The DIAMOND System for Argumentation: Preliminary Report*

Stefan Ellmauthaler and Hannes Strass

Computer Science Institute, Leipzig University

Abstract Abstract dialectical frameworks (ADFs) are a powerful generalisation of Dung's abstract argumentation frameworks. In this paper we present an answer set programming based software system, called DIAMOND (DIAlectical MOdels eNcoDing). It translates ADFs into answer set programs whose stable models correspond to models of the ADF with respect to several semantics (i.e. admissible, complete, stable, grounded).

1   Introduction

Formal argumentation has established itself as a vibrant subfield of artificial intelligence, contributing to such diverse topics as legal decision making and multi-agent interactions. A particular, well-known formalism to model argumentation scenarios are Dung's abstract argumentation frameworks [1]. In that model, arguments are treated as abstract atomic entities. The only information given about them is a binary relation expressing that one argument attacks another.

A criticism often advanced against Dung frameworks is their restricted expressive capability of allowing only attacks between arguments. This leads to quite a number of extensions of Dung AFs, for example with attacks from sets of arguments [2], attacks on attacks [3] and meta-argumentation [4]. Unifying these and other extensions to AFs, Brewka and Woltran [5] proposed a general model, abstract dialectical frameworks (ADFs). In ADFs, attack, support, joint support, combined attacks and many more relations between arguments (called statements there) can be expressed, while the whole formalism stays on the same abstraction level as Dung's.

In this paper we present the DIAMOND software system that computes models of ADFs with respect to several different semantics. The name DIAMOND abbreviates "DIAlectical MOdels eNcoDing" and hints at the fact that DIAMOND is built on top of the state of the art in answer set programming: abstract dialectical frameworks are encoded into logic programs, and an answer set solver is used to compute the models of the ADF. By providing an expressive argumentation formalism with an implementation, we pave the way for practical applications of ADFs in scenarios where dialectical aspects are of interest, for example in group decision making.

The paper proceeds as follows. We first introduce the necessary background in abstract dialectical frameworks and answer set programming. We then present

the DIAMOND system – how ADFs are represented there, and how the ADF semantics are encoded into answer set programs. We conclude with a contrasting discussion of the most significant related work.

2 Background

An abstract dialectical framework (ADF) [5] is a directed graph whose nodes represent statements or positions that can be accepted or not. The links represent dependencies: the status of a node s only depends on the status of its parents (denoted $par(s)$), that is, the nodes with a direct link to s. In addition, each node s has an associated acceptance condition C_s specifying the exact conditions under which s is accepted. C_s is a function assigning to each subset of $par(s)$ one of the truth values $\mathbf{t}$, $\mathbf{f}$. Intuitively, if for some $R \subseteq par(s)$ we have $C_s(R) = \mathbf{t}$, then s will be accepted provided the nodes in R are accepted and those in $par(s) \setminus R$ are not accepted.

Definition 1. *An* abstract dialectical framework *is a tuple* $D = (S, L, C)$ *where*

 – *S is a set of statements (positions, nodes),*
 – *$L \subseteq S \times S$ is a set of links,*
 – *$C = \{C_s\}_{s \in S}$ is a set of total functions $C_s : 2^{par(s)} \to \{\mathbf{t}, \mathbf{f}\}$.*

In many cases it is convenient to represent the acceptance condition of a statement s by a propositional formula φ_s, as is done in our running example.

Example 1. Consider the ADF $D = (S, L, C)$ with a support cycle and one attack relation: $S = \{a, b, c\}$, $L = \{(a, b), (b, a), (b, c)\}$, $\varphi_a = b, \varphi_b = a, \varphi_c = \neg b$. This ADF can also be represented as a graph, where the nodes are statements and the relations between them are directed edges. The boxes below each node are the acceptance conditions for the particular statement.
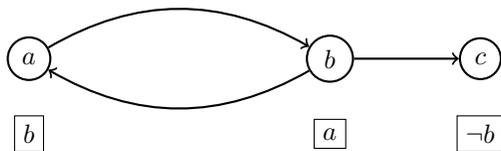
In recent work [6], we redefined several standard ADF semantics and defined additional ones. In this paper, we use these revised definitions, which are based on three-valued logic.[1] The three truth values true ($\mathbf{t}$), false ($\mathbf{f}$) and unknown ($\mathbf{u}$) are partially ordered by $\leq_i$ according to their information content: we have $\mathbf{u} <_i \mathbf{t}$ and $\mathbf{u} <_i \mathbf{f}$ and no other pair in $<_i$, which intuitively means that the classical truth values contain more information than the truth value unknown. On the set of truth values, we define a meet operation, *consensus*, which assigns

