The DIAMOND System for Computing with Abstract Dialectical Frameworks*

Stefan ELLMAUTHALER and Hannes STRASS
Computer Science Institute, Leipzig University, Germany

Abstract We present DIAMOND, an implementation of Brewka and Woltran's abstract dialectical frameworks (ADFs). The system uses answer set programming encodings to compute interpretations of ADFs according to various semantics. We evaluate the performance of the system using an actual reasoning problem as opposed to using randomly generated frameworks.

Keywords. abstract argumentation, abstract dialectical frameworks, implementation, answer set programming

1. Introduction

Abstract dialectical frameworks (ADFs) [Brewka and Woltran, 2010] are a powerful generalisation of Dung argumentation frameworks (AFs) that have recently received considerable attention in the literature [Brewka et al., 2013; Strass, 2013; Strass and Wallner, 2014]. However, most of this attention has been theoretical in nature. To open up more practical research avenues for ADFs, we have developed a software suite – DIAMOND – for computing interpretations for ADFs under various semantics. This paper is the first comprehensive description of DIAMOND.

An important prerequisite for implementing an approach – to us – is a good knowledge of the computational complexity of the problems to be implemented. This has been recently studied for ADFs [Strass and Wallner, 2014], with significant results: ADFs are more complex than AFs; more precisely, they are one level up in the polynomial hierarchy. However, and more significantly, there is a subclass of ADFs – bipolar ADFs – that is computationally as complex as AFs, but offers more modelling capacities. In bipolar ADFs we can not only express AF-like individual attack, but also set attack, set support and individual support; these notions can also be combined.

ADFs obtain their generality from abstracting away not only from the internals of arguments, as AFs do, but also from the internals of links between arguments. While a link between two arguments in an AF is always an individual attack, a link between to arguments in an ADF (called statements there) can be much more, as mentioned above. What the link is, actually, is specified by the acceptance condition associated with each statement. The acceptance condition of a statement s can be represented by a propositional formula φ_s over the parents of s, that is, over those statements with a direct link to s.

*This research has been funded by DFG (projects FOR 1513 and BR-1817/7-1).

The AF-like relationship where statements a and b individually attack c can then be expressed by $\varphi_c = \neg a \wedge \neg b$. That is, c is accepted (true) if neither of its attackers is accepted (true). A set attack from a and b to c is written as $\varphi_c = \neg a \vee \neg b$ where c is only rejected if both a and b are accepted. The same works for support: $\varphi_c = a \wedge b$ means that c needs support from both a and b, and $\varphi_c = a \vee b$ says that c can be accepted if at least one of a or b is accepted. And now recall that all these additional means of expression come *for free* in terms of computational cost!

Implementing approaches to abstract argumentation has somewhat of a tradition, as is witnessed by the several implementations for Dung AFs: ASPARTIX [Egly et al., 2010], Dung-O-Matic (`http://www.arg.dundee.ac.uk/?page_id=279`), and ConArg [Bistarelli and Santini, 2011]. As ADFs generalise AFs, our system DIAMOND is also yet another AF implementation (utilising the ASPARTIX input format), but at the same time so much more: DIAMOND can also compute the semantics for (bipolar) ADFs in various different input formats, decide whether a given ADF is bipolar, or transform an ADF from one representation into another. To achieve its ends, DIAMOND employs the declarative programming paradigm of *answer set programming* [Gebser et al., 2012].

As the final contribution of this paper, we perform an experimental evaluation of DIAMOND. However, contrary to previous works in this vein [Bistarelli et al., 2013], we do not use randomly generated graphs. Rather, we encode an actual reasoning problem – the factorisation of integers – into abstract argumentation. We consider this an important contribution in its own right as it shows that A(D)F implementations can be used to solve non-trivial problems. Indeed, our results suggest that abstract argumentation implementations may be more powerful than previously believed, being able to solve problems for AFs with around 20,000 arguments in less than 10 seconds.

In the rest of the paper, we first give a short background on ADFs. We next describe how DIAMOND is implemented and give some explanation on its usage; in Section 4 we describe our experimental setup and evaluation. The last section concludes with a discussion and some avenues for future work. An earlier version of this paper appeared at a workshop [Ellmauthaler and Strass, 2013].

