Argumentation-theoretic proof procedures for default reasoning

23 December 1997

P.M. Dung
Division of Computer Science, Asian Institute of Technology
PO Box 2754, Bangkok 10501, Thailand
dung@cs.ait.ac.th

R.A. Kowalski, **F. Toni**
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
{rak,ft}@doc.ic.ac.uk

Abstract

In this paper we define an abstract proof procedure for default reasoning. The proof procedure is abstract because it applies to any logic for default reasoning which can be defined in an abstract argumentation-theoretic framework [3].

The abstract framework includes default logic, logic programming, autoepistemic logic, non-monotonic modal logic, and certain instances of circumscription as special cases. The framework can be understood as a generalisation of Theorist, in which default reasoning in general is understood as extending a theory formulated in a monotonic logic by a defeasible set of assumptions. An assumption can be defeated (or "attacked") if its "contrary" can be proved, possibly with the aid of other conflicting assumptions.

The standard (stable) semantics of most logics for default reasoning can be understood as sanctioning a set of assumptions, as an extension of a given theory, if and only if the set of assumptions does not attack itself and it attacks every assumption not in the set. In this paper we focus on proof procedures for admissibility semantics, which is more liberal than the stable semantics. It sanctions a set of assumptions if and only if it does not attack itself and it defends itself by attacking every set of assumptions which attacks it.

The proof procedure for admissibility is defined in the form of a logic program and is derived from a specification expressed in logical form. The derivation guarantees the correctness of the proof procedure. We also show that the proof procedure is complete.

1 Introduction

In [3], we have shown that many logics for default reasoning [21, 29, 22, 23] can be understood as special cases of a single abstract framework, based upon an argumentation-theoretic interpretation of the semantics of logic programming [11, 12] and its abstractions [8, 9, 4, 18], and extending Theorist [26].

In particular, we have shown that the standard (stable) semantics of most logics for default reasoning can be understood as sanctioning a set of assumptions as an extension of a given theory, formulated in some monotonic logic, if and only if the set of assumptions does not *attack* itself and it attacks every assumption not in the set. In abstract terms, an assumption can be attacked if its *contrary* can be proved, possibly with the aid of other conflicting assumptions. We call this abstract semantics the *stable* semantics, because it generalises the stable expansion semantics of autoepistemic logic [23] and the stable model semantics [13] of logic programming.

In [3] we also proposed a new semantics for logics for default reasoning, more liberal than the stable semantics, and generalising the admissibility semantics for logic programming [8]. In abstract terms, a set of assumptions is an *admissible* extension of a given theory, formulated in an underlying monotonic logic, if it does not attack itself and if it defends itself against all attacks (by attacking all sets of assumptions which attack it).

The admissibility semantics is more general than the stable semantics, in the sense that every set of assumptions sanctioned by the stable semantics is a maximal (with respect to set inclusion) set of assumptions sanctioned by the admissibility semantics. However, not every maximal admissible set of assumptions is stable.

The "global" character of the stable semantics renders it computationally unfeasible, in general. To show that a given sentence is justified by the stable semantics, it is necessary to construct a stable set of assumptions which together with the initially given theory monotonically implies the given sentence. For this purpose, it is necessary to perform a global search among all the assumptions in the language, to determine for each such assumption whether it or its contrary can be derived, independently of the sentence to be justified. Despite this computational unfeasibility, a number of proof procedures for the computation of the stable semantics have been proposed, for some concrete formalisms (mainly for logic programming and default logic). However, each of these proof procedures is defined only for highly restricted cases, e.g. for propositional and finite theories, and this undermines their use for practical applications. In recognition of this problem, in a recent survey on non-monotonic reasoning, Brewka, Dix and Konolige state that "[...] future research in the field should put more emphasis on computational aspects" since "the field [of non-monotonic reasoning in general and default reasoning in particular] can only have a lasting impact on AI and Computer Science in general if it can provide useful tools helping people to better handle their real world problems" ([5], page 144).

Differently from the stable semantics, the admissibility semantics requires only a "local" search, first to construct a set of assumptions which (monotonically) implies the sentence to be justified, and then to augment the set with further assumptions to defend the constructed set against attacks. Because of this "locality", the admissibility semantics is computationally more feasible than stable semantics.

In this paper we define two abstract proof procedures for computing admissibility semantics. The second proof procedure is a computationally more efficient refinement of the first. Both procedures are based upon a proof procedure [12] originally intended for computing stable semantics in logic programming. Although the original proof procedure was proved correct for locally stratified logic programs, it was shown in [12] to be incorrect for stable semantics in general. Subsequent evaluation of the proof procedure [8, 15, 16] has suggested that it was the semantics, rather than the proof procedure, which was at fault. The proof procedure [12] is correct for the admissibility semantics.

Both proof procedures are formulated as logic programs, derived from a formal specification. The derivation guarantees the correctness of the proof procedures. We also show that the proof procedures are complete.

The rest of the paper has the following structure: Section 2 describes the abstract framework, its concrete instances and the stable semantics in abstract terms. Section 3 discusses and illustrates the computational problems associated with the stable semantics. Section 4 describes the admissibility semantics. Section 5 introduces the top-level of the first, abstract proof procedure for the admissibility semantics. Section 6 introduces the top-level of the more efficient proof procedure. Section 7 defines the lower-level predicates. Section 8 compares our proposed proof procedures with other existing proof procedures. Section 9 concludes.

This paper is a revised and extended version of [10]. Whereas [10] concentrates on the (program transformation) techniques used to develop the proof procedures, this paper emphasises the significance of the proof procedures for default reasoning. Section 3, the last part of section 6 and sections 7 and 8 are new.

This paper continues the earlier paper [3], which described the abstract framework and its applications in detail. The paper [19] presents further examples, complementing those in [3]. Because the earlier papers provide the motivating examples, in this paper we focus upon the technical details of the proof procedure(s).

2 Assumption-based frameworks and stable semantics

In this section we describe an abstract form of the stable semantics and show how any deductive system for a monotonic logic can be extended to a non-monotonic logic for default reasoning.

A *deductive system* is a pair $(\mathcal{L}, \mathcal{R})$ where

- $\mathcal{L}$ is a formal language consisting of countably many sentences, and
- $\mathcal{R}$ is a set of inference rules of the form

  $$\frac{\alpha_1, \ldots, \alpha_n}{\alpha}$$

  where $\alpha, \alpha_1, \ldots, \alpha_n \in \mathcal{L}$ and $n \geq 0$. If $n = 0$, then the inference rule is an axiom.

A set of sentences $T \subseteq \mathcal{L}$ is called a *theory*.

A *deduction* from a theory T is a sequence $\beta_1, \ldots, \beta_m$, where $m > 0$, such that, for all $i = 1, \ldots, m$,

- $\beta_i \in T$, or
- there exists $\dfrac{\alpha_1, \ldots, \alpha_n}{\beta_i}$ in $\mathcal{R}$ such that $\alpha_1, \ldots, \alpha_n \in \{\beta_1, \ldots, \beta_{i-1}\}$.

$T \vdash \alpha$ means that there is a deduction (of α) from T whose last element is α. $Th(T)$ is the set $\{\alpha \in \mathcal{L} \,|\, T \vdash \alpha\}$. Deductive systems are *monotonic*, in the sense that $T \subseteq T'$ implies $Th(T) \subseteq Th(T')$. They are also *compact*, in the sense that $T \vdash \alpha$ implies $T' \vdash \alpha$ for some finite subset T' of T.

Given a deductive system $(\mathcal{L}, \mathcal{R})$, an *assumption-based framework* with respect to $(\mathcal{L}, \mathcal{R})$ is a tuple $\langle T, Ab, \overline{} \rangle$ where

- T, $Ab \subseteq \mathcal{L}$, $Ab \neq \{\}$
- $\overline{}$ is a mapping from Ab into $\mathcal{L}$. $\overline{\alpha}$ is called the *contrary* of α.

The theory T can be viewed as a given set of beliefs, and Ab as a set of candidate *assumptions* that can be used to extend T. An *extension* of a theory T is a theory $Th(T \cup \Delta)$, for some $\Delta \subseteq Ab$. Sometimes, informally, we refer to the extension simply as $T \cup \Delta$ or Δ.

Theorist [26] can be understood as a framework $\langle T, Ab, \overline{}\rangle$ where T and Ab are both arbitrary sets of sentences of first-order logic and the contrary $\overline{\alpha}$ of an assumption α is just its negation. *Circumscription* [21] can be understood similarly, except that the set of candidate assumptions is the set of all negations of (variable-free) atomic sentences $\neg p(t)$, for all predicates p which are minimised, and of all (variable-free) atomic sentences $q(t)$ or their negations, for all predicates q which are fixed.

Logic programs can be viewed as frameworks $\langle T, Ab, \overline{}\rangle$ where T is a set of clauses in logic programming form, the set of candidate assumptions Ab is the set of all negations $not\, p$ of (variable-free) atomic sentences p, and the contrary $\overline{not\, p}$ of an assumption is p.

Default logic [29] is the case where the monotonic logic is first-order logic augmented with domain-specific inference rules of the form

$$\frac{\alpha_1, \ldots, \alpha_m, M\beta_1, \ldots, M\beta_n}{\gamma}$$

where α_i, β_j, γ are first-order sentences. T is a first-order theory and Ab consists of all expressions of the form $M\beta$ where β is a sentence of first-order logic. The contrary $\overline{M\beta}$ of an assumption $M\beta$ is $\neg\beta$.

In *autoepistemic logic* [23] the language $\mathcal{L}$ is the language of modal logic, with a modal operator L, but the inference rules are those of classical logic. The assumptions have the form $\neg L\alpha$ or $L\alpha$. The contrary of $\neg L\alpha$ is α, and the contrary of $L\alpha$ is $\neg L\alpha$. *Non-monotonic modal logic* [22] is similar, except the inference rules are those of some modal logic and only assumptions of the form $\neg L\alpha$, having contraries α, are considered.