---

[1] For further details on those newly introduced semantics we refer the interested reader to Brewka et al. [6].

$\mathbf{t} \sqcap \mathbf{t} = \mathbf{t}$, $\mathbf{f} \sqcap \mathbf{f} = \mathbf{f}$, and returns $\mathbf{u}$ otherwise. The information ordering $\leq_i$ extends in a straightforward way to valuations v_1, v_2 over S in that $v_1 \leq_i v_2$ iff $v_1(s) \leq_i v_2(s)$ for all $s \in S$. Obviously, a three-valued interpretation v is two-valued if all statements are mapped to either true or false. For a three-valued interpretation v, we say that a two-valued interpretation w *extends* v iff $v \leq_i w$. We denote by $[v]_2$ the set of all two-valued interpretations that extend v. A three-valued interpretation E_v has an *associated extension* $E_v = \{s \in S \mid v(s) = \mathbf{t}\}$.

Brewka and Woltran [5] defined an operator Γ_D over three-valued interpretations. For each statement s, the operator returns the consensus truth value for its acceptance formula φ_s, where the consensus takes into account all possible two-valued interpretations w that extend the input valuation v.

Definition 2. *Let D be an ADF and v be a three-valued interpretation. Then the interpretation $\Gamma_D(v)$ is given by $s \mapsto \prod\{w(\varphi_s) \mid w \in [v]_2\}$. Furthermore v is admissible iff $v \leq_i \Gamma_D(v)$; complete iff $\Gamma_D(v) = v$, that is, v is a fixpoint of Γ_D; grounded iff v is the $\leq_i$-least fixpoint of Γ_D.*

A two-valued interpretation v is a model *of D iff $\Gamma_D(v) = v$; it is a* stable model *of $D = (S, L, C)$ iff v is a model of D and E_v equals the grounded extension of the reduced ADF $D^v = (E_v, L^v, C^v)$, where $L^v = L \cap (E_v \times E_v)$ and for $s \in E_v$ we set $\varphi_s^v = \varphi_s[r/\mathbf{f} : v(r) = \mathbf{f}]$.*

Example 2. We will now show the models with respect to the different semantics for the ADF introduced in Example 1. For readability, we write interpretations v as sets of literals $L_v = \{s \in S \mid v(s) = \mathbf{t}\} \cup \{\neg s \mid s \in S, v(s) = \mathbf{f}\}$. There are

- five admissible interpretations: $\emptyset$, $\{a, b\}$, $\{a, b, \neg c\}$ $\{\neg a, \neg b, c\}$, $\{\neg a, \neg b\}$,
- three complete models: $\{\neg a, \neg b, c\}$, $\emptyset$, $\{a, b, \neg c\}$; of which $\emptyset$ is grounded;
- two models: $\{a, b, \neg c\}$, $\{\neg a, \neg b, c\}$, of which one is stable: $\{\neg a, \neg b, c\}$

Brewka et al. [6] also defined an approach to handle preferences in ADFs. The approach generalises the one for AFs from Amgoud and Cayrol [7]. Since DIAMOND also implements this treatment of preferences, we recall it here. For this approach, the links are restricted to links that are attacking or supporting.

