Solving Argumentation Problems Using Answer Set Programming with Quantifiers: Preliminary Report

Wolfgang Faber[1]

[1]University of Klagenfurt, Austria

Abstract

Problems in abstract argumentation are typically beyond NP, but stay in the polynomial hierarchy. Answer set programming with quantifiers, ASP(Q), has recently been proposed as an extension of answer set programming, suitable for expressing problems in the polynomial hierarchy in arguably elegant ways. Already in the original paper, argumentation has been mentioned as an application domain for ASP(Q), but to our knowledge this has not been followed up yet. In this paper we provide a preliminary study of encoding problems in argumentation using ASP(Q). We also examine the computational behaviour of these encodings.

Keywords

abstract argumentation, answer set programming with quantifiers

1. Introduction

Following Dung's seminal paper [1], a lot of research has been done on abstract argumentation. Some of this work was done on defining semantics, other work extended the original notion of argumentation framework, yet more work studied complexity and provided implementations. Usually, (Dung's) argumentation frameworks are defined as labeled graphs, and the computational tasks often involve reasoning about set relations. These tasks usually stay within the polynomial hierarchy (PH). Since many tasks are within the second level of the polynomial hierarchy, Answer Set Programming (ASP) has been suggested early on as a computational back-end for solving argumentation problems [2].

While ASP is a declarative formalism, encoding problems beyond NP is often involved and involves techniques such as saturation, which are notoriously difficult to handle and read. Several attempts were made to improve this, the most recent being Answer Set Programming with Quantifiers, ASP(Q), which combines several ASP parts with quantifiers over answer sets, which is an arguably much more accessible way of expressing problems beyond NP (but within PH).

It is therefore natural to use ASP(Q) for representing computational tasks arising in abstract argumentation, which we propose in this paper. Actually, one such task has already been discussed in [3], namely argumentation coherence, which is the problem of deciding whether two semantics (stable and preferred extensions) coincide for a given argumentation framework. In this paper, we actually take a step back from this problem and discuss the problem of computing extensions.

In particular, we focus on three "classic" semantics for Dung-style argumentation frameworks, *preferred* [1], *semi-stable* [4] , and *stage* [5] extensions. The ASP encodings for these semantics use saturation and are therefore difficult to read and understand. We show that, as expected, ASP(Q) encodings are very concise and readable. A natural question though is whether there is a performance penalty to pay for this. We report on preliminary experiments using the ASP(Q) solver *pyqasp*, which indicate that there is not a big performance penalty.

We conjecture that many problems in argumentation can be encoded in similarly readable ways, which would allow for new syntaxtic extensions and semantics to be handled using ASP(Q) relatively easily, while still providing a reasonable computational tool in terms of performance.

ASPOCP 2024: 17th Workshop on Answer Set Programming and Other Computing Paradigms, October, 2024, Dallas, USA.

2. Abstract Argumentation

We recall some definitions of abstract argumentation, originally defined in [1].

Definition 1. *An* argumentation framework (AF) *is a pair $F = (A, R)$ where A is a set of arguments and $R \subseteq A \times A$. $(a, b) \in R$ means that a attacks b. An argument $a \in A$ is* defended *by $S \subseteq A$ (in F) if, for each $b \in A$ such that $(b, a) \in R$, there exists a $c \in S$, such that $(c, b) \in R$. An argument a is admissible (in F) w.r.t. a set $S \subseteq A$ if each $b \in A$ which attacks a is defended by S.*

While many semantics have been defined for AFs, we focus here on extension-based semantics, where an extension is a set of acceptable arguments. There are different extension-based semantics, reflecting different notions of acceptability. In this paper we look at admissible, preferred, semi-stable, and stage extensions. The definition mostly follows the one in [6].