Definition 2.1 A set of assumptions $\Delta \subseteq Ab$ is *closed* if and only if $\Delta = \{\alpha \in Ab \mid T \cup \Delta \vdash \alpha\}$. An assumption-based framework is *flat* if and only if every set of assumptions $\Delta \subseteq Ab$ is closed.

All frameworks for logic programming and default logic are flat. Informally speaking, flat frameworks are ones in which explicit assumptions (in Δ) do not imply (together with T) "hidden" assumptions (not in Δ). [1]

Definition 2.2 A set of assumptions Δ is *stable* if and only if

1. Δ is closed,
2. for every assumption $\alpha \in \Delta$, $\overline{\alpha} \notin Th(T \cup \Delta)$,
3. for every assumption $\alpha \notin \Delta$, $\overline{\alpha} \in Th(T \cup \Delta)$,

i.e. Δ is closed and $\Delta = \{\alpha \in Ab \,|\, \overline{\alpha} \notin Th(T \cup \Delta)\}$.

Stable extensions correspond to scenaria in Theorist [26], minimal models in (some cases of) circumscription [21] extensions in default logic [29], stable expansions in autoepistemic logic [23], fixed points in non-monotonic modal logic [22] and stable models in logic programming [13]. Whereas circumscription is *sceptical*, in the sense that it sanctions a formula as a *default consequence* of a given theory if and only if the formula belongs to *all* stable extensions of the theory, the other logics are neutral, and can be used either in a sceptical manner, as in the case of circumscription, or in a *credulous* manner, to sanction a formula as a *default consequence* of a given theory if and only if the formula belongs to *some* stable extension of the theory.

In the following section we discuss some of the problems inherent in the computation of the stable semantics. All examples used in the paper are drawn from logic programming, because of its simplicity. However, all of these examples can also be expressed in all of the other formalisms (see [19]).

3 Computation of stable semantics

The complexity of computing the stable semantics is very high [20], [2] due to the fact that, for every assumption, it is necessary to determine that either it or its

[1]The framework with $T = \{\neg Lp\}$, in both auto-epistemic and non-monotonic modal logic, is an example of a non-flat framework. Indeed, the empty set of assumption implies, together with T, a "hidden" assumption, $\neg Lp$.

[2]For example, the problem of detecting credulous (sceptical) default consequences is Σ_2^P-complete (Π_2^P-complete, respectively) for default logic, autoepistemic logic and (some) non-monotonic modal logics, and NP-complete (co-NP-complete, respectively) for logic programming.

contrary (but not both) belong to any given extension.

Example 3.1 Consider the problem of determining whether, according to the stable model semantics, the atom $innocent(father(Mary))$ is a default consequence of the logic program T [3]

$$innocent(X) \leftarrow accused(X), not\, guilty(X)$$
$$accused(father(Mary)).$$

It is necessary to determine, for every assumption in the infinite set

$$Ab = \{not\, innocent(father^i(Mary)), not\, guilty(father^i(Mary)) | i \geq 0\}$$

whether or not it belongs to a stable set. However, only the assumption $not\, guilty(father(Mary))$ is "relevant" to $innocent(father(Mary))$, in the sense that it alone, together with the given theory, derives $innocent(father(Mary))$. Indeed, $not\, guilty(father(Mary))$ belongs to the unique stable set

$$\{not\, guilty(father^i(Mary)) | i \geq 0\}.$$

Some of the existing proof procedures for stable semantics do consider only such relevant assumptions (e.g. [31], see below), guaranteeing that they belong to some stable set. However, in general, it is not always sufficient to perform only a local search among relevant assumptions, as illustrated by the following example.

Example 3.2 Consider the problem of determining whether, according to the stable semantics, the atom $innocent(father(Mary))$ is a default consequence of the logic program T'

$$innocent(X) \leftarrow accused(X), not\, guilty(X)$$
$$accused(father(Mary))$$
$$accused(Mary) \leftarrow not\, accused(Mary).$$

The last clause can be seen as a bug in the program. It is not sufficient to restrict attention to the set of assumptions $\Delta = \{not\, guilty(father(Mary))\}$,

[3] This program represents the principle that if a person is accused of a crime, then the burden of proof is on the prosecution to show that the accused is guilty, rather than on the defence to show that he is not. The accused is assumed innocent, by default, unless proved guilty.

which together with the program implies the atom $innocent(father(Mary))$. It is necessary to find a superset of Δ which either contains $not\,accused(Mary)$ or derives $accused(Mary)$ (but not both), even though $innocent(father(Mary))$ is completely unrelated to the assumption $not\,accused(Mary)$. Since $T \cup \Delta \cup \{not\,accused(Mary)\} \vdash accused(Mary)$, there exists no such subset of Δ. Consequently, there exists no stable extension of T, and therefore $innocent(father(Mary))$ does not hold in the stable semantics.

As a consequence of its behaviour with examples like this, the stable semantics is not only computationally expensive, but unfeasible in the general case.

Despite the computational overheads, a number of proof procedures have been defined to compute the stable semantics in some cases. To the best of our knowledge, these proof procedures have been developed only for (restricted cases of) logic programming, default logic and circumscription.

Satoh and Iwayama [31] define a proof procedure for logic programming, computing credulous stable semantics for range-restricted programs that admit at least one stable model. Therefore, the proof procedure cannot be applied to the program in example 3.2. However, it successfully terminates for example 3.1, and it generates only the one relevant assumption needed in the example.

Chen and Warren [6] define a proof procedure for computing stable semantics for finite propositional programs. They use the proof procedure to compute credulous default consequences of non-propositional programs under the partial stable model semantics [30]. [4] In example 3.2, $innocent(father(Mary))$ is entailed by the partial stable model semantics, and the proof procedure gives a positive answer for $innocent(father(Mary))$ generating only the one relevant assumption needed in the example. Chen and Warren discuss possible adaptations for the stable model semantics, but do not present any concrete proof procedure.

Niemelä and Simmons [25] define proof procedures computing both credulous and sceptical stable semantics consequences for range-restricted, function-free logic programs. Because of these restrictions, these proof procedures cannot be applied to examples 3.1 and 3.2.

Satoh [32] adapts the (credulous) proof procedure in [31] to determine consequences with respect to the semantics of extensions in default logic. The proof procedure applies to consistent default theories whose default rules are expressed in clausal form. The procedure is defined only for the propositional case but is

[4] As we will see in section 4, partial stable models correspond to maximal admissible extensions.

claimed to be applicable in the non-propositional case, by appropriately instantiating variables.

Niemelä [24] defines proof procedures for the computation of extensions in default logic. These procedures are defined only for the propositional case and for a decidable fragment of first-order logic.

Barback and Lobo [2] define a proof procedure for the computation of stable extensions and the detection of credulous stable semantics consequence in default logic. The proof procedure is sound and complete for finite semi-normal default theories for which every subset is guaranteed to admit an extension.

All existing proof procedures for circumscription are defined only for theories in which every model is a Herbrand model, and under additional restrictions. For example, Przymusinski [27] restricts himself to the propositional case and Ginsberg [14] and Baker and Ginsberg [1] restrict themselves to theories without function symbols.

Therefore, to the best of our knowledge, no existing proof procedure for computing (credulous or sceptical) stable semantics consequences (for any formalism) applies in general. Moreover, we believe that no such general procedure can be defined, due to the global nature of the semantics.

In this paper, we propose proof procedures for computing (credulous) consequences under the admissibility semantics. This semantics is presented in the following section. The semantics and the proof procedures (that we will present later) are defined for abstract assumption-based frameworks but can be appropriately instantiated for different logics, as outlined in section 2.

4 Admissibility semantics

The admissibility semantics applies to any assumption-based framework $\langle T,\ Ab,\ \overline{} \rangle$.

Definition 4.1 Given a set of assumptions $A \subseteq Ab$,

- *A attacks an assumption* $\alpha \in Ab$ if and only if $T \cup A \vdash \overline{\alpha}$. The corresponding deduction of $\overline{\alpha}$ is said to be an *argument* for $\overline{\alpha}$ and against α, *supported* by A.

- *A attacks a set of assumptions* $\Delta \subseteq Ab$ if and only if A attacks some assumption $\alpha \in \Delta$. The corresponding deduction of $\overline{\alpha}$ is said to be an argument against Δ, *supported* by A.

It is easy to see that the stable semantics can be reformulated in argumentation-theoretic terms as follows: a set of assumptions Δ is *stable* if and only if

1. Δ is closed,

2. Δ does not attack itself, and

3. Δ attacks each assumption $\alpha \notin \Delta$

i.e. Δ is closed and Δ is the set of all assumptions it does not attack. Note that the totality requirement (3) of the stable semantics has a global character, which forces Δ to take a stand for, or against, every assumption α, no matter whether or not α is relevant to a given candidate consequence. The admissibility semantics, on the other hand, requires only that Δ takes a stand against those (relevant) assumptions that participate in a closed attack against Δ.

Definition 4.2 A set of assumptions Δ is *admissible* if and only if

1. Δ is closed,

2. Δ does not attack itself, and

3. for each closed set of assumptions $A \subseteq Ab$,
   if A attacks Δ then Δ attacks A.

Admissible extensions correspond to admissible scenaria for logic programming [8]. Maximally admissible extensions correspond to preferred extensions [8] and partial stable models [30] for logic programming. Admissibility semantics for other logics for default reasoning can be obtained by appropriately instantiating the abstract admissibility semantics for other instances of the framework.

Every stable extension is (maximal) admissible, but not every maximal admissible extension is stable. For instance, T' in example 3.2 has a maximal admissible extension, containing $not\,guilty(father(Mary))$, but this extension is not stable. However, in many cases, e.g. for order-consistent assumption-based frameworks (see [3]), maximal admissible extensions are always stable.