2. Background

For a serious lack of space, we cannot present all necessary background on abstract dialectical frameworks (ADFs), but have to restrict ourselves to the definition of semantics and refer the interested reader to [Brewka et al., 2013]. Roughly, ADFs are directed graphs whose nodes represent statements. Each statement comes with an acceptance condition, that can be represented by a propositional formula over its parent statements. The semantics of ADFs are defined via the *characteristic operator* Γ_D of an ADF D. While most of the semantics we define now have appeared in the literature before [Brewka et al., 2013], some are new, but straightforward to define [Strass, 2013]. These are the (three-valued) conflict-free, naive and stage semantics, as well as so-called semi-models, an ADF version of AFs' semi-stable semantics (defined as *admissible stages* by Verheij [1996]).

Definition 1. Let D be an ADF. A three-valued interpretation v is

- *admissible* iff $v \leq_i \Gamma_D(v)$;
- *preferred* iff it is $\leq_i$-maximal with respect to being admissible;[2]
- a *semi-model* iff $\mathbf{u}(v)$ is $\subseteq$-minimal with respect to being admissible;
- *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;
- *conflict-free* iff for all $s \in S$ we have:
  if $v(s) = \mathbf{t}$ then $\Gamma_D(v)(s) \neq \mathbf{f}$, and if $v(s) = \mathbf{f}$ then $\Gamma_D(v)(s) = \mathbf{f}$.
- *naive* iff it is $\leq_i$-maximal with respect to being conflict-free;
- *stage* iff $\mathbf{u}(v)$ is $\subseteq$-minimal with respect to being conflict-free.

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 all $s \in \mathbf{t}(v)$ are true in the grounded semantics of the reduced ADF $D^{\mathbf{t}(v)} = (\mathbf{t}(v), L \cap (\mathbf{t}(v) \times \mathbf{t}(v)), C^{\mathbf{t}(v)})$, where for $s \in \mathbf{t}(v)$ the reduced acceptance formula is given by $\varphi_s[r/\mathbf{f} : v(r) = \mathbf{f}]$.

3. The DIAMOND System

Our software system DIAMOND is a tool-set aimed at argumentation researchers to compute various interpretations with respect to the semantics for a given ADF and to do different checks (for example, test whether an ADF is bipolar) and transformations. The heart of the system is a collection of answer set programming encodings which are designed around the *Potsdam Answer Set Solving Collection (Potassco)* [Gebser et al., 2011a] with *clingo* as the primarily used solver. 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 *sourceforge*: `https://sourceforge.net/projects/diamond-adf/`. To make the usage of our system more convenient for the user, it offers a *Python*-based command line interface. Different switches are used to designate the desired semantics, the used framework (i.e. Dung AFs, bipolar ADFs, prioritised ADFs, and general ADFs), the input file, and its format. It is not necessary to specify the format if it is indicated by the right file extension.

3.1. Input Format

DIAMOND supports these different input formats (and conversions between them):

(i) Propositional formula representation	file extension `.adf`
(ii) Bipolar propositional formula representation	file extension `.badf`
(iii) Priority-based representation	file extension `.padf`
(iv) ASPARTIX syntax for Dung AFs	file extension `.af`
(v) Functional/extensional representation	any other file extension

In the following we will describe how the syntax of those formats are defined and how they are utilised by DIAMOND. All of the input formats use the unary predicate `s` to specify the statements of the ADF.

Formats (i) and (ii) use the binary predicate `ac(s,`φ`)` to associate to each statement `s` one propositional formula φ. Each formula φ is constructed in the

[2] For an interpretation v over statements S, define $\mathbf{u}(v) = \{s \in S \mid v(s) = \mathbf{u}\}$, likewise for $\mathbf{t}, \mathbf{f}$.

usual inductive way, where atomic formulae are statements and the truth constants *true* – `c(v)` – and *false* – `c(f)` –, and the connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ are written as functions `neg, and, or, imp, iff`.

Input in form of (ii) – bipolar ADFs with acceptance formulas – additionally requires the specification of link types. Those are given by the predicates `att(a,b)` and `sup(a,b)`, which means that `a` attacks (respectively supports) `b`.

Input format (iii) implements the approach on prioritised ADFs by Brewka et al. [2013], to which we refer the interested reader for further details. To describe a prioritised ADF, 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)`.

Format (iv) is the input format of ASPARTIX [Egly et al., 2010], which is a collection of answer set programming encodings to compute extensions of Dung's AFs. (The system ConArg [Bistarelli and Santini, 2011] uses the same input format.) There, an argument is represented by the unary predicate `arg` and the attack relation is modelled by the binary predicate `att`.