Definition 2. *Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is said to be* conflict-free (in F)*, if there are no $a, b \in S$, such that $(a, b) \in R$. For a set $S \subseteq A$, let the range S^+ be $S \cup \{x \mid \exists y \in S : (y, x) \in R\}$ (so S and all arguments attacked from within S).*

A set $S \subseteq A$ is an admissible extension *of F, if S is conflict-free in F and each $a \in S$ is admissible in F w.r.t. S. The set of admissible extensions of F is denoted as $adm(F)$.*

A set $S \subseteq A$ is an preferred extension *of F, if $S \in adm(F)$ and there is no $T \in adm(F)$ with $T \supset S$. The set of preferred extensions of F is denoted as $prf(F)$.*

A set $S \subseteq A$ is a semi-stable extension *of F, if $S \in adm(F)$ and there is no $T \in adm(F)$ with $T^+ \supset S^+$. The set of semi-stable extensions of F is denoted as $sem(F)$.*

A set $S \subseteq A$ is a stage extension *of F, if S is conflict-free in F and there is no conflict-free T in F with $T^+ \supset S^+$. The set of stage extensions of F is denoted as $stg(F)$.*

3. Answer Set Programming with Quantifiers

Answer Set Programming with Quantifiers (ASP(Q)) has been proposed in [7], providing a formalism reminiscent of Quantified Boolean Formulas, but based on ASP, and quantifying over answer sets rather than propositional variables. An ASP(Q) program is of the form

$$\Box_1 P_1 \Box_2 P_2 \cdots \Box_n P_n : C,$$

where, for each $i \in \{1, \ldots, n\}$, $\Box_i \in \{\exists, \forall\}$, P_i is an ASP program, and C is a stratified normal ASP program (this is, as intended by the ASP(Q) authors, a "check" in the sense of constraints). $\exists$ and $\forall$ are called *existential* and *universal answer set quantifiers*, respectively.

As a brief example, the intuitive reading of an ASP(Q) program $\exists P_1 \forall P_2 : C$ is that there exists an answer set A_1 of P_1 such that for each answer set A_2 of $P_2 \cup A_1$ it holds that $C \cup A_2$ is consistent (i.e. has an answer set).

Let us be more precise about the program $P \cup A$, that is, a program P being extended by an answer set A (or rather by an interpretation A): For an interpretation I, let $f_P(I)$ be the ASP program that contains all atoms in I as facts and all atoms a appearing in P but not in I as constraints (i.e. as a rule $\bot \leftarrow a$). Furthermore, for a program P and an interpretation I, let $f_P(\Pi, I)$ be the ASP(Q) program obtained from an ASP(Q) program Π by replacing the first program P_1 in Π with $P_1 \cup f_P(I)$. *Coherence* of an ASP(Q) program is then defined inductively:

- $\exists P : C$ is coherent if there exists an answer set M of P such that $C \cup f_P(M)$ has at least one answer set.
- $\forall P : C$ is coherent if for all answer sets M of P it holds that $C \cup f_P(M)$ has at least one answer set.
- $\exists P \Pi$ is coherent if there exists an answer set M of P such that $f_P(\Pi, M)$ is coherent.
- $\forall P \Pi$ is coherent if for all answer sets M of P it holds that $f_P(\Pi, M)$ is coherent.

In addition, for an existential ASP(Q) program Π (one that starts with $\exists$), the witnessing answer sets of the first ASP program P_1 are referred to as *quantified answer sets*.

4. ASP(Q) Encodings

Here, we provide ASP(Q) encodings for preferred, semi-stable, and stage extensions of argumentation frameworks. Here we assume (A, R) to be given in the apx format, which is in fact an ASP fact base: for each $a \in A$ it contains a fact `arg(a).`, and for each $(a, b) \in R$ it contains a fact `att(a,b).`. For representing an ASP(Q) program we use the *pyqasp* syntax, in which $\exists$ and $\forall$ are replaced by the strings `%@exists` and `%@forall`, respectively, and the colon is replaced by `%@constraint`.

