flat ABA frameworks and AA frameworks and logic programs, we expect the characterisation for flat ABA frameworks to be straightforward. Whether a concept similar to SCUPs can also be identified for *non-flat* ABA frameworks will be an interesting line of research.

8.2.4 Computation, Implementation, and Applications

Our argumentative justifications of literals (not) contained in an answer set constitute the only part of this thesis that we implemented. However, the *LABAS Justifier* does currently not support logic programs containing variables, which is a limitation since many applications of ASP involve logic programs with variables. We thus intend to extend the computation of Attack Trees and LABAS Justifications in the LABAS Justifier to logic programs with variables and test its usefulness on applications. For example, Athakravi et al. [ASL$^+$15] extract logic programs from past legal cases and apply the answer set semantics to determine how to proceed when faced with a new legal case. They mention explanations of their solutions (i.e. of answer sets) as future work, so our LABAS Justifications may be useful in combination with their approach. Furthermore, we plan to integrate the LABAS Justifier into an IDE (Integrated Development Environment) for ASP such as ASPIDE [FRR11] or SeaLion [BOP$^+$13]. This will promote the usage of our justification methods in real-world applications and may lead to the increased use of ASP in application areas where explainability of solutions is crucial, such as medical decision support.

Argument labellings have been used in various algorithms for the computation of the semantics of AA frameworks (see [CDG$^+$15] for an overview). It will thus be interesting to investigate algorithms for the the computation of semantics of ABA frameworks using the *assumption labellings* presented in this thesis. Furthermore, argument labellings have been used in a software for teaching the semantics of AA frameworks to novices [DS14, SD16]. Future work will show if assumption labellings can be used in a similar way to teach the semantics of ABA frameworks.

Concerning the non-existence of stable argument labellings of AA frameworks, future work involves both complexity analysis and the development of an implementation for determining sets of arguments responsible for the non-existence and for turning a preferred argument labelling into a stable argument labelling.

Various implementations have been developed for debugging inconsistent logic programs, e.g. [BOP$^+$13, DMA15, DGM$^+$15, Shc15, GDM$^+$16]. In future work, we plan to evaluate these implementations to find the most suitable one for integrating our characterisations and explanations of inconsistency scenarios.

Bibliography

[ACP^{+}16] Abdallah Arioua, Madalina Croitoru, Laura Papaleo, Nathalie Pernelle, and Swan Rocher. On the Explanation of SameAs Statements Using Argumentation. In *Proceedings of the 10th International Conference on Scalable Uncertainty Management (SUM'16)*, pages 51–66, 2016.

[ADF^{+}13] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A Native ASP Solver Based on Constraint Learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, pages 54–66, 2013.

[ADLR15] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 40–54, 2015.

[AEF^{+}16] Giovanni Amendola, Thomas Eiter, Michael Fink, Nicola Leone, and João Moura. Semi-Equilibrium Models for Paracoherent Answer Set Programs. *Artificial Intelligence*, 234:219–271, 2016.

[Ari16] Ofer Arieli. On the Acceptance of Loops in Argumentation Frameworks. *Journal of Logic and Computation*, 26(4):1203—-1234, 2016.

[ARR^{+}93] Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. Explaining Program Execution in Deductive Systems. In *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 101–119, 1993.

[ASL^{+}15] Duangtida Athakravi, Ken Satoh, Mark Law, Krysia Broda, and Alessandra Russo. Automated Inference of Rules with Exception from Past Legal Cases Using ASP. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 83–96, 2015.

[ATC15] Abdallah Arioua, Nouredine Tamani, and Madalina Croitoru. Query Answering Explanation in Inconsistent Datalog +/- Knowledge Bases. In

Proceedings of the 26th International Conference Database and Expert Systems Applications (DEXA'15), pages 203–219, 2015.

[Bau11] Ringo Baumann. Splitting an Argumentation Framework. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 40–53, 2011.

[Bau12] Ringo Baumann. What Does it Take to Enforce an Argument? Minimal Change in Abstract Argumentation. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*, pages 127–132, 2012.

[BB10] Ringo Baumann and Gerhard Brewka. Expanding Argumentation Frameworks: Enforcing and Monotonicity Results. In *Proceedings of the 3rd International Conference on Computational Models of Argument (COMMA'10)*, pages 75–86, 2010.

[BBC$^+$14] Pietro Baroni, Guido Boella, Federico Cerutti, Massimiliano Giacomin, Leendert W. N. van der Torre, and Serena Villata. On the Input/Output Behavior of Argumentation Frameworks. *Artificial Intelligence*, 217:144–197, 2014.

[BBDW12] Ringo Baumann, Gerhard Brewka, Wolfgang Dvoák, and Stefan Woltran. Parameterized Splitting: A Simple Modification-Based Approach. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, pages 57–71. Springer Berlin Heidelberg, 2012.

[BC03] Trevor J. M. Bench-Capon. Persuasion in Practical Argument Using Value-based Argumentation Frameworks. *Journal of Logic and Computation*, 13(3):429–448, 2003.

[BC16] Trevor J. M. Bench-Capon. Dilemmas and Paradoxes: Cycles in Argumentation Frameworks. *Journal of Logic and Computation*, 26(4):1055—-1064, 2016.

[BCdSCLS13] Pierre Bisquert, Claudette Cayrol, Florence Dupin de Saint-Cyr, and Marie-Christine Lagasquie-Schiex. Enforcement in Argumentation Is a Kind of Update. In *Proceedings of the 7th International Conference on Scalable Uncertainty Management (SUM'13)*, pages 30–43, 2013.

[BCG11] Pietro Baroni, Martin Caminada, and Massimiliano Giacomin. An Introduction to Argumentation Semantics. *The Knowledge Engineering Review*, 26(04):365–410, 2011.

[BCGG11] Pietro Baroni, Federico Cerutti, Massimiliano Giacomin, and Giovanni Guida. AFRA: Argumentation framework with Recursive Attacks. *International Journal of Approximate Reasoning*, 52(1):19–37, 2011.

[BCLM91] Trevor J. M. Bench-Capon, D. Lowes, and A. M. McEnery. Argument-Based Explanation of Logic Programs. *Knowledge Based Systems*, 4(3):177–183, 1991.

[BCT$^+$04] Chitta Baral, Karen Chancellor, Nam Tran, Nhan Tran, Anna M. Joy, and Michael E. Berens. A Knowledge Based Approach for Representing and Reasoning about Signaling Networks. In *Proceedings of the 12th International Conference on Intelligent Systems for Molecular Biology and the 3rd European Conference on Computational Biology (ISMB/ECCB'04)*, pages 15–22, 2004.

[BD05] Martin Brain and Marina De Vos. Debugging Logic Programs under the Answer Set Semantics. In *Proceedings of the 3rd Workshop on Answer Set Programming, Advances in Theory and Implementation (ASP'05)*, 2005.

[BD08] Martin Brain and Marina De Vos. Answer Set Programming - a Domain in Need of Explanation: A Position Paper. In *Proceedomgs of the 3rd International Workshop on Explanation-aware Computing (ExaCt'08)*, pages 37–48, 2008.

[BDKT97] Andrei Bondarenko, Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. An Abstract, Argumentation-Theoretic Approach to Default Reasoning. *Artificial Intelligence*, 93(1-2):63–101, 1997.

[BE99] Gerhard Brewka and Thomas Eiter. Preferred Answer Sets for Extended Logic Programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

[BET11] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BG03] Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In *Proceedings of the International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, 2003.

