Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)

Solving Advanced Argumentation Problems with Answer-Set Programming

Gerhard Brewka
Universität Leipzig, Leipzig, Germany

Martin Diller, Georg Heissenberger, Thomas Linsbichler, and **Stefan Woltran**
TU Wien, Vienna, Austria

Abstract

Powerful formalisms for abstract argumentation have been proposed. Their complexity is often located beyond NP and ranges up to the third level of the polynomial hierarchy. The combined complexity of Answer-Set Programming (ASP) exactly matches this complexity when programs are restricted to predicates of bounded arity. In this paper, we exploit this coincidence and present novel efficient translations from abstract dialectical frameworks (ADFs) and GRAPPA to ASP. We also empirically compare our approach to other systems for ADF reasoning and report promising results.

Introduction

Argumentation is an active area of research with applications in legal reasoning (Bench-Capon and Dunne 2005), decision making (Amgoud and Prade 2009), e-governance (Cartwright and Atkinson 2009) and multi-agent systems (McBurney, Parsons, and Rahwan 2012). Dung's argumentation frameworks (Dung 1995), AFs for short, are widely used in argumentation. They focus entirely on conflict resolution among arguments, treating the latter as abstract items without logical structure. Although AFs are quite popular, various generalizations aiming for easier and more natural representations have been proposed, see (Brewka, Polberg, and Woltran 2014) for an overview.

We focus on two such generalizations, namely ADFs (Brewka and Woltran 2010; Brewka et al. 2013) and GRAPPA (Brewka and Woltran 2014), which are expressive enough to capture many of the other available frameworks. Reasoning in ADFs spans the first three levels of the polynomial hierarchy (Strass and Wallner 2015). These results carry over to GRAPPA (Brewka and Woltran 2014). ADFs, in particular, have received increasing attention recently, see e.g. (Gaggl and Strass 2014; Booth 2015; Al-Abdulkarim, Atkinson, and Bench-Capon 2016).

Two approaches to implement ADF reasoning have been proposed in the literature. `QADF` (Diller, Wallner, and Woltran 2014; 2015) encodes problems as quantified Boolean formulas (QBFs) such that a single call of a QBF solver delivers the result. `DIAMOND` (Ellmauthaler and Strass 2014; 2016), on the other hand, employs ASP. Since `DIAMOND` relies on *static* encodings, i.e. the encoding does not change for different framework instances, this approach is limited by the *data complexity* of ASP (which only reaches the second level of the polynomial hierarchy (Eiter and Gottlob 1995; Eiter, Gottlob, and Mannila 1997)). Therefore, preferred semantics (which comprise the hardest problems for ADFs and GRAPPA) need a more complicated treatment involving two consecutive calls to ASP solvers with a possibly exponential blowup for the input of the second call. A GRAPPA interface has been added to `DIAMOND` (Berthold 2016), but we are not aware of any systems for GRAPPA not requiring a translation to ADFs as intermediate step.

In this paper, we introduce a new method for implementing reasoning tasks related to both ADFs and GRAPPA such that even the hardest among the problems are treated with a *single* call to an ASP solver. The reason for choosing ASP is that the rich syntax of GRAPPA is much easier captured by ASP than by other formalisms like QBFs. Our approach makes use of the fact that the *combined complexity* of ASP for programs with predicates of bounded arity (Eiter et al. 2007) exactly matches the complexity of ADFs and GRAPPA. This approach is called *dynamic*, because the encodings are generated individually for every instance. We are not aware of any comparable work in argumentation.

More specifically, we provide encodings for admissible, complete, and preferred semantics for ADFs and GRAPPA. Depending on the semantics (and their complexity) the encodings yield normal or, in the case of preferred semantics, disjunctive programs. We specify the encodings in a modular way, which makes our approach amenable for extensions to other semantics. We also provide a preliminary experimental analysis in order to determine the potential of this method.

Background

ADFs. An ADF is a directed graph whose set of nodes S represents statements. 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 conditions under which s is acceptable.

It is convenient to represent the acceptance conditions as a collection $C = \{\varphi_s\}_{s \in S}$ of propositional formulas. This leads to the logical representation of ADFs we will use in this paper where an ADF D is a pair (S, C) with the set of

links L implicitly given as $(a, b) \in L$ iff a appears in φ_b.

Semantics assign to ADFs a collection of (3-valued) interpretations, i.e. mappings of the statements to truth values $\{1, 0, \mathbf{u}\}$, denoting true, false and undecided, respectively. The three truth values are partially ordered by $\leq_i$ according to their information content: we have $\mathbf{u} <_i 1$ and $\mathbf{u} <_i 0$ and no other pair in $<_i$. The information ordering $\leq_i$ extends in a straightforward way to interpretations 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 $s \in S$.

An interpretation v is 2-valued if all statements are mapped to 1 or 0. For interpretations v and w, we say that w extends v iff $v \leq_i w$. We denote by $[v]_2$ the set of all completions of v, i.e. 2-valued interpretations that extend v.

For an ADF $D = (S, C)$, $s \in S$ and an interpretation v, the characteristic function $\Gamma_D(v) = v'$ is given by

$$v'(s) = \begin{cases} 1 \text{ if } w(\varphi_s) = 1 \text{ for all } w \in [v]_2 \\ 0 \text{ if } w(\varphi_s) = 0 \text{ for all } w \in [v]_2 \\ \mathbf{u} \text{ otherwise} \end{cases}$$

That is, the operator returns an interpretation mapping a statement s to 1 (resp. 0) iff all 2-valued interpretations extending v evaluate φ_s to true (resp. false). Intuitively, Γ_D checks which truth values can be justified based on the information in v and the acceptance conditions. Note that Γ_D is defined on 3-valued interpretations, while we evaluate acceptance conditions under their 2-valued extensions.

