TPLP **20** (3): 391–431, 2020. © The Author(s), 2020. Published by Cambridge University Press.

doi:10.1017/S1471068419000474 First published online 15 January 2020

Solving Advanced Argumentation Problems with Answer Set Programming[*]

GERHARD BREWKA
Universität Leipzig, Leipzig, Germany

MARTIN DILLER
TU Dresden, Dresden, Germany

GEORG HEISSENBERGER, THOMAS LINSBICHLER and STEFAN WOLTRAN
TU Wien, Vienna, Austria

submitted 8 December 2018; revised 11 December 2019; accepted 18 December 2019

Abstract

Powerful formalisms for abstract argumentation have been proposed, among them abstract dialectical frameworks (ADFs) that allow for a succinct and flexible specification of the relationship between arguments and the GRAPPA framework which allows argumentation scenarios to be represented as arbitrary edge-labeled graphs. The complexity of ADFs and GRAPPA is 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 ADFs and GRAPPA to ASP. More specifically, we provide reductions for the five main ADF semantics of admissible, complete, preferred, grounded, and stable interpretations, and exemplify how these reductions need to be adapted for GRAPPA for the admissible, complete, and preferred semantics.

KEYWORDS: ADFs, GRAPPA, encodings, ASP

1 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 *et al.* 2012). Dung's argumentation frameworks (AFs) (Dung 1995) 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 *et al.* (2014) for an overview.

[*] This research has been supported by DFG (projects BR 1817/7-2 as well as 389792660 - TRR 248) and FWF (projects I2854, Y698, P32830, S11409-N23, and W1255-N23). The authors also thank Jörg Pührer for his helpful observations regarding the encodings. Also thanks to Wolfgang Dvořák and Ringo Baumann for guiding the paper through the editing process.

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; see also the recent handbook article Brewka *et al*. (2018) which surveys both formalisms. 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 for example, Gaggl and Strass (2014); Booth (2015) including also practical applications in fields such as legal reasoning (Al-Abdulkarim *et al*. 2016; Atkinson and Bench-Capon 2018), text exploration (Cabrio and Villata 2016), or discourse analysis (Neugebauer 2018).

Two approaches to implement ADF reasoning have been proposed in the literature. `QADF` (Diller *et al*. 2014, 2015) encodes problems as quantified Boolean formulas (QBFs) such that a single call of a QBF solver delivers the result. The `DIAMOND` family of systems (Ellmauthaler and Strass 2014, 2016; Strass and Ellmauthaler 2017), on the other hand, employ Answer Set Programming (ASP). Since the `DIAMOND` systems rely on *static* encodings, that is, 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 *et al*. 1997)). Therefore, the preferred semantics in particular (which comprises the hardest problems for ADFs and GRAPPA) needs 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 employing a translation to ADFs as an 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 (and avoiding any exponential blowup in data or program size). The reason for choosing ASP is that the rich syntax of GRAPPA is captured much more easily 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. This allows to generate rules of arbitrary length that can take care of NP-hard subtasks themselves. This particular method has been advocated in Bichler *et al*. (2016b) in combination with tools that decompose such long rules whenever possible in order to avoid huge groundings (Bichler *et al*. 2016a). To the best of our knowledge, our work is the first to apply this technique in the field of argumentation.

More specifically, we provide encodings for the admissible, complete, preferred, grounded, and stable semantics for ADFs and discuss how such encodings can be adapted to 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 further provide some details about the resulting system `YADF` ("Y" standing for dYnamic) which is publicly available at `https://www.dbai.tuwien.ac.at/proj/adf/yadf/`. Finally, we give an overview of recent empirical evaluations, including our own, comparing the performance of `YADF` with the other main existing ADF systems.

This paper is an extended version of Brewka *et al.* (2017), which did not contain the encodings for the grounded and stable semantics. In addition, we provide some prototypical proofs for the correctness of the encodings. We also update the discussion on empirical evaluations. The paper is based on (Section 3.2 of) the second author's thesis (Diller 2019).

2 Background

ADFs. An ADF is a directed graph whose nodes represent statements. The links represent dependencies: the acceptance status of a node s only depends on the acceptance status of its parents (denoted $par(s)$; often also with a subscript as in $par_D(s)$ to make the reference to the ADF D explicit), 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 (three-valued) interpretations, that is, 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 two-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, that is, two-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 two-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 three-valued interpretations, while we evaluate acceptance conditions under their two-valued completions.

Given an ADF $D = (S, \{\varphi_s\}_{s \in S})$, 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. $\leq_i$. An interpretation v is the (unique) *grounded* interpretation w.r.t. D if v is complete and there is no other complete interpretation w for which $w <_i v$.

Turning to semantics returning two-valued interpretations, a two-valued interpretation v is a *model* of D if $v(s) = v(\varphi_s)$ for every $s \in S$. The definition of the stable semantics for

Table 1. *All admissible interpretations of the ADF from Figure 1. The rightmost column shows further semantics the interpretations belong to*

	a	b	c	
v_1	$\mathbf{u}$	$\mathbf{u}$	$\mathbf{u}$	*adm*
v_2	$\mathbf{u}$	0	$\mathbf{u}$	*adm*
v_3	$\mathbf{u}$	1	$\mathbf{u}$	*adm*
v_4	1	1	$\mathbf{u}$	*adm*
v_5	$\mathbf{u}$	1	1	*adm*
v_6	1	$\mathbf{u}$	$\mathbf{u}$	*adm, com, grd*
v_7	1	0	$\mathbf{u}$	*adm, com, prf*
v_8	1	1	1	*adm, com, prf, mod*

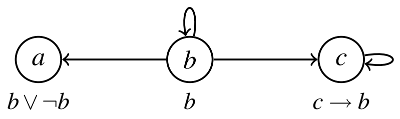

Fig. 1. ADF example.

ADFs is inspired by the stable semantics for logic programs, its purpose being to disallow cyclic support within a model. First of all, in order to be a *stable* model of D, v needs to be a model of D. Secondly, $E_v = \{s \in S \mid v(s) = 1\}$ must equal the statements set to true in the grounded interpretation of the reduced ADF $D^v = (E_v, \{\varphi_s^v\}_{s\in E_v})$, where for $s \in E_v$ we set $\varphi_s^v := \varphi_s[b/\bot : v(b) = 0]$. If $v \downharpoonleft_{E_v}$ is the interpretation v projected on E_v, that is, $v \downharpoonleft_{E_v} (s) = v(s)$ for $s \in E_v$ and undefined otherwise, then the latter amounts to the fact that $v \downharpoonleft_{E_v}$ be the grounded interpretation of D^v.

As shown in Brewka *et al.* (2013), these semantics generalize the corresponding notions defined for AFs. For $\sigma \in \{adm, com, prf, grd, mod, stb\}$, $\sigma(D)$ denotes the set of all admissible (resp. complete, preferred, grounded, model, stable) interpretations w.r.t. D.

Example 1
In Figure 1, we see an example ADF $D = (\{a, b, c\}, C)$ with the acceptance conditions C given by $\varphi_a = b \vee \neg b$, $\varphi_b = b$, and $\varphi_c = c \rightarrow b$. The acceptance conditions are shown below the statements in the figure.

The admissible interpretations of D are shown in Table 1. Moreover, the rightmost column shows further semantics the interpretations belong to. For instance, the interpretation v_8 mapping each statement to true is admissible, complete, and preferred in D and a model of D. The only model of D is v_8, with the reduct of this model being $D^{v_8} = D$. The grounded interpretation of D is v_6, which is different from v_8. Therefore, v_8 is not a stable model. In fact, D does not have a stable model. ◇

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 labeled 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-labeled 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 three-valued interpretations. The major difference is that acceptance conditions are no longer specified in terms of 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 of the form $(L \to \mathbb{N}) \to \{1, 0\}$, where L is the set of all labels.

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 $a_1t_1 + \cdots + a_nt_n \, R \, a$, 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 $G = (S, E, L, \lambda, \alpha)$ 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 α 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:

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

$min_{(t)}$, $max_{(t)}$, $sum_{(t)}$ are undefined in case of non-numerical labels. For $\emptyset$, they yield the neutral element of the corresponding operation, that is, $val_s^m(sum) = val_s^m(sum_t) = 0$, $val_s^m(min) = val_s^m(min_t) = \infty$, and $val_s^m(max) = val_s^m(max_t) = -\infty$. Let m and s be as before. For a basic acceptance pattern $\alpha = a_1t_1 + \cdots + a_nt_nR$ we define $\alpha(m, s) = 1$ if $\sum_{i=1}^{n} \big(a_i \; val_s^m(t_i)\big) \; R \; a$, while $\alpha(m, s) = 0$ otherwise. The extension to the evaluation of Boolean combinations is as usual.

The characteristic function Γ_G for a GRAPPA instance G, as is the case for the characteristic function for ADFs, takes a three-valued interpretation v and produces a new one v'. Again v' is constructed by considering all two-valued completions w 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 w. The acceptance function then takes this multiset as argument and produces the truth value induced by w.

Let v be a two-valued interpretation. The multiset of active labels of $s \in S$ in G under v, m_s^v, is defined as

$$m_s^v(l) = |\{(p,s) \in E \mid v(p) = 1, \lambda((p,s)) = l\}|,$$

for each $l \in L$. Then the characteristic function $\Gamma_G(v) = v'$ for a GRAPPA instance G is given by

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

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.

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

ASP. In ASP (Leone *et al.* 2006; Brewka *et al.* 2011), problems are described using logic programs, which are sets of rules of the form

$$a_1 \vee \ldots \vee a_n \text{ :- } b_1, \ldots, b_k,\ \textit{not } b_{k+1}, \ldots,\ \textit{not } b_m.$$

Here each a_i $(1 \leq i \leq n)$ and b_j $(1 \leq j \leq m)$ is a ground atom. The symbol *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 $n = 0$. $B(r)$ denotes the body of a rule and $H(r)$ the head. A *program* is a finite set of disjunctive rules. If each rule in a program is normal we call the program normal, otherwise the program is disjunctive.

Each logic program π induces a collection of so-called *answer sets*, denoted as $\mathscr{AS}(\pi)$, which are distinguished models of the program determined by the answer set semantics. The answer sets of a program π are the subset minimal models satisfying the Gelfond–Lifschitz reduct π^I of π; 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. We denote by $Gr(\pi)$ the ground instance of a program π. 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).

Table 2. *Complexity results for ADFs, GRAPPA, and ASP*

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

Complexity. The complexity results that are central for our work are given in Table 2. 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 *et al.* (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 to non-ground normal programs of bounded arity for credulous reasoning w.r.t. the admissible, complete, preferred, and stable semantics; skeptical reasoning for the stable semantics can be reduced to skeptical reasoning for normal programs. Skeptical preferred reasoning needs to be treated with disjunctive programs. We provide such reductions in what follows.

