FernUniversität in Hagen
Fakultät für Mathematik und Informatik

Artificial Intelligence Group

Implementing Serialization Sequences of Abstract Argumentation Frameworks using Answer Set Programming

Bachelorarbeit

zur Erlangung des Grades einer Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Ulrich Karkmann

Erstgutachter: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Betreuer: Lars Bengel
Artificial Intelligence Group

Erklärung

Ich erkläre, dass ich die Bachelorarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Bachelorarbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Bachelorarbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Der Veröffentlichung dieser Arbeit auf der Webseite des Lehrgebiets Künstliche Intelligenz und damit dem freien Zugang zu dieser Arbeit stimme ich ausdrücklich zu.

Für diese Arbeit erstellte Software wurde quelloffen verfügbar gemacht, ein entsprechender Link zu den Quellen ist in dieser Arbeit enthalten. Gleiches gilt für angefallene Forschungsdaten.

Groß Lindow, den 28.03.2025

....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Eine einfache und leistungsfähige Methode zur Darstellung und Analyse menschlicher Argumentation bieten abstrakte Argumentationsgraphen. Die Argumente bilden die Knoten des Graphen, zwischen denen gerichtete Kanten die Widerlegung eines Arguments durch ein anderes repräsentieren. Diejenigen Mengen von Argumenten, die sich innerhalb eines Graphen durchsetzen, werden als Extensionen bezeichnet, wobei es verschiedene Arten zur Bestimmung von Extensionen gibt. Die Argumente bestimmter Extensionen lassen sich in eine Folge von *initialen Mengen*, die jeweils die Lösung eines lokalen Konflikts repräsentieren, serialisieren. Damit werden die initialen Mengen so in eine Reihenfolge gebracht, dass sie der menschlichen Art entspricht, sequentiell zu argumentieren. Die Serialisierung kann die Überzeugungskraft einer Argumentation verbessern und zum Vergleich verschiedener Argumentationen herangezogen werden. Die Berechnung solcher Extensionen und ihrer Serialisierungen ist ein nichtdeterministisches kombinatorisches Problem. Ein Programmierparadigma, das für die Lösung solcher Probleme konzipiert wurde, ist in Gestalt des Answer Set Programming (ASP) realisiert. In der vorliegenden Arbeit werden ASP-Kodierungen zur Berechnung verschiedener Arten von initialen Mengen und von Serialisierungssequenzen für serialisierbare Semantiken vorgestellt und diskutiert. Diese ASP-Kodierungen werden anhand verschiedener Beispiele von abstrakten Argumentationsgraphen hinsichtlich ihrer Korrektheit und Laufzeit miteinander verglichen. Im Ergebnis stimmen die vom ASP-Solver ausgegebenen Serialisierungssequenzen mit denen der Java-Implementierung in jeder Semantik überein. Der wesentliche Faktor in Bezug auf die Laufzeit ist die Anzahl der Argumente des jeweiligen Argumentationsgraphen für jede Semantik. Der ASP-Solver ist bei fünf Semantiken schneller und in den verbleibenden zwei Semantiken langsamer als die Java-Implementierung. Mit Blick auf die lösbare Größe eines Argumentationsgraphen ist die Laufzeit der maßgebliche Faktor für die Java -Implementierung, wohingegen der ASP-Solver durch die Größe des zur Verfügung stehenden Speichers begrenzt sein kann.

Abstract

Abstract argumentation frameworks offer a simple and powerful method for representing and analyzing human argumentation. The arguments form the nodes of the framework, between which directed edges represent the refutation of one argument by another. The sets of arguments that prevail within a framework are called extensions, and there are different ways of determining extensions. The argumentation of certain extensions can be serialized into a sequence of *initial sets*, each representing a solution to a local conflict. The initial sets are arranged in an order that corresponds to the human way of arguing sequentially. Serialization can improve the persuasiveness of an argumentation and can be used to compare different argumentations. The calculation of such extensions and their serializations is a nondeterministic combinatorial problem. A programming paradigm that was designed to solve such problems is realized in the form of Answer Set Programming (ASP). In this thesis, ASP encodings for computing different types of initial sets and serialization sequences for semantics that can be serialized are presented and discussed. These ASP encodings are compared with a Java implementation for the same task in terms of correctness and runtimes using various example argumentation frameworks. As a result, both solvers show the same serialization sequences for the argumentation frameworks tested in each of the semantics. The main parameter affecting runtime for all semantics and both solvers is the argument count of the argumentation framework to be solved. The ASP solver is faster than the Java solver in five semantics, while it is slower in the remaining two semantics. With respect to the solvable size of an argumentation framework, for the Java solver the runtime is the main limiting factor, whereas the ASP solver may be primarily limited by the available memory.

Contents

1. Introduction

An important way to describe and follow individual conclusions is to argue, i.e. to present arguments for or arguments against a certain position. In areas like knowledge representation, reasoning and explainable artificial intelligence it is crucial to make decisions coherent and transparent. Especially in domains where decisions or conclusions must be revisable, e.g. if they are relevant to security or ethics, traceability of arguments become essential. Various models have been proposed to represent human reasoning in computer science [4], where *abstract argumentation frameworks* have proven to be a simple and powerful representation for explaining the acceptance of arguments. Abstract argumentation frameworks are directed graphs with single arguments as nodes, where the edge between two arguments represents the invalidation of one argument by the other. Sets of arguments representing a (coherent) point of view are called extensions, that can identify the outcome(s) of a discussion represented by an argumentation framework. The abstract way to compute the extension of a given argumentation framework is defined by the corresponding semantics, with a variety of different semantics available.

Unfortunately, the explanatory power of those semantics does not satisfy the human need to consider arguments in a particular sequential order. To compensate for this disadvantage, the concept of *serialization* was proposed [24, 27], in which the desired extension is constructed step by step starting from the argumentation framework with a subset of arguments, so-called *initial sets*. Initial sets are minimal (with respect to set inclusion) acceptable sets of arguments, that represent a single solved issue within an argumentation framework [24]. Each initial set can be considered as a single step within a sequential argumentation. They are selected iteratively from the original framework and its induced reducts. Merging all initial sets of a particular serialization sequence leads to the desired extension. This allows to 'follow' the argumentation with respect to the corresponding extension and is also suitable for comparing different argumentation frameworks [5].

The computation of such serialization sequences is a nondeterministic and complex combinatorial task, for which Answer Set Programming (ASP) is likely to be suitable. ASP is based on a declarative programming paradigm without any control structures. While traditional imperative programming is mainly based on control structures such as conditional loops, variable assignments and I/O statements, a declarative program does not provide the algorithmic way to find a solution, but rather defines what counts as a solution to the problem. In logical programming such as ASP, this is done through the process of automated reasoning, where the programming system searches for solutions in a knowledge base that satisfy the given conditions [21]. In particular, ASP can be well suited to solve complex combinatorial problems if the human description of the problem comes close to the facts, rules and constraints used for programming[1]. In contrast to imperative programming languages, the program in such cases can be relatively short and is easier to

[1]Therefore some authors refer to 'modelling' instead of 'programming' [18].

understand.

The aim of this bachelor thesis is to combine the described task and the corresponding programming paradigm to provide an implementation for the computation of serialization sequences of abstract argumentation frameworks using ASP. The ASP encoding is expected to come closer to the logical description of the structures of abstract argumentation frameworks and the considered semantics than the encoding of an imperative programming language. To what extend these expectations can actually be fulfilled will be shown in this thesis.

First, the theoretical background of abstract argumentation frameworks and serializable semantics is described in Section 2. The properties and conditions required to determine the serialization sequences of each suitable semantics are shown in preparation for their implementation. Some properties and limitations of ASP are described to understand the encodings, which are described in detail in Section 3 for initial sets and in Section 4 for serialization sequences. In Section 5 the ASP encodings are evaluated against an implementation in Java with respect to correctness and runtime. Some suggestions for improving the results of this thesis through future work are presented in Section 6. Finally, the result are summarized and discussed in Section 7. The complete encodings are detailed in the Appendix and are also available online a https://github.com/ukarkmann /ASP-encoding-for-serialization-sequences.

2. Background

This section first describes the basic properties of abstract argumentation frameworks, some of their extension-based semantics, and the concept of serialization sequences. This is the basis for the ASP code to be implemented and at the same time describes the objective of this thesis. We then briefly describe some of the features and limitations of ASP to help readers who are not familiar with ASP understand the code.

2.1. Abstract Argumentation Frameworks

Abstract argumentation frameworks [11] consist of a finite set of arguments and a single attack relation between two arguments, thus spanning a directed graph with the arguments as nodes and the relation as directed edges. There is no inner structure of the arguments to be considered.

Definition 1 *An abstract argumentation framework is a pair $AF = (A, \succ)$ with A as the set of arguments and $\succ$ the binary attack relation with $\succ \subseteq A \times A$.*

We have $a \succ b$ when a attacking b with $a, b \in A$ (and $a \not\succ b$ when a not attacking b, respectively). The symbol $\succ$ can also be used to illustrate an attack between two sets of arguments $S_1 \succ S_2$ if $a \in S_1, b \in S_2$ and $a \succ b$. Having defined the syntax for abstract argumentation frameworks, the next step is to select an appropriate semantics. It turns out to be a rather complex problem with several reasonable solutions possible [10, 8, 19, 7].

Among such proposed semantics, only the so-called extension-based semantics will be considered here, which define specific subsets of arguments within the argumentation framework (*extensions*), that are accepted within the argumentation framework and considered to be 'meaningful' from a human perspective [1]. A single extension can be interpreted as a particular (coherent) position taken in a discussion. Depending on the type of extension there can be more than one such set for a single argumentation framework.

2.1.1. Extension-based Semantics

The basic idea of extension-based semantics is that an argument a rules out an argument b in case of $a \succ b$ (including self-attacking $a \succ a$ and pairwise attacking $a \succ b \wedge b \succ a$). A large number of sets and extensions have been defined, of which only some of the most important are described here and whose definitions have been adopted from [4, 11]. First of all, any extension must reasonably be *conflict-free*, i.e. there must be no attacks (= relations) within an extension, which leads to the definition of conflict-free sets:

Definition 2 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then S is a* conflict-free *set iff for all $a, b \in S$ it holds that $a \not\succ b$.*

For further discussions it is useful to define the set of arguments S^+_{AF} which are attacked by at least one argument of S and the set of arguments S^-_{AF} which are attacking at least one argument of S:

Definition 3 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then we define*

$$S^+_{AF} := \{b \in A \mid \exists a \in S : a \succ b\}$$
$$S^-_{AF} := \{b \in A \mid \exists a \in S : b \succ a\}$$

For each conflict-free set S it follows $S \cap S^+_{AF} = \emptyset$ and $S \cap S^-_{AF} = \emptyset$, as otherwise S would not be conflict-free. Being conflict-free is a necessary but not a sufficient condition for an extension, since arguments in a conflict-free set S can be attacked from external arguments $b \in A \setminus S$. This leads to the concept of 'defending' an argument by (other) arguments. As there is only the attack relation, the defense can be realized by attacking all attackers. An argument a is defended by a set S iff all arguments attacking a are attacked by arguments from S. Arguments that are not attacked at all within an argumentation framework are at least defended by the empty set. The set of arguments defended by a set S is given by the *characteristic function* $\Delta_{AF}(S)$:

Definition 4 *Let $AF = (A, \succ)$ be an argumentation framework. The* characteristic function *Δ_{AF} of AF is the function $\Delta_{AF} : 2^A \to 2^A$ defined as $\Delta_{AF}(S) := \{a \in A \mid a$ is defended by $S\}$.*

Both conflict-freeness and defence lead to the definition of an admissible set.

Definition 5 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then S is an* admissible *set iff S is conflict-free and a is defended by S for all $a \in S$.*

Equivalent to this definition, a set S is admissible iff it is conflict-free and $S^- \subseteq S^+$. Therefore the empty set is always admissible and $S \subseteq \Delta_{AF}(S)$. Furthermore, an admissible set remains admissible if a defended argument is added; admissibility also remains, if two admissible sets are merged and their union is conflict-free [4].

Admissibility is the minimum property of any extension-based semantics, i.e. that all other semantics described below require admissibility. Since A is finite and the inclusion of defended arguments does not alter admissibility, collecting all defended arguments of an admissible set must come to an end. This motivates the definition of *complete* semantics:

Definition 6 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then S is a* complete *extension iff S is admissible and contains each $a \in A$, that is defended by S.*

From this definition it follows for a complete extension S that $S = \Delta_{AF}(S)$. There can be complete extensions that are proper subsets of another complete extension, or conversely, that are proper supersets of other complete extensions, which leads to the definition of *preferred* semantics:

Definition 7 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then S is a* preferred *extension iff S is complete and there exists no complete $S' \subseteq A$ with $S \subsetneq S'$.*

As the empty set is always admissible, one can start from the empty set, add all defended arguments to obtain a complete extension, and take the 'maximal' complete extension (with respect to set inclusion) as the preferred extension. Therefore, each argumentation framework AF must have at least one preferred extension. From a human perspective, each preferred extension represents a set of arguments that cannot be extended by other arguments. In particular, admissibility is lost when two different preferred extensions are merged. Finally, if we want to obtain an even stronger extension, we can additionally require attacking every external argument, which leads to the definition of a *stable* semantics:

Definition 8 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then S is a* stable *extension iff for each $a \in A \setminus S$ there exists $b \in S$ with $b \succ a$.*

This definition is equivalent to $S_{st} \cup S_{st}^+ = A$ with S_{st} being a stable extension. It should be noted that a stable extension does not necessarily exist for an argumentation framework. Furthermore, we can define the sets of the extensions described for a specific argumentation framework AF.

Definition 9 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then*

$$\begin{aligned} cf(AF) &:= \{S \mid S \textit{ is conflict-free in AF}\} \\ ad(AF) &:= \{S \mid S \textit{ is admissible in AF}\} \\ co(AF) &:= \{S \mid S \textit{ is complete in AF}\} \\ pr(AF) &:= \{S \mid S \textit{ is preferred in AF}\} \\ st(AF) &:= \{S \mid S \textit{ is stable in AF}\} \end{aligned}$$

These sets can be ordered with respect to set inclusion:

Proposition 1 *Let $AF = (A, \succ)$ be an argumentation framework, then the following holds*

$$cf(AF) \supseteq ad(AF) \supseteq co(AF) \supseteq pr(AF) \supseteq st(AF).$$

While a preferred extension represents the maximal set (with respect to set inclusion) of arguments of a particular viewpoint, the *grounded extension S_{gr}* represents a kind of minimal compromise, i.e. the set of arguments accepted by all different (complete) extensions.

Definition 10 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then S is the* grounded *extension iff S is complete and there exists no complete $S^‘ \subseteq A$ with $S^‘ \subsetneq S$.*

S_{gr} is defined as such a 'minimal' complete extension that has no other complete extension as a subset. Every argumentation framework has exactly one grounded extension (which could be the empty set if no argument is generally accepted). This is equivalent to the intersection of all complete extensions

$$S_{gr} = S_{co_1} \cap S_{co_2} ... \cap S_{co_n}$$

and also corresponds to the so-called *minimal fixpoint* of the characteristic function, where the characteristic function is repeatedly applied to its result, starting with the empty set:

$$S_{gr} = \bigcup_{i=1}^{\infty} \Delta^i_{AF}(\emptyset)$$

Finally, we should mention the *strongly admissible* semantics (sa) [2, 9], since we will also refer to this semantics later:

Definition 11 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq ad(AF)$ an admissible set, then S is a* strongly admissible extension *iff $S = \emptyset$ or each $a \in S$ is defended by some strongly admissible $S^‘ \subseteq S \setminus \{a\}$.*

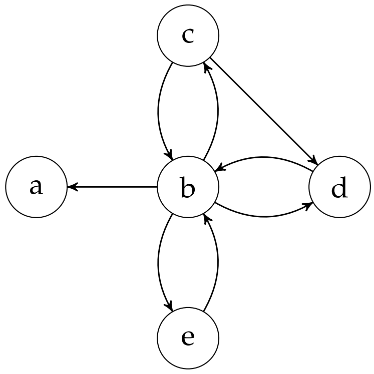


Figure 1: Example AF_1 of an abstract argumentation framework.

Example 1 *Figure 1 shows an example of an abstract argumentation framework with* $AF := (A, \succ)$, $A := \{a, b, c, d, e\}$ *and* $\succ := \{(b, a), (b, c), (b, d), (b, e), (c, b), (c, d), (d, b), (e, b)\}$. *The corresponding extensions are*

$$\begin{aligned}
ad(AF) &= \{\emptyset, \{b\}, \{c\}, \{e\}, \{a, e\}, \{a, c\}, \{c, e\}, \{a, c, e\}\} \\
co(AF) &= \{\emptyset, \{b\}, \{a, c, e\}\} \\
pr(AF) &= \{\{b\}, \{a, c, e\}\} \\
st(AF) &= \{\{b\}, \{a, c, e\}\} \\
gr(AF) &= \{\emptyset\} \\
sa(AF) &= \{\emptyset\}
\end{aligned}$$

2.1.2. Serialization of Argumentation Semantics

The concept of serialization was motivated by the observation, that each admissible set of an argumentation framework can be constructed from minimal acceptable sets of arguments in a step by step process, starting from the original argumentation framework. Such sets are non-empty minimal admissible sets and called *initial sets*, which each represents a single solved issue within an argumentation framework. The construction starts with the choice of a first initial set S_1, which is then 'substracted' from the argumentation framework resulting in a so-called *S-reduct*. All arguments from S_1 and from S_1^+ are removed from the original argumentation framework to yield the first S-reduct. Then the next initial set S_2 is chosen from the first S-reduct and the process of recursively selecting an initial set from the preceding S-reduct creates a finite sequence of initial sets S_1, S_2.... The union of the sets of a sequence is always an admissible set.

As mentioned, an initial set is defined as an admissible set that is non-empty and minimal with respect to set inclusion [27]:

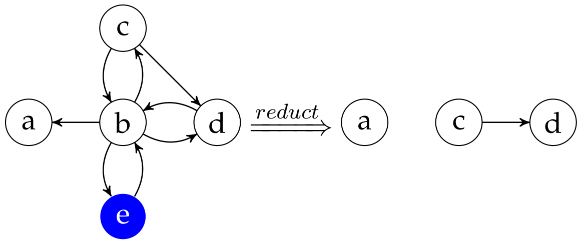


Figure 2: Construction of S-reduct AF_1^S with $S = \{e\}$
.

Definition 12 *Let $AF = (A, \succ)$ be an argumentation framework and $S \in ad(AF)$, then S is an* initial *set iff $S \neq \emptyset$ and there exists no $S' \in ad(AF)$ with $S' \neq \emptyset$ and $S' \subset S$.*

Example 2 *For the argumentation framework shown in Figure 1 the initial sets are $\{b\}$, $\{c\}$ and $\{e\}$, since these sets are those elements from $ad(AF)$, that are non-empty and minimal with respect to set inclusion (see Example 1).*

The S-reduct mentioned above is denoted AF^{S_i} and is derived from the argumentation framework AF and a subset S_i of its arguments. We make use of a projection of AF onto $X \subseteq A$, defined as the argumentation framework, which only contains the arguments from X and the relevant relations:

Definition 13 *Let $AF = (A, \succ)$ be an argumentation framework and $X \subseteq A$, then the* projection *of AF onto X is defined as $AF\,|_X := (X, \succ \cap (X \times X))$*

Now the S-reduct AF^S is defined as the projection of AF onto $A \setminus (S \cup S^+)$ such that the reduct does not contain the arguments from S and the arguments attacked S^+ attacked by S [3]:

Definition 14 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, then the* S-reduct *is defined as $AF^S := AF\,|_{A\setminus(S\cup S^+)}$*

Example 3 *Figure 2 shows the construction of the S-reduct AF_1^S with $S = \{e\}$. Since e attacks b, these two arguments (and all corresponding relations) are deleted from AF_1. The arguments a, c and d remain in the S-reduct AF_1^S with the only attack $c \succ d$ left.*

The observation, that each admissible set can be obtained by recursively selecting an initial set from the last S-reduct, can be derived from the following theorem [24]:

Theorem 1 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$. S is admissible iff either*

$S = \emptyset$ *or*

$S = S_1 \cup S_2$, *whereas S_1 is an initial set in AF and S_2 is admissible in AF^{S_1}.*

A similar observation was made for the grounded semantics, since the grounded extension of an argumentation framework can be constructed by recursively selecting all non-attacked arguments from the preceding reduct [27]. This motivated the construction of other extension-based semantics via a recursive selection process, which was eventually called serialization sequence. A serialization sequence of an abstract argumentation framework with respect to a particular semantics is a sequence of initial sets whose union is equal to the corresponding extension. In other words, a serialization sequence is an ordered decomposition of an extension by initial sets, computed from the argumentation framework itself. The selection process terminates, if a specific *termination condition* is met. Depending on the semantics to be serialized, different types of initial sets are used and different termination conditions apply. Since more than one initial set can be selected at each step, the process is non-deterministic.

The set of all initial sets of AF is denoted as $is(AF)$. We will need three classes of initial sets for serialization, $is^{\not\leftarrow}$ (*unattacked*), $is^{\not\leftrightarrow}$ (*unchallenged*) and $is^{\leftrightarrow}$ (*challenged*). The unattacked initials sets contain only arguments, which are not attacked at all. The unchallenged initial sets contain only arguments that are attacked by non-initial sets and the challenged initials sets are attacked by another initial set.

Definition 15 *Let $AF = (A, \succ)$ be an argumentation framework and S an initial set then*

$$is^{\not\leftarrow}(AF) := \{S \mid S^- = \emptyset\}$$
$$is^{\not\leftrightarrow}(AF) := \{S \mid S^- \neq \emptyset, \nexists S' \in is(AF) \text{ with } S' \succ S\}$$
$$is^{\leftrightarrow}(AF) := \{S \mid \exists S' \in is(AF) \text{ with } S' \succ S\}$$

Example 4 *For the argumentation framework shown in Figure 3 there are five initial sets: $\{a\}$, $\{b\}$, $\{d\}$, $\{f\}$ and $\{g\}$. The argument g is not attacked at all and therefore belongs to $is^{\not\leftarrow}(AF_2)$, the arguments d and f belong to $is^{\not\leftrightarrow}(AF_2)$ since they are only attacked by $\{e\}$ (which is not an initial set due to the undefeated attack from c). $\{a\}$ and $\{b\}$ are admissible and attack each other, so that they belong to $is^{\leftrightarrow}(AF_2)$.*

An important property of unattacked initial sets for their computation is that they always contain exactly one argument:

Proposition 2 *It holds that, if $S \in is^{\not\leftarrow}(AF)$ then $|S| = 1$.*

Each initial set is non-empty and therefore must contain at least one argument. Sets with more than one unattacked argument cannot be minimal, since such sets can always be decomposed into sets containing single unattacked arguments.

