Synthesis of proof procedures
for default reasoning

Phan Minh Dung\(^1\), Robert A. Kowalski\(^2\) and Francesca Toni\(^3\)

\(^1\) Asian Institute of Technology, Division of Computer Science
    PO Box 2754, Bangkok 10501, Thailand
    dung@cs.ait.ac.th

\(^2\) Imperial College, Department of Computing
    180 Queen's Gate, London SW7 2BZ, UK
    rak@doc.ic.ac.uk

\(^3\) National Technical University of Athens
    Department of Electrical and Computer Engineering, Division of Computer Science
    15773 Zographou, Athens, Greece
    ftoni@softlab.ece.ntua.gr

Abstract. We apply logic program development technology to define
abstract proof procedures, in the form of logic programs, for computing
the admissibility semantics for default reasoning proposed in [2].
The proof procedures are derived from a formal specification. The derivation
 guarantees the soundness of the proof procedures. The completeness
of the proof procedures is shown by employing a technique of symbolic
execution of logic programs to compute (an instance of) a relation implied
by the specification.

1 Introduction

In [2], we have shown that many default logics [13, 19, 14, 15] can be understood as special cases of a single abstract framework, based upon an abductive interpretation of the semantics of logic programming [7, 8] and its abstractions [4, 5, 1, 11], and extending Theorist [18]. Moreover, we have proposed a new semantics for default logics, more liberal than their standard semantics and generalising the admissibility semantics for logic programming [4], equivalent to the partial stable model semantics [20] (see [16]).

In this paper, we define two proof procedures for computing the abstract admissibility semantics. The second proof procedure is a computationally more efficient refinement of the first. Both procedures generalise and abstract a proof procedure [8] for logic programming, but are formulated as logic programs. The relationships of the proof procedures with other existing proof procedures for default reasoning and the relevance of the proof procedures in the field of default reasoning are discussed in an extended version of this paper [6]. In the present paper, we describe the technology used to define the abstract proof procedures. Both are derived from a formal specification by conventional techniques of deductive synthesis of logic programs (e.g. those described already in [12], Chapter 10, and, more recently, in [3]). The derivation guarantees the soundness
of the proof procedures. The completeness of the proof procedures is shown via symbolic execution of the logic programs to compute (an instance of) a relation implied by the specification.

The logic programs are derived top-down in two stages: the top-most level is derived first, relative to lower-level predicates that can then be "developed". The top-level program is proved correct and complete, parametrically with respect to the lower-level predicates. (Generalised) logic programs computing the lower-level predicates are given in [6].

The rest of the paper has the following structure: Section 2 revises the main features of the abstract framework and the admissibility semantics; Section 3 introduces the top-level of the first abstract proof procedure to compute the admissibility semantics; Section 4 introduces the top-level of the more efficient proof procedure; Section 5 gives conclusions.

2 Argumentation-theoretic framework and admissibility semantics

An argumentation-theoretic framework consists of a set of sentences $T$, the theory, viewed as a given set of beliefs, a (non-empty) set of sentences $Ab$, viewed as assumptions that can be used to extend $T$, and a notion of attack, namely a (binary) relation between sets of assumptions.

Both theory and assumptions are formulated in some underlying language provided with a notion of derivability $Th$ which is monotonic, in the sense that $T \subseteq T'$ implies $Th(T) \subseteq Th(T')$, and compact, in the sense that $\alpha \in Th(T)$ implies $\alpha \in Th(T')$ for some finite subset $T'$ of $T$.

The notion of attack is monotonic, in the sense that, for any sets of assumptions $A, A', \Delta, \Delta' \subseteq Ab$, if $A$ attacks $\Delta$ then:

- $A'$ attacks $\Delta$ for any $A' \supseteq A$;
- $A$ attacks $\Delta'$ for any $\Delta' \supseteq \Delta$.

Moreover, the notion of attack satisfies the property that no set of assumptions attacks the empty set of assumptions.

Theorist [18], circumscription [13], logic programming, default logic [19], autopoietic logic [15] and non-monotonic modal logic [14] are all instances of the abstract argumentation-theoretic framework (see [2]).

A set of assumptions $\Delta \subseteq Ab$ is closed iff $\Delta = Ab \cap Th(T \cup \Delta)$.