3 ADF encodings

We construct ASP encodings π_σ for the semantics $\sigma \in \{adm, com, prf, grd, stb\}$ such that there is a correspondence between the σ interpretations of an ADF $D = (S, C)$ and the answer sets of $\pi_\sigma(D)$ (the encoding function π_σ applied to D). More precisely, we will use atoms $asg(s, x)$ 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 $s \in S$, $v(s) = x$ iff $asg(s, x) \in I$. 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 (ADF) interpretations V and a collection of sets of ground atoms (ASP interpretations) $\mathscr{I}$, we say that V and $\mathscr{I}$ correspond, $V \cong \mathscr{I}$, if

1. for every $v \in V$, there is an $I \in \mathscr{I}$ s.t. $v \cong I$;
2. for every $I \in \mathscr{I}$, there is a $v \in V$ s.t. $v \cong I$.

Having encodings π_σ for $\sigma \in \{adm, com, prf, grd, stb\}$ for which $\sigma(D) \cong \mathscr{AS}(\pi_\sigma(D))$ for any ADF D allows to enumerate the σ interpretations of an ADF D by reading the

ADF interpretations that correspond (via $\cong$) to each $I \in \mathscr{AS}(\pi_\sigma(D))$ off the predicates $asg(s,x) \in I$ ($s \in S$, $x \in \{1, 0, \mathbf{u}\}$). Results for credulous and skeptical reasoning for each of the semantics are obtained via the homonymous ASP reasoning tasks applied on our encodings.

3.1 Encoding for the admissible semantics

In the course of presenting our dynamic ASP encodings for the admissible semantics, we introduce several elements we will make use of throughout Section 3. Among these is that all encodings will assume a simple set of facts indicating the statements of the input ADF $D = (S, \{\varphi_S\}_{s\in S})$:

$$\pi_{arg}(D) := \{arg(s). \mid s \in S\}.$$

Also, several of the encodings will need facts 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:

$$\pi_{lt} := \{lt(\mathbf{u}, 0).\ lt(\mathbf{u}, 1).\ lt(1, 1).\ lt(0, 0).\}.$$

Here, for instance, the atoms $lt(\mathbf{u}, 0)$ and $lt(\mathbf{u}, 1)$ together express that if an ADF interpretation maps a statement to the truth value $\mathbf{u}$ then a completion (of the interpretation in question) can map the same statement to the truth values 0 or 1. Note, in particular, that $lt(\mathbf{u}, \mathbf{u}) \notin \pi_{lt}$ since completions can map a statement only to the truth value 0 or 1.

All of our encodings, including the one for the admissible semantics, follow the guess & check methodology that is at the heart of the ASP paradigm (Janhunen and Niemelä 2016). Here, parts of a program delineate candidates for a solution to a problem. These are often referred to as "guesses". Other parts of the program, the "constraints", then check whether the guessed candidates are indeed solutions. In the case of the encodings for ADFs, the guessing part of the programs outline possible assignments of truth values to the statements, that is, an ADF interpretation. For the three-valued semantics, as the admissible semantics, the rules are as follows:

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

We follow Bichler *et al.* (2016b) in encoding NP-checks in large non-ground rules having bodies with predicates of bounded arity. In particular, in all our encodings, we will need rules encoding the semantic evaluation of propositional formulas, for example, the evaluation of the acceptance conditions by completions of an interpretation. Given a propositional formula ϕ, for this, we introduce the function Ω. For assignments of truth values (1 and 0) to the propositional variables in ϕ, $\Omega(\phi)$ gives us a set of atoms corresponding to the propagation of the truth values to the subformulas of ϕ in accordance with the semantics of classical propositional logic. The atoms make use of ASP variables V_ψ where ψ is a subformula of ϕ. The variables V_p, where p is a propositional variable occurring in ϕ, can be used by other parts of ASP rules employing the atoms in $\Omega(\phi)$ for purposes of assigning intended truth values to the propositional variables in ϕ.

For the definition of the atoms $\Omega(\phi)$, we rely on the ASP built-in arithmetic functions `&` (bitwise AND), `?` (bitwise OR), and `-` (subtraction). We also use the built-in comparison predicate $=$. Let ϕ be a propositional formula over a set of propositional variables P; then the set of atoms in question is defined as

$$\Omega(\phi) := \begin{cases} \Omega(\phi_1) \cup \Omega(\phi_2) \cup \{V_\phi = V_{\phi_1}\texttt{\&}V_{\phi_2}\} & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \Omega(\phi_1) \cup \Omega(\phi_2) \cup \{V_\phi = V_{\phi_1}\texttt{?}V_{\phi_2}\} & \text{if } \phi = \phi_1 \vee \phi_2 \\ \Omega(\psi) \cup \{V_\phi = 1\texttt{-}V_\psi\} & \text{if } \phi = \neg\psi \\ \emptyset & \text{if } \phi = p \in P \end{cases}$$

where V_ϕ, V_{ϕ_1} V_{ϕ_2}, and V_ψ are variables representing the subformulas of ϕ.

Our encoding for the admissible semantics, π_{adm}, is based on the fact that 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. $w(\varphi_s) = 1$.

This is a simple consequence of the definition of the admissible semantics. Any $w \in [v]_2$ which contradicts this simple observation (e.g. $v(s) = 1$ and $w(\varphi_s) = 0$) is a "counter-model" to v being an admissible interpretation. The constraining part of our encoding for the admissible semantics essentially disallows guessed assignments of truth values to the statements of an ADF corresponding to ADF interpretations which have counter-models to them being admissible.

To encode the constraints of our encoding, we need auxiliary rules firing when the guessed assignments have counter-models to them being admissible. These rules, two for each $s \in S$, make use of bodies ω_s where $\Omega(\varphi_s)$ is employed to evaluate the acceptance conditions by the completions. The latter is obtained by setting variables V_t for $t \in par_D(s)$ with the adequate truth values using the predicates *asg* and *lt* defined in π_{guess} and π_{lt}:

$$\omega_s := \{asg(t, Y_t), lt(Y_t, V_t) \mid t \in par_D(s)\} \cup \Omega(\varphi_s).$$

The two rules for every statement $s \in S$ have heads $sat(s)$ and $inv(s)$ that fire in case there is some completion of the interpretation corresponding to the assignments guessed in the program fragment π_{guess} such that the acceptance condition φ_s evaluates to 1 and 0, respectively:

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

Here for an ADF interpretation v "guessed" via the fragment π_{guess} (and encoded using the predicate "*asg*") and a $s \in S$, $sat(s)$ is derived whenever $v(s) = 0$ and there is a $w \in [v]_2$ for which $w(\phi_s) = 1$. On the other hand, $inv(s)$ is derived whenever $v(s) = 1$ and there is a $w \in [v]_2$ for which $w(\phi_s) = 0$. The atoms using predicates "*asg*" and "*lt*" in ω_s are used to encode possible assignments a completion $w \in [v]_2$ can take, while the atoms in $\Omega(\varphi_s)$ propagate such assignments to φ_s in accordance with the semantics of classical propositional logic by making use of ASP built-ins and auxiliary variables V_ψ for every subformula ψ of φ_s.

The encoding for the admissible semantics now results from compounding the program fragments $\pi_{arg}(D)$, π_{lt}, π_{guess}, and $\pi_{sat}(D)$ together with ASP constraints which filter out guessed assignments of statements to truth values (via π_{guess}) corresponding to interpretations of D having counter-models to being admissible:

$$\pi_{adm}(D) := \pi_{arg}(D) \ \cup \ \pi_{lt} \ \cup \ \pi_{guess} \ \cup \ \pi_{sat}(D) \cup$$
$$\{\texttt{:-}\,arg(S), asg(S,1), inv(S). \ \ \texttt{:-}\,arg(S), asg(S,0), sat(S).\}.$$

For instance, the last constraint in $\pi_{adm}(D)$ disallows guessed interpretations v for which there is a $s \in S$ and $w \in [v]_2$ such that $v(s) = 0$ and $w(\varphi_s) = 1$.

Proposition 1 formally states that π_{adm} is an adequate encoding function. For the proof, which is prototypical for most of the proofs of correctness in this work, we use the notation

$$I_p := \{p(t_1, \ldots, t_n) \in I\}.$$

For an ASP interpretation I (set of ground atoms), I_p represents I projected onto the predicate p (with arity n).

Proposition 1
For every ADF D it holds that $adm(D) \cong \mathscr{AS}(\pi_{adm}(D))$.

Proof
Let $D = (S, \{\varphi_S\}_{s\in S})$ be an ADF and $v \in adm(D)$. Let also

$$\begin{aligned} I := \ & \{arg(s) \mid s \in S\} \cup \\ & \{lt(\mathbf{u},0), lt(\mathbf{u},1), lt(1,1), lt(0,0)\} \cup \\ & \{asg(s,x) \mid s \in S, v(s) = x\} \cup \\ & \{sat(s) \mid \textit{if there is a } w \in [v]_2 \textit{ s.t. } w(\phi_s) = 1\} \cup \\ & \{inv(s) \mid \textit{if there is a } w \in [v]_2 \textit{ s.t. } w(\phi_s) = 0\}, \end{aligned}$$

be a set of ground atoms (such that $v \cong I$). We prove now that $I \in \mathscr{AS}(\pi_{adm}(D))$.

We start by proving that I satisfies $\pi_{adm}(D)^I$. First note that I satisfies $\pi_{arg}(D)^I = \pi_{arg}(D)$ as well as $\pi_{lt}^I = \pi_{lt}$ since all the atoms making up the facts in these two modules are in I (first two lines of the definition of I). I also satisfies

$$\begin{aligned} \pi_{guess}^I = \{ & asg(s,x)\,\texttt{:-}\,arg(s). \mid s \in S, \\ & asg(s,y) \notin I, asg(s,z) \notin I, x \in \{1,0,\mathbf{u}\}, y,z \in (\{1,0,\mathbf{u}\} \setminus \{x\})\}, \end{aligned}$$

since, first of all, $arg(s) \in I$ iff $s \in S$ by the first line of the definition of I (and the fact that the predicate arg does not appear in the head of any rules other than $\pi_{arg}(D)^I = \pi_{arg}(D)$). Secondly, for any $s \in S$, $asg(s,x) \in I$ whenever $asg(s,y) \notin I$ and $asg(s,z) \notin I$ for $x \in \{1,0,\mathbf{u}\}$ and $y,z \in (\{1,0,\mathbf{u}\} \setminus \{x\})$ by the fact that $v \cong I$ (third line of the definition of I).

Now consider the rule $r \in \pi_{sat}(D)$ with $H(r) = sat(S)$ and a substitution θ s.t. $\theta r \in \pi_{sat}(D)^I$. This means that θr is of the form

$$sat(s)\,\texttt{:-}\,\theta\omega_s, \theta(V_{\varphi_s} = 1),$$

with

$$\theta\omega_s = \{asg(t,y_t), lt(y_t,v_t) \mid t \in par_D(s)\} \cup \theta\Omega(\varphi_s),$$

and where $\theta(Y_t) = y_t$, $\theta(V_t) = v_t$. If $B(\theta r) \in I$, it must be the case that $y_t \in \{1, 0, \mathbf{u}\}$, $v_t \in \{0, 1\}$ for $t \in par_D(s)$ and $\theta\Omega(\varphi_s) \in I$. Now it should be easy for the reader to see that from the fact that $\{asg(t, y_t), lt(y_t, v_t) \mid t \in par_D(s)\} \subseteq I$ and $v \cong I$ it is the case that $w \in [v]_2$ for the ADF interpretation w defined as $w(t) = v_t$ for every $t \in par_D(s)$. It is also simple to establish that $\theta\Omega(\varphi_s) \in I$ and $\theta(V_{\varphi_s} = 1) \in I$ imply that $w(\varphi_s) = 1$. Hence, by the fourth line of the definition of I we have $sat(s) \in I$, that is, I satisfies θr. In the same manner, by the fifth line of the definition of I it follows that I satisfies any grounding $\theta r \in \pi_{sat}(D)^I$ for the rule r s.t. $H(r) = inv(S)$. In conclusion, I satisfies $\pi_{sat}(D)^I$.