Given an ADF $D = (S, C)$, an interpretation v is *admissible* w.r.t. D if $v \leq_i \Gamma_D(v)$; it is *complete* w.r.t. D if $v = \Gamma_D(v)$; it is *preferred* w.r.t. D if v is maximal admissible w.r.t. $\leq_i$. As shown in (Brewka et al. 2013) these semantics generalize the corresponding notions defined for AFs. For $\sigma \in \{adm, com, prf\}$, $\sigma(D)$ denotes the set of all admissible (resp. complete, preferred) interpretations w.r.t. D.

Example 1. Given the ADF $D = (\{a, b\}, \{\varphi_a{=}a \vee \neg b, \varphi_b{=}\neg a\})$ and $v_1 = \{a{\to}\mathbf{u}, b{\to}\mathbf{u}\}$, $v_2 = \{a{\to}1, b{\to}\mathbf{u}\}$, $v_3 = \{a{\to}1, b{\to}0\}$, $v_4 = \{a{\to}0, b{\to}1\}$, we get $adm(D) = \{v_1, v_2, v_3, v_4\}$, $com(D) = \{v_1, v_3, v_4\}$ (note that $\Gamma_D(v_2){=}v_3$, thus $v_2 \notin com(D)$), and $prf(D) = \{v_3, v_4\}$.

GRAPPA. ADFs are particularly useful as target formalism of translations from graph-based approaches. This raises the question whether an ADF style semantics can be directly defined for arbitrary labelled graphs, thus circumventing the need for any translations. GRAPPA (Brewka and Woltran 2014) fulfills exactly that goal. GRAPPA allows argumentation scenarios to be defined using arbitrary directed edge-labelled graphs. The nodes in S represent statements, as before. Labels of links, which may be chosen as needed, describe the type of relationship between a node and its parents. As for ADFs, each node has its own acceptance condition, and the semantics of a graph is defined in terms of 3-valued interpretations. The major difference is that acceptance conditions no longer depend on the acceptance status of the parents of a node, but on the labels of its active incoming links, where a link is active if its source node is true and a label is active if it is the label of an active link. More precisely, since it can make an important difference whether a specific label appears once or more often on active links, the acceptance condition depends on the multiset of active labels of a node, that is, an acceptance condition is a function $c : (L \to \mathbb{N}) \to \{t, f\}$, where L is the set of all labels.

The characteristic function Γ_G of a graph G needs to be adapted accordingly: again the function takes a 3-valued interpretation v and produces a new one v'; again v' is constructed by considering all 2-valued extensions v'' of v, picking a classical truth value only if all extensions produce the same result. But this time an intermediate step is needed to determine the truth value of a node s: one first has to determine the multiset of active labels of s generated by v''. The acceptance function c then takes this multiset as argument and produces the truth value induced by v''.

With this new characteristic function the semantics of a graph G can be defined as for ADFs, that is, an interpretation v is *admissible* w.r.t. G if $v \leq_i \Gamma_G(v)$; it is *complete* w.r.t. G if $v = \Gamma_G(v)$; it is *preferred* w.r.t. G if v is maximal admissible w.r.t. $\leq_i$. As before $\sigma(G)$ ($\sigma \in \{adm, com, prf\}$) denotes the set of all respective interpretations.

GRAPPA acceptance functions are specified using *acceptance patterns* over a set of labels L defined as follows:

- A *term* over L is of the form $\#(l)$, $\#_t(l)$ (with $l \in L$), or min, min_t, max, max_t, sum, sum_t, $count$, $count_t$.
- A *basic acceptance pattern* (over L) is of the form $a_1 t_1 + \cdots + a_n t_n \, R \, a$, where the t_i are terms over L, the a_is and a are integers and $R \in \{<, \leq, =, \neq, \geq, >\}$.
- An *acceptance pattern* (over L) is a basic acceptance pattern or a Boolean combination of acceptance patterns.

A GRAPPA *instance* is a tuple $G = (S, E, L, \lambda, \pi)$ where S is a set of statements, E a set of edges, L a set of labels, λ an assignment of labels to edges, and π an assignment of acceptance patterns over L to nodes. For a multiset of labels $m : L \to \mathbb{N}$ and $s \in S$ the value function val_s^m is:

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

$min_{(t)}$, $max_{(t)}$, $sum_{(t)}$ are undefined in case of non-numerical labels. For $\emptyset$ they yield the neutral element of the corresponding operation, i.e. $val_s^m(sum) = val_s^m(sum_t) = 0$, $val_s^m(min) = val_s^m(min_t) = \infty$, and $val_s^m(max) = val_s^m(max_t) = -\infty$.

Let m and s be as before. For basic acceptance patterns the *satisfaction relation* $\models$ is defined by

$$(m, s) \models a_1 t_1 + \cdots + a_n t_n R \, a \;\text{ iff }\; \sum_{i=1}^{n} \left(a_i \; val_s^m(t_i)\right) \; R \; a.$$

The extension to Boolean combinations is as usual. For each node s the associated acceptance function $c(s)$ is ob-

Table 1: Complexity results for ADFs, GRAPPA and ASP.

	ADF and GRAPPA			ASP bounded arity	
	adm	*com*	*prf*	normal	disjunctive
cred	Σ_2^P	Σ_2^P	Σ_2^P	Σ_2^P	Σ_3^P
skept	trivial	coNP	Π_3^P	Π_2^P	Π_3^P

tained from the corresponding pattern $\pi(s)$ by $c(s)(m) = 1$ iff $(m, s) \models \pi(s)$.

Example 2. Consider the GRAPPA instance G with $S = \{a, b, c\}$, $E = \{(a, b), (b, b), (c, b), (b, c)\}$, $L = \{+, -\}$, all edges being labelled with + except (b, b) with −, and the acceptance condition $\#_t(+) - \#(+) = 0 \wedge \#(-) = 0$ (i.e. all +-links must be active and no −-link is active) for each statement. The following interpretations are admissible w.r.t. G: $v_1 = \{a{\to}\mathbf{u}, b{\to}\mathbf{u}, c{\to}\mathbf{u}\}$, $v_2 = \{a{\to}\mathbf{u}, b{\to}0, c{\to}0\}$, $v_3 = \{a{\to}1, b{\to}\mathbf{u}, c{\to}\mathbf{u}\}$, $v_4 = \{a{\to}1, b{\to}0, c{\to}0\}$. Moreover, $\mathit{com}(G) = \{v_3, v_4\}$ and $\mathit{prf}(G) = \{v_4\}$.