We begin with the encoding for preferred extensions, which builds on the well-known encoding for admissible extensions available in the system ASPARTIX[1]. ASPARTIX is a collection of ASP encodings for a wide range of argumentation tasks.

```
%@exists

% apx facts go here

%% Guess S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).

%% Argument X is defeated by S
defeated(X) :- in(Y), att(Y,X).

%% Argument X is not defended by S
not_defended(X) :- att(Y,X), not defeated(Y).

%% Each X \in S has to be defended by S
:- in(X), not_defended(X).

%@forall

%% Guess a set S1 \supseteq S
in1(X) :- in(X).
in1(X) :- not out1(X), arg(X).
out1(X) :- not in1(X), arg(X).

%% Admissibility of S1

%% S1 has to be conflict-free
:- in1(X), in1(Y), att(X,Y).

%% Argument X is defeated by S1
defeated1(X) :- in1(Y), att(Y,X).

%% Argument X is not defended by S1
not_defended1(X) :- att(Y,X), not defeated1(Y).

%% Each X \in S has to be defended by S
:- in1(X), not_defended1(X).
```

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

```
%@constraint

%% If one S1 is a proper superset and admissible, then S is not preferred.
:- in1(X), not in(X), arg(X).
```

In fact, the admissible extension encoding is essentially duplicated in the $\forall$ program, with the addition of a rule that only admits supersets of the admissible extension determined in the $\exists$ program. The check just makes sure that no strict superset is actually admissible. It is clear that the quantified answer sets correspond to preferred extensions.

We next present the encoding for semi-stable extensions, which is similar to the previous ASP(Q) program, but it additionally defines the ranges of the sets and uses these for the check.

```
%@exists

% apx facts go here

%% Guess S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).

%% Argument X is defeated by the set S
defeated(X) :- in(Y), att(Y,X).

%% Argument X is not defended by S
not_defended(X) :- att(Y,X), not defeated(Y).

%% Each X \in S has to be defended by S
:- in(X), not_defended(X).

%% S+ : S plus all arguments attacked by any argument in S

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

%@forall

%% Guess a set S1 \supseteq S
in1(X) :- in(X).
in1(X) :- not out1(X), arg(X).
out1(X) :- not in1(X), arg(X).

%% Admissibility of S1

%% S1 has to be conflict-free
:- in1(X), in1(Y), att(X,Y).
```

```
%% Argument X is defeated by S1
defeated1(X) :- in1(Y), att(Y,X).

%% Argument X is not defended by S1
not_defended1(X) :- att(Y,X), not defeated1(Y).

%% Each X \in S has to be defended by S
:- in1(X), not_defended1(X).

%% S1+ : S1 plus all arguments attacked by any argument in S1

inplus1(X) :- in1(X).
inplus1(X) :- in1(Y), att(Y,X).

%@constraint

%% If one S1+ is a proper superset of S+ and S1 is admissible, then S is not preferred.

:- inplus1(X), not inplus(X), arg(X).
```

So here it is checked that no range of a superset of an admissible extension is a superset of the range of the admissible extension. Again it is clear that the quantified answer sets correspond to semi-stable extensions.

Finally, we present the encoding for stage extensions. Here, we only look at conflict-free sets rather than admissible extensions, but the check involves the ranges, as for semi-stable extensions.

```
%@exists

% apx facts go here

%% Guess S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).

%% S+ : S plus all arguments attacked by any argument in S

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

%@forall


%% Guess a set S1 \supseteq S
in1(X) :- in(X).

in1(X) :- not out1(X), arg(X).
out1(X) :- not in1(X), arg(X).
```

```
inplus1(X) :- in1(X).
inplus1(X) :- in1(Y), att(Y,X).

%% Conflict-freeness of S1

%% S1 has to be conflict-free
:- in1(X), in1(Y), att(X,Y).

%@constraint

%% If one S1+ is a proper superset of S+ and S1 is conflict-free, then S is not a stage extensi

:- inplus1(X), not inplus(X), arg(X).
```

Again it is clear that the quantified answer sets correspond to stage extensions.

5. Experimental Results

We have conducted some preliminary results with the encodings presented in the previous section. We have used the 107 instances of the ICCMA 2019 competition[2] and ran them using *pyqasp* in the version presented in [8]. The machine used was i7-1165G7 at 2.80GHz with 64GiB RAM running Ubuntu 22.04.5 LTS. Since the machine was also running other jobs, we have restricted the memory usage to 8GiB. The runtime was restricted to 5 minutes. The computational task was to compute one extension.