Having defined initial sets and the S-reduct, a serialization sequence for admissible sets can be obtained by repeatedly selecting an initial set from the last S-reduct starting with the original argumentation framework:

$$(AF, \emptyset) \xrightarrow{S_1 \in is(AF)} (AF^{S_1}, S_1) \xrightarrow{S_2 \in is(AF^{S_1})} (AF^{S_1 \cup S_2}, S_1 \cup S_2)...$$

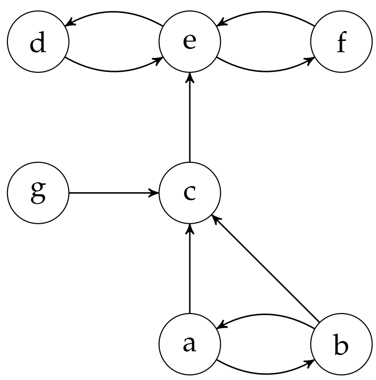


Figure 3: Example AF_2 of an abstract argumentation framework.

Definition 16 *Let* $AF = (A, \succ)$ *be an argumentation framework. A sequence* $\mathscr{S} = (S_1, .., S_n)$ *is a* serialization sequence *iff* $S_1 \in is(AF)$ *and* $S_i \in is(AF^{S_1 \cup ... \cup S_{i-1}})$ *for all* $i = 2, .., n$. *The set* $\widehat{\mathscr{S}} = S_1 \cup ... \cup S_n$ *is called the extension induced by* $\mathscr{S}$.

The above definition describes the serialization sequence for admissible sets. If we reduce the selectable initial sets to unattacked initial sets and additionally require that the selection continues until no more unattacked initial set is left in the last S-reduct, the resulting serialization sequence represents the grounded extension of the argumentation framework. To put this more generally, we can chose the type of initial set to select and define the condition under which the selection ends. Formally, the selection is performed by a selection function called α and the process terminates if the termination function called β becomes 1.

Definition 17 *Let* $\mathscr{U}$ *be the universal set of all arguments. The* selection function α *is defined as* $\alpha : 2^{2^{\mathscr{U}}} \times 2^{2^{\mathscr{U}}} \times 2^{2^{\mathscr{U}}} \rightarrow 2^{2^{\mathscr{U}}}$ *with* $\alpha(X, Y, Z) \subseteq X \cup Y \cup Z$ *for all* $X, Y, Z \subseteq \mathscr{U}$.

The different types of initial sets (see Definition 15) are assigned to the three parameters of the selection function, such that it selects subsets of initial sets for the construction of the serialization sequence. Therefore $\alpha(X, Y, Z)$ has the form $\alpha(is^{\not\leftarrow}(AF), is^{\not\leftrightarrow}(AF), is^{\leftrightarrow}(AF))$.

The termination function β can take 0 or 1 as value with 1 indicating the end of the selection process:

Definition 18 *The* termination function β *is defined as* $\beta : (2^{\mathscr{U}} \times 2^{\mathscr{U} \times \mathscr{U}}) \times 2^{\mathscr{U}} \rightarrow \{0, 1\}$.

Each step of a serialization can be understood as a transition from one *serialization state* to another serialization state, where the serialization state is defined as a pair (AF, S) with $S \subseteq A$. Each step is guided by α with respect to the selection of the initial set and by β with respect to termination in case β is 1. A finite number of consecutive transitions from one serialization state (AF, S) to another serialization state (AF', S') is denoted as $(AF, S) \rightsquigarrow^{\alpha} (AF', S')$. If β terminates the process at the last state, then $(AF, S) \rightsquigarrow^{\alpha,\beta} (AF', S')$. Now serializability of a semantics can be defined:

Definition 19 *A semantic σ is* serializable *by the selection function α and the termination function β iff for all argumentation frameworks AF we have that $\sigma(AF) = \{S \mid (AF, \emptyset) \rightsquigarrow^{\alpha,\beta} (AF', S)\}$.*

Depending on the semantics to be serialized, the types of initial sets that can be selected and the condition to terminate the construction differ. Not every extension-based semantics can be serialized, but those described here can [24].

Theorem 2 *Let $AF = (A, \succ)$ be an argumentation framework and S an initial set.*

Admissible semantics is serializable with

$\alpha_{adm}(X, Y, Z) = X \cup Y \cup Z$ *and* $\beta_{adm}(AF, S) = 1.$

Complete semantics is serializable with

α_{adm} *and* $\beta_{co}(AF, S) = 1$ *if* $is^{\not\leftarrow}(AF) = \emptyset$*, 0 otherwise.*

Preferred semantics serializable with

α_{adm} *and* $\beta_{co}(AF, S) = 1$ *if* $is(AF) = \emptyset$*, 0 otherwise.*

Stable semantics is serializable with

α_{adm} *and* $\beta_{st}(AF, S) = 1$ *if* $AF = (\emptyset, \emptyset)$*, 0 otherwise.*

Grounded semantics is serializable with

$\alpha_{gr}(X, Y, Z) = X$ *and* β_{co}.

Strongly admissible semantics is serializable with

α_{gr} *and* β_{adm}.

A particular semantics, the *unchallenged* semantics (uc), is defined solely by its serialization sequence [6]:

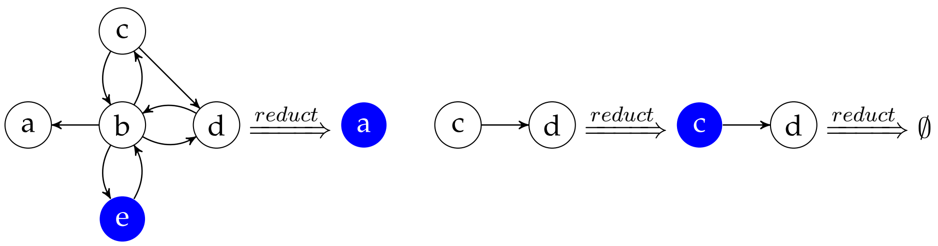
Definition 20 *Let $AF = (A, \succ)$ be an argumentation framework, $S \subseteq A$ and $(S_1, .., S_n)$ be a serialization sequence with $S = S_1 \cup ... \cup S_n$. Then S is an* unchallenged extension *($S \in uc(AF)$) iff for all S_i it holds that $S_i \in is^{\not\leftarrow}(AF^{S_1 \cup ... \cup S_{i-1}}) \cup is^{\not\leftrightarrow}(AF^{S_1 \cup ... \cup S_{i-1}})$ and it holds that $is^{\not\leftarrow}(AF^{S_1 \cup ... \cup S_n}) \cup is^{\not\leftrightarrow}(AF^{S_1 \cup ... \cup S_n}) = \emptyset$.*

To summarize, the corresponding selectable initials sets and termination conditions of each serializable semantics are listed in Table 1.

Example 5 *Figure 4 shows an example of serialization of the argumentation framework given in Figure 1 for a preferred extension with $\{e\}$, $\{a\}$ and $\{c\}$ as exemplified initial sets subsequently chosen, resulting in the serialization sequence ($\{e\}$, $\{a\}$, $\{c\}$). This is not the only solution, since other serialization sequences are possible here, e.g. ($\{c\}$, $\{a\}$, $\{e\}$).*

Semantics	Selectable initial sets	Termination condition
ad	$is(AF)$	after each step
co	$is(AF)$	$is^{\not\leftarrow}(AF) = \emptyset$
pr	$is(AF)$	$is(AF) = \emptyset$
st	$is(AF)$	$AF = (\emptyset, \emptyset)$
gr	$is^{\not\leftarrow}(AF)$	$is^{\not\leftarrow}(AF) = \emptyset$
sa	$is^{\not\leftarrow}(AF)$	after each step
uc	$(is^{\not\leftarrow}(AF) \cup is^{\not\leftrightarrow}(AF))$	$(is^{\not\leftarrow}(AF) \cup is^{\not\leftrightarrow}(AF)) = \emptyset$

Table 1: Selectable initial sets and termination conditions for serializable semantics.



selected initial sets: {e}, {a}, {c}

Figure 4: Example of serialization sequence for a preferred extension.

2.2. Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm that is used primarily for knowledge representation and reasoning. One of the advantages of ASP is its so called "elaboration-tolerance" [16]. This means that the human description of a problem comes close to the ASP encoding, and the ASP encoding is therefore comparatively short. Additionally, minor changes to the underlying problem require only minor changes in the ASP encoding. Due to its purely declarative nature, an ASP encoding for a given problem is the same as the corresponding knowledge base, which consists (mainly) of *rules*, where *facts* and *constraints* are special rules. The knowledge base is processed by the so-called *ASP-Solver*. This solver generates the so-called *stable models* or *answer sets*, which are the minimal models (with respect to set inclusion) that satisfy the given knowledge base. ASP allows the use of default negation, thus enabling non-monotonic reasoning. Unlike Prolog, ASP is not able to handle infinite search spaces, which is not necessary for the purpose of this work, since we only deal with finite sets. On the other hand, ASP can provide all answer sets in one step.

The knowledge base consists of a set of rules of the form

$$H_1, .., H_i : -B_1, .., B_j, not\ C_1, .., not\ C_k$$

where 'not' represents default negation. The rule is - under the so-called *closed world assumption* - equivalent to the propositional logic formula

$$H_1 \lor .. \lor H_i \leftarrow B_1 \land .. \land B_j, \neg C_1 \land .. \land \neg C_k$$

$H_1, .., H_i$ is the *head* of a rule, $B_1, .., B_j, not\ C_1, .., not\ C_k$ is called the *body* of a rule and describes the conditions under which the head becomes true. A fact is a rule without a body (and therefore always true) and a constraint is a rule without a head (and therefore always false). The individual symbols H, B and C are called atoms, which can be predicates or comparisons containing terms with variables and constants. An atom without variables is called a *ground* atom and an answer set of a logic program is a set of ground atoms [21].

2.2.1. Basic Syntax and Semantics

For ASP programming, the integrated ASP system *clingo* (consisting of the grounder *gringo* and the solver *clasp*, [17]), is available at the University of Potsdam[2]. This section gives a brief overview of the syntax for the ASP solver clingo used for this thesis, as far as it is necessary to understand the encodings presented here.

[2]https://potassco.org

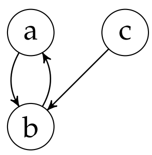
Predicates and constants begin with lowercase letters, while variables begin with uppercase letters. Each rule line must end with a period. The following code examples use abstract argumentation frameworks and simple extensions for demonstration. Applied to an argumentation framework $AF := (A, \succ)$, the facts are given as the set A of arguments and the relation $\succ$. The property of being an element of a certain set is described in ASP as a predicate with arity of one, e.g. 'a' is a constant for $a \in A$ and is encoded as an argument as

```
arg(a).
```

A binary relation is described with a predicate with arity of two, e.g. $a \succ b$ can be encoded as

```
att(a,b).
```

Both correspond to the Aspartix-format [12], which is used for graph encoding. An example of an abstract argumentation framework and its complete coding is shown in Figure 5.



arg(a).
arg(b).
arg(c).
att(a,b).
att(b,a).
att(c,b).

Figure 5: Encoding of an abstract argumentation framework.

Typically, facts are stored in a separate file, because the rules (without facts) are intended to apply to different sets of facts (e.g. different argumentation frameworks). Unlike Prolog, the order of the rules does not matter for the ASP solver and the answer sets. Nevertheless, it has proven advantageous to divide the rules (logically) into the parts: generating, defining and testing. In the first part, a set of solution candidates is generated, then some auxiliary predicates are defined and finally the solution candidates are tested with integrity constraints, eliminating all unwanted solution candidates. The desired solution candidates are retained as answer sets and passed on by the solver. The production of solution candidates with successively elimination of all unwanted candidates, called "guess and check", is the essential functional principle of ASP.

The production of all possible subsets of a given set is particularly useful for finding the extensions of an argumentation framework, since the extensions are subsets of the set of arguments. The subsets of A for example can be constructed using the two *rules*[3]

[3] These rules are from Aspartix (https://www.dbai.tuwien.ac.at/research/argumentation/aspartix/dung.html) for use with clingo

```
in(X)   :-  not out(X), arg(X).
out(X)  :-  not in(X), arg(X).
```
(1)

where X is a variable and `in(X)` indicates that X is an element of the subset; `out(X)` is an auxiliary predicate, that indicates that X is not an element of the subset. These rules can be interpreted as propositional logic formulas:

$$in(X) \leftarrow \neg\, out(X) \wedge arg(X)$$
$$out(X) \leftarrow \neg\, in(X) \wedge arg(X)$$

Example 6 shows the construction of subsets of the set of arguments from the argumentation framework in Figure 5. An alternative way to construct the same subsets (without the auxiliary predicate `out`) is to use a so-called *choice rule*:

```
{ in(X) } :-  arg(X).
```
(2)

Example 6 *For the AF in Figure 5, all eight subsets are given as* $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a,b\}$, $\{a,c\}$, $\{b,c\}$, $\{a,b,c\}$. *The corresponding output of the ASP solver clingo with the encoding in 1 or 2 with respect to the predicate* `in/1` *is:*

Answer 1:[4]

Answer 2:
in(a)
Answer 3:
in(b)
Answer 4:
in(c)
Answer 5:
in(a), in(b)
Answer 6:
in(a), in(c)
Answer 7:
in(b), in(c)
Answer 8:
in(a), in(b), in(c)

To obtain the conflict-free sets of AF, for example, we need to implement the definition of conflict-freeness (Definition 2) with a constraint [3]:

[4] An empty line as answer set represents the empty set.

```
:- in(X), in(Y), att(X,Y).        (3)
```

This constraint is equivalent to the propositional logic formula:

$$false \leftarrow in(X) \land in(Y) \land att(X,Y)$$

and falsifies all solutions with the property described by the constraint. Here solutions with an attack relation within the considered subset are falsified and therefore eliminated from the answer sets.

Example 7 *For the AF in Figure 5, the five conflict-free sets are given as $\emptyset$, $\{a\}$, $\{b\}$, $\{c\}$, $\{a,c\}$. The constraint 3 excludes the answers 5, 7 and 8 .*

The further coding to obtain all admissible sets of AF is shown in 4. The auxiliary predicate `attacked/1`[5] is introduced to flag all arguments, that are attacked from the corresponding subset. The auxiliary predicate `not_defended/1` flags all arguments, that are attacked by arguments, that are not attacked themselves. Finally, a subset containing undefended arguments cannot be admissible and is falsified by the last constraint.

```
       attacked(X) :- in(Y), att(Y,X).
not_defended(X) :- att(Y,X), not attacked(Y).        (4)
                  :- in(X), not_defended(X).
```

ASP allows to determine the cardinality of a set and use it for further processing. The following line flags an answer set with the predicate `not_empty`, if the predicate `in/1` holds for at least one element in the set.

```
not_empty :- { in(X) } > 0.        (5)
```

Comment lines start with a "%", alternatively comments can be embedded between "%*" and "*%". Usually, the user of the program is not interested in the complete answer sets, but in certain predicates of the answer sets. To restrict the solver's output to these predicates, the command `#show pred/n.` can be used, where `pred/n` stands for the desired predicate and its arity n. There are other features of clingo, such as placeholders, strong negation, directives, aggregates etc., that cannot be described here[6].

[5] Predicates are used to be written as "name/n" where n is the arity of the predicate.

[6] See the 'Potassco User Guide' at https://github.com/potassco/guide/releases/ for further details of clingo

Source Code	Output Grounder
arg(a). arg(b). arg(c).	arg(a). arg(b). arg(c).
att(a,b). att(b,a). att(c,b).	att(a,b). att(b,a). att(c,b).
in(X) :- not out(X), arg(X).	in(a):-not out(a). in(b):-not out(b). in(c):-not out(c).
out(X) :- not in(X), arg(X).	out(a):-not in(a). out(b):-not in(b). out(c):-not in(c).
:- in(X), in(Y), att(X,Y).	:-in(b),in(a). :-in(a),in(b). :-in(b),in(c).
defeated(X) :- in(Y), att(Y,X).	defeated(b):-in(a). defeated(a):-in(b). defeated(b):-in(c).
not_defended(X) :- att(Y,X), not defeated(Y).	not_defended(b):-not defeated(a). not_defended(a):-not defeated(b). not_defended(b).
:- in(X), not_defended(X).	:-not_defended(a),in(a). :-in(b).

Table 2: Example of grounding the source code

2.2.2. Grounding

Although the internal operation of the ASP solver is not the focus of this thesis, some aspects are roughly described here, which will be necessary to understand some limitations arising for particular tasks later. An ASP solver works in two steps: first all variables contained in the rules are exchanged by ground atoms (so-called *grounding*), so that the resulting rules are variable-free and only contain ground atoms. As all ground atoms are either true or false, the grounded rules are propositions in first order logic. Those grounded rules are passed to the *solver*, that works similar to a SAT-solver and determines those (minimal) interpretations (answer sets), that correspond to the grounded rules .

Example 8 *Table 2 shows the grounding of the argumentation framework of Figure 5 with the rules 1, 3 and 4. The left column lists the knowledge base with the first two lines containing the facts describing the argumentation framework. The right column shows the corresponding grounded rules. Note that the first two lines are already grounded in the source code.*

2.3. Related Work

Since the introduction of abstract argumentation frameworks by Dung [11], a great amount of work has been published on related topics, particularly with regard to extension based semantics of argumentation frameworks (see [1], for example). The semantics *ad*, *co*, *pr*, *st* and *gr* were already defined in this first publication. The strongly admissible semantics (*sa*) has been introduced by [2] and the unchallenged semantics (*uc*) was recently introduced by [6, 24]. Initial sets as minimal non-empty admissible sets for the construction of set-based extensions were introduced by [27]. The construction principle for initial sets described above, which will also be used for the bachelor thesis, has been described in detail by [24]. In this work, it was also proved that the extensions considered here for serialization are indeed serializable.

3. ASP Encodings for Initial Sets

Encodings for abstract argumentation frameworks in ASP, in particular for the computation of extension-based semantics, have been published by Egly et al. [12] and are also available as "ASPARTIX - Answer Set Programming Argumentation Reasoning Tool"[7]. However, the computation of initial sets and of serialization sequences is not covered in this collection. The published encodings for admissible sets, for complete semantics and preferred semantics[3] are quite straightforward and were therefore (partly) used as a basis for the encodings considered here.

Since ASP code is a set of rules, we can partition an entire ASP program, with each partition typically performing a specific task within the program. Therefore, in the following we will describe the individual tasks of each partition with the applicable rules and define the complete program as a union of these partitions, e.g. for the two ASP encodings P_1 and P_2 their union is defined as $P_1 \cup P_2$[8].

First, the encodings for initial sets are described, since these are the building blocks of the serialization sequences. In the next section encodings for the serialization sequences are presented. In all programs, the arguments belonging to the solution, i.e. initial sets or serialization sequences, are specified with the predicate `in/1` or `in/2`, respectively. Therefore, it is convenient to restrict the output of the ASP solver to these predicates with the rule `#show in/1.` or `#show in/2.`

3.1. Initial Sets

The goal of the ASP program in this section is to provide subsets of arguments that are (plain) initial sets. As described in Section 2.1.2, initial sets are non-empty minimal admissible sets. Typically, the "guess and check"-paradigm is applied by first

[7] https://www.dbai.tuwien.ac.at/research/argumentation/aspartix/

[8] Although there is no formal order of the rules, their tasks must necessarily be described one after the other. However, this should not be understood as if the ASP solver processes the rules in an orderly manner.

generating all possible subsets as solution candidates, which are then each checked for the desired property [14, 13]. Solution candidates that do not fulfil the desired property are excluded from the answer sets. Since ASP-code looks unusual compared to imperative programming languages, the code is first described in more detail and later parts are more tightly bundled.

3.1.1. Admissibility

To generate all subsets of the set of arguments, we use a choice rule and define the program I_{guess}:

Listing 1: The encoding I_{guess} to generate all subsets of arguments.

```
{ in(X) }       :-      arg(X).
```

The unary relation `in/1` indicates that the corresponding argument is an element of the subset. The choice rule implies that each argument may or may not be an element of a solution candidate, which means that every possible subset is a solution candidate, including the empty set and the identity. By definition, every initial set is non-empty, so empty sets must be excluded from the answer. This is achieved by the program I_{non_empty} consisting of two rules:

Listing 2: The encoding I_{non_empty} to exclude empty solution candidates.

```
non_empty       :-      in(X).

                :-      not non_empty.
```

The first rule flags a solution candidate with the predicate `non_empty` if it contains at least one argument. Therefore, only the empty subset is not flagged, which is excluded by the second rule, a constraint. The next property of initial sets is admissibility, which first requires that the set is conflict-free[9]. The program I_{cf} excludes all non conflict-free solution candidates:

Listing 3: The encoding I_{cf} to exclude non-conflict-free solution candidates.

```
                :-      in(X),
                        in(Y),
                        att(X,Y).
```

If a solution candidate has an attack-relation between two of its arguments (represented by the variables X and Y), then it is not conflict-free and is ruled out by the constraint shown. $I_{defence}$ completes the admissibility check:

Listing 4: The encoding $I_{defence}$ to exclude solution candidates with non-defended arguments.

```
attacked(X)     :-      in(Y),
                        att(Y,X).

```

[9] For a better clarity, the body literals of the same rule are placed on top of each other.

```
                    :-      att(Y,X),
                            in(X),
                            not attacked(Y).
```

The first rule of $I_{defence}$ marks all arguments with the unary predicate `attacked/1` that are attacked by the solution candidate (in other words, if the solution candidate is S then `attacked/1` represents S^{+}). Since the solution candidate is conflict-free, the marked arguments cannot be elements of the solution candidate. The second rule is a constraint that excludes all solution candidates with non-defended arguments. So far, all remaining solution candidates are non-empty and admissible. We can therefore define the program $I_{admissible}$ that selects all non-empty admissible sets as the union of the above programs:

$$I_{admissible} := I_{guess} \cup I_{non\_empty} \cup I_{cf} \cup I_{defence} \tag{6}$$

3.1.2. Minimality

Now minimality is the last property to check. One suggestion for checking for minimality could be to reapply the "guess and check"-paradigm to the subsets of each solution candidate. All proper subsets of each solution candidate are checked for admissibility, and minimality of the solution candidate is confirmed if no non-empty subset is admissible. The program P' implements this proposal by using the rules for checking admissibility already presented (with the exception of the check for conflict-freeness, since the subset of a conflict-free set is always conflict-free)[10]:

Listing 5: The encoding P' to generate subsets of solution candidates.

```
{ sub(X) }          :-      in(X).

sub_non_empty       :-      sub(X).

                    :-      not sub_non_empty

sub_attacked(X)     :-      sub(Y),
                            att(Y,X).

                    :-      att(Y,X),
                            sub(X),
                            not sub_attacked(Y).
```

Unfortunately, the proposed program P' does not fulfil the desired task. The choice rule that creates the subsets of each solution candidate does not generate a solution candidate together with all subsets in the same answer, but rather pairs of a solution candidate and only one corresponding subset. Example 9 illustrates this behaviour of the ASP solver.

[10]For reasons of clarity, the treatment of the identity is not shown.

Example 9 *For the argumentation framework in Figure 6 $\{a, b\}$ is obviously not an initial set, since it is not minimal with $\{a\}$ as a non-empty admissible subset. The solution candidate $\{a, b\}$ with the subset $\{a\}$ is therefore excluded. However, the solution candidate $\{a, b\}$ with $\{b\}$ as subset is not excluded, but confirms $\{a, b\}$ as an initial set, since it only checks $\{b\}$ for admissibility.*

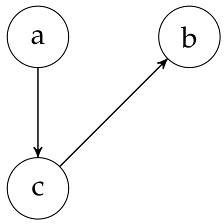


Figure 6: Example AF_3 of an abstract argumentation framework.

