Utilizing ASP for Generating and Visualizing Argumentation Frameworks

Günther Charwat, Johannes Peter Wallner, and Stefan Woltran

Vienna University of Technology, Institute of Information Systems 184/2,
Favoritenstraße 9-11, 1040 Vienna, Austria
`{gcharwat,wallner,woltran}@dbai.tuwien.ac.at`

Abstract. Within the area of computational models of argumentation, the instantiation-based approach is gaining more and more attention, not at least because meaningful input for Dung's abstract frameworks is provided in that way. In a nutshell, the aim of instantiation-based argumentation is to form, from a given knowledge base, a set of arguments and to identify the conflicts between them. The resulting network is then evaluated by means of extension-based semantics on an abstract level, i.e. on the resulting graph. While several systems are nowadays available for the latter step, the automation of the instantiation process itself has received less attention. In this work, we provide a novel approach to construct and visualize an argumentation framework from a given knowledge base. The system we propose relies on Answer-Set Programming and follows a two-step approach. A first program yields the logic-based arguments as its answer-sets; a second program is then used to specify the relations between arguments based on the answer-sets of the first program. As it turns out, this approach not only allows for a flexible and extensible tool for instantiation-based argumentation, but also provides a new method for answer-set visualization in general.

1 Introduction

Instantiation-based argumentation [7] is a central paradigm in nonmonotonic reasoning since it gives a formal handle to separate the logical and non-classical contents of reasoning in the presence of contradicting information. Hereby, one starts with a knowledge base and constructs arguments from it. Arguments typically consist of two parts, namely a support, which is grounded in the knowledge base and a claim derived from it. In [4] the process is described with an underlying propositional knowledge base using minimal sets of consistent support classically entailing the claim. In a second step, conflicts between these arguments have to be identified. The obtained arguments and the relation between them yield a so-called argumentation framework [9]. This simple, yet expressive formalism is basically a directed graph whereby the arguments are represented via vertices and the conflicts with directed edges. Argumentation frameworks are then evaluated with one of the numerous semantics for abstract argumentation available, resulting in potentially multiple acceptable sets of arguments [3].

Here we are only interested in the instantiation part, however, which received less attention wrt. realized systems. Notable exceptions are the Carneades system, which can construct arguments using heuristics [16] and the recent TOAST implementation

for the ASPIC+ framework [27]. The reason for the lack of implementations is potentially twofold: First, due to the inherent high complexity of the problem; already constructing a single argument is hard for the second level of the polynomial hierarchy [23]. Secondly, standard instantiation schemes for propositional knowledge bases result in infinite argumentation frameworks even for finite knowledge bases [1]. The first obstacle calls for highly expressive languages, making answer-set programming [6, 21, 22] (ASP, for short) a well suited candidate. For the second obstacle, we restrict ourselves here to arguments that have their claims coming from an a priori specified set of formulae.

To summarize, we aim here for a system which takes as input a knowledge base as well as a set of potential claims and produces the instantiated argumentation framework, such that the latter can be processed by other argumentation tools, e.g. ASPARTIX [11] or CEGARTIX [10]. More specifically, our contributions are as follows:

– We provide ASP programs[1] to encode the construction of arguments as well as the construction of the conflicts. For the second task, the answer-sets of the first encoding are used as input. Thus we can make use of the high sophistication modern ASP systems have reached [15, 20]. Moreover, since the argument construction and conflict identification are declaratively described via ASP code, the system is easily adaptable to other notions of arguments or conflicts.
– We present a system that, on the one hand, takes care of passing the answer-sets from one program to another. On the other hand, the system uses the answer-sets of the two programs for visualization in form of a graph. In our case we obtain an argumentation framework. Finally, this result can be exported and used by other systems for abstract argumentation.

As a by-product, we observed that this method is by no means restricted to the argumentation domain. Ultimately, it allows for a user-driven graph representation of the collection of answer-sets of a given input program, thus acting as a tool for ASP visualization in general. The most interesting feature of the tool is that the concrete specification for two answer-sets being in relation is given by an ASP program itself. In recent years, ASP has benefited from the rising number of development and visualization tools, e.g. ASPViz [8], ASPIDE [14], Kara [19] and IDPDraw [29]. These tools so far have focused on presentations of single answer-sets of the given program. However, in certain applications it is not only the single answer-sets which are of interest, but the relation between them.

While visualization is a rather new research branch in ASP, it has gained more attention in the argumentation community, where dedicated visualization tools have been proposed already in the late 90s (e.g. [2, 5, 11, 18, 24–26, 28], including Debategraph[2] and Rationale[3]). Many of these support the argument construction by a user via different means, such as automated reasoning, input masks and database querying. Compared to these systems, our approach combines the computational power of high-sophisticated ASP systems with visualization aspects. Moreover, thanks to the declarative nature of

---

[1] `http://dbai.tuwien.ac.at/proj/argumentation/vispartix/`

[2] `http://debategraph.org`

[3] `http://rationale.austhink.com/`

the ASP encodings specifying the instantiation step, we believe that the strength of our approach lies in its flexibility and its expandability to new argumentation formalisms.

This paper is structured as follows: We briefly introduce argumentation and ASP in Section 2. Then, in Section 3, we present ASP encodings for constructing an argumentation framework. In Section 4 we outline a novel visualization tool which is used for representing relations between answer-sets. A final discussion is given in Section 5.

2   Preliminaries

We provide the necessary background from argumentation theory and ASP for this work. In particular we will explain the argumentation process based on argumentation frameworks [9] as well as briefly recall the concepts for disjunctive logic programs.

2.1   Argumentation

In this section we introduce formal argumentation. We start with the underlying process [7], which we will utilize in our context. The general process consists of three steps. First, given a knowledge base, arguments and their relationships are instantiated. After this instantiation the arguments are treated as abstract entities, without considering their concrete content. Secondly conflicts are resolved using appropriate semantics on the abstract instantiation and finally conclusions are drawn.