Input format (v), the extensional representation, defines links between two statements with 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`. (More on the extensional representation format can be found in [Ellmauthaler and Strass, 2013].)

Example 1. Consider this ADF given in (bipolar) propositional formula representation (i) with link-type-information of (ii) (on the left) and the functional ASP representation of the same ADF (on the right):

```
s(a). s(b). s(c). s(d).
ac(a,c(v)).
ac(b, b).
ac(c, and(a,b)).
ac(d, neg(b)).
sup(b,b). att(b,d).
sup(b,c). sup(a,c).
```

```
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).
```

DIAMOND provides different format conversions to switch to an appropriate one. In case of arbitrary and prioritised ADFs, the input is converted to the extensional format, where the size of the instance may grow considerably compared to the propositional formula or priority-based representation. If the ADF is bipolar and the link-types are given, no conversion will be done. There is also no conversion for Dung AFs if they are given in ASPARTIX format.

We have chosen this functional representation of acceptance conditions on general ADFs for pragmatic reasons. An alternative would have been to represent acceptance conditions by propositional formulas, what is already done for bipolar ADFs. In this case, computing a single step of the operator would entail solving several NP-hard problems. The standard way to solve these is the

use of saturation [Eiter and Gottlob, 1995], which however causes undesired side-effects when employed together with meta-ASP [Gebser et al., 2011b]. Furthermore, other ADF semantics (e.g. preferred) utilise concepts like $\leq_i$-maximality, which also require the use of meta-ASP. We plan to extend DIAMOND to further semantics and therefore chose the functional representation of acceptance conditions to forestall potential implementation issues. To reduce the instance size for computationally easier problems (like bipolar ADFs or Dung AFs), we have implemented less complex operators such that propositional formula representation and ASPARTIX format can be utilised in a native manner. Here the modularity of our encodings is important and useful: Instead of encoding different versions of semantics for ADFs, bipolar ADFs and AFs, we have one encoding of each semantics, and three different encodings of the operator Γ_D [Brewka et al., 2013] (general ADFs, bipolar ADFs, AFs), which are combined by DIAMOND as needed.

3.2. Implementation Details

DIAMOND in version 1.0.0 is a *Python* program (needing Python 3.3 onwards) that utilises *clingo 4.3.0* to compute the interpretations of a given abstract dialectical framework. In general most parts of the computation are done with answer set programming encodings for clingo. The Python program analyses the given input format and the desired results to decide which (if any) input format conversions shall be done before the solving starts. In addition it manages the selection of the used encodings to provide an easy way to query for interpretations.

All of the encodings are designed in a modular manner. The semantics for the different types of abstract dialectical frameworks are defined over a specific operator. As this operator is the only difference, DIAMOND defines those semantics such that the operators can be substituted easily. The $\leq_i$-optimisation for different semantics is handled via two clingo calls, managed by Python to avoid the need of optimisation constructs in ASP. To compute the link-types of bipolar ADFs, the already mentioned saturation technique has been utilised, as the decision for one link is already coNP-hard. Due to the possibility that a link can be attacking and supporting it is also required to use the union of all answers of the link-type computation instead of one, which is also handled by the Python program.

4. Experimental Evaluation

To analyse the runtime behaviour of DIAMOND, we performed a series of practical experiments. Current work on comparing state-of-the art AF implementations create random AFs for running experiments, see [Bistarelli et al., 2013] for an example. Instead of creating random instances of AFs for testing, we decided to encode an actual reasoning problem – the factorisation of integers – into abstract argumentation. This allows to analyse not only the behaviour of different implementations but also the influence on problem encoding on this behaviour.

4.1. Experiment Design

The encoding is inspired by the work of Horie and Watanabe [1997] in SAT solving, who proposed this method to create hard, but satisfiable instances for the SAT problem in propositional logic. Roughly, the approach works as follows: For a given natural number n, we guess two distinct prime numbers p and q that both

have exactly n significant bits in binary representation. Then we compute the product $z = p \cdot q$ and create the problem instance associated with z by encoding the formula $(\exists x, y)x \cdot y = z$ into abstract argumentation, where x, y (variables) and z (a number) are represented in binary. Clearly this problem has exactly two solutions, $\{x \mapsto p, y \mapsto q\}$ and $\{x \mapsto q, y \mapsto p\}$. To obtain a problem encoding with exactly one solution, we add the constraint $x < y$.