In this paper we focus upon the computation of consequences using the (credulous) admissibility semantics. The demonstration that a given sentence is a credulous admissibility semantics consequence of a theory requires only a local search, first to find an argument for the given sentence, and then to show that this support can defend itself by counter attacking all attacks against it, without attacking itself. In example 3.2, $innocent(father(Mary))$ is a (credulous and sceptical) admissibility semantics consequence of the program. Moreover, the demonstration of $innocent(father(Mary))$ does not need to consider the irrelevant assumption $not\,accused(Mary)$. Similarly, in example 3.1, $innocent(father(Mary))$ is a (credulous and sceptical) admissibility semantics consequence of the program. Its demonstration does not need to consider any of the irrelevant assumptions

$not\,innocent(Mary)$, $not\,innocent(father(father((Mary)))$, $\ldots$,

$not\,guilty(Mary)$, $not\,guilty(father(father((Mary)))$,$\ldots$.

Note that every admissible extension is contained in a maximal admissible extension. As a consequence, for all cases in which every maximal admissible extension is stable, e.g. for example 3.1 and, more generally, for all stratified or order-consistent assumption-based frameworks [3], admissibility semantics consequences are stable semantics consequences. Therefore, in all these cases, proof procedures for the admissibility semantics can be used for the stable semantics.

The following simple, but important theorem provides an alternative characterisation of admissibility. By virtue of this theorem, the condition that an admissible set of assumptions Δ does not attack itself does not need to be checked explicitly. It can be shown to hold implicitly if, for all closed attacks A against Δ, we restrict attention to counter attacks against assumptions in $A - \Delta$. This restriction has the additional computational advantage of reducing the number of candidate counter attacks that need to be considered.

Theorem 4.1 A set of assumptions $\Delta \subseteq Ab$ is admissible if and only if

1. Δ is closed, and
2. for each closed set of assumptions $A \subseteq Ab$,

   if A attacks Δ then Δ attacks $A - \Delta$.

The proof of this and all other results in the paper can be found in the appendix.

5 Proof procedure for admissibility semantics

In this section, we define a proof procedure for computing admissible sets of assumptions for any assumption-based framework. The proof procedure is in the form of a *metalevel logic program*, consisting of top-level clauses, defining a predicate $adm(\Delta_0, \Delta)$, i.e. Δ is an admissible superset of Δ_0, and of lower-level clauses, defining predicates $defends(\mathcal{D}, \Delta)$, i.e. $\mathcal{D}$ defends Δ by attacking all closed attacks against Δ, $closed(\Delta)$, i.e. Δ is a closed set of assumptions, as well as set-theoretic constructs.

In this and the next section we focus on the top-level of the program. In section 7 we will give (part of) the lower-level.

The top-level of the program is derived from logical *specifications* of the predicates adm and $defends$, given below. The specifications play the role of the semantics for the proof procedure. This method of obtaining a proof procedure, as a logic program derived from a specification under the assumption that low-level predicates in the program meet their specifications, renders the proof of soundness of the proof procedure almost trivial.

Specification 5.1 Let $\langle T, Ab, \overline{} \rangle$ be an assumption-based framework. For any sets of assumptions $\Delta_0, \Delta \subseteq Ab$

$adm(\Delta_0, \Delta) \leftrightarrow [\Delta_0 \subseteq \Delta \wedge \Delta$ is admissible$]$.

In the remainder of the paper, this specification of adm together with definition 4.2 of admissibility will be referred to as $\mathcal{S}pec_{adm}$.

The goal of the proof procedure is, given some set Δ_0, to generate Δ, such that $adm(\Delta_0, \Delta)$ holds. Typically, Δ_0 is given such that $T \cup \Delta_0 \vdash \alpha$ for some sentence $\alpha \in \mathcal{L}$. Therefore, the proof procedure can be used to compute credulous admissibility semantics consequences. For instance, in examples 3.1 and 3.2, $\alpha = innocent(father(Mary))$, $\Delta_0 = \{not\, guilty(father(Mary))\}$ and $\Delta = \Delta_0$.

Specification 5.2 Let $\langle T, Ab, \overline{} \rangle$ be an assumption-based framework. For any sets of assumptions $\mathcal{D}, \Delta \subseteq Ab$,

$defends(\mathcal{D}, \Delta) \leftrightarrow \forall A \subseteq Ab\ [[A$ attacks $\Delta \wedge closed(A)] \rightarrow \mathcal{D}$ attacks $A - \Delta]$
where $closed(A)$ means "A is closed" as specified in definition 2.1. We also say that $\mathcal{D}$ *defends* Δ.

In the remainder of the paper, this specification of the predicate $defends$ together with the auxiliary specifications of $attack$ and $closed$ given in definitions 4.1 and 2.1, respectively, and with appropriate definitions of set-theoretic constructs will be referred to as $\mathcal{S}pec_{lower}$.

The following corollary, which follows directly from theorem 4.1, gives a definition of adm, equivalent to $\mathcal{S}pec_{adm}$, in terms of $defends$, as specified by $\mathcal{S}pec_{lower}$.

Corollary 5.1
$\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta) \leftrightarrow [\Delta_0 \subseteq \Delta \wedge defends(\Delta, \Delta) \wedge closed(\Delta)]$.

The top-most level of the proof procedure is given by the logic program

$$Prog_{adm}: \boxed{\begin{array}{l} adm(\Delta, \Delta) \leftarrow defends(\Delta, \Delta), closed(\Delta) \\ adm(\Delta, \Delta') \leftarrow defends(\mathcal{D}, \Delta), closed(\Delta \cup \mathcal{D}), adm(\Delta \cup \mathcal{D}, \Delta') \end{array}}$$

Note that the choice of a defence $\mathcal{D}$ and the test that the set $\Delta \cup \mathcal{D}$ is closed are performed independently. If this test fails, then a different defence needs to be chosen. A more efficient procedure would construct defences so that the closedness condition is satisfied. This can be achieved by replacing the second clause in $Prog_{adm}$ by

$$adm(\Delta, \Delta') \leftarrow defends(\mathcal{D}, \Delta), closure(\Delta \cup \mathcal{D}, \mathcal{D}^*), adm(\mathcal{D}^*, \Delta')$$

where

$$closure(\Delta_1, \Delta_2) \leftrightarrow \Delta_2 = Th(T \cup \Delta_1) \cap Ab$$

i.e. $closure(\Delta_1, \Delta_2)$ holds if and only if Δ_2 is the smallest closed set of assumptions containing Δ_1.

In the case of flat assumption-based frameworks, every set of assumptions is closed, in which case, $Prog_{adm}$ simplifies to

$$\boxed{\begin{array}{l} adm(\Delta, \Delta) \leftarrow defends(\Delta, \Delta) \\ adm(\Delta, \Delta') \leftarrow defends(\mathcal{D}, \Delta), adm(\Delta \cup \mathcal{D}, \Delta') \end{array}}$$

Given some logic program $Prog_{lower}$ implementing $defends$, $closed$ and the set-theoretic constructs, we can apply SLD or SLDNF resolution to $Prog_{adm} \wedge Prog_{lower}$ with a PROLOG-style left-to-right selection rule and with a query

$$\leftarrow adm(\Delta_0, X)$$

for some given Δ_0. In this case, the proof procedure can be understood in algorithmic terms as repeatedly adding defences to the initially given set of assumptions Δ_0 until no further defences need to be added. Every set, obtained by adding a defence to the previous accumulated set of assumptions, must be closed. More precisely,

> given a current set of assumptions Δ, initially given as Δ_0,
> the proof procedure repeatedly
>
> **(1)** finds a set of assumptions $\mathcal{D}$ such that $\mathcal{D}$ defends Δ and $\Delta \cup \mathcal{D}$ is closed;
>
> **(2)** replaces Δ by $\Delta \cup \mathcal{D}$
>
> until $\mathcal{D} = \Delta$, when it returns Δ (by assigning it to X).

Note that step (1) is non-deterministic, since there might be more than one set of assumptions $\mathcal{D}$ defending the current Δ and such that $\Delta \cup \mathcal{D}$ is closed. The proof procedure potentially needs to explore a search tree of alternatives to find a branch which terminates with a set which defends itself and is closed.

Under this algorithmic interpretation, given the theory in example 3.1 (or, similarly, the theory in example 3.2) and $\Delta_0 = \{not\, guilty(father(Mary))\}$, one possible execution of the goal $\leftarrow adm(\Delta_0, X)$ is the following:

- X is instantiated by $\{not\, guilty(father(Mary))\}$, using the first clause of the program, and the subgoal

  $defends(\{not\, guilty(father(Mary))\}, \{not\, guilty(father(Mary))\})$

  is executed next;

- since

  $defends(\{not\, guilty(father(Mary))\}, \{not\, guilty(father(Mary))\})$

  holds, the substitution $X = \{not\, guilty(father(Mary))\}$ is returned as output.

However, other algorithmic interpretations can be obtained by employing other execution strategies. In particular, the execution of the three subgoals in the second clause can be interleaved, rather than executed left-to-right, PROLOG-style. Such interleaved execution will normally give rise to a more efficient proof procedure.

Example 5.1 Given the logic program in example 3.2 and
$\Delta_0 = \{not\, accused(Mary),\ not\, guilty(father(Mary))\}$,
one possible PROLOG execution of the goal $\leftarrow adm(\Delta_0, X)$ is the following:

- $\mathcal{D}_1 = \{not\,accused(Mary)\}$ is computed (since $defends(\mathcal{D}_1, \Delta_0)$ holds), and the subgoal $\leftarrow adm(\Delta_0 \cup \mathcal{D}_1)$ is executed next; $\mathcal{D}_1 = \mathcal{D}_1^1 \cup \mathcal{D}_1^2$, with $\mathcal{D}_1^1 = \{not\,accused(Mary)\}$ defending $not\,accused(Mary)$ in Δ_0 and $\mathcal{D}_1^2 = \{\}$ defending $not\,guilty(father(Mary))$ in Δ_0.

- $\leftarrow adm(\Delta_0 \cup \mathcal{D}_1)$ is reduced to
$\leftarrow defends(\mathcal{D}_2, \Delta_0 \cup \mathcal{D}_1), adm(\Delta_0 \cup \mathcal{D}_1 \cup \mathcal{D}_2, X)$,
which fails, since there is no defence for $\{not\,accused(Mary)\}$.

