Extending the DIAMOND System to work with GRAPPA

Matti BERTHOLD
Computer Science Institute, Leipzig University, Germany

Abstract The subject of this paper is to extend the DIAMOND system, which analyses Abstract Dialectical Frameworks (ADFs), such that it will be able to work with a variant of ADFs called GRAPPA (GRaph-based Argument Processing with Patterns of Acceptance). The implementation uses answer set programming encodings to analyze acceptance patterns of GRAPPA Systems semantically.

Keywords. abstract argumentation, abstract dialectical frameworks, DIAMOND, answer set programming, clingo, potassco, implementation

1. Introduction

Argumentation is the interdisciplinary study of finding conclusions to premises by logical reasoning. Computer scientists are concerned with automating this process in a manner that works quickly and for a wide array of instances. Before developing an algorithm to solve a problem though, one has to come up with a fitting representation. In 1995 Phan Minh Dung introduced Argumentation Frameworks (AFs) [1]. They constitute a milestone in the field of argumentation because they are the first abstract argumentation formalism, very intuitive to use and – as shown in the initial paper – they are capable to express established mathematical problems such as the stable marriage problem. AFs are graphs, whose nodes depict arguments. Edges between arguments represent a binary attack relation ($aRb :\Leftrightarrow$ a attacks b). Arguments are said to be true, if none of their parents are. Since AFs were introduced, there have been many suggestions for formalisms that cover a wider array of problem instances. Abstract Dialectical Frameworks (ADFs) [2] are one of them. They feature arbitrary acceptance conditions for nodes, which make it possible to model support and weak attacks between arguments. Semantics of AF have successfully been generalized for ADFs. The complexity of finding models on those abstracted semantics has been examined in detail [3]. The observations of this examination are promising, because bipolar ADFs (which are special ADFs) are in the same complexity class as AFs, while still offering more expressiveness.

Labelled graphs are an often used model in Artificial Intelligence (for example Bayes Nets of probabilistic reasoning). In 2014 Brewka and Woltran introduced a variant of ADFs – GRAPPA (GRaph-based Argument Processing with Patterns of Acceptance) [4] – to bring established argumentation semantics to labeled graphs.

The DIAMOND System is a collection of tools to calculate models of ADFs and variations of them [5]. The subject of this work is to extend DIAMOND to support GRAPPA inputs. For this matter we will first define an input format to represent GRAPPA Systems that is suited to be interpreted by Answer Set Programming (ASP) and easy to use. Secondly we will introduce algorithms that process this input. Our approach is different to GRAPPAVIS [6] – the only other software system that is able to process GRAPPA inputs. Instead of implementing native algorithms to find models of GRAPPA Systems we will rely on working ADF implementations in DIAMOND. To do this we will convert GRAPPA Systems into ADFs in a format that is accepted by DIAMOND. As DIAMOND is subject of active development (as seen in [7]) this approach will vastly reduce the need for maintenance. Any improvement affecting ADF-implementations will directly benefit calculations on GRAPPA Systems. If for example DIAMOND is extended to be capable of finding models of new semantics on ADFs, there is no need to implement the algorithm again for GRAPPA Systems.

This paper features a working algorithm that functions as a proof of concept as well as suggestions for improvements to it.

2. Preliminaries

For subsequent algorithms the background will first be set with intuitions and some formal definitions of ADFs and GRAPPA Systems (Section 2.1), Answer Set Programming and DIAMOND (Section 2.2). Definitions in Section 2.1 are heavily based on [2] and [4]. Some concepts have purposely been rephrased to offer an easier overview. In particular the definitions of ADFs have been brought in line with GRAPPA, to make their similarities more obvious. The definition of GRAPPA acceptance patterns has been changed to fit our implementation better. For more formal details we direct interested readers to the original sources.

2.1. Formal Background

ADFs are argumentation graphs, whose nodes depict arguments with fully arbitrary acceptance conditions, represented by propositional formulas.

Definition 1. An Abstract Dialectical Framework is a tuple $D = (S, E, \pi)$ where

- S is a set of nodes (statements),
- $E \subseteq S \times S$ is a set of edges (dependencies),
- $\pi : S \to FOR$ assigns propositional formulas to nodes as acceptance condition. Acceptance formulas of nodes contain all parents of the respective node and only those parents as atoms.