Since minimality for the described algorithm requires that each subset is tested for admissibility, it is not sufficient to perform a pairwise check. An alternative would be to reason over the collection of answer sets, which is not possible within the ASP program, but requires external post-processing. Another possibility could be to apply the so called "saturation technique", which uses a disjunctive program that exploits the minimality criterion for answer sets. However, this technique is advanced and not easily applicable; moreover, the use of default-negation within saturation encodings is limited [22]. Since we make use of default-negations, it is not indicated to use of the saturation technique here. Consequently, a solution within a single ASP program requires a data structure that represents the subsets within the same solution candidate. To check all subsets of the solution candidate 2^n checks would have to be performed (with cardinality n of the corresponding solution candidate), which is not efficient. Instead, to check for minimality in polynomial time, we use the following results [24]:

Proposition 3 *Let $AF = (A, \succ)$ be an argumentation framework. To verify whether a set $S \subseteq A$ is an initial set, can be computed in polynomial time.*

Proposition 4 *Let $AF = (A, \succ)$ be an argumentation framework and $S \subseteq A$, $S \in cf(AF)$ and $a \in S$. Deciding whether there is an admissible set $S' \subseteq S$ with $a \in S'$ can be computed in polynomial time.*

The algorithm proposed in the proof of Proposition 3 works by checking subsets of S, each decremented by one argument, for admissible sets. For each $a \in S$ it is checked whether the subset $S \setminus a$ contains an admissible set. If none of these subsets contains an admissible subset, then S is an initial set (we have already shown that S itself is admissible). To check for admissible subsets according to Proposition 4 the non-defended arguments are gradually removed from the subset and checked for admissibility (for proofs see [24]). The pseudocode for this algorithm is shown in Algorithm 1.

Algorithm 1 Checking minimality of S

Input: $AF = (A, \succ), S \subseteq A, S \in cf(AF)$
Output: YES iff S is minimal admissible, otherwise NO
for all $a \in S$ **do**
  $S' := S \setminus a$
  if $S' \neq \emptyset$ **then**
    if S' is admissible **then**
      return NO
    else
      $i = 0$
      $S_0 = S'$
      while $S_i \neq \emptyset$ **do**
        $i = i + 1$
        $S_i = S_{i-1} \cap \Delta_{AF}(S_{i-1})$
        if S_i is admissible **then**
          return NO
        end if
      end while
    end if
  end if
end for
return YES

To test the described subsets, an auxiliary data structure is required that allows targeted access to the arguments individually and in an ordered manner. We use the relation "<" already integrated in clingo, which allows the arguments to be sorted alphabetically, used by I_{order}[3]:

Listing 6: The encoding I_{order} to define an order over the arguments of the solution candidates.

```
lt(X, Y)            :-      in(X),
                            in(Y),
                            X<Y.

nsucc(X, Z)         :-      lt(X, Y),
                            lt(Y, Z).

succ(X, Y)          :-      lt(X, Y),
                            not nsucc(X, Y).

ninf(X)             :-      lt(Y, X).
```

The first rule of the program I_{order} defines a less-than relation (`lt/2`) over the arguments of the solution candidate. Then the relation `nsucc/2` denotes those pairs of arguments that do not directly follow one another. The negation of `nsucc/2` then designates those arguments that follow one another directly and finally the fourth rule with `ninf/1` designates all those arguments that are not in first place in the sequence, i.e. only the first argument does not have this predicate. This order is then used to define subsets of the solution candidates in I_{sub} (see lines 3-4 of the pseudocode):

Listing 7: The encoding I_{sub} to construct decremented subsets of the solution candidates.

```
excl(X, 1)          :-      not ninf(X),
                            in(X).

excl(Y, No+1)       :-      excl(X, No),
                            in(Y),
                            succ(X, Y).

sub(X, No)          :-      in(X),
                            not excl(X, No),
                            cArg(C),
                            No = 1..C.

sub(X, No, 0)       :-      sub(X,No).
```

The predicate `excl/2` lists all arguments in an ordered manner to define the subsets decremented by one argument with the predicate `sub/2`. In `sub/3`, the predicate `sub/2` is extended by a third dimension, which represents the first "level" of the algorithm described for Proposition 4 (corresponding to the variable i in line 12 of the pseudocode). Next, the subsets must be tested for admissibility by I_{sub_adm} (see lines 6 and 14 of the pseudocode), which works in part similarly to I_{adm}:

Listing 8: The encoding I_{sub_adm} to check for admissibility of subsets.

```
sub_attacked(Y, No, Level):-    sub(X, No, Level),
                                att(X, Y).

non_def(Y, No, Level)   :-      sub(Y, No, Level),
                                att(X, Y),
                                not sub_attacked(X, No, Level).

non_adm(No, Level)      :-      non_def(Y, No, Level).
```

The first rule corresponds to the first rule of I_{adm}. The body of the second rule is similar to that of I_{adm}, but the rule is not a constraint but rather collects the non-defended arguments of each subset. If a subset contains at least one non-defended argument, it is flagged with `non_adm/2` in the third rule. After collecting the non-defended arguments, we can use I_{next} to define the corresponding subsubset that does not contain the non-defended arguments (see line 13 of the pseudocode):

Listing 9: The encoding I_{next} to construct subsets with non-defended arguments.

```
card(C)             :-      { in(X) } == C.
```

```
2
3        sub(X, No, Level+1)     :-      sub(X, No, Level),
4                                        not non_def(X, No, Level),
5                                        card(C),
6                                        Level < C.
```

The first rule stores the number of arguments in the predicate `card/1`, since the maximum level must be smaller than this value. Please note that the predicate `sub/3` in the body of the second rule (line 3) is required to ensure the so called *safety* of the rule. Every rule of an ASP program must be safe in the sense that every variable in that rule (especially those in the head and in negative literals) must occur in at least one positive literal in the body of that rule [16].

Finally, if there is a non-empty admissible subset of the solution candidate, this subset is not flagged by `non_adm/2`. In this case, the solution candidate cannot be a minimal non-empty admissible set and thus must be excluded by a constraint in I_{non_min} (lines 7 and 15 of the pseudocode):

Listing 10: The encoding I_{non_min} to exclude solution candidates with non-empty admissible subset.

```
                                 :-      not non_adm(No, Level),
                                         sub(X, No, Level).
```

The part $I_{minimality}$ of the program that checks minimality can now be defined:

$$I_{minimality} := I_{order} \cup I_{sub} \cup I_{adm\_sub} \cup I_{next} \cup I_{non\_min} \tag{7}$$

The complete ASP program for selecting initial sets $I_{initial}$ is:

$$I_{initial} := I_{admissible} \cup I_{minimality} \tag{8}$$

Unfortunately, the data structure needed to check for minimality requires comparatively high effort and leads to a longer encoding. In addition, the encoding becomes more difficult to understand, so that the elaboration tolerance of ASP (see Section 2.2) is partially lost.

3.2. Unattacked Initial Sets

For the algorithm for selecting unattacked initial sets, we take advantage of the property that unattacked initial sets always have a cardinality of 1 (see Proposition 2). Following the "guess and check"-paradigm, the solution candidates are created by I_{guess} (see Listing 1). Solution candidates with cardinality not equal to 1 and solution candidates with attacked arguments are excluded by I_{att}:

Listing 11: The encoding I_{att} to exclude solution candidates with unsuitable sequence terms.

```
                                 :-      { in(X) } != 1.
```

```

                                      :-      in(X),
                                              att(Y,X),
                                              arg(Y).
```

The first rule of I_{att} is a so called *cardinality constraint*, where the term `{ in(X) }` represents the cardinality of the set of atoms with the predicate `in/1` in a solution candidate. The program $I_{unattacked}$ for selecting unattacked initial sets is defined as follows:

$$I_{unattacked} := I_{guess} \cup I_{att} \tag{9}$$

3.3. Unchallenged Initial Sets

Unchallenged initial sets are attacked, but not from another initial set. To select unchallenged initial sets, we first take all initial sets (selected by $I_{initial}$, see 8) and exclude solution candidates that are not attacked at all or are attacked by another initial set. The exclusion of unattacked sets is done by I_{excl_unatt}:

Listing 12: The encoding I_{excl_unatt} to exclude unattacked solution candidates.

```
      in_attacked             :-      in(X),
                                      att(Y,X),
                                      arg(Y).

                              :-      not in_attacked.
```

To check whether a solution candidate is attacked by another initial set, it must be tested whether the "attacking set" is initial. Since the attacking set can be any set of arguments (except for the solution candidate itself), one suggestion may be to check all $2^n - 1$ subsets of arguments, which is not very efficient due to the exponential number of subsets. Another suggestion could be to use the Algorithm 1, but this algorithm is not applicable because it requires a conflict-free set as input, and we cannot assume that the set of arguments is conflict-free. Instead, we could select the maximum conflict-free sets (or all conflict-free sets) before algorithm 1 could be applied. A conflict-free set is identical to a so called "independent set" of edges of a graph, which is defined as a set of edges of which no two are adjacent. Finding the maximum independent sets of a graph is the same as finding the so called "maximum cliques" of the complementary graph [25]. Both problems are NP-complete [26], so that this approach is not necessarily better. However, the so called "Bron-Kerbosch algorithm", which solves this problem, has been reported to be the fastest algorithm in practise [15]. This algorithm is not available for ASP and it is unclear whether this would be more efficient in ASP than reasoning over all subsets. Therefore, for better understanding, here a data structure is created within each solution candidate that contains all subsets of the arguments.

Similar to I_{order} in Listing 6, all arguments are ordered and numbered accordingly (starting with 0) using I_{order_args}:

arg		a	b	c	d	**SetNo**
ArgNo		0	1	2	3	
subset	{a}	1	0	0	0	1
	{b}	0	1	0	0	2
	{a, b}	1	1	0	0	3
	{b, c, d }	0	1	1	1	14

Table 3: Example of bit vectors to represent subsets.

Listing 13: The encoding I_{order_args} to order and number the arguments.

```
a_lt(X,Y)              :-      arg(X),
                               arg(Y), X<Y.

a_nsucc(X,Z)           :-      a_lt(X,Y),
                               a_lt(Y,Z).

a_succ(X,Y)            :-      a_lt(X,Y),
                               not a_nsucc(X,Y).

a_ninf(X)              :-      a_lt(Y,X).

arg(X, 0)              :-      not a_ninf(X),
                               arg(X).

arg(Y, ArgNo+1)        :-      arg(X, ArgNo),
                               arg(Y),
                               a_succ(X, Y).
```

Now we assign each subset to a corresponding bit vector. The length of this vector equals the argument count and each position within this vector refers to the corresponding number of the argument. The value “1” indicates, that the corresponding argument is an element of the subset, “0” means the opposite. Interpreted as a bit value, the vector also represents the (consecutive) number of the corresponding subset. Table 3 shows an example of such a bit vector. The bit vector and the required relations are generated by the program I_{bit}:

Listing 14: The encoding I_{bit} generating the bit vectors representing subsets.

```
a_card(C)              :-      { arg(X) } == C.

a_set(1..SetNo)        :-      (2 ** C) - 1 == SetNo,
                               a_card(C).

a_vec(SetNo, 0, SetNo\2, SetNo/2) :-
                               a_set(SetNo).

a_vec(SetNo, ArgNo+1, Result\2, Result/2) :-
                               a_vec(SetNo, ArgNo, _, Result),
                               SetNo >= (2 ** (ArgNo+1)).
```

```

a_elem(SetNo, ArgNo)     :-      a_vec(SetNo, ArgNo, Rest, _),
                                 Rest = 1.

a_sub(SetNo, SubSet)     :-      a_vec(SetNo, ArgNo, Rest, Result),
                                 Rest = 1,
                                 SubSet = 2 ** (ArgNo).

a_sub(SetNo, SubA+SubB) :-       a_sub(SetNo, SubA),
                                 a_sub(SetNo, SubB),
                                 SubA != SubB,
                                 SubA + SubB < SetNo.
```

The numbers of the subsets are stored in the predicate `a_set/1` (with the maximum $2^C - 1$, where C is number of arguments). With `a_set/1` we can calculate the bit vector `a_vec` by repeatedly divide the set number by 2. The rest of the division is 1 or 0 and assigns argument to the corresponding subset. The predicate `a_elem/2` assigns set numbers to argument numbers, and the predicate `a_sub/2` is used to decide whether a set is a subsets of another set. With this data structure we can now flag all subsets that are conflicting, non-admissible or non-minimal,which is done by I_{flag}:

Listing 15: The encoding I_{flag} to flag non-initial subsets.

```
a_flag(SetNo)            :-      a_elem(SetNo, ArgNo1),
                                 a_elem(SetNo, ArgNo2),
                                 arg(X, ArgNo2),
                                 arg(Y, ArgNo1),
                                 att(X, Y).

a_attacked(SetNo, X)     :-      a_elem(SetNo, ArgNo),
                                 arg(Y, ArgNo),
                                 att(Y, X).

a_flag(SetNo)            :-      a_elem(SetNo, ArgNo),
                                 arg(X, ArgNo),
                                 att(Y,X),
                                 not a_attacked(SetNo, Y).

a_flag(SetNo1)           :-      a_set(SetNo1),
                                 a_set(SetNo2),
                                 SetNo1 != SetNo2,
                                 a_sub(SetNo1, SetNo2),
                                 not a_flag(SetNo2).
```

Finally, the non-flagged subsets are assigned as initial sets and their elements are assigned as elements of initial sets. If such an argument attacks the solution candidate, this answer set is excluded by I_{ini_att}:

Listing 16: The encoding I_{ini_att} to exclude solution candidates attacked by initial sets.

```
iniSet(SetNo)            :-      a_set(SetNo),
```

```
                                     not a_flag(SetNo).

        elemIni(SetNo, X)       :-   iniSet(SetNo),
                                     a_elem(SetNo, ArgNo),
                                     arg(X, ArgNo).

                                :-   elemIni(SetNo, X),
                                     in(Y),
                                     att(X,Y).
```

The complete encoding for unchallenged initial sets can be defined as:

$$I_{unchallenged} := I_{initial} \cup I_{excl\_unatt} \cup I_{order\_args} \cup I_{bit} \cup I_{flag} \cup I_{ini\_att} \tag{10}$$

3.4. Challenged Initial Sets

Challenged initial sets are attacked from other initial sets. Apart from that, the computation is the same as for unchallenged initial sets. Therefore, we only need to change the last rule of program I_{ini_att}, to obtain the program $I_{non\_ini\_att}$, which excludes solution candidates that are not attacked by an initial set:

Listing 17: The encoding $I_{non\_ini\_att}$ to exclude solution candidates not attacked by initial sets.

```
        iniSet(SetNo)           :-   a_set(SetNo),
                                     not a_flag(SetNo).

        elemIni(SetNo, X)       :-   iniSet(SetNo),
                                     a_elem(SetNo, ArgNo),
                                     arg(X, ArgNo).

        ini_attack              :-   elemIni(SetNo, X),
                                     in(Y),
                                     att(X,Y).

                                :-   not ini_attack.
```

The complete encoding for challenged initial sets can now be defined as:

$$I_{challenged} := I_{unchallenged} \setminus I_{ini\_att} \cup I_{non\_ini\_att} \tag{11}$$

4. ASP Encodings for Serialization Sequences

In this section the encodings for the serialization sequences are presented with respect to the described semantics.

Since a serialization sequence is an ordered set of initial sets, the solution candidates for serialization sequences must consist of ordered sets of subsets of arguments (sequence terms[11]). Unlike I_{guess} (see Listing 1), the predicate representing a solution candidate must be binary and specify the arguments of each sequence term and its index in the sequence. As usual, the index is specified by ascending integers starting with 1. The maximum length of a sequence (= maximum number of sequence terms) is bounded by the number of arguments in the complete argumentation framework, since each initial set must contain at least one argument.

The following subsections describe the programs for computing the serialization sequences of the different semantics, starting with the serialization sequence for admissible sets. Subsequently, the programs for the other semantics are described, often using building blocks from the previous programs and/or slightly adapting them.

4.1. Admissible Sets

The program to compute the serialization sequences for admissible sets requires to generate sequences of initial sets. First the length of the sequence must be limited to the number of arguments in the argumentation framework. Because initial sets are non-empty, they must each contain at least one argument, so that the number of initial sets in a sequence is limited by the number of arguments in the framework. The program P_{count} is used to determine this number and the corresponding indices:

Listing 18: The encoding P_{count} to count arguments.

```
index(1..C)          :-     { arg(X) } == C.
```

The number of arguments is stored to the variable C and the rule assigns the indices from 1 to C to the predicate `index/1`. The next part is the construction of the reduct which is shown in program P_{reduct}:

Listing 19: The encoding P_{reduct} to construct the reducts of a sequence.

```
reduct(X, 1)         :-     arg(X).

collect(X, Step)     :-     in(X, Step).

collect(X, Step)     :-     in(Y, Step),
                            att(Y, X).

reduct(X, Step+1)    :-     reduct(X, Step),
                            not collect(X, Step),
                            index(Step).

att(X, Y, Step)      :-     reduct(X, Step),
                            reduct(Y, Step),
                            att(X,Y).
```

The reduct is represented by the predicate `reduct/2`. The first rule creates the first reduct, which corresponds to the complete argumentation framework. The

[11]To avoid confusion, the initial sets that make up the serialization sequence are called “sequence terms” and the arguments belonging to a sequence term are called “elements”.

arguments in the current sequence term (`in/2`, see below) and the arguments attacked from arguments of the current sequence term are collected with the predicate `collect/2`. All non-collected arguments of the current reduct are then assigned to the next reduct. Finally, the attack-relation is defined for each reduct. Next the solution candidates are created by the program P_{guess}:

Listing 20: The encoding P_{guess} to generate sequences of sets of arguments.

```
        { in(X, Step) }         :-      reduct(X, Step).
```

This choice rule generates sequences of subsets as solution candidates (similar to I_{guess}, see Listing 1), which are represented by `in/2`, where `Step` specifies the index within the sequence. The arguments for the solution candidates are taken from the corresponding reduct. After the solution candidates have been created, each sequence term is tested for non-emptiness and admissibility using the program P_{adm}:

Listing 21: The encoding P_{adm} to exclude solution candidates with empty and non-admissible sequence terms.

```
        non_empty(Step)         :-      in(X, Step).

                                :-      not non_empty(Step),
                                        non_empty(Step+1),
                                        index(Step).

                                :-      in(X, Step),
                                        in(Y, Step),
                                        att(X, Y).

        attacked(X, Step)       :-      in(Y, Step),
                                        att(Y, X, Step).

                                :-      att(Y, X, Step),
                                        in(X, Step),
                                        not attacked(Y, Step).
```

The first two rules exclude solution candidates with empty sequence terms that are not at the end of the sequence. The third rule excludes solution candidates with sequence terms, that are not conflict-free (similar to I_{cf} in Listing 3). The fourth and fifth rule exclude non-admissible solution candidates (see $I_{defence}$ in Listing 4). As a result, all remaining solution candidates have only non-empty admissible sequence terms.

The minimality condition must be fulfilled by every sequence term, i.e. solution candidates with at least one non-minimal sequence terms must be excluded. For this purpose the program $I_{minimality}$ (see Listing 7) is extended with an additional dimension to represent the single sequence terms of a solution candidate (specified by `Step`):

Listing 22: The encoding P_{order} to define an order over the arguments of the sequence terms.

```
lt(X, Y, Step)          :-      in(X, Step),
                                in(Y, Step),
                                X<Y.

nsucc(X, Z, Step)       :-      lt(X, Y, Step),
                                lt(Y, Z, Step).

succ(X, Y, Step)        :-      lt(X, Y, Step),
                                not nsucc(X, Y, Step).

ninf(X, Step)           :-      lt(Y, X, Step).
```

Listing 23: The encoding P_{sub} to construct decremented subsets of the sequence terms.

```
excl(X, 1, Step)        :-      not ninf(X, Step),
                                in(X, Step).

excl(Y, No+1, Step)     :-      excl(X, No, Step),
                                in(Y, Step),
                                succ(X, Y, Step).

sub(X, No, Step)        :-      in(X, Step),
                                not excl(X, No, Step),
                                in_index(No, Step).

sub(X, No, Step, 0)     :-      sub(X, No, Step).
```

Listing 24: The encoding P_{adm_sub} to check for admissibility of subsets of sequence terms.

```
sub_attacked(Y, No, Step, Level):-
                                sub(X, No, Step, Level),
                                att(X, Y, Step).

non_def(Y, No, Step, Level):-   sub(Y, No, Step, Level),
                                att(X, Y, Step),
                                not sub_attacked(X, No, Step, Level).

non_adm(No, Step, Level):-      non_def(Y, No, Step, Level).
```

Listing 25: The encoding P_{next} to construct subsets with non-defended arguments.

```
sub(X, No, Step, Level+1):-     sub(X, No, Step, Level),
                                not non_def(X, No, Step, Level),
                                in_card(C, Step),
                                Level < C.
```

Listing 26: The encoding P_{non_min} to exclude solution candidates with non-empty admissible subsets.

```
                        :-      not non_adm(No, Step, Level),
```

```
                                        sub(X, No, Step, Level).
```

For P_{sub} (Listing 23) and P_{next} (Listing 25) the indices and the cardinality of the single sequence terms are needed, which is provided by P_{count_sub}:

Listing 27: The encoding P_{count_sub} to count the arguments of the sequence terms.

```
in_index(1..C, Step)     :-      { in(X, Step) } == C,
                                 index(Step).

in_card(C, Step)         :-      { in(X, Step) } == C,
                                 index(Step).
```

The part $P_{minimality}$ of the program that checks minimality can now be defined:

$$P_{minimality} := P_{order} \cup P_{sub} \cup P_{adm\_sub} \cup P_{next} \cup P_{non\_min} \cup P_{count\_sub} \tag{12}$$

The following program P_{SerSeq_ad} generates serialization sequences that consist of initial sets, which is the serialization sequence for admissible sets:

$$P_{SerSeq\_ad} := P_{count} \cup P_{reduct} \cup P_{guess} \cup P_{adm} \cup P_{minimality} \tag{13}$$

4.2. Complete Semantics

A serialization sequence for complete semantics consists of initial sets with the restriction that the last reduct of the sequence must satisfy the termination condition. The latter is $is^{\not\leftarrow}(AF) = \emptyset$, i.e. there are no unattacked arguments in the reduct. An suitable algorithm is to take the serialization sequences generated for admissible sets and exclude those solution candidates that do not satisfy the termination condition. This is done by P_{term_co}:

Listing 28: The encoding P_{term_co} to exclude solution candidates not fulfilling the termination condition for complete semantics.

```
flag(X, Step)            :-      reduct(X, Step),
                                 reduct(Y, Step),
                                 att(Y, X, Step).

non_terminate(Step)      :-      reduc(X, Step),
                                 not flag(X, Step).

                         :-      non_empty(Step),
                                 not non_empty(Step+1),
                                 non_terminate(Step+1),
                                 Step > 0.

                         :-      not non_empty(1),
                                 non_terminate(1).
```

First, all attacked arguments in the reduct are flagged, non-flagged arguments indicate unattacked initial sets. If there is no non-flagged argument in the reduct, the termination condition is not met. The third rule excludes solution candidates whose last reduct does not satisfy the termination condition. The forth rule is to treat the empty set as a solution candidate.