In this work the knowledge base $\mathcal{K}$ is a (potentially inconsistent) set of propositional logic formulae. We construct the formulae with the usual connectives $\neg, \vee, \wedge, \rightarrow$, the negation, disjunction, conjunction and implication, respectively. Furthermore entailment and logical equivalence of formulae is denoted by $\models$ and $\equiv$, respectively. We write formulae with lowercase Greek letters $\alpha, \beta, \gamma, ...$

Example 1. Consider the following simple and inconsistent example knowledge base:

$$\mathcal{K} = \{a, a \rightarrow b, \neg b\}$$

The instantiation step now constructs arguments and relations among them based on the information available in $\mathcal{K}$ according to [4]. The abstract representation we utilize for this purpose is the widely studied argumentation framework [9]. An argumentation framework (AF) is a directed graph $F = (Args, Att)$, with the vertices $(Args)$ being abstract arguments and the directed edges (Att) denote attacks between them to represent conflicts.

The instantiation of an AF now consists of two parts, namely the argument construction and the attack relation construction. An argument $A = (S, C)$ consists of a support for the argument and a claim. The support is a subset of the knowledge base $\mathcal{K}$ and the claim is a single logical formula. The support must be a consistent and subset minimal set of formulae, which entails the claim.

Here the arguments are pairs of support and claim to provide a formal basis for argument construction. When plugged in the argumentation framework we abstract from this "inner" structure and collapse every pair of support and claim into one abstract argument. This is the abstraction procedure of the overall process.

We note that argument construction here differs from the usual argument definition in the literature. In particular the claim can be taken only from a pre-defined set $\mathcal{C}$. Using a pre-defined set of claims, we can restrict ourselves to reasonable claims, e.g. not involving tautologies. In this way we prohibit the construction of infinitely many arguments that could otherwise result from infinitely many syntactically different formulae which are semantically equivalent. This restriction comes with a disadvantage however, as the set of pre-defined claims must be chosen with care, since inconsistent conclusions might be drawn otherwise. Indeed, [17] identify conditions for rational and consistent end results, which require the existence of specific arguments, which must be included in $\mathcal{C}$. On the other hand, this restriction is in line with the concept of cores of argumentation frameworks [1], which try to preserve desired properties while using only a subset of all possible arguments.

Example 2. Continuing Example 1, let the set of claims be $\mathcal{C} = \mathcal{K} \cup \{\neg a, b, a \wedge \neg b\}$. Then we can construct the following arguments:

$$a_1 = (\{a\}, a) \qquad\qquad a_4 = (\{\neg b, a \rightarrow b\}, \neg a)$$
$$a_2 = (\{a \rightarrow b\}, a \rightarrow b) \qquad\qquad a_5 = (\{\neg b\}, \neg b)$$
$$a_3 = (\{a, a \rightarrow b\}, b) \qquad\qquad a_6 = (\{a, \neg b\}, a \wedge \neg b)$$

For the construction of the attack relation several options were studied in literature. The basic idea for attacks between arguments underlying all of these options is that some sort of inconsistency occurs between them. We take the attack definitions from [17] and illustrate two types, defeat and directed defeat. An argument $A = (S, C)$ attacks an argument $A' = (S' = \{\phi'_1, ..., \phi'_m\}, C')$ using defeat if $C \models \neg(\phi'_1 \wedge ... \wedge \phi'_m)$. The former directly defeats the latter if $C \models \neg\phi'_i$ for one i, $1 \leq i \leq m$.

Example 3. Continuing Example 2, the AF in Figure 1 illustrates the result using the direct defeat on the arguments built from $\mathcal{K}$ and the claims $\mathcal{C}$. Note that e.g. a_3 and a_5 are not mutually attacking each other, since the claim of a_5 does not entail a negated support formula of a_3.

Fig. 1. Argumentation framework
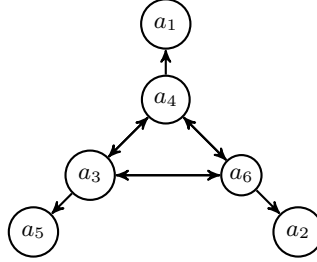
This completes the first step of the argumentation process, namely the AF construction out of the knowledge base. For the conflict resolution a plethora of argumentation framework semantics exist. A basic property for semantics is the conflict-free property, which states that a set M of arguments in an AF F is conflict free if there are no attacks between them in F. A set of arguments M is stable in an AF

$F = (Args, Att)$ if it is conflict free and all arguments outside are attacked from M, i.e. $\forall a \in (Args \setminus M) \; \exists b \in M$ with $(b, a) \in Att$.

Example 4. If we take the argumentation framework from Example 3, then the stable semantics selects $\big\{ \{a_1, a_5, a_6\}, \{a_1, a_2, a_3\}, \{a_2, a_4, a_5\} \big\}$ as acceptable subsets of arguments.

The last step of the argumentation process deals with drawing conclusions from the sets of acceptable arguments. One can look at the content of the abstract arguments which were accepted, e.g. one can derive the deductive closure of this content.

In general every step of this process is intractable. Hence we need sophisticated systems for tackling these steps, which makes ASP a suitable choice for embedding the process in. A more detailed computational complexity analysis can be found in [23].

2.2 Answer-Set Programming

In this section we recall the basics of disjunctive logic programs under the answer-sets semantics [6, 22].

We fix a countable set $\mathcal{U}$ of *(domain) elements*, also called *constants*. An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity $n \geq 0$ and each t_i is either a variable or an element from $\mathcal{U}$. An atom is *ground* if it is free of variables. $B_{\mathcal{U}}$ denotes the set of all ground atoms over $\mathcal{U}$.

A *(disjunctive) rule* r is of the form

$$a_1 \; \vee \; \cdots \; \vee \; a_n \; \leftarrow b_1, \ldots, b_k, \; not\, b_{k+1}, \ldots, \; not\, b_m$$

with $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$, where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms, and "not" stands for *default negation*. The *head* of r is the set $H(r) = \{a_1, \ldots, a_n\}$ and the *body* of r is $B(r) = \{b_1, \ldots, b_k, \; not\, b_{k+1}, \ldots, \; not\, b_m\}$. Furthermore, $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. A rule r is a *constraint* if $n = 0$. A rule r is *safe* if each variable in r occurs in $B^+(r)$. A rule r is *ground* if no variable occurs in r. A *fact* is a ground rule without disjunction and empty body. A program is a finite set of disjunctive rules.