By interleaving, this failure can be detected immediately, as soon as $not\,accused(Mary)$ is identified as part of $\mathcal{D}_1$.

The soundness of $Prog_{adm}$ is expressed by corollary 5.2 below, which is a direct consequence of the following theorem:

Theorem 5.1 $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models Prog_{adm}$.

Corollary 5.2 For all $\Delta_0, \Delta \subseteq Ab$,
if $Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$,
then $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

(I.e. if the execution of the program generates Δ as output, given Δ_0 as input, then $\Delta_0 \subseteq \Delta$ and Δ is admissible).

The procedure $Prog_{adm}$ is also complete in the following sense:

Theorem 5.2 For all $\Delta_0, \Delta \subseteq Ab$,
if $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$
then $Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

(I.e. if, according to the specification, $\Delta_0 \subseteq \Delta$ and Δ is admissible, then the program will generate Δ as output, given Δ_0 as input.)

The full proof procedure is obtained by adding to $Prog_{adm}$ a program $Prog_{lower}$ for computing $defends$, $closed$ and the set-theoretic constructs, $\cup, \subseteq$, etc. If the program $Prog_{lower}$ is sound with respect to the specification $\mathcal{S}pec_{lower}$, then $Prog_{adm} \wedge Prog_{lower}$ is sound with respect to $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower}$. More formally:

Theorem 5.3 Given $Prog_{lower}$ such that, for all $\Delta, \mathcal{D} \subseteq Ab$,
if $Prog_{lower} \models defends(\mathcal{D}, \Delta)$ then $\mathcal{S}pec_{lower} \models defends(\mathcal{D}, \Delta)$,
if $Prog_{lower} \models closed(\Delta)$ then $\mathcal{S}pec_{lower} \models closed(\Delta)$, and
if $Prog_{lower} \models \Delta = \Delta_1 \cup \Delta_2$ then $\mathcal{S}pec_{lower} \models \Delta = \Delta_1 \cup \Delta_2$, etc.
then, for all $\Delta_0, \Delta \subseteq Ab$,
if $Prog_{adm} \wedge Prog_{lower} \models adm(\Delta_0, \Delta)$
then $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

Moreover, if a given program $Prog_{lower}$ is complete wrt the specification $\mathcal{S}pec_{lower}$, then $Prog_{adm} \wedge Prog_{lower}$ is complete with respect to $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower}$. More formally:

Theorem 5.4 Given $Prog_{lower}$ such that, for all $\Delta, \mathcal{D} \subseteq Ab$,
if $\mathcal{S}pec_{lower} \models defends(\mathcal{D}, \Delta)$ then $Prog_{lower} \models defends(\mathcal{D}, \Delta)$,
if $\mathcal{S}pec_{lower} \models closed(\Delta)$ then $Prog_{lower} \models closed(\Delta)$, and
if $\mathcal{S}pec_{lower} \models \Delta = \Delta_1 \cup \Delta_2$ then $Prog_{lower} \models \Delta = \Delta_1 \cup \Delta_2$, etc.
then, for all $\Delta_0, \Delta \subseteq Ab$,
if $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$
then $Prog_{adm} \wedge Prog_{lower} \models adm(\Delta_0, \Delta)$.

6 More efficient proof procedure

The proof procedure given by the program $Prog_{adm} \wedge Prog_{lower}$ performs a great deal of redundant computation. Suppose, for example, that $\mathcal{D}$ is computed as a defence for some currently accumulated set of assumptions Δ, i.e. $defends(\mathcal{D}, \Delta)$ has been successfully executed; and suppose that $\Delta \cup \mathcal{D}$ is closed. On the next invocation of the procedure, $\Delta \cup \mathcal{D}$ is the new accumulated set of assumptions Δ' and the set $\mathcal{D}'$ defending Δ' redundantly defends Δ. To make matters worse, the redundantly computed defence might unnecessarily be different from $\mathcal{D}$, and the unnecessary defence gives rise to further inefficiency because it has to be defended in turn. These inefficiencies are illustrated by the following example.

Example 6.1 Consider the program

$$\begin{array}{llll} p \leftarrow not\, q & q \leftarrow not\, s & s \leftarrow not\, t & t \leftarrow not\, u \\ & q \leftarrow not\, r & & \end{array}$$

The following hold

$defends(\{not\, r\}, \{not\, p\})$	$defends(\{\}, \{not\, r\})$
$defends(\{not\, s\}, \{not\, p\})$	$defends(\{not\, u\}, \{not\, s\})$
$defends(\{not\, s\}, \{not\, p, not\, r\})$	$defends(\{not\, r\}, \{not\, p, not\, r\})$
$defends(\{not\, u, not\, r\}, \{not\, s, not\, p, not\, r\})$	etc.

Given $\Delta_0 = \{not\, p\}$, one possible computation for the goal $\leftarrow adm(\Delta_0, X)$ is the following:

- $\mathcal{D}_1 = \{not\, r\}$ is computed (since $defends(\mathcal{D}_1, \Delta_0)$ holds).
  The subgoal $\leftarrow adm(\Delta_0 \cup \mathcal{D}_1, X)$ is executed next.

- $\mathcal{D}_2 = \{not\, s\}$ is computed (since $defends(\mathcal{D}_2, \Delta_0 \cup \mathcal{D}_1)$ holds).
  The subgoal $\leftarrow adm(\Delta_0 \cup \mathcal{D}_1 \cup \mathcal{D}_2, X)$ is executed next.

- $\mathcal{D}_3 = \{not\, u, not\, r\}$ is computed (since $defends(\mathcal{D}_3, \Delta_0 \cup \mathcal{D}_1 \cup \mathcal{D}_2)$ holds).

The computation of $\mathcal{D}_2$ redundantly defends $not\, p$, which is already defended by $\mathcal{D}_1$. Moreover, the addition of $\mathcal{D}_2$ to $\Delta_0 \cup \mathcal{D}_1$ necessitates the computation of $\mathcal{D}_3$ to defend $\mathcal{D}_2$. It would have been sufficient to terminate with $X = \Delta_0 \cup \mathcal{D}_1 = \{not\, p, not\, r\}$.

To avoid these redundancies, it suffices to distinguish in the new set of assumptions, $\Delta \cup \mathcal{D}$, between those assumptions Δ that have just been "defended" by $\Delta \cup \mathcal{D}$ itself and those new assumptions $\mathcal{D}$ that have just been added to $\Delta \cup \mathcal{D}$ and require further defence. For this purpose, we employ a variant $adm^e(\Delta, \mathcal{D}, \Delta')$ of the predicate $adm(\Delta, \Delta')$.

Specification 6.1 Let $\langle T, Ab, \overline{} \rangle$ be an assumption-based framework. For any sets of assumptions $\Delta, \mathcal{D}, \Delta' \subseteq Ab$,

$$adm^e(\Delta, \mathcal{D}, \Delta') \leftrightarrow \Delta \cup \mathcal{D} \subseteq \Delta' \;\wedge\; [[defends(\Delta \cup \mathcal{D}, \Delta) \wedge closed(\Delta \cup \mathcal{D})] \rightarrow \Delta' \text{ is admissible}].$$

We refer to this specification, together with definition 4.2 of admissibility, as $\mathcal{S}pec_{adm^e}$.

The relationship between adm and adm^e is given by the following

Lemma 6.1 For all sets of assumptions Δ_0 and Δ,

1. if $\mathcal{S}pec_{adm^e} \wedge \mathcal{S}pec_{lower} \models adm^e(\{\}, \Delta_0, \Delta) \wedge closed(\Delta_0)$
   then $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$;

2. if $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$
   then $\mathcal{S}pec_{adm^e} \wedge \mathcal{S}pec_{lower} \models adm^e(\{\}, \Delta_0, \Delta)$

The top-most level of a procedure which computes the predicate adm^e is given by the logic program

$$Prog_{adm^e}: \boxed{\begin{array}{ll} adm^e(\Delta, \{\}, \Delta) & \\ adm^e(\Delta, \mathcal{D}, \Delta') \leftarrow & defends^e(\mathcal{D}', \Delta, \mathcal{D}), \\ & closed(\Delta \cup \mathcal{D} \cup \mathcal{D}'), \\ & adm^e(\Delta \cup \mathcal{D}, \mathcal{D}' - (\Delta \cup \mathcal{D}), \Delta') \end{array}}$$

where $defends^e$ is the variant of the predicate $defends$ specified as follows:

Definition 6.1 Let $\langle T,\ Ab,\ \overline{}\rangle$ be an assumption-based framework. For any sets of assumptions $\mathcal{D}', \Delta, \mathcal{D} \subseteq Ab$,

$$defends^e(\mathcal{D}', \Delta, \mathcal{D}) \leftrightarrow$$
$$\forall A \subseteq Ab\,[[A\,attacks\,\mathcal{D} \wedge closed(A)] \rightarrow \mathcal{D}' \cup \Delta \cup \mathcal{D}\,attacks\,A - (\Delta \cup \mathcal{D})].$$

Namely, $\mathcal{D}'$ defends the new part $\mathcal{D}$ of the currently accumulated set of assumptions $\Delta \cup \mathcal{D}$ by counter attacking assumptions in any attack not in $\Delta \cup \mathcal{D}$. We will refer to this specification together with the definitions of *attack*, *closed* and the set-theoretic constructs as $\mathcal{S}pec_{lower^e}$.

As for $Prog_{adm}$, $Prog_{adm^e}$ can be made more efficient by replacing the second clause by

$$\begin{array}{ll} adm^e(\Delta, \mathcal{D}, \Delta') \leftarrow & defends^e(\mathcal{D}', \Delta, \mathcal{D}), \\ & closure(\Delta \cup \mathcal{D} \cup \mathcal{D}', \mathcal{D}^*), \\ & adm^e(\Delta \cup \mathcal{D}, \mathcal{D}^* - (\Delta \cup \mathcal{D}), \Delta') \end{array}$$

where *closure* is specified as in section 5. Furthermore, as for $Prog_{adm}$, if the given assumption-based framework is flat, then $Prog_{adm^e}$ can be simplified by eliminating all calls to the predicate *closed*. Moreover, by adopting different execution strategies we obtain different proof procedures. If we apply SLD or SLDNF resolution with a PROLOG-style left-to-right selection rule and with a query

$$\leftarrow adm^e(\{\}, \Delta_0, X)$$

for some given Δ_0, the proof procedure can be understood in algorithmic terms as follows:

Let $\Delta = \{\}$ and $\mathcal{D} = \{\}$.