Definition 3. *A prioritised ADF (PADF) is a tuple $P = (S, L^+, L^-, >)$ where S is the set of nodes, L^+ and L^- are subsets of $S \times S$, the supporting and attacking links, and $>$ is a strict partial order (irreflexive, transitive, antisymmetric) on S representing preferences among the nodes.*

Here $(a, b) \in >$ (alternatively: $a > b$) expresses that a is preferred to b. The semantics of prioritised ADFs is given by a translation to standard ADFs: P translates to $(S, L^+ \cup L^-, C)$, where for each statement $s \in S$ the acceptance condition C_s is defined as: $C_s(M) = \mathbf{t}$ iff for each $a \in M$ such that $(a, s) \in L^-$ and not $s > a$ we have: for some $b \in M$, $(b, s) \in L^+$ and $b > a$. Intuitively, an attacker does not succeed if the attacked node is more preferred or if there is a more preferred supporting node.

2.1 Answer Set Programming

A *propositional normal logic program* Π is a set of finite rules r over a set of ground atoms $\mathcal{A}$. A rule r is of the form $\alpha \leftarrow \beta_1, \ldots, \beta_m, \text{not } \beta_{m+1} \ldots, \text{not } \beta_n$, where $\alpha \in \mathcal{A}$, $\beta_i \in \mathcal{A}$ are ground atoms and $m \leq n \leq 0$. Each rule consists of a *body* $B(r) = \{\beta_1, \ldots, \beta_m, \text{not } \beta_{m+1} \ldots, \text{not } \beta_n\}$ and a *head* $H(r) = \{\alpha\}$, divided by the $\leftarrow$-symbol. We will split up the body into two parts, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_m\}$ and $B^-(r) = \{\text{not } \beta_{m+1} \ldots, \text{not } \beta_n\}$. A rule r is said to be positive if $B^-(r) = \emptyset$ and a program Π is positive if every rule $r \in \Pi$ is positive. For a positive program Π, its immediate consequence operator T_Π is defined for $S \subseteq \mathcal{A}$ by $T_\Pi(S) = \{H(r) \in \mathcal{A} \mid r \in \Pi, B^+(r) \subseteq S\}$. A set $A \subseteq \mathcal{A}$ of ground atoms is a *minimal model* of a positive propositional logic program Π iff A is the least fixpoint of T_Π. To allow rules with negative body atoms, Gelfond and Lifschitz [8] proposed the stable model semantics (also called answer set semantics).

Definition 4. *Let $A \subseteq \mathcal{A}$ be a set of ground atoms. A is a* stable model *for the propositional normal logic program Π iff A is the minimal model of the reduced program Π^A, where $\Pi^A = \{H(r) \leftarrow B^+(r) \mid r \in \Pi, B^-(r) \cap A = \emptyset\}$.*

We use *clasp* from the Potsdam Answer Set Solving Collection *Potassco*[2] [9] as the back-end answer set solver for our software system. Potassco allows us to use an enriched input language where in addition to the above pictured propositional logic programs we can use first-order variables and predicates. Ground atoms are generally written in lower case while variables are represented with upper case characters. Additionally *Potassco* offers features like aggregates, cardinality constraints, choice rules and conditional literals. For further details we refer to the recent book by Gebser et al. [10].

3 DIAMOND

Our software system DIAMOND is a collection of answer set programming encodings and tools to compute the various models with respect to the semantics for a given ADF. The different encodings are designed around the *Potsdam Answer Set Solving Collection (Potassco)* [9] and the additional provided tools utilise clasp as solver, too. Note that the encodings for DIAMOND are built in a modular way. To compute the models of an ADF with respect to a semantics, different modules need to be grounded together to get the desired behaviour.

DIAMOND is available for download and experimentation at the web page `http://www.informatik.uni-leipzig.de/~ellmau/diamond`. There we also provide further documentation on its usage. In short, DIAMOND is a Python-script,[3] which can be invoked via the command line. Different switches are used to designate the desired semantics, and the input file is given as a file name or via the standard input. The options for the command line are as follows:

---

[2] Available at `http://potassco.sourceforge.net`
[3] Python is available at `http://www.python.org`.

```
usage: diamond.py [-h] [-cf] [-m] [-sm] [-g] [-c] [-a]
       [--transform_pform | --transform_prio] [-all] [--version] instance
positional arguments:
   instance             File name of the ADF instance
optional arguments:
  -h, --help            show this help message and exit
  -cf, --conflict-free  compute the conflict free sets
  -m, --model           compute the two-valued models
  -sm, --stablemodel    compute the stable models
  -g, --grounded        compute the grounded model
  -c, --complete        compute the complete models
  -a, --admissible      compute the admissible models
  --transform_pform     transform a propositional formula ADF before the computation
  --transform_prio      transform a prioritized ADF before the computation
  -all, --all           compute all sets and models
  --version             prints the current version
```

We next describe how specific ADF instances are represented in DIAMOND.