The program P_{SerSeq_co} for the complete semantics can be defined as follows:

$$P_{SerSeq\_co} := P_{SerSeq\_ad} \cup P_{term\_co} \tag{14}$$

4.3. Stable Semantics

The serialization sequence for stable semantics consists of initial sets that leave an empty reduct, i.e. $AF = \emptyset$. The only difference from the code for complete semantics is the termination condition, encoded by P_{term_st}:

Listing 29: The encoding P_{term_st} to exclude solution candidates not fulfilling the termination condition for stable semantics.

```
                    :-      not non_empty(Step),
                            reduct(X, Step).
```

The rule excludes all solution candidates with an argument in the last reduct. The program P_{SerSeq_st} for the stable semantics can be defined as follows:

$$P_{SerSeq\_st} := P_{SerSeq\_ad} \cup P_{term\_st} \tag{15}$$

4.4. Preferred Semantics

The serialization sequence for preferred semantics consists of initial sets with the restriction that the last reduct of the sequence must not contain an initial set, i.e. $is(AF) = \emptyset$. Other than for complete or stable semantics, the termination condition requires reasoning over all subsets of the reduct. As with the computation of unchallenged initial sets (see Section 3.3) Algorithm 1 is not applicable, since this requires a conflict-free set as input and we can not assume here the reduct to be conflict-free. As a consequence, a data structure containing all subsets of the reduct is needed for each solution candidate. This is done similar to I_{order_out} and I_{bit} with the distinction, that here subsets of the reducts need to be generated. First, the elements of the reducts are ordered and numbered with the program P_{order_reduct}:

Listing 30: The encoding P_{order_reduct} to order and number the arguments of the reducts.

```
r_lt(X, Y, Step)    :-      reduct(X, Step),
                            reduct(Y, Step),
                            X<Y.

```

```
        r_nsucc(X, Z, Step)     :-      r_lt(X, Y, Step),
                                        r_lt(Y, Z, Step).

        r_succ(X, Y, Step)      :-      r_lt(X, Y, Step),
                                        not r_nsucc(X, Y, Step).

        r_ninf(X, Step)         :-      r_lt(Y, X, Step).

        reduct(X, Step, 0)      :-      not r_ninf(X, Step),
                                        reduct(X, Step).

        reduct(Y, Step, ArgNo+1):-      reduct(X, Step, ArgNo),
                                        reduct(Y, Step),
                                        r_succ(X, Y, Step).
```

Similar to I_{bit} a bit vector is needed, that represents the subsets of the reducts, which is done by P_{bit_pr}:

Listing 31: The encoding P_{bit_pr} to generate bit vectors representing subsets of the reducts.

```
        r_card(C, Step)         :-      {reduct(X, Step)} == C,
                                        card(Ca),
                                        RStep = Ca + 1,
                                        Step = 1..RStep.

        binVec(SetNo, 0, SetNo\2, SetNo/2) :-
                                        r_card(C, 2),
                                        (2 ** C) - 1 = Max,
                                        SetNo = 1..Max.

        binVec(SetNo, ArgNo+1, Result\2, Result/2) :-
                                        binVec(SetNo, ArgNo, _, Result),
                                        SetNo >= (2 ** (ArgNo+1)).

        r_set(1..MaxSet, Step)  :-      (2 ** C) - 1 == MaxSet,
                                        r_card(C, Step).

        r_elem(SetNo, ArgNo)    :-      binVec(SetNo, ArgNo, Rest, _),
                                        Rest = 1.

        r_elem(SetNo, ArgNo, Step) :-   r_set(SetNo, Step),
                                        r_elem(SetNo, ArgNo).
```

Other than with unchallenged initial sets we do not have to test for minimality here. It is sufficient to check, whether the reduct contains any non-empty admissible set. If this is the case, then the reduct must also contain an initial set. Therefore, conflicting subsets and non-admissible subsets are flagged with the program P_{flag_pr}:

Listing 32: The encoding P_{flag_pr} to flag non-admissible subsets of the reducts.

```
        flag(SetNo, Step)       :-      r_elem(SetNo, ArgNo1, Step),
                                        reduct(X, Step, ArgNo1),
```

```
                                    r_elem(SetNo, ArgNo2, Step),
                                    reduct(Y,Step,ArgNo2),
                                    att(X,Y).

r_attacked(SetNo, X, Step) :-   r_elem(SetNo, ArgNo, Step),
                                    reduct(Y, Step, ArgNo),
                                    att(Y, X, Step).

flag(SetNo, Step)       :-      r_elem(SetNo, ArgNo, Step),
                                    reduct(X, Step, ArgNo),
                                    att(Y, X, Step),
                                    not r_attacked(SetNo, Y, Step).
```

Reducts with a non-flagged subset must contain an initial set. Solution candidates with such a reduct at the last position cannot represent a serialization sequence for *pr* and must be excluded. The same is true, if the solution candidate is the empty set with a reduct containing an initial set. This is provided by the program P_{term_pr}:

Listing 33: The encoding P_{term_pr} to exclude solution candidates with improper last reducts.

```
non_terminate(Step)     :-      r_set(SetNo, Step),
                                    not flag(SetNo, Step).

                        :-      non_empty(Step),
                                    not non_empty(Step+1),
                                    non_terminate(Step+1),
                                    Step > 0.

                        :-      not non_empty(1),
                                    non_terminate(1).
```

The complete program for computing the serialization sequences of *pr* is defined as follows:

$$P_{SerSeq\_pr} := P_{SerSeq\_ad} \cup P_{order\_reduct} \cup P_{bit} \cup P_{flag} \cup P_{term\_pr} \tag{16}$$

4.5. Grounded Semantics

Unlike the four previous semantics, the sequence terms of the serialization sequence for the grounded semantics are unattacked initial sets. Since unattacked initial sets are singletons, this reduces the algorithmic effort. To generate the solution candidates we use of the already defined programs P_{count}, P_{reduct} and P_{guess} (see Listings 18, 19 and 20). The following program P_{excl} is used to exclude unsuitable solution candidates:

Listing 34: The encoding P_{excl} to exclude unsuitable solution candidates for grounded semantics.

```
non_empty(Step)         :-      in(X, Step).

```

```
                              :-    not non_empty(Step),
                                    non_empty(Step+1),
                                    index(Step).

                              :-    in(X, Step),
                                    in(Y, Step),
                                    X != Y.

                              :-    in(X, Step),
                                    att(Y, X, Step),
                                    reduct(Y, Step).
```

The first two rules exclude solution candidates with 'intermediate' empty sequence terms (see Listing 21). The third rule excludes solution candidates with non-singleton sequence terms and the last rule excludes solution candidates with attacked arguments.

The termination condition is the same as for complete semantics, so we can take the program P_{term_co}. The program P_{SerSeq_gr} for the grounded semantics can now be defined:

$$P_{SerSeq\_gr} := P_{count} \cup P_{reduct} \cup P_{guess} \cup P_{excl} \cup P_{term\_gr} \tag{17}$$

4.6. Strongly Admissible Semantics

Similar to grounded semantics, the serialization sequence for strong admissible semantics consists of unattacked initial sets. Unlike grounded semantics, there is no termination condition to be computed. Therefore, the program P_{SerSeq_sa} for the strongly admissible semantics can be defined a follows:

$$P_{SerSeq\_sa} := P_{count} \cup P_{reduct} \cup P_{guess} \cup P_{excl} \tag{18}$$

4.7. Unchallenged Semantics

The serialization sequence for the unchallenged semantics consists of non-challenged initial sets. To obtain such sequences, sequences consisting of initial sets are first generated using P_{SerSeq_ad} (see 13). Subsequently, sequences containing challenged terms must be excluded, i.e. that are attacked by initial sets. For this purpose, the subsets of the reduct must be checked for being initial sets, using the data structure of the bit vector already used for the unchallenged initial sets and the preferred semantics (see Sections 3.3 and 4.4). To order and number the elements of the reduct, we can use the already defined program P_{order_reduct} (see Listing 30). The program for the bit vector P_{bit_uc} is slightly modified compared to P_{bit_pr} (see Listing 31) and also allows to relate sets to its subsets (see program I_{bit}, Listing 14):

Listing 35: The encoding P_{bit_uc} for the bit vector used for unchallenged semantics.

```
binVec(SetNo, 0, SetNo\2, SetNo/2) :-
                                card(C),
                                (2 ** C) - 1 = Max,
                                SetNo = 1..Max.

binVec(SetNo, ArgNo+1, Result\2, Result/2) :-
                                binVec(SetNo, ArgNo, _, Result),
                                SetNo >= (2 ** (ArgNo+1)).

elem(SetNo, ArgNo)      :-      binVec(SetNo, ArgNo, Rest, _),
                                Rest = 1.

sub(SetNo, SubSet)      :-      binVec(SetNo, ArgNo, Rest, Result),
                                Rest = 1,
                                SubSet = 2 ** (ArgNo).

sub(SetNo, SubA+SubB)   :-      sub(SetNo, SubA),
                                sub(SetNo, SubB),
                                SubA != SubB,
                                SubA + SubB <= SetNo.

r_card(C, Step)         :-      { reduct(X, Step) } = C,
                                index(Step).

r_set(1..MaxSet, Step)  :-      (2 ** C) - 1 == MaxSet,
                                r_card(C, Step).

r_elem(SetNo, ArgNo, Step):-    r_set(SetNo, Step),
                                elem(SetNo, ArgNo).

r_sub(SetNo, SubSet, Step):-    r_set(SetNo, Step),
                                sub(SetNo, SubSet).
```

Those non-empty subsets of the reduct that are not conflict-free, not admissible, or not minimal are flagged with the program P_{flag_uc}:

Listing 36: The encoding P_{flag_uc} to flag non-initial subsets of the reduct.

```
flag(SetNo, Step)       :-      r_elem(SetNo, ArgNo1, Step),
                                r_elem(SetNo, ArgNo2, Step),
                                reduct(X, Step, ArgNo2),
                                reduct(Y, Step, ArgNo1),
                                att(X, Y, Step).

r_attacked(SetNo, X, Step):-    r_elem(SetNo, ArgNo, Step),
                                reduct(Y, Step, ArgNo),
                                att(Y, X, Step).

flag(SetNo, Step)       :-      r_elem(SetNo, ArgNo, Step),
                                reduct(X, Step, ArgNo),
                                att(Y, X, Step),
                                not r_attacked(SetNo, Y, Step).

```

```
        flag(SetNo1, Step)      :-      r_set(SetNo1, Step),
                                        r_set(SetNo2, Step),
                                        SetNo1 != SetNo2,
                                        r_sub(SetNo1, SetNo2, Step),
                                        not flag(SetNo2, Step).
```

The non-flagged subsets are initial sets of the reduct. Terms attacked by non-flagged subsets are challenged initial sets, which must be excluded with the program P_{excl_cha} (see similar I_{ini_att}, Listing 16):

Listing 37: The encoding P_{excl_cha} to exclude solution candidates with challenged initials sets.

```
        iniSet(SetNo, Step)     :-      r_set(SetNo, Step),
                                        not flag(SetNo, Step).

        elemIni(SetNo, X, Step):-       iniSet(SetNo, Step),
                                        r_elem(SetNo, ArgNo, Step),
                                        reduct(X, Step, ArgNo).

                                :-      elemIni(SetNo, X, Step),
                                        in(Y, Step),
                                        att(X,Y, Step).
```

The remaining sequence terms are unattacked or unchallenged initials sets. To satisfy the termination condition, the reduct must not contain any unattacked or unchallenged initial sets. This is checked with the program P_{term_uc}:

Listing 38: The encoding P_{term_uc} checking the termination condition for unchallenged semantics.

```
        r_sign(SetNo, Step)     :-      flag(SetNo, Step).

        r_sign(SetNo1, Step)    :-      r_elem(SetNo1, ArgNo1, Step),
                                        reduct(X, Step, ArgNo1),
                                        att(Y, X, Step),
                                        reduct(Y, Step, ArgNo2),
                                        r_elem(SetNo2, ArgNo2, Step),
                                        not flag(SetNo2, Step).

        non_terminate(Step)     :-      r_set(SetNo, Step),
                                        not r_sign(SetNo, Step).

                                :-      non_empty(Step),
                                        not non_empty(Step+1),
                                        non_terminate(Step+1),
                                        Step > 0.

                                :-      not non_empty(1),
                                        non_terminate(1).
```

In addition to the already flagged subsets of the reduct, those subsets that are attacked by an initial set are also signed. All unsigned subsets are now unattacked or unchallenged initial sets. If there is at least one such unsigned subset in the reduct, the termination condition is not satisfied.

The complete program P_{SerSeq_uc} for the unchallenged semantics can be defined as follows:

$$P_{SerSeq\_uc} := P_{SerSeq\_ad} \cup P_{order\_reduct} \cup P_{bit\_uc} \cup P_{flag\_uc} \cup P_{excl\_cha} \cup P_{term\_uc} \quad (19)$$

5. Evaluation

In this section, the presented ASP encodings are compared with existing implementations for computing serialization sequences. This includes comparing the correctness and runtimes of computing certain example argumentation frameworks for each of the presented semantics (*ad*, *co*, *pr*, *gr*, *st*, *sa* and *uc*). So far, such encodings are only available in Java from the "Tweety-Project" by Thimm [23], which is also used here[12]. Regarding correctness, both solvers were compared to provide the same serialization sequences. For this purpose, the serialization sequences of various simple argumentation frameworks and the argumentation frameworks generated for the Subsection 5.2 were checked for consistency. Both solvers show the same serialization sequences for the argumentation frameworks tested in each of the semantics.

The further evaluation is carried out in terms of runtime, i.e. the runtimes for the computation of serialization sequences of particular argumentation frameworks using Java or ASP are compared. This is done with regard to the different semantics, the argument count of the argumentation framework and its density[13]. Furthermore, it is tested whether any individual properties of the argumentation frameworks affect runtime. Besides, the runtimes of the ASP solver are analysed with regard to the ratio of grounding and solving time to the total runtimes. Therefore, the following experiments are conducted with each of the semantics mentioned:

1. Runtime dependence of argument count: computing serialization sequences of argumentation frameworks with different argument counts and constant density.

2. Runtime dependence of density: computing serialization sequences of argumentation frameworks with different densities and constant argument count.

3. Standard deviation of runtimes: computing serialization sequences of various argumentation frameworks with constant argument count and constant density.

[12] All specifications on Java classes given here refer to this collection

[13] The density of a graph is defined as the ratio between the number of edges and the maximum number of edges.

4. Ratio of solving time: determine the ratio of solving time to the total runtimes of the ASP solver.

5.1. Experimental Setup

The ASP encodings presented in the previous section are tested with the ASP solver *clingo*, which has an integrated method for measuring runtime.

For Java, the “Tweety-Project” provides a reasoner class for each semantics to be tested (e.g. `SerializedAdmissibleReasoner` for *ad*). Short Java classes with the corresponding reasoner class have been implemented, which take the APX file of the sample argumentation framework as an argument. The runtime of the Java implementation was measured using the Java method `System.nanoTime()`. Although this method does not provide the exact CPU time, it should be sufficient for this work, as empirically the differences are less than one second. The code of the Java class used is shown in the Appendix in Listing B.1 for admissible sets as an example. The code for the other semantics was adapted by changing the reasoner. The corresponding Java class was then exported and used as a JAR file. To ensure comparability, all computations were performed on the same system (Fedora Linux 41, Workstation Edition, AMD Ryzen 7 3800X x 16, 32 GB RAM).

The sample argumentation frameworks were generated using the Tweety class `DungTheoryGenerator`, which provides APX files, each representing a randomly generated argumentation framework. This class allows customizing the argument count with the parameter “numberOfArguments”, the density of the argumentation framework with the parameter “attackPropability’, self-attack avoidance and enforcing a tree-shape of the argumentation framework. The last two parameters were left at the default setting since they can be neglected for the purpose of this work, i.e. self-attacks are avoided and tree-shape is not enforced. The code of the Java class used is shown in the Appendix in Listing B.2.

5.2. Experiment 1: Runtime Dependence of Argument Count

To test the runtime dependence with respect the the argument count, argumentation frameworks with 1 to 35 arguments and a constant density of 0.5 were generated. Additionally, argumentation frameworks with 50, 100 ... 500 arguments of equal density were generated. For each argument count four APX files were generated and the runtimes were averaged over these samples. To keep the effort within a reasonable range, the maximum runtime for each computation of a serialization sequence was set to 20 minutes (1200 s).

The results are visualized in Figures 7, 8, 9 and 10 and in more detail listed in the Tables 4, 5 and 6. As expected, the argument count turns out to be the most important parameter influencing the runtime. The experiments showed that the runtime of the Java solver increases sharply in the range of 16 to 23 arguments for all semantics before reaching the timeout. Regarding the runtime of the Java solver, the semantics can be divided into two groups: The semantics *ad*, *co*, *gr*, *st* and *sa*

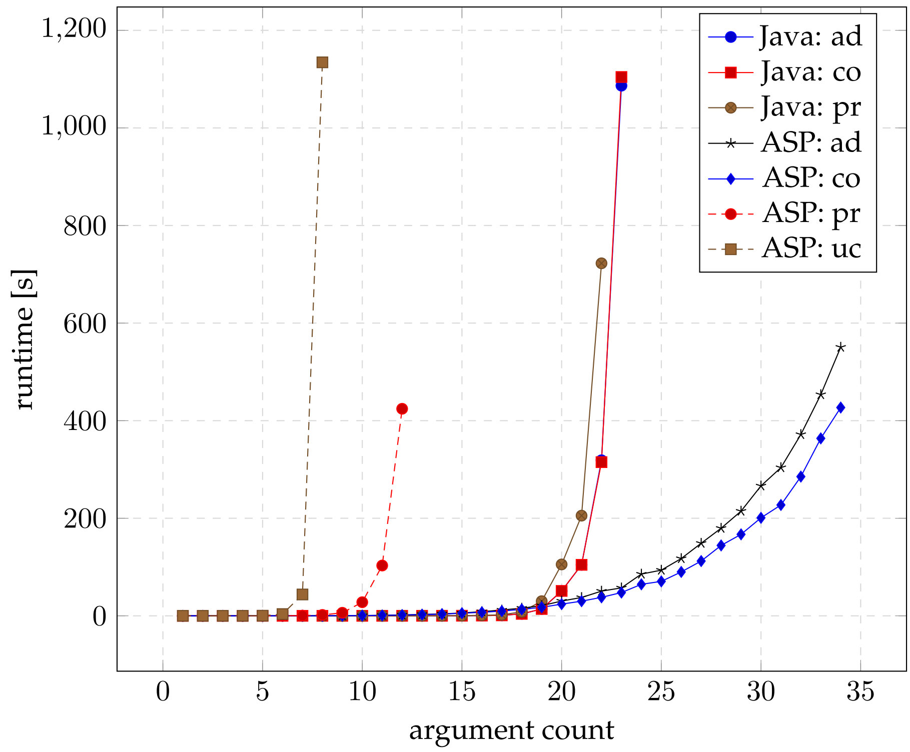


Figure 7: Dependence of runtimes from argument count for Java and ASP fro the semantics *ad*, *co* and *pr*.

have similar runtimes, while the runtimes for *pr* and *uc* almost double. Both groups exhibit a rather exponential behaviour (see Figure 9).

The runtimes of the ASP solver increase less with the argument count than the Java solver, except for *pr* and *uc*. The semantics *gr* and *sa* have the shortest ASP encoding and show a comparatively small increase compared to the Java solver, so that samples with up to 400 (*gr*) and 450 (*sa*) arguments can be solved. For *pr* and *uc* on the other hand, the process was already killed when the argumentation framework had more than 12 or 8 arguments, respectively. This is most likely due to the elaborate data structures required for processing these semantics. For the semantics *ad*, *co* and *st* argumentation frameworks with up to 34 arguments can be solved on the system used. It must be emphasized that the limiting factor here is not time, but memory. The ASP solver is killed by the operating system due to lack of memory when processing samples with more than 34 arguments. The memory consumption of the ASP solver is quite high, likely due to the memory required for grounding. For example, the memory consumption of the ASP solver for computing the serialization sequences for *ad* is about 20.2 GB for 31 arguments and increases almost linearly to 32.9 GB with 34 arguments.

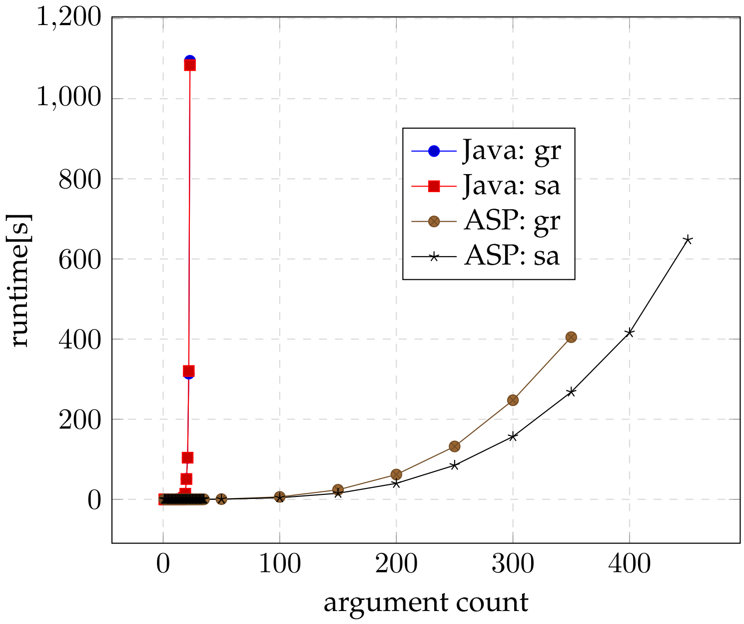


Figure 8: Dependence of runtimes from argument count for *gr* and *sa* (Java and ASP).

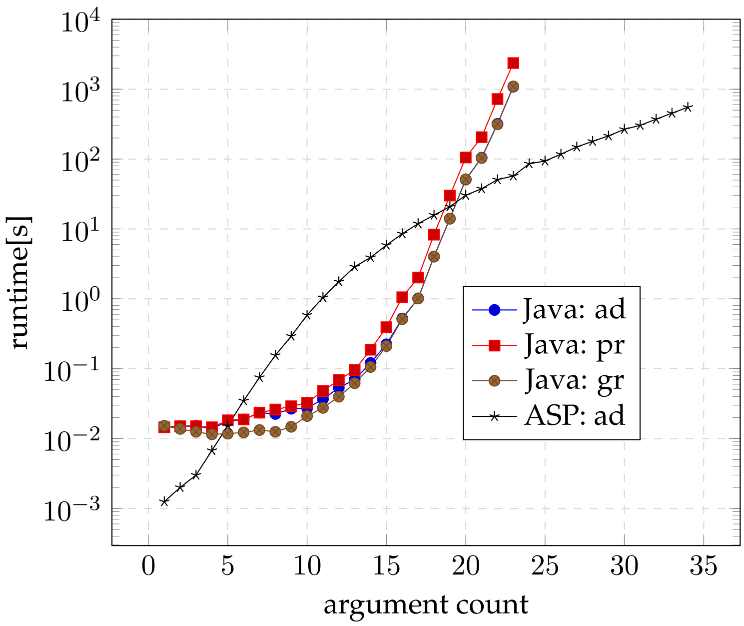


Figure 9: Logarithmic dependence of runtime from argument count for Java (*ad*, *pr*, *gr*) and ASP (*ad*).