[BG07] Pietro Baroni and Massimiliano Giacomin. On Principle-Based Evaluation of Extension-Based Argumentation Semantics. *Artificial Intelligence*, 171(10-15):675–700, 2007.

[BG10] Marcello Balduccini and Sara Girotto. Formalization of Psychological Knowledge in Answer Set Programming and its Application. *Theory and Practice of Logic Programming*, 10(4-6):725–740, 2010.

[BGG05] Pietro Baroni, Massimiliano Giacomin, and Giovanni Guida. SCC-Recursiveness: A General Schema for Argumentation Semantics. *Artificial Intelligence*, 168(1-2):162–210, 2005.

[BGG16] Pietro Baroni, Dov M. Gabbay, and Massimiliano Giacomin. Introduction to the Special Issue on Loops in Argumentation. *Journal of Logic and Computation*, 26(4):1051–1053, 2016.

[BGH$^+$14] Philippe Besnard, Alejandro J. García, Anthony Hunter, Sanjay Modgil, Henry Prakken, Guillermo R. Simari, and Francesca Toni. Introduction to Structured Argumentation. *Argument & Computation*, 5(1):1–4, 2014.

[BGK$^+$14] Richard Booth, Dov M. Gabbay, Souhila Kaci, Tjitze Rienstra, and Leendert W. N. van der Torre. Abduction and Dialogical Proof in Argumentation and Logic Programming. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*, pages 117–122, 2014.

[BGL15] Pietro Baroni, Massimiliano Giacomin, and Beishui Liao. I don't care, I don't know ... I know too much! On Incompleteness and Undecidedness in Abstract Argumentation. In Thomas Eiter, Hannes Strass, Mirosław Truszczyński, and Stefan Woltran, editors, *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation - Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, pages 265–280. Springer International Publishing, 2015.

[BGP$^+$07a] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. Debugging ASP Programs by Means of ASP. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, pages 31–43, 2007.

[BGP$^+$07b] Martin Brain, Martin Gebser, Jörg Pührer, Torsten Schaub, Hans Tompits, and Stefan Woltran. "That is illogical captain!" - The debugging support tool spock for answer-set programs: System description. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pages 71–85, 2007.

[BGP$^+$11] Guido Boella, Dov M. Gabbay, Alan Perotti, Leendert W. N. van der Torre, and Serena Villata. Conditional Labelling for Abstract Argumentation. In *Revised Selected Papers of the 1st International Workshop on Theorie and Applications of Formal Argumentation (TAFA'11)*, pages 232–248, 2011.

[BH01] Philippe Besnard and Anthony Hunter. A Logic-Based Theory of Deductive Arguments. *Artificial Intelligence*, 128(1-2):203–235, 2001.

[BKRvdT13] Richard Booth, Souhila Kaci, Tjitze Rienstra, and Leendert W. N. van der Torre. A Logical Theory about Dynamics in Abstract Argumentation. In *Proceedings of the 7th International Conference on Scalable Uncertainty Management (SUM'13)*, pages 148–161, 2013.

[BLGS16] Christopher Béatrix, Claire Lefèvre, Laurent Garcia, and Igor Stéphan. Justifications and Blocking Sets in a Rule-Based Answer Set Computation. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP'16)*, pages 6:1–6:15, 2016.

[BLMM92] Antonio Brogi, Evelina Lamma, Paolo Mancarella, and Paola Mello. Normal Logic Programs as Open Positive Programs. In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming (JICSLP'92)*, pages 783–797, 1992.

[Boc03] Alexander Bochman. Collective Argumentation and Disjunctive Logic Programming. *Journal of Logic and Computation*, 13(3):405–428, 2003.

[Boc16] Alexander Bochman. Abstract Dialectical Argumentation Among Close Relatives. In *Proceedings of the 6th International Conference on Computational Models of Argument (COMMA'16)*, pages 127–138, 2016.

[BOP$^+$13] Paula-Andra Busoniu, Johannes Oetsch, Jörg Pührer, Peter Skocovsky, and Hans Tompits. SeaLion: An Eclipse-Based IDE for Answer-Set Programming with Advanced Debugging Support. *Theory and Practice of Logic Progrmming*, 13(4-5):657–673, 2013.

[BS92] Chitta Baral and V. S. Subrahmanian. Stable and Extension Class Theory for Logic Programs and Default Logics. *Journal of Automated Reasoning*, 8(3):345–366, 1992.

[BS13] Ringo Baumann and Hannes Strass. On the Maximal and Average Numbers of Stable Extensions. In *2nd International Workshop on Theory and Applications of Formal Argumentation (TAFA'13)*, pages 111–126, 2013.

[BT09] Gustavo Adrian Bodanza and Fernando A. Tohmé. Two approaches to the problems of self-attacking arguments and general odd-length cycles of attack. *Journal of Applied Logic*, 7(4):403–420, 2009.

[BTK93] Andrei Bondarenko, Francesca Toni, and Robert A. Kowalski. An Assumption-Based Framework for Non-Monotonic Reasoning. In *Proceedings of the Second International Workshop on Logic Programming and Non-monotonic Reasoning (LPNMR'93)*, pages 171–189, 1993.

[BW16] Ringo Baumann and Stefan Woltran. The Role of Self-Attacking Arguments in Characterizations of Equivalence Notions. *Journal of Logic and Computation*, 26(4):1293–1313, 2016.

[Cam06a] Martin Caminada. On the Issue of Reinstatement in Argumentation. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA'06)*, pages 111–123, 2006.

[Cam06b] Martin Caminada. Semi-Stable Semantics. In *Proceedings of the 1st International Conference on Computational Models of Argument (COMMA'06)*, pages 121–130, 2006.

[Cam11] Martin Caminada. A Labelling Approach for Ideal and Stage Semantics. *Argument & Computation*, 2(1):1–21, 2011.

[Cam14] Martin Caminada. Strong Admissibility Revisited. In *Proceedings of the 5th International Conference on Computational Models of Argument (COMMA'14)*, pages 197–208, 2014.

[CDG^{+}15] Günther Charwat, Wolfgang Dvoák, Sarah A. Gaggl, Johannes P. Wallner, and Stefan Woltran. Methods for Solving Reasoning Problems in Abstract Argumentation - A Survey. *Artificial intelligence*, 220:28–63, 2015.

[CdSCLS10] Claudette Cayrol, Florence Dupin de Saint-Cyr, and Marie-Christine Lagasquie-Schiex. Change in Abstract Argumentation Frameworks: Adding an Argument. *Journal of Artificial Intelligence Research*, 38:49–84, 2010.

[CF17] Pedro Cabalar and Jorge Fandinno. Enablers and Inhibitors in Causal Justifications of Logic Programs. *Theory and Practice of Logic Programming*, 17(1):49–74, 2017.

[CFF14] Pedro Cabalar, Jorge Fandinno, and Michael Fink. Causal Graph Justifications of Logic Programs. *Theory and Practice of Logic Programming*, 14(4-5):603–618, 2014.

[CG09] Martin Caminada and Dov M. Gabbay. A Logical Account of Formal Argumentation. *Studia Logica*, 93(2-3):109–145, 2009.

[CGVZ14] Federico Cerutti, Massimiliano Giacomin, Mauro Vallati, and Marina Zanella. An SCC Recursive Meta-Algorithm for Computing Preferred Labellings in Abstract Argumentation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, 2014.

[CLS05] Claudette Cayrol and Marie-Christine Lagasquie-Schiex. On the Acceptability of Arguments in Bipolar Argumentation Frameworks. In *Proceedings of the 8th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU'05)*, pages 378–389, 2005.