ASP. In Answer-Set Programming (Leone et al. 2006; Brewka, Eiter, and Truszczyński 2011) problems are described as logic programs, which are sets of rules of the form

$$a_1 \vee \ldots \vee a_n \,\mathbf{:}\!-\, b_1, \ldots, b_k, \; \mathit{not}\; b_{k+1}, \ldots, \; \mathit{not}\; b_m$$

where each a_i is a ground atom and *not* stands for *default negation*. We call a rule a *fact* if $n = 0$. An *(input) database* is a set of facts. A rule r is *normal* if $n \leq 1$ and a *constraint* if $n = 0$. A *program* is a finite set of disjunctive rules. If each rule in a program is normal we call the program normal, otherwise disjunctive. Each logic program π induces a collection of so-called *answer sets*, denoted as $\mathcal{AS}(\pi)$, which are distinguished models of the program determined by answer sets semantics; see (Gelfond and Lifschitz 1991) for details.

For non-ground programs, which we use here, rules with variables are viewed as shorthand for the set of their ground instances. Modern ASP solvers offer further additional language features such as built-in arithmetics and aggregates which we make use of in our encodings (we refer to (Gebser et al. 2015) for an explanation).

Complexity. The complexity results which are central for our work are given in Table 1. Here credulous reasoning means deciding whether a statement (resp. atom) is true in at least one interpretation (resp. answer set) of the respective type, skeptical reasoning whether it is true in all such interpretations (resp. answer sets).

The results for ADFs (Strass and Wallner 2015) carry over to GRAPPA, as argued in (Brewka and Woltran 2014). The results for normal and disjunctive ASP-programs we use here refer to the combined complexity for non-ground programs of *bounded* predicate arity (i.e. there exists a constant $n \in \mathbb{N}$ such that the arity of every predicate occurring in the program is smaller than n) and are due to (Eiter et al. 2007). We recall that the combined complexity of arbitrary programs is much higher (NEXP-hard, see e.g. (Eiter, Gottlob, and Mannila 1997)) while data complexity (i.e. the ASP-program is assumed to be static and only the database of the program is changing) is one level lower in the polynomial hierarchy (follows from (Eiter and Gottlob 1995)).

These results indicate that there exist efficient translations for credulous reasoning in ADFs and GRAPPA to non-ground normal programs of bounded arity, while skeptical preferred reasoning needs to be treated with disjunctive programs. We provide such reductions in what follows.

ADF Encodings

We assume an ADF $D = (S, \{\varphi_s\}_{s \in S})$ with $S = \{s_1, \ldots, s_k\}$ for some $k > 0$. We construct ASP encodings π_σ for $\sigma \in \{\mathit{adm}, \mathit{com}, \mathit{prf}\}$ such that there is a certain one to one correspondence between the σ interpretations of D and the answer sets of $\pi_\sigma(D)$. More precisely, we will use atoms $\mathit{ass}(s, x)$ with $s \in S, x \in \{1, 0, \mathbf{u}\}$ to represent ADF interpretations in our encodings. An interpretation v of D and a set of ground atoms (interpretation of an ASP program) I correspond to each other, $v \cong I$, whenever for every $s \in S$,

$$v(s) = x \Leftrightarrow \mathit{ass}(s, x) \in I.$$

We overload $\cong$ to get the correspondence between sets of interpretations and sets of answer sets we aim for.

Definition 1. Given a set of interpretations V and a collection of sets of ground atoms $\mathcal{I}$, we say that V and $\mathcal{I}$ correspond, $V \cong \mathcal{I}$, if (1) for every $v \in V$ there is an $I \in \mathcal{I}$ s.t. $v \cong I$; (2) for every $I \in \mathcal{I}$ there is a $v \in V$ s.t. $v \cong I$.

All encodings will make use of a simple set of facts for the statements of the ADF and for encoding the possible truth values that can be assigned to a statement s by a completion of an interpretation mapping s to $\mathbf{u}$, 1, and 0, respectively.

$$\pi_0(D) := \{\mathit{arg}(s). \mid s \in S\} \cup \{\mathit{lt}(\mathbf{u}, 0).\mathit{lt}(\mathbf{u}, 1).\mathit{lt}(1, 1).\mathit{lt}(0, 0).\}$$

Moreover, the actual guess for an assignment to the statements is used in all encodings as well:

$$\begin{aligned}\pi_{\mathit{guess}} := \{&\mathit{ass}(S, 0) \,\mathbf{:}\!-\, \mathit{not}\; \mathit{ass}(S, 1), \mathit{not}\; \mathit{ass}(S, \mathbf{u}), \mathit{arg}(S).\\ &\mathit{ass}(S, 1) \,\mathbf{:}\!-\, \mathit{not}\; \mathit{ass}(S, \mathbf{u}), \mathit{not}\; \mathit{ass}(S, 0), \mathit{arg}(S).\\ &\mathit{ass}(S, \mathbf{u}) \,\mathbf{:}\!-\, \mathit{not}\; \mathit{ass}(S, 0), \mathit{not}\; \mathit{ass}(S, 1), \mathit{arg}(S).\}\end{aligned}$$

The structure of the ADF D will be reflected in the complex bodies of rules that encode the acceptance conditions (more precisely, these rules will be grounded with respect to the completions $[v]_2$ of the actual interpretation v) of the statements of D. We first give a recursive definition that yields a set of atoms that correspond to the subformulas of the acceptance conditions and that we then use in the bodies of the relevant rules. For the definition of the atoms in question we rely on the ASP built-in arithmetic functions `&` (bitwise AND), `?` (bitwise OR), and `-` (subtraction).