For preferred extensions, 42 instances were solved within the time limit, with 65 timing out. For semi-stable extensions, 43 instances were solved within the time limit, with 64 timing out. For stage extensions, only 17 were solved within the time limit, with 90 timing out. Overall, we believe that this is an acceptable result, as these are comparatively hard instances.

We have also compared the runtime to the ASPARTIX encodings for clingo. Interestingly, clingo had memory issues when computing semi-stable extensions, 56 instances exceeded the memory limit. 19 more timed out, leaving 32 solved instances. The picture was quite different when computing stage extensions: 91 were successfully solved by clingo, and only 16 timed out. Clingo was very performant for preferred extensions, only 7 timed out.

In Figures 3, 1, and 2 we provide scatter plots comparing pyqasp's and clingo's performance. The runtime for pyqasp on a specific instance determines the vertical position, while the runtime for clingo for the same instance determines the horizontal position of a dot. So, each dot in these diagrams represents one instance - if it is in the upper left of the diagram, clingo was faster, if it is in the lower right, then pyqasp was faster. We can see that pyqasp seems more performant for computing a semi-stable extension, whereas clingo seems more performant for computing a preferred or stage extension overall.

6. Conclusions

We have shown that some well-known semantics for argumentation frameworks can be encoded in a very intuitive way using ASP(Q). While this is not surprising, we believe that these are the most readable representations available. What we could show in our experiments is that there is no significant penalty in terms of performance, which was less clear. Indeed, for the semi-stable semantics the more readable encoding actually also seems to be computationally better with the compared tools, which is perhaps surprising.

[2]Folder 2019 in https://argumentationcompetition.org/2021/instances.tar.gz

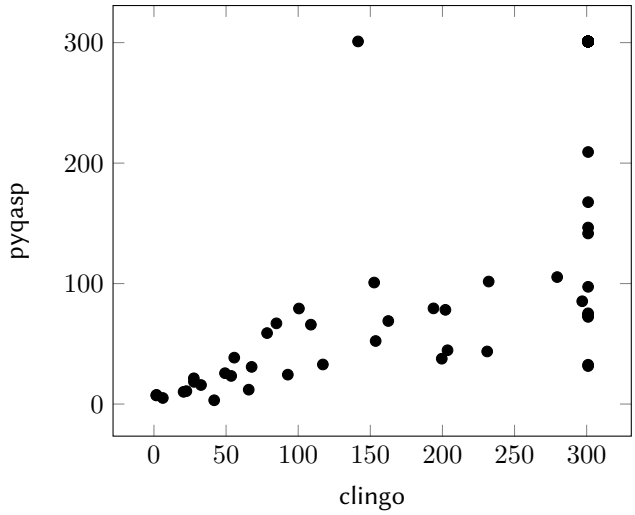

Figure 1: Semi-stable extensions scatter plot

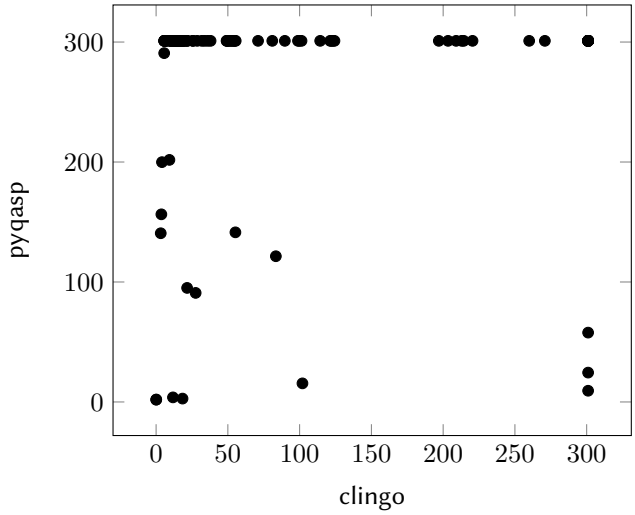

Figure 2: Stage extensions scatter plot

We believe that this can open the ground for a flexible tool similar to ASPARTIX, which can allow for rapid prototyping for many variations and extensions of argumentation frameworks.

Acknowledgments

This research was funded in part by the Austrian Science Fund (FWF) projects 10.55776/PIN8782623 and 10.55776/COE12.