3.1   Instance Representation

In order to represent an ADF for DIAMOND its acceptance conditions need to
be in the functional representation as given in Definition 1. The statements of
an ADF are declared by the predicate s, and the links are represented by the
binary predicate l, such that l(b,a) reflects that there is a link from b to a. The
acceptance condition is modelled via the unary and tertiary predicates ci and
co. Intuitively ci (resp. co) identifies the parents which need to be accepted,
such that the acceptance condition maps to true (i.e. *in*) (resp. false (i.e. *out*)).
To achieve a flat representation of each set of parent statements, we use an
arbitrary third term in the predicate to identify them. To express what happens
to a statement when none of the parents is accepted we use the unary versions
of ci and co. Here is the DIAMOND representation of Example 1:

```
        s(a). s(b). s(c). l(b,a). l(a,b). l(b,c).
        co(a). ci(a,1,b). co(b). ci(b,1,a). ci(c). co(c,1,b).
```

The first line declares the statements and links. The second line expresses the
acceptance conditions: statement a is *out* if b is *out* and *in* if b is; likewise b gets
the same status as a; statement c is *in* if b is *out*, and c is *out* if b is *in*.

As a part of the DIAMOND software bundle, we also provide an ECL^iPS^e
Prolog[4] [11] program that transforms acceptance functions given as formulas
into the functional representation used by DIAMOND.

We have chosen this functional representation of acceptance conditions for
pragmatic reasons. An alternative would have been to represent acceptance con-
ditions by propositional formulas. In this case, computing a single step of the
operator would entail solving several **NP**-hard problems. The standard way to

---

[4] ECL^iPS^e is available at `http://eclipseclp.org/`.

solve these is the use of saturation [12], which however causes undesired side-effects when employed together with meta-ASP [13]. Furthermore, other ADF semantics (e.g. preferred) utilise concepts like $\subseteq$-minimality, which also require the use of meta-argumentation. We plan to extend DIAMOND to further semantics and therefore chose the functional representation of acceptance conditions to forestall potential implementation issues.

Due to compatibility considerations, it is possible for DIAMOND to understand the propositional formula representation as well as a PADF. The propositional formula representation uses the unary predicate `statement` to identify statements. The binary predicate `ac(s,`ϕ`)` associates to each statement `s` one formula ϕ. Each formula ϕ is constructed in the usual inductive way, where atomic formulae are other statements and the truth constants (i.e. `c(v)` and `c(f)`) and the operators are written as functions. The allowed operators are `neg, and, or, imp,` and `iff` for their respective logical operators. To describe a PADF, we use the unary predicate `s` to describe the set of statements. In addition the support (i.e. L^+) and attack (i.e. L^-) links are represented by the binary predicates `lp` and `lm` (i.e. positive resp. negative links). To express a preference, such as $a > b$, we use the predicate `pref(a,b)`. Note that DIAMOND provides a method to translate propositional formula ADFs and PADFs into ADFs with total functions and only computes the models using the functional representation.

For illustration, let us look at another, slightly more complicated example.

Example 3. Consider the ADF $D_2 = (S_2, L_2, C_2)$ with $S_2 = \{a, b, c, d\}$, $L_2 = \{(a, c), (b, b), (b, c), (b, d)\}$, and $C_2 = \{\varphi_a = \mathbf{t}, \varphi_b = b, \varphi_c = a \wedge b, \varphi_d = \neg b\}$.
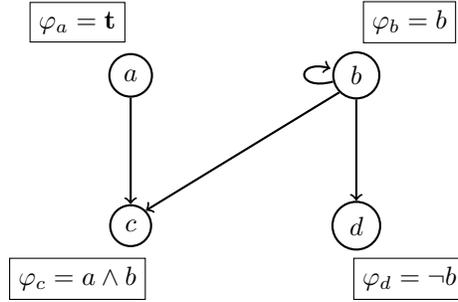
For this ADF there are