Let ϕ be a propositional formula over S; then the relevant set of atoms is defined as

$$B(\phi) := \begin{cases} B(\phi_1) \cup B(\phi_2) \cup \{V_\phi = V_{\phi_1} \,\&\, V_{\phi_2}\} & \text{if } \phi = \phi_1 \wedge \phi_2 \\ B(\phi_1) \cup B(\phi_2) \cup \{V_\phi = V_{\phi_1} \,?\, V_{\phi_2}\} & \text{if } \phi = \phi_1 \vee \phi_2 \\ B(\psi) \cup \{V_\phi = 1 - V_\psi\} & \text{if } \phi = \neg\psi \\ \emptyset & \text{if } \phi = s \in S \end{cases}$$

where V_ϕ, V_{ϕ_1} V_{ϕ_2} and V_ψ are variables representing the subformulas of ϕ. We now link variables V_s for $s \in S$ to the completion of the guessed assignment, defining the following rule bodies for each $s \in S$.

$$E_s := \{ass(t, Y_t), lt(Y_t, V_t) \mid t \in par(s)\} \cup B(\varphi_s)$$

Finally, we add for every $s \in S$ rules with head $sat(s)$ and $inv(s)$ that fire in case there is some completion of the interpretation guessed in the fragment π_{guess} such that the acceptance condition φ_s evaluates to 1 and 0, respectively.

$$\begin{aligned}\pi_{sat}(D) := \{&sat(s) \text{:-} E_s, V_{\varphi_s} = 1.\\ &inv(s) \text{:-} E_s, V_{\varphi_s} = 0. \mid s \in S\}\end{aligned}$$

Admissible and Complete Semantics

The encoding $\pi_{adm}(D)$ is based on the following property:

Observation 1. *An interpretation v for an ADF D is admissible iff for every $s \in S$ it is the case that*

- *if $v(s) = 1$ then there is no $w \in [v]_2$ s.t. $w(\varphi_s) = 0$,*
- *if $v(s) = 0$ then there is no $w \in [v]_2$ s.t. $w(\varphi_s) = 1$.*

The encoding now results from compounding the program fragments $\pi_0(D)$, π_{guess}, and $\pi_{sat}(D)$ together with ASP constraints which filter out assignments corresponding to interpretations of D violating Observation 1.

$$\begin{aligned}&\pi_{adm}(D) := \pi_0(D) \cup \pi_{guess} \cup \pi_{sat}(D) \cup\\ &\{\text{:-} arg(S), ass(S,1), inv(S).\ \ \text{:-} arg(S), ass(S,0), sat(S).\}\end{aligned}$$

Theorem 1. *For every ADF D it holds that $adm(D) \cong \mathcal{AS}(\pi_{adm}(D))$.*

Example 3. Considering the ADF D from Example 1 (recall $\varphi_a{=}a \vee \neg b, \varphi_b{=}\neg a$), $\pi_{adm}(D)$ looks as follows:

```
arg(a). arg(b).  lt(u,0). lt(u,1). lt(1,1). lt(0,0).
ass(S,0) :- not ass(S,1), not ass(S,u), arg(S).
ass(S,1) :- not ass(S,0), not ass(S,u), arg(S).
ass(S,u) :- not ass(S,0), not ass(S,1), arg(S).
sat(a) :- ass(a,Ya), lt(Ya,Va), ass(b,Yb), lt(Yb,Vb),
          Vnb=1-Vb, Vab=Va?Vnb, Vab=1.
inv(a) :- ass(a,Ya), lt(Ya,Va), ass(b,Yb), lt(Yb,Vb),
          Vnb=1-Vb, Vab=Va?Vnb, Vab=0.
sat(b) :- ass(a,Ya), lt(Ya,Va), Vna=1-Va, Vna=1.
inv(b) :- ass(a,Ya), lt(Ya,Va), Vna=1-Va, Vna=0.
:- arg(S), ass(S,1), inv(S).
:- arg(S), ass(S,0), sat(S).
```

A possible output of an ASP solver given this instance looks as follows (not showing `arg` and `lt` predicates):

```
ass(a,0) ass(b,1) inv(a) sat(b)
ass(a,1) ass(b,0) sat(a) inv(b)
ass(a,1) ass(b,u) sat(a) inv(b)
ass(a,u) ass(b,u) sat(a) inv(a) sat(b) inv(b)
```

For the encoding of the complete semantics we only need to add two constraints to the encoding of the admissible semantics. These express that, in addition to satisfying Observation 1, in order to be complete, an interpretation v of D also needs to fulfill the following condition for every $s \in S$:

- if $v(s) = \mathbf{u}$ then there are $w_1, w_2 \in [v]_2$ s.t. $w_1(\varphi_s) = 0$ and $w_2(\varphi_s) = 1$

Hence, the encoding for the complete semantics just requires two additional constraints.

Theorem 2. *Let $\pi_{com}(D) = \{$:-$arg(S), ass(S, \mathbf{u}), not\ inv(S)$. :-$arg(S), ass(S, \mathbf{u}), not\ sat(S).\} \cup \pi_{adm}(D)$. For every ADF D it holds that $com(D) \cong \mathcal{AS}(\pi_{com}(D))$.*

Saturation Encodings for Preferred Semantics

For the encoding of the preferred semantics we make use of the saturation technique (Eiter and Gottlob 1995), see (Charwat et al. 2015) for its use in computing the preferred extensions of Dung AFs. The saturation technique allows checking that a property holds for a *set* of guesses within a disjunctive ASP program, by generating a unique "saturated" guess that "verifies" the property for any such guess. In the encoding of the preferred semantics for an ADF D we extend $\pi_{adm}(D)$ by making use of this technique to verify that all interpretations of D that are greater w.r.t. $\leq_i$ than the interpretation determined by the assignments guessed in the program fragment π_{guess} are either identical to the interpretation in question or not admissible. Hence, the relevant interpretation must be preferred according to the definition.