Let us turn now to a ground instance $r \in \pi_{adm}(D)^I$

$$:\!-\, arg(s), asg(s, 0), sat(s).$$

of the constraint

$$:\!-\, arg(S), asg(S, 0), sat(S).$$

$\in \pi_{adm}(D)$. By the fourth line of the definition of I, $sat(s) \in I$ iff there is a $w \in [v]_2$ s.t. $w(\varphi_s) = 1$. But then by the fact that $v \in adm(D)$ and $v \cong I$, $asg(s, 0) \notin I$, that is, r cannot be satisfied by I. In the same manner also any ground instance in $\pi_{adm}(D)^I$ of the constraint

$$:\!-\, arg(S), asg(S, 1), inv(S).$$

cannot be satisfied by I.

We have established that I satisfies $\pi_{adm}(D)^I$. We continue our proof of $I \in \mathscr{AS}(\pi_{adm}(D))$ by now showing that there is no $I' \subset I$ that satisfies $\pi_{adm}(D)^I$.

In effect, consider any other I' that satisfies $\pi_{adm}(D)^I$. Note first of all that then $I'_{arg} \supseteq I_{arg}$ and $I'_{lt} \supseteq I_{lt}$ because both I' and I satisfy $\pi_{arg}(D)^I$ as well as π^I_{lt}. Hence, also $I'_{asg} \supseteq I_{asg}$ because I' satisfies π^I_{guess} (see the proof of I satisfies $\pi_{adm}(D)^I$ for the structure of π^I_{guess}) and $I'_{arg} \supseteq I_{arg}$, that is, $B(r) \subseteq I'$ for every $r \in \pi^I_{guess}$. But then, since $I'_{arg} \supseteq I_{arg}$ and $I'_{asg} \supseteq I_{asg}$, and I' satisfies all the comparison predicates with arithmetic functions that I does by definition, I' satisfies all the rules in $\pi_{sat}(D)^I$ that I does (see again the proof of I satisfies $\pi_{adm}(D)^I$ for the form of such rules). Hence, also $I'_{sat} \supseteq I_{sat}$ and $I'_{inv} \supseteq I_{inv}$. In conclusion, $I' \supseteq I$.

Since I' was general, we derive that there is no $I' \subset I$ that satisfies $\pi_{adm}(D)^I$. Together with the fact that I satisfies $\pi_{adm}(D)^I$, we have that $I \in \mathscr{AS}(\pi_{adm}(D))$.

We now turn to proving that for any $I \in \mathscr{AS}(\pi_{adm}(D))$, it holds that $v \in adm(D)$ for $v \cong I$. Note first that for such an I, since I satisfies $\pi_{arg}(D)^I = \pi_{arg}(D)$ as well as

$$\pi^I_{guess} = \{asg(s, x) :\!-\, arg(s). \mid s \in S,$$
$$asg(s, y) \notin I, asg(s, z) \notin I, x \in \{1, 0, \mathbf{u}\}, y, z \in (\{1, 0, \mathbf{u}\} \setminus \{x\})\},$$

for every $s \in S$ there is a $x \in \{1, 0, \mathbf{u}\}$ such that $asg(s, x) \in I$. Also, $asg(s, x) \in I$ whenever $asg(s, y) \notin I$ and $asg(s, z) \notin I$ for $y, z \in (\{1, 0, \mathbf{u}\} \setminus \{x\})$. That is, v s.t. $v \cong I$ is well defined.

Now assume that $v \notin adm(D)$. Then there are $s \in S$, $w \in [v]_2$ for which either i) $v(s) = 1$ and $w(\varphi_s) = 0$ or ii) $v(s) = 0$ and $w(\varphi_s) = 1$. Let us consider the case i). In that case, consider a substitution θ for the rule $r \in \pi_{sat}(D)$

$$inv(s) :\!-\, \Omega_s, V_{\varphi_s} = 0.$$

where

$$\Omega_s = \{asg(t, Y_t), lt(Y_t, V_t) \mid t \in par_D(s)\} \cup \Omega(\varphi_s).$$

The substitution θ is defined as $\theta(Y_t) = v(t)$ and $\theta(V_t) = w(t)$ for every $t \in par_D(s)$. Since $v \cong I$, we have that $asg(t, \theta(Y_t)) \in I$ for every $t \in par_D(s)$. Also $lt(\theta(Y_t), \theta(V_t)) \in I$, since I satisfies π^I_{lt}.

Now, by definition $\theta\Omega(\varphi_s) \subseteq I$ and from $w(\varphi_s) = 0$, it is easy to see that it follows that also $\theta(V_{\varphi_s} = 0) \in I$, that is, $\theta r \in \pi_{sat}(D)^I$ and $B(\theta r) \subseteq I$. This means that also $inv(s) \in I$. As a consequence, we have that $B(r') \subseteq I$ for the constraint r'

$$:\!-\, arg(s), asg(s, 1), inv(s).$$

in $\pi_{adm}(D)^I$. This is a contradiction to $I \in \mathscr{AS}(\pi_{adm}(D))$. From the case ii) $v(s) = 0$ and $w(\varphi_s) = 1$, a contradiction can be derived in analogous manner. Hence, $v \in adm(D)$ must be the case. □

Example 3

Considering the ADF D from Example 1, $\pi_{adm}(D)$ (as implemented by our system YADF with minor formatting for purposes of readability; see Section 5) looks as follows:

```
arg(a).
arg(b).
arg(c).
leq(u,0).
leq(u,1).
leq(0,0).
leq(1,1).
asg(S,u) :- arg(S),not asg(S,0),not asg(S,1).
asg(S,0) :- arg(S),not asg(S,1),not asg(S,u).
asg(S,1) :- arg(S),not asg(S,u),not asg(S,0).
sat(a) :- asg(b,Y0),leq(Y0,V0),V1=1-V0,V2=V1?V0,V2=1.
sat(b) :- asg(b,Y0),leq(Y0,V0),V0=1.
sat(c) :- asg(c,Y0),leq(Y0,V0),asg(b,Y1),leq(Y1,V1),
                                  V3=1,V3=V2?V1,V2=1-V0.
inv(a) :- asg(b,Y0),leq(Y0,V0),V1=1-V0,V2=V1?V0,V2=0.
inv(b) :- asg(b,Y0),leq(Y0,V0),V0=0.
inv(c) :- asg(c,Y0),leq(Y0,V0),asg(b,Y1),leq(Y1,V1),
                                  V3=V2?V1,V3=0,V2=1-V0.
:- arg(S),asg(S,1),inv(S).
:- arg(S),asg(S,0),sat(S).
```

A possible output of an ASP solver (the current one is the simplified output of clingo version 4.5.4) given this instance looks as follows (only showing *asg*, *sat*, and *inv* predicates):

```
Answer: 1
asg(c,u) asg(b,0) asg(a,u) sat(c) sat(a) inv(c) inv(b)
```

```
Answer: 2
asg(c,u) asg(b,0) asg(a,1) sat(c) sat(a) inv(c) inv(b)
Answer: 3
asg(c,u) asg(b,1) asg(a,u) sat(b) sat(c) sat(a)
Answer: 4
asg(c,u) asg(b,1) asg(a,1) sat(b) sat(c) sat(a)
Answer: 5
asg(c,u) asg(b,u) asg(a,1) sat(b) sat(c) sat(a) inv(c) inv(b)
Answer: 6
asg(c,u) asg(b,u) asg(a,u) sat(b) sat(c) sat(a) inv(c) inv(b)
Answer: 7
asg(c,1) asg(b,1) asg(a,u) sat(b) sat(c) sat(a)
Answer: 8
asg(c,1) asg(b,1) asg(a,1) sat(b) sat(c) sat(a)
SATISFIABLE
```

◇

The encoding π_{adm} allows to enumerate the admissible interpretations of an ADF D from the answer sets of $\pi_{adm}(D)$ (as explained in the opening paragraphs of Section 3). Skeptical reasoning for the admissible semantics is trivial (as the interpretation mapping every statement to $\mathbf{u}$ is always admissible), but note that via credulous reasoning for ASP programs we directly obtain results for credulous reasoning w.r.t. the admissible semantics from $\pi_{adm}(D)$ (for any ADF D). The latter translation and thus the encoding π_{adm} are adequate from the point of view of the complexity (see Table 2) as $\pi_{adm}(D)$ is a normal logic program for any ADF D. Also, given our recursive definition of the evaluation of the acceptance conditions within ASP rules, the arity of predicates in our encodings are bounded (in fact, the maximum arity of predicates is 2).

3.2 Encoding for the complete semantics

For the ASP encoding of the complete semantics, we only need to add two constraints to the encoding of the admissible semantics. These express a further condition that an interpretation v for an ADF $D = (S, \{\varphi_S\}_{s \in S})$ has to fulfill to be complete, in addition to not having counter-models for being an admissible interpretation as expressed in Section 3.1. The condition in question is that 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 $w_2(\varphi_s) = 1$.

Expressing this condition in the form of constraints gives us the encoding

$$\begin{aligned}\pi_{com}(D) := \pi_{adm}(D) \cup \\ \{ &:\!-\, \mathit{arg}(S), \mathit{asg}(S, \mathbf{u}), \mathit{not}\ \mathit{inv}(S). \\ &:\!-\, \mathit{arg}(S), \mathit{asg}(S, \mathbf{u}), \mathit{not}\ \mathit{sat}(S).\}.\end{aligned}$$

Proposition 2
For every ADF D, it holds that $\mathit{com}(D) \cong \mathscr{AS}(\pi_{com}(D))$.

Proof
(sketch) The proof extends that of Proposition 1. Only the additional constraints used in the encoding for the complete semantics (w.r.t. the encoding for the admissible semantics) need to be accounted for. □

The encoding π_{com} allows to enumerate the complete interpretations of an ADF D by applying the encoding on D ($\pi_{com}(D)$) and considering $\mathscr{AS}(\pi_{com}(D))$. Since credulous acceptance for the complete semantics is equivalent to credulous acceptance for the admissible semantics, we obtain a complexity adequate means of computing credulous acceptance for the complete semantics via applying credulous (ASP) reasoning on $\pi_{adm}(D)$ (π_{adm} being the encoding presented in Section 3.1). Applying credulous reasoning on $\pi_{com}(D)$ is nevertheless also an option[1].

3.3 Saturation encoding for the 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. Existence of a non-saturated guess hence implies that the property of interest does not hold for the guess in question.

In the encoding of the preferred semantics for an ADF D, we extend $\pi_{adm}(D)$ by making use of the saturation 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. As a consequence, the relevant interpretation must be preferred according to the definition of this semantics for ADFs.