- 16 admissible interpretations: $\emptyset$, $\{a\}$, $\{b\}$, $\{\neg b\}$, $\{b, \neg d\}$, $\{a, b\}$, $\{a, \neg b\}$, $\{\neg b, d\}$, $\{\neg b, \neg d\}$, $\{a, b, c\}$, $\{a, b, \neg d\}$, $\{a, \neg b, d\}$, $\{a, \neg b, \neg c\}$, $\{\neg b, \neg c, d\}$, $\{a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$
- three complete models: $\{a\}$, $\{a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$; of these, $\{a\}$ is the grounded model;
- two models: $\{a, b, c, \neg d\}$, $\{a, \neg b, \neg c, d\}$, of which one is stable: $\{a, \neg b, \neg c, d\}$. Its propositional formula representation for DIAMOND (inherited from `ADFsys`) is given by the following ASP code:

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

The functional ASP representation of the same ADF looks thus:

```
s(a). s(b). s(c). s(d).
l(a,c). l(b,b). l(b,c). l(b,d).
ci(a).
co(b). ci(b,1,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).
```

Arguably, the formula representation is easier to read for humans.

3.2   Implementation of Γ_D

Since all of the semantics are defined via the operator Γ_D, we will now present
how the implementation of the operator is done in DIAMOND. The unary pre-
dicate `step` with an arbitrary term is used to apply the operator several times.
The input for the operator is given by the predicates `in` and `out` to represent
mappings to **t** and **f**. The resulting interpretation can be read off the predicates
`valid` and `unsat`. Predicates `fp` and `nofp` denote whether a fixpoint is reached
or not. First, DIAMOND decides which of the mappings to **t** are still of interest
(`cii`) (i.e. which of those can still be satisfied under the given interpretation):

```
ciui(S,J,I) :- lin(X,S,I), not ci(S,J,X), ci(S,J).
ciui(S,J,I) :- lout(X,S,I), ci(S,J,X).
cii(S,J,I) :- not ciui(S,J,I), ci(S,J), step(I).
```

The predicates `lin` and `lout` are those links between arguments which are
already decided by the given three-valued interpretation. The binary predic-
ate `ci` (resp. `co`) is just the projection of its tertiary version to express that
at least one predicate with a specific statement occurs in a specific acceptance
condition. The treatment of the interesting mappings to **f** (`coi`) is dual:

```
coui(S,J,I) :- lin(X,S,I), not co(S,J,X), co(S,J).
coui(S,J,I) :- lout(X,S,I), co(S,J,X).
coi(S,J,I) :- not coui(S,J,I), co(S,J), step(I).
```

Afterwards it is checked whether there exist two-valued extensions of the given
interpretation that are a model or not, which is denoted by the predicates `pmodel`
(resp. `imodel`). Then a statement can be seen to be `valid` (resp. `unsat`) if there
does not exist an interpretation which is not a model (is a model). The predicate
`verum` (resp. `falsum`) represents that the acceptance condition is always true
(resp. false).

```
pmodel(S,I) :- cii(S,J,I). pmodel(S,I) :- verum(S), step(I).
pmodel(S,I) :- not lin(S,I), ci(S), step(I).
pmodel(S,I) :- not lin(S,I), ci(S), step(I).
valid(S,I) :- pmodel(S,I), not imodel(S,I).

imodel(S,I) :- coi(S,J,I).
imodel(S,I) :- falsum(S), step(I).
imodel(S,I) :- not lin(S,I), co(S), step(I).
unsat(S,I) :- imodel(S,I), not pmodel(S,I).
```

At last, either `nofp` or `fp` is deduced. To achieve this, DIAMOND checks whether the application of the operator resulted in an interpretation that is different from the given one.

```
nofp(I) :- in(X,I), not valid(X,I), step(I).
nofp(I) :- valid(X,I), not in(X,I), step(I).
nofp(I) :- out(X,I), not unsat(X,I), step(I).
nofp(I) :- unsat(X,I), not out(X,I), step(I).
fp(I) :- not nofp(I), step(I).
```