References

[1] P. M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, Artif. Intell. 77 (1995) 321–358. URL: https://doi.org/10.1016/0004-3702(94)00041-X. doi:10.1016/0004-3702(94)00041-X.

[2] U. Egly, S. A. Gaggl, S. Woltran, ASPARTIX: implementing argumentation frameworks using answer-set programming, in: M. G. de la Banda, E. Pontelli (Eds.), Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings, volume 5366 of *Lecture Notes*

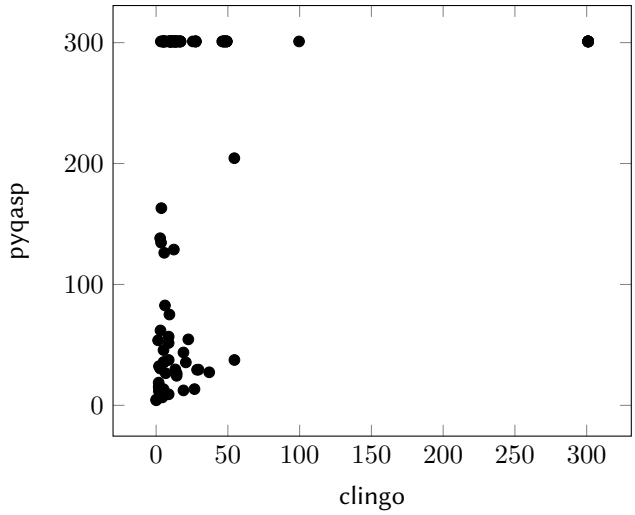

Figure 3: Preferred extensions scatter plot

in Computer Science, Springer, 2008, pp. 734–738. URL: https://doi.org/10.1007/978-3-540-89982-2_67. doi:10.1007/978-3-540-89982-2_67.

[3] G. Amendola, B. Cuteri, F. Ricca, M. Truszczynski, Solving problems in the polynomial hierarchy with ASP(Q), in: G. Gottlob, D. Inclezan, M. Maratea (Eds.), Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings, volume 13416 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 373–386. URL: https://doi.org/10.1007/978-3-031-15707-3_29. doi:10.1007/978-3-031-15707-3_29.

[4] M. Caminada, Semi-stable semantics, in: P. E. Dunne, T. J. M. Bench-Capon (Eds.), Computational Models of Argument: Proceedings of COMMA 2006, September 11-12, 2006, Liverpool, UK, volume 144 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2006, pp. 121–130. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=1932.

[5] B. Verheij, Two approaches to dialectical argumentation: Admissible sets and argumentation stages, in: Proc. NAIC, 1996, pp. 357–368.

[6] W. Dvorák, S. A. Gaggl, J. P. Wallner, S. Woltran, Making use of advances in answer-set programming for abstract argumentation systems, in: H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, A. Wolf (Eds.), Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers, volume 7773 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 114–133. URL: https://doi.org/10.1007/978-3-642-41524-1_7. doi:10.1007/978-3-642-41524-1_7.

[7] G. Amendola, F. Ricca, M. Truszczynski, Beyond NP: quantifying over answer sets, Theory Pract. Log. Program. 19 (2019) 705–721. URL: https://doi.org/10.1017/S1471068419000140. doi:10.1017/S1471068419000140.

[8] W. Faber, G. Mazzotta, F. Ricca, An efficient solver for ASP(Q), Theory Pract. Log. Program. 23 (2023) 948–964. URL: https://doi.org/10.1017/s1471068423000121. doi:10.1017/S1471068423000121.

A. ASPARTIX Encodings

For completeness and comparison, we also provide the encodings for preferred, semi-stable, and stage extensions of ASPARTIX[3]

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

A.1. Preferred Extensions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Encoding for preferred extensions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Guess a set S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).

%% The argument x is defeated by the set S
defeated(X) :- in(Y), att(Y,X).

%% The argument x is not defended by S
not_defended(X) :- att(Y,X), not defeated(Y).