The module π_{guess2} amounts to "making a second guess" (indicated by the predicate *asg2*) extending the "first guess" (*asg*) from π_{guess}:

$$\begin{aligned}\pi_{\text{guess2}} := \{&\mathit{asg2}(S,0)\,\text{:-}\,\mathit{asg}(S,0).\\ &\mathit{asg2}(S,1)\,\text{:-}\,\mathit{asg}(S,1).\\ &\mathit{asg2}(S,1) \vee \mathit{asg2}(S,0) \vee \mathit{asg2}(S,\mathbf{u})\,\text{:-}\,\mathit{asg}(S,\mathbf{u}).\}.\end{aligned}$$

Note that the first two rules express that if an ADF interpretation v corresponding to the "first guess" (captured via the predicate *asg*) maps a statement to either 0 or 1 then so does a interpretation v' corresponding to the "second guess" (captured via the predicate *asg2*). The last rule, on the other hand, indicates that if the first guess maps a statement to $\mathbf{u}$ then the second guess can map the statement to either of the truth values $\mathbf{u}$, 0, or 1. Thus, $v' \geq_i v$ is guaranteed.

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

$$\begin{aligned}\pi_{sat2}(D) := \{&\mathit{sat2}(s)\,\text{:-}\,\omega 2_s, V_{\varphi_s} = 1.\\ &\mathit{inv2}(s)\,\text{:-}\,\omega 2_s, V_{\varphi_s} = 0. \mid s \in S\}\end{aligned}$$

[1] Which may in fact (because of redundancy) be more efficient in practice.

with

$$\omega 2_s := \{asg2(t, Y_t), lt(Y_t, V_t) \mid t \in par_D(s)\} \cup \Omega(\varphi_s).$$

The only difference between the fragment $\pi_{sat2}(D)$ and the fragment $\pi_{sat}(D)$ that we introduced in Section 3.1 is that we now evaluate acceptance conditions w.r.t. completions of the second guess given via the predicate *asg2*.

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 module π_{guess}) or is not admissible (last two rules of $\pi_{check}(D)$). We will say that in this case, the second guess is *not* a counterexample to the first guess corresponding to a preferred interpretation of D. We here assume that the statements S of D are numbered, that is, $S = \{s_1, \ldots, s_k\}$ for a $k \geq 1$:

$$\begin{aligned}\pi_{check}(D) := \{&saturate\text{:-}asg(s_1, X_1), asg2(s_1, X_1), \ldots\\ &\qquad asg(s_k, X_k), asg2(s_k, X_k).\\ &saturate\text{:-}asg2(S, 1), inv2(S).\\ &saturate\text{:-}asg2(S, 0), sat2(S).\}.\end{aligned}$$

The module $\pi_{saturate}$ now assures that whenever the atom *saturate* is derived, first of all $asg2(s, 0)$, $asg2(s, 1)$, and $asg2(s, \mathbf{u})$ are derived for every $s \in S$ for which $asg(s, \mathbf{u})$ has been derived. Also, $sat2(s)$ and $inv2(s)$ are derived for every $s \in S$:

$$\begin{aligned}\pi_{saturate} := \{&asg2(S, 0)\text{:-}asg(S, \mathbf{u}), saturate.\\ &asg2(S, 1)\text{:-}asg(S, \mathbf{u}), saturate.\\ &asg2(S, \mathbf{u})\text{:-}asg(S, \mathbf{u}), saturate.\\ &sat2(S)\text{:-}arg(S), saturate.\\ &inv2(S)\text{:-}arg(S), saturate.\}.\end{aligned}$$

The effect of this fragment is that whenever all the "second guesses" (computed via π_{guess2}) are *not* counterexamples 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 *asg2*, *sat2*, and *inv2*, that is, the same ground instances of these predicates will be included in any answer set. Thus, all answer sets (corresponding to the ADF interpretation determined by the first guess) will be indistinguishable on the new predicates used for the encoding of the preferred interpretation; meaning: those not in $\pi_{adm}(D)$. On the other hand, were there to be a counterexample to the first guess corresponding to a preferred interpretation of D, then a non-saturated and hence smaller (w.r.t $\subseteq$) answer set could be derived. We disallow the latter by adding to the program fragments $\pi_{adm}(D)$, π_{guess2}, $\pi_{sat2}(D)$, $\pi_{check}(D)$, $\pi_{saturate}$, a constraint filtering out precisely such answer sets. The latter being those for which the atom *saturate* is *not* derived. We thus arrive at the following encoding for the preferred semantics:

$$\begin{aligned}\pi_{prf}(D) := &\pi_{adm}(D) \cup \pi_{guess2} \cup \pi_{sat2}(D) \cup\\ &\quad \pi_{check}(D) \cup \pi_{saturate} \cup \{\text{:-}not\ saturate.\}.\end{aligned}$$

Proposition 3
For every ADF D, it holds that $\mathit{prf}(D) \cong \mathscr{AS}(\pi_{prf}(D))$.

Proof
Let $D = (S, \{\varphi_S\}_{s \in S})$ be an ADF and $v \in \mathit{prf}(D)$. Let also

$$I' := I \cup \{\mathit{asg2}(s,1) \mid s \in S, v(s) = 1\} \cup \{\mathit{asg2}(s,0) \mid s \in S, v(s) = 0\} \cup I^{\triangle},$$

be a set of ground atoms where I is defined as in the "only if" direction of the proof of Proposition 1 (hence, $v \cong I'$). Moreover, $I^{\triangle}$ is the set of ground atoms forming the "saturation" of the predicates *asg2*, *sat2*, *inv2*, *saturate* (*asg2* is saturated only for $s \in S$ s.t. $v(s) = \mathbf{u}$) defined as

$$\begin{aligned} I^{\triangle} := \; & \{\mathit{asg2}(s,x) \mid s \in S, x \in \{1,0,\mathbf{u}\}, v(s) = \mathbf{u}\} \; \cup \\ & \{\mathit{sat2}(s) \mid s \in S\} \; \cup \\ & \{\mathit{inv2}(s) \mid s \in S\} \; \cup \\ & \{\mathit{saturate}\}. \end{aligned}$$

Note first that since none of the predicates occurring in $I' \setminus I$ appear in $\pi_{adm}(D)$, we have that $\pi_{adm}(D)^{I'} = \pi_{adm}(D)^{I}$. As thus also all of the atoms appearing in $\pi_{adm}(D)^{I'}$ that are in I' are those which are in I, we have that I' and I satisfy the bodies and heads of the same rules in $\pi_{adm}(D)^{I'}$. By the proof of the "only if" direction of Proposition 1 (i.e. that I satisfies $\pi_{adm}(D)^{I} = \pi_{adm}(D)^{I'}$), it then follows that I' satisfies $\pi_{adm}(D)^{I'}$.

I' also satisfies each of $\pi_{sat2}(D)^{I'} = Gr(\pi_{sat2}(D))$, $\pi_{check}(D)^{I'} = Gr(\pi_{check}(D))$ as the heads of all possible ground instances of the rules of each of the modules $\pi_{sat2}(D)$ and $\pi_{check}(D)$ are contained in $I^{\triangle} \subset I'$. Moreover, I' satisfies all groundings of the first two rules of π_{guess2} (that are in $\pi_{guess2}^{I'} = Gr(\pi_{guess2})$) as both $\mathit{asg}(s,x) \in I'$ and $\mathit{asg2}(s,x) \in I'$ whenever $v(s) = x$ for $x \in \{1,0\}$. I' also satisfies all groundings of the third rule of π_{guess2} as whenever $\mathit{asg}(s,u) \in I'$ this means that $v(s) = \mathbf{u}$ and then $\mathit{asg2}(s,x) \in I^{\triangle} \subset I'$ for every $x \in \{1,0,\mathbf{u}\}$. For the same reason, I' also satisfies all possible groundings of the first three rules of $\pi_{saturate}$ (contained in $\pi_{saturate}^{I'} = Gr(\pi_{saturate})$). Furthermore, I' satisfies all possible groundings of the last two rules of $\pi_{saturate}$ since whenever $\mathit{arg}(s) \in I'$ this means that $s \in S$ and then $\mathit{sat2}(s) \in I^{\triangle} \subset I'$ as well as $\mathit{inv2}(s) \in I^{\triangle} \subset I'$. Finally, since $\mathit{saturate} \in I^{\triangle} \subset I'$ the constraint

$$:\!-\, \mathit{not\ saturate}.$$

is deleted from $\pi_{prf}(D)$ when forming the reduct $\pi_{prf}(D)^{I'}$. We thus have that I' satisfies all of the rules in $\pi_{prf}(D)^{I'}$; hence, I' satisfies $\pi_{prf}(D)^{I'}$.

Consider now that there is a $I'' \subset I'$ that satisfies $\pi_{prf}(D)^{I'}$. Since I'' satisfies $\pi_{adm}(D)^{I'} = \pi_{adm}(D)^{I}$, we have by the argument in the "only if" direction of the proof of Proposition 1 that $I \subseteq I''$. Note that then $\mathit{asg}(s,x) \in I''$ for every $s \in S$ s.t. $v(s) = x$ for $x \in \{1,0\}$. On the other hand, I'' satisfies the groundings of the first two rules in π_{guess2} (since $\pi_{guess2}^{I'} = Gr(\pi_{guess2})$). It hence follows that also $\mathit{asg2}(s,x) \in I''$ for every $s \in S$ s.t. $v(s) = x$ for $x \in \{1,0\}$. Moreover, since I'' satisfies the groundings of the last rule in π_{guess2} and $\{\mathit{asg}(s,\mathbf{u}) \mid s \in S, v(s) = \mathbf{u}\} \subset I \subset I''$, it must be the case that there is some $x \in \{\mathbf{u}, 1, 0\}$ s.t. $asg2(s,x) \in I''$ for every $s \in S$ s.t. $v(s) = \mathbf{u}$. We thus have that there is an ADF interpretation $v' \geq_i v$ s.t. there is an atom $asg2(s,x) \in I''$ whenever $v'(s) = x$.

Assume now that $saturate \notin I''$. Since I'' satisfies $\pi_{saturate}^{I'} = Gr(\pi_{saturate})$, this means that $B(r) \not\subseteq I''$ for every $r \in Gr(\pi_{saturate})$. Hence, in particular, $B(r) \not\subseteq I''$ for the rule r

$$\begin{aligned} saturate \text{:-} asg(s_1, v(s_1)), asg2(s_1, v'(s_1)), \ldots \\ asg(s_k, v(s_k)), asg2(s_k, v'(s_k)). \end{aligned}$$

This amounts to $v \neq v'$ and, hence, $v <_i v'$. Also, $B(r) \not\subseteq I''$ for the rule r

$$saturate \text{:-} asg2(s,1), inv2(s).$$

for every $s \in S$. This amounts to (since I'' satisfies $\pi_{sat2}(D)^{I'} = Gr(\pi_{sat2}(D))$; see proof of Proposition 1) there not being any $s \in S$ and $w \in [v']_2$ for which $v'(s) = 1$ and $w(s) = 0$. In the same manner, the fact that $B(r) \not\subseteq I''$ for the rule r

$$saturate \text{:-} asg2(s,0), sat2(s).$$

for every $s \in S$ means that there is no $s \in S$ and $w \in [v']_2$ for which $v'(s) = 0$ and $w(s) = 1$. But then $v' \in adm(D)$ which, together with the fact that $v <_i v'$, is a contradiction to $v \in prf(D)$.