An argumentation-theoretic framework is flat iff every set of assumptions is closed. The frameworks for logic programming and default logic are flat.

A set of assumptions $\Delta$ is

- admissible iff $\Delta$ is closed, $\Delta$ does not attack itself and
  for each closed $A \subseteq Ab$, if $A$ attacks $\Delta$ then $\Delta$ attacks $A$.

Admissible sets of assumptions correspond to admissible scenarios for logic programming [4]. The standard semantics of scenario in Theorist [18], extensions
in default logic [19], stable expansions in autoepistemic logic [15], fixed points in non-monotonic modal logic [14] and stable models in logic programming [9] correspond to the less liberal notion of stable sets of assumptions, i.e. sets of assumptions which are admissible and attack every assumption they do not contain.  

The semantics of admissible and stable sets of assumptions are credulous, in the sense that a sentence $\delta$ is a non-monotonic consequence of a theory $T$ if $\delta$ belongs to some extension sanctioned by the semantics. Corresponding to every credulous semantics there is a sceptical semantics in which $\delta$ is a non-monotonic consequence of $T$ if $\delta$ belongs to all extensions sanctioned by the semantics. Many cases of circumscription [13] can be understood as the sceptical semantics corresponding to stable sets of assumptions.

In this paper we focus upon the computation of non-monotonic consequences using the (credulous) admissibility semantics. We define proof procedures for computing the admissibility semantics for any abstract argumentation-theoretic framework.

3 Proof procedure for admissibility

The procedure is defined in the form of a metalogic program, the top-level clauses of which define the predicate $\text{adm}$, whose specification is given as follows:

**Definition 1.** Let $(T, \text{Ab}, \text{attacks})$ be an argumentation-theoretic framework. For any sets of assumptions $\Delta_0, \Delta \subseteq \text{Ab}$

$$\text{adm}(\Delta_0, \Delta) \iff [\Delta_0 \subseteq \Delta \land \Delta \text{ is admissible}].$$

Typically, the set $\Delta_0$ will be given, such that $T \cup \Delta_0 \vdash \alpha$ for some formula $\alpha \in \mathcal{L}$, and the problem will be to generate $\Delta$ such that $\text{adm}(\Delta_0, \Delta)$. Consequently, $T \cup \Delta \vdash \alpha$ as well, and the set $\Delta$ provides an admissible "explanation" for the query $\alpha$.

This characterisation of the predicate $\text{adm}$ provides a specification for the proof procedure. In the remainder of this section, this specification together with the definition of admissibility given earlier will be referred to as $\text{Spec}_{\text{adm}}$. The logic program providing the proof procedure will consist of top-level clauses defining $\text{adm}$ and lower-level clauses, defining the predicate $\text{defends}$ given later in the section, in definition 3. The predicate $\text{adm}$ takes names of sets of sentences as arguments, and is therefore a metapredicate.  

We focus on the top-level of the program. This part of the program will be derived from $\text{Spec}_{\text{adm}}$ and from the specification (given later, in definition 3) of the lower-level predicate $\text{defends}$.

---

4 Trivially, a set of assumptions $A \subseteq \text{Ab}$ attacks an assumption $\alpha \in \text{Ab}$ iff $A$ attacks $\{\alpha\}$.

5 Moreover, there is an additional, implicit argument $T$ in $\text{adm}$ and all predicates considered in these paper.
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 \( \Delta \) 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 \( \Delta \), 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 2.** A set of assumptions \( \Delta \subseteq Ab \) is admissible iff
- \( \Delta \) is closed, and
- for each closed \( A \subseteq Ab \), if \( A \) attacks \( \Delta \) then \( \Delta \) attacks \( A - \Delta \).

The proof of this theorem can be found in the appendix.

**Definition 3.** Let \((\mathcal{T}, Ab, \text{attacks})\) be an argumentation-theoretic framework. For any sets of assumptions \( \mathcal{D}, \Delta \subseteq Ab \),

\[
de\text{fends}(\mathcal{D}, \Delta) \iff \forall A \subseteq Ab \left[ \left[ \text{attacks} \Delta \land \text{closed}(A) \right] \rightarrow \mathcal{D} \text{ attacks } A - \Delta \right]
\]

where \( \text{closed}(A) \) means "\( A \) is closed". We also say that \( \mathcal{D} \) defends \( \Delta \).