%% All arguments x \in S need to be defended by S (admissibility)
:- in(X), not_defended(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For the remaining part we need to put an order on the domain.
% Therefore, we define a successor-relation with infinum and supremum
% as follows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lt(X,Y) :- arg(X),arg(Y), X<Y, not input_error.
nsucc(X,Z) :- lt(X,Y), lt(Y,Z).
succ(X,Y) :- lt(X,Y), not nsucc(X,Y).
ninf(X) :- lt(Y,X).
nsup(X) :- lt(X,Y).
inf(X) :- not ninf(X), arg(X).
sup(X) :- not nsup(X), arg(X).


%% Guess S' \supseteq S
inN(X) :- in(X).
inN(X) | outN(X) :- out(X).

%% If S' = S then spoil.
%% Use the sucessor function and check starting from supremum whether
%% elements in S' is also in S. If this is not the case we "stop"
%% If we reach the supremum we spoil up.

% eq indicates whether a guess for S' is equal to the guess for S

eq_upto(Y) :- inf(Y), in(Y), inN(Y).
eq_upto(Y) :- inf(Y), out(Y), outN(Y).

eq_upto(Y) :- succ(Z,Y), in(Y), inN(Y), eq_upto(Z).
```

```
eq_upto(Y) :- succ(Z,Y), out(Y), outN(Y), eq_upto(Z).

eq :- sup(Y), eq_upto(Y).


%% get those X \notin S' which are not defeated by S'
%% using successor again...

undefeated_upto(X,Y) :- inf(Y), outN(X), outN(Y).
undefeated_upto(X,Y) :- inf(Y), outN(X),  not att(Y,X).

undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z), outN(Y).
undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z), not att(Y,X).

undefeated(X) :- sup(Y), undefeated_upto(X,Y).

%% spoil if the AF is empty
not_empty :- arg(X).
spoil :- not not_empty.

%% spoil if S' equals S for all preferred extensions
spoil :- eq.

%% S' has to be conflict-free - otherwise spoil
spoil :- inN(X), inN(Y), att(X,Y).

%% S' has to be admissible - otherwise spoil
spoil :- inN(X), outN(Y), att(Y,X), undefeated(Y).

inN(X) :- spoil, arg(X).
outN(X) :- spoil, arg(X).

%% do the final spoil-thing ...
:- not spoil.

%in(X)?
#show in/1.
```

A.2. Semi-stable Extensions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Encoding for semi-stable extensions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Guess a set S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).
```

```
%% The argument x is defeated by the set S
defeated(X) :- in(Y), att(Y,X).

%% The argument x is not defended by S
not_defended(X) :- att(Y,X), not defeated(Y).

%% All arguments x \in S need to be defended by S (admissibility)
:- in(X), not_defended(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For the remaining part we need to put an order on the domain.
% Therefore, we define a successor-relation with infinum and supremum
% as follows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lt(X,Y) :- arg(X),arg(Y), X<Y, not input_error.
nsucc(X,Z) :- lt(X,Y), lt(Y,Z).
succ(X,Y) :- lt(X,Y), not nsucc(X,Y).
ninf(X) :- lt(Y,X).
nsup(X) :- lt(X,Y).
inf(X) :- not ninf(X), arg(X).
sup(X) :- not nsup(X), arg(X).

%% Guess S' \supseteq S for semi-stable
inN(X) | outN(X) :- arg(X), not input_error.

% eqplus checks wheter S'+ equals S+
eqplus_upto(Y) :- inf(Y), in(Y), inN(Y).
eqplus_upto(Y) :- inf(Y), in(Y), inN(X), att(X,Y).
eqplus_upto(Y) :- inf(Y), in(X), inN(Y), att(X,Y).
eqplus_upto(Y) :- inf(Y), in(X), inN(Z), att(X,Y), att(Z,Y).
eqplus_upto(Y) :- inf(Y), out(Y), outN(Y), not defeated(Y), undefeated(Y).
eqplus_upto(Y) :- succ(Z,Y), in(Y), inN(Y), eqplus_upto(Z).
eqplus_upto(Y) :- succ(Z,Y), in(Y), inN(X), att(X,Y), eqplus_upto(Z).
eqplus_upto(Y) :- succ(Z,Y), in(X), inN(Y), att(X,Y), eqplus_upto(Z).
eqplus_upto(Y) :- succ(Z,Y), in(X), inN(U), att(X,Y), att(U,Y), eqplus_upto(Z).
eqplus_upto(Y) :- succ(Z,Y), out(Y), outN(Y), not defeated(Y), undefeated(Y), eqplus_upto(Z

eqplus :- sup(Y), eqplus_upto(Y).

%% get those X \notin S' which are not defeated by S'
%% using successor again...

undefeated_upto(X,Y) :- inf(Y), outN(X), outN(Y).
undefeated_upto(X,Y) :- inf(Y), outN(X),  not att(Y,X).

undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z), outN(Y).
undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z), not att(Y,X).

undefeated(X) :- sup(Y), undefeated_upto(X,Y).

%% spoil if the AF is empty
```

```
not_empty :- arg(X).
spoil :- not not_empty.


%% spoil if S'+ equals S+
spoil :- eqplus.

%% S' has to be conflictfree - otherwise spoil
spoil :- inN(X), inN(Y), att(X,Y).

%% spoil if not semi-stable
spoil :- inN(X), outN(Y), att(Y,X), undefeated(Y).
spoil :- in(X), outN(X), undefeated(X).
spoil :- in(Y), att(Y,X), outN(X), undefeated(X).

inN(X) :- spoil, arg(X).
outN(X) :- spoil, arg(X).

%% do the final spoil-thing ...
:- not spoil.
```