3.3 Semantics

The admissible model is computed by the use of a guess and check approach. At first a three-valued interpretation is guessed, by an assignment of the statements to be in, out, or neither. The last two lines remove all guesses which violate the definition of the admissible model (i.e. check which guesses are right):

```
step(0).
{in(S,0):s(S)}.
{out(S,0):s(S)}.
:- in(S,0), out(S,0).
:- in(S), not valid(S,0).
:- out(S), not unsat(S,0).
```

The complete model encoding uses the same concept as used for the admissible model. The only difference is that the guessed model needs to be a fixpoint. To this effect the last two rules of the above encoding are replaced by the constraint ":- nofp(0).".

To compute the grounded model, we need to apply Γ_D until a fixpoint is reached. This is done via a sequence of steps, where the result of one step is taken as the used given interpretation for the next step:

```
maxit(I) :- I:={s(S)}. step(0).
in(S,I+1) :- valid(S,I). out(S,I+1) :- unsat(S,I).
step(I+1) :- step(I), not maxit(I).
in(S) :- fp(I), in(S,I).
out(S) :- fp(I), out(S,I).
udec(S) :- fp(I), s(S), not in(S), not out(S).
```

Note that we use the number of statements as the upper bound on the number of operator applications as this is the maximal number of steps needed to reach a fixpoint.

To implement the model semantics, the operator is not essential: as the model is only two-valued, there do not remain undecided parts. So each variable is mapped to a truth-value and therefore every acceptance condition may only map to one value (i.e. $\mathbf{t}$ or $\mathbf{f}$). The encoding just guesses a two-valued interpretation and checks whether the guessed interpretation agrees with the acceptance conditions of each statement or not. The stable model combines the encoding for models with the operator encoding to check for each model whether it is also the grounded extension of its reduced ADF or not.

4   Discussion and Future Work

We presented the DIAMOND software system that uses answer set programming to compute models of abstract dialectical frameworks under various semantics. DIAMOND can be seen as a continuation of the trend to utilise ASP for implementing abstract argumentation. The most important existing tool in this line of work is the ASPARTIX system[5] [14] for computing extensions of Dung argumentation frameworks.

Quite recently, Ellmauthaler and Wallner presented their system `ADFsys`[6] for determining the semantics of ADFs [15]. Since their system likewise uses answer set programming, it is natural to ask where the differences lie. For one, after the discovery of several examples where some original ADF semantics do not behave as intended, Brewka et al. [6] proposed revised and generalised versions of these semantics. The DIAMOND system implements the new semantics while `ADFsys` still computes the old versions. For another, `ADFsys` relies solely on the representation of acceptance conditions via propositional formulas, while DIAMOND can additionally deal with functional representations. Due to the new semantics it is not trivial to compare those two systems. In fact only the model and the grounded semantics have not changed. During preliminary tests, we used different methods to generate randomised ADF instances. Depending on the used generation method, DIAMOND could compete with `ADFsys` and even outperform it. Alas, there were also instances for which `ADFsys` outperformed DIAMOND. We consider it an important future task to determine specific classes of ADFs that distinguish the two systems, and to connect these ADF classes to possible real-world applications.

To adapt `ADFsys` to the new semantics, it would be needed to decide at each operation of Γ_D which acceptance formulae are (under the given three-valued interpretation) irrefutable (resp. unsatisfiable). To solve such an embedded co-**NP**

---

[5] ASPARTIX is available at `http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/`

[6] `ADFsys` is available at `http://www.dbai.tuwien.ac.at/research/project/argumentation/adfsys/`

problem it would be necessary to use the saturation technique or similar concepts, which will make the use of disjunctive logic programs obligatory. Therefore there would also be issues with more complex semantics (like the preferred semantics). There the use of meta-ASP would conflict with the use of saturation in the disjunctive program.