For any program π, let U_π be the set of all constants appearing in π. $Gr(\pi)$ is the set of rules $r\sigma$ obtained by applying, to each rule $r \in \pi$, all possible substitutions σ from the variables in r to elements of U_π. An *interpretation* $I \subseteq B_{\mathcal{U}}$ *satisfies* a ground rule r iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies a ground program π, if each $r \in \pi$ is satisfied by I. A non-ground rule r (resp., a program π) is satisfied by an interpretation I iff I satisfies all groundings of r (resp., $Gr(\pi)$). $I \subseteq B_{\mathcal{U}}$ is an *answer-set* of π iff it is a subset-minimal set satisfying the *Gelfond-Lifschitz reduct* $\pi^I = \{H(r) \leftarrow B^+(r) \mid I \cap B^-(r) = \emptyset, r \in Gr(\pi)\}$.

3 Instantiation-based Argumentation

In this section we provide our ASP encodings for the construction of arguments from a knowledge-base $\mathcal{K}$ and a set $\mathcal{C}$ of claims. As input, each formula in $\mathcal{K}$ and $\mathcal{C}$ is given by the unary predicate $\mathrm{kb}(\cdot)$ and $\mathrm{cl}(\cdot)$, respectively.

Example 5. The input, as given in Example 1 and 2, is specified by:

$$\{\mathrm{kb}(a).\ \mathrm{kb}(\mathrm{imp}(a,b)).\ \mathrm{kb}(\mathrm{neg}(b)).$$
$$\mathrm{cl}(a).\ \mathrm{cl}(\mathrm{imp}(a,b)).\ \mathrm{cl}(\mathrm{neg}(b)).\ \mathrm{cl}(\mathrm{neg}(a)).\ \mathrm{cl}(b).\ \mathrm{cl}(\mathrm{and}(a,\mathrm{neg}(b))).\}$$

First, we introduce the ASP encodings for checking whether a certain variable assignment is a model for a given formula (or not). Model checking plays a crucial role for our instantiation-based approach. Then, we present encodings for the computation of arguments. Finally, we provide ASP code for some types of attack relations. Note that an argumentation framework is obtained by two separate ASP program calls where the first one takes as input $\mathcal{K}$ and $\mathcal{C}$ and returns a separate answer-set for each resulting argument. The second program receives as input a "flattened" version of all arguments and computes the attacks between arguments based on different attack type encodings.

3.1 Model Checking

Propositional formulae provide the basis for the construction of arguments and their attack relations. In fact, we can express most of the defining properties of arguments (such as entailment of the support to the claim) and attacks by means of propositional formulae. In this section we provide an ASP encoding that allows us to check whether a formula α is true under a given interpretation I, i.e. I is a model for α. First, the formula is split into sub-formulae until we obtain the contained atoms or constants. Due to brevity, the following encodings only exemplify this for the connectives $\land$, $\neg$ and $\rightarrow$. Note that $\lor$, $\not\leftrightarrow$ and $\leftrightarrow$ are supported as well.

$$
\begin{aligned}
\pi_{\mathrm{subformula}} = \{\ &\mathrm{subformula}(F) \leftarrow \mathrm{subformula}(\mathrm{and}(F,\_)); &(1)\\
&\mathrm{subformula}(F) \leftarrow \mathrm{subformula}(\mathrm{and}(\_,F)); &(2)\\
&\mathrm{subformula}(F) \leftarrow \mathrm{subformula}(\mathrm{neg}(F)); &(3)\\
&\mathrm{subformula}(F) \leftarrow \mathrm{subformula}(\mathrm{imp}(F,\_)); &(4)\\
&\mathrm{subformula}(F) \leftarrow \mathrm{subformula}(\mathrm{imp}(\_,F)). \} &(5)
\end{aligned}
$$

The atoms and constants of α are then obtained via the encoding π_{atom}. Consider rule (1) which denotes that a formula is not an atom in case it is of the form $\mathrm{and}(\cdot,\cdot)$ [4].

$$
\begin{aligned}
\pi_{\mathrm{atom}} = \{\ &\mathrm{noatom}(F) \leftarrow \mathrm{subformula}(F;F1;F2), F := \mathrm{and}(F1,F2); &(1)\\
&\mathrm{noatom}(F) \leftarrow \mathrm{subformula}(F;F1), F := \mathrm{neg}(F1); &(2)\\
&\mathrm{noatom}(F) \leftarrow \mathrm{subformula}(F;F1;F2), F := \mathrm{imp}(F1,F2); &(3)\\
&\mathrm{atom}(X) \leftarrow \mathrm{subformula}(X), not\ \mathrm{noatom}(X). \} &(4)
\end{aligned}
$$

Now we compute whether the interpretation is a model by first evaluating the atoms and constants. In case an atom gets assigned true (false) we derive that the interpretation for this sub-formula is a model (not a model). Now, the connectives are evaluated bottom-up based on the model information of the sub-formulae. In particular, this allows to check whether I is a model for our original formula α, or not.

The encoding π_{ismodel} exemplifies this approach for some of the connectives. In the subsequent sections we have to apply model checking several times within a single ASP encoding. In order to avoid side effects of different checks, we introduce an additional

---

[4] Note that the syntax of our encodings is specific to the grounder gringo [15].

parameter, K, which serves as a key for identifying the origin of the interpretation that is currently checked. Suppose, for example, that we want to check satisfiability of two different formulae. As the formulae may evaluate to true under different interpretations we have to distinguish between the truth assignments.

$$\pi_{\text{ismodel}} = \{ \text{ismodel}(K, X) \leftarrow \text{atom}(X), \text{true}(K, X); \tag{1}$$

$$\text{ismodel}(K, F) \leftarrow \text{subformula}(F; F1), F := \text{neg}(F1), \\ \text{nomodel}(K, F1); \tag{2}$$

$$\text{ismodel}(K, F) \leftarrow \text{subformula}(F), F := \text{and}(F1, F2), \\ \text{ismodel}(K, F1; F2); \tag{3}$$

$$\text{ismodel}(K, F) \leftarrow \text{subformula}(F), F := \text{imp}(F1, F2), \\ \text{ismodel}(K, F1; F2). \} \tag{4}$$

Due to brevity we omit the encoding π_{nomodel} here. Analogous to π_{ismodel} it derives the predicate $\text{nomodel}(K, F)$ whenever an atom gets assigned false or a sub-formula is false under the current interpretation. The complete program for checking whether a formula evaluates to true under a given variable assignment consists of

$$\pi_{\text{modelcheck}} = \pi_{\text{subformula}} \cup \pi_{\text{atom}} \cup \pi_{\text{ismodel}} \cup \pi_{\text{nomodel}}$$