Valuations on ADFs are partial functions that map nodes to $true$ or $false$. The option to assign no truth value to a node may be used to model missing knowledge. We will represent valuations conveniently as sets of literals containing elements of S which are evaluated to *true* unnegated and those evaluated to $false$ negated.

Various semantics capture different intuitions of what makes a valuation sensible. Most of them are defined on the characteristic operator Γ, which is a function that maps valuations to valuations. Intuitively it calculates the acceptance condition of every node in every extension of its input. A valuation e *extends* another valuation v, if e is total and $v \subseteq e$. If there is consensus about the truth value tv of a node s between all extensions of a valuation v, $\Gamma(v)(s) = tv$, otherwise $\Gamma(v)(s)$ is undefined.

Definition 2. Let $D = (S, E, \pi)$ be an ADF and v a valuation on D. We say

- v is *admissible* in D iff $v \subseteq \Gamma_D(v)$,
- v is *complete* in D iff $v = \Gamma_D(v)$,
- v is *grounded* in D iff v is the least fixed point of Γ_D ,
- v is *preferred* in D iff v is $\subseteq$-maximal admissible in D,
- v is a *model* of D iff v is total and $v = \Gamma_D(v)$,
- v is a *stable model* of D iff v is a model of D and v restricted to $S^v(= v \cap S)$ is the grounded valuation of $D^v = (S^v, E^v, \pi^v)$, the v-reduct of D, where $E^v = E \cap (S^v \times S^v)$, π^v is π restricted to S^v.

GRAPPA Systems were introduced, to bring ADF-semantics to labeled argumentation graphs. Their only difference to ADFs is that their edges have labels which are incorporated in acceptance conditions of nodes [4].

Definition 3. A GRAPPA is a tuple $G = (S, E, L, \lambda, \pi)$ where

- S is a set of nodes (statements),
- E is a set of edges (dependencies),
- L is a set of labels,
- $\lambda : E \to L$ assigns labels to edges,
- $\pi : S \to P^L$ assigns acceptance patterns over L to nodes

Definitions of valuations and semantics on GRAPPA can be directly adopted from ADFs. We will call an edge active in a valuation if its origin is true.

The language of acceptance patterns is specifically designed to make quantitative statements about labels of edges entering nodes. Such a statement may read "There are more than 3 active '+'-links" or "Less than half of all '−'-links are active". For labels that are integers we can calculate the sum or the least and greatest element to make statements such as "The sum of all labels is greater than 0.". Because acceptance patterns of nodes are tied to labels, several nodes might receive the same pattern though they have totally different parents.

Definition 4. Let L be a set of labels.

- A GRAPPA term over L is of the form:
  min, min_t, max, max_t, sum, sum_t, $count$, $count_t$ or $(\#l)$, $(\#_t l)$ for arbitrary $l \in L$
- A term (over L) is a GRAPPA term, a constant integer or any arithmetic combination of the two using the connectors $+$, $-$, $*$, div, mod, exp
- A basic acceptance pattern[1] (over L) is of the form $term_1 \odot term_2$ with $\odot \in \{<, \leq, =, \neq, \geq, >\}$ or one of the constants $\bot$ (*false*) or $\top$ (*true*)
- An *acceptance pattern* (over L) is a basic acceptance pattern or a Boolean combination of acceptance patterns.

The value of GRAPPA terms is determined as follows:

Definition 5. Let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA System. For $m : L \to \mathbb{N}$ and $s \in S$ the value function val_s^m is defined as:

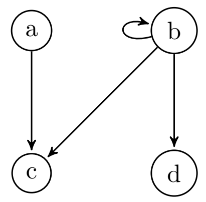
$$\begin{aligned}
val_s^m(\#l) &= m(l) & val_s^m(\#_t l) &= |\{(e,s) \in E \mid \lambda((e,s)) = l\}| \\
val_s^m(min) &= min_m & val_s^m(min_t) &= min\{\lambda((e,s)) \mid (e,s) \in E\} \\
val_s^m(max) &= max_m & val_s^m(max_t) &= max\{\lambda((e,s)) \mid (e,s) \in E\} \\
val_s^m(sum) &= \Sigma_{l \in L} m(l) & val_s^m(sum_t) &= \Sigma_{(e,s) \in E} \lambda((e,s)) \\
val_s^m(count) &= |\{l \mid m(l) > 0\}| & val_s^m(count_t) &= |\{\lambda((e,s)) \mid (e,s) \in E\}|
\end{aligned}$$