This definition provides a specification for the predicate \( \text{defends} \). This specification together with the auxiliary definitions of attack and \( \text{closed} \) and with definitions of set-theoretic operations and relationships will be referred to as \( \text{Spec}_{\text{defends}} \).

The following corollary, which follows directly from theorem 2, characterises admissibility and \( \text{Spec}_{\text{adm}} \) in terms of \( \text{defends} \), and will be used to prove theorems 5 and 14 below.

**Corollary 4.**
1. \( \Delta \subseteq Ab \) is admissible iff \( \Delta \) is closed and \( \Delta \) defends \( \Delta \).
2. The specification \( \text{Spec}_{\text{adm}} \) can be expressed equivalently as

\[
\text{adm}(\Delta_0, \Delta) \iff \Delta_0 \subseteq \Delta \land \text{defends}(\Delta_0, \Delta) \land \text{closed}(\Delta).
\]

The proof procedure is given by the logic program

<table>
<thead>
<tr>
<th>\text{Progam}_{\text{adm}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{adm}(\Delta, \Delta) \iff \text{defends}(\Delta, \Delta), \text{closed}(\Delta)</td>
</tr>
<tr>
<td>\text{adm}(\Delta, \Delta') \iff \text{defends}(\mathcal{D}, \Delta), \text{closed}(\Delta \cup \mathcal{D}), \text{adm}(\Delta \cup \mathcal{D}, \Delta')</td>
</tr>
</tbody>
</table>

Note that, in the case of flat argumentation-theoretic frameworks, every set of assumptions is closed. Therefore, in this case, the conditions \( \text{closed}(\Delta) \) and \( \text{closed}(\Delta \cup \mathcal{D}) \) in \text{Progam}_{\text{adm}} \) can be omitted.

The soundness of \text{Progam}_{\text{adm}} \) is expressed by corollary 6 below, which is a direct consequence of the following theorem:

**Theorem 5.** \( \text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{Progam}_{\text{adm}} \).

**Proof:** We prove the theorem by deriving the program \text{Progam}_{\text{adm}} \) from the specification. By letting \( \Delta_0 = \Delta \) in \text{Spec}_{\text{adm}}, as formulated in corollary 4.2
\[ \text{adm}(\Delta_0, \Delta) \vdash \Delta_0 \subseteq \Delta \land \text{defends}(\Delta, \Delta) \land \text{closed}(\Delta) \]

we immediately obtain the first clause of the program.

To obtain the second clause, we let \( \Delta_0' = \Delta_0' \cup D \) in the only-if half of \( \text{Spec}_{\text{adm}} \), as formulated in corollary 4.2, and observe that \( \Delta_0' \cup D \subseteq \Delta \) implies \( \Delta_0' \subseteq \Delta \), obtaining

\[ \text{adm}(\Delta_0' \cup D, \Delta) \vdash [\Delta_0' \subseteq \Delta \land \text{defends}(\Delta_0', \Delta) \land \text{closed}(\Delta) \].

Then, by applying the if half of \( \text{Spec}_{\text{adm}} \), by transitivity of \( \vdash \), we obtain

\[ \text{adm}(\Delta_0' \cup D, \Delta) \vdash \text{adm}(\Delta_0', \Delta) \]

which implies

\[ \text{adm}(\Delta_0' \cup D, \Delta) \land \text{closed}(\Delta_0' \cup D) \land \text{defends}(D, \Delta_0') \vdash \text{adm}(\Delta_0', \Delta) \]

By renaming \( \Delta_0' \) to \( \Delta \) and \( \Delta \) to \( \Delta' \), we obtain the second clause of the program. \( \Box \)

Note that the derivation of the program \( \text{Prog}_{\text{adm}} \) from the specifications \( \text{Spec}_{\text{adm}} \) and \( \text{Spec}_{\text{defends}} \) is achieved by simple deductive steps (e.g., transitivity of \( \vdash \) and or introduction) possibly exploiting properties of the relations involved (e.g., of \( \subseteq \)).

**Corollary 6.** For all \( \Delta_0, \Delta \subseteq \text{Ab} \),

if \( \text{Prog}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \),

then \( \text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \).