Example 6. Consider the formula $a \rightarrow b$ from $\mathcal{K}$ of Example 5, i.e. $\text{kb}(\text{imp}(a, b))$. In order to check whether there exists a model we can make use of $\pi_{\text{modelcheck}}$ in the following way: Initially, we have to define an additional rule $\text{subformula}(X) \leftarrow \text{kb}(X)$ as $\pi_{\text{subformula}}$ only considers formulae given by the predicate $\text{subformula}(\cdot)$. By adding the program $\pi_{\text{subformula}} \cup \pi_{\text{atom}}$ the following answer-set is returned:

$$\{\text{kb}(\text{imp}(a, b)). \text{subformula}(\text{imp}(a, b)). \text{subformula}(a). \text{subformula}(b). \\ \text{noatom}(\text{imp}(a, b)). \text{atom}(b). \text{atom}(a).\}$$

Each atom now gets assigned true or false, representing an interpretation. We encode this by the rule $\text{true}(k, X) \vee \text{false}(k, X) \leftarrow \text{atom}(X)$. Note that the specification of a key (in this case k) is mandatory although $\pi_{\text{modelcheck}}$ is not applied several times in this example. By adding and running $\pi_{\text{ismodel}} \cup \pi_{\text{nomodel}}$ four answer-sets are returned. Each contains the predicates from the previously given answer-set as well as the truth assignment for the atoms a and b and either $\text{ismodel}(k, \text{imp}(a, b))$ or $\text{nomodel}(k, \text{imp}(a, b))$. The answer-set obtained by $\text{false}(k, a)$ and $\text{true}(k, b)$ contains (amongst others)

$$\{\text{false}(k, a). \text{true}(k, b). \text{ismodel}(k, b). \text{nomodel}(k, a). \text{ismodel}(k, \text{imp}(a, b)).\}$$

denoting that $I(a) = \text{false}, I(b) = \text{true}$ is a model for $a \rightarrow b$.

3.2 Forming Arguments

We now derive the arguments from a knowledge base $\mathcal{K}$ and a set $\mathcal{C}$ of claims. According to [4], we have to check whether the support entails the claim and if the support is subset minimal as well as consistent. In order to obtain arguments we first guess exactly one claim and a subset of formulae from $\mathcal{K}$. This *guess* is encoded as follows:

$$\pi_{\text{arg}} = \{ \ 1\{ \text{sclaim}(X) : \text{cl}(X) \}1; \tag{1}$$

$$1\{ \text{fs}(X) : \text{kb}(X) \}. \} \tag{2}$$

The selected claim is denoted by $\text{sclaim}(\cdot)$. The predicate $\text{fs}(\cdot)$ is derived if the respective formula from $\mathcal{K}$ is contained in the support S of an argument $A = (S, C)$.

Entailment: In order to be a valid argument, the support must entail the claim, i.e. $S \models C$ must hold. As $S \models C$, $\models S \to C$ must hold as well. Hence, $\neg(S \to C) \equiv \neg(\neg S \vee C) \equiv S \wedge \neg C$ must be unsatisfiable. Unsatisfiability of the formula $S \wedge \neg C$ can be checked by making use of the saturation technique [12]: We first assign $\text{true}(entail, x)$ or $\text{false}(entail, x)$ to each atom x in the formula using a disjunctive rule. This allows *both* $\text{true}(entail, x)$ and $\text{false}(entail, x)$ to be contained in the resulting answer-set. Furthermore, all formulae in S and the negated claim C are conjunctively connected. Hence, in case any of those formulae evaluates to false under a variable assignment (i.e. $\text{nomodel}(entail, \cdot)$ is derived) we know that $\neg(S \to C)$ is not satisfied which implies that $S \models C$ evaluates to true under the given interpretation. In this case we *saturate*, i.e. we derive $\text{true}(entail, x)$ and $\text{false}(entail, x)$ for any atom x. On the other hand, if no formula in S and C derives entails_claim the constraint $\leftarrow not$ entails_claim removes the answer-set. If this is the case, due to the definition of stable model semantics in answer-set programming, no answer-set is returned. Only in case there exists no model for $\neg(S \to C)$ all guesses are saturated and we obtain a single answer-set representing a support S and claim C where $S \models C$ holds.

In the following the program $\pi_{\text{entailment}}$ is given. Note that *entail* is simply used as a key for identifying the variable assignment and model check.

$$
\begin{aligned}
\pi_{\text{entailment}} = \{\ &\text{true}(entail, X) \vee \text{false}(entail, X) \leftarrow \text{atom}(X); &&(1)\\
&\text{entails\_claim} \leftarrow \text{nomodel}(entail, neg(X)), \text{sclaim}(X); &&(2)\\
&\text{entails\_claim} \leftarrow \text{nomodel}(entail, X), \text{fs}(X); &&(3)\\
&\leftarrow not\ \text{entails\_claim}; &&(4)\\
&\text{true}(entail, X) \leftarrow \text{entails\_claim}, \text{atom}(X); &&(5)\\
&\text{false}(entail, X) \leftarrow \text{entails\_claim}, \text{atom}(X).\ \} &&(6)
\end{aligned}
$$

Subset minimality: The support S of an argument must be a subset minimal set of formulae, i.e. there must not exist an $S' \subset S$ s.t. $S' \models C$. Here, we apply the concept of a *loop* (see e.g. [13]). For a candidate support S we consider all $S' \subset S$ where there exists exactly one formula $\alpha \in S$ but $\alpha \notin S'$. In case any such S' exists where $S' \models C$ we know that S is not a support for C. Due to monotonicity of classical logic this is sufficient since if $S' \not\models C$ then also for all $S'' \subset S'$ it holds that $S'' \not\models C$. First, we define a total ordering over all formulae $\text{fs}(\cdot)$ in S:

$$
\begin{aligned}
\pi_{<} = \{\ &\text{lt}(X, Y) \leftarrow \text{fs}(X), \text{fs}(Y), X < Y; &&(1)\\
&\text{nsucc}(X, Z) \leftarrow \text{lt}(X, Y), \text{lt}(Y, Z); &&(2)\\
&\text{succ}(X, Y) \leftarrow \text{lt}(X, Y), not\ \text{nsucc}(X, Y); &&(3)\\
&\text{ninf}(Y) \leftarrow \text{lt}(X, Y); &&(4)\\
&\text{inf}(X) \leftarrow \text{fs}(X), not\ \text{ninf}(X); &&(5)\\
&\text{nsup}(X) \leftarrow \text{lt}(X, Y); &&(6)\\
&\text{sup}(X) \leftarrow \text{fs}(X), not\ \text{nsup}(X).\ \} &&(7)
\end{aligned}
$$

For any S' we now assign $\text{true}(m(K), x)$ or $\text{false}(m(K), x)$ to all atoms x. $m(K)$ is used as key for identifying the truth assignment. K is the formula α where $\alpha \notin S'$.