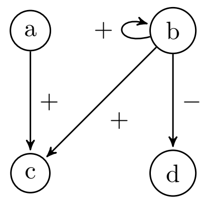
Here m can assign any integer value to labels. Since we are interested in truth values of nodes in particular valuations m is usually determined by the number of active links of a valuation.
Values of terms and patterns are determined inductively as one might expect:

$$\begin{aligned}
val_s^m(term_1 + term_2) &= val_s^m(term_1) + val_s^m(term_2) \\
val_s^m(term_1 - term_2) &= val_s^m(term_1) - val_s^m(term_2) \\
&\vdots \\
val_s^m(pattern_1 \wedge pattern_2) &= val_s^m(pattern_1) \wedge val_s^m(pattern_2) \\
val_s^m(pattern_1 \vee pattern_2) &= val_s^m(pattern_1) \vee val_s^m(pattern_2) \\
&\vdots
\end{aligned}$$

[1]The definition of basic acceptance patterns here has been adapted to fit the GRAPPA representation in our implementation better. It has to be noted that it is more general than in [4]. Previously it was not possible to connect two GRAPPA terms directly with operators other than $+$ or $-$

Example 1
These are an ADF and a GRAPPA System representing the same argumentation structure.





$\pi(a) = \top \qquad \pi(b) = b \qquad \forall s \in S : \pi(s) = (\#{+} = \#_t{+}) \wedge (\#{-} = 0)$

$\pi(c) = a \wedge b \qquad \pi(d) = \neg b$

2.2. Technical Background

Answer set programming (ASP) is a declarative programming language designed to represent and solve computational problems easily [8]. It uses logical implications (*rules*) to deduce models. Rules that contain no premises are *facts*. We will use the phrase "rule" only to refer to implications with premises.
It is hard to draw a line between a program and its input, because ASP makes no difference between the two. What we consider a program usually is a set of implications that predominantly consists of rules. We will use a set of facts as a program's input. Models of the aggregation of a program and its input are sets of facts as well and function as output.

This paper discusses algorithms detached from their implementation. Thus there is no need to define the syntax and semantics of ASP formally, though we strongly suggest to read about them in [8]. However here is a quick overview of the syntax of facts: Facts in ASP are *atoms* or *predicates*. Atoms are strings that start with a lowercase letter, predicates look the same, but are followed by parentheses that contain the predicates arguments which may also be atoms. Here are some examples:

```
it_is_raining.
is_mortal(socrates).
```

Facts are confined by dots. Predicates also take *functions* as arguments, which again are lowercase strings with parentheses that contain arguments. For example:

```
is_mortal(father_of(socrates)).
```

This rudimentary introduction to the syntax of facts is sufficient to explain the inputs of the DIAMOND System.

The DIAMOND system is a collection of tools that are aimed at computing models of ADFs. It is based on ASP and offers five different input formats to define regular or special kinds of ADFs. The two formats that define regular ADFs are: propo-

sitional formula representation, extensional representation. Both formats use the binary predicate `l/2` to depict edges of ADFs.
The formula representation uses the binary predicate `ac(N,`ϕ`)` to associate nodes `N` with their acceptance formula ϕ. ϕ is any logical combination (using the connectives $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, $\oplus$, $\neg$) of parent nodes and the constants $\top$ and $\bot$ (represented by `c(v)` and `c(f)`).
In the extensional format the truth value of every valuation is stated explicitly with the help of `ci/1`, `co/1`, `ci/3`, `co/3`. The predicates `ci/1` or `co/1` are used to describe whether a node is true if none of its parents are. A number of `ci/3` or `co/3` predicates that use the same arbitrary index as second argument describe all the other valuations. The way they work is best seen in an example.