The major advantage of such a problem-encoding approach is that we do not only measure computation time of our implementation, but can also check its correctness: we know the single intended solution to the problem and can therefore check whether (a) the implementation does return a single interpretation, and (b) can inspect the interpretation to make sure that the right p and q have been computed. Furthermore, the approach is scalable, since the problem instance size basically depends on the number n of bits in p and q.

The main technical issue is the encoding of the formula $(\exists x, y)x \cdot y = z$ into AFs and ADFs. In both cases, we can make use of basic standard knowledge in technical computer science, and AFs' and ADFs' close relationship to classical propositional logic. The relationship is clear for ADFs due to their Boolean acceptance functions. For AFs the relationship is equally straightforward, but perhaps less well-known [Gabbay, 2011]. Roughly, AFs can be seen as Boolean circuits where the only kind of gate is a NOR gate (computing a negated logical disjunction). This stems from the fact that an argument is accepted (its output is true) if and only if none of its attackers (inputs) is accepted (true). Making use of this, it is then mostly straightforward to devise the circuit for $(\exists x, y)x \cdot y = z \wedge x < y$. We illustrate the construction with the smallest possible example where $n = 2$, $p = 2$, $q = 3$ and thus $z = 6$. For a number like z, we use z_i to refer to its i-th bit in binary. For example, asserting that $z = 6 = 2^2 + 2^1$ results in the formula $\varphi_{z=6} = \neg z_0 \wedge z_1 \wedge z_2 \wedge \neg z_3$. To encode $x \cdot y = z$, we express all possible multiplications of two 2-bit numbers x and y into a 4-bit number z. This is done by a sequence of additions $z = x \cdot 2^0 \cdot y_0 + x \cdot 2^1 \cdot y_1 + x \cdot 2^2 \cdot y_2$. Multiplication by powers of two is easily realised by just shifting bits; for a single one of such additions, we can thus use a sequence of one-bit full adders. A one-bit full adder takes as input two bits a and b and an incoming carry bit c_i, and yields as output the sum s of the two bits and an outgoing carry bit c_o. The functions for the two outputs are given by $s \equiv (a \oplus b) \oplus c_i$ and $c_o \equiv (a \wedge b) \vee (c_i \wedge (a \oplus b))$, where $\oplus$ is exclusive-or.

This yields a set of propositional formulas whose models correspond to solutions of the factorisation problem. As it turns out, most of the formulas are of the form $s \equiv \varphi$ or literals and thus perfectly amenable to be compiled into an ADF. To express the formulas as an AF, additional internal NOR gates (arguments) are needed to emulate all other kinds of gates. The number of additional gates is linear in the size of the original formula set, but leads to considerable encoding sizes, where the ADF-based encoding is an order of magnitude smaller than the AF-based encoding of the same problem instance.[3]

[3]The size of an AF is the number of arguments plus the number of attacks; the size of an ADF is the sum of sizes of all acceptance formulas, where the size of a formula is the number of atoms and connectives occurring in it.

4.2. Experimental Results

We created random instances of the factorisation problem with varying numbers of bits, $n = 5, \ldots, 20$ with 50 instances for each n. We encoded each problem instance both as AF (under stable extension semantics) and as ADF (under model semantics). We then used DIAMOND to solve the problems and recorded the runtime. To have a comparison baseline, we also used the ASPARTIX encoding of stable extension semantics[4] and the same version of clingo that underlies DIAMOND. The experiments were performed on a desktop PC with Intel Core Duo Processor and 4GB RAM running ubuntu Linux 12.04. The timeout for each instance was set to 300 seconds. The results are discussed in Figure 1 below.