The idea is now to "iterate" over the the ordering, beginning at the infimum $\inf(\cdot)$. Based on the ordering, we now consider every formula from the support: In case the formula is satisfied or corresponds to the removed formula α (i.e. the key K) we derive model_upto$(m(\alpha), \cdot)$. If we can derive hasmodel$(m(\alpha))$ we know that the support $S' = S \setminus \alpha$ is satisfiable and can therefore not be a valid support for our claim. On the other hand, if any S' is a valid support we can not derive hasmodel$(m(\alpha))$ and the answer-set is removed by the constraint $\leftarrow not$ hasmodel$(m(\alpha))$, fs(α).

$$
\begin{aligned}
\pi_{\mathrm{minimize}} = \{\, &\mathrm{true}(m(K), X) \leftarrow not\, \mathrm{false}(m(K), X), \mathrm{atom}(X), \mathrm{fs}(K); & (1)\\
&\mathrm{false}(m(K), X) \leftarrow not\, \mathrm{true}(m(K), X), \mathrm{atom}(X), \mathrm{fs}(K); & (2)\\
&\mathrm{model\_upto}(m(K), X) \leftarrow \mathrm{inf}(X), \mathrm{ismodel}(m(K), X), \\
&\qquad\qquad \mathrm{fs}(X), X \neq K; & (3)\\
&\mathrm{model\_upto}(m(K), K) \leftarrow \mathrm{inf}(K), \mathrm{fs}(K); & (4)\\
&\mathrm{model\_upto}(m(K), X) \leftarrow \mathrm{succ}(Z, X), \mathrm{ismodel}(m(K), X), \\
&\qquad\qquad \mathrm{fs}(X), \mathrm{model\_upto}(m(K), Z), X \neq K; & (5)\\
&\mathrm{model\_upto}(m(K), K) \leftarrow \mathrm{succ}(Z, K), \mathrm{model\_upto}(m(K), Z), \\
&\qquad\qquad \mathrm{fs}(K); & (6)\\
&\mathrm{hasmodel}(m(K)) \leftarrow \mathrm{sup}(K), \mathrm{model\_upto}(m(K), X), \\
&\qquad\qquad \mathrm{ismodel}(m(K), \mathrm{neg}(Z)), \mathrm{sclaim}(Z); & (7)\\
&\leftarrow not\, \mathrm{hasmodel}(m(K)), \mathrm{fs}(K). \,\} & (8)
\end{aligned}
$$

Consistency: The support S must be a consistent set of formulae. In other words, there exists a model for the conjunction of all formulae in S. The program $\pi_{\mathrm{consistent}}$ simply consists of a guess which assigns truth values to all atoms and a constraint that removes any unsatisfiable support.

$$
\begin{aligned}
\pi_{\mathrm{consistent}} = \{\, &1\{\, \mathrm{true}(\mathrm{consistent}, X), \mathrm{false}(\mathrm{consistent}, X) \,\}1 \leftarrow \mathrm{atom}(X). & (1)\\
&\leftarrow \mathrm{nomodel}(\mathrm{consistent}, X), \mathrm{fs}(X). \,\}. \,\} & (2)
\end{aligned}
$$

The following program then gives all arguments that can be computed from a knowledge base $\mathcal{K}$ and a set of claims $\mathcal{C}$:

$$
\pi_{\mathrm{arguments}} = \pi_{\mathrm{modelcheck}} \cup \pi_{\mathrm{arg}} \cup \pi_{\mathrm{entailment}} \cup \pi_{<} \cup \pi_{\mathrm{minimize}} \cup \pi_{\mathrm{consistent}}
$$

Each answer-set obtained by $\pi_{\mathrm{arguments}}$ contains the predicate sclaim$(\cdot)$ and a set of predicates fs$(\cdot)$, representing claim and support.

Example 7. Consider the input as given in Example 5. The program $\pi_{\mathrm{arguments}}$ returns the following answer-sets (we restrict ourselves to the relevant predicates):

$$
\begin{aligned}
a_1 &: \{\mathrm{fs}(a).\ \mathrm{sclaim}(a).\} \\
a_2^* &: \{\mathrm{fs}(\mathrm{imp}(a,b)).\ \mathrm{sclaim}(\mathrm{imp}(a,b)).\} \\
a_3 &: \{\mathrm{fs}(a).\ \mathrm{fs}(\mathrm{imp}(a,b)).\ \mathrm{sclaim}(b).\} \\
a_4 &: \{\mathrm{fs}(\mathrm{neg}(b)).\ \mathrm{fs}(\mathrm{imp}(a,b)).\ \mathrm{sclaim}(\mathrm{neg}(a)).\} \\
a_5 &: \{\mathrm{fs}(\mathrm{neg}(b)).\ \mathrm{sclaim}(\mathrm{neg}(b)).\} \\
a_6 &: \{\mathrm{fs}(a).\ \mathrm{fs}(\mathrm{neg}(b)).\ \mathrm{sclaim}(and(a, \mathrm{neg}(b))).\}
\end{aligned}
$$

Note that due to the definition of program π_{minimize} and $\pi_{\mathrm{consistent}}$ several resulting answer-sets may represent the same derived argument: This is the case for a_2^* where actually three models are derived by the program $\pi_{\mathrm{consistent}}$. They only differ in the respective truth assignments $\mathrm{true}(\mathrm{consistent}, \cdot)$ and $\mathrm{false}(\mathrm{consistent}, \cdot)$. We eliminate duplicates in an additional post-processing step in order to remove redundant information.

3.3 Identifying Conflicts between Arguments

We now want to compute attacks between arguments. Therefore we first specify encodings that are used by every attack type (such as defeat and direct defeat). We then present encodings for the computation of these attack types.

In order to reason over all arguments we first have to "flatten" the answer-sets obtained by $\pi_{\mathrm{arguments}}$. We specify this by the predicates $\mathrm{as}(A, fs, \cdot)$ and $\mathrm{as}(A, claim, \cdot)$. A is a numeric key identifying the argument.