The module π_{guess2} amounts to "making a second guess" ($ass2(\cdot)$) extending the "first guess" ($ass(\cdot)$) from π_{guess}.

$$\begin{aligned}\pi_{\text{guess2}} := \{&ass2(S,0) \text{:-} ass(S,0).\ \ ass2(S,1) \text{:-} ass(S,1).\\ &ass2(S,1) \mid ass2(S,0) \mid ass2(S,\mathbf{u}) \text{:-} ass(S,\mathbf{u}).\}\end{aligned}$$

The fragment $\pi_{sat2}(D)$ will allow us to check whether the second guess obtained from π_{guess2} is admissible:

$$\begin{aligned}\pi_{sat2}(D) := \{&sat2(s) \text{:-} E2_s, V_{\varphi_s} = 1.\\ &inv2(s) \text{:-} E2_s, V_{\varphi_s} = 0. \mid s \in S\} \text{ with}\\ E2_s := \{&ass2(t, Y_t), lt(Y_t, V_t) \mid t \in par(s)\} \cup B(\varphi_s)\end{aligned}$$

The only difference between the fragment $\pi_{sat2}(D)$ and $\pi_{sat}(D)$ is that we now evaluate acceptance conditions w.r.t. the second guess given via $ass2(\cdot)$.

The following program fragment guarantees that the atom *saturate* is derived whenever the second guess (computed via π_{guess2}) is either identical (first rule of $\pi_{check}(D)$) to the first guess (computed via the predicate $\pi_{\text{guess}}(D)$) or is not admissible (last two rules of $\pi_{check}(D)$). We will say that in this case the second guess is *not* a counter-example to the first guess corresponding to a preferred interpretation of D.

$$\begin{aligned}\pi_{check}(D) := \{&saturate \text{:-} ass(s_1, X_1), ass2(s_1, X_1), \ldots\\ &ass(s_k, X_k), ass2(s_k, X_k).\\ &saturate \text{:-} arg(S), ass2(S,1), inv2(S).\\ &saturate \text{:-} arg(S), ass2(S,0), sat2(S).\}\end{aligned}$$

The module $\pi_{saturate}$ now assures that whenever the atom *saturate* is derived, also *ass2*$(S,0)$, *ass2*$(S,1)$, *ass2*$(S,\mathbf{u})$, *sat2*(S), and *inv2*(S) are derived for every $s \in S$.

$$\begin{aligned}\pi_{saturate} := \{&ass2(S,0) : \text{-}arg(S), ass(S,\mathbf{u}), saturate.\\ &ass2(S,1) : \text{-}arg(S), ass(S,\mathbf{u}), saturate.\\ &ass2(S,\mathbf{u}) : \text{-}arg(S), ass(S,\mathbf{u}), saturate.\\ &sat2(S) : \text{-}arg(S), saturate.\ inv2(S) : \text{-}arg(S), saturate.\}\end{aligned}$$

The effect of this fragment is that whenever all the "second guesses" (computed via π_{guess2}) are *not* counter-examples to the first guess (computed via π_{guess}) corresponding to a preferred interpretation of D, then all the answer sets will be saturated on the predicates *ass2*, *sat2*, and *inv2*, i.e. all ground instances of these predicates will be included in any answer set. Finally, all that needs to be added to the program fragments $\pi_0(D)$, $\pi_{guess}(D)$, $\pi_{sat}(D)$, $\pi_{guess2}(D)$, $\pi_{check}(D)$, $\pi_{sat2}(D)$ for the encoding of the preferred semantics, $\pi_{prf}(D)$, is a constraint disallowing those assignments which have a counter-example to them corresponding to a preferred interpretation of D, i.e. those for which the atom *saturate* is *not* derived. We obtain the following result.

Theorem 3. *For every ADF D it holds that prf$(D) \cong \mathcal{AS}(\pi_{prf}(D))$, where $\pi_{prf}(D) = \pi_{adm}(D) \cup \pi_{guess2} \cup \pi_{sat2}(D) \cup \pi_{check}(D) \cup \pi_{saturate} \cup \{$: -not saturate.$\}$.*

Observe that the encodings are adequate from a complexity viewpoint (cf. Table 1), since admissible and complete semantics are encoded by normal programs, while the encodings of preferred semantics yield disjunctive programs.

Grappa Encodings

We turn to the encodings for GRAPPA instances. We again make use of the correspondence $\cong$ between 3-valued interpretations (now for GRAPPA instances) and sets of ground atoms (interpretations of ASP programs) defined via ASP atoms $ass(s,x)$ for statements s and $x \in \{1,0,\mathbf{u}\}$. Hence, we now strive for encodings π_σ for $\sigma \in \{adm, com, prf\}$ s.t. for every GRAPPA instance G we get $\sigma(G) \cong \mathcal{AS}(\pi_\sigma(G))$ (see Definition 1 for the formal meaning of the latter overloaded use of $\cong$).

In the same manner as the semantics of GRAPPA mirror the semantics of ADFs, the ASP encodings for GRAPPA instances presented in this section closely resemble the encodings for ADF instances given in the previous section. In fact, we will reuse most of the ASP fragments. Formally this amounts to extending the corresponding encoding functions to also admit GRAPPA instances as arguments. The main difference is the way in which we encode the evaluation of the acceptance patterns, this also clearly being the crucial difference between GRAPPA and ADF instances.