On the other hand if $saturate \in I''$, since I'' satisfies all possible groundings of the first three rules of $\pi_{saturate}$ (as $\pi_{saturate}^{I'} = Gr(\pi_{saturate})$), it would be the case that whenever $asg(s, \mathbf{u}) \in I''$ and hence $v(s) = \mathbf{u}$ (since $I \subset I''$) also $asg2(s,x) \in I''$ for every $x \in \{\mathbf{u}, 0, 1\}$. Moreover, if $saturate \in I''$, since I'' satisfies all possible groundings of the last two rules of $\pi_{saturate}$, it would also follow that $sat(s) \in I''$ as well as $inv(s) \in I''$ for every $s \in S$. This means that if $saturate \in I''$, then $I' \subseteq I''$. This is a contradiction to our assumption that $I'' \subset I'$. In conclusion, there is no $I'' \subset I'$ that satisfies $\pi_{prf}(D)^{I'}$. Therefore, $I' \in \mathscr{AS}(\pi_{prf}(D))$.

We turn now to proving that for any $I \in \mathscr{AS}(\pi_{prf}(D))$, it holds that $v \in prf(D)$ for $v \cong I$. Note first of all that since I satisfies $\pi_{adm}(D)^I$ by the proof of the "if" direction of Proposition 1, we obtain that v is well defined and, moreover, $v \in adm(D)$.

Since $v \in adm(D)$, $v \notin prf(D)$ would mean that there is a $v' \in adm(D)$ s.t. $v' >_i v$. Now, notice first of all that since $I \in \mathscr{AS}(\pi_{prf}(D))$, $saturate \in I$ since otherwise the constraint

$$\text{:-} not\ saturate.$$

would not be deleted from $\pi_{prf}(D)$ (as must be the case) when forming the reduct $\pi_{prf}(D)^I$ and hence $\pi_{prf}(D)$ would have no answer set. We know from the proof of the "only if" direction of Proposition 3 that from $saturate \in I$ it then follows that $I^{\triangle} \subseteq I$ where

$$\begin{aligned} I^{\triangle} = \; & \{asg2(s,x) \mid s \in S, x \in \{1, 0, \mathbf{u}\}, v(s) = \mathbf{u}\} \cup \\ & \{sat2(s) \mid s \in S\} \cup \\ & \{inv2(s) \mid s \in S\} \cup \\ & \{saturate\}. \end{aligned}$$

Now let us define

$$\begin{aligned} I' := & \cup_{p\in\{arg,lt,asg,sat,inv\}} I_p \cup \{asg2(s,v'(s)) \mid s\in S\} \cup \\ & \{sat2(s) \mid s\in S, \text{ there is a } w\in[v']_2 \text{ s.t. } w(\varphi_s)=1\} \cup \\ & \{inv2(s) \mid s\in S, \text{ there is a } w\in[v']_2 \text{ s.t. } w(\varphi_s)=1\} \end{aligned}$$

for which by construction (and $v' >_i v$) $I' \subset I$ holds. Notice first of all that since all negative atoms of $\pi_{prf}(D)$ occur in $\pi_{adm}(D) \cup \{\, \textit{:-not saturate.}\}$, we have that

$$\pi_{prf}(D)^I = \pi_{adm}(D)^I \cup Gr(\pi_{guess2}) \cup Gr(\pi_{sat2}(D)) \cup Gr(\pi_{check}(D)) \cup Gr(\pi_{saturate}).$$

Now, since I and I' are the same when considering the atoms occurring in $\pi_{adm}(D)^I$ (meaning: $\cup_{p\in\{arg,lt,asg,sat,inv\}} I_p \subset I'$) and I satisfies $\pi_{adm}(D)^I$ so does I'. Moreover, since $v' >_i v$ by construction $asg2(s,x) \in I'$ whenever $asg(s,x) \in I$ for $x \in \{1,0\}$ and there is a $y \in \{\mathbf{u},1,0\}$ s.t. $asg2(s,y) \in I'$ whenever $asg(s,\mathbf{u}) \in I$. Hence, I' also satisfies $\pi^I_{guess2} = Gr(\pi_{guess2})$.

Using analogous arguments as in the "only if" direction of the proof of Proposition 1, from the fact that $asg2(s,x) \in I'$ iff $v'(s) = x$ (for $s \in S$ and $x \in \{1,0,\mathbf{u}\}$) and the definition for when $sat2(s)$ and $inv2(s)$ are in I', it follows that v' satisfies $\pi^I_{sat2} = Gr(\pi_{sat2})$. We have also seen in the proof of the "only if" direction of Proposition 3 that $v' \neq v$ and $v' \in adm(D)$ implies that I' does not satisfy the body of any of the rules in $\pi_{check}(D)^I = Gr(\pi_{check}(D))$. Finally, since $saturate \notin I'$ it is also the case that I' satisfies $\pi^I_{saturate} = Gr(\pi_{saturate})$. In conclusion, we have that I' satisfies $\pi_{prf}(D)^I$ and $I' \subset I$ which contradicts $I \in \mathcal{AS}(\pi_{prf}(D)))$. Hence, there cannot be a $v' >_i v$ s.t. $v' \in adm(D)$ and therefore $v \in prf(D)$ must be the case. □

Example 4

The encoding $\pi_{prf}(D)$ for the ADF D from Example 1 as implemented by our system `YADF` looks as follows:

```
leq(u,0).
leq(u,1).
leq(0,0).
leq(1,1).
arg(a).
arg(b).
arg(c).
asg(S,u) :- arg(S),not asg(S,0),not asg(S,1).
asg(S,0) :- arg(S),not asg(S,1),not asg(S,u).
asg(S,1) :- arg(S),not asg(S,u),not asg(S,0).
sat(a) :- asg(b,Y0),leq(Y0,V0),V1=1-V0,V2=V1?V0,V2=1.
sat(b) :- asg(b,Y0),leq(Y0,V0),V0=1.
sat(c) :- asg(c,Y0),leq(Y0,V0),asg(b,Y1),leq(Y1,V1),
                                  V3=1,V3=V2?V1,V2=1-V0.
inv(a) :- asg(b,Y0),leq(Y0,V0),V1=1-V0,V2=V1?V0,V2=0.
inv(c) :- asg(c,Y0),leq(Y0,V0),asg(b,Y1),leq(Y1,V1),
                                  V3=V2?V1,V3=0,V2=1-V0.
inv(b) :- asg(b,Y0),leq(Y0,V0),V0=0.
```

```
:- arg(S),asg(S,1),inv(S).
:- arg(S),asg(S,0),sat(S).
asg2(S,0) :- asg(S,0).
asg2(S,1) :- asg(S,1).
asg2(S,0)|asg2(S,1)|asg2(S,u) :- asg(S,u).
sat2(a) :- asg2(b,Y0),leq(Y0,V0),V1=1-V0,V2=V1?V0,V2=1.
sat2(b) :- asg2(b,Y0),leq(Y0,V0),V0=1.
sat2(c) :- asg2(c,Y0),leq(Y0,V0),asg2(b,Y1),leq(Y1,V1),
                                  V3=1,V3=V2?V1,V2=1-V0.
inv2(b) :- asg2(b,Y0),leq(Y0,V0),V0=0.
inv2(c) :- asg2(c,Y0),leq(Y0,V0),asg2(b,Y1),leq(Y1,V1),
                                  V3=V2?V1,V3=0,V2=1-V0.
inv2(a) :- asg2(b,Y0),leq(Y0,V0),V1=1-V0,V2=V1?V0,V2=0.
saturate :- asg(c,X0),asg2(c,X0),asg(b,X1),asg2(b,X1),
                                  asg(a,X2),asg2(a,X2).
saturate :- arg(S),asg2(S,0),sat2(S).
saturate :- arg(S),asg2(S,1),inv2(S).
asg2(S,u) :- asg(S,u),saturate.
asg2(S,0) :- asg(S,u),saturate.
asg2(S,1) :- asg(S,u),saturate.
sat2(S) :- arg(S),saturate.
inv2(S) :- arg(S),saturate.
:- not saturate.
```

An output of an ASP solver given this instance looks as follows (only showing *asg* and *saturate* predicates):

```
Answer: 1
asg(c,u) asg(b,0) asg(a,1) saturate
Answer: 2
asg(c,1) asg(b,1) asg(a,1) saturate
SATISFIABLE
```

◇

Note that π_{prf}, in addition to providing a means of enumerating the preferred interpretations of any ADF, also gives us a complexity adequate means of deciding skeptical acceptance problems. The latter via skeptical reasoning of ASP disjunctive programs with predicates of bounded arity (see Table 2). Credulous reasoning for the preferred semantics is equivalent to credulous reasoning for the admissible semantics; applying credulous reasoning on the encoding given in Section 3.1 hence provides a means of computation at the right level of complexity for this reasoning task.

3.4 Encoding for the grounded semantics

Our encoding for the grounded semantics is based on the fact that (see Strass and Wallner (2015)) $v \in grd(D)$ for an interpretation v and an ADF $D = (S, \{\phi_s\}_{s \in S})$ iff v is the (unique) $\leq_i$-minimal interpretation satisfying

- for each $s \in S$ such that $v(s) = 1$, there exists an interpretation $w \in [v]_2$ for which $w(\phi_s) = 1$,
- for each $s \in S$ such that $v(s) = 0$, there exists an interpretation $w \in [v]_2$ for which $w(\phi_s) = 0$, and
- for each $s \in S$ such that $v(s) = \mathbf{u}$, there exist interpretations $w_1 \in [v]_2$ and $w_2 \in [v]_2$ such that $w_1(\phi_s) = 1$ and $w_2(\phi_s) = 0$.

We say that an interpretation v for the ADF D that satisfies exactly one of the above for a specific $s \in S$ (e.g. $v(s) = 1$ and there exists an interpretation $w \in [v]_2$ for which $w(\phi_s) = 1$), that it satisfies the properties for being a candidate for being the grounded interpretation w.r.t. s. The completion w, or alternatively the completions w_1 and w_2, verify this fact for v and s. If v satisfies the properties w.r.t. every $s \in S$, then v is a candidate for being the grounded interpretation of D. An interpretation $v' <_i v$ that is also a candidate for being the grounded interpretation is a counter-model (alternatively, counterexample) to v being the right candidate (for being the grounded interpretation).

Our encoding for the grounded semantics essentially consists first of all, once more in the guessing part π_{guess} where we guess assignments of truth values to the statements of the ADF of interest D. This corresponds to guessing an interpretation v for D. Constraints in our encoding filter out guessed interpretations which either are not candidates to being the grounded interpretation or which have counter-models to being the right candidate. These constraints rely on the rules in $\pi_{sat}(D)$ defined in Section 3.1 and rules defining when an interpretation has a counter-model to being the right candidate, respectively.