While $\mathcal{D}$ is not empty, repeatedly (and non-deterministically)

(1) find $\mathcal{D}'$ such that $defends^e(\mathcal{D}', \Delta, \mathcal{D})$ holds and $\Delta \cup \mathcal{D} \cup \mathcal{D}'$ is closed;

(2) replace Δ by $\Delta \cup \mathcal{D}$ and $\mathcal{D}$ by $\mathcal{D}' - (\Delta \cup \mathcal{D})$.

Return $X = \Delta$.

Under this proof procedure, given the theory in example 6.1 and $\Delta_0 = \{not\, p\}$, one possible execution of the goal $\leftarrow adm^e(\{\}, \Delta_0, X)$ is the following:

- $\mathcal{D}_1 = \{not\, r\}$ is computed (since $defends^e(\mathcal{D}_1, \{\}, \Delta_0)$ holds). The subgoal $\leftarrow adm^e(\Delta_0 \cup \{\}, \mathcal{D}_1, X)$ is executed next.
- $\mathcal{D}_2 = \{\}$ is computed (since $defends^e(\mathcal{D}_2, \Delta_0, \mathcal{D}_1)$ holds). The subgoal $\leftarrow adm^e(\Delta_0 \cup \mathcal{D}_1, \mathcal{D}_2, X)$ is executed next.
- $X = \Delta_0 \cup \mathcal{D}_1 = \{not\, p, not\, r\}$ is returned.

As in the case of $Prog_{adm}$, the program can be made more efficient by interleaving steps (1) and (2), using (2) to defend the new assumptions in $\mathcal{D}'$ as soon as they are generated by (1), without waiting for all of $\mathcal{D}'$ to be generated. The resulting proof procedure is an abstraction of the proof procedure for logic programs given by Eshghi and Kowalski [12].

The soundness of $Prog_{adm^e}$ is given by corollary 6.1 below, which follows directly from lemma 6.1 and from the following theorem:

Theorem 6.1 $\mathcal{S}pec_{adm^e} \wedge \mathcal{S}pec_{lower^e} \models Prog_{adm^e}$.

Corollary 6.1 For all $\Delta_0, \Delta \subseteq Ab$,
if $Prog_{adm^e} \wedge \mathcal{S}pec_{lower^e} \models adm^e(\{\}, \Delta_0, \Delta) \wedge closed(\Delta_0)$,
then $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

(I.e. if the execution of the program generates Δ as output, given a closed set of assumptions Δ_0 as input, then $\Delta_0 \subseteq \Delta$ and Δ is admissible).

The completeness of $Prog_{adm^e}$ is given by the following theorem.

Theorem 6.2 For all $\Delta_0, \Delta \subseteq Ab$,
if $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$,
then $Prog_{adm^e} \wedge \mathcal{S}pec_{lower^e} \models adm^e(\{\}, \Delta_0, \Delta)$.

The full proof procedure is obtained by adding to $Prog_{adm^e}$ a program $Prog_{lower^e}$ for computing $defends^e$, *closed* and the set-theoretic constructs, $\cup, \subseteq$, etc. In the next section 7 we give the top-most level of a (generalised) logic program for computing $defends^e$, which is sound but not complete.

7 Computing defences

The efficient implementation of the proof procedures described in the previous section depends upon the efficient computation of $defends^e$. The specification of $defends^e(\mathcal{D}', \Delta, \mathcal{D})$ requires $\mathcal{D}' \cup \Delta \cup \mathcal{D}$ to attack *all* closed attacks A against $\mathcal{D}$. In the general case, the number of such attacks A can be infinite, because any superset of an attack is also an attack. The following theorem shows that it is possible to restrict considerably the number of attacks A that need to be considered, by limiting attention to the closure of minimal attacks.

Definition 7.1 A set of assumptions $A \subseteq Ab$ is a *minimal closed attack* against another set of assumptions $\Delta \subseteq Ab$ if and only if

- A is a closed attack against Δ, and
- there exists no closed attack A' against Δ such that $A' \subset A$.

Note that, in the case of flat assumption-based frameworks, all attacks are automatically closed, and therefore all minimal attacks (with respect to set inclusion) are automatically minimal closed attacks.

Theorem 7.1 A set of assumptions $\Delta \subseteq Ab$ is admissible if and only if

1. Δ is closed, and
2. for each set of assumptions $A \subseteq Ab$,
   if A is a minimal closed attack against Δ, then Δ attacks $A - \Delta$.

Minimal closed attacks might be hard to compute, due to the minimality condition. Indeed, it is sufficient for the proof procedure to consider any superset of the set of all minimal closed attacks. In the remainder of this section, we will assume *lean-attacks* whose specification is given by:

Specification 7.1 Let $\langle T, Ab, \overline{} \rangle$ be an assumption-based framework. For any sets of assumptions $A, \Delta \subseteq Ab$

$$[lean\text{-}attacks(A,\Delta) \rightarrow A\, attacks\ \Delta] \wedge$$
$$[A \text{ is a minimal closed attack against } \Delta \rightarrow \exists A' \subseteq Ab\, [A' \supseteq A \wedge lean\text{-}attacks(A',\Delta)]].$$

We refer to this specification, together with the definitions of attack and minimal closed attack as $\mathcal{S}pec_{lean\text{-}attacks}$.

Directly from theorem 7.1, the specification of $defends^e$ can be expressed "more efficiently":

Corollary 7.1 For all sets of assumptions $\mathcal{D}', \Delta, \mathcal{D}$,
$$\mathcal{S}pec_{lower^e} \models defends^e(\mathcal{D}',\Delta,\mathcal{D}) \leftrightarrow$$
$$\forall A \subseteq Ab\, [lean\text{-}attacks(A,\mathcal{D}) \rightarrow \mathcal{D}' \cup \Delta \cup \mathcal{D}\, attacks\, A - (\Delta \cup \mathcal{D})].$$

Then, the following (generalised) logic program computes $defends^e$:

$$Prog_{defends^e}: \boxed{\begin{array}{l} defends^e(\mathcal{D}',\Delta,\mathcal{D}) \leftarrow \mathcal{D}' = \bigcup\{\mathcal{D}^* \mid \\ \qquad\qquad lean\text{-}attacks(A,\mathcal{D}) \wedge \\ \qquad\qquad lean\text{-}attacks(\mathcal{D}^*, A - (\Delta \cup \mathcal{D}))\} \end{array}}$$

The proof procedure $Prog_{adm^e} \wedge Prog_{defends^e}$ needs to be extended with a program $Prog_{rest}$ defining *lean-attacks*, *closed* and the set-theoretic constructs. Such a program will be different for different instances of the abstract assumption-based framework. (See section 8 for the logic programming case.)

Note that, in theory, it is possible that $Prog_{rest} \models lean\text{-}attacks(A,\Delta)$ and A is infinite, and $Prog_{defends^e} \wedge Prog_{rest} \models defends^e(\mathcal{D}',\Delta,\mathcal{D})$ and $\mathcal{D}'$ is infinite. In practice, however, concrete implementations of the proof procedure $Prog_{adm^e}$ $\wedge$ $Prog_{defends^e}$ $\wedge$ $Prog_{rest}$ might generate finite sets only. Therefore, any such implementation would be incomplete, as illustrated by the following example.

Example 7.1 Let $\langle T, Ab, \overline{} \rangle$ be the assumption-based framework corresponding to the following logic program

$$\{p(X) \leftarrow not\, p(f(X))\}$$

where the language also contains a constant 0. The following set of assumptions is admissible:

$$\Delta = \{not\, p(f^{2i}(0))|i \geq n\}$$

Consider the set $\Delta_0 = \{not\, p(0)\}$. Then, there exists an admissible set $\Delta \supseteq \Delta_0$. However, if the program $Prog_{adm^e}$ generates a finite set as a defence for a finite set, then the infinite set Δ can not be generated by a finite computation.

8 Comparisons

The proof procedure of section 6 is an abstraction of the proof procedure of [12] for logic programming, computing a superset of the set of minimal (closed) attacks against any given set of assumptions, using SLD resolution extended to accumulate assumptions. The proof procedure of [12] (or rather its corrected version presented in [8]) corresponds to the program $Prog_{adm^e} \wedge Prog_{defends^e} \wedge Prog_{rest}$ with $Prog_{rest} = Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld} \wedge Prog_{set}$, where $Prog_{set}$ defines the set-theoretic constructs and

$$Prog^{LP}_{lean\text{-}attacks}: \boxed{\begin{array}{ll} lean\text{-}attacks(A, \Delta) \leftarrow & not\, P \in \Delta, \\ & sld(T \cup A, P), \\ & A \subseteq Ab \end{array}}$$

$$Prog_{sld}: \boxed{\begin{array}{l} sld(X, P) \leftarrow (P \leftarrow B) \in X, sld(X, B) \\ sld(X, P \wedge Q) \leftarrow sld(X, P), sld(X, Q) \\ sld(X, true) \end{array}}$$

Note that $Prog_{sld}$ is the vanilla meta-interpreter for logic programming. We assume that clauses $p \leftarrow q_1 \wedge \ldots \wedge q_n$, with p atom, $q_1, \ldots, q_n$ literals and $n \geq 0$, are represented as $p \leftarrow q_1 \wedge \ldots \wedge q_n \wedge true$, and that hypotheses $not\, p$ are represented as unit clauses $not\, p \leftarrow true$, with $not\, p$ understood as a new atom.