[CLS13] Claudette Cayrol and Marie-Christine Lagasquie-Schiex. Bipolarity in Argumentation Graphs: Towards a Better Understanding. *International Journal of Approximate Reasoning*, 54(7):876–899, 2013.

[CMKMM14a] Sylvie Coste-Marquis, Sébastien Konieczny, Jean-Guy Mailly, and Pierre Marquis. A Translation-Based Approach for Revision of Argumentation Frameworks. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA'14)*, pages 397–411, 2014.

[CMKMM14b] Sylvie Coste-Marquis, Sébastien Konieczny, Jean-Guy Mailly, and Pierre Marquis. On the Revision of Argumentation Systems: Minimal Change of Arguments Statuses. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, pages 52–61, 2014.

[CMKMM15] Sylvie Coste-Marquis, Sébastien Konieczny, Jean-Guy Mailly, and Pierre Marquis. Extension Enforcement in Abstract Argumentation as an Optimization Problem. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 2876–2882, 2015.

[Cos06] Stefania Costantini. On the Existence of Stable Models of Non-Stratified Logic Programs. *Theory and Practice of Logic Programming*, 6(1-2):169–212, 2006.

[CP11] Martin Caminada and Gabriella Pigozzi. On Judgment Aggregation in Abstract Argumentation. *Autonomous Agents and Multi-Agent Systems*, 22(1):64–102, 2011.

[CS06] Martin Caminada and Chiaki Sakama. On the Existence of Answer Sets in Normal Extended Logic Programs. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 743–744, 2006.

[CS15] Martin Caminada and Claudia Schulz. On the Equivalence between Assumption-Based Argumentation and Logic Programming. In *Proceedings of the 1st International Workshop on Argumentation and Logic Programming (ArgLP'15)*, 2015.

[CSAD15a] Martin Caminada, Samy Sá, João Alcântara, and Wolfgang Dvoák. On the Difference between Assumption-Based Argumentation and Abstract

Argumentation. *The IfCoLog Journal of Logics and their Applications*, 2(1):16–34, 2015.

[CSAD15b] Martin Caminada, Samy Sá, João Alcântara, and Wolfgang Dvoák. On the Equivalence between Logic Programming Semantics and Argumentation Semantics. *International Journal of Approximate Reasoning*, 58:87–111, 2015.

[CT16a] Robert Craven and Francesca Toni. Argument Graphs and Assumption-Based Argumentation. *Artificial Intelligence*, 233:1–59, 2016.

[ČT16b] Kristijonas Čyras and Francesca Toni. ABA+: Assumption-Based Argumentation with Preferences. In *Proceedings of the15th International Conference on Principles of Knowledge Representation and Reasoning (KR'16)*, pages 553–556, 2016.

[CVG15] Federico Cerutti, Mauro Vallati, and Massimiliano Giacomin. ArgSemSAT-1.0: Exploiting SAT Solvers in Abstract Argumentation. In *System Descriptions of the 1st International Competition on Computational Models of Argumentation (ICCMA'15)*, 2015.

[DAA13] Carlos Viegas Damásio, Anastasia Analyti, and Grigoris Antoniou. Justifications for Logic Programming. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, pages 530–542, 2013.

[DBC02] Paul E. Dunne and Trevor J. M. Bench-Capon. Coherence in Finite Argument Systems. *Artificial Intelligence*, 141(1-2):187–203, 2002.

[DBS15] Marc Denecker, Gerhard Brewka, and Hannes Strass. A Formal Theory of Justifications. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 250–264, 2015.

[DG16] Wolfgang Dvoák and Sarah A. Gaggl. Stage Semantics and the SCC-recursive Schema for Argumentation Semantics. *Journal of Logic and Computation*, 26(4):1149–1202, 2016.

[DGH09] James P Delgrande, Torsten Grote, and Aaron Hunter. A General Approach to the Verification of Cryptographic Protocols Using Answer Set Programming. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 355–367, 2009.

[DGM$^+$15] Carmine Dodaro, Philip Gasteiger, Benjamin Musitsch, Francesco Ricca, and Kostyantyn M. Shchekotykhin. Interactive Debugging of Non-ground

ASP Programs. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 279–293, 2015.

[DHP14] Sylvie Doutre, Andreas Herzig, and Laurent Perrussel. A Dynamic Logic Framework for Abstract Argumentation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, pages 62–71, 2014.

[DKT06] Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. Dialectic Proof Procedures for Assumption-Based, Admissible Argumentation. *Artificial Intelligence*, 170(2):114–159, 2006.

[DKT09] Phan Minh Dung, Robert A. Kowalski, and Francesca Toni. Assumption-Based Argumentation. In Guillermo R. Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 199–218. Springer US, 2009.

[DMA15] Carlos Viegas Damásio, João Moura, and Anastasia Analyti. Unifying Justifications and Debugging for Answer-Set Programs. In *Technical Communications of the 31st International Conference on Logic Programming (ICLP'15)*, 2015.

[DMT07] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. Computing Ideal Sceptical Argumentation. *Artificial Intelligence*, 171(10-15):642–674, 2007.

[DP10] Agostino Dovier and Enrico Pontelli, editors. *A 25-Year Perspective on Logic Programming: Achievements of the Italian Association for Logic Programming GULP*, volume 6125 of *Lecture Notes in Computer Science*. Springer, 2010.

[DR91] Phan Minh Dung and Phaiboon Ruamviboonsuk. Well-Founded Reasoning with Classical Negation. In *Proceedings of the 1st International Workshop on Logic Programming and Non-monotonic Reasoning (LPNMR'91)*, pages 120–132, 1991.

[DS14] Jeremie Dauphin and Claudia Schulz. ArgTeach - A Learning Tool for Argumentation Theory. In *Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'14)*, pages 776–783, 2014.

[DT96] Yannis Dimopoulos and Alberto Torres. Graph Theoretical Structures in Logic Programs and Default Theories. *Theoretical Computer Science*, 170(1-2):209–244, 1996.

[Dun91] Phan Minh Dung. Negations as Hypotheses: An Abductive Foundation for Logic Programming. In *Logic Programming, Proceedings of the 8th*

International Conference on Logic Programming (ICLP'91), pages 3–17, 1991.

[Dun92] Phan Minh Dung. On the Relations between Stable and Well-Founded Semantics of Logic Programs. *Theoretical Computer Science*, 105(1):7–25, 1992.

[Dun95a] Phan Minh Dung. An Argumentation-Theoretic Foundation for Logic Programming. *The Journal of Logic Programming*, 22(2):151–177, 1995.

[Dun95b] Phan Minh Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artificial Intelligence*, 77(2):321–357, 1995.

[EFM10] Thomas Eiter, Michael Fink, and João Moura. Paracoherent Answer Set Programming. In *Proceedings of the 12th International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, 2010.

[EK89] Kave Eshghi and Robert A. Kowalski. Abduction Compared with Negation by Failure. In *Proceedings of the 6th International Conference on Logic Programming (ICLP'89)*, pages 234–254, 1989.

[ELM$^+$97] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Non-Monotonic Reasoning. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, pages 364–375, 1997.

[ELS97] Thomas Eiter, Nicola Leone, and Domenico Saccà. On the Partial Semantics for Disjunctive Deductive Databases. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):59–96, 1997.

[EÖ15] Esra Erdem and Umut Öztok. Generating Explanations for Biomedical Queries. *Theory and Practice of Logic Programming*, 15(1):35–78, 2015.

[Erd11] Esra Erdem. Applications of Answer Set Programming in Phylogenetic Systematics. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, pages 415–431, 2011.