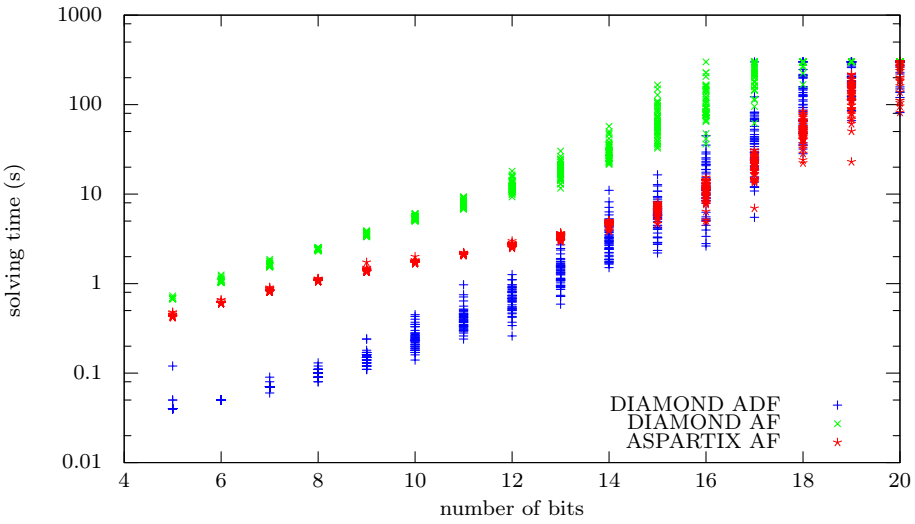

Figure 1. Solving the factorisation problem in three different ways: DIAMOND + ADF encoding, DIAMOND + AF encoding, ASPARTIX + AF encoding. For small to medium problems of up to 13 bits, running DIAMOND on the ADF encoding is clearly the best choice. From 14 bits on, running ASPARTIX on the AF encoding becomes competitive. While the plot allows to compare the behaviour of DIAMOND on AFs with that of ASPARTIX on AFs, we remind the reader that this comparison is not exactly fair since DIAMOND implements general ADFs in contrast to ASPARTIX only treating AFs. However, we can observe that for one and the same implementation, DIAMOND, the ADF encoding clearly outperforms the AF encoding. In general, it can be seen that the experiments went to the hardware limit of the machine we used. (Already for 15 bits, the number of arguments plus attacks in the AF encoding is almost 50,000.)

5. Discussion

We have described the DIAMOND system, an implementation of abstract dialectical frameworks. DIAMOND employs the ASP solver clingo to compute interpretations of ADFs for various semantics. The system can read (subclasses of) ADFs in several input formats, among them AFs in the ASPARTIX format. We analysed the computational performance of DIAMOND using a novel approach to test instance generation: we encoded the factorisation problem of natural numbers into AFs and ADFs and used DIAMOND to factorise numbers of up to 40 bits in binary.

[4]The encoding was downloaded from `http://www.dbai.tuwien.ac.at/research/project/argumentation/systempage/Data/stable.dl`.

In the future, we plan to incorporate into DIAMOND the novel solving paradigm of *reactive answer set solving* [Gebser et al., 2012]. There, a solver is not called once on a specific instance, but rather receives continuous input and produces continuous output. This will help us deal with ADFs' increased computational complexity in a clean way. Another goal of us is to incorporate further, less abstract input languages into DIAMOND.

References

Stefano Bistarelli and Francesco Santini. Modeling and solving AFs with a constraint-based tool: ConArg. In *TAFA*, pages 99–116, 2011.

Stefano Bistarelli, Fabio Rossi, and Francesco Santini. A first comparison of abstract argumentation systems: A computational perspective. In *CILC*, pages 241–245, 2013.

Gerhard Brewka and Stefan Woltran. Abstract Dialectical Frameworks. In *KR*, pages 102–111, 2010.

Gerhard Brewka, Stefan Ellmauthaler, Hannes Strass, Johannes Peter Wallner, and Stefan Woltran. Abstract Dialectical Frameworks Revisited. In *IJCAI*. IJCAI/AAAI, August 2013.

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

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

Stefan Ellmauthaler and Hannes Strass. The DIAMOND System for Argumentation: Preliminary Report. In *ASPOCP*, pages 97–107, 2013.

Dov M. Gabbay. Dung's argumentation is essentially equivalent to classical propositional logic with the Peirce-Quine dagger. *Logica Universalis*, 5(2):255–318, 2011.

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

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

Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4–5): 821–839, 2011b.

Satoshi Horie and Osamu Watanabe. Hard instance generation for SAT. In *Algorithms and Computation*, pages 22–31. Springer Berlin Heidelberg, 1997.

Hannes Strass. Approximating operators and semantics for abstract dialectical frameworks. *Artificial Intelligence*, 205:39–70, December 2013.

Hannes Strass and Johannes Peter Wallner. Analyzing the Computational Complexity of Abstract Dialectical Frameworks via Approximation Fixpoint Theory. In *KR*, pages 101–110, Vienna, Austria, July 2014.

Bart Verheij. Two approaches to dialectical argumentation: Admissible sets and argumentation stages. In *Proceedings of NAIC*, pages 357–368, 1996.