It is easy to see that:

Theorem 8.1 Given $\langle T, Ab, \overline{} \rangle$ for logic programming and $A, \Delta \subseteq Ab$,
if $Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld} \wedge Prog_{set} \models lean\text{-}attacks(A, \Delta)$
then $\mathcal{S}pec_{lean\text{-}attacks} \models lean\text{-}attacks(A, \Delta)$.

Therefore, $Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld}$ is sound. It is not complete because it cannot compute infinite sets (see example 7.1). Moreover, it is not efficient. More efficient (sound) definitions of lean-attacks can be easily given, by modifying $Prog_{sld}$.

As a result of the soundness of $Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld}$, in the logic programming case $Prog_{adm^e} \wedge Prog_{defends^e} \wedge Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld} \wedge Prog_{set}$ is sound. It is not complete because $Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld}$ is not complete.

As already mentioned in section 6, the proof procedure of [12] adopts an interleaved, rather than PROLOG-style, execution of the given logic program. Our proof procedure is an abstract version of the one in [12] in two ways: it is not restricted to the logic programming case only and it is not restricted to any execution strategy.

Toni and Kakas [34] also develop abstract argumentation-theoretic proof procedures, which abstract the one in [12], for computing admissibility as well as weak stability [17] and acceptability semantics [18] in the abstract case and, by instantiation, in the special cases we have considered in this paper. Their proof procedure for computing admissibility corresponds to the proof procedure in section 6 but is formulated in terms of derivations of trees (whose nodes are sets of assumptions) rather than as a logic program. In this paper and in [10] we have shown how such abstract proof procedures in logic programming form can be derived systematically from their logical specifications.

Although our proof procedure aims to compute credulous default consequences relative to the admissibility semantics, it can also be used as a basis for computing (credulous and/or sceptical) default consequences relative to other semantics. In particular, since every admissible extension is contained in a maximal admissible extension, our proof procedure is sound with respect to (credulous) maximal admissible extension semantics, i.e. partial stable model [30]/preferred extension semantics [8] in the logic programming case.

Thielscher [33] defines a proof procedure which is sound for computating sceptical consequences relative to the partial stable model semantics for logic programming. This proof procedure is formulated in [33] in imperative style, but can be reformulated as the following (generalised) logic program

$$Prog_{all\text{-}pref}: \quad \boxed{\begin{array}{l} all\text{-}pref(X) \leftarrow sld(T \cup \Delta_0, X) \wedge \\ \qquad adm^e(\{\}, \Delta_0, \Delta_0') \wedge \\ \qquad\quad \forall A \subseteq Ab\,[[lean\text{-}attacks(A, \Delta_0') \wedge \\ \qquad\qquad adm^e(\{\}, A, \Delta)] \rightarrow \\ \qquad\qquad \exists \Delta' \subseteq Ab\,[\ adm^e(\{\}, \Delta, \Delta') \rightarrow \\ \qquad\qquad\qquad sld(T \cup \Delta', X)]]. \end{array}}$$

together with $Prog_{adm^e} \wedge Prog_{defends^e} \wedge Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld} \wedge Prog_{set}$ given earlier. Intuitively, a sentence X (conjunction of literals) is a sceptical default consequence of a logic program T under the preferred extension semantics if X holds in some admissible set of assumptions Δ_0' (built from an existing "explanation" Δ_0 for X) and, for every admissible set of assumptions Δ which is a superset of a lean attack against Δ_0', X holds with respect to some admissible superset Δ' of Δ.

This proof procedure is sound in the following sense: if $Prog_{all\text{-}pref} \wedge Prog_{adm^e} \wedge Prog_{defends^e} \wedge Prog^{LP}_{lean\text{-}attacks} \wedge Prog_{sld} \wedge Prog_{set} \models all\text{-}pref(X)$ then $X \in Th(T \cup \Delta)$ for all maximal admissible sets of assumptions Δ. The proof of soundness can be found in the appendix.

Many of the procedures for computing stable semantics in the literature (e.g. [30, 25, 24]) can be obtained by appropriately instantiating and executing (more efficient versions of) the following procedure:

$$\boxed{\begin{array}{l} stable(\Delta, \Delta) \leftarrow fullcover(\Delta), not\, attacks(\Delta, \Delta) \\ stable(\Delta_0, \Delta) \leftarrow expand(\Delta_0, \Delta_1), nocover(\Delta_1, \alpha), stable(\Delta_1 \cup \{\alpha\}, \Delta) \end{array}}$$

where $attacks(\Delta, \Delta)$ stands for "Δ attacks Δ" and the predicates $fullcover$ and $nocover$ are defined as follows, for all $\Delta \subseteq Ab$ and $\alpha \in Ab$:

$$fullcover(\Delta) \leftrightarrow \forall \alpha \in Ab\ [\alpha \in \Delta \vee attacks(\Delta, \alpha)]$$
$$nocover(\Delta, \alpha) \leftrightarrow [\alpha \notin \Delta \wedge not\, attacks(\Delta, \alpha)]$$

where $attacks(\Delta, \alpha)$ stands for "Δ attacks α". Different definitions for the predicate *expand* give rise to different proof procedures. For example

$$expand(\Delta_0, \Delta_1) \leftrightarrow adm(\Delta_0, \Delta_1).$$

Alternatively

$$expand(\Delta_0, \Delta_1) \leftrightarrow \Delta_1 = lfp(Def_{\Delta_0})$$

where lfp stands for "least fixed point" and the operator Def_Δ, for any set of assumptions Δ, is defined as follows: given an assumption-based framework $\langle T, Ab, \overline{} \rangle$ and $\mathcal{D} \subseteq Ab$,

$Def_\Delta(\mathcal{D}) = \Delta \cup Def(\Delta \cup \mathcal{D})$
$Def(\mathcal{D}) = \{\alpha \,|\, defends(\mathcal{D}, \{\alpha\})\}$.

The (credulous version of the) proof procedure of [25] for logic programming, employs a version of the second definition of *expand*.

Other semantics for default reasoning have been proposed, e.g. the complete extension semantics [3], where a set of assumptions Δ is *complete* if and only if Δ is closed and $\Delta = Def(\Delta)$. [5] Complete extensions correspond to complete scenaria [8] and stationary expansions [28] for logic programming. Moreover, the well-founded semantics [35] of logic programming can be understood as the sceptical semantics corresponding to complete extensions.

Our proof procedures can be employed to give proof procedures to compute credulous default consequences under the complete extension semantics, for example:

$$\boxed{\begin{array}{l} cpl(\Delta, \Delta) \leftarrow complete(\Delta) \\ cpl(\Delta_0, \Delta) \leftarrow adm(\Delta_0, \Delta_1), defends(\Delta_1, \{a\}), cpl(\Delta_1 \cup \{a\}, \Delta) \end{array}}$$

where $complete(\Delta)$ stands for Δ is complete.

9 Conclusion and future work

We have presented proof procedures for the credulous admissibility semantics for the abstract, argumentation framework presented in [3]. These proof procedures are applicable to all instances of the framework, including logic programming, de-

[5] Note that Δ is complete if and only if Δ is closed, Δ defends itself ($\Delta \subseteq Def(\Delta)$) and Δ contains every assumption it defends ($\Delta \supseteq Def(\Delta)$), i.e. Δ is complete if and only if Δ is admissible and Δ contains every assumption it defends.

fault logic, autoepistemic logic and non-monotonic modal logic. We have sketched a possible instance for logic programming.

We have compared our proof procedures with existing, state-of-the-art proof procedures, and analysed how our proof procedures can be used as the kernel of proof procedures to compute other semantics, notably the stable semantics.

We have concentrated on the logic underlying the proof procedures, and have not considered implementation-related issues. Future work includes implementations of the (efficient version of the) proof procedure for special instances of the abstract framework, possibly exploiting some of the techniques of [24, 25].

Acknowledgements

This research was supported by the Fujitsu Research Laboratories and by the EEC activity KIT011-LPKRR.

References

[1] A.B. Baker, M.L. Ginsberg, A theorem prover for prioritised circumscription. *Proc. IJCAI'89*, Morgan Kaufmann, pages 463–467

[2] M.D. Barback, J. Lobo, A resolution-based procedure for default theories with extensions. *Proc. Non-monotonic Extensions of Logic Programs: Theory and Applications* (J. Dix, L.M. Pereira and T. Przymusinski eds.) Springer-Verlag, LNAI 927 (1995) pages 101–126

[3] A. Bondarenko, P.M. Dung, R.A. Kowalski, F. Toni, An abstract, argumentation-theoretic framework for default reasoning. *Artificial Intelligence* 93 (1,2) (1997) pages 63–101

[4] A. Bondarenko, F. Toni, R.A. Kowalski, An assumption-based framework for non-monotonic reasoning. *Proc. 2nd International Workshop on Logic Programming and Non-monotonic Reasoning* (A. Nerode and L. Pereira eds.) MIT Press (1993) pages 171–189

[5] G. Brewka, J. Dix. K. Konolige, *Nonmonotonic reasoning: an overview.* CLSI Lecture Notes 73 (Standford, California) (1997)

[6] W. Chen, D.S. Warren, Computation of stable models and its integration with logical query processing.
Available by ftp `ftp.ms.uky.edu:pub/lpnmr/chen2.ps` (1994)

[7] Y. Dimopoulos, A.C. Kakas, Logic programming without negation as failure. *Proc. ILPS'95*, MIT Press

[8] P.M. Dung, An argumentation theoretic foundation of logic programming *J. Logic Programming* 22 (1995), 151-177

[9] P.M. Dung, The acceptability of arguments and its fundamental role in non-monotonic reasoning and logic programming and n-person game *Artificial Intelligence* 77 (1995), 321-357

[10] P.M. Dung, R.A. Kowalski, F. Toni, Synthesis of proof procedures for default reasoning. *Proc. LOPSTR'96*, Springer Verlag LNCS 1207

[11] K. Eshghi, R.A. Kowalski, Abduction through deduction. Imperial College Technical Report (1988)

[12] K. Eshghi, R.A. Kowalski, Abduction compared with negation as failure. *Proc. ICLP'89*, MIT Press

[13] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming. *Proc. ICSLP'88*, MIT Press