Namely, if, for some given \( \Delta_0 \subseteq \text{Ab} \), the goal \( \vdash \text{adm}(\Delta_0, X) \) succeeds for \( X = \Delta \), with respect to \( \text{Prog}_{\text{adm}} \) and assuming \( \text{Spec}_{\text{defends}} \), then \( \Delta \) is an admissible superset of \( \Delta_0 \). As a consequence, the procedure \( \text{Prog}_{\text{adm}} \) is sound. The procedure \( \text{Prog}_{\text{adm}} \) is also complete in the following sense:

**Theorem 7.** For all \( \Delta_0, \Delta \subseteq \text{Ab} \),

if \( \text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \),

then \( \text{Prog}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \).

**Proof:** Assume \( \text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \). Then,

\( \text{Spec}_{\text{defends}} \models \Delta_0 \subseteq \Delta \land \text{defends}(\Delta_0, \Delta) \land \text{closed}(\Delta) \).

Then, by the first clause of \( \text{Prog}_{\text{adm}} \)

\( \text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \).

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

In the first case, \( \text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \) immediately.

In the second case, since, trivially, any defence of a set \( \Delta \) also defends any subset of \( \Delta \), i.e., for any sets of assumptions \( D, \Delta, \Delta' \subseteq \text{Ab} \)

\( \text{Spec}_{\text{defends}} \models \text{defends}(D, \Delta) \land \Delta' \subseteq \Delta \rightarrow \text{defends}(D, \Delta') \)

then

\( \text{Spec}_{\text{defends}} \models \text{defends}(\Delta, \Delta_0) \land \text{closed}(\Delta) \).

Then, \( \text{Prog}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta, \Delta_0) \land \text{defends}(\Delta, \Delta_0) \land \text{closed}(\Delta) \).

But \( \Delta = \Delta \cup \Delta_0 \). Therefore,

\( \text{Prog}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta \cup \Delta_0, \Delta) \land \text{defends}(\Delta, \Delta_0) \land \text{closed}(\Delta \cup \Delta_0) \).

But then, by the second clause of \( \text{Prog}_{\text{adm}} \),

\( \text{Prog}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta) \). \( \Box \)
Namely, if $\Delta$ is an admissible superset of a given set of assumptions $\Delta_0$, then the program $\text{Progadm}$, assuming $\text{Specdefs}$, successfully computes $X = \Delta$, given the goal $\text{adm}(\Delta_0, X)$. Note that the proof of completeness is achieved by symbolic execution of the program $\text{Progadm}$, and by appropriately choosing defences satisfying $\text{Specdefs}$.

The full proof procedure is obtained by adding to $\text{Progadm}$ a program $\text{Progdefs}$ for computing $\text{defs}$, for checking $\text{closed}$ and for computing the set-theoretic constructs, $\cup, \subseteq$, etc. This program may or may not be in the form of a logic program. If such a program is sound with respect to the specification $\text{Specdefs}$, then $\text{Progadm} \land \text{Progdefs}$ is also sound, with respect to $\text{Specadm}$ and $\text{Specdefs}$.

**Theorem 8.** Given $\text{Progdefs}$ such that, for all $\Delta, D \subseteq A_b$,
- if $\text{Progdefs} \models \text{defs}(D, \Delta)$ then $\text{Specdefs} \models \text{defs}(D, \Delta)$, and
- if $\text{Progdefs} \models \text{closed}(\Delta)$ then $\text{Specdefs} \models \text{closed}(\Delta)$,
then, for all $\Delta_0, \Delta \subseteq A_b$,
- if $\text{Progadm} \land \text{Progdefs} \models \text{adm}(\Delta_0, \Delta)$
then $\text{Specadm} \land \text{Specdefs} \models \text{adm}(\Delta_0, \Delta)$.

The proof of this and the following theorem can be found in the appendix.

Moreover, if a given program $\text{Progdefs}$ is complete with respect to the specification $\text{Specdefs}$, then $\text{Progadm} \land \text{Progdefs}$ is also complete, with respect to $\text{Specadm}$ and $\text{Specdefs}$. More precisely:

**Theorem 9.** Given $\text{Progdefs}$ such that, for all $\Delta, D \subseteq A_b$,
- if $\text{Specdefs} \models \text{defs}(D, \Delta)$ then $\text{Progdefs} \models \text{defs}(D, \Delta)$, and
- if $\text{Specdefs} \models \text{closed}(\Delta)$ then $\text{Progdefs} \models \text{closed}(\Delta)$,
then, for all $\Delta_0, \Delta \subseteq A_b$,
- if $\text{Specadm} \land \text{Specdefs} \models \text{adm}(\Delta_0, \Delta)$
then $\text{Progadm} \land \text{Progdefs} \models \text{adm}(\Delta_0, \Delta)$.