<table>
<tr><th></th><th colspan="3">Java</th><th colspan="3">ASP</th></tr>
<tr><th>Arg Count</th><th>ad</th><th>co</th><th>st</th><th>ad</th><th>co</th><th>st</th></tr>
<tr><td>1</td><td>0.01</td><td>0.01</td><td>0.01</td><td>< 0.01</td><td>< 0.01</td><td>< 0.01</td></tr>
<tr><td>2</td><td>0.02</td><td>0.02</td><td>0.02</td><td>< 0.01</td><td>< 0.01</td><td>< 0.01</td></tr>
<tr><td>3</td><td>0.02</td><td>0.02</td><td>0.02</td><td>< 0.01</td><td>< 0.01</td><td>< 0.01</td></tr>
<tr><td>4</td><td>0.01</td><td>0.01</td><td>0.01</td><td>0.01</td><td>0.01</td><td>0.01</td></tr>
<tr><td>5</td><td>0.02</td><td>0.02</td><td>0.02</td><td>0.02</td><td>0.02</td><td>0.02</td></tr>
<tr><td>6</td><td>0.02</td><td>0.02</td><td>0.02</td><td>0.03</td><td>0.04</td><td>0.03</td></tr>
<tr><td>7</td><td>0.02</td><td>0.02</td><td>0.02</td><td>0.08</td><td>0.08</td><td>0.08</td></tr>
<tr><td>8</td><td>0.02</td><td>0.02</td><td>0.02</td><td>0.16</td><td>0.16</td><td>0.15</td></tr>
<tr><td>9</td><td>0.03</td><td>0.03</td><td>0.03</td><td>0.29</td><td>0.30</td><td>0.29</td></tr>
<tr><td>10</td><td>0.03</td><td>0.03</td><td>0.03</td><td>0.58</td><td>0.57</td><td>0.58</td></tr>
<tr><td>11</td><td>0.04</td><td>0.04</td><td>0.04</td><td>1.04</td><td>0.93</td><td>1.02</td></tr>
<tr><td>12</td><td>0.05</td><td>0.05</td><td>0.05</td><td>1.76</td><td>1.67</td><td>1.76</td></tr>
<tr><td>13</td><td>0.07</td><td>0.07</td><td>0.07</td><td>2.88</td><td>2.46</td><td>2.73</td></tr>
<tr><td>14</td><td>0.12</td><td>0.12</td><td>0.12</td><td>3.91</td><td>3.53</td><td>3.73</td></tr>
<tr><td>15</td><td>0.22</td><td>0.23</td><td>0.23</td><td>5.86</td><td>5.36</td><td>5.61</td></tr>
<tr><td>16</td><td>0.52</td><td>0.52</td><td>0.53</td><td>8.54</td><td>7.31</td><td>8.34</td></tr>
<tr><td>17</td><td>1.01</td><td>1.01</td><td>1.01</td><td>11.89</td><td>9.76</td><td>11.15</td></tr>
<tr><td>18</td><td>4.02</td><td>4.02</td><td>4.06</td><td>15.76</td><td>13.27</td><td>15.62</td></tr>
<tr><td>19</td><td>14.03</td><td>14.03</td><td>14.03</td><td>20.82</td><td>17.46</td><td>20.36</td></tr>
<tr><td>20</td><td>50.81</td><td>51.06</td><td>51.06</td><td>30.35</td><td>24.02</td><td>27.62</td></tr>
<tr><td>21</td><td>104.56</td><td>104.56</td><td>104.06</td><td>37.62</td><td>30.04</td><td>36.40</td></tr>
<tr><td>22</td><td>318.88</td><td>315.01</td><td>316.88</td><td>50.66</td><td>37.93</td><td>46.92</td></tr>
<tr><td>23</td><td>1086.63</td><td>1104.38</td><td>1085.38</td><td>57.31</td><td>47.70</td><td>57.57</td></tr>
<tr><td>24</td><td rowspan="12">timeout</td><td rowspan="12">timeout</td><td rowspan="12">timeout</td><td>85.59</td><td>64.60</td><td>77.90</td></tr>
<tr><td>25</td><td>93.64</td><td>70.63</td><td>93.65</td></tr>
<tr><td>26</td><td>117.43</td><td>90.00</td><td>117.99</td></tr>
<tr><td>27</td><td>148.77</td><td>112.27</td><td>152.23</td></tr>
<tr><td>28</td><td>179.54</td><td>144.29</td><td>172.54</td></tr>
<tr><td>29</td><td>214.58</td><td>167.02</td><td>216.53</td></tr>
<tr><td>30</td><td>266.34</td><td>200.88</td><td>237.53</td></tr>
<tr><td>31</td><td>303.79</td><td>227.23</td><td>272.97</td></tr>
<tr><td>32</td><td>371.68</td><td>285.37</td><td>386.68</td></tr>
<tr><td>33</td><td>453.45</td><td>363.72</td><td>462.77</td></tr>
<tr><td>34</td><td>550.62</td><td>426.92</td><td>559.08</td></tr>
<tr><td>35</td><td>killed</td><td>killed</td><td>killed</td></tr>
</table>

Table 4: Average runtimes (in seconds) for *ad*, *co* and *st* with different argument counts.

	Java		ASP	
Arg Count	pr	uc	pr	uc
1	0.01	0.02	0.01	0.01
2	0.02	0.01	0.02	0.01
3	0.02	0.02	0.02	0.01
4	0.01	0.01	0.02	0.03
5	0.02	0.01	0.03	0.25
6	0.02	0.01	0.12	3.63
7	0.02	0.01	0.49	43.81
8	0.03	0.02	1.83	1134.41
9	0.03	0.02	6.27	killed
10	0.03	0.08	27.93	
11	0.05	0.04	103.12	
12	0.07	0.06	424.37	
13	0.10	0.10	killed	
14	0.19	0.18		
15	0.39	0.39		
16	1.05	1.01		
17	2.02	2.01		
18	8.29	8.26		
19	30.05	29.55		
20	105.35	110.35		
21	205.35	207.85		
22	722.52	718.49		
23	timeout	timeout		

Table 5: Average runtimes (in seconds) for *pr* and *uc* with different argument counts.

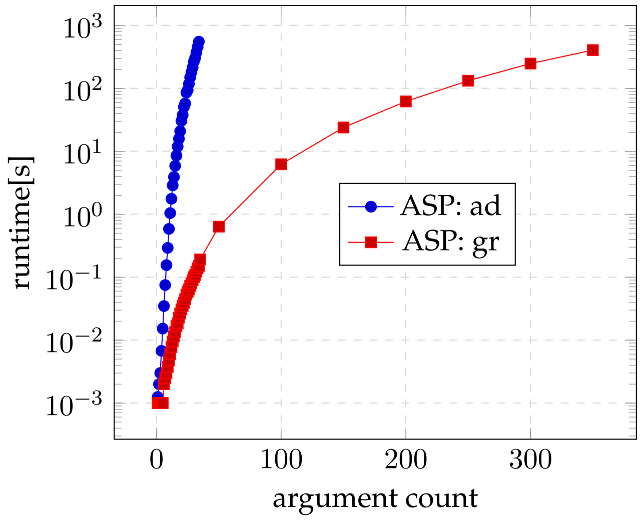


Figure 10: Logarithmic dependence of runtime from argument count for ASP (*ad* and *gr*).

	Java		ASP	
Arg Count	gr	sa	gr	sa
1	0.02	0.01	< 0.01	< 0.01
5	0.01	0.01	< 0.01	< 0.01
10	0.02	0.07	< 0.01	< 0.01
15	0.21	0.21	0.01	0.01
20	51.30	50.80	0.03	0.02
25	timeout	timeout	0.06	0.04
30			0.10	0.07
35			0.19	0.13
50			0.63	0.42
100			6.22	4.10
150			23.87	15.23
200			61.94	39.92
250			132.10	85.12
300			247.43	156.69
350			404.83	267.97
400			killed	416.04
450				647.95
500				killed

Table 6: Average runtimes (in seconds) for *gr* and *sa* with different argument counts.

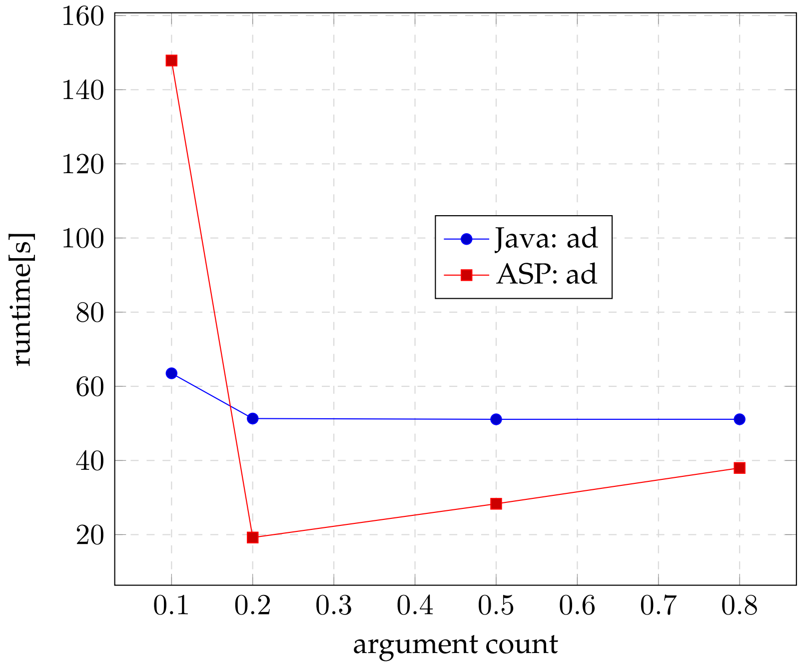


Figure 11: Dependence of runtime from density for Java and ASP (both *ad*).

5.3. Experiment 2: Runtime Dependence of Density

To test the runtime dependence on the density of the argumentation frameworks, 25 APX files were generated with densities of 0.1, 0.2, 0.5 and 0.8, respectively, and 20 arguments each. The runtimes with equal density were averaged. The results are shown in Figure 11 and in Table 7. The effect of density on the runtime is comparatively moderate. The runtime with the lowest density is longer for each semantics and for Java as well as for ASP. For Java, the runtime drops to a broadly similar level at higher densities, while for ASP, the runtime has a minimum at a density of 0.2 and increases again at higher densities.

5.4. Experiment 3: Standard Deviation of Runtimes

To determine whether individual properties of the argumentation frameworks (except for the argument count and the density) affect runtime, 50 different sample APX files were generated, each with 20 arguments and a density of 0.5. Table 8 shows the mean runtimes and the corresponding standard deviations. The runtimes were relatively constant within the same semantics, with a slightly higher standard deviation for the ASP solver. Due to the small standard deviation, it is reasonable to neglect the influence of individual properties of the argumentation frameworks on the runtime for both solvers, if averaged over a sufficient number of instances.

		Density			
		0.1	0.2	0.5	0.8
Java	ad	63.51	51.31	51.10	51.11
	co	63.57	51.21	51.11	51.10
	pr	145.24	112.97	104.45	114.85
	gr	63.48	51.30	51.10	51.18
	st	64.35	51.29	50.99	51.22
	sa	62.70	51.35	50.97	51.07
	uc	139.01	110.80	108.75	112.03
ASP	ad	147.87	19.22	28.32	37.98
	co	81.25	16.47	23.10	32.43
	pr	killed			
	gr	0.13	0.02	0.03	0.04
	st	80.33	18.27	27.47	38.77
	sa	0.24	0.01	0.02	0.03
	uc	killed			

Table 7: Average runtimes (in seconds) with different densities.

		ad	co	pr	gr	st	sa	uc
Java	average runtime	50.94	50.98	106.62	50.98	50.98	51.03	106.52
	standard deviation	0.31	0.14	7.57	0.32	0.32	0.12	7.63
ASP	average runtime	28.72	23.22	killed	0.03	27.51	0.02	killed
	standard deviation	2.88	1.25	-	< 0.01	2.49	< 0.01	-

Table 8: Average and standard deviations of runtimes (in seconds) of argumentation frameworks of equal size.

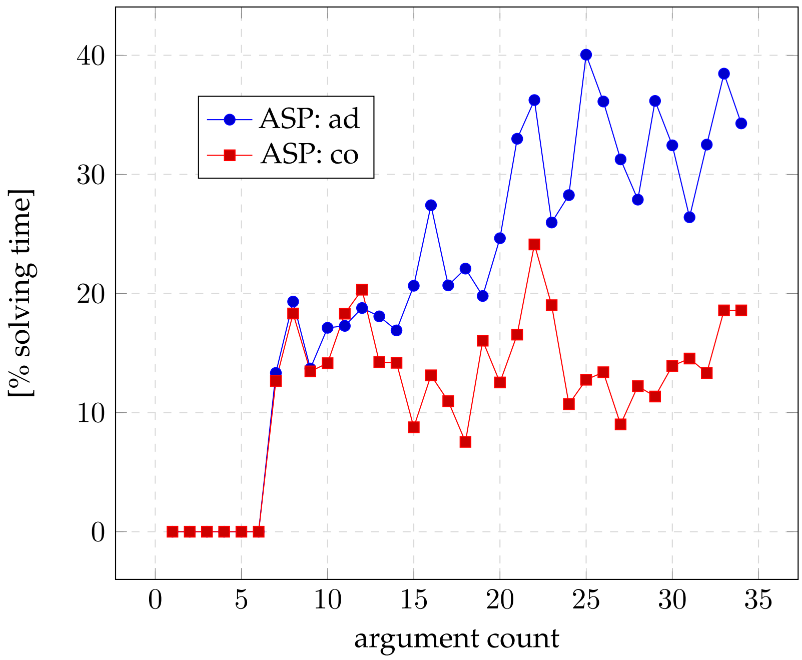


Figure 12: Percentage of solving time of ASP solver for *ad* and *co*.

5.5. Experiment 4: Ratio of Solving Time

As described in section 2.2.2 the ASP solver works in two consecutive steps, grounding and solving. The grounding generally requires a longer part of the total runtime. In addition to the total CPU time, the used ASP solver provides the time required to solve the grounded rules (then the grounding time is the difference of the total runtime and the solving time).

Figure 12 shows the percentage of the solving time for the semantics *ad* and *co* depending on the argument count. Apart from the fact that the solving time is higher for *ad* than for *co*, the data show a high variability and are therefore quite difficult to interpret.

6. Future Work

In order to place the results obtained here on a solid foundation, the correctness of the ASP encodings would have to be formally proven. So far, the correctness of the developed encodings has only been verified empirically on the argumentation frameworks from the evaluation.

Another goal for future work can focus on improving the runtime of the ASP solver, especially for the semantics *pr* and *uc*. The main runtime issue is the need to verify initial sets, which often requires to reason over all subsets of a given set. This can be inefficient, when an exponential number of instances has to be checked. To overcome this, the application of the saturation technique, as mentioned in Section 3.1.2, could possibly be a solution. This might be very difficult to implement, because the use of default negation is limited. The saturation technique would allow to apply a second "guess and check" to reason over subsets of a solution candidate. Another possibility to improve the runtime for computing the serialization sequences of *uc* could be to implement the *Bron-Kerbosch* algorithm in ASP, as mentioned in Section 3.3. This would allow to select conflict-free subsets of a given set of arguments, which then could be used as input for the Algorithm 1. It should be emphasized that it is not guaranteed, that these proposals will lead to an improvement of runtimes. A third suggestion to improve runtimes could be to use metaprogramming, which has been particularly recommended for the multiple use of the "guess and check" paradigm [20]. This would require an additional programming language like Python, that is able to handle different ASP encodings on a meta level.

7. Conclusion

Abstract argumentation frameworks are directed graphs used to represent human reasoning, where the nodes represent arguments and directed edges represent the refutation of one argument by another. Sets of arguments that represent a (coherent) point of view are called extensions that can identify the outcome(s) of a discussion represented by an argumentation framework. The abstract way to compute an extension is defined by the corresponding semantics, with a variety of different semantics available. The minimum property of any extension is to be conflict-free and admissible, i.e. that there are no conflicts within an extension and that every argument of an extension is defended against external attacks. To satisfy the human need to consider arguments in a sequential order, the concept of *serialization* was proposed, in which the desired extension is constructed step by step using subsets of arguments. These subsets are called *initial sets* and represent a single resolved local issue. The serialization is possible for admissible sets (*ad*) and for the semantics *co*, *pr*, *gr*, *sa*, *st* and *uc*.

In this thesis, ASP encodings to compute initial sets and their subtypes (unattacked, unchallenged and challenged initials sets) and for the serialization sequences of *ad*, *co*, *pr*, *gr*, *sa*, *st* and *uc* are presented and discussed. The advantages of ASP in being elaboration-tolerant and requiring comparatively short code could only be confirmed for the semantics *gr* and *sa*. In contrast, the computation of the other semantics requires additional auxiliary data structures that are rather difficult to encode and understand.

The encodings for serialization sequences are compared with an existing implementation in Java from the "Tweety-Project" in terms of correctness and runtime. For this purpose, various example argumentation frameworks were generated and tested with both solver types. In terms of correctness, both solvers yield equal results for all seven semantics. Regarding runtime, the argument count of the argumentation framework is the most important parameter. The Java solver showed an exponential behaviour for all semantics, so that argumentation frameworks with up to 22 arguments for *pr* and *uc* and up to 23 arguments for *ad*, *co*, *gr*, *sa* and *st* could be solved within the runtime limit of 20 min. The ASP solver showed an exponential behaviour for *pr* and *uc* and was killed due to lack of memory for argumentation frameworks with only 13 or 9 arguments, respectively. For the other semantics the ASP solver was faster than the corresponding Java solver. The high memory consumption of the ASP solver lead to an abort when processing argumentation frameworks for *ad*, *co* and *st* with more than 34 arguments. For *gr* and *sa* argumentation frameworks of up to 400 or 450 arguments, respectively, were solvable.

References

[1] P. Baroni, M. Caminada, and M. Giacomin. Abstract argumentation frameworks and their semantics. *Handbook of Formal Argumentation*, pages 159–236, 2018.

[2] P. Baroni and M. Giacomin. On principle-based evaluation of extension-based argumentation semantics. *Artificial Intelligence*, 171:675–700, 2007.

[3] G. Baumann, R. Brewka, and M. Ulbricht. Revisiting the foundations of abstract argumentation–semantics based on weak admissibility and weak defense. *Proceedings of the AAAI Conference on Artificial Intelligence*, (34):2742–2749, 2020.

[4] C. Beierle and G. Kern-Isberner. *Methoden wissensbasierter Systeme*. 6th edition, 2019.

[5] L. Bengel, J. Sander, and M. Thimm. Characterising Serialisation Equivalence for Abstract Argumentation. *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI'24)*, 2024.

[6] L. Bengel and M. Thimm. Serialisable Semantics for Abstract Argumentation. *Computational Models of Argumentation: Proceedings of COMMA*, pages 80–91, 2022.

[7] L. Blümel and M. Thimm. A Ranking Semantics for Abstract Argumentation Based on Serialisability. *Computational Models of Argument*, pages 104–115, 2022.

[8] E. Bonzon, J. Delobelle, S. Konieczny, and N. Maudet. A Comparative Study of Ranking-based Semantics for Abstract Argumentation. *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI'16)*, pages 914–920, 2016.

[9] M. Caminada. *Strong Admissibility revisited*. 2014.

[10] M. Caminada and D. M. Gabbay. A Logical Account of Formal Argumentation. *Studio Logica: An International Journal for Symbolic Logic*, 93(2/3):109–145, 2009.

[11] P. Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artificial Intelligence*, 77(2):321–358, 1995.

[12] U. Egly, S. A. Gaggl, and S. Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation 1.2*, pages 147–177, 2010.

[13] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. *Declarative Problem-Solving Using the DLV System*, pages 79–103. Springer US, Boston, MA, 2000.

[14] T. Eiter and A. Polleres. Toward automated integration of guess and check programs in Answer Set Programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1–2):23–60, 2006.

[15] D. Eppstein, M. Löffler, and D. Strash. Listing All Maximal Cliques in Large Sparase Real-World Graphs. *Journal of Experimental Algorithmics*, 18:3.1–3.21, 2013.

[16] W. Faber. *An Introduction to Answer Set Programming and Some of Its Extensions*, pages 149–185. Springer, 2020.

[17] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, and P. Wanko. *Potassco User Guide*. 2019.

[18] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practise*. Springer, 2022.

[19] A. Hunter and M. Thimm. Probabilistic Reasoning with Abstract Argumentation Frameworks. *Journal of Artificial Intelligence Research*, 59:565–611, 2017.

[20] R. Kaminsky, J. Romero, T. Schaub, and P. Wanko. *How to Build Your Own ASP-based System?!*, pages 299–361. Number 23. 2023.

[21] V. Lifschitz. *Answer Set Programming*. Springer, 2019.

[22] C. Redl. Answer Set Programs with Queries over Subprograms. In M. Balduccini and T. Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning*, Lecture Notes in Computer Science, 2017.

[23] M. Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2014.

[24] M. Thimm. Revisiting initial sets in abstract argumentation. *Argument & Computation*, 13(3):325–360, 2022.

[25] V. Turau and C. Weyer. *Algorithmische Graphentheorie*. de Gruyter, 4. edition, 2015.

[26] I. Wegener. *Komplexitätstheorie*. Springer, 2003.

[27] Y. Xu and C. Cayrol. Initial Sets in Abstract Argumentation Frameworks. *Proceedings of the 1st Chinese Conference on Logic and Argumentation (CLAR'16)*, 2016.

A. Complete ASP Encodings

The following encodings are also available online at https://github.com/ukarkmann/ASP-encoding-for-serialization-sequences.

A.1. Initial Sets

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for initial sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sets of arguments as solution candidates
%
% 1.    Exclude non-initial solution candidates
%
%       1.1     Exclude empty set
%       1.2     Exclude conflicting sets
%       1.3     Exclude non-admissible sets
%               => Remaining sets are non-empty admissible
%       1.4     Exclude non-minimal admissible sets
%
%               1.4.1   Define subsets decremented by one element
%               1.4.2   Define subsubsets by removing all non-defended arguments
%               1.4.3   Flag non-admissible subsets
%               => Non-flagged subsets are admissible
%               1.4.4   Exclude solution candidates with admissible subset
%
%       => Remaining sets are initial sets
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% attacked/1    attacked argument
% card/1        cardinality of solution candidate
% excl/2        argument excluded from set
% in/1          argument of solution candidate
% lt/2          lower-than relation over arguments of solution candidate
% ninf/1        non-smallest arguments of solution candidate
% non_adm/2     indicates non-admissibility of subset
% non_def/3     non-defended argument of subset of solution candidate
% non_empty     indicates non-emptyness of solution candidate
% nsucc/2       non-successor relation over arguments of solution candidate
% sub/2         argument of decremented set
% sub/3         argument of descending subsets
% sub_attacked/3 argument attacked by subset
% succ/2        successor-relation over arguments of solution candidate

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       0.      Generate sets of arguments as candidates for initial sets

        { in(X) }               :-      arg(X).