We start with a few facts needed for our encoding. First of all, we use facts analogous to those in π_{lt} defined in Section 3.1 for encoding the truth values a possible counter-model to the interpretation guessed via π_{guess} (being the right candidate for the grounded interpretation) can assign to the statements. Here, we also need an additional argument (the first argument of the predicate *lne*) allowing us to check whether the interpretation in question is distinct from the one determined by the predicate *asg*:

$$\pi_{lne} := \{\mathit{lne}(1, \mathbf{u}, 1).\ \mathit{lne}(1, \mathbf{u}, 0).\} \cup$$
$$\{\mathit{lne}(0, 1, 1).\ \mathit{lne}(0, 0, 0).\ \mathit{lne}(0, \mathbf{u}, \mathbf{u}).\}.$$

Given a candidate for the grounded interpretation v determined by the atoms *asg*, for instance, the two first facts in π_{lne} express (using the last two arguments of the predicate *lne*) that if for a statement s, $v(s) = x$ with $x \in \{1, 0\}$, then a counter-model v' to v being the right candidate (for the grounded interpretation) can map s to the truth value $\mathbf{u}$. Moreover, the first argument of the alluded to facts indicates that in this case $v'(s) \neq v(s)$.

Secondly, we need a set of facts for checking whether an interpretation satisfies the properties required for candidates to being the grounded interpretation mentioned at the beginning of this section. Specifically, given a statement s of the ADF D, $\mathit{prop}(z_1, z_2, z_3)$ can be used to check whether the correct relationship between $z_1 = v(s)$, $z_2 = w_1(\varphi_s)$, and $z_3 = w_2(\varphi_s)$ holds for an interpretation v for D, and $w_1, w_2 \in [v]_2$ (e.g. that if $v(s) = \mathbf{u}$ then there must be $w_1 \in [v]_2$, $w_2 \in [v]_2$ s.t. $w_1(\varphi_s) = 1$ and $w_2(\varphi_s) = 0$). In particular, note that $w_1 = w_2$ is possible and hence $\mathit{prop}(x, y, z)$ can also be used to check the properties for when $v(s) = x$ and $x \in \{1, 0\}$ (first two facts in π_{prop}):

$$\pi_{prop} := \{\mathit{prop}(1, 1, 1).\ \mathit{prop}(0, 0, 0).\ \mathit{prop}(\mathbf{u}, 0, 1).\}.$$

The following module consists of constraints checking whether the interpretation corresponding to the assignments guessed via π_{guess} is a candidate (hence the use of the identifier "*ca*") for being the grounded interpretation:

$$\pi_{ca}(D) := \{\text{:-}\,arg(S), asg(S,1), not\ sat(S).\ \ \text{:-}\,arg(S), asg(S,0), not\ inv(S).\} \cup \{\text{:-}\,arg(S), asg(S,\mathbf{u}), not\ inv(S).\ \ \text{:-}\,arg(S), asg(S,\mathbf{u}), not\ sat(S).\}.$$

The module $\pi_{ca}(D)$ assumes, as we stated earlier, that the rules in $\pi_{sat}(D)$ (and thus the facts in π_{lt}) defined in Section 3.1 are also part of the encoding for the grounded semantics. For instance, the first constraint then checks that there is no $s \in S$ for which it holds that $v(s) = 1$ but there is no $w \in [v]_2$ for which $w(\varphi_s) = 1$ for the interpretation v guessed via the atoms constructed with the predicate *asg*.

Now, note that, since, as we explained before, the grounded interpretation is the minimal (w.r.t. $\leq_i$) of the interpretations that are candidates for being the grounded interpretation, this interpretation can be obtained via choosing the minimal interpretation w.r.t. $\leq_i$ from all interpretations that correspond to some answer set of the encoding

$$\pi_{ca\text{-}grd}(D) := \pi_{arg}(D) \cup \pi_{lt} \cup \pi_{guess} \cup \pi_{sat}(D) \cup \pi_{ca}(D);$$

that is, what essentially boils down to a skeptical acceptance problem for $\pi_{ca\text{-}grd}(D)$.

In order to obtain an encoding not requiring (in the worst case) processing of all answer sets, we need a rule defining when an interpretation is a counter-model to the interpretation determined via the predicate *asg* being the right candidate. For this, we will need to make repeated use of the function Ω defined in Section 3.1 within a single rule. We therefore first of all make the symbol ranging over the ASP variables representing subformulas of a propositional formula ϕ within $\Omega(\phi)$ an explicit parameter of the function. This is straightforward, but for completeness we give the full definition of our parametrized version of the function Ω. Here ϕ is once more a propositional formula built from propositional variables in a set P, while now V is an arbitrary (meta-) symbol used to refer to the variables introduced by the function:

$$\Omega^V(\phi) := \begin{cases} \Omega^V(\phi_1) \cup \Omega^V(\phi_2) \cup \{V_\phi = V_{\phi_1} \& V_{\phi_2}\} & \text{if } \phi = \phi_1 \wedge \phi_2 \\ \Omega^V(\phi_1) \cup \Omega^V(\phi_2) \cup \{V_\phi = V_{\phi_1} ? V_{\phi_2}\} & \text{if } \phi = \phi_1 \vee \phi_2 \\ \Omega^V(\psi) \cup \{V_\phi = 1\text{-}V_\psi\} & \text{if } \phi = \neg\psi \\ \emptyset & \text{if } \phi = p \in P. \end{cases}$$

Again, V_ϕ, V_{ϕ_1} V_{ϕ_2}, and V_ψ are variables representing the subformulas of ϕ. From now on, whenever we introduce sets $\Omega^{V_1}(\phi_1)$ and $\Omega^{V_2}(\phi_2)$ for possibly identical formulas ϕ_1 and ϕ_2 but distinct symbols V_1 and V_2, we implicitly also assume that then $\Omega^{V_1}(\phi_1) \cap \Omega^{V_2}(\phi_2) = \emptyset$.

Our rule for defining counter-models to an interpretation being the right candidate for the grounded interpretation requires first of all a part for "generating" an interpretation less informative (w.r.t $\leq_i$) and distinct from the interpretation determined by π_{guess}, that is, a candidate counter-model. For this, we use the atoms

$$\lambda_D := \{asg(s, X_s), lne(E_s, Y_s, X_s) \mid s \in S\} \cup \Omega^E(\vee_{s \in S} s) \cup \{E_{\vee_{s \in S} s} = 1\}.$$

Here, given an interpretation v determined by the atoms $asg(s, X_s)$ $(s \in S)$, the atoms $lne(E_s, Y_s, X_s)$ are used to generate an assignment of truth values to the statements (via argument Y_s) corresponding to an interpretation $v' \leq_i v$. Then $\Omega^E(\vee_{s\in S} s) \cup \{E_{\vee_{s\in S} s} = 1\}$ are used (via the arguments E_s of the atoms $lne(E_s, Y_s, X_s)$) to check that there is a $s \in S$ for which $v'(s) \neq v(s)$ and hence in fact $v' <_i v$. Thus, if v is a candidate for the grounded interpretation, then v' is a candidate counter-model for v being the grounded interpretation.

We now introduce the following set of atoms to check whether the candidate counter-model is indeed a counter-model to the interpretation determined by *asg* being the grounded interpretation. We need to check the properties candidates for being the grounded interpretation need to satisfy for each of the statements s of the ADF of interest D; therefore, the need for having sets of atoms $\kappa_{s,D}$ defined for every statement s:

$$\begin{aligned}\kappa_{s,D} :=& \{lt(Y_t, V^{(t,s),1}) \mid t \in par_D(s)\} \cup \Omega^{V^{(t,s),1}}(\varphi_s) \cup \\ & \{lt(Y_t, V^{(t,s),2}) \mid t \in par_D(s)\} \cup \Omega^{V^{(t,s),2}}(\varphi_s) \cup \\ & \{prop(Y_s, V^{(t,s),1}_{\varphi_s}, V^{(t,s),2}_{\varphi_s})\}.\end{aligned}$$

Note here the use of the predicate *lt* (defined via the module π_{lt} from Section 3.1) for generating assignments to statements corresponding to completions. The first two lines of the definition of $\kappa_{s,D}$ are used for generating completions $w_1, w_2 \in [v']_2$ of an interpretation v' and the outcome of the evaluation of φ_s by the completions. Then the third line is used to check that w_1, w_2 verify that v' is a candidate for being the grounded interpretation (and hence a counter-model for a $v >_i v'$ being the right candidate for the grounded interpretation).

Putting all the above together, we have quite a large rule defining when a candidate counter-model is indeed a counter-model to the interpretation determined by π_{guess} being the right candidate for the grounded interpretation:

$$\pi_{cm}(D) := \{cm\text{:-}\lambda_D \cup \bigcup_{s\in S} \kappa_{s,D}\}.$$

The following is then an encoding allowing to compute the grounded interpretation for the ADF D in one go:

$$\pi_{grd}(D) := \pi_{arg}(D) \cup \pi_{lt} \cup \pi_{lne} \cup \pi_{prop} \cup \pi_{guess} \cup \pi_{sat}(D) \cup \pi_{ca}(D) \cup \pi_{cm}(D) \cup \{\text{:-}cm.\}.$$

Note, in particular, the constraint $\{\text{:-}cm.\}$ disallowing interpretations having a counter-model to them being the right candidate for the grounded interpretation.

Proposition 4
For every ADF D, it holds that $grd(D) \cong \mathscr{AS}(\pi_{grd}(D))$,

Proof
(sketch) The proof is similar to that of Proposition 5. Indeed note that $\pi_{stb}(D)$ (defined in Section 3.5) essentially builds on $\pi_{grd}(D)$, the main difference being the slightly more complex versions of π_{lt}, π_{lne}, π_{prop}, $\pi_{cm}(D)$ and the use of $\pi_{model}(D)$ (see Section 3.5) rather than $\pi_{ca}(D)$ (and $\pi_{sat}(D)$). □

We do not obtain complexity sensitive means of deciding credulous and skeptical reasoning w.r.t. the grounded semantics via the encoding π_{grd}. Nevertheless, the encoding offers an alternative strategy to deriving the grounded interpretation of an ADF to that of the static encodings at the basis of the `DIAMOND` family of systems mentioned in the introduction to this work. Also, the encoding forms the basis of the complexity adequate (w.r.t. the reasoning problems) encoding for the stable semantics we present in Section 3.5.

3.5 Encoding for the stable semantics

As already indicated and is to be expected given the definition of this semantics, our encoding for the stable semantics is based on the encoding for the grounded semantics. Nevertheless, some modifications are required. First of all, we need to guess assignments to statements of an ADF D corresponding to a two-valued rather than a three-valued interpretation v for D. Secondly, we need to check that v is a model of D. Third, we need to ensure that v assigns the truth value 1 to the same statements as the grounded interpretation of the reduct of D^v (i.e. $v \downharpoonright_{E_v} = \mathit{grd}(D^v)$) rather than D simpliciter.