[Fab13] Wolfgang Faber. Answer Set Programming. In *Tutorial Lectures of the 9th International Summer School on Reasoning Web*, pages 162–193, 2013.

[Fag94] François Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1(1):51–60, 1994.

[FKIS02] Marcelo A. Falappa, Gabriele Kern-Isberner, and Guillermo R. Simari. Explanations, Belief Revision and Defeasible Reasoning. *Artificial Intelligence*, 141(1-2):1–28, 2002.

[FLLP09] Paolo Ferraris, Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Symmetric Splitting in the General Theory of Stable Models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 797–803, 2009.

[FLT06] Gérard Ferrand, Willy Lesaint, and Alexandre Tessier. Explanations and Proof Trees. *Computers and Informatics*, 25(2-3):105–122, 2006.

[FPF13] Melanie Frühstück, Jörg Pührer, and Gerhard Friedrich. Debugging Answer-Set Programs with Ouroboros - Extending the SeaLion Plugin. In *Proceedings of the 12th International ConferenceLogic Programming and Nonmonotonic Reasoning LPNMR'13)*, pages 323–328, 2013.

[FRR11] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: Integrated Development Environment for Answer Set Programming. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 317–330, 2011.

[FT14] Xiuyi Fan and Francesca Toni. A General Framework for Sound Assumption-Based Argumentation Dialogues. *Artificial Intelligence*, 216:20–54, 2014.

[Gab16a] Dov M. Gabbay. Logical Foundations for Bipolar and Tripolar Argumentation Networks: Preliminary Results. *Journal of Logic and Computation*, 26(1):247–292, 2016.

[Gab16b] Dov M. Gabbay. The Handling of Loops in Argumentation Networks. *Journal of Logic and Computation*, 26(4):1065–1147, 2016.

[GCRS13] Alejandro J. García, Carlos Iván Chesñevar, Nicolás D. Rotstein, and Guillermo R. Simari. Formalizing Dialectical Explanation Support for Argument-Based Reasoning in Knowledge-Based Systems. *Expert Systems with Applications*, 40(8):3233–3247, 2013.

[GDM$^+$16] Philip Gasteiger, Carmine Dodaro, Benjamin Musitsch, Kristian Reale, Francesco Ricca, and Konstantin Schekotihin. An integrated Graphical User Interface for Debugging Answer Set Programs. In *Proceedings of the Workshop on Trends and Applications of Answer Set Programming (TAASP'16)*, 2016.

[GE09] François Gagnon and Babak Esfandiari. Using Answer Set Programming to Enhance Operating System Discovery. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, pages 579–584, 2009.

[Gel08] Michael Gelfond. Answer Sets. In *Handbook of Knowledge Representation*, pages 285–316. 2008.

[GKK$^+$11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, 24(2):107–124, 2011.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary Report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, pages 1–9, 2014.

[GL88] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, pages 1070–1080, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[GL02] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[Got95] Georg Gottlob. Translating Default Logic into Standard Autoepistemic Logic. *Journal of the ACM*, 42(4):711–740, 1995.

[GPST08] Martin Gebser, Jörg Pührer, Torsten Schaub, and Hans Tompits. A Meta-Programming Technique for Debugging Answer-Set Programs. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI'08)*, pages 448–453, 2008.

[GS04] Alejandro J. García and Guillermo R. Simari. Defeasible Logic Programming: An Argumentative Approach. *Theory and Practice of Logic Programming*, 4(1-2):95–138, 2004.

[GS14] Alejandro J. García and Guillermo R. Simari. Defeasible Logic Programming: DeLP-Servers, Contextual Queries, and Explanations for Answers. *Argument & Computation*, 5(1):63–88, 2014.

[GSTV11] Martin Gebser, Torsten Schaub, Sven Thiele, and Philippe Veber. Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360, 2011.

[GTZ07] Sergio Greco, Irina Trubitsyna, and Ester Zumpano. On the semantics of logic programs with preferences. *Journal of Artificial Intelligence Research*, 30:501–523, 2007.

[Han14] Anubhav Hanjura. *Heroku Cloud Application Development.* Packt Publishing, 2014.

[HS16] Jesse Heyninck and Christian Straßer. Relations between Assumption-Based Approaches in Nonmonotonic Logic and Formal Argumentation. In *Proceedings of the 16th International Workshop on Non-Monotonic Reasoning (NMR'16)*, 2016.

[Imi87] Tomasz Imielinski. Results on Translating Defaults to Circumscription. *Artificial Intelligence*, 32(1):131–146, 1987.

[Inc15] Daniela Inclezan. An Application of Answer Set Programming to the Field of Second Language Acquisition. *Theory and Practice of Logic Programming*, 15(01):1–17, 2015.

[Ino93] Katsumi Inoue. *Studies on Abductive and Nonmonotonic Reasoning.* PhD thesis, Kyoto University, 1993.

[IS98] Katsumi Inoue and Chiaki Sakama. Negation as Failure in the Head. *Journal of Logic Programming*, 35(1):39–78, 1998.

[Jan99] Tomi Janhunen. On the Intertranslatability of Non-monotonic Logics. *Annals of Mathematics and Artificial Intelligence*, 27(1-4):79–128, 1999.

[Ji15] Jianmin Ji. Discovering Classes of Strongly Equivalent Logic Programs with Negation as Failure in the Head. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management (KSEM'15)*, pages 147–153, 2015.

[KBM$^+$13] Dionysios Kontarinis, Elise Bonzon, Nicolas Maudet, Alan Perotti, Leendert W. N. van der Torre, and Serena Villata. Rewriting Rules for the Computation of Goal-Oriented Changes in an Argumentation System. In *Proceedings of the14th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA'13)*, pages 51–68, 2013.

[KM92] Antonis C. Kakas and Paolo Mancarella. Short Note: Preferred Extensions are Partial Stable Models. *The Journal of Logic Programming*, 14(3-4):341–348, 1992.

[KOJS15] Laura Koponen, Emilia Oikarinen, Tomi Janhunen, and Laura Säilä. Optimizing Phylogenetic Supertrees Using Answer Set Programming. *Theory and Practice of Logic Programming*, 15(4-5):604–619, 2015.

[KvdT08] Souhila Kaci and Leendert W. N. van der Torre. Preference-based argumentation: Arguments supporting multiple values. *International Journal of Approximate Reasoning*, 48(3):730–751, 2008.

[LD04] Carmen Lacave and Francisco J. Díez. A Review of Explanation Methods for Heuristic Expert Systems. *The Knowledge Engineering Review*, 19(2):133–146, 2004.

[LGR16] Ho-Pun Lam, Guido Governatori, and Régis Riveret. On ASPIC+ and Defeasible Logic. In *Proceedings of the 6th International Conference on Computational Models of Argument (COMMA'16)*, pages 359–370, 2016.

[Lia13] Beishui Liao. Toward Incremental Computation of Argumentation Semantics: A Decomposition-Based Approach. *Annals of Mathematics and Artificial Intelligence*, 67(3-4):319–358, 2013.

[LJK11] Beishui Liao, Li Jin, and Robert C. Koons. Dynamics of Argumentation Systems: A Division-Based Method. *Artificial Intelligence*, 175(11):1790–1814, 2011.

[LLD13] Beishui Liao, Liyun Lei, and Jianhua Dai. Computing Preferred Labellings by Exploiting SCCs and Most Sceptically Rejected Arguments. In *Revised Selected Papers of the 2nd International Workshop on Theory and Applications of Formal Argumentation (TAFA'13)*, pages 194–208, 2013.