Example 8. We illustrate this by the answer-sets a_1, a_2 and a_3 from Example 7. This input is given by the following facts:

$$\{\mathrm{as}(1, \mathrm{fs}, a).\ \mathrm{as}(1, \mathrm{sclaim}, a).\ \mathrm{as}(2, \mathrm{fs}, \mathrm{imp}(a, b)).\ \mathrm{as}(2, \mathrm{sclaim}, \mathrm{imp}(a, b)).\ \mathrm{as}(3, \mathrm{fs}, a).$$
$$\mathrm{as}(3, \mathrm{fs}, \mathrm{imp}(a, b)).\ \mathrm{as}(3, \mathrm{sclaim}, b).\}$$

In order to identify conflicts between arguments we first guess two arguments. $\mathrm{selected1}(\cdot)$ and $\mathrm{selected2}(\cdot)$ contain the keys of the selected arguments.

$$\pi_{\mathrm{att}} = \{\ 1\{\mathrm{selected1}(A) : \mathrm{as}(A, \_, \_)\}1. \tag{1}$$
$$1\{\mathrm{selected2}(A) : \mathrm{as}(A, \_, \_)\}1.\ \} \tag{2}$$

Furthermore, we construct one single support formula for each argument A by conjunction of all formulae in $\mathrm{as}(A, fs, \cdot)$. As in the previous section we first define an ordering over all formulae that are contained in the support. The only difference is that we add the argument's key A to the predicates $\inf(A, \cdot)$, $\sup(A, \cdot)$ and $\mathrm{succ}(A, \cdot, \cdot)$. Due to brevity, the corresponding program $\pi_{<_{\mathrm{key}}}$ is omitted. We can then construct the support formula by iterating over the ordering and connecting the formulae by conjunction. Note that the last parameter of $\mathrm{fs\_conj}(A, \cdot, \cdot)$ is simply used as an identifier for the current position in the iteration. When the supremum is reached we derive $\mathrm{support}(A, \cdot)$ for A containing the support formula.

$$\pi_{\mathrm{support}} = \{\ \mathrm{fs\_conj}(A, X, X) \leftarrow \inf(A, X), \sup(A, X); \tag{1}$$
$$\mathrm{fs\_conj}(A, \mathrm{and}(X, Y), Y) \leftarrow \inf(A, X), \mathrm{succ}(A, X, Y); \tag{2}$$
$$\mathrm{fs\_conj}(A, \mathrm{and}(O, N), N) \leftarrow \mathrm{succ}(A, C, N), \mathrm{fs\_conj}(A, O, C); \tag{3}$$
$$\mathrm{support}(A, X) \leftarrow \mathrm{fs\_conj}(A, X, C), \sup(A, C).\ \} \tag{4}$$

For the computation of attacks we again apply the saturation technique. The program $\pi_{\mathrm{att\_sat}}$ is used to saturate all attack computations. First, we derive all attack type keys t from the truth assignments $\mathrm{true}(t, \cdot)$ and $\mathrm{false}(t, \cdot)$ of the applied attack type programs. Note that the corresponding assignments are defined separately in each attack program. In case attack is derived for all truth assignments in some attack program

we saturate. Finally, the binary predicate $\text{attack}(\cdot, \cdot)$ is generated. It is used for the representation of the attack relation between the two arguments.

$$
\begin{aligned}
\pi_{\text{att\_sat}} = \{\ & \text{attacktype}(T) \leftarrow \text{true}(T, \_); & (1)\\
& \text{attacktype}(T) \leftarrow \text{false}(T, \_); & (2)\\
& \text{true}(T, X) \leftarrow \text{attack}, \text{atom}(X), \text{attacktype}(T); & (3)\\
& \text{false}(T, X) \leftarrow \text{attack}, \text{atom}(X), \text{attacktype}(T); & (4)\\
& \leftarrow not\ \text{attack}; & (5)\\
& \text{attack}(X, Y) \leftarrow \text{selected1}(X), \text{selected2}(Y).\ \} & (6)
\end{aligned}
$$

We now consider the attack types *defeat* and *direct defeat*. The basic idea is to define a propositional formula that represents the attack condition. We then assign $\text{true}(t, x)$ or $\text{false}(t, x)$ to any atom x using a disjunctive rule. In case every such interpretation is a model for our attack formula we know that the formula is valid. Otherwise, if any interpretation is not a model (i.e. attack is not derived) the resulting answer-set is strictly smaller than those where the interpretation is a model. Such answer-sets are removed by the constraint $\leftarrow not\ \text{attack}$. Note that this also works in case we consider several attack types: If we derive attack for all interpretations of a single attack type we saturate all interpretations of *all* attack types. As the predicate attack as well as all assignments $\text{true}(t, x)$ and $\text{false}(t, x)$ are then contained in all answer-sets for the two selected arguments we derive every attack relation.

The following program π_{defeat} exemplifies the above described approach. For two arguments $A = (S, C)$ and $A' = (S' = \{\phi'_1, ..., \phi'_m\}, C')$ we have that A defeats A' if $C \models \neg(\phi'_1 \wedge ... \wedge \phi'_m)$. Here we make use of the support derived by $\pi_{<_{\text{key}}} \cup \pi_{\text{support}}$. As the support is defined as a conjunction of formulae we can directly make use of claim C and the negated support $\neg S'$.

$$
\begin{aligned}
\pi_{\text{defeat}} = \{ & \text{checkDefeat}(\text{imp}(C, \text{neg}(S'))) \leftarrow \text{selected1}(X), \text{selected2}(Y), \\
& \qquad\qquad\qquad\qquad\qquad \text{as}(X, \text{sclaim}, C), \text{support}(Y, S'); & (1)\\
& \text{subformula}(X) \leftarrow \text{checkDefeat}(X); & (2)\\
& \text{true}(\text{defeat}, X) \vee \text{false}(\text{defeat}, X) \leftarrow \text{atom}(X); & (3)\\
& \text{attack} \leftarrow \text{ismodel}(\text{defeat}, X), \text{checkDefeat}(X).\ \} & (4)
\end{aligned}
$$

The second program we consider here is π_{ddefeat}. A directly defeats A' if $C \models \neg\phi'_i$ for a $\phi'_i \in S'$. Hence, we have to consider each formula in S' separately. Therefore we use a combination of attack type and ϕ'_i to identify the truth assignment.