### 4 More efficient proof procedure

The proof procedure given by the program $\text{Progadm}$ performs a great deal of redundant computation. When a defence for the currently accumulated set of assumptions is generated, it is added to the accumulated set, without distinguishing between old assumptions that have already been defended and new assumptions that still have to be defended. As a consequence, defences for the old assumptions are recomputed redundantly when generating a defence for the new set. Moreover, when re-defending assumptions, new defences for such assumptions might be selected, different from the ones generated before, and these may need to be defended in turn. To avoid these redundancies, it suffices to distinguish in the currently accumulated set of assumptions, $\Delta \cup D$, between those assumptions $\Delta$ that are already "defended" by $\Delta \cup D$ itself and those assumptions $D$ that have just been added to $\Delta \cup D$ and require further defence. For this purpose, we employ a variant $\text{adm}^*(\Delta_0, D, \Delta)$ of the predicate $\text{adm}(\Delta_0, \Delta)$. 
Definition 10. Let $(T, Ab, \text{attacks})$ be an argumentation-theoretic framework. For any sets of assumptions $\Delta_0, D, \Delta \subseteq Ab$,
$$\text{adm}^e(\Delta_0, D, \Delta) \iff \Delta_0 \cup D \subseteq \Delta \land$$
$$[\text{defends}^e(\Delta_0 \cup D, \Delta_0) \land \text{closed}(\Delta_0 \cup D)] \rightarrow \Delta \text{ is admissible}].$$

We refer to this definition, together with that of admissibility, as $\text{Spec}_{\text{adm}^e}$.

The relationship between $\text{adm}$ and $\text{adm}^e$ is given by the following lemma, whose proof can be found in the appendix.

Lemma 11. For all sets of assumptions $\Delta_0$ and $\Delta$,

1. if $\text{Spec}_{\text{adm}^e} \land \text{Spec}_{\text{defends}} \models \text{adm}^e(\emptyset, \Delta_0, \Delta) \land \text{closed}(\Delta_0)$
   then $\text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta)$;
2. if $\text{Spec}_{\text{adm}} \land \text{Spec}_{\text{defends}} \models \text{adm}(\Delta_0, \Delta)$
   then $\text{Spec}_{\text{adm}^e} \land \text{Spec}_{\text{defends}} \models \text{adm}^e(\emptyset, \Delta_0, \Delta)$

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

$$\text{Prog}_{\text{adm}^e}:$$

$$\begin{align*}
\text{adm}^e(\Delta_0, D, \Delta) & \iff \text{defends}^e(D', \Delta, D), \\
& \land \text{closed}(\Delta \cup D \cup D'), \\
& \land \text{adm}^e(\Delta \cup D, D' - (\Delta \cup D), \Delta')
\end{align*}$$

where $\text{defends}^e$ is the variant of the predicate $\text{defends}$ specified as follows:

Definition 12. Let $(T, Ab, \text{attacks})$ be an argumentation-theoretic framework. For any sets of assumptions $\Delta, D, \Delta' \subseteq Ab$,
$$\text{defends}^e(\Delta', \Delta, D) \iff$$
$$\forall A \subseteq Ab[[A \text{ attacks } D \land \text{closed}(A)] \rightarrow \Delta' \cup \Delta \cup D \text{ attacks } A - (\Delta \cup D)].$$

We will refer to this specification together with the definitions of attack, closed and the set-theoretic constructs as $\text{Spec}_{\text{defends}^e}$.

The following corollary, which follows directly from theorem 2, characterises admissibility and $\text{Spec}_{\text{adm}^e}$ in terms of $\text{defends}^e$, and will be used to prove theorem 14.

Corollary 13.