Throughout this section, let $G = (S, E, L, \lambda, \pi)$ be a GRAPPA instance with $S = \{s_1, \ldots, s_k\}$. We redefine the set of atoms $B(\phi)$ corresponding to the subformulas of acceptance conditions of ADF statements to encode acceptance patterns of GRAPPA instances. The recursive function representing these patterns needs statement s as additional parameter for the encoding of the basic patterns and is defined as $B_s(\phi) :=$

$$\begin{array}{ll}B_s(\phi_1) \cup B_s(\phi_2) \cup \{V_\phi = V_{\phi_1} \& V_{\phi_2}\} & \text{if } \phi = \phi_1 \wedge \phi_2\\ B_s(\phi_1) \cup B_s(\phi_2) \cup \{V_\phi = V_{\phi_1} ? V_{\phi_2}\} & \text{if } \phi = \phi_1 \vee \phi_2\\ B_s(\psi) \cup \{V_\phi = 1 - V_\psi\} & \text{if } \phi = \neg\psi\\ P_s(\tau) \cup \{V_\phi = \#\mathrm{sum}\{1 : V_\tau \bar{R} a\}\ \} & \text{if } \phi = \tau R a\end{array}$$

The difference between B_s and B (note the missing subscript s) as defined in the previous section is in the last line where $P_s(\tau) \cup \{V_\phi = \#\mathrm{sum}\{1 : V_\tau \bar{R} a\}\}$ encodes the evaluation of a basic pattern $\phi = \tau R a$. We make use of the ASP aggregate $\#\mathrm{sum}$ as well as the simple function $\bar{R} := <=$ (resp. $>=$, $!=$) if $R = \leq$ (resp. $\geq, \neq$) and $\bar{R} = R$ otherwise, relating GRAPPA and ASP syntax for relational operators.

The function $P_s(\tau)$ on the other hand gives us a set of atoms corresponding to the evaluation of a sum τ of terms.

$$P_s(\tau) := \begin{cases} P_s(\chi) \cup T_s(t) \cup \{V_\tau = a * V_t + V_\chi\} & \text{if } \tau = at + \chi \\ T_s(t) \cup \{V_\tau = a * V_t\} & \text{if } \tau = at \end{cases}$$

The definition of P_s in turn makes use of the function $T_s(t)$ that returns an atom representing a term t. Here let $s \in S$ be fixed and $par(s) = \{r_1, \ldots, r_q\}$, $l_r = \lambda(r,s)$ for $r \in par(s)$, and $par(s,l) = \{r \in par(s) \mid l_r = l\}$. In order to define atoms corresponding to the evaluation of terms depending on the active labels (those without subscript t) we use the ASP aggregates $\#\mathrm{sum}$, $\#\mathrm{min}$, $\#\mathrm{max}$, and $\#\mathrm{count}$, as well as variables Z_r corresponding to completions of the guessed assignment of statements $r \in S$. Atoms corresponding to terms whose evaluation is independent on the active labels, on the other hand, can be constructed based on the instance G only. We define $T_s(t)$ as

$$\begin{array}{ll}\{ V_t = \#\mathrm{sum}\{Z_{r_{i_1}}, r_{i_1} ; \ldots ; Z_{r_{i_w}}, r_{i_w}\}\ \} & \\ \text{with } \{r_{i_1}, \ldots, r_{i_w}\} = par(s,l) & \text{if } t = \#l \text{ and } par(s,l) \neq \emptyset\\ \{ V_t = N \} \text{ with } N = |par(s,l)| & \text{if } t = \#_t l\\ \{ V_t = \#\mathrm{min}\{l_{r_1} : Z_{r_1} = 1 ; \ldots ; l_{r_q} : Z_{r_q} = 1\}\ \} & \text{if } t = \mathrm{min}\\ \{ V_t = N \} \text{ with } N = min\{l_{r_1}, \ldots, l_{r_q}\} & \text{if } t = \mathrm{min_t}\\ \{ V_t = \#\mathrm{max}\{l_{r_1} : Z_{r_1} = 1 ; \ldots ; l_{r_q} : Z_{r_q} = 1\}\ \} & \text{if } t = \mathrm{max}\\ \{ V_t = N \} \text{ with } N = max\{l_{r_1}, \ldots, l_{r_q}\} & \text{if } t = \mathrm{max_t}\\ \{ V_t = \#\mathrm{sum}\{l_{r_1}, r_1 : Z_{r_1} = 1 ; \ldots ; l_{r_q}, r_q : Z_{r_q} = 1\}\ \} & \text{if } t = \mathrm{sum} \text{ and } par(s) \neq \emptyset\\ \{ V_t = N \} \text{ with } N = l_{r_1} + \ldots + l_{r_q} & \text{if } t = \mathrm{sum_t}\\ \{ V_t = \#\mathrm{count}\{l_{r_1} : Z_{r_1} = 1 ; \ldots ; l_{r_q} : Z_{r_q} = 1\}\ \} & \text{if } t = \mathrm{count} \text{ and } par(s) \neq \emptyset\\ \{ V_t = N \} \text{ with } N = |\{l_r \mid r \in par(s)\}| & \text{if } t = \mathrm{count_t}\\ \{ V_t = 0\} & \text{if } t = \#l \text{ and } par(s,l) = \emptyset\\ & \text{or } t = \mathrm{sum}, t = \mathrm{count} \text{ and } par(s) = \emptyset\end{array}$$

As pointed out earlier, the encodings for GRAPPA instances for $\sigma \in \{adm, com, prf\}$ differ from the corresponding ADF encodings only in the fragments handling

the evaluation of acceptance patterns (under the completions of an interpretation). Hence, the encodings $\pi_\sigma(G)$ for the GRAPPA instance G boil down to the programs

$$\begin{aligned}\pi_{adm}(G) :=& \pi_0(G) \cup \pi_{guess} \cup \pi'_{sat}(G) \cup \\ & \{:\!-arg(S), ass(S,1), inv(S).\ :\!-arg(S), ass(S,0), sat(S).\} \\ \pi_{com}(G) :=& \pi_{adm}(G) \cup \{:\!-arg(S), ass(S,\mathbf{u}), not\ inv(S). \\ & :\!-arg(S), ass(S,\mathbf{u}), not\ sat(S).\} \\ \pi_{prf}(G) :=& \pi_{adm}(G) \cup \pi_{guess2} \cup \pi'_{sat2}(G) \cup \pi_{check}(G) \cup \pi_{saturate} \cup \\ & \{:\!-not\ saturate.\}\end{aligned}$$