$$
\begin{aligned}
\pi_{\text{ddefeat}} = \{ & \text{checkDirectdefeat}(\Phi, \text{imp}(C, \text{neg}(\Phi))) \leftarrow \text{selected1}(X), \text{selected2}(Y), \\
& \qquad\qquad\qquad\qquad\qquad \text{as}(X, \text{sclaim}, C), \text{as}(Y, \text{fs}, \Phi); & (1)\\
& \text{subformula}(X) \leftarrow \text{checkDirectdefeat}(\_, X); & (2)\\
& \text{true}(\text{ddefeat}(T), X) \vee \text{false}(\text{ddefeat}(T), X) \leftarrow \text{atom}(X), \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{checkDirectdefeat}(T, \_); & (3)\\
& \text{attack} \leftarrow \text{ismodel}(\text{ddefeat}(T), X), \text{checkDirectdefeat}(T, X).\ \} & (4)
\end{aligned}
$$

The program π_{attacks} in combination with any attack type programs (such as π_{defeat}, π_{ddefeat}, ...) computes the respective attack relations.

$$
\pi_{\text{attacks}} = \pi_{\text{modelcheck}} \cup \pi_{\text{att}} \cup \pi_{\text{support}} \cup \pi_{<_{\text{key}}} \cup \pi_{\text{att\_sat}}
$$

Example 9. Continuing our running example, we now consider the flattened input as exemplified in Example 8 and the program $\pi_{\text{attacks}} \cup \pi_{\text{ddefeat}}$. We obtain 9 answer-sets that contain the attack information between arguments:

$$\{\text{attack}(4,1).\}, \{\text{attack}(6,2).\}, \{\text{attack}(4,3).\}, \{\text{attack}(6,4).\}, \{\text{attack}(3,5).\},$$
$$\{\text{attack}(3,4).\}, \{\text{attack}(6,3).\}, \{\text{attack}(4,6).\}, \{\text{attack}(3,6).\}$$

The first answer-set, for example, represents that argument $a_4 = (\{\neg b, a \rightarrow b\}, \neg a)$ attacks (directly defeats) $a_1 = (\{a\}, a)$.

3.4   Overall Approach at a Glance

To sum it up, the overall process of our instantiation-based approach for generating argumentation frameworks consists of the following steps:

1. A knowledge-base $\mathcal{K}$ and a set $\mathcal{C}$ of claims are used as input.
2. The encoding $\pi_{\text{arguments}} = \pi_{\text{modelcheck}} \cup \pi_{\text{arg}} \cup \pi_{\text{entailment}} \cup \pi_{<} \cup \pi_{\text{minimize}} \cup \pi_{\text{consistent}}$ defines how arguments are derived from $\mathcal{K}$ and $\mathcal{C}$: $\pi_{\text{modelcheck}}$ is generally used for evaluating formulae under truth assignments. Within π_{arg}, for each argument $A = (S, C)$ a claim $C \in \mathcal{C}$ and a support $S \subseteq \mathcal{K}$ is guessed. The encodings $\pi_{\text{entailment}}$, $\pi_{<} \cup \pi_{\text{minimize}}$ and $\pi_{\text{consistent}}$ guarantee that the support entails the claim, the support is subset minimal and that S is a consistent set of formulae.
3. The resulting arguments are "flattened" and used as input for π_{attacks}.
4. The encoding $\pi_{\text{attacks}} = \pi_{\text{modelcheck}} \cup \pi_{\text{att}} \cup \pi_{\text{support}} \cup \pi_{<_{\text{key}}} \cup \pi_{\text{att\_sat}}$ is shared by all attack types. $\pi_{\text{modelcheck}}$ is again needed for model checking. π_{att} guesses two arguments at once. $\pi_{\text{support}} \cup \pi_{<_{\text{key}}}$ constructs a single support formula for those arguments by connecting the contained formulae by conjunction. $\pi_{\text{att\_sat}}$ saturates all attack computations.
5. Any attack encodings, such as π_{defeat} and π_{ddefeat} can be used in combination with π_{attacks} in order to compute the respective attack relations.

4   Visualization of Argumentation Frameworks

In order to visualize argumentation frameworks we make use of the purpose-built tool ARVis[5]. ARVis is intended for the visualization of answer-sets and their relations by means of a directed graph. Each node in the graph represents an answer-set and a directed edge between two arguments represents a relation.

We now describe the process of generating and visualizing argumentation frameworks by using the encodings $\pi_{\text{arguments}}$ and π_{attacks}. ARVis provides a wizard that handles the respective steps:

1. Obtain arguments: The program $\pi_{\text{arguments}}$ and a problem instance must be specified within ARVis. ARVis computes the arguments by invoking an ASP solver.
2. Flatten arguments: The arguments obtained in the previous step are "flattened", i.e. a single set of facts is generated in order to be able to reason over all obtained arguments when computing the attacks.

---

[5] `http://dbai.tuwien.ac.at/proj/argumentation/vispartix/#ARVis`

3. Obtain attacks: The program π_{attacks} and the attack type programs are now specified. The relations between the arguments are computed.
4. Select attack predicate: In general, ARVis accepts any binary predicate that represents answer-set relations. We define $\mathrm{attack}/2$ here.
5. Argumentation framework: We obtain a graph visualization consisting of arguments (vertices) and attacks (edges).
6. Export: The obtained argumentation framework can be exported for further processing.

Fig. 2. ARVis: Resulting argumentation framework
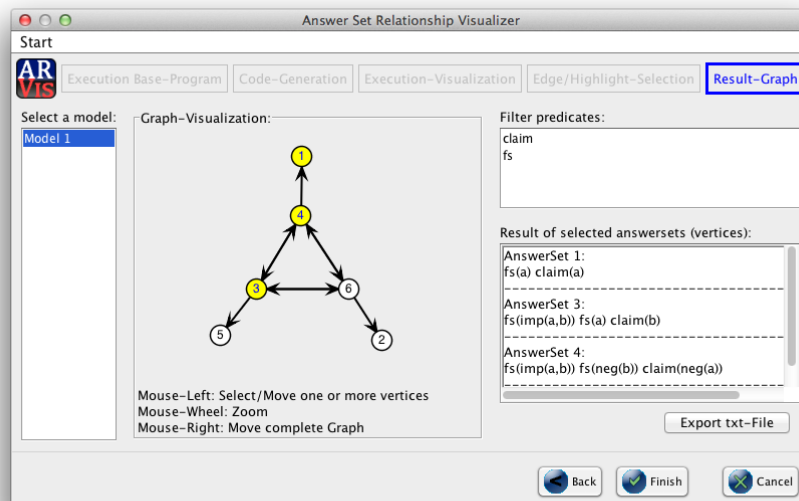
The argumentation framework resulting from our running example, as represented by ARVis, is given in Figure 2. The attacks correspond to direct defeats between arguments. Each argument in the graph is represented by its id. By selecting an argument its claim and support are shown in the text field on the right side. All encodings, ARVis and detailed configuration information is available at

```
http://dbai.tuwien.ac.at/proj/argumentation/vispartix
```