To start, we slightly modify some facts used in previous encodings. Our encoding once more follows the guess & check methodology and there will therefore be a part used to guess a candidate v for being the stable interpretation of our ADF of interest D. We will then need a modified version of π_{lt} (defined in Section 3.1) to set the right truth values for completions of a candidate counter-model v' to v (actually $v \downharpoonright_{E_v}$) being the right candidate for the grounded interpretation (as explained in Section 3.4) of the reduct D^v:

$$\pi'_{lt} := \{\mathit{lt2}(1,\mathbf{u},0).\ \mathit{lt2}(1,\mathbf{u},1).\ \mathit{lt2}(1,0,0).\ \mathit{lt2}(1,1,1).\} \cup$$
$$\{\mathit{lt2}(0,\mathbf{u},0).\ \mathit{lt2}(0,0,0).\ \mathit{lt2}(0,1,0).\}.$$

The first four facts in π'_{lt} are as to those in π_{lt}. The only difference is in the first argument which we use to encode the assignment of a truth value to a statement s by a guessed candidate for the grounded interpretation v of the reduct D^v. The other two values express possible values a completion w of a candidate counter-model v' to v (being the right candidate for the grounded interpretation of D^v) can assign to the statement s. For instance, $\mathit{lt2}(1,\mathbf{u},0)$ expresses that if $v(s) = 1$ and $v'(s) = \mathbf{u}$ then one of the two possible assignments of a truth value to s by $w \in [v']_2$ is $w(s) = 0$. The three last facts in π'_{lt} now indicate possible assignments of truth values to a statement s by a $w \in [v']_2$ when $v(s) = 0$. In order to simulate the evaluation of the acceptance conditions of the reduct D_v by the completions of v' in other parts of our encoding, we enforce that in this case $w(s) = 0$ whatever the value of $v'(s)$. This amounts to replacing each statement s for which $v(s) = 0$ within an acceptance condition $\varphi_{s'}$ in which the statement s occurs by $\bot$ as is required by the definition of the reduct.

The module π_{lne} defined in Section 3.4 also needs to be modified (by one fact) to account for the fact that a counter-model v' to an interpretation v being the right candidate for satisfying $v \downharpoonright_{E_v} = \mathit{grd}(D^v)$ must be distinct from v on the statements assigned the truth value 1 (i.e. there must be at least one statement s to which v assigns the truth value 1 and v' the truth value $\mathbf{u}$):

$$\pi'_{lne} := \{\mathit{lne}(1,\mathbf{u},1).\ \mathit{lne}(0,\mathbf{u},0).\} \cup$$
$$\{\mathit{lne}(0,1,1).\ \mathit{lne}(0,0,0).\ \mathit{lne}(0,\mathbf{u},\mathbf{u}).\}.$$

The difference of π'_{lne} w.r.t π_{lne} is thus in the second fact "$\mathit{lne}(0,\mathbf{u},0)$." where the first argument in the corresponding fact in π_{lne} is 1 rather than 0.

We also need to modify π_{prop} defined in Section 3.4 adding an extra-argument (again, the first one) to indicate whether the property required per statement of an ADF for candidates for the grounded interpretation is verified or not:

$$\begin{aligned}\pi'_{prop} := \{&\mathit{prop2}(1,1,1,1).\ \mathit{prop2}(1,0,0,0).\ \mathit{prop2}(1,\mathbf{u},0,1).\ \mathit{prop2}(1,\mathbf{u},1,0).\} \cup\\ \{&\mathit{prop2}(0,1,0,1).\ \mathit{prop2}(0,1,1,0).\ \mathit{prop2}(0,1,0,0).\} \cup\\ \{&\mathit{prop2}(0,0,0,1).\ \mathit{prop2}(0,0,1,0).\ \mathit{prop2}(0,0,1,1).\} \cup\\ \{&\mathit{prop2}(0,\mathbf{u},0,0).\ \mathit{prop2}(0,\mathbf{u},1,1).\}.\end{aligned}$$

Here for an ADF of interest (in our encoding, the reduct D^v of the interpretation v guessed to be stable), we list all possible combinations of truth values of $v(s)$, $w_1(\varphi_s)$, $w_2(\varphi_s)$ for $w_1, w_2 \in [v]_2$ (three last arguments in the facts) and indicate (first argument in the facts) whether the combination in question makes w_1, w_2 witnesses of v being a candidate for the grounded interpretation w.r.t. the statement s (as explained in Section 3.4). For instance, $\mathit{prop2}(1,0,0,0)$ indicates that $w_1(\phi_s) = w_2(\phi_s) = 0$ makes w_1, w_2 witnesses of v being a candidate (w.r.t. s) when $v(s) = 0$.

As already indicated, also our encoding for the stable semantics builds on a module guessing possible assignments to the statements of the ADF D. We only need to slightly modify π_{guess} as defined in Section 3.1 to obtain a conjecture for the stable interpretation corresponding to a two-valued rather than three-valued interpretation for D:

$$\begin{aligned}\pi'_{guess} := \{&\mathit{asg}(S,0) \text{ :- } \mathit{not}\ \mathit{asg}(S,1), \mathit{arg}(S).\\ &\mathit{asg}(S,1) \text{ :- } \mathit{not}\ \mathit{asg}(S,0), \mathit{arg}(S).\}.\end{aligned}$$

In order to check that the guessed interpretation is a model of D, we again need to evaluate the acceptance conditions of D but this time by the guessed interpretation. For this, we make use of the following sets of atoms per statement s of D:

$$\mu_s := \{\mathit{asg}(t, V_t) \mid t \in \mathit{par}_D(s)\} \cup \Omega(\varphi_s).$$

Here, we again make use of the function Ω defined in Section 3.1 but this time to evaluate the acceptance condition φ_s by the interpretation guessed to be stable (and thus a model) of D via π'_{guess}.

The following are then constraints, one per statement, filtering out guesses that are not models of D:

$$\pi_{model}(D) := \{\text{:- } \mathit{asg}(s, V_s), \mu_s, V_s \neq V_{\varphi_s}. \mid s \in S\}.$$

More to the point, the constraints filter out any guessed interpretation v (via π'_{guess}), for which $v(s) \neq v(\phi_s)$ for some statement s.

Now, note that for any $v \in \mathit{mod}(D)$, $v \downharpoonright_{E_v}$ is a candidate to being the grounded interpretation of the reduct D^v. The reason is first of all that $[v \downharpoonright_{E_v}]_2 = \{v \downharpoonright_{E_v}\}$ and, hence, for any $w \in [v \downharpoonright_{E_v}]_2$, $w(\varphi'_s) = v \downharpoonright_{E_v} (\varphi'_s) = v(\varphi_s)$ for the modified acceptance conditions $\varphi'_s = \phi_s[b/\bot : v(b) = 0]$ of D^v. As a consequence, clearly whenever $v \downharpoonright_{E_v} (s) = x$ for $x \in \{1, 0\}$ (in fact, $x = 1$) there is a $w \in [v]_2$, namely $w = v \downharpoonright_{E_v}$, for which $w(\varphi_s) = x$. In effect, the latter is the case by virtue of $v \in \mathit{mod}(D)$ and hence $v(\varphi_s) = v \downharpoonright_{E_v} (\varphi'_s) = x$ whenever $v(s) = x$. Also, there are no statements for which $v \downharpoonright_{E_v} (s) = \mathbf{u}$. The consequence for our encoding for the stable semantics is that $\pi_{model}(D)$ suffices for checking whether our guessed interpretation, when projected on the statements to which it assigns the truth value 1, is a candidate for being the grounded interpretation of D^v.

All that remains for our encoding of the stable semantics is therefore, as we have for the encoding of the grounded semantics, a constraint filtering out guessed interpretations which have counter-models to being the right candidate for being the grounded interpretation of D^v. For this, we introduce a slightly modified version of $\pi_{cm}(D)$ (defined in Section 3.4) accounting for the fact that we need to check for counter-models to $v \downharpoonright_{E_v}$ being the right candidate for the reduct D^v rather than v and D. This means that completions of potential counter-models need to set any statement set to the truth value 0 by v also to 0. To encode this, we use the predicate *lt2* rather than *lt* in our modified version $\kappa'_{s,D}$ of the set of atoms $\kappa_{s,D}$ (from Section 3.4). Also, we need to check the properties that candidates of the grounded interpretation need to satisfy only for statements s for which $v(s) = 1$. To encode this, we make use of the predicate *prop2* rather than *prop* and add a corresponding check using ASP built-in Boolean arithmetic functions:

$$\begin{aligned}\kappa'_{s,D} :=&\{\mathit{lt2}(X_t, Y_t, V^{(t,s),1}) \mid t \in \mathit{par}_D(s)\} \cup \Omega^{V^{(t,s),1}}(\varphi_s) \cup \\ &\{\mathit{lt2}(X_t, Y_t, V^{(t,s),2}) \mid t \in \mathit{par}_D(s)\} \cup \Omega^{V^{(t,s),2}}(\varphi_s) \cup \\ &\{\mathit{prop2}(P_s, Y_s, V^{(t,s),1}_{\varphi_s}, V^{(t,s),2}_{\varphi_s})\} \cup \{CX_s = 1 - X_s, O_s = P_s?CX_s, O_s = 1\}.\end{aligned}$$

Our modified module $\pi'_{cm}(D)$ of $\pi_{cm}(D)$ is then as follows:

$$\pi'_{cm}(D) := \{cm\texttt{:-}\lambda_D \cup \bigcup_{s\in S} \kappa'_{s,D}\}.$$

Note that we here make use of the set of atoms λ_D as defined in Section 3.4, yet relying on the definition of the predicate *lne* as given by the module π'_{lne} rather than π_{lne}. Putting everything together, the encoding for the stable semantics has the following form:

$$\begin{aligned}\pi_{stb}(D) :=&\pi_{arg}(D) \ \cup\ \pi'_{lt} \ \cup\ \pi'_{lne} \ \cup\ \pi'_{prop} \ \cup \\ &\pi'_{guess} \ \cup\ \pi_{model}(D) \ \cup\ \pi'_{cm}(D) \ \cup\ \{\texttt{:-}cm.\}.\end{aligned}$$

Proposition 5
For every ADF D, it holds that $stb(D) \cong \mathscr{AS}(\pi_{stb}(D))$.

Proof
Let ADF be an ADF and $v \in stb(D)$. Let also

$$I := \pi_{arg}(D) \ \cup\ \pi'_{lt} \ \cup\ \pi'_{lne} \ \cup\ \pi'_{prop} \ \cup\ \{asg(s, x) \mid s \in S, v(s) = x\}$$

be a set of ground atoms (such that $v \cong I$). (We slightly abuse the notation here using, for example, $\pi_{arg}(D)$ to refer to the set of atoms rather than the facts in the module.) We prove now that $I \in \mathscr{AS}(\pi_{stb}(D))$.

We start by proving that I satisfies $\pi_{stb}(D)^I$. Note first that I satisfies each of $\pi_{arg}(D)^I = \pi_{arg}(D)$, $\pi'^I_{lt} = \pi'_{lt}$, $\pi'^I_{lne} = \pi'_{lne}$, $\pi'^I_{prop} = \pi'_{prop}$ since all of the facts in each of these modules are in I. I also satisfies

$$\pi'^I_{guess} = \{asg(s, x)\texttt{:-}arg(s). \mid s \in S, asg(s, y) \notin I, x \in \{1, 0\}, y \in (\{1, 0\}\ \{x\})\}$$

by the fact that $v \cong I$.