Example 2
These two code examples (propositional representation on the left, extensional representation on the right) are two ways to describe the example ADF of Section 2.1 in DIAMOND.

```
l(a,c). l(b,c). l(b,b). l(b,d).
ac(a,c(v)).
ac(b,b).
ac(c,and(a,b)).
ac(d,neg(b)).
```

```
l(a,c). l(b,c). l(b,b). l(b,d).
ci(a). co(b). ci(b,0,b).
co(c). co(c,1,a). co(c,2,b).
ci(c,3,a). ci(c,3,b).
ci(d). co(d,1,b).
```

3. Transformations

Subsequently we will introduce the new GRAPPA representation and processes[2] that convert it to an input format of DIAMOND. The algorithm in Section 3.2 purposely includes a minimal amount of modifications that improve running times. It is a proof of concept and sets an upper bound for complexity.
In Section 3.3 we will introduce variants of this algorithm that are aimed at reducing running times. An empirical evaluation of running times and comparisons to competing systems are subject of future work, as the implementation of these improvements are still in development.

3.1. The Input Language

The new GRAPPA representation closely resembles the propositional formula representation for ADFs. Its advantage is that acceptance conditions can be described by a single formula tree that fits into a single predicate.

Nodes can explicitly be defined by `node/1` which is only necessary if there is no link entering or leaving the node. Links between nodes are defined by `l/3` where the the first argument denotes the parent node, the third argument denotes the node the link is entering and the second argument represents the link's label. Acceptance patterns are assigned to nodes with the help of predicate `gac/2` (GRAPPA

[2]Implementations can be viewed at:
`https://sourceforge.net/p/diamond-adf/grappa/ref/master/`

acceptance condition), where the first argument denotes the node and the second argument the respective acceptance pattern. The language of acceptance patterns is described by a context free grammar in Figure 1.

```
<gac>        ::= gac(<node id>,<sf>).
<sf>         ::= and(<sf>,<sf>) | or(<sf>,<sf>) |
                 imp(<sf>,<sf>) | iff(<sf>,<sf>) |
                 xor<sf>,<sf>) | neg(<sf>) |
                 c(v) | c(f)
<sf>         ::= eq(<t>,<t>) | neq(<t>,<t>) |lt(<t>,<t>) |
                 lte(<t>,<t>) | gt(<t>,<t>) | gte(<t>,<t>)
<t>          ::= plus(<t>,<t>) | minus(<t>,<t>) | times(<t>,<t>) |
                 div(<t>,<t>) | mod(<t>,<t>) | exp(<t>,<t>) |
                 c(<whole number>) | <grappaterm>
<grappaterm> ::= count(<label>) | count_t(<label>) | count |
                 count_t | sum | sum_t | max | max_t | min | min_t
<node id>    ::= <arbitrary (lower case) string (without spaces)>
<label>      ::= <arbitrary (lower case) string (without spaces)>
```

Figure 1. The language of acceptance patterns given as context free grammar

Example 3
The following knowledge base describes the GRAPPA System that was given as an example in Section 2.1

```
l(a,plus,c). l(b,plus,b). l(b,plus,c). l(b,minus,d).
gac(a,and(eq(count(plus),count_t(plus)),eq(count(minus),c(0))).
gac(b,and(eq(count(plus),count_t(plus)),eq(count(minus),c(0))).
gac(c,and(eq(count(plus),count_t(plus)),eq(count(minus),c(0))).
gac(d,and(eq(count(plus),count_t(plus)),eq(count(minus),c(0))).
```

3.2. A Full Semantic Analysis

The DIAMOND format we are going to interface with is the extensional representation. To generate an equivalent ADF in this format all we have to do is to calculate the truth value of acceptance patterns.
GRAPPA acceptance patterns are extensional (the value of a pattern is determined by the value of its subpatterns). Hence to calculate the truth value of a pattern, we can resort to classical divide and conquer tactics – break the pattern down into its components and calculate the value one subpattern at a time. To solve an acceptance pattern for a specific valuation, we start at the smallest subpattern, find out its value and work our way up until we find out the value of the main formula.