%       1.1     Exclude empty set
```

```
        non_empty           :-      in(X).

                            :-      not non_empty.

%       1.2     Exclude conflicting sets

                            :-      in(X),
                                    in(Y),
                                    att(X, Y).

%       1.3     Exclude non-admissible sets

%       Select arguments attacked by 'in'

        attacked(X)         :-      in(Y),
                                    att(Y, X).

%       Exclude sets with non-defended arguments

                            :-      att(Y, X),
                                    in(X),
                                    not attacked(Y).

%       RESULT: Remaining sets are non-empty admissible

%       1.4     Exclude non-minimal admissible sets

%       Define an order on each set with succ-relation

        lt(X, Y)            :-      in(X),
                                    in(Y),
                                    X<Y.

        nsucc(X, Z)         :-      lt(X, Y),
                                    lt(Y, Z).

        succ(X, Y)          :-      lt(X, Y),
                                    not nsucc(X, Y).

        ninf(X)             :-      lt(Y, X).

%       Define numbered arguments to be excluded

        excl(X, 1)          :-      not ninf(X),
                                    in(X).

        excl(Y, No+1)       :-      excl(X, No),
                                    in(Y),
                                    succ(X, Y).

%       1.4.1   Define subsets decremented by one element

        sub(X, No)          :-      in(X),
                                    not excl(X, No),
                                    card(C),
                                    No = 1..C.

%       Define first 'level' of subsets

        sub(X, No, 0)       :-      sub(X,No).
```

```
%       Select arguments attacked by 'sub'

        sub_attacked(Y, No, Level):-    sub(X, No, Level),
                                        att(X, Y).

%       Select non-defendet arguments of 'sub'

        non_def(Y, No, Level)   :-      sub(Y, No, Level),
                                        att(X, Y),
                                        not sub_attacked(X, No, Level).

%       1.4.2   Define subsubsets by removing non-defended arguments

        card(C)                 :-      { in(X) } == C.

        sub(X, No, Level+1)     :-      sub(X, No, Level),
                                        not non_def(X, No, Level),
                                        card(C),
                                        Level < C.

%       1.4.3   Flag non-admissible subsets

        non_adm(No, Level)      :-      non_def(Y, No, Level).


%       1.4.4   Exclude solution candidate with admissible subset

                                :-      not non_adm(No, Level),
                                        sub(X, No, Level).

#show in/1.
```

A.2. Unattacked Initial Sets

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for unattacked initial sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Generate sets of arguments as solution candidates

        { in(X) }               :-      arg(X).


%       Exclude solution candidates with cardinality not 1

                                :-      { in(X) } != 1.

%       Exclude solution candidates with attacked arguments

                                :-      in(X),
                                        att(Y,X),
                                        arg(Y).

#show in/1.
```

A.3. Unchallenged Initial Sets

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for unchallenged initial sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Algorithm
%
% 0.    Generate sets of arguments as solution candidates
%
% 1.    Each solution must be an initial set
%
%       1.1     Exclude empty set
%       1.2     Exclude conflicting sets
%       1.3     Exclude non-admissible sets
%               => Remaining sets are non-empty admissible
%       1.4     Exclude non-minimal admissible sets
%
%               1.4.1   Define subsets decremented by one element
%               1.4.2   Define subsubsets by removing non-defended arguments
%               1.4.3   Flag non-admissible subsets
%               => Non-flagged subsets are admissible
%               1.4.4   Exclude solution candidates with admissible subset
%
%       => Remaining sets are initial sets
%
% 2.    Exclude unattacked sets
%
%       => Remaining sets are attacked initial sets
%
% 3.    Exclude solution candidates attacked by initial sets
%
%       3.1     Define all non-empty subsets of arguments
%       3.2     Flag non-initial subsets
%               3.2.1   Flag conflicting subsets
%               3.2.2   Flag non-admissible subsets
%               3.2.3   Flag non-minimal subsets
%               => Non-flagged subsets are initial sets
%       3.3     Exclude solution candidates attacked by initial set
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List of predicates
%
% a_attacked/2 attacked argument
% a_card/1     cardinality of set of arguments
% a_elem/2     element of subset of arguments
% a_flag/1     flag non-initial subsets
% a_lt/2       lower-than relation over arguments
% a_nsucc/2    non-successor relation over arguments
% a_ninf/1     non-smallest arguments
% a_set/1      number of subset
% a_sub/2      subsets of arguments
% a_succ/2     non-successor relation over arguments
% a_vec/4      binary vector
% arg/1        arguments of AF
% arg/2        numbered argument of AF
% att/2        attack-relation
% attacked/1   attacked argument of solution candidate
% card/1       cardinality of solution candidate
% elemIni/2    element of initial set
```

```
% excl/2          argument excluded from solution candidate
% in/1            argument of solution candidate
% in_attacked     attacked solution candidate
% iniSet/1        initial set
% lt/2            lower-than relation over arguments of solution candidate
% ninf/1          non-smallest arguments of solution candidate
% non_adm/2       indicates non-admissibility of subset
% non_def/3       non-defended argument of subset of solution candidate
% non_empty       non-empty solution candidate
% nsucc/2         non-successor relation over arguments of solution candidate
% sub/2           argument of decremented solution candidate
% sub/3           argument of descending subsets
% sub_attacked/3  argument attacked by subset
% succ/2          successor-relation over arguments of solution candidate

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       0.      Generate sets of arguments as solution candidates

        { in(X) }               :-      arg(X).

%       1.      Each solution must be an initial set
%       1.1     Exclude empty set

        non_empty               :-      in(X).

                                :-      not non_empty.

%       1.2     Exclude conflicting sets

                                :-      in(X),
                                        in(Y),
                                        att(X, Y).

%       1.3     Exclude non-admissible sets

%       Select arguments attacked by 'in'

        attacked(X)             :-      in(Y),
                                        att(Y, X).

%       Exclude sets with non-defended arguments

                                :-      att(Y, X),
                                        in(X),
                                        not attacked(Y).

%       RESULT: Remaining sets are non-empty admissible

%       1.4     Exclude non-minimal admissible sets

%       Define an order on each set with succ-relation

        lt(X, Y)                :-      in(X),
                                        in(Y),
                                        X<Y.

        nsucc(X, Z)             :-      lt(X, Y),
                                        lt(Y, Z).

        succ(X, Y)              :-      lt(X, Y),
                                        not nsucc(X, Y).
```

```
ninf(X)                 :-      lt(Y, X).

%       Define numbered arguments to be excluded

excl(X, 1)              :-      not ninf(X),
                                in(X).

excl(Y, No+1)           :-      excl(X, No),
                                in(Y),
                                succ(X, Y).

%       1.4.1   Define subsets decremented by one element

sub(X, No)              :-      in(X),
                                not excl(X, No),
                                card(C),
                                No = 1..C.

%       Define first 'level' of subsets

sub(X, No, 0)           :-      sub(X,No).

%       Select arguments attacked by 'sub'

sub_attacked(Y, No, Level):-    sub(X, No, Level),
                                att(X, Y).

%       Select non-defendet arguments of 'sub'

non_def(Y, No, Level)   :-      sub(Y, No, Level),
                                att(X, Y),
                                not sub_attacked(X, No, Level).

%       1.4.2   Define subsubsets by removing non-defended arguments

card(C)                 :-      { in(X) } == C.

sub(X, No, Level+1)     :-      sub(X, No, Level),
                                not non_def(X, No, Level),
                                card(C),
                                Level < C.

%       1.4.3   Flag non-admissible subsets

non_adm(No, Level)      :-      non_def(Y, No, Level).


%       1.4.4   Exclude solution candidate with admissible subset

                        :-      not non_adm(No, Level),
                                sub(X, No, Level).

%       RESULT: Remaining sets are initial sets

%       2.      Exclude unattacked sets

in_attacked             :-      in(X),
                                att(Y,X),
                                arg(Y).

                        :-      not in_attacked.
```

```
%       RESULT: Remaining sets are attacked initial sets

%       3.      Exclude solution candidates attacked by initial sets

%       3.1     Define all non-empty subsets

%       Define an order over all arguments with succ-relation

        a_lt(X,Y)               :-      arg(X),
                                        arg(Y), X<Y.

        a_nsucc(X,Z)            :-      a_lt(X,Y),
                                        a_lt(Y,Z).

        a_succ(X,Y)             :-      a_lt(X,Y),
                                        not a_nsucc(X,Y).

        a_ninf(X)               :-      a_lt(Y,X).

%       Each argument is numbered accordingly

        arg(X, 0)               :-      not a_ninf(X),
                                        arg(X).

        arg(Y, ArgNo+1)         :-      arg(X, ArgNo),
                                        arg(Y),
                                        a_succ(X, Y).

%       Define numbered subsets and use binary vector of subset-number
%       to assign arguments to the corresponding subset

%       Number of subsets equals cardinality of power-set
%       = 2^ (cardinality of set), "0" corresponds to empty set,
%       maximum corresponds to identity

        a_card(C)               :-      { arg(X) } == C.

        a_set(1..SetNo)         :-      (2 ** C) - 1 == SetNo,
                                        a_card(C).

%       Calculate binary vector by repeatedly divide number by 2.
%       Rest is 1 or 0 and assigns argument to subset
%       Result is needed for the next division

%       Start

        a_vec(SetNo, 0, SetNo\2, SetNo/2) :-
                                        a_set(SetNo).

%       Next

        a_vec(SetNo, ArgNo+1, Result\2, Result/2) :-
                                        a_vec(SetNo, ArgNo, _, Result),
                                        SetNo >= (2 ** (ArgNo+1)).

%       Define elements of subsets

        a_elem(SetNo, ArgNo)    :-      a_vec(SetNo, ArgNo, Rest, _),
                                        Rest = 1.

%       Define subsets of out-subset (w/o empty set and identity)
```

```

    a_sub(SetNo, SubSet)     :-      a_vec(SetNo, ArgNo, Rest, Result),
                                     Rest = 1,
                                     SubSet = 2 ** (ArgNo).

    a_sub(SetNo, SubA+SubB)  :-      a_sub(SetNo, SubA),
                                     a_sub(SetNo, SubB),
                                     SubA != SubB,
                                     SubA + SubB < SetNo.

%   3.2     Flag non-initial subsets

%   3.2.1   Flag conflicting subsets

    a_flag(SetNo)            :-      a_elem(SetNo, ArgNo1),
                                     a_elem(SetNo, ArgNo2),
                                     arg(X, ArgNo2),
                                     arg(Y, ArgNo1),
                                     att(X, Y).

%   3.2.2   Flag non-admissible subsets

    a_attacked(SetNo, X)     :-      a_elem(SetNo, ArgNo),
                                     arg(Y, ArgNo),
                                     att(Y, X).

    a_flag(SetNo)            :-      a_elem(SetNo, ArgNo),
                                     arg(X, ArgNo),
                                     att(Y,X),
                                     not a_attacked(SetNo, Y).

%   3.2.3   Flag non-minimal subsets

    a_flag(SetNo1)           :-      a_set(SetNo1),
                                     a_set(SetNo2),
                                     SetNo1 != SetNo2,
                                     a_sub(SetNo1, SetNo2),
                                     not a_flag(SetNo2).

%   RESULT: Non-flagged subsets are initial sets

%   3.3     Exclude solution candidates attacked by initial set

    iniSet(SetNo)            :-      a_set(SetNo),
                                     not a_flag(SetNo).

    elemIni(SetNo, X)        :-      iniSet(SetNo),
                                     a_elem(SetNo, ArgNo),
                                     arg(X, ArgNo).

                             :-      elemIni(SetNo, X),
                                     in(Y),
                                     att(X,Y).

#show in/1.
```

A.4. Challenged Initial Sets

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for challenged initial sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Algorithm
%
% 0.    Generate sets of arguments as solution candidates
%
% 1.    Each solution must be an initial set
%
%       1.1     Exclude empty set
%       1.2     Exclude conflicting sets
%       1.3     Exclude non-admissible sets
%               => Remaining sets are non-empty admissible
%       1.4     Exclude non-minimal admissible sets
%
%               1.4.1   Define subsets decremented by one element
%               1.4.2   Define subsubsets by removing non-defended arguments
%               1.4.3   Flag non-admissible subsets
%               => Non-flagged subsets are admissible
%               1.4.4   Exclude solution candidates with admissible subset
%
%       => Remaining sets are initial sets
%
% 2.    Exclude unattacked sets
%
%       => Remaining sets are attacked initial sets
%
% 3.    Exclude solution candidates not attacked by initial set
%
%       3.1     Define all non-empty subsets of arguments
%       3.2     Flag non-initial subsets
%               3.2.1   Flag conflicting subsets
%               3.2.2   Flag non-admissible subsets
%               3.2.3   Flag non-minimal subsets
%               => Non-flagged subsets are initial sets
%       3.3     Exclude solution candidates not attacked by initial set
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List of predicates
%
% a_attacked/2 attacked argument
% a_card/1     cardinality of set of arguments
% a_elem/2     element of subset of arguments
% a_flag/1     flag non-initial subsets
% a_lt/2       lower-than relation over arguments
% a_nsucc/2    non-successor relation over arguments
% a_ninf/1     non-smallest arguments
% a_set/1      number of subset
% a_sub/2      subsets of arguments
% a_succ/2     non-successor relation over arguments
% a_vec/4      binary vector
% arg/1        arguments of AF
% arg/2        numbered argument of AF
% att/2        attack-relation
% attacked/1   attacked argument of solution candidate
% card/1       cardinality of solution candidate
% elemIni/2    element of initial set
```

```
% excl/2         argument excluded from solution candidate
% in/1           argument of solution candidate
% in_attacked    attacked solution candidate
% ini_attack     indicates solution candidates attacked by initial set
% iniSet/1       initial set
% lt/2           lower-than relation over arguments of solution candidate
% ninf/1         non-smallest arguments of solution candidate
% non_adm/2      indicates non-admissibility of subset
% non_def/3      non-defended argument of subset of solution candidate
% non_empty      non-empty solution candidate
% nsucc/2        non-successor relation over arguments of solution candidate
% sub/2          argument of decremented solution candidate
% sub/3          argument of descending subsets
% sub_attacked/3 argument attacked by subset
% succ/2         successor-relation over arguments of solution candidate

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       0.      Generate sets of arguments as solution candidates

        { in(X) }               :-      arg(X).

%       1.      Each solution must be an initial set
%       1.1     Exclude empty set

        non_empty               :-      in(X).

                                :-      not non_empty.

%       1.2     Exclude conflicting sets

                                :-      in(X),
                                        in(Y),
                                        att(X, Y).

%       1.3     Exclude non-admissible sets

%       Select arguments attacked by 'in'

        attacked(X)             :-      in(Y),
                                        att(Y, X).

%       Exclude sets with non-defended arguments

                                :-      att(Y, X),
                                        in(X),
                                        not attacked(Y).

%       RESULT: Remaining sets are non-empty admissible

%       1.4     Exclude non-minimal admissible sets

%       Define an order on each set with succ-relation

        lt(X, Y)                :-      in(X),
                                        in(Y),
                                        X<Y.

        nsucc(X, Z)             :-      lt(X, Y),
                                        lt(Y, Z).

        succ(X, Y)              :-      lt(X, Y),
```

```
                                        not nsucc(X, Y).

        ninf(X)                 :-      lt(Y, X).

%       Define numbered arguments to be excluded

        excl(X, 1)              :-      not ninf(X),
                                        in(X).

        excl(Y, No+1)           :-      excl(X, No),
                                        in(Y),
                                        succ(X, Y).

%       1.4.1   Define subsets decremented by one element

        sub(X, No)              :-      in(X),
                                        not excl(X, No),
                                        card(C),
                                        No = 1..C.

%       Define first 'level' of subsets

        sub(X, No, 0)           :-      sub(X,No).

%       Select arguments attacked by 'sub'

        sub_attacked(Y, No, Level):-    sub(X, No, Level),
                                        att(X, Y).

%       Select non-defendet arguments of 'sub'

        non_def(Y, No, Level)   :-      sub(Y, No, Level),
                                        att(X, Y),
                                        not sub_attacked(X, No, Level).

%       1.4.2   Define subsubsets by removing non-defended arguments

        card(C)                 :-      { in(X) } == C.

        sub(X, No, Level+1)     :-      sub(X, No, Level),
                                        not non_def(X, No, Level),
                                        card(C),
                                        Level < C.

%       1.4.3   Flag non-admissible subsets

        non_adm(No, Level)      :-      non_def(Y, No, Level).


%       1.4.4   Exclude solution candidate with admissible subset

                                :-      not non_adm(No, Level),
                                        sub(X, No, Level).

%       RESULT: Remaining sets are initial sets

%       2.      Exclude unattacked sets

        in_attacked             :-      in(X),
                                        att(Y,X),
                                        arg(Y).
```

```
                          :-      not in_attacked.

%   RESULT: Remaining sets are attacked initial sets

%   3.      Exclude solution candidates attacked by initial sets

%   3.1     Define all non-empty subsets

%   Define an order over all arguments with succ-relation

    a_lt(X,Y)             :-      arg(X),
                                  arg(Y), X<Y.

    a_nsucc(X,Z)          :-      a_lt(X,Y),
                                  a_lt(Y,Z).

    a_succ(X,Y)           :-      a_lt(X,Y),
                                  not a_nsucc(X,Y).

    a_ninf(X)             :-      a_lt(Y,X).

%   Each argument is numbered accordingly

    arg(X, 0)             :-      not a_ninf(X),
                                  arg(X).

    arg(Y, ArgNo+1)       :-      arg(X, ArgNo),
                                  arg(Y),
                                  a_succ(X, Y).

%   Define numbered subsets and use binary vector of subset-number
%   to assign arguments to the corresponding subset

%   Number of subsets equals cardinality of power-set
%   = 2^ (cardinality of set), "0" corresponds to empty set,
%   maximum corresponds to identity

    a_card(C)             :-      { arg(X) } == C.

    a_set(1..SetNo)       :-      (2 ** C) - 1 == SetNo,
                                  a_card(C).

%   Calculate binary vector by repeatedly divide number by 2.
%   Rest is 1 or 0 and assigns argument to subset
%   Result is needed for the next division

%   Start

    a_vec(SetNo, 0, SetNo\2, SetNo/2) :-
                                  a_set(SetNo).

%   Next

    a_vec(SetNo, ArgNo+1, Result\2, Result/2) :-
                                  a_vec(SetNo, ArgNo, _, Result),
                                  SetNo >= (2 ** (ArgNo+1)).

%   Define elements of subsets

    a_elem(SetNo, ArgNo)  :-      a_vec(SetNo, ArgNo, Rest, _),
                                  Rest = 1.
```

```
%       Define subsets of out-subset (w/o empty set and identity)

        a_sub(SetNo, SubSet)    :-      a_vec(SetNo, ArgNo, Rest, Result),
                                        Rest = 1,
                                        SubSet = 2 ** (ArgNo).

        a_sub(SetNo, SubA+SubB) :-      a_sub(SetNo, SubA),
                                        a_sub(SetNo, SubB),
                                        SubA != SubB,
                                        SubA + SubB < SetNo.

%       3.2     Flag non-initial subsets

%       3.2.1   Flag conflicting subsets

        a_flag(SetNo)           :-      a_elem(SetNo, ArgNo1),
                                        a_elem(SetNo, ArgNo2),
                                        arg(X, ArgNo2),
                                        arg(Y, ArgNo1),
                                        att(X, Y).

%       3.2.2   Flag non-admissible subsets

        a_attacked(SetNo, X)    :-      a_elem(SetNo, ArgNo),
                                        arg(Y, ArgNo),
                                        att(Y, X).

        a_flag(SetNo)           :-      a_elem(SetNo, ArgNo),
                                        arg(X, ArgNo),
                                        att(Y,X),
                                        not a_attacked(SetNo, Y).

%       3.2.3   Flag non-minimal subsets

        a_flag(SetNo1)          :-      a_set(SetNo1),
                                        a_set(SetNo2),
                                        SetNo1 != SetNo2,
                                        a_sub(SetNo1, SetNo2),
                                        not a_flag(SetNo2).

%       RESULT: Non-flagged subsets are initial sets

%       3.3     Exclude solution candidates not attacked by initial set

        iniSet(SetNo)           :-      a_set(SetNo),
                                        not a_flag(SetNo).

        elemIni(SetNo, X)       :-      iniSet(SetNo),
                                        a_elem(SetNo, ArgNo),
                                        arg(X, ArgNo).

        ini_attack              :-      elemIni(SetNo, X),
                                        in(Y),
                                        att(X,Y).

                                :-      not ini_attack.

#show in/1.
```

A.5. Serialization Sequence for Admissible Sets

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for serialization sequence of admissible sets
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences.
%
% 1.    Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial terms
%
%               1.1.1   Exclude sequences with 'intermediate' empty term
%               1.1.2   Exclude sequences with conflicting term
%               1.1.3   Exclude sequences with non-admissible term
%                       => Remaining sequences only have non-empty admissible
%                          terms
%               1.1.4   Exclude sequences with non-minimal admissible terms
%
%                       1.1.4.1 Create subsets decremented by one element
%                       1.1.4.2 Define subsubsets by removing non-defended arguments
%                       1.1.4.3 Flag non-admissible subsets
%                       => Non-flagged subsets are admissible
%                       1.1.4.4 Exclude sequences with admissible subset
%
%               => Remaining sequence terms are initial sets
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% att/3         attack-relation within reduct
% attacked/2    argument attacked by sequence term
% collect/2     argument outside reduct
% excl/3        argument excluded from term
% in/2          argument of sequence term
% index/1       index of sequence term
% in_card/2     cardinality of sequence term
% in_index/2    index of arguments of sequence term
% lt/3          lower-than relation over arguments of sequence term
% ninf/2        non-smallest arguments of sequence term
% non_adm/3     non-admissibility of numbered subset of sequence term
% non_def/4     non-defended arguments of subset of sequence term
% non_empty/1   non-empty sequence term
% nsucc/3       non-successor relation over arguments of sequence term
% reduct/2      argument of reduct
% sub/3         argument of decremented term
% sub/4         argument of subset of sequence term
% sub_attacked/4 argument attacked by subset of sequence term
% succ/3        successor-relation over arguments of sequence term

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments

        index(1..C)             :-      { arg(X) } == C.
```

```
%       0.      GENERATE sequences of sets of arguments as solution candidates
%               for serialization sequences

        { in(X, Step) }         :-      reduct(X, Step).

%       Get cardinality of sequence terms

        in_index(1..C, Step)    :-      { in(X, Step) } == C,
                                        index(Step).

        in_card(C, Step)        :-      { in(X, Step) } == C,
                                        index(Step).

%       Define reduct

%       First reduct equals AF

        reduct(X, 1)            :-      arg(X).

%       Collect arguments from sequence term

        collect(X, Step)        :-      in(X, Step).

%       Collect arguments attacked by sequence term

        collect(X, Step)        :-      in(Y, Step),
                                        att(Y, X).

%       Next reduct has all non-collected arguments

        reduct(X, Step+1)       :-      reduct(X, Step),
                                        not collect(X, Step),
                                        index(Step).

%        .. and the relations between contained arguments

        att(X, Y, Step)         :-      reduct(X, Step),
                                        reduct(Y, Step),
                                        att(X,Y).