1. $\Delta$ is admissible iff $\text{defends}^e(\Delta, \emptyset, \Delta)$ and $\text{closed}(\Delta)$.
2. $\text{Spec}_{\text{adm}^e}$ is equivalent to
   $$\text{adm}^e(\Delta_0, D, \Delta) \iff \Delta_0 \cup D \subseteq \Delta \land$$
   $$[\text{defends}^e(D, \Delta_0, \Delta_0) \land \text{closed}(\Delta_0 \cup D)] \rightarrow \Delta \text{ is admissible}].$$

The soundness of $\text{Prog}_{\text{adm}^e}$ is given by corollary 16 below, which follows directly from lemma 11 and from the following theorem:

Theorem 14. $\text{Spec}_{\text{adm}^e} \land \text{Spec}_{\text{defends}^e} \models \text{Prog}_{\text{adm}^e}$. 
**Proof**: We prove the theorem by deriving the program $Prog_{adm^e}$ from the specification. By letting $D = \emptyset$ and $\Delta = \Delta_0$ in $Spec_{adm^e}$

$$adm^e(\Delta_0, D, \Delta) \iff \Delta_0 \cup D \subseteq \Delta \land \left[ \text{defends}(\Delta_0 \cup D, \Delta_0) \land \text{closed}(\Delta_0 \cup D) \rightarrow \Delta \text{ is admissible} \right]$$

we obtain

$$adm^e(\Delta, \emptyset, \Delta) \iff \Delta \subseteq \Delta \land \left[ \text{defends}(\Delta, \Delta) \land \text{closed}(\Delta) \rightarrow \Delta \text{ is admissible} \right]$$

equivalent to the first clause of $Prog_{adm^e}$ because of corollary 4.1.

To obtain the second clause, first replace the predicate $adm^e$ in the second clause of the program by the equivalent specification in terms of $\text{defends}^e$ given by corollary 13.2, obtaining

$$[\Delta \cup D \subseteq \Delta' \land \left[ \text{defends}^e(D, \Delta, \Delta) \land \text{closed}(\Delta \cup D) \rightarrow \Delta' \text{ is admissible} \right] \land \left[ \text{defends}^e(D', \Delta, \Delta) \land \text{closed}(\Delta \cup D \cup D') \land \Delta \cup D \cup D' \subseteq \Delta' \land \left[ \text{defends}^e(D' - (\Delta \cup D), \Delta \cup D, \Delta \cup D) \land \text{closed}(\Delta \cup D \cup D') \rightarrow \Delta' \text{ is admissible} \right] \right].$$

This can be rewritten in the logically equivalent form

$$[\Delta \cup D \subseteq \Delta' \land \Delta' \text{ is admissible} \land \left[ \text{defends}^e(D, \Delta, \Delta) \land \text{closed}(\Delta \cup D) \land \text{defends}^e(D', \Delta, \Delta) \land \text{closed}(\Delta \cup D \cup D') \land \Delta \cup D \cup D' \subseteq \Delta' \land \left[ \text{defends}^e(D' - (\Delta \cup D), \Delta \cup D, \Delta \cup D) \land \text{closed}(\Delta \cup D \cup D') \rightarrow \Delta' \text{ is admissible} \right] \right].$$

which follows immediately from the fact that

$$\Delta \cup D \subseteq \Delta' \rightarrow \Delta \cup D \cup D' \subseteq \Delta'$$

and from the following lemma, whose proof can be found in the appendix. \(\square\)

**Lemma 15.**

$$\text{defends}^e(D, \Delta, \Delta) \land \text{defends}^e(D', \Delta, \Delta) \rightarrow \text{defends}^e(D' - (\Delta \cup D), \Delta \cup D, \Delta \cup D).$$

As for $Prog_{adm}$ given in section 3, the derivation of $Prog_{adm^e}$ from the specifications $Spec_{adm^e}$ and $Spec_{defends^e}$ consists of simple deductive steps (here presented in a backward fashion), possibly exploiting properties of the relations involved (e.g., $\subseteq$ and $defends$, as expressed by lemma 15).

**Corollary 16.** For all $\Delta_0, \Delta \subseteq Ab$,

if $Prog_{adm^e} \land Spec_{defends^e} \models adm^e(\emptyset, \Delta_0, \Delta) \land \text{closed}(\Delta_0)$,

then $Spec_{adm} \land Spec_{defends} \models adm(\Delta_0, \Delta)$.