Before going up the pattern tree in this manner, we need to break the pattern down, to form a pattern tree in the first place. Doing this, we need to consider the kind of value that the subpatterns will hold later. Subformulas hold truth values (true/false) and terms hold integer values. For this reason, both kinds are represented by two different predicates `gacSF/2` (gac subformula) and `gacT/2` (gac term). Their first argument describes the node they belong to and the second one their actual form. Subpatterns are broken down into subformulas as long as their main connectives are logical, subformulas whose main connectives are comparisons are broken down into two terms and terms are divided until atomic terms are reached.

Example 4
These are the predicates representing the pattern tree for node `d` in our example:

```
gacSF(c,and(eq(count(plus),count_t(plus)),eq(count(minus),c(0)).
gacSF(c,eq(count(plus),count_t(plus))).
gacT(c,count(plus)).                      gacT(c,count_t(plus)).
gacSF(c,eq(count(minus),c(0)).
gacT(c,count(minus)).                     gacT(c,c(0)).
```

For a node with n parents all 2^n valuations are generated, represented by `active/3` predicates. For valuation 0 there are no such predicates, for valuation 1 there is one of them, representing the first parent and so on. For valuation $2^n - 1$ there are n of them representing each node. More generally, for each valuation m with $0 \leq m \leq 2^n - 1$, the representation of m in binary, $(m)_2$ indicates exactly which parents of the node are active, namely those whose bit is set in $(m)_2$.

Example 5
Continuing our running example, the following predicates are generated to represent the valuations of node `c`:

```
active(a,1,c).   active(b,2,c).   active(a,3,c).   active(b,3,c).
```

The way up the pattern tree begins at the atom level, which are GRAPPA functions or constants. GRAPPA functions are implemented with their matching aggregation statements in ASP. They use `active/3` predicates to determine the value of the GRAPPA terms in each valuation and put them into `gacTV/4` predicates (v for value).

Example 6
The acceptance pattern's atoms of node `c` in valuation `2` receive the following values:

```
gacTV(c,2,count_t(plus),2).        gacTV(c,2,count(plus),1).
gacTV(c,2,count(minus),0).         gacTV(c,2,c(0),0).
```

After calculating the value of all GRAPPA term the subpatterns are put back together respecting the valuation specific value. Each time there is a subpattern predicate `gacT/3` or `gacSF/3` of a non-atomic subpattern `SP` and a value holding predicate `gacTV/4` or `gacSFV/4` for both children of `SP` a value holding predicate for `SP` respecting the arithmetic or truth function of the main connective of `SP` is generated. For example the connective `and/2` is implemented in this way: If both children of the `and/2` connective hold value `true`, their conjunction will as well.

If at least one child holds value `false`, the conjunction will be false as well. The other connectives are handled analogously.

Example 7
This process generates the following new predicates for node `c`, valuation `2` in our example:

```
gacSFV(c,2,eq((count(plus),count_t(plus)),false).
gacSFV(c,2,eq((count(minus),c(0)),true).
gacSFV(c,2,
  and(eq((count(plus),count_t(plus)),eq((count(minus),c(0)),false).
```

After the pattern is put back together to its original form, the output for the DIAMOND System is generated. For each valuation there exists a number of `active/3` predicates. Depending on the truth value of the overall pattern under a respecting valuation, for each of those `active/3` predicates a `co/3` or `ci/3` predicate is generated. For the first valuation there is no `active/3` predicate (because all parents are not active), so the parent node independent predicate `co/1` or `ci/1` is generated.

Example 8
Node `c` is `false` in valuation `2`, where there is one active node: `a`, thus the following output is generated:

```
co(c,2,a).
```

The calculation of each node's models is done entirely hermetically. Hence we can consider which factors scale the running time for each node independent of all the other nodes. For the complexity there are two factors to be considered. On one hand the running time scales with the length of acceptance patterns, on the other with the number of parent nodes. For each connective in the acceptance pattern there is one more subpattern that needs to be looked at on the way down and up the pattern tree, i.e. the running time scales *linearly* with the number of connectives of the acceptance pattern. For n parent nodes 2^n valuations are generated and evaluated, i.e. running time scales *exponentially* with the number of parent nodes. Brewka and Woltrand observed a similar blowup in their transformation from GRAPPA Systems to ADFs [4].