Here, the difference to the encoding for ADF semantics is the use of the program fragments

$$\begin{aligned}\pi'_{sat}(G) :=& \{sat(s) :\!- E_s, V_{\pi(s)} = 1. \\ & inv(s) :\!- E_s, V_{\pi(s)} = 0. \mid s \in S\} \\ \pi'_{sat2}(G) :=& \{sat2(s) :\!- E2_s, V_{\pi(s)} = 1. \\ & inv2(s) :\!- E2_s, V_{\pi(s)} = 0. \mid s \in S\}\end{aligned}$$

where we make use of the shortcuts E_s and $E2_s$. In their definitions we in turn use the function $B_s(\phi)$ returning the atoms corresponding to a GRAPPA acceptance function:

$$\begin{aligned}E_s :=& \{ass(r, Y_r), lt(Y_r, Z_r) \mid r \in par(s)\} \cup B_s(\pi(s)) \\ E2_s :=& \{ass2(r, Y_r), lt(Y_r, Z_r) \mid r \in par(s)\} \cup B_s(\pi(s))\end{aligned}$$

Theorem 4. *For $\sigma \in \{adm, com, prf\}$ it holds for every GRAPPA instance G that $\sigma(G) \cong \mathcal{AS}(\pi_\sigma(G))$.*

Experiments

We implemented all encodings for ADFs and GRAPPA presented in this work. To make use of the encodings for reasoning, these need to be fed to an ASP solver such as `clingo` (Gebser et al. 2011). We carried out experiments to compare the performance of our approach with existing systems for ADFs. Specifically, we compared the performance of our prototype system, `YADF` ("Y" stands for "dynamic"), with that of `DIAMOND` and the QBF based system `QADF`. We focused on credulous and skeptical reasoning for admissible and preferred semantics, respectively.

To generate ADFs, we first used a "grid-based" ADF generator which has been employed in previous evaluations (Diller, Wallner, and Woltran 2014). Here statements have as parents a subset of 8 possible neighbors of a randomly generated grid of width 7. Acceptance conditions are generated by connecting parents via $\wedge$ or $\vee$. Probabilities determine the choice of these connectives and whether parents appear negated or are replaced by truth constants.

We also wrote our own graph-based generator which takes a directed graph as input and generates an ADF inheriting the structure of the graph. Each parent of a statement is assigned to one of 5 different groups (with equal probability

	Cred-*adm*			Skept-*prf*		
	`DIAMOND`	`QADF`	`YADF`	`DIAMOND`	`QADF`	`YADF`
Gri-10	0.11 (0)	0.62 (0)	0.66 (0)	0.31 (0)	0.9 (0)	0.75 (0)
Gri-20	0.35 (0)	0.8 (0)	0.96 (0)	51.17 (20)	41.53 (0)	1.26 (0)
Gri-30	0.9 (0)	1.01 (0)	1.13 (0)	51.48 (38)	497.4 (39)	1.76 (0)
Gri-40	1.64 (0)	1.21 (0)	1.34 (0)	- (40)	- (40)	2.68 (0)
Gri-50	2.8 (0)	1.47 (0)	1.52 (0)	- (40)	- (40)	4.83 (0)
Gri-60	4.3 (0)	2.08 (0)	1.86 (0)	- (40)	- (40)	9.6 (0)
Gri-70	6.52 (0)	3.52 (0)	2.08 (0)	- (40)	- (40)	68.48 (1)
Gri-80	8.83 (0)	3.08 (1)	2.37 (0)	- (40)	- (40)	84.37 (6)
Metro	5.7 (0)	5.86 (7)	1.6 (0)	- (40)	- (40)	43.01 (11)

Table 2: Mean running times in seconds for credulous reasoning under *adm* and skeptical reasoning under *prf* on ADF instances generated by the grid-based (Gri-X = ADFs with X statements) and graph-based generator (5 ADFs per city). Number of time-outs (out of 40 instances; with time-out of 600 seconds) in parentheses. Mean running times are computed disregarding time-outs.

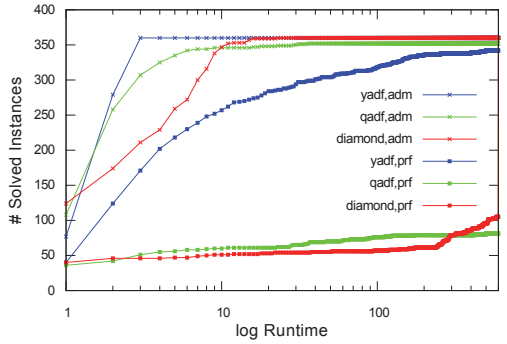

Figure 1: Number of instances solved in running time less than x seconds for credulous reasoning under admissible and skeptical reasoning under preferred semantics. All grid-based and graph-based instances were considered (360 total).

in our experiments), determining whether the parent participates in a subformula of the statement's acceptance condition representing the notions of attack, group-attack, support, or group-support familiar from argumentation. Also, the parents can appear as literals connected by XOR (in order to capture the full complexity of ADFs). The subformulas are connected via $\wedge$ or $\vee$ with equal probability. In our experiments the input graphs represent public transport networks of 8 different cities.

Experiments were carried out on a 48 GB `Debian` (8.5) machine with 8 `Intel Xeon` processors (2.33 GHz). Due to known problems with the latest available versions[1] of `DIAMOND`, 2.0.2 and 2.0.0, we report on our results with version 0.9 (modified to support credulous and skeptical reasoning). For `QADF` we used version 0.3.2 with `bloqqer` 035 (Biere, Lonsing, and Seidl 2011) and `DepQBF` 4.0 (Lonsing and Biere 2010). `YADF` is version 0.1.0 with the rule decomposition tool `lpopt` (Bichler, Morak, and Woltran 2016) and `clingo` 4.4.0.

As can be seen from Table 2 and Figure 1, our system performed comparably to `DIAMOND` and `QADF` for credulous reasoning under *adm*, somewhat better on the public transport (or "metro") based instances. There is a clear advantage

[1]The version of `DIAMOND` reported in (Ellmauthaler and Strass 2016) was not yet available at time of submission.

in the use of our encodings over DIAMOND and QADF for skeptical reasoning under *prf*, although there are 7 time-outs on the grid based instances with 70 and 80 statements as well as on 11 of the metro-based instances. Experiments on very dense randomly generated graphs (not in the table) showed slightly better performance of DIAMOND compared to YADF for credulous reasoning.