[LM11] João Leite and João Martins. Social Abstract Argumentation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 2287–2292, 2011.

[LPF$^+$02] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Georg Gottlob, Giovambattista Ianni, Giuseppe Ielpa, Christoph Koch, Simona Perri, and Axel Polleres. The DLV System. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 537–540, 2002.

[LPF$^+$06] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[LSI16] Gaurab Luitel, Matthew Stephan, and Daniela Inclezan. Model Level Design Pattern Instance Detection Using Answer Set Programming. In

Proceedings of the 8th International Workshop on Modeling in Software Engineering (MiSE@ICSE'16), pages 13–19, 2016.

[LT94] Vladimir Lifschitz and Hudson Turner. Splitting a Logic Program. In *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, 1994.

[MIBD02] Bernard Moulin, Hengameh Irandoust, Micheline Bélanger, and G Desbordes. Explanation and Argumentation Capabilities: Towards the Creation of More Persuasive Agents. *Artificial Intelligence Review*, 17(3):169–222, 2002.

[MM09] Maxime Morge and Paolo Mancarella. Assumption-Based Argumentation for the Minimal Concession Strategy. In *Revised Selected and Invited Papers of the 6th International Workshop on Argumentation in Multi-Agent Systems (ArgMAS'09)*, pages 114–133, 2009.

[Mod09] Sanjay Modgil. Reasoning about Preferences in Argumentation Frameworks. *Artificial Intelligence*, 173(9):901–934, 2009.

[Moo85] Robert C. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[MP10] Sanjay Modgil and Henry Prakken. Reasoning about Preferences in Structured Extended Argumentation Frameworks. In *Proceedings of the 3rd International Conference on Computational Models of Argument (COMMA'10)*, pages 347–358, 2010.

[MP13] Sanjay Modgil and Henry Prakken. A General Account of Argumentation with Preferences. *Artificial Intelligence*, 195:361–397, 2013.

[MPR14] Marco Maratea, Luca Pulina, and Francesco Ricca. A Multi-Engine Approach to Answer-Set Programming. *Theory and Practice of Logic Programming*, 14(6):841–868, 2014.

[MPR15] Marco Maratea, Luca Pulina, and Francesco Ricca. Multi-level Algorithm Selection for ASP. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 439–445, 2015.

[MS13] Neil Middleton and Richard Schneeman. *Heroku: Up and Running.* O'Reilly Media, 2013.

[Nou13] Farid Nouioua. AFs with Necessities: Further Semantics and Labelling Characterization. In *Proceedings of the 7th International Conference on Scalable Uncertainty Management (SUM'13)*, pages 120–133, 2013.

[NR10] Farid Nouioua and Vincent Risch. Bipolar Argumentation Frameworks with Specialized Supports. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'10)*, pages 215–218, 2010.

[NW14] Farid Nouioua and Eric Würbel. Removed Set-Based Revision of Abstract Argumentation Frameworks. In *Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence, (ICTAI'14)*, pages 784–791, 2014.

[ON08] Nir Oren and Timothy J. Norman. Semantics for Evidence-Based Argumentation. In *Proceedings of the 2nd International Conference on Computational Models of Argument (COMMA'08)*, pages 276–284, 2008.

[OPT10] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Catching the Ouroboros: On Debugging Non-Ground Answer-Set Programs. *Theory and Practice of Logic Programming*, 10(4-6):513–529, 2010.

[OPT11] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Stepping through an Answer-Set Program. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 134–147, 2011.

[ORL10] Nir Oren, Chris Reed, and Michael Luck. Moving Between Argumentation Frameworks. In *Proceedings of the 2nd International Conference on Computational Models of Argument (COMMA'08)*, pages 379–390, 2010.

[OZNC05] Mauricio Osorio, Claudia Zepeda, Juan Carlos Nieves, and Ulises Cortés. Inferring Acceptable Arguments with Answer Set Programming. In *Proceedings of the 6th Mexican International Conference on Computer Science (ENC'05)*, pages 198–205, 2005.

[PDB03] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Translation of Aggregate Programs to Normal Logic Programs. In *Proceedings of the 2nd International Workshop on Answer Set Programming, Advances in Theory and Implementation (ASP'03)*, 2003.

[Pea96] David Pearce. A New Logical Characterisation of Stable Models and Answer Sets. In *Selected Papers of Non-Monotonic Extensions of Logic Programming (NMELP'96)*, pages 57–70, 1996.

[PFSF13] Axel Polleres, Melanie Frühstück, Gottfried Schenner, and Gerhard Friedrich. Debugging Non-ground ASP Programs with Choice Rules, Cardinality and Weight Constraints. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, pages 452–464, 2013.

[PO14] Sylwia Polberg and Nir Oren. Revisiting Support in Abstract Argumentation Systems. In *Proceedings of the 5th International Conference on Computational Models of Argument (COMMA'14)*, pages 369–376, 2014.

[Pol17] Sylwia Polberg. *Intertranslatability of Abstract Argumentation Frameworks.* PhD thesis, Technische Universität Wien, 2017.

[Pra10] Henry Prakken. An Abstract Framework for Argumentation with Structured Arguments. *Argument & Computation*, 1(2):93–124, 2010.

[Prz90] Teodor C. Przymusinski. Well-Founded Semantics Coincides with Three-Valued Stable Semantics. *Fundamenta Informaticae*, 13(4):445–463, 1990.

[Prz91a] Teodor C. Przymusinski. Semantics of Disjunctive Logic Programs and Deductive Databases. In *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases (DOOD'91)*, pages 85–107, 1991.

[Prz91b] Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9(3-4):401–424, 1991.

[PSEK09] Enrico Pontelli, Tran Cao Son, and Omar El-Khatib. Justifications for Logic Programs under Answer Set Semantics. *Theory and Practice of Logic Programming*, 9(1):1–56, 2009.

[RGA$^+$12] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-Building with Answer Set Programming in the Gioia-Tauro Seaport. *Theory and Practice of Logic Programming*, 12(3):361–381, 2012.

[RPV$^+$11] Tjitze Rienstra, Alan Perotti, Serena Villata, Dov M. Gabbay, and Leendert W. N. van der Torre. Multi-Sorted Argumentation. In *Revised Selected Papers of the 1st International Workshop on Theorie and Applications of Formal Argumentation (TAFA'11)*, pages 215–231, 2011.

[RS09] Iyad Rahwan and Guillermo R. Simari. *Argumentation in Artificial Intelligence.* Springer US, 2009.

[Sac95] Domenico Saccà. Deterministic and Non-Deterministic Stable Model Semantics for Unbound DATALOG Queries. In *Proceedings of the 5th International Conference on Database Theory (ICDT'95)*, pages 353–367, 1995.

[SB14] Anthony J. Smith and Joanna J. Bryson. A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games. In *Proceedings of the 50th Annual Convention of the AISB (AISB'14)*, 2014.

[SBL14] Martin Slota, Martin Baláz, and João Leite. On Supporting Strong and Default Negation in Answer-Set Program Updates. In *Proceedings of the 14th Ibero-American Conference on AI (IBERAMIA'14)*, pages 41–53, 2014.

[SD16] Claudia Schulz and Dragos Dumitrache. The ArgTeach Web-Platform. In *Proceedings of the 6th International Conference on Computational Models of Argument (COMMA'16)*, pages 475–476, 2016.

[Shc15] Kostyantyn M. Shchekotykhin. Interactive Query-Based Debugging of ASP Programs. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*, pages 1597–1603, 2015.