[14] M.L. Ginsberg, A circumscriptive theorem prover. *Artificial Intelligence* 39 (1989) pages 209–230

[15] A.C. Kakas, R.A. Kowalski, F. Toni, Abductive logic programming. *Journal of Logic and Computation* 2(6) (1993) pages 719-770

[16] A.C. Kakas, R.A. Kowalski, F. Toni, The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming* 5 (1998) pages 235-324, Oxford University Press

[17] A.C. Kakas, P. Mancarella, Stable theories for logic programs. *Proc. ISLP'91*, MIT Press

[18] A.C. Kakas, P. Mancarella, P.M. Dung, The acceptability semantics for logic programs. *Proc. ICLP'94*, MIT Press, pages 504–519

[19] R. A. Kowalski, F. Toni, Abstract argumentation. *Journal of Artificial Intelligence and Law* 4(3–4) pages 275–296, 1996, Special Issue on Logical Models of Argumentation, Kluwer Academic Publishers

[20] V.W. Marek, M. Truszczynski, *Nonmonotonic logic: context-dependent reasoning.* Springer Verlag (1993)

[21] J. McCarthy, Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence* (13) (1980) pages 27–39

[22] D. McDermott, Nonmonotonic logic II: non-monotonic modal theories. *JACM* 29(1) (1982)

[23] R. Moore, Semantical considerations on non-monotonic logic. *Artificial Intelligence* 25 (1985)

[24] N. Niemelä, Towards efficient default reasoning. *Proc. IJCAI'95*, Morgan Kaufman, pages 312–318

[25] N. Niemelä, P. Simons, Efficient implementation of the well-founded and stable model semantics. *Proc. JICSLP'96*, MIT Press, pages 289–303.

[26] D. Poole, A logical framework for default reasoning. *Artificial Intelligence* 36 (1988)

[27] T. Przymusinski, An algorithm to compute circumscription. *Artificial Intelligence* 38 (1989) pages 49–73

[28] T. Przymusinski, Semantics of disjunctive logic programs and deductive databases. *Proc. DOOD '91*

[29] R. Reiter, A logic for default reasoning. *Artificial Intelligence* 13 (1980)

[30] D. Saccà, C. Zaniolo, Stable models and non-determinism for logic programs with negation. *Proc. ACM SIGMOD-SIGACT Symposium on Principles of Database Systems* (1990)

[31] K. Satoh and N. Iwayama. A correct top-down proof procedure for general logic programs with integrity constraints. *Proc. 3rd International Workshop on Extensions of Logic Programming*, (E. Lamma, P. Mello, eds) Springer Verlag LNAI 660 (1992) pages 19–34

[32] K. Satoh, A top-down proof procedure for default logic by using abduction. *Proc. ECAI'94*

[33] M. Thielscher, A nonmonotonic disputation-based semantics and proof procedure for logic programs. *JICSLP'96*, MIT Press, pages 483–497

[34] F. Toni, A.C. Kakas, Computing the acceptability semantics. *Proc. 3rd International Workshop on Logic Programming and Non-monotonic Reasoning,* (V. W. Marek, A. Nerode and M. Truszczynski eds.) Springer Verlag LNAI 928 pages 401–415 (1995)

[35] A. Van Gelder, K.A. Ross, J.S. Schlipf, Unfounded sets and the well-founded semantics for general logic programs. *Proc. ACM SIGMOD-SIGACT, Symposium on Principles of Database Systems* (1988)

Appendix

Alternative characterisation of admissibility

Proof of theorem 4.1

$\Rightarrow$ Given a closed attack A against Δ, we need to prove only that Δ attacks $A - \Delta$. Since Δ is admissible, Δ attacks A. But, if Δ attacks $A \cap \Delta$, then Δ attacks itself, contradicting the hypothesis that Δ is admissible.

$\Leftarrow$ We need to prove only that Δ does not attack itself. Suppose that Δ attacks itself. Then, Δ attacks $\Delta - \Delta = \{\}$. But, by definition of attack, no set can attack $\{\}$.

Soundness and completeness of $Prog_{adm}$

Proof of theorem 5.1

We prove the theorem by deriving the program $Prog_{adm}$ from the specification. By letting $\Delta_0 = \Delta$ in $\mathcal{S}pec_{adm}$, as formulated in corollary 5.1

$adm(\Delta_0, \Delta) \leftrightarrow \Delta_0 \subseteq \Delta \wedge defends(\Delta, \Delta) \wedge closed(\Delta)$

we immediately obtain the first clause of the program.

To obtain the second clause, we let $\Delta_0 = \Delta'_0 \cup \mathcal{D}$ in the only-if half of $\mathcal{S}pec_{adm}$, as formulated in corollary 5.1, and observe that $\Delta'_0 \cup \mathcal{D} \subseteq \Delta$ implies $\Delta'_0 \subseteq \Delta$, obtaining

$adm(\Delta'_0 \cup \mathcal{D}, \Delta) \rightarrow \Delta'_0 \subseteq \Delta \wedge defends(\Delta, \Delta) \wedge closed(\Delta)$.

Then, by applying the if half of $\mathcal{S}pec_{adm}$, by transitivity of $\rightarrow$, we obtain

$adm(\Delta'_0 \cup \mathcal{D}, \Delta) \rightarrow adm(\Delta'_0, \Delta)$

which implies

$adm(\Delta'_0 \cup \mathcal{D}, \Delta) \wedge closed(\Delta'_0 \cup \mathcal{D}) \wedge defends(\mathcal{D}, \Delta'_0) \rightarrow adm(\Delta'_0, \Delta)$.

By renaming Δ'_0 to Δ and Δ to Δ', we obtain the second clause of the program.

Proof of theorem 5.2

Assume $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$. Then,

$\mathcal{S}pec_{lower} \models \Delta_0 \subseteq \Delta \wedge defends(\Delta, \Delta) \wedge closed(\Delta)$.

Then, by the first clause of $Prog_{adm}$

$Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta, \Delta)$.

There are two cases: (1) $\Delta_0 = \Delta$ and (2) $\Delta_0 \subset \Delta$.

In the first case, $Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$ immediately.

In the second case, since any defence of a set Δ also defends any subset of Δ,

$\mathcal{S}pec_{lower} \models defends(\Delta, \Delta_0) \wedge closed(\Delta)$.

Then,

$Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta, \Delta) \wedge defends(\Delta, \Delta_0) \wedge closed(\Delta)$.

But $\Delta = \Delta \cup \Delta_0$. Therefore,

$Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta \cup \Delta_0, \Delta) \wedge defends(\Delta, \Delta_0) \wedge closed(\Delta \cup \Delta_0)$.

But then, by the second clause of $Prog_{adm}$,

$Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

Proof of theorem 5.3

Assume $Prog_{adm} \wedge Prog_{lower} \models adm(\Delta_0, \Delta)$. Then, since $Prog_{lower}$ is sound with respect to $\mathcal{S}pec_{lower}$,

$Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

Then, directly from corollary 5.2,

$\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$.

Proof of theorem 5.4

Assume $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$. Then, directly from theorem 5.2,

$Prog_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$. By completeness of $Prog_{lower}$,

$Prog_{adm} \wedge Prog_{lower} \models adm(\Delta_0, \Delta)$.

Soundness and completeness of $Prog_{adm^e}$

Proof of lemma 6.1

1. First, note that, $adm^e(\{\}, \Delta_0, \Delta) \wedge closed(\Delta_0)$ implies

   $\Delta_0 \subseteq \Delta \ \wedge [[defends(\Delta_0, \{\}) \wedge closed(\Delta_0)] \rightarrow \Delta$ is admissible$] \wedge closed(\Delta_0)$.
   But $\mathcal{S}pec_{lower}$ trivially implies $defends(\Delta_0, \{\})$. Therefore

   $\Delta_0 \subseteq \Delta \ \wedge [closed(\Delta_0) \rightarrow \Delta$ is admissible$] \wedge closed(\Delta_0)$
   which, in $\mathcal{S}pec_{adm}$, implies $adm(\Delta_0, \Delta)$.

2. $adm(\Delta_0, \Delta)$ implies

   $\Delta_0 \subseteq \Delta \ \wedge \Delta$ is admissible.
   This trivially implies

   $\Delta_0 \subseteq \Delta \ \wedge [[closed(\Delta_0) \wedge defends(\Delta, \{\})] \rightarrow \Delta$ is admissible$]$
   which, in $\mathcal{S}pec_{adm^e}$ implies

   $adm^e(\{\}, \Delta_0, \Delta)$.

Proof of theorem 6.1
We prove the theorem by deriving the program $Prog_{adm^e}$ from the specification and by exploiting the following corollary, which follows directly from theorem 4.1:

Corollary 9.1
$\mathcal{S}pec_{adm^e} \wedge \mathcal{S}pec_{lower^e} \models adm^e(\Delta, \mathcal{D}, \Delta') \leftrightarrow \Delta \cup \mathcal{D} \subseteq \Delta' \;\wedge$
$[[defends^e(\mathcal{D}, \Delta, \Delta) \wedge closed(\Delta \cup \mathcal{D})] \rightarrow \Delta' \text{ is admissible}]$.