Discussion

In this work, we have developed novel ASP encodings for advanced reasoning problems in argumentation that reach up to the third level of the polynomial hierarchy. Compared to previous work, we rely on translations that make a single call to an ASP-solver sufficient. The key idea is to reduce one dimension of complexity to "long" rule bodies, exploiting the fact that checking whether such a rule fires is already NP-complete (as witnessed by the respective complexity of conjunctive queries (Chandra and Merlin 1977)).

We implemented our approach for ADF and GRAPPA[2]. Our experiments show the potential and improved performance of our encodings. Still, the number of statements we can handle is somewhat limited. Our encodings thus might also be interesting benchmarks for ASP competitions.

Acknowledgments. This research has been supported by DFG (project BR 1817/7-2) and FWF (projects I2854, S11409-N23, and W1255-N23). The authors also thank Jörg Pührer for his helpful observations regarding the encodings.

References

Al-Abdulkarim, L.; Atkinson, K.; and Bench-Capon, T. 2016. A methodology for designing systems to reason with legal cases using abstract dialectical frameworks. *Artif. Intell. Law* 24(1):1–49.

Amgoud, L., and Prade, H. 2009. Using Arguments for Making and Explaining Decisions. *Artif. Intell.* 173(3-4):413–436.

Bench-Capon, T. J. M., and Dunne, P. E. 2005. Argumentation in AI and Law: Editors' Introduction. *Artif. Intell. Law* 13(1):1–8.

Berthold, M. 2016. Extending the DIAMOND system to work with GRAPPA. In *Proc. SAFA*, 52–62.

Bichler, M.; Morak, M.; and Woltran, S. 2016. lpopt: A rule optimization tool for answer set programming. *CoRR* abs/1608.05675.

Biere, A.; Lonsing, F.; and Seidl, M. 2011. Blocked Clause Elimination for QBF. In *Proc. CADE*, 101–115. Springer.

Booth, R. 2015. Judgment aggregation in abstract dialectical frameworks. In *Advances in KR, Logic Programming, and Abstract Argumentation*, 296–308. Springer.

Brewka, G., and Woltran, S. 2010. Abstract Dialectical Frameworks. In *Proc. KR*, 102–111. AAAI Press.

Brewka, G., and Woltran, S. 2014. GRAPPA: A Semantical Framework for Graph-Based Argument Processing. In *Proc. ECAI*, 153–158. IOS Press.

Brewka, G.; Strass, H.; Ellmauthaler, S.; Wallner, J. P.; and Woltran, S. 2013. Abstract Dialectical Frameworks Revisited. In *Proc. IJCAI*, 803–809. IJCAI/AAAI.

[2]The system for ADF reasoning is available at https://www.dbai.tuwien.ac.at/proj/adf/yadf/. The encodings for GRAPPA are available as part of a larger system: https://www.dbai.tuwien.ac.at/proj/adf/grappavis/.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Com. ACM* 54(12):92–103.

Brewka, G.; Polberg, S.; and Woltran, S. 2014. Generalizations of Dung Frameworks and Their Role in Formal Argumentation. *IEEE Intelligent Systems* 29(1):30–38.

Cartwright, D., and Atkinson, K. 2009. Using Computational Argumentation to Support E-participation. *IEEE Intelligent Systems* 24(5):42–52.

Chandra, A. K., and Merlin, P. M. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC*, 77–90. ACM.

Charwat, G.; Dvořák, W.; Gaggl, S. A.; Wallner, J. P.; and Woltran, S. 2015. Methods for solving reasoning problems in abstract argumentation - A survey. *Artif. Intell.* 220:28–63.

Diller, M.; Wallner, J. P.; and Woltran, S. 2014. Reasoning in Abstract Dialectical Frameworks Using Quantified Boolean Formulas. In *Proc. COMMA*, 241–252. IOS Press.

Diller, M.; Wallner, J. P.; and Woltran, S. 2015. Reasoning in Abstract Dialectical Frameworks Using Quantified Boolean Formulas. *Argument & Computation* 6(2):149–177.

Dung, P. M. 1995. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artif. Intell.* 77(2):321–358.

Eiter, T., and Gottlob, G. 1995. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.* 15(3-4):289–323.

Eiter, T.; Faber, W.; Fink, M.; and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.* 51(2-4):123–165.

Eiter, T.; Gottlob, G.; and Mannila, H. 1997. Disjunctive datalog. *ACM Trans. Database Syst.* 22(3):364–418.

Ellmauthaler, S., and Strass, H. 2014. The DIAMOND System for Computing with Abstract Dialectical Frameworks. In *Proc. COMMA*, 233–240. IOS Press.

Ellmauthaler, S., and Strass, H. 2016. DIAMOND 3.0 – A native C++ implementation of DIAMOND. In *Proc. COMMA*, 471–472. IOS Press.

Gaggl, S. A., and Strass, H. 2014. Decomposing abstract dialectical frameworks. In *Proc. COMMA*, 281–292. IOS Press.

Gebser, M.; Kaufmann, B.; Kaminski, R.; Ostrowski, M.; Schaub, T.; and Schneider, M. T. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.* 24(2):107–124.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Lindauer, M.; Ostrowski, M.; Romero, J.; Schaub, T.; and Thiele, S. 2015. Potassco User Guide. https://sourceforge.net/projects/potassco/files/guide/, accessed: 2016-11-22.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput.* 9(3/4):365–386.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7(3):499–562.

Lonsing, F., and Biere, A. 2010. DepQBF: A Dependency-Aware QBF Solver. *JSAT* 7(2-3):71–76.

McBurney, P.; Parsons, S.; and Rahwan, I., eds. 2012. *Proc. ArgMAS*, volume 7543 of *LNCS*. Springer.

Strass, H., and Wallner, J. P. 2015. Analyzing the computational complexity of abstract dialectical frameworks via approximation fixpoint theory. *Artif. Intell.* 226:34–74.