Apart from the semantics implemented in this paper, there are also ADF semantics that DIAMOND cannot yet deal with – these remain for future work. For example, the preferred semantics is based on maximisation, and so we will need meta-ASP to implement that. In general, ADFs are a quite new formalism, and we expect that further ADF semantics will be defined in the future. Naturally, we plan to implement these new semantics using the infrastructure already available through DIAMOND.

Another future research interest concerns a possible practical application for ADFs: We intend to analyse discussions in social media, where opinions and viewpoints can be modelled by statements that are in some relation to each other. ADF semantics can guide the respective online community, for example as to what positions everybody can agree on, or how a group decision can be justified. Such an approach was proposed by Toni and Torroni [16] as a possible application of assumption-based argumentation frameworks [17]. However, assumption-based argumentation inherits the expressiveness limitations of abstract argumentation, that is, it can also express only attack relationships between statements. We expect that ADFs with their greater expressiveness are better suited to model online interactions in social media.

A similar application of argumentation in online social communities is the approach by Snaith et al. [18]. They utilise their database for arguments in the Argument Interchange Format [19] to capture discussions via different blogging-sites and use their tool TOAST [20] to compute an acceptable consensus about the issues under discussion. Again we think that ADFs are more suitable for this application due to their expressiveness.

References

1. Dung, P.M.: On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. Artificial Intelligence **77** (1995) 321–358
2. Nielsen, S.H., Parsons, S.: A generalization of Dung's abstract framework for argumentation: Arguing with sets of attacking arguments. In: Argumentation in Multi-Agent Systems. Volume 4766 of LNCS., Springer (2006) 54–73
3. Modgil, S.: Reasoning about preferences in argumentation frameworks. Artif. Intell. **173**(9-10) (2009) 901–934
4. Boella, G., Gabbay, D.M., van der Torre, L., Villata, S.: Meta-argumentation modelling I: Methodology and techniques. Studia Logica **93**(2–3) (2009) 297–355
5. Brewka, G., Woltran, S.: Abstract Dialectical Frameworks. In: KR. (2010) 102–111
6. Brewka, G., Ellmauthaler, S., Strass, H., Wallner, J.P., Woltran, S.: Abstract Dialectical Frameworks Revisited. In: IJCAI, AAAI Press (August 2013) To appear.
7. Amgoud, L., Cayrol, C.: On the acceptability of arguments in preference-based argumentation. In: UAI, Morgan Kaufmann (1998) 1–7

8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. (1988) 1070–1080

9. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2) (2011) 105–124

10. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers (2012)

11. Schimpf, J., Shen, K.: ECLiPSe – from LP to CLP. CoRR **abs/1012.4240** (2010)

12. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence **15**(3–4) (1995) 289–323

13. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming **11**(4–5) (2011) 821–839

14. Egly, U., Gaggl, S.A., Woltran, S.: Answer-set programming encodings for argumentation frameworks. Argument and Computation **1**(2) (2010) 147–177

15. Ellmauthaler, S., Wallner, J.P.: Evaluating Abstract Dialectical Frameworks with ASP. [21] 505–506

16. Toni, F., Torroni, P.: Bottom-up argumentation. In Modgil, S., Oren, N., Toni, F., eds.: TAFA. Volume 7132 of Lecture Notes in Computer Science., Springer (2011) 249–262

17. Bondarenko, A., Dung, P.M., Kowalski, R.A., Toni, F.: An abstract, argumentation-theoretic approach to default reasoning. Artificial Intelligence **93** (1997) 63–101

18. Snaith, M., Bex, F., Lawrence, J., Reed, C.: Implementing argublogging. [21] 511–512

19. Rahwan, I., Reed, C.: The Argument Interchange Format. In: Argumentation in Artificial Intelligence. Springer (2009) 383–402

20. Snaith, M., Reed, C.: Toast: Online aspic+ implementation. [21] 511–512

21. Verheij, B., Szeider, S., Woltran, S., eds.: Computational Models of Argument - Proceedings of COMMA 2012, Vienna, Austria, September 10-12, 2012. In Verheij, B., Szeider, S., Woltran, S., eds.: COMMA. Volume 245 of Frontiers in Artificial Intelligence and Applications., IOS Press (2012)