[SI95] Chiaki Sakama and Katsumi Inoue. Paraconsistent Stable Semantics for Extended Disjunctive Programs. *Journal of Logic and Computation*, 5(3):265–285, 1995.

[SI96] Chiaki Sakama and Katsumi Inoue. Representing Priorities in Logic Programs. In *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming (JICSLP'96)*, pages 82–96, 1996.

[SN01] Tommi Syrjänen and Ilkka Niemelä. The Smodels System. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, pages 434–438, 2001.

[SPE06] Tran Cao Son, Enrico Pontelli, and Islam Elkabani. An Unfolding-Based Semantics for Logic Programming with Aggregates. *CoRR*, abs/cs/060, 2006.

[SST13] Claudia Schulz, Marek Sergot, and Francesca Toni. Argumentation-Based Answer Set Justification. In *Proceedings of the 11th International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense'13)*, 2013.

[SST15] Claudia Schulz, Ken Satoh, and Francesca Toni. Characterising and Explaining Inconsistency in Logic Programs. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 467–479, 2015.

[ST14] Claudia Schulz and Francesca Toni. Complete Assumption Labellings. In *Proceedings of the 5th International Conference on Computational Models of Argument (COMMA'14)*, pages 405–412, 2014.

[ST15] Claudia Schulz and Francesca Toni. Logic Programming in Assumption-Based Argumentation Revisited - Semantics and Graphical Representation. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*, pages 1569–1575, 2015.

[ST16] Claudia Schulz and Francesca Toni. Justifying Answer Sets using Argumentation. *Theory and Practice of Logic Programming*, 16(01):59–110, 2016.

[ST17a] Claudia Schulz and Francesca Toni. Labellings for Assumption-Based and Abstract Argumentation. *International Journal of Approximate Reasoning*, 84:110 – 149, 2017.

[ST17b] Claudia Schulz and Francesca Toni. On the Non-Existence and Restoration of Stable Labellings in Abstract Argumentation Frameworks. *Under Review*, 2017.

[Syr06] Tommi Syrjänen. Debugging Inconsistent Answer Set Programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning (NMR'06)*, pages 77–84, 2006.

[SZ90] Domenico Saccà and Carlo Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, pages 205–217, 1990.

[SZ91] Domenico Saccà and Carlo Zaniolo. Partial Models and Three-Valued Models in Logic Programs with Negation. In *Proceddings of the 1st International Workshop on Logic Programming and Non-monotonic Reasoning (LPNMR'91)*, pages 87–101, 1991.

[TKI08] Matthias Thimm and Gabriele Kern-Isberner. On the Relationship of Defeasible Argumentation and Answer Set Programming. In *Proceedings of the 2nd International Conference on Computational Models of Argument (COMMA'08)*, pages 393–404, 2008.

[TKI14] Matthias Thimm and Gabriele Kern-Isberner. On Controversiality of Arguments and Stratified Labelings. In *Proceedings of the 5th International Conference on Computational Models of Argument (COMMA'14)*, pages 413–420, 2014.

[TML13] Giorgio Terracina, Alessandra Martello, and Nicola Leone. Logic-Based Techniques for Data Cleaning: An Application to the Italian National Healthcare System. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, pages 524–529, 2013.

[TMP$^+$17] Sjoerd T. Timmer, John-Jules Ch. Meyer, Henry Prakken, Silja Renooij, and Bart Verheij. A two-phase Method for Extracting Explanatory Arguments from Bayesian Networks. *International Journal of Approximate Reasoning*, 80:475–494, 2017.

[Ton95] Francesca Toni. A Semantics for the Kakas-Mancarella Procedure for Abductive Logic Programming. In *Proceedings of the 1995 Joint Conference on Declarative Programming*, pages 231–244, 1995.

[Ton12] Francesca Toni. Reasoning on the Web with Assumption-Based Argumentation. In *Proceedings of the 8th International Summer School on Reasoning Web*, pages 370–386, 2012.

[Ton13] Francesca Toni. A Generalised Framework for Dispute Derivations in Assumption-Based Argumentation. *Artificial Intelligence*, 195:1–43, 2013.

[Ton14] Francesca Toni. A Tutorial on Assumption-Based Argumentation. *Argument & Computation*, 5(1):89–117, 2014.

[UTB16] Markus Ulbricht, Matthias Thimm, and Gerhard Brewka. Measuring Inconsistency in Answer Set Programs. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA'16)*, pages 577—-583, 2016.

[Van93] Allen Van Gelder. The Alternating Fixpoint of Logic Programs with Negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VPRV16] Charlotte S. Vlek, Henry Prakken, Silja Renooij, and Bart Verheij. A Method for Explaining Bayesian Networks for Legal Evidence with Scenarios. *Artificial Intelligence and Law*, 24(3):285–324, 2016.

[VRS88] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. Unfounded Sets and Well-Founded Semantics for General Logic Programs. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'88)*, pages 221–230, 1988.

[VRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Wak14] Toshiko Wakaki. Assumption-Based Argumentation Equipped with Preferences. In *Proceedings of the17th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA'14)*, pages 116–132, 2014.

[WCG09] Yining Wu, Martin Caminada, and Dov M. Gabbay. Complete Extensions in Argumentation Coincide with 3-Valued Stable Models in Logic Programming. *Studia Logica*, 93(2-3):383–403, 2009.

[YMR16] Anthony P. Young, Sanjay Modgil, and Odinaldo Rodrigues. Prioritised Default Logic as Rational Argumentation. In *Proceedings of the 15th International Conference on Autonomous Agents & Multiagent Systems (AAMAS'16)*, pages 626–634, 2016.

[YY90] Jia-Huai You and Li Yan Yuan. Three-Valued Formalization of Logic Programming: Is It Needed? In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'90)*, pages 172–182, 1990.

[YY94] Jia-Huai You and Li Yan Yuan. A Three-Valued Semantics for Deductive Databases and Logic Programs. *Journal of Computer and System Sciences*, 49(2):334–361, 1994.

[YY95] Jia-Huai You and Li Yan Yuan. On the Equivalence of Semantics for Normal Logic Programs. *The Journal of Logic Programming*, 22(3):211–222, 1995.

[YYG00] Jia-Huai You, Li Yan Yuan, and Randy Goebel. An Abductive Approach to Disjunctive Logic Programming. *Journal of Logic Programming*, 44(1-3):101–127, 2000.

[ZF97] Yan Zhang and Norman Y. Foo. Answer Sets for Prioritized Logic Programs. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS'97)*, pages 69–83, 1997.

[ZFTL14] Qiaoting Zhong, Xiuyi Fan, Francesca Toni, and Xudong Luo. Explaining Best Decisions via Argumentation. In *Proceedings of the European Conference on Social Intelligence (ECSI'14)*, pages 224–237, 2014.

Appendix A

Auxiliary Results

This appendix comprises auxiliary results used in the proofs of Chapter 6.