Then, by letting $\mathcal{D} = \{\}$ and $\Delta = \Delta_0$ in $\mathcal{S}pec_{adm^e}$

$$adm^e(\Delta_0, \mathcal{D}, \Delta) \leftrightarrow \Delta_0 \cup \mathcal{D} \subseteq \Delta \;\wedge$$
$$[defends(\Delta_0 \cup \mathcal{D}, \Delta_0) \wedge closed(\Delta_0 \cup \mathcal{D}) \rightarrow \Delta \text{ is admissible}]$$

we obtain

$$adm^e(\Delta, \{\}, \Delta) \leftrightarrow \Delta \subseteq \Delta \wedge [defends(\Delta, \Delta) \wedge closed(\Delta) \rightarrow \Delta \text{ is admissible}]$$

equivalent to the first clause of $Prog_{adm^e}$ because of corollary 5.1.
To obtain the second clause, first replace the predicate adm^e in the second clause of the program by the following equivalent specification in terms of $defends^e$ given by corollary 9.1, obtaining

$$[\Delta \cup \mathcal{D} \subseteq \Delta' \;\wedge [[defends^e(\mathcal{D}, \Delta, \Delta) \wedge closed(\Delta \cup \mathcal{D})] \rightarrow \Delta' \text{ is admissible}]] \leftarrow$$
$$[defends^e(\mathcal{D}', \Delta, \mathcal{D}) \wedge closed(\Delta \cup \mathcal{D} \cup \mathcal{D}') \wedge \Delta \cup \mathcal{D} \cup \mathcal{D}' \subseteq \Delta' \wedge$$
$$[[defends^e(\mathcal{D}' - (\Delta \cup \mathcal{D}), \Delta \cup \mathcal{D}, \Delta \cup \mathcal{D}) \wedge closed(\Delta \cup \mathcal{D} \cup \mathcal{D}')] \rightarrow$$
$$\Delta' \text{ is admissible}]].$$

Since

$$\Delta \cup \mathcal{D} \subseteq \Delta' \leftarrow \Delta \cup \mathcal{D} \cup \mathcal{D}' \subseteq \Delta'$$

it is obvious that we need to prove only the following

$$\Delta' \text{ is admissible} \leftarrow$$
$$[defends^e(\mathcal{D}, \Delta, \Delta) \wedge closed(\Delta \cup \mathcal{D}) \wedge$$
$$defends^e(\mathcal{D}', \Delta, \mathcal{D}) \wedge closed(\Delta \cup \mathcal{D} \cup \mathcal{D}') \wedge$$
$$[defends^e(\mathcal{D}' - (\Delta \cup \mathcal{D}), \Delta \cup \mathcal{D}, \Delta \cup \mathcal{D}) \wedge closed(\Delta \cup \mathcal{D} \cup \mathcal{D}')$$
$$\rightarrow \Delta' \text{ is admissible}]]$$

which follows immediately from the following

Lemma
$defends^e(\mathcal{D}, \Delta, \Delta) \wedge defends^e(\mathcal{D}', \Delta, \mathcal{D}) \rightarrow defends^e(\mathcal{D}' - (\Delta \cup \mathcal{D}), \Delta \cup \mathcal{D}, \Delta \cup \mathcal{D})$.

Proof of Lemma: Assume (i) $defends^e(\mathcal{D}, \Delta, \Delta)$ and (ii) $defends^e(\mathcal{D}', \Delta, \mathcal{D})$. Assume $A \subseteq Ab$ attacks $\Delta \cup \mathcal{D}$. We need to show that $\mathcal{D}' \cup \Delta \cup \mathcal{D}$ attacks $A - (\Delta \cup \mathcal{D})$.

- If A attacks $\mathcal{D}$ then, by (ii)$\mathcal{D}' \cup \Delta \cup \mathcal{D}$ attacks $A - (\Delta \cup \mathcal{D})$.
- If A attacks Δ then, by (i), $\mathcal{D} \cup \Delta$ attacks $A - \Delta$. It suffices to show that $\mathcal{D} \cup \Delta$ does not attack $\mathcal{D}$. Suppose, on the contrary, that $\mathcal{D} \cup \Delta$ attacks $\mathcal{D}$. Then, by (ii), $\mathcal{D}' \cup \Delta \cup \mathcal{D}$ attacks $(\mathcal{D} \cup \Delta) - (\mathcal{D} \cup \Delta) = \{\}$. But this is not possible, because there are no attacks against $\{\}$.

Proof of theorem 6.2

Assume $\mathcal{S}pec_{adm} \wedge \mathcal{S}pec_{lower} \models adm(\Delta_0, \Delta)$. Then

$\mathcal{S}pec_{lower} \models \Delta_0 \subseteq \Delta \wedge defends(\Delta, \Delta) \wedge closed(\Delta)$, and

$\mathcal{S}pec_{lower^e} \models \Delta_0 \subseteq \Delta \wedge defends^e(\Delta, \{\}, \Delta) \wedge closed(\Delta)$.

Moreover, $\mathcal{S}pec_{lower^e} \models [[\Delta_0 \subseteq \Delta \wedge defends^e(\Delta, \{\}, \Delta)] \rightarrow defends^e(\Delta, \{\}, \Delta_0)]$.

Therefore, (i) $\mathcal{S}pec_{lower^e} \models defends^e(\Delta, \{\}, \Delta_0)$.

Similarly, $\mathcal{S}pec_{lower^e} \models [\Delta_0 \subseteq \Delta \wedge defends^e(\Delta, \{\}, \Delta) \rightarrow defends^e(\Delta, \Delta_0, \Delta - \Delta_0)]$.

Therefore, (ii) $\mathcal{S}pec_{lower^e} \models defends^e(\Delta, \Delta_0, \Delta - \Delta_0)$.

To show $Prog_{adm^e} \wedge \mathcal{S}pec_{lower^e} \models adm^e(\{\}, \Delta_0, \Delta)$, use the following instance of the second clause of the program:

$$\begin{aligned} adm^e(\{\}, \Delta_0, \Delta) \leftarrow\ & defends^e(\mathcal{D}', \{\}, \Delta_0), \\ & closed(\Delta_0 \cup \mathcal{D}'), \\ & adm^e(\Delta_0, \mathcal{D}' - \Delta_0, \Delta) \end{aligned}$$

Let $\mathcal{D}' = \Delta$. Then, the first condition is provable from $\mathcal{S}pec_{lower^e}$ by (i), and the second condition is provable from $\mathcal{S}pec_{lower^e}$ since $\Delta_0 \subseteq \Delta$ and Δ is closed. To prove the third condition, use the following instance of the second clause of the program:

$$\begin{aligned} adm^e(\Delta_0, \Delta - \Delta_0, \Delta) \leftarrow\ & defends^e(\mathcal{D}', \Delta_0, \Delta - \Delta_0), \\ & closed(\Delta \cup \mathcal{D}'), \\ & adm^e(\Delta, \mathcal{D}' - \Delta, \Delta) \end{aligned}$$

Let $\mathcal{D}' = \Delta$. Then, the first condition is provable from $\mathcal{S}pec_{lower^e}$ by (ii), and the second condition is provable from $\mathcal{S}pec_{lower^e}$ since Δ is closed. Moreover, the third condition is provable by the first clause of $Prog_{adm^e}$. Therefore,

$Prog_{adm^e} \wedge \mathcal{S}pec_{lower^e} \models adm^e(\{\}, \Delta_0, \Delta)$.

Computing $defends^e$

Proof of theorem 7.1

The theorem follows directly from $1 \Leftrightarrow 2$ below, for any sets of assumptions $\Delta_1, \Delta_2, \Delta_3$:

1. for all $\Delta' \subseteq Ab$,
   if A is a closed attack against Δ_1 then Δ_2 attacks $A - \Delta_3$

2. for all $A \subseteq Ab$,
   if A is a minimal closed attack against Δ_1 then Δ_2 attacks $A - \Delta_3$.

$1 \Rightarrow 2$ since every minimal closed attack against a set is a closed attack against the set.

$2 \Rightarrow 1$ since every non-minimal closed attack contains some minimal closed attack, and if one set attacks another, then the first set attacks any superset of the second set.

Soundness of $Prog_{all\text{-}pref}$

Soundness of the proof procedure $Prog_{all\text{-}pref}$ ($\wedge$ $Prog_{adm^e}$ $\wedge$ $Prog^{LP}_{lean\text{-}attacks}$ $\wedge$ $Prog_{sld}$ $\wedge$ $Prog_{set}$) is sanctioned by the following theorem, which is a reformulation of theorem 16 in [33].

Theorem 9.1 Given an (abstract) assumption-based framework $\langle T, Ab, \overline{} \rangle$ and a sentence α, $\alpha \in Th(T \cup \Delta)$ for all maximal admissible $\Delta \subseteq Ab$, if and only if

1. $\alpha \in Th(T \cup \Delta_0)$, for some admissible set of assumptions $\Delta_0 \subseteq Ab$, and

2. for every set of assumptions $\Delta \subseteq Ab$,
   if Δ is admissible and Δ attacks Δ_0,
   then $\alpha \in Th(T \cup \Delta')$ for some set of assumptions $\Delta' \subseteq Ab$ such that

   (a) $\Delta' \supseteq \Delta$, and

   (b) Δ' is admissible.

Proof: The only if half is trivial. The if half is proved by contradiction:

Suppose there exists a set of assumptions Δ^* such that Δ^* is preferred and $\alpha \notin Th(T \cup \Delta^*)$.

If Δ_0 is the set of assumptions provided in part 1 and $\Delta_0 = \{\}$, then $\alpha \in Th(T)$ and therefore $\alpha \in Th(T \cup \Delta^*)$, thus contradicting the hypothesis that $\alpha \notin Th(T \cup \Delta^*)$. Therefore, $\Delta_0 \neq \{\}$.

If $\Delta^* \cup \Delta_0$ does not attack itself, then $\Delta^* \cup \Delta_0$ is admissible, thus contradicting the hypothesis that Δ^* is preferred (and therefore maximal admissible).

Therefore, $\Delta^* \cup \Delta_0$ attacks itself. Thus, either (i) $\Delta^* \cup \Delta_0$ attacks Δ_0 or (ii) $\Delta^* \cup \Delta_0$ attacks Δ^*.

In case (i), since Δ_0 does not attack itself, Δ^* attacks Δ_0 and therefore, by part 2, since Δ^* is preferred, $\alpha \in Th(T \cup \Delta^*)$, thus contradicting the hypothesis.

In case (ii), since Δ^* is admissible, Δ^* attacks $\Delta^* \cup \Delta_0$ therefore, since Δ_0 does not attack itself, Δ_0 attacks Δ^* thus, since Δ^* is admissible, Δ^* attacks Δ_0 and, by part 2, since Δ^* is preferred, $\alpha \in Th(T \cup \Delta^*)$, thus contradicting the hypothesis.