Namely, if, for some given set of assumptions $\Delta_0$, the goal $adm^e(\emptyset, \Delta_0, X)$ succeeds for $X = \Delta_0$ with respect to $Prog_{adm^e}$ and assuming $Spec_{defends^e}$, then $\Delta$ is an admissible superset of $\Delta_0$. As a consequence, the proof procedure $Prog_{adm^e}$ is sound. $Prog_{adm^e}$ is also complete in the following sense:

**Theorem 17.** For all $\Delta_0, \Delta \subseteq Ab$,

if $Spec_{adm} \land Spec_{defends} \models adm(\Delta_0, \Delta)$,

then $Prog_{adm^e} \land Spec_{defends} \models adm^e(\emptyset, \Delta_0, \Delta)$.

**Proof**: Assume $Spec_{adm} \land Spec_{defends} \models adm(\Delta_0, \Delta)$. Then $Spec_{defends} \models \Delta_0 \subseteq \Delta \land \text{defends}(\Delta, \Delta) \land \text{closed}(\Delta)$, and
\[ Spec_{defends} \models \Delta_0 \subseteq \Delta \land defends^e(\Delta, \emptyset, \Delta) \land \text{closed}(\Delta). \]
Moreover, it is easy to see that
\[ Spec_{defends} \models [\Delta_0 \subseteq \Delta \land \text{defends}^e(\Delta, \emptyset, \Delta)] \rightarrow \text{defends}^e(\Delta, \emptyset, \Delta_0). \]
Therefore, (i) \[ Spec_{defends} \models \text{defends}^e(\Delta, \emptyset, \Delta_0). \]
Similarly,
\[ Spec_{defends} \models [\Delta_0 \subseteq \Delta \land \text{defends}^e(\Delta, \emptyset, \Delta) \rightarrow \text{defends}^e(\Delta, \Delta_0, \Delta - \Delta_0)]. \]
Therefore, (ii) \[ Spec_{defends} \models \text{defends}^e(\Delta, \Delta_0, \Delta - \Delta_0). \]

To show \( Prog_{adm} \land Spec_{defends} \models adm^e(\emptyset, \Delta_0, \Delta) \), use the following instance of the second clause of the program:
\[ adm^e(\emptyset, \Delta_0, \Delta), \text{defends}^e(D', \emptyset, \Delta_0), \text{closed}(\Delta_0 \cup D'), \text{adm}^e(\Delta_0, D' - \Delta_0, \Delta) \]

Let \( D' = \Delta \). Then, the first condition is provable from \( Spec_{defends} \) by (i), and the second condition is provable from \( Spec_{defends} \) since \( \Delta_0 \subseteq \Delta \) and \( \Delta \) is closed.

To prove the third condition, use the following instance of the second clause of the program:
\[ adm^e(\emptyset, \Delta_0, \Delta), \text{defends}^e(D', \Delta_0, \Delta - \Delta_0), \text{closed}(\Delta \cup D'), \text{adm}^e(\Delta, D' - \Delta, \Delta) \]

Let \( D' = \Delta \). Then, the first condition is provable from \( Spec_{defends} \) by (ii), and the second condition is provable from \( Spec_{defends} \) since \( \Delta \) is closed. Moreover, the third condition is provable by the first clause of \( Prog_{adm} \). Therefore, \( Prog_{adm} \land Spec_{defends} \models adm^e(\emptyset, \Delta_0, \Delta) \).

As in section 3, the proof of completeness is achieved by symbolic execution of the procedure \( Prog_{adm} \), with two calls to the specification \( Spec_{defends} \).

The full proof procedure is obtained by adding to \( Prog_{adm} \) a program \( Prog_{defends} \) for computing \( \text{defends}^e \), for checking \( \text{closed} \) and for computing the set-theoretic constructs, \( \cup, \subseteq \), etc. In [6] we give the top-most level of a (generalised) logic program for computing \( \text{defends}^e \), which provides a sound but incomplete proof procedure.

5 Conclusion

We have used logic program development technology to define two proof procedures for the admissibility semantics for the abstract, argumentation-theoretic framework presented in [2].

Rather than develop new methods, we have employed existing techniques of deductive synthesis [3] to derive two small but non-trivial programs and to prove them sound, and techniques of symbolic execution to prove them complete.