Lemma A.1. *Let $LabArg$ be a labelling of $\mathcal{AA}$, $Args \subseteq Ar$ and $A \in Ar \setminus Args$. Let $\mathcal{AA}^{\circledast}$ be a revision of $\mathcal{AA}$ w.r.t. $Args$ by $LabArg$ and $LabArg^{\circledast}$ a revision labelling of $\mathcal{AA}^{\circledast}$. Then A is legally labelled by $LabArg$ in $\mathcal{AA}$ if and only if A is legally labelled by $LabArg^{\circledast}$ in $\mathcal{AA}^{\circledast}$.*

Proof. From left to right: Let A be legally labelled by $LabArg$. By Definition 6.1, $(B, A) \in Att$ if and only if $(B, A) \in Att^{\circledast}$. Furthermore, $LabArg^{\circledast}(A) = LabArg(A)$ and for all $B \in Ar$, $LabArg^{\circledast}(B) = LabArg(B)$. Since it only depends on the labels of attackers of A whether or not A is legally labelled, it follows that A is legally labelled by $LabArg^{\circledast}$. The proof of the opposite direction is analogous. □

Lemma A.2. *Let $LabArg$ and $LabArg'$ be two labellings of $\mathcal{AA}$ such that $LabArg \sqsubseteq LabArg'$. Then, $\forall A \in \mathtt{in}(LabArg) \cup \mathtt{out}(LabArg)$ it holds that if A is legally labelled by $LabArg$, then A is legally labelled by $LabArg'$.*

Proof. Let $A \in \mathtt{in}(LabArg)$. Then for all attackers B of A, $B \in \mathtt{out}(LabArg)$. By definition of $LabArg'$, $A \in \mathtt{in}(LabArg')$ and for all attackers B of A, $B \in \mathtt{out}(LabArg')$. Thus, A is legally labelled `in` by $LabArg'$. Let $A \in \mathtt{out}(LabArg)$. Then there exists an attacker B of A such that $B \in \mathtt{in}(LabArg)$. By definition of $LabArg'$, $A \in \mathtt{out}(LabArg')$ and $B \in \mathtt{in}(LabArg')$. Thus, A is legally labelled `out` by $LabArg'$. □

Lemma A.3. *Let $Args$ be an enforcement set w.r.t. $LabArg_{pref}$ and $LabArg$ an enforcement labelling w.r.t. $Args$. Then $\forall A \in Ar \setminus Args$: A is legally labelled by $LabArg$.*

Proof. Let $A \in Ar \setminus Args$. By Definition 6.4, if $A \in \mathtt{undec}(LabArg_{pref})$, then A is legally labelled by $LabArg$. If $A \in \mathtt{in}(LabArg_{pref}) \cup \mathtt{out}(LabArg_{pref})$, then by Lemma A.2 A is legally labelled by $LabArg$, since $LabArg_{pref} \sqsubset LabArg$. □

Definition A.1 (Compatible Labelling)**.** Let $Args_1, Args_2 \subseteq Ar$ such that $Args_1 \cap Args_2 = \emptyset$ and $Args_1 \cup Args_2 = Ar$. Let $LabArg_1$ be a labelling of $\mathcal{AA}{\downarrow}_{Args_1}$ and $LabArg_2$

a labelling of $\mathcal{AA}{\downarrow}_{Args_2}$. $LabArg_1$ is *compatible* with $LabArg_2$ if and only if $LabArg_1$ is a complete labelling w.r.t. $(\mathcal{AA}{\downarrow}_{Args_1}, Args_2, LabArg_2, Att \cap (Args_2 \times Args_1))$.

Lemma A.4. *Let $Args_1, Args_2 \subseteq Ar$ such that $Args_1 \cap Args_2 = \emptyset$ and $Args_1 \cup Args_2 = Ar$. Let $LabArg_1$ be a labelling of $\mathcal{AA}{\downarrow}_{Args_1}$ and $LabArg_2$ a labelling of $\mathcal{AA}{\downarrow}_{Args_2}$. $LabArg = LabArg_1 \cup LabArg_2$ is a complete labelling of $\mathcal{AA}$ if and only if $LabArg_1$ is compatible with $LabArg_2$ and $LabArg_2$ is compatible with $LabArg_1$.*

Proof. Follows from Definition A.1 and Theorem 3 in [BBC$^+$14]. □

Lemma A.5. *Let $Args_1, Args_2 \subseteq Ar$ such that $Args_1 \cap Args_2 = \emptyset$, $Args_1 \cup Args_2 = Ar$, and $Args_2$ does not attack $Args_1$. Let $LabArg_1$ be a complete labelling of $\mathcal{AA}{\downarrow}_{Args_1}$ and $LabArg_2$ a labelling of $\mathcal{AA}{\downarrow}_{Args_2}$. $LabArg = LabArg_1 \cup LabArg_2$ is a complete labelling of $\mathcal{AA}$ if and only if $LabArg_2$ is compatible with $LabArg_1$.*

Proof. From left to right: Let $LabArg = LabArg_1 \cup LabArg_2$ be a complete labelling of $\mathcal{AA}$. Then by Lemma A.4, $LabArg_2$ is compatible with $LabArg_1$.
From right to left: Let $LabArg_2$ be compatible with $LabArg_1$. Since $LabArg_1$ is a complete labelling of $\mathcal{AA}{\downarrow}_{Args_1}$, by Proposition 1 in [BBC$^+$14] $LabArg_1$ is a complete labelling w.r.t. $(\mathcal{AA}{\downarrow}_{Args_1}, \emptyset, \emptyset, \emptyset)$. Since $Args_2$ does not attack $Args_1$, it follows that $LabArg_1$ is a complete labelling w.r.t. $(\mathcal{AA}{\downarrow}_{Args_1}, Args_2, LabArg_2, \emptyset)$, so $LabArg_1$ is compatible with $LabArg_2$. Thus by Lemma A.4, $LabArg_1 \cup LabArg_2$ is a complete labelling of $\mathcal{AA}$. □

We can generalise Lemma A.5 to SCCs.

Corollary A.6. *Let $Args_1, \ldots, Args_n$ $(n \geq 1)$ be a sequence of all SCCs of $\mathcal{AA}$ and for all $i \neq j$, $Args_i \neq Args_j$, and if $Args_i$ is attacked by $Args_k$ $(i \neq k)$, then $k < i$. Let $LabArg_i$ be a labelling of $\mathcal{AA}{\downarrow}_{Args_i}$. Then $LabArg = LabArg_1 \cup \ldots \cup LabArg_n$ is a complete labelling of $\mathcal{AA}$ if and only if $LabArg_1$ is a complete labelling of $Args_1$ and $LabArg_i$ is compatible with $LabArg_1 \cup \ldots \cup LabArg_{i-1}$ for all $i \in \{2 \ldots n\}$.*

Lemma A.7. *Let $Args_1, Args_2 \subseteq Ar$ such that $Args_1 \cap Args_2 = \emptyset$ and $Args_1 \cup Args_2 = Ar$. Let $LabArg_1$ be a labelling of $\mathcal{AA}{\downarrow}_{Args_1}$ and $LabArg_2$ a labelling of $\mathcal{AA}{\downarrow}_{Args_2}$. If $\forall A \in Args_1$ it holds that A is legally labelled by $LabArg_1 \cup LabArg_2$ in $\mathcal{AA}$, then $LabArg_1$ is compatible with $LabArg_2$.*

Proof. Let $LabArg = LabArg_1 \cup LabArg_2$ and let $A \in Args_1$.

- If $A \in \texttt{in}(LabArg_1)$, then clearly $A \in \texttt{in}(LabArg)$. Thus, $\forall B$ attacking A, $B \in \texttt{out}(LabArg)$. It follows that if $B \in Args_1$, $B \in \texttt{out}(LabArg_1)$, and if $B \in Args_2$, then $B \in \texttt{out}(LabArg_2)$.
- If $A \in \texttt{out}(LabArg_1)$, then clearly $A \in \texttt{out}(LabArg)$. Thus, $\exists B$ attacking A such that $B \in \texttt{in}(LabArg)$. It follows that $B \in Args_1$ and $B \in \texttt{in}(LabArg_1)$, or $B \in Args_2$ and $B \in \texttt{in}(LabArg_2)$.