A.3. Stage Extensions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Encoding for stage extensions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Guess a set S \subseteq A
in(X) :- not out(X), arg(X).
out(X) :- not in(X), arg(X).

%% S has to be conflict-free
:- in(X), in(Y), att(X,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% For the remaining part we need to put an order on the domain.
% Therefore, we define a successor-relation with infinum and supremum
% as follows
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

lt(X,Y) :- arg(X),arg(Y), X<Y, not input_error.
nsucc(X,Z) :- lt(X,Y), lt(Y,Z).
succ(X,Y) :- lt(X,Y), not nsucc(X,Y).
ninf(X) :- lt(Y,X).
nsup(X) :- lt(X,Y).
inf(X) :- not ninf(X), arg(X).
sup(X) :- not nsup(X), arg(X).

%% Computing the range S+ of the guessed set S
in_range(X) :- in(X).
in_range(X) :- in(Y), att(Y,X).
```

```
not_in_range(X) :- arg(X), not in_range(X).

%% Guess S' \supseteq S for semi-stable
inN(X) | outN(X) :- arg(X), not input_error.

% eqplus checks wheter S'+ equals S+
eqplus_upto(X) :- inf(X), in_range(X), in_rangeN(X).
eqplus_upto(X) :- inf(X), not_in_range(X), not_in_rangeN(X).
eqplus_upto(X) :- succ(Z,X), in_range(X), in_rangeN(X), eqplus_upto(Z).
eqplus_upto(X) :- succ(Z,X), not_in_range(X), not_in_rangeN(X), eqplus_upto(Z).

eqplus :- sup(X), eqplus_upto(X).

%% get those X \notin S' which are not defeated by S'
%% using successor again...

undefeated_upto(X,Y) :- inf(Y), outN(X), outN(Y).
undefeated_upto(X,Y) :- inf(Y), outN(X),  not att(Y,X).

undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z), outN(Y).
undefeated_upto(X,Y) :- succ(Z,Y), undefeated_upto(X,Z), not att(Y,X).

not_in_rangeN(X) :- sup(Y), outN(X), undefeated_upto(X,Y).
in_rangeN(X) :- inN(X).
in_rangeN(X) :- outN(X), inN(Y), att(Y,X).

%% fail if the AF is empty
not_empty :- arg(X).
fail :- not not_empty.

%% S' has to be conflictfree - otherwise fail
fail :- inN(X), inN(Y), att(X,Y).

%% fail if S'+ equals S+
fail :- eqplus.

%% fail if S'+ \subset S+
fail :- in_range(X), not_in_rangeN(X).

inN(X) :- fail, arg(X).
outN(X) :- fail, arg(X).

%% do the final spoil-thing ...
:- not fail.
```