%       1.      Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial term
%
%       1.1.1   Exclude sequences with 'intermediate' empty term

        non_empty(Step)         :-      in(X, Step).

                                :-      not non_empty(Step),
                                        non_empty(Step+1),
                                        index(Step).

%       1.1.2   Exclude sequences with conflicting terms

                                :-      in(X, Step),
                                        in(Y, Step),
                                        att(X, Y).

%       1.1.3   Exclude sequences with non-admissible term

%       Select arguments attacked by term
```

```
        attacked(X, Step)       :-      in(Y, Step),
                                        att(Y, X, Step).

%       Exclude sequences with non-defended arguments in term

                                :-      att(Y, X, Step),
                                        in(X, Step),
                                        not attacked(Y, Step).

%       RESULT: Remaining sequences only have non-empty admissible terms

%       1.1.4   Exclude sequences with non-minimal admissible term

%       1.1.4.1 Create subsets decremented by one element

%       Define an order over 'in' with succ-relation

        lt(X, Y, Step)          :-      in(X, Step),
                                        in(Y, Step),
                                        X<Y.

        nsucc(X, Z, Step)       :-      lt(X, Y, Step),
                                        lt(Y, Z, Step).

        succ(X, Y, Step)        :-      lt(X, Y, Step),
                                        not nsucc(X, Y, Step).

        ninf(X, Step)           :-      lt(Y, X, Step).

%       Define numbered arguments to be excluded

        excl(X, 1, Step)        :-      not ninf(X, Step),
                                        in(X, Step).

        excl(Y, No+1, Step)     :-      excl(X, No, Step),
                                        in(Y, Step),
                                        succ(X, Y, Step).

%       Define decremented sets (w/o excluded argument)

        sub(X, No, Step)        :-      in(X, Step),
                                        not excl(X, No, Step),
                                        in_index(No, Step).

%       Define first 'level' of subsets

        sub(X, No, Step, 0)     :-      sub(X, No, Step).

%       Select arguments attacked by 'sub'

        sub_attacked(Y, No, Step, Level):-
                                        sub(X, No, Step, Level),
                                        att(X, Y, Step).

%       Select non-defended arguments of 'sub'

        non_def(Y, No, Step, Level):-   sub(Y, No, Step, Level),
                                        att(X, Y, Step),
                                        not sub_attacked(X, No, Step, Level).

%       1.1.4.2 Define subsubsets by removing all non-defended arguments
```

```

        sub(X, No, Step, Level+1):-     sub(X, No, Step, Level),
                                        not non_def(X, No, Step, Level),
                                        in_card(C, Step),
                                        Level < C.

%       1.1.4.3 Flag all non-admissible subsets

        non_adm(No, Step, Level):-      non_def(Y, No, Step, Level).


%       1.1.4.4 Exclude sequences with admissible subset

                                :-      not non_adm(No, Step, Level),
                                        sub(X, No, Step, Level).

#show in/2.
```

A.6. Serialization Sequence for Complete Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for serialization sequence of complete semantics.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences.
%
% 1.    Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial terms
%
%               1.1.1   Exclude sequences with 'intermediate' empty term
%               1.1.2   Exclude sequences with conflicting term
%               1.1.3   Exclude sequences with non-admissible term
%                       => Remaining sequences only have non-empty admissible
%                          terms
%               1.1.4   Exclude sequences with non-minimal admissible terms
%
%                       1.1.4.1 Create subsets decremented by one element
%                       1.1.4.2 Define subsubsets by removing non-defended arguments
%                       1.1.4.3 Flag non-admissible subsets
%                       => Non-flagged subsets are admissible
%                       1.1.4.4 Exclude sequences with admissible subset
%
%               => Remaining sequence terms are initial sets
%
% 2.    Termination condition: no unattacked arguments in reduct
%
%       2.1     Flag attacked arguments
%       2.2     Indicate reducts containing unattacked arguments
%       2.3     Exclude sequences with improper last reduct
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% att/3         attack-relation within reduct
% attacked/2    argument attacked by sequence term
% collect/2     argument outside reduct
% excl/3        argument excluded from term
% flag/2        attacked arguments of reduct
% in/2          argument of sequence term
% index/1       index of sequence term
% in_card/2     cardinality of sequence term
% in_index/2    index of arguments of sequence term
% lt/3          lower-than relation over arguments of sequence term
% ninf/2        non-smallest arguments of sequence term
% non_adm/3     non-admissibility of numbered subset of sequence term
% non_def/4     non-defended arguments of subset of sequence term
% non_empty/1 non-empty sequence term
% non_terminate/1 reduct with unattacked arguments
% nsucc/3       non-successor relation over arguments of sequence term
% reduct/2      argument of reduct
% sub/3         argument of decremented term
% sub/4         argument of subset of sequence term
```

```
% sub_attacked/4 argument attacked by subset of sequence term
% succ/3        successor-relation over arguments of sequence term

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments

        index(1..C)             :-      { arg(X) } == C.

%       0.      GENERATE sequences of sets of arguments as solution candidates
%               for serialization sequences

        { in(X, Step) }         :-      reduct(X, Step).

%       Get cardinality of sequence terms

        in_index(1..C, Step)    :-      { in(X, Step) } == C,
                                        index(Step).

        in_card(C, Step)        :-      { in(X, Step) } == C,
                                        index(Step).

%       Define reduct

%       First reduct equals AF

        reduct(X, 1)            :-      arg(X).


%       Collect arguments from sequence term

        collect(X, Step)        :-      in(X, Step).

%       Collect arguments attacked by sequence term

        collect(X, Step)        :-      in(Y, Step),
                                        att(Y, X).

%       Next reduct has all non-collected arguments

        reduct(X, Step+1)       :-      reduct(X, Step),
                                        not collect(X, Step),
                                        index(Step).

%       .. and the relations between contained arguments

        att(X, Y, Step)         :-      reduct(X, Step),
                                        reduct(Y, Step),
                                        att(X, Y).

%       1.      Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial term
%
%       1.1.1   Exclude sequences with 'intermediate' empty term

        non_empty(Step)         :-      in(X, Step).

                                :-      not non_empty(Step),
                                        non_empty(Step+1),
                                        index(Step).

```

```
%       1.1.2   Exclude sequences with conflicting terms

                        :-      in(X, Step),
                                in(Y, Step),
                                att(X, Y, Step).


%       1.1.3   Exclude sequences with non-admissible term

%       Select arguments attacked by element

        attacked(X,Step)        :-      in(Y, Step),
                                        att(Y, X, Step).

%       Exclude sequences with non-defended arguments in term

                        :-      att(Y, X, Step),
                                in(X, Step),
                                not attacked(Y, Step).

%       RESULT: Remaining sequences only have non-empty admissible terms

%       1.1.4   Exclude sequences with non-minimal admissible term

%       1.1.4.1 Create subsets decremented by one element

%       Define an order over 'in' with succ-relation

        lt(X, Y, Step)          :-      in(X, Step),
                                        in(Y, Step),
                                        X<Y.

        nsucc(X, Z, Step)       :-      lt(X, Y, Step),
                                        lt(Y, Z, Step).

        succ(X, Y, Step)        :-      lt(X, Y, Step),
                                        not nsucc(X, Y, Step).

        ninf(X, Step)           :-      lt(Y, X, Step).

%       Define numbered arguments to be excluded

        excl(X, 1, Step)        :-      not ninf(X, Step),
                                        in(X, Step).

        excl(Y, No+1, Step)     :-      excl(X, No, Step),
                                        in(Y, Step),
                                        succ(X, Y, Step).

%       Define decremented sets (w/o excluded argument)

        sub(X, No, Step)        :-      in(X, Step),
                                        not excl(X, No, Step),
                                        in_index(No, Step).

%       Define first 'level' of subsets

        sub(X, No, Step, 0)     :-      sub(X, No, Step).

%       Select arguments attacked by 'sub'

        sub_attacked(Y, No, Step, Level):-
```

```
                                    sub(X, No, Step, Level),
                                    att(X, Y, Step).

%       Select non-defended arguments of 'sub'

        non_def(Y, No, Step, Level):-   sub(Y, No, Step, Level),
                                    att(X, Y, Step),
                                    not sub_attacked(X, No, Step, Level).

%       1.1.4.2 Define subsubsets by removing non-defended arguments

        sub(X, No, Step, Level+1):-     sub(X, No, Step, Level),
                                    not non_def(X, No, Step, Level),
                                    in_card(C, Step),
                                    Level < C.

%       1.1.4.3 Flag non-admissible subsets

        non_adm(No, Step, Level):-      non_def(Y, No, Step, Level).


%       1.1.4.4 Exclude sequences with admissible subset

                            :-      not non_adm(No, Step, Level),
                                    sub(X, No, Step, Level).

%       RESULT: Remaining sequence terms are initial sets

%       2.      Termination condition: no unattacked arguments in reduct

%       2.1     Flag attacked arguments

        flag(X, Step)       :-      reduct(X, Step),
                                    reduct(Y, Step),
                                    att(Y, X, Step).

%       2.2     Indicate reducts containing unattacked arguments

        non_terminate(Step) :-      reduct(X, Step),
                                    not flag(X, Step).

%       2.3     Exclude sequences with improper last reduct

                            :-      non_empty(Step),
                                    not non_empty(Step+1),
                                    non_terminate(Step+1),
                                    Step > 0.

%       Exclude improper empty set

                            :-      not non_empty(1),
                                    non_terminate(1).
#show in/2.
```

A.7. Serialization Sequence for Stable Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for serialization sequence of stable semantics
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences.
%
% 1.    Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial terms
%
%               1.1.1   Exclude sequences with 'intermediate' empty term
%               1.1.2   Exclude sequences with conflicting term
%               1.1.3   Exclude sequences with non-admissible term
%                       => Remaining sequences only have non-empty admissible
%                          terms
%               1.1.4   Exclude sequences with non-minimal admissible terms
%
%                       1.1.4.1 Create subsets decremented by one element
%                       1.1.4.2 Define subsubsets by removing non-defended arguments
%                       1.1.4.3 Flag non-admissible subsets
%                       => Non-flagged subsets are admissible
%                       1.1.4.4 Exclude sequences with admissible subset
%
%               => Remaining sequence terms are initial sets
%
% 2.    Termination condition: last reduct must be empty
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% att/3         attack-relation within reduct
% attacked/2    argument attacked by sequence term
% collect/2     argument outside reduct
% excl/3        argument excluded from term
% in/2          argument of sequence term
% index/1       index of sequence term
% in_card/2     cardinality of sequence term
% in_index/2    index of arguments of sequence term
% lt/3          lower-than relation over arguments of sequence term
% ninf/2        non-smallest arguments of sequence term
% non_adm/3     non-admissibility of numbered subset of sequence term
% non_def/4     non-defended arguments of subset of sequence term
% non_empty/1   non-empty sequence term
% nsucc/3       non-successor relation over arguments of sequence term
% reduct/2      argument of reduct
% sub/3         argument of decremented term
% sub/4         argument of subset of sequence term
% sub_attacked/4 argument attacked by subset of sequence term
% succ/3        successor-relation over arguments of sequence term

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments
```

```
index(1..C)             :-      { arg(X) } == C.

%       0.      GENERATE sequences of sets of arguments as solution candidates
%               for serialization sequences

{ in(X, Step) }         :-      reduct(X, Step).

%       Get cardinality of sequence terms

in_index(1..C, Step)    :-      { in(X, Step) } == C,
                                index(Step).

in_card(C, Step)        :-      { in(X, Step) } == C,
                                index(Step).

%       Define reduct

%       First reduct equals AF

reduct(X, 1)            :-      arg(X).


%       Collect arguments from sequence term

collect(X, Step)        :-      in(X, Step).

%       Collect arguments attacked by sequence term

collect(X, Step)        :-      in(Y, Step),
                                att(Y, X).

%       Next reduct has all non-collected arguments

reduct(X, Step+1)       :-      reduct(X, Step),
                                not collect(X, Step),
                                index(Step).

%        .. and the relations between contained arguments

att(X, Y, Step)         :-      reduct(X, Step),
                                reduct(Y, Step),
                                att(X,Y).

%       1.      Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial term
%
%       1.1.1   Exclude sequences with 'intermediate' empty term

non_empty(Step)         :-      in(X, Step).

                        :-      not non_empty(Step),
                                non_empty(Step+1),
                                index(Step).

%       1.1.2   Exclude sequences with conflicting terms

                        :-      in(X, Step),
                                in(Y, Step),
                                att(X, Y).
```

```
%       1.1.3   Exclude sequences with non-admissible term

%       Select arguments attacked by term

        attacked(X, Step)       :-      in(Y, Step),
                                        att(Y, X, Step).

%       Exclude sequences with non-defended arguments in term

                                :-      att(Y, X, Step),
                                        in(X, Step),
                                        not attacked(Y, Step).

%       RESULT: Remaining sequences only have non-empty admissible terms

%       1.1.4   Exclude sequences with non-minimal admissible term

%       1.1.4.1 Create subsets decremented by one element

%       Define an order over 'in' with succ-relation

        lt(X, Y, Step)          :-      in(X, Step),
                                        in(Y, Step),
                                        X<Y.

        nsucc(X, Z, Step)       :-      lt(X, Y, Step),
                                        lt(Y, Z, Step).

        succ(X, Y, Step)        :-      lt(X, Y, Step),
                                        not nsucc(X, Y, Step).

        ninf(X, Step)           :-      lt(Y, X, Step).

%       Define numbered arguments to be excluded

        excl(X, 1, Step)        :-      not ninf(X, Step),
                                        in(X, Step).

        excl(Y, No+1, Step)     :-      excl(X, No, Step),
                                        in(Y, Step),
                                        succ(X, Y, Step).

%       Define decremented sets (w/o excluded argument)

        sub(X, No, Step)        :-      in(X, Step),
                                        not excl(X, No, Step),
                                        in_index(No, Step).

%       Define first 'level' of subsets

        sub(X, No, Step, 0)     :-      sub(X, No, Step).

%       Select arguments attacked by 'sub'

        sub_attacked(Y, No, Step, Level):-
                                        sub(X, No, Step, Level),
                                        att(X, Y, Step).

%       Select non-defended arguments of 'sub'

        non_def(Y, No, Step, Level):-   sub(Y, No, Step, Level),
```

```
                                        att(X, Y, Step),
                                        not sub_attacked(X, No, Step, Level).

%       1.1.4.2 Define subsubsets by removing non-defended arguments

        sub(X, No, Step, Level+1):-     sub(X, No, Step, Level),
                                        not non_def(X, No, Step, Level),
                                        in_card(C, Step),
                                        Level < C.

%       1.1.4.3 Flag non-admissible subsets

        non_adm(No, Step, Level):-      non_def(Y, No, Step, Level).


%       1.1.4.4 Exclude sequences with admissible subset

                                :-      not non_adm(No, Step, Level),
                                        sub(X, No, Step, Level).

%       2.      Termination condition: last reduct must be empty

                                :-      not non_empty(Step),
                                        reduct(X, Step).

#show in/2.
```

A.8. Serialization Sequence for Preferred Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding of serialization sequence for preferred semantics.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences
%
% 1.    Each sequence term must be an initial set
%
%       1.1     Exclude sequences with non-initial term
%
%               1.1.1   Exclude sequences with 'intermediate' empty term
%               1.1.2   Exclude sequences with conflicting term
%               1.1.3   Exclude sequences with non-admissible term
%                       => Remaining sequences only have non-empty admissible
%                          terms
%               1.1.4   Exclude sequences with non-minimal admissible term
%
%                       1.1.4.1 Create subsets decremented by one element
%                       1.1.4.2 Define subsubsets by removing non-defended arguments
%                       1.1.4.3 Flag non-admissible subsets
%                       => Non-flagged subsets are admissible
%                       1.1.4.4 Exclude sequences with admissible subset
%
%               => Remaining sequence terms are initial sets
%
% 2.    Termination condition: no non-empty admissible set in reduct
%
%       2.1     Create all non-empty subsets of reduct
%       2.2     Flag conflicting subsets of reduct
%       2.3     Flag non-admissible subsets of reduct
%       2.4     Indicate reducts containing admissible sets
%       2.5     Exclude sequences with improper last reduct
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% att/3         attack-relation within reduct
% attacked/2    argument attacked by sequence term
% binvec/4      binary vector
% card/1        cardinality of set of all arguments
% collect/2     argument outside reduct
% excl/3        argument excluded from term
% flag/2        indicates conflicting and non-admissible subsets
% in/2          argument of sequence element
% index/1       index of sequence term
% in_card/2     cardinality of sequence term
% in_index/2    index of arguments of sequence term
% lt/3          lower-than relation over arguments of sequence term
% ninf/2        non-smallest arguments of sequence term
% non_adm/3     non-admissibility of numbered subset of sequence term
% non_def/4     non-defended arguments of subset of sequence term
% non_empty/1   non-empty sequence term
% non_terminate/1 reduct with non-empty admissible subset
```

```
% nsucc/3      non-successor relation over arguments of sequnce term
% r_attacked/3 argument attacked by subset of reduct
% r_card/2     cardinality of reduct
% r_elem/2     element of subset of reduct
% r_elem/3     element of subset of reduct
% r_lt/3       Lower-than relation over arguments of reduct
% r_ninf/2     non-smallest arguments of reduct
% r_nsucc/3    non-successor relation over arguments of reduct
% r_set/2      numbered subset of reduct
% r_succ/3     successor-relation over arguments of reduct
% reduct/2     argument of reduct
% reduct/3     numbered argument of reduct
% sub/3        argument of decremented set
% sub/4        argument of subset of sequence term
% sub_attacked/4 argument attacked by subset of sequence term
% succ/3       successor-relation over arguments of sequence term

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments

        index(1..C)             :-      { arg(X) } == C.
        card(C)                 :-      { arg(X) } == C.


%       0.      Generate sequences of sets of arguments as solution candidates
%               for serialization sequences

        { in(X, Step) }         :-      reduct(X, Step).

%       Get cardinality of sequence terms

        in_index(1..C, Step)    :-      { in(X, Step) } == C,
                                        index(Step).

        in_card(C, Step)        :-      { in(X, Step) } == C,
                                        index(Step).

%       Define reduct

%       First reduct equals AF

        reduct(X,1)             :-      arg(X).


%       Collect arguments from sequence term

        collect(X, Step)        :-      in(X, Step).


%       Collect arguments attacked by sequence term

        collect(X, Step)        :-      in(Y, Step),
                                        att(Y, X).


%       Next reduct has all non-collected arguments

        reduct(X, Step+1)       :-      reduct(X, Step),
                                        not collect(X, Step),
                                        index(Step).

```

```
%         .. and the relations between contained arguments

          att(X, Y, Step)         :-      reduct(X, Step),
                                          reduct(Y, Step),
                                          att(X, Y).

%         1.      Each sequence term must be an initial set

%         1.1     Exclude sequences with non-initial term

%         1.1.1   Exclude sequences with 'intermediate' empty term

          non_empty(Step)         :-      in(X, Step).

                                  :-      not non_empty(Step),
                                          non_empty(Step+1),
                                          index(Step).

%         1.1.2   Exclude sequences with conflicting term

                                  :-      in(X, Step),
                                          in(Y, Step),
                                          att(X, Y).


%         1.1.3   Exclude sequences with non-admissible term

%         Select arguments attacked by term

          attacked(X, Step)       :-      in(Y, Step),
                                          att(Y, X, Step).

%         Exclude sequences with non-defended arguments in term

                                  :-      att(Y, X, Step),
                                          in(X, Step),
                                          not attacked(Y, Step).

%         RESULT: Remaining sequences only have non-empty admissible terms
%
%         1.1.4   Exclude sequences with non-minimal admissible term

%         1.1.4.1 Create subsets decremented by one element

%         Define an order over 'in' with succ-relation

          lt(X, Y, Step)          :-      in(X, Step),
                                          in(Y, Step),
                                          X<Y.

          nsucc(X, Z, Step)       :-      lt(X, Y, Step),
                                          lt(Y, Z, Step).

          succ(X, Y, Step)        :-      lt(X, Y, Step),
                                          not nsucc(X, Y, Step).

          ninf(X, Step)           :-      lt(Y, X, Step).

%         Define numbered arguments to be excluded

          excl(X, 1, Step)        :-      not ninf(X, Step),
                                          in(X, Step).
```

```
excl(Y, No+1, Step)      :-      excl(X, No, Step),
                                 in(Y, Step),
                                 succ(X, Y, Step).

%       Define decremented sets (w/o excluded argument)

sub(X, No, Step)         :-      in(X, Step),
                                 not excl(X, No, Step),
                                 in_index(No, Step).

%       Define first 'level' of subsets

sub(X, No, Step, 0)      :-      sub(X, No, Step).

%       Select arguments attacked by 'sub'

sub_attacked(Y, No, Step, Level):-
                                 sub(X, No, Step, Level),
                                 att(X, Y, Step).

%       Select non-defended arguments of 'sub'

non_def(Y, No, Step, Level):-    sub(Y, No, Step, Level),
                                 att(X, Y, Step),
                                 not sub_attacked(X, No, Step, Level).

%       1.1.4.2 Define subsubsets by removing all non-defended arguments

sub(X, No, Step, Level+1):-      sub(X, No, Step, Level),
                                 not non_def(X, No, Step, Level),
                                 in_card(C, Step),
                                 Level < C.

%       1.1.4.3 Flag all non-admissible subsets

non_adm(No, Step, Level):-       non_def(Y, No, Step, Level).


%       1.1.4.4 Exclude sequences with admissible subset

                         :-      not non_adm(No, Step, Level),
                                 sub(X, No, Step, Level).

%       RESULT: Remaining sequence terms are initial sets

%       2.      Termination condition: no non-empty admissible set in reduct

%       2.1     Create all non-empty subsets of reduct (including identity)

%       Store cardinalities of reducts

r_card(C, Step)          :-      {reduct(X, Step)} == C,
                                 card(Ca),
                                 RStep = Ca + 1,
                                 Step = 1..RStep.

%       Define an order on reduct with succ-relation

r_lt(X, Y, Step)         :-      reduct(X, Step),
                                 reduct(Y, Step),
                                 X<Y.
```

```

        r_nsucc(X, Z, Step)       :-      r_lt(X, Y, Step),
                                          r_lt(Y, Z, Step).

        r_succ(X, Y, Step)        :-      r_lt(X, Y, Step),
                                          not r_nsucc(X, Y, Step).

        r_ninf(X, Step)           :-      r_lt(Y, X, Step).

%       Each argument of reduct is numbered accordingly

        reduct(X, Step, 0)        :-      not r_ninf(X, Step),
                                          reduct(X, Step).

        reduct(Y, Step, ArgNo+1):-        reduct(X, Step, ArgNo),
                                          reduct(Y, Step),
                                          r_succ(X, Y, Step).

%       Calculate binary vector by repeatedly divide number by 2.
%       'Rest' is 1 or 0 and assigns argument to subset.
%       'Result' is needed for the next division

%       Start

        binVec(SetNo, 0, SetNo\2, SetNo/2) :-
                                          r_card(C, 2),
                                          (2 ** C) - 1 = Max,
                                          SetNo = 1..Max.

%       Next

        binVec(SetNo, ArgNo+1, Result\2, Result/2) :-
                                          binVec(SetNo, ArgNo, _, Result),
                                          SetNo >= (2 ** (ArgNo+1)).

%       Use binary vector to relate reduct-arguments to the corresponding subset

        r_set(1..MaxSet, Step)    :-      (2 ** C) - 1 == MaxSet,
                                          r_card(C, Step).

%       Relate subsets to contained arguments

        r_elem(SetNo, ArgNo)      :-      binVec(SetNo, ArgNo, Rest, _),
                                          Rest = 1.

%       Relate subsets of sequence elements to contained arguments

        r_elem(SetNo, ArgNo, Step) :-     r_set(SetNo, Step),
                                          r_elem(SetNo, ArgNo).

%       2.2     Flag conflicting subsets of reduct

        flag(SetNo, Step)         :-      r_elem(SetNo, ArgNo1, Step),
                                          reduct(X, Step, ArgNo1),
                                          r_elem(SetNo, ArgNo2, Step),
                                          reduct(Y,Step,ArgNo2),
                                          att(X,Y).

%       2.3     Flag non-admissible subsets of reduct

        r_attacked(SetNo, X, Step) :-     r_elem(SetNo, ArgNo, Step),
                                          reduct(Y, Step, ArgNo),
```

```
                                        att(Y, X, Step).

        flag(SetNo, Step)       :-      r_elem(SetNo, ArgNo, Step),
                                        reduct(X, Step, ArgNo),
                                        att(Y, X, Step),
                                        not r_attacked(SetNo, Y, Step).

%       2.4     Indicate reducts containing admissible sets

        non_terminate(Step)     :-      r_set(SetNo, Step),
                                        not flag(SetNo, Step).

%       2.5     Exclude sequences with improper last reduct

                                :-      non_empty(Step),
                                        not non_empty(Step+1),
                                        non_terminate(Step+1),
                                        Step > 0.

%       Exclude improper empty set

                                :-      not non_empty(1),
                                        non_terminate(1).

#show in/2.
```

A.9. Serialization Sequence for Grounded Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for serialization sequence of grounded semantics.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences.
%
% 1.    Sequence elements must be unattacked initial sets
%
%       1.1     Exclude sequences with 'intermediate' empty term
%       1.2     Exclude sequences with more than one argument in term
%       1.3     Exclude sequences with attacked arguments
%
%               => Remaining sequence terms are unattacked initial sets
%
% 2.    Termination condition: no unattacked arguments in reduct
%
%       2.1     Flag attacked arguments
%       2.2     Indicate reducts containing unattacked arguments
%       2.3     Exclude sequences with improper last reduct
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% att/3         attack-relation within reduct
% collect/2     argument outside reduct
% flag/2        attacked argument of term
% in/2          argument of solution candidate
% index/1       index of sequence term
% non_empty/1 non-empty term
% non_terminate non-terminating reduct
% reduct/2      argument of reduct

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments

        index(1..C)             :-      { arg(X) } == C.