- If $A \in \texttt{undec}(\mathit{LabArg}_1)$, then clearly $A \in \texttt{undec}(\mathit{LabArg})$. Thus, $\forall B$ attacking A, $B \notin \texttt{in}(\mathit{LabArg})$, and $\exists C$ attacking A such that $C \in \texttt{undec}(\mathit{LabArg})$. It follows that if $B \in \mathit{Args}_1$, $B \notin \texttt{in}(\mathit{LabArg}_1)$, and if $B \in \mathit{Args}_2$, then $B \notin \texttt{in}(\mathit{LabArg}_2)$. Furthermore, it follows that $C \in \mathit{Args}_1$ and $C \in \texttt{undec}(\mathit{LabArg}_1)$ or $C \in \mathit{Args}_2$ and $C \in \texttt{undec}(\mathit{LabArg}_2)$.

Thus, all $A \in \mathit{Args}_1$ satisfy the conditions in Definition A.1, so Args_1 is compatible with Args_2. □

Lemma A.8. *Let* $\mathit{ArgsIO} = \texttt{in}(\mathit{LabArg}_{\mathit{pref}}) \cup \texttt{out}(\mathit{LabArg}_{\mathit{pref}})$ *and* $\mathit{ArgsU} = \texttt{undec}(\mathit{LabArg}_{\mathit{pref}})$*. Then* $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsU}}$ *is the only complete labelling w.r.t.* $(\mathcal{AA}{\downarrow}_{\mathit{ArgsU}}, \mathit{ArgsIO}, \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}, \mathit{Att} \cap (\mathit{ArgsIO} \times \mathit{ArgsU}))$.

Proof. Since $\mathit{LabArg}_{\mathit{pref}}$ is a complete labelling of $\mathcal{AA}$, it holds by Lemma A.4 that $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}$ is compatible with $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsU}}$ and vice versa. By Definition A.1, it follows that $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsU}}$ is a complete labelling w.r.t.
$(\mathcal{AA}{\downarrow}_{\mathit{ArgsU}}, \mathit{ArgsIO}, \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}, \mathit{Att} \cap (\mathit{ArgsIO} \times \mathit{ArgsU}))$.
To prove that $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsU}}$ is the only such labelling, assume there exists a labelling $\mathit{LabArgU} \neq \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsU}}$ of $\mathcal{AA}{\downarrow}_{\mathit{ArgsU}}$ such that $\mathit{LabArgU}$ is a complete labelling w.r.t.
$(\mathcal{AA}{\downarrow}_{\mathit{ArgsU}}, \mathit{ArgsIO}, \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}, \mathit{Att} \cap (\mathit{ArgsIO} \times \mathit{ArgsU}))$.
Thus by Definition A.1, $\mathit{LabArgU}$ is compatible with $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}$.
Clearly, $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}} \cup \mathit{LabArgU}$, so by Lemma A.2 all $A \in \mathit{ArgsIO}$ are legally labelled by $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}} \cup \mathit{LabArgU}$.
Then by Lemma A.7, $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}$ is compatible with $\mathit{LabArgU}$. It follows by Lemma A.4, that $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}} \cup \mathit{LabArgU}$ is a complete labelling of $\mathcal{AA}$. Contradiction, since $\mathit{LabArg}_{\mathit{pref}} \sqsubset \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}} \cup \mathit{LabArgU}$ and $\mathit{LabArg}_{\mathit{pref}}$ is a preferred labelling. Thus, $\mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsU}}$ is the only complete labelling w.r.t.
$(\mathcal{AA}{\downarrow}_{\mathit{ArgsU}}, \mathit{ArgsIO}, \mathit{LabArg}_{\mathit{pref}}{\downarrow}_{\mathit{ArgsIO}}, \mathit{Att} \cap (\mathit{ArgsIO} \times \mathit{ArgsU}))$. □

Lemma A.9. *Let* $\mathit{Args}_1, \mathit{Args}_2 \subseteq \mathit{Ar}$ *such that* $\mathit{Args}_1 \cap \mathit{Args}_2 = \emptyset$ *and* $\mathit{Args}_1 \cup \mathit{Args}_2 = \mathit{Ar}$*. Let* LabArg_2 *be a labelling of* $\mathcal{AA}{\downarrow}_{\mathit{Args}_2}$*. Then there exists a labelling* LabArg_1 *of* $\mathcal{AA}{\downarrow}_{\mathit{Args}_1}$ *such that* LabArg_1 *is compatible with* LabArg_2*.*

Proof. Since Definition A.1 mirrors the definition of canonical local function of the complete semantics (Definition 24 in [BBC+14]), a labelling LabArg_1 of Args_1 is compatible with a labelling LabArg_2 of Args_2 if and only if LabArg_1 is an element of the canonical local function of the complete semantics of the argumentation framework with input $(\mathcal{AA}{\downarrow}_{\mathit{Args}_1}, \mathit{Args}_2, \mathit{LabArg}_2, \mathit{Att} \cap (\mathit{Args}_2 \times \mathit{Args}_1))$. By Definition 13 in [BBC+14], the canonical local function of the complete semantics of $(\mathcal{AA}{\downarrow}_{\mathit{Args}_1}, \mathit{Args}_2, \mathit{LabArg}_2, \mathit{Att} \cap (\mathit{Args}_2 \times \mathit{Args}_1))$ can be computed via the complete labellings of the standard argumentation framework of $(\mathcal{AA}{\downarrow}_{\mathit{Args}_1}, \mathit{Args}_2, \mathit{LabArg}_2, \mathit{Att} \cap (\mathit{Args}_2 \times \mathit{Args}_1))$. Since a standard argumentation framework always exists, it has a complete labelling, so the canonical local function of the complete semantics for $(\mathcal{AA}{\downarrow}_{Args_1}, Args_2, LabArg_2, Att \cap (Args_2 \times Args_1))$ is non-empty. Thus $LabArg_1$ exists. $\square$

Lemma A.10. *Let* $ArgsIO = \mathtt{in}(LabArg_{pref}) \cup \mathtt{out}(LabArg_{pref})$ *and* $ArgsU \subseteq \mathtt{undec}(LabArg_{pref})$. *Let* $LabArgIO = LabArg_{pref}{\downarrow}_{ArgsIO}$ *and let* $LabArgU$ *be some labelling of* $\mathcal{AA}{\downarrow}_{ArgsU}$. *Then* $LabArgIO$ *is compatible with* $LabArgU$.

Proof. We note that $\forall B \in ArgsU$ attacking some $A \in ArgsIO$ it holds that $A \in \mathtt{out}(LabArgIO)$ since arguments labelled `in` are not attacked by arguments labelled `undec` in $LabArg_{pref}$. Let $A \in ArgsIO$.

- If $A \in \mathtt{in}(LabArgIO)$, then for all $B \in ArgsIO$ attacking A it holds that $B \in \mathtt{out}(LabArgIO)$ since $LabArg_{pref}$ is a complete labelling. Furthermore, no $B \in ArgsU$ attacks A.
- If $A \in \mathtt{out}(LabArgIO)$, then there exists $B \in ArgsIO$ attacking A such that $B \in \mathtt{in}(LabArgIO)$ since $LabArg_{pref}$ is a complete labelling.

Thus, $LabArgIO$ is a complete labelling w.r.t. $(\mathcal{AA}{\downarrow}_{ArgsIO}, ArgsU, LabArgU, Att \cap (ArgsU \times ArgsIO))$, and therefore $LabArgIO$ is compatible with $LabArgU$. $\square$