ARVis is a general-purpose tool that may also be used in many other areas of research: Consider, for example, the Traveling Salesperson Problem (TSP) where the first program computes cities and the second program outputs routes where every city is visited exactly once. In fact, ARVis can be used for any problem where one is interested in the relation between answer-sets. It is only necessary to specify two answer-set programs: One for generating answer-sets and a second one for computing relations between those answer-sets.

Our approach is different from other available ASP visualization tools: ASPViz [8] takes two answer-set programs as input, one for the problem encoding and one for the visualization. The latter is used for visualization for each answer-set of the former

separately. It is realized in Java and works with pre-defined predicates to extract the visualization of answer-sets. IDPDraw [29] works in a similar fashion, which augments the presentation by providing also time points to show the result in different evolutionary states. Kara [19] from the SeaLion development environment for ASP also provides visualization of answer-sets using special predicates. ASPIDE [14] gives the user the opportunity to visualize the dependency graph of the input program and thus allows for another type of representation.

5   Conclusion

In this paper, we have presented a novel ASP-based tool for constructing argumentation frameworks from a given knowledge base. We have provided here the concrete ASP encodings used to obtain such frameworks when logic-based arguments cf. [4] are employed. However, similar encodings for further approaches of argumentation are possible and subject of future work, as well as a performance evaluation of the presented approach to check its scalability with large knowledge bases.

When designing our tool, ARVis, we tried to keep it as flexible as possible such that the concrete construction of the framework can be specified in the logic programs. As it has turned out, ARVis is thus not only a tool for generating and visualizing argumentation frameworks but also for graphically representing relations between answer-sets in a user-specified manner. Ongoing work thus focuses on application areas where it is the relation between the answer-sets (rather than the single answer-sets) that can support the designer of logic programs or where this relation is the relevant output of an ASP encoding.

References

1. L. Amgoud, P. Besnard, and S. Vesic. Identifying the Core of Logic-Based Argumentation Systems. In *ICTAI'11*, pages 633–636, 2011.
2. K. Ashley, N. Pinkwart, C. Lynch, and V. Aleven. Learning by diagramming Supreme Court oral arguments. In *ICAIL'07*, pages 271–275, 2007.
3. P. Baroni and M. Giacomin. Semantics of Abstract Argument Systems. In G. Simari and I. Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 25–44. 2009.
4. P. Besnard and A. Hunter. A logic-based theory of deductive arguments. *Artif. Intell.*, 128(1-2):203–235, 2001.
5. J. Börding, A. Voss, J. Walther, V. Wolff, A. Ocakli, R. Groot, and B. Baurens. DUNES-Dialogic and Argumentative Negotiation Educational Software-Technical Realization. In *ICTE'02*, 2002.

6. G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
7. M. Caminada and L. Amgoud. On the evaluation of argumentation formalisms. *Artif. Intell.*, 171(5-6):286–310, 2007.
8. O. Cliffe, M. D. Vos, M. Brain, and J. A. Padget. ASPVIZ: Declarative Visualisation and Animation Using Answer Set Programming. In *ICLP'08*, volume 5366 of *LNCS*, pages 724–728, 2008.
9. P. M. 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.
10. W. Dvořák, M. Järvisalo, J. P. Wallner, and S. Woltran. Complexity-Sensitive Decision Procedures for Abstract Argumentation. In *KR'12*, pages 54–64, 2012.
11. U. Egly, S. A. Gaggl, and S. Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.
12. T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995.
13. T. Eiter, G. Ianni, and T. Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web*, volume 5689 of *LNCS*, pages 40–110, 2009.
14. O. Febbraro, K. Reale, and F. Ricca. ASPIDE: Integrated Development Environment for Answer Set Programming. In *LPNMR*, volume 6645 of *LNCS*, pages 317–330, 2011.
15. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):105–124, 2011.
16. T. Gordon. An Overview of the Carneades Argumentation Support System. In Tindale, C.W. and Reed, C. *Dialectics, Dialogue and Argumentation. An Examination of Douglas Walton's Theories of Reasoning*, pages 145–156. 2010.
17. N. Gorogiannis and A. Hunter. Instantiating abstract argumentation with classical logic arguments: Postulates and properties. *Artif. Intell.*, 175(9-10):1479–1497, 2011.
18. N. Karacapilidis and D. Papadias. Computer supported argumentation and collaborative decision making: the HERMES system. *Inf. Syst.*, 26(4):259–277, 2001.
19. C. Kloimüllner, J. Oetsch, J. Pührer, and H. Tompits. Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs. In *WLP'11*, 2011.
20. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
21. V. W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. 1999.
22. I. Niemelä. Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. *Ann. Math. Artif. Intell.*, 25(3–4):241–273, 1999.
23. S. Parsons, M. Wooldridge, and L. Amgoud. Properties and Complexity of Some Formal Inter-agent Dialogues. *J. Log. Comput.*, 13(3):347–376, 2003.
24. C. Reed and G. Rowe. Araucaria: Software for Argument Analysis, Diagramming and Representation. *Int. Journal on Artif. Intel. Tools*, 13(4):983–, 2004.
25. D. Schneider, C. Voigt, and G. Betz. ArguNet a software tool for collaborative argumentation analysis and research. In *CMNA VII*, 2007.
26. S. B. Shum. Cohere: Towards Web 2.0 Argumentation. In *COMMA'08*, pages 97–108, 2008.
27. M. Snaith and C. Reed. TOAST: online ASPIC+ implementation. In *COMMA'12 (to appear)*.
28. B. Verheij. ArguMed - A Template-Based Argument Mediation System for Lawyers. In *JURIX'98*, pages 113–130, 1998.
29. J. Wittocx. IDPDraw, a tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation`, 2009.