3.3. Improvements – A Partial Semantic Analysis

In the algorithm of Section 3.2 for each possible valuation of a node the value of the overall pattern was calculated, to explicitly transfer all models to DIAMOND. This approach is very complex, as the number of valuations scales exponentially with the number of parent nodes. For some instances we might be able to reduce the complexity significantly. Consider the following example:

```
l(p1,plus,a).  l(p2,plus,a).  ... l(p50,plus,a).
l(m1,minus,a). l(m1,minus,a). ... l(m50,minus,a).
gac(a,and(eq(count(plus),count_t(plus)),eq(count(minus),c(0))).
```

a has 100 parents. A full semantic analysis evaluates 2^{100} valuations while the first and second conjunct actually are entirely independent of each other – they depend on disjoint sets of nodes. Separating the evaluation into two groups would dramatically reduce the number of valuations to be considered to $2^{50}+2^{50}$ $(= 2^{51})$.

In order to achieve this separation we do the following: We apply the previous algorithm on each of the basic acceptance patterns (BAP) separately under consideration which parent nodes they depend on. A BAP depends on the aggregation of parents its leaves depend on: `count` depends on all parents. `count(l)` depends on all `l`-parents. `min`, `max` and `sum` depend on parents whose links have integers as labels. The terms `count_t(l)`, `count_t`, `min_t`, `max_t` and `sum_t` have no dependencies, as their values are fixed.

An implementation[3] that generates propositional formulas in DNF from this information has been tested and observed to run into some obstacles. ASP does not perform well on deep predicates, which is exactly what needs to be generated when constructing a DNF. As using the extensional interface is no option either, all chances to feed the information received from this process into DIAMOND are depleted. The conclusion of this paper will feature a number of suggestions for future input formats of DIAMOND one of which is especially suited to handle this problem.

4. Conclusion and Future Work

This paper offers a new approach to analyze GRAPPA Systems – to rely on existing algorithms that solve ADFs. The algorithms introduced in this paper are a first step toward a fast conversation from GRAPPA System to ADFs. The first one (a full semantic analysis) is designed as a proof of concept and has a working implementation. The second one (a partial semantic analysis) which is based on the first one, is a suggestion on how to reduce the complexity. To make use of it we suggest an input format similar to the propositional formula format, where the formula tree of a acceptance formula is represented by several predicates that are flat. A similar variant of the here newly defined GRAPPA format should also be added.

Furthermore it should be a goal to adapt principles of special ADFs to GRAPPA Systems – bipolar GRAPPA representation and priority based GRAPPA representation being two of them. An implementation that converts them to their ADF-counterparts would complement the DIAMOND System.

Adding interfaces to DIAMOND that use the same language as GRAPPAVIS would offer a common ground for benchmarks.

[3]It can also be found at: `https://sourceforge.net/p/diamond-adf/grappa/ref/master/`

References

[1] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.

[2] Gerhard Brewka, Hannes Strass, Stefan Ellmauthaler, Johannes Peter Wallner, and Stefan Woltran. Abstract dialectical frameworks revisited. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

[3] Hannes Strass and Johannes Peter Wallner. Analyzing the computational complexity of abstract dialectical frameworks via approximation fixpoint theory. *Artif. Intell.*, 226:34–74, 2015.

[4] Gerhard Brewka and Stefan Woltran. GRAPPA: A semantical framework for graph-based argument processing. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 153–158. IOS Press, 2014.

[5] Stefan Ellmauthaler and Hannes Strass. The DIAMOND system for computing with abstract dialectical frameworks. In Simon Parsons, Nir Oren, Chris Reed, and Federico Cerutti, editors, *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014*, volume 266 of *Frontiers in Artificial Intelligence and Applications*, pages 233–240. IOS Press, 2014.

[6] Georg Heißenberger. A system for advanced graphical argumentation formalisms. Master's thesis, TU Wien, 2016.

[7] Stefan Ellmauthaler and Hannes Strass. DIAMOND 3.0 – A native C++ implementation of DIAMOND. In Pietro Baroni, editor, *Proceedings of the Sixth International Conference on Computational Models of Argument (COMMA)*, Frontiers in Artificial Intelligence and Applications, Potsdam, Germany, September 2016. IOS Press.

[8] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[9] Stephen C. Kleene. *Introduction to Metamathematics.* D. Van Nostrand Company, Princeton, New Jersey, 1952.