%       0.      GENERATE sequences of sets of arguments as solution candidates
%               for serialization sequences.

        { in(X, Step) }         :-      reduct(X, Step),
                                        index(Step).

%       Define reduct

%       First reduct equals AF

        reduct(X,1)             :-      arg(X).


%       Collect arguments from sequence term

```

```
        collect(X, Step)        :-      in(X,Step),
                                        index(Step).


%       Collect arguments attacked by sequence term

        collect(X, Step)        :-      in(Y,Step),
                                        att(Y,X),
                                        index(Step).


%       Next reduct has all non-collected arguments

        reduct(X, Step+1)       :-      reduct(X,Step),
                                        not collect(X,Step),
                                        index(Step).

%        .. and the relations between contained arguments

        att(X, Y, Step)         :-      reduct(X, Step),
                                        reduct(Y, Step),
                                        att(X, Y).

%       1       Sequence terms must be unattacked initial sets

%       1.1     Exclude sequences with 'intermediate' empty term

        non_empty(Step)         :-      in(X, Step).

                                :-      not non_empty(Step),
                                        non_empty(Step+1),
                                        index(Step).

%       1.2     Exclude sequences with more than one argument in term

                                :-      in(X, Step),
                                        in(Y, Step),
                                        X != Y.

%       1.3     Exclude sequences with attacked arguments

                                :-      in(X, Step),
                                        att(Y, X, Step),
                                        reduct(Y, Step).

%       RESULT: Remaining sequence terms are unattacked initial sets


%       2.      Termination condition: no unattacked arguments in last reduct

%       2.1     Flag attacked arguments

        flag(X, Step)           :-      reduct(X, Step),
                                        reduct(Y, Step),
                                        att(Y, X, Step).

%       2.2     Indicate reducts containing unattacked arguments

        non_terminate(Step)     :-      reduct(X, Step),
                                        not flag(X, Step).

%       2.3     Exclude sequences with improper last reduct
```

```
                                         :-      non_empty(Step),
                                                 not non_empty(Step+1),
                                                 non_terminate(Step+1),
                                                 Step > 0.

%        Exclude improper empty set

                                         :-      not non_empty(1),
                                                 non_terminate(1).
#show in/2.
```

A.10. Serialization Sequence for Strongly Admissible Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-Encoding for serialization sequence of strongly admissible semantics.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences.
%
% 1.    Sequence elements must be unattacked initial sets
%
%       1.1     Exclude sequences with 'intermediate' empty term
%       1.2     Exclude sequences with more than one argument in term
%       1.3     Exclude sequences with attacked arguments
%
%               => Remaining sequence terms are unattacked initial sets

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% List of predicates
%
% arg/1        arguments of AF
% att/2        attack-relation
% att/3        attack-relation within reduct
% collect/2    argument outside reduct
% flag/2       attacked argument of term
% in/2         argument of solution candidate
% index/1      index of sequence term
% non_empty/1  non-empty term
% reduct/2     argument of reduct

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments

        index(1..C)             :-      { arg(X) } == C.

%       0.      GENERATE sequences of sets of arguments as solution candidates
%               for serialization sequences.

        { in(X, Step) }         :-      reduct(X, Step).

%       Define reduct

%       First reduct equals AF

        reduct(X,1)             :-      arg(X).


%       Collect arguments from sequence term

        collect(X, Step)        :-      in(X,Step).


%       Collect arguments attacked by sequence term

        collect(X, Step)        :-      in(Y,Step),
                                        att(Y,X).

```

```

%        Next reduct has all non-collected arguments

         reduct(X, Step+1)        :-       reduct(X,Step),
                                           not collect(X,Step),
                                           index(Step).

%         .. and the relations between contained arguments

         att(X, Y, Step)          :-       reduct(X, Step),
                                           reduct(Y, Step),
                                           att(X, Y).

%        1        Sequence terms must be unattacked initial sets

%        1.1      Exclude sequences with 'intermediate' empty term

         non_empty(Step)          :-       in(X, Step).

                                  :-       not non_empty(Step),
                                           non_empty(Step+1),
                                           index(Step).

%        1.2      Exclude sequences with more than one argument in term

                                  :-       in(X, Step),
                                           in(Y, Step),
                                           X != Y.

%        1.3      Exclude sequences with attacked arguments

                                  :-       in(X, Step),
                                           att(Y, X, Step),
                                           reduct(Y, Step).

%        RESULT: Remaining sequence terms are unattacked initial sets

#show in/2.
```

A.11. Serialization Sequence for Unchallenged Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASP-encoding of serialization sequence for unchallenged semantics
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Algorithm
%
% 0.    Generate sequences of sets of arguments as solution candidates
%       for serialization sequences
%
% 1.    Each sequence term must be an initial set
%
%       1.1     Exclude sequences with 'intermediate' empty term
%       1.2     Exclude sequences with conflicting term
%       1.3     Exclude sequences with non-admissible term
%               => Remaining sequence terms are non-empty admissible
%       1.4     Exclude sequences with non-minimal admissible term
%
%               1.4.1   Define subsets decremented by one term
%               1.4.2   Define subsubsets by removing non-defended arguments
%               1.4.3   Flag non-admissible subsets
%               => Non-flagged subsets are admissible
%               1.4.4 Exclude sequences with admissible subset
%
%       => Remaining sequence terms are initial sets
%
% 2.    Exclude sequences with challenged term (attacked by initial set)
%
%       2.1     Define all non-empty subsets of reduct
%       2.2     Flag non-initial subsets
%               2.3.1   Flag conflicting subsets
%               2.3.2   Flag non-admissible subsets
%               2.3.3   Flag non-minimal subsets
%               => Non-flagged subsets are initial sets
%       2.4     Exclude sequences with terms attacked by non-flagged
%               subsets
%
%       => Remaining sequence terms are unattacked or unchallenged initial sets
%
% 3.    Termination condition: no unattacked or unchallenged initial set in reduct
%
%       3.1     Sign flagged subsets (unsigned subsets are initial sets)
%       3.2     Sign all subsets attacked by non-signed subsets
%               => Non-signed subsets are unattacked or unchallenged initial sets
%       3.3     Indicate reducts containing non-signed subsets
%       3.4     Exclude sequences with improper last reduct

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% List of predicates
%
% arg/1         arguments of AF
% att/2         attack-relation
% att/3         attack-relation within reduct
% attacked/2    argument attacked by sequence term
% binVec/4      binary vector
% card/1        cardinality of set of all arguments
% collect/2     argument outside reduct
% elem/2        relation of subsets and contained arguments
% elemIni/3     argument of initial set
```

```
% excl/3        argument excluded from set
% flag/2        flagged subsets of reduct
% in/2          argument of sequence term
% index/1.      index of sequence term
% iniSet/2      initial set
% in_card/2     cardinality of sequence term
% in_index/2    index of elements of sequence term
% lt/3          lower-than relation over arguments of sequence term
% ninf/2        non-smallest arguments of sequence term
% non_adm/3     indicates non-admissiblity of subset of sequence term
% non_def/4     mom-defended arguments of subset
% non_empty/1 non-emptiness of sequence term
% non_terminate/1 indicates non-terminating reducts
% nsucc/3       non-successor relation over arguments of sequence term
% r_attacked/3 argument attacked by subset of reduct
% r_card/2      cardinality of reduct
% r_elem/3      relation of reduct-subset and contained arguments
% r_lt/3        lower-than relation over arguments of reduct
% r_ninf/2      non-smallest arguments of reduct
% r_nonIS/2     non-initial sets of reduct
% r_nsucc/3     non-successor relation over arguments of reduct
% r_set/2       numbered subset of reduct
% r_sign/2      non-initial set in reduct
% r_sub/3       relation of reduct-subsets and contained subsubsets
% r_succ/3      successor-relation over arguments of reduct
% reduct/2      argument of reduct
% reduct/3      numbered argument of reduct
% set/2         numbered subset of sequence term
% sub/2         relation of subsets to contained subsets
% sub/3         argument of decremented set
% sub/4         argument of subsubset
% sub_attacked/4 argument attacked by subset of sequence term
% succ/3        successor-relation over arguments of sequence term

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       Get number of arguments

        index(1..C)             :-      { arg(X) } == C.
        card(C)                 :-      { arg(X) } == C.


%       0.      Generate sequences of sets of arguments as solution candidates
%               for serialization sequences

        { in(X, Step) }         :-      reduct(X, Step).

%       Get cardinality of sequence terms

        in_index(1..C, Step)    :-      { in(X, Step) } == C,
                                        index(Step).

        in_card(C, Step)        :-      { in(X, Step) } == C,
                                        index(Step).

%       Define reduct

%       First reduct equals AF

        reduct(X,1)             :-      arg(X).

%       Collect arguments from sequence term
```

```
        collect(X, Step)        :-      in(X, Step).

%       Collect arguments attacked by sequence term

        collect(X, Step)        :-      in(Y, Step),
                                        att(Y, X).

%       Next reduct has all non-collected arguments

        reduct(X, Step+1)       :-      reduct(X, Step),
                                        not collect(X, Step),
                                        index(Step).

%        .. and the relations between contained arguments

        att(X, Y, Step)         :-      reduct(X, Step),
                                        reduct(Y, Step),
                                        att(X, Y).

%       1.      Each sequence term must be an initial set

%       1.1     Exclude sequences with non-initial term

%       1.1.1   Exclude sequences with 'intermediate' empty term

        non_empty(Step)         :-      in(X, Step).

                                :-      not non_empty(Step),
                                        non_empty(Step+1),
                                        index(Step).

%       1.1.2   Exclude sequences with conflicting term

                                :-      in(X, Step),
                                        in(Y, Step),
                                        att(X, Y).


%       1.1.3   Exclude sequences with non-admissible term

%       Select arguments attacked by term

        attacked(X, Step)       :-      in(Y, Step),
                                        att(Y, X, Step).

%       Exclude sequences with non-defended arguments in term

                                :-      att(Y, X, Step),
                                        in(X, Step),
                                        not attacked(Y, Step).

%       RESULT: Remaining sequences only have non-empty admissible terms
%
%       1.1.4   Exclude sequences with non-minimal admissible term

%       1.1.4.1 Create subsets decremented by one term

%       Define an order over 'in' with succ-relation

        lt(X, Y, Step)          :-      in(X, Step),
                                        in(Y, Step),
```

```
                                     X<Y.

        nsucc(X, Z, Step)       :-      lt(X, Y, Step),
                                        lt(Y, Z, Step).

        succ(X, Y, Step)        :-      lt(X, Y, Step),
                                        not nsucc(X, Y, Step).

        ninf(X, Step)           :-      lt(Y, X, Step).

%       Define numbered arguments to be excluded

        excl(X, 1, Step)        :-      not ninf(X, Step),
                                        in(X, Step).

        excl(Y, No+1, Step)     :-      excl(X, No, Step),
                                        in(Y, Step),
                                        succ(X, Y, Step).

%       Define decremented sets (w/o excluded argument)

        sub(X, No, Step)        :-      in(X, Step),
                                        not excl(X, No, Step),
                                        in_index(No, Step).

%       Define first 'level' of subsets

        sub(X, No, Step, 0)     :-      sub(X, No, Step).

%       Select arguments attacked by 'sub'

        sub_attacked(Y, No, Step, Level):-
                                        sub(X, No, Step, Level),
                                        att(X, Y, Step).

%       Select non-defended arguments of 'sub'

        non_def(Y, No, Step, Level):-   sub(Y, No, Step, Level),
                                        att(X, Y, Step),
                                        not sub_attacked(X, No, Step, Level).

%       1.1.4.2 Define subsubsets by removing all non-defended arguments

        sub(X, No, Step, Level+1):-     sub(X, No, Step, Level),
                                        not non_def(X, No, Step, Level),
                                        in_card(C, Step),
                                        Level < C.

%       1.1.4.3 Flag all non-admissible subsets

        non_adm(No, Step, Level):-      non_def(Y, No, Step, Level).


%       1.1.4.4 Exclude sequences with admissible subset

                                :-      not non_adm(No, Step, Level),
                                        sub(X, No, Step, Level).

%       RESULT: Remaining sequence terms are initial sets

% 2.    Exclude sequences with challenged term (attacked by initial set)

```

```
%       2.1     Define all non-empty subsets of reduct

%       Define an order over reduct with succ-relation

        r_lt(X, Y, Step)         :-      reduct(X, Step),
                                         reduct(Y, Step),
                                         X<Y.

        r_nsucc(X, Z, Step)      :-      r_lt(X, Y, Step),
                                         r_lt(Y, Z, Step).

        r_succ(X, Y, Step)       :-      r_lt(X, Y, Step),
                                         not r_nsucc(X, Y, Step).

        r_ninf(X, Step)          :-      r_lt(Y, X, Step).

%       Each argument of reduct is numbered accordingly

        reduct(X, Step, 0)       :-      not r_ninf(X, Step),
                                         reduct(X, Step).

        reduct(Y, Step, ArgNo+1):-       reduct(X, Step, ArgNo),
                                         reduct(Y, Step),
                                         r_succ(X, Y, Step).

%       Calculate binary vector by repeatedly divide number by 2.
%       'Rest' is 1 or 0 and assigns argument to subset.
%       'Result' is needed for the next division

%       Start

        binVec(SetNo, 0, SetNo\2, SetNo/2) :-
                                         card(C),
                                         (2 ** C) - 1 = Max,
                                         SetNo = 1..Max.

%       Next

        binVec(SetNo, ArgNo+1, Result\2, Result/2) :-
                                         binVec(SetNo, ArgNo, _, Result),
                                         SetNo >= (2 ** (ArgNo+1)).

%       Relate subsets to contained arguments

        elem(SetNo, ArgNo)       :-      binVec(SetNo, ArgNo, Rest, _),
                                         Rest = 1.

%       Relate subsets to contained subsets (w/o empty set)

        sub(SetNo, SubSet)       :-      binVec(SetNo, ArgNo, Rest, Result),
                                         Rest = 1,
                                         SubSet = 2 ** (ArgNo).

        sub(SetNo, SubA+SubB)    :-      sub(SetNo, SubA),
                                         sub(SetNo, SubB),
                                         SubA != SubB,
                                         SubA + SubB <= SetNo.

%       Use binary vector to relate reduct-arguments and subsubsets to the
%       corresponding subset

        r_card(C, Step)          :-      { reduct(X, Step) } = C,
```

```
                                        index(Step).

        r_set(1..MaxSet, Step)   :-     (2 ** C) - 1 == MaxSet,
                                        r_card(C, Step).

%       Relate subsets of reduct to contained arguments for each step

        r_elem(SetNo, ArgNo, Step):-    r_set(SetNo, Step),
                                        elem(SetNo, ArgNo).

%       Relate subsets of reduct to contained subsets for each step

        r_sub(SetNo, SubSet, Step):-    r_set(SetNo, Step),
                                        sub(SetNo, SubSet).

%       2.2     Flag non-initial subsets
%       2.3.1   Flag conflicting subsets

        flag(SetNo, Step)        :-     r_elem(SetNo, ArgNo1, Step),
                                        r_elem(SetNo, ArgNo2, Step),
                                        reduct(X, Step, ArgNo2),
                                        reduct(Y, Step, ArgNo1),
                                        att(X, Y, Step).

%       2.3.2   Flag non-admissible subsets

        r_attacked(SetNo, X, Step):-    r_elem(SetNo, ArgNo, Step),
                                        reduct(Y, Step, ArgNo),
                                        att(Y, X, Step).

        flag(SetNo, Step)        :-     r_elem(SetNo, ArgNo, Step),
                                        reduct(X, Step, ArgNo),
                                        att(Y, X, Step),
                                        not r_attacked(SetNo, Y, Step).

%       2.3.3   Flag non-minimal subsets

        flag(SetNo1, Step)       :-     r_set(SetNo1, Step),
                                        r_set(SetNo2, Step),
                                        SetNo1 != SetNo2,
                                        r_sub(SetNo1, SetNo2, Step),
                                        not flag(SetNo2, Step).

%       RESULT: Non-flagged subsets are initial sets

%       2.3     Exclude sequences with terms attacked by non-flagged subsets

        iniSet(SetNo, Step)      :-     r_set(SetNo, Step),
                                        not flag(SetNo, Step).

        elemIni(SetNo, X, Step):-       iniSet(SetNo, Step),
                                        r_elem(SetNo, ArgNo, Step),
                                        reduct(X, Step, ArgNo).

                                 :-     elemIni(SetNo, X, Step),
                                        in(Y, Step),
                                        att(X,Y, Step).

%       RESULT: Remaining sequence terms are unattacked or unchallenged initial sets

%       3.      Termination condition: no unattacked or unchallenged initial set in reduct
%
```

```
%       3.1     Sign flagged subsets (unsigned subsets are initial sets)

        r_sign(SetNo, Step)     :-      flag(SetNo, Step).

%       3.2     Sign all subsets attacked by non-flagged subsets

        r_sign(SetNo1, Step)    :-      r_elem(SetNo1, ArgNo1, Step),
                                        reduct(X, Step, ArgNo1),
                                        att(Y, X, Step),
                                        reduct(Y, Step, ArgNo2),
                                        r_elem(SetNo2, ArgNo2, Step),
                                        not flag(SetNo2, Step).

%       RESULT: Non-signed subsets are unattacked or unchallenged initial sets

%       3.4     Indicate reducts containing non-signed subsets

        non_terminate(Step)     :-      r_set(SetNo, Step),
                                        not r_sign(SetNo, Step).

%       3.5     Exclude sequences with improper last reduct

                                :-      non_empty(Step),
                                        not non_empty(Step+1),
                                        non_terminate(Step+1),
                                        Step > 0.

%       Exclude improper empty set

                                :-      not non_empty(1),
                                        non_terminate(1).
#show in/2.
```

B. Java Code

B.1. Computing Serialization Sequences

```java
package mytweety;

import org.tweetyproject.arg.dung.parser.ApxParser;
import org.tweetyproject.arg.dung.reasoner.serialisable.SerialisedAdmissibleReasoner;
import org.tweetyproject.arg.dung.syntax.DungTheory;

import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class SerSeqAd {

   public static void main(String[] args) {

        String pathname = args[0];
        File inputFile = new File(pathname);
        FileReader apxReader = null;
        DungTheory af = null;
        String filename = null;
        SerialisedAdmissibleReasoner reasoner = new SerialisedAdmissibleReasoner();

        String sequence;

        long startTime = System.nanoTime();

        try {
               apxReader = new FileReader(inputFile);
               filename = inputFile.getName();
               af = new ApxParser().parse(apxReader);
        } catch (IOException e) {

        e.printStackTrace();
        }

        sequence = reasoner.getSequences(af).toString();

        long endTime = System.nanoTime();
        double duration = (endTime - startTime)/1000000000.0;

        System.out.println(filename + ": " + sequence);
        System.out.println(filename + ": " + duration + " s");
  }
}
```

B.2. Generating Sample Argumentation Frameworks

```java
package mytweety;

import java.io.File;
import java.io.IOException;

import org.tweetyproject.arg.dung.util.DefaultDungTheoryGenerator;
import org.tweetyproject.arg.dung.util.DungTheoryGenerationParameters;
import org.tweetyproject.arg.dung.util.DungTheoryGenerator;
import org.tweetyproject.arg.dung.writer.ApxWriter;

```

```java

public class GenerateTestAF {

    public static void main(String[] args) throws IOException{

        int[] sizes = {10, 20, 30, 40};
        double[] density = {.5};
        int count = 4;
        ApxWriter writer = new ApxWriter();
        String path = System.getProperty("user.home")
                + File.separator + "Dropbox"
                + File.separator + "Fernuni"
                + File.separator + "Bachelorarbeit"
                + File.separator + "Evaluation"
                + File.separator + "DiffSize35";
        createDir(path);
        DungTheoryGenerationParameters params = new DungTheoryGenerationParameters();

        for (int j = 0; j < sizes.length; j++) {
            params.numberOfArguments = sizes[j];
            for(int k = 0; k < density.length; k++) {
                params.attackProbability = density[k];
                DungTheoryGenerator gen2 = new DefaultDungTheoryGenerator(params);
                for (int i = 0; i < count; i++) {
                    File f = new File(path + File.separator
                     +  sizes[j] + "-" + density[k]
                     + "AF" + i + ".apx");
                    writer.write(gen2.next(), f);
            }
            }
        }

    }

    private static void createDir(String path) {
        File customDir = new File(path);
        customDir.mkdirs();
    }
}
```