The second program is an improvement of the first, obtained by adding an argument, \( \mathcal{D} \), to the predicate \( adm \), thus obtaining a predicate \( adm^e \). The new argument plays the role of an accumulator, and gives rise to a more efficient proof procedure \( Prog_{adm} \). This is re-synthesised from scratch from a new specification for \( adm^e \). As a subject for future work, it would be interesting to explore
the possibility of deriving Prolog from the initial, inefficient proof procedure, Prolog*; using standard techniques of logic program transformation (fold, unfold and so on, see [17]) and/or techniques borrowed from functional programming (e.g., see [16]).

Acknowledgements

This research was supported by the EEC activity KIT011-LPKRR. The third author was partially supported by EEC HCM Project no CHRX-CT93-00414, "Logic Program Synthesis and Transformation". The authors are grateful to the LOPSTR96 participants for helpful comments and suggestions.

References

5. P. M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming, Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambery, France (1993), Morgan Kaufmann (R. Bajcsy, ed.) 852–857

**Appendix**

**Proof of theorem 2**

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

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

**Proof of theorem 8**

Assume $Prog_{\text{adm}} \land Prog_{\text{defends}} \models adm(\Delta_0, \Delta)$. Then, since $Prog_{\text{defends}}$ is sound with respect to $Spec_{\text{defends}}$,

$Prog_{\text{adm}} \land Spec_{\text{defends}} \models adm(\Delta_0, \Delta)$.

Then, directly from corollary 6,

$Spec_{\text{adm}} \land Spec_{\text{defends}} \models adm(\Delta_0, \Delta)$.

**Proof of theorem 9**

Assume $Spec_{\text{adm}} \land Spec_{\text{defends}} \models adm(\Delta_0, \Delta)$. Then, directly from theorem 7,

$Prog_{\text{adm}} \land Spec_{\text{defends}} \models adm(\Delta_0, \Delta)$. By completeness of $Prog_{\text{defends}}$,

$Prog_{\text{adm}} \land Prog_{\text{defends}} \models adm(\Delta_0, \Delta)$.

**Proof of lemma 11**

1. First, note that, $adm(\emptyset, \Delta_0, \Delta) \land closed(\Delta_0)$ implies

$\Delta_0 \subseteq \Delta \land ([defends(\Delta_0, \emptyset) \land closed(\Delta_0)] \rightarrow \Delta \text{ is admissible}) \land closed(\Delta_0)$.

But $Spec_{\text{defends}}$ trivially implies $defends(\Delta_0, \emptyset)$. Therefore
\[ \Delta_0 \subseteq \Delta \land [\text{closed}(\Delta_0) \rightarrow \Delta \text{ is admissible}] \land \text{closed}(\Delta_0) \]
which, in \( Spec_{adm} \), implies \( adm(\Delta_0, \Delta) \).

2. \( adm(\Delta_0, \Delta) \) implies \( \Delta_0 \subseteq \Delta \land \Delta \text{ is admissible} \).

This trivially implies
\[ \Delta_0 \subseteq \Delta \land [[\text{closed}(\Delta_0) \land \text{defends}(\Delta, \emptyset)] \rightarrow \Delta \text{ is admissible}] \]
which, in \( Spec_{adm} \), implies \( adm^e(\emptyset, \Delta_0, \Delta) \).

**Proof of Lemma 15**: Assume

(i) \( \text{defends}^e(D, \Delta, \Delta) \), and
(ii) \( \text{defends}^e(D', \Delta, D) \).

Assume \( A \subseteq Ab \) attacks \( \Delta \cup D \). We need to show that \( D' \cup \Delta \cup D \) attacks \( A \rightarrow (\Delta \cup D) \).

- If \( A \) attacks \( D \) then, by (ii), \( D' \cup \Delta \) attacks \( A \rightarrow (\Delta \cup D) \), and thus \( D' \cup \Delta \cup D \) attacks \( A \rightarrow (\Delta \cup D) \).
- If \( A \) attacks \( \Delta \) then, by (i), \( D' \cup \Delta \) attacks \( A \rightarrow \Delta \). It suffices to show that \( D \cup \Delta \) does not attack \( D \). Suppose, on the contrary, that \( D \cup \Delta \) attacks \( D \). Then, by (ii), \( D' \cup \Delta \) attacks \( (D \cup \Delta) \rightarrow (D \cup \Delta) = \emptyset \). But this is not possible, because, by definition of attack, there are no attacks against \( \emptyset \).