Assume now that I satisfies the body of some constraint in $\pi_{model}(D)^I$, that is, there is a $s \in S$ and a substitution θ s.t. $asg(s, \theta(V_s)) \in I$, $asg(t, \theta(V_t)) \in I$ for each $t \in par_D(s)$, $\theta(\Omega(\varphi_s)) \in I$, and $\theta(V_s \neq V_{\varphi_s}) \in I$. This translates to $v(s) \neq v(\varphi_s)$ which means $v \notin mod(D)$ and contradicts $v \in stb(D)$. Therefore, I does not satisfy any of the constraints in $\pi_{model}(D)^I$.

Consider on the other hand that I satisfies the body of some rule in $\pi'_{cm}(D)^I$. This means that there is a substitution θ such that first of all $asg(s, \theta(X_s)) \in I$ as well as $lne(\theta(E_s), \theta(Y_s), \theta(X_s)) \in I$ for every $s \in S$. Also $\theta(\Omega^E(\vee_{s \in S} s)) \in I$ and $\theta(E_{\vee_{s\in S} s} = 1) \in I$. All of this together means that $v'(s) <_i v(s)$ for the interpretation v' defined as $v'(s) := \theta(Y_s)$. Moreover, since I satisfies π'_{lt}, there is an $s \in S$ s.t. $v(s) = 1$ and $v'(s) = \mathbf{u}$. This means that also $E_v \neq \emptyset$ and $v' \downharpoonleft_{E_v} (s) <_i v \downharpoonleft_{E_v} (s)$ where $v \downharpoonleft_{E_v}$ and $v' \downharpoonleft_{E_v}$ are interpretations of the reduct D^v.

Secondly, for every $s \in S$, we have that $lt2(\theta(X_t), \theta(Y_t), \theta(V^{(t,s),1})) \in I$ and it is also the case that $lt2(\theta(X_t), \theta(Y_t), \theta(V^{(t,s),2})) \in I$ for every $t \in par_D(s)$. Consider the interpretations w_i for $i \in \{1, 2\}$ defined as $w_i(t) = \theta(V^{(t,s),i})$ for every $t \in par_D(s)$. Then $w_i(t) \geq_i v'(t)$ whenever $v(t) = 1$, but $w_i(t) = v(t) = 0$ if $v(t) = 0$. This means that $w_i(\varphi_s) = w_i \downharpoonleft_{E_v} (\varphi'_s)$ for $\varphi'_s = \varphi_s[b/\bot : v(b) = 0]$, and $w_i \downharpoonleft_{E_v} \in [v' \downharpoonleft_{E_v}]_2$. Now from $\theta(\Omega^{V^{(t,s),1}}(\varphi_s)) \in I$ for every $t \in par_D(s)$, $\theta(\Omega^{V^{(t,s),2}}(\varphi_s)) \in I$ for every $t \in par_D(s)$, and the fact that $prop2(\theta(P_s), \theta(Y_s), \theta(V^{(t,s),1}_{\varphi_s}), \theta(V^{(t,s),2}_{\varphi_s})) \in I$ we have that $\theta(P_s) = 1$ whenever $w_1 \downharpoonleft_{E_v}$ and $w_2 \downharpoonleft_{E_v}$ verify that $v' \downharpoonleft_{E_v}$ satisfies the properties for being a candidate for the grounded interpretation of D^v w.r.t. $s \in E_v$. Otherwise, $\theta(P_s) = 0$. Moreover, from $\theta(CX_s = 1 - X_s) \in I$, $\theta(O_s = P_s?CX_s) \in I$, and $\theta(O_s = 1) \in I$, it follows that either $\theta(P_s) = 1$ or $\theta(X_s) = v(s) = 0$ for every $s \in S$.

In other words, whenever $s \in E_v$ (remember: $E_v \neq \emptyset$), there are completions that verify that $v' \downharpoonleft_{E_v}$ satisfies the properties for being a candidate for the grounded interpretation of D^v w.r.t. s. This means that $v' \downharpoonleft_{E_v}$ is a counter-model to $v \downharpoonleft_{E_v}$ being the grounded interpretation of D^v. This is a contradiction to $v \in stb(D)$. Therefore, I does not satisfy the body of any rule in $\pi'_{cm}(D)^I$ and, hence, satisfies $\pi'_{cm}(D)^I$. Finally, since $cm \notin I$, I does not satisfy the body of the constraint $\{{:}{-}cm.\} \in \pi_{stb}(D)^I$. In conclusion, I satisfies $\pi_{stb}(D)^I$.

Now consider any other I' that satisfies $\pi_{stb}(D)^I$. Clearly, since I' satisfies all of the facts in $\pi_{stb}(D)^I$, we have that $\pi_{arg}(D) \cup \pi'_{lt} \cup \pi'_{lne} \cup \pi'_{prop} \subseteq I'$. But also because of the form of π'^{I}_{guess} (see above) and the fact that I' satisfies $\pi_{arg}(D)^I$, it must be the case that $\{asg(s, x) \mid s \in S, v(s) = x\} \subset I'$. This means that in addition to I satisfying $\pi_{stb}(D)^I$ there is also no $I' \subset I$ that satisfies $\pi_{stb}(D)^I$; that is, we have that $I \in \mathscr{AS}(\pi_{stb}(D))$.

We now turn to proving that for any $I \in \mathscr{AS}(\pi_{stb}(D))$, it holds that $v \in stb(D)$ for $v \cong I$. Note first that for any such I, since I satisfies $\pi_{arg}(D)^I = \pi_{arg}(D)$ and π'^{I}_{guess}, v s.t. $v \cong I$ is well defined. Now assume that $v \notin stb(D)$. Then either (i) $v \notin mod(D)$ or (ii) $v \in mod(D)$ but $v \downharpoonleft_{E_v} \notin grd(D^v)$.

In the first case, (i) there must be a $s \in S$ s.t. $v(s) \neq v(\varphi_s)$. Consider hence the substitution θ defined as $\theta(V_s) = v(s)$ and $\theta(V_t) = v(t)$ for $t \in par_D(s)$. This substitution is s.t. $\theta(B(r)) \subseteq I$ for the constraint in $\pi_{model}(D)$ corresponding to s. This would mean that I does not satisfy $\pi_{stb}(D)^I$ which is a contradiction. Therefore, $v \in mod(D)$ which also means that $v \downharpoonleft_{E_v}$ is a candidate for the grounded interpretation of D^v (as we argued in detail while explaining our encoding $\pi_{stb}(D)$).

Consider now the case (ii). Since $v \downharpoonleft_{E_v}$ is a candidate for the grounded interpretation of D^v but $v \downharpoonleft_{E_v} \notin grd(D^v)$ this means there is a counter-model $v' \downharpoonleft_{E_v}$ for $v \downharpoonleft_{E_v}$ being the right candidate for the grounded interpretation of D^v. First define v' to be s.t. $v'(s) = v' \downharpoonleft_{E_v} (s)$ for $s \in E_v$ while $v'(s) = v(s)$ for $s \notin E_v$. Define then the substitution θ for which $\theta(X_s) = v(s)$ and $\theta(Y_s) = v'(s)$ for every $s \in S$. Since $v' \neq v$ (because $v' \downharpoonleft_{E_v} \neq v \downharpoonleft_{E_v}$), there must be an $s \in E_v \subseteq S$ for which $v(s) \neq v'(s)$. Set $\theta(E_s) = 1$ for all such $s \in S$, but $\theta(E_s) = 0$ whenever $v(s) = v'(s)$. We thus have that $\theta(\Omega^E(\vee_{s \in S} s)) \in I$ and $E_{\vee_{s \in S} s} = 1 \in I$. Hence, $\theta(\lambda_D) \in I$.

Now, since $v' \downharpoonleft_{E_v}$ is a counter-model to $v \downharpoonleft_{E_v}$ being the right candidate for the grounded interpretation of D^v we have that for every $s \in E_v$ there are completions $w_{s,1}$ and $w_{s,2}$ of $v' \downharpoonleft_{E_v}$ that are witnesses for $v' \downharpoonleft_{E_v}$ satisfying the properties candidates for the grounded interpretation need to satisfy w.r.t. s. Hence, we continue defining the substitution θ s.t. $\theta(V^{(t,s),i}) = w_{s,i}(t)$ for every $s \in E_v$ and $t \in par_D(s) \cap E_v$. On the other hand, $\theta(V^{(t,s),i}) = 0$ for $t \in par_D(s) \setminus E_v$. Also, $\theta(P_s) = 1$ for every $s \in E_v$.

For $s \notin E_v$, we on the other hand define $\theta(V^{(t,s),i}) = w(t)$ $(i \in \{1, 2\})$, $t \in par_D(s) \cap E_v$ for some arbitrary $w \in [v']_2$. On the other hand, $\theta(V^{(t,s),i}) = 0$ for $t \in par_D(s) \setminus E_v$. Also, $\theta(P_s) = 1$ whenever $v'(s) = w(\varphi'_s)$ $(\varphi'_s = \phi_s[b/\bot : v(b) = 0])$ and $\theta(P_s) = 0$ otherwise. Then, we have that $\theta(\Omega^{V^{(t,s),i}}(\varphi_s)) \in I$ for $s \in S$, $t \in par_D(s)$, $(i \in \{1, 2\})$. Also, $\theta(CX_s = 1 - X_s) \in I$, $\theta(O_s = P_s?CX_s) \in I$, and $O_s = 1 \in I$ for every $s \in S$ $(\theta(P_s) = 1$ for $s \in E_v$, while $\theta(CX_s) = 1$ for $s \notin E_v)$. That is, $\theta(\kappa'_{s,D}) \in I$ for every $s \in S$. Hence, since also $\theta(\lambda_D) \in I$, we have that the body of a rule in $\pi'_{cm}(D)^I$ is satisfied by I and, therefore, $cm \in I$. This means that the constraint $\texttt{:-}cm. \in \pi_{stb}(D)^I$ is satisfied by I which contradicts $I \in \mathcal{AS}(\pi_{stb}(D))$. Therefore, the case ii) is also not possible and $v \in stb(D)$ must be the case. $\square$

Example 5
The encoding $\pi_{stb}(D)$ for the ADF D from Example 1 as implemented by our system `YADF` (we slightly condense the encoding by generating some facts using rules) looks as follows:

```
arg(a).
arg(b).
arg(c).
val(u).
val(0).
val(1).
lt2(1,u,1).
lt2(1,u,0).
lt2(1,0,0).
lt2(1,1,1).
lt2(0,X,0) :- val(X).
lne(1,u,1).
lne(0,u,0).
lne(0,X,X):- val(X).
prop(1,1,1,1).
prop(1,0,0,0).
prop(1,u,1,0).
prop(1,u,0,1).
prop(0,X1,X2,X3) :- val(X1),val(X2),val(X3),not prop(1,X1,X2,X3).
```

```
asg(S,1) :- arg(S),not asg(S,0).
asg(S,0) :- arg(S),not asg(S,1).
:- asg(b,V0),V0!=V0.
:- asg(a,V1),asg(b,V0),V3=V2?V0,V3!=V1,V2=1-V0.
:- asg(c,V0),asg(b,V1),V3=V2?V1,V3!=V0,V2=1-V0.
cm :- asg(c,X0),asg(b,X1),asg(a,X2),lne(E0,Y0,X0),lne(E1,Y1,X1),
      lne(E2,Y2,X2),E20=E0?E1,E21=E2?E20,E21=1,lt2(X0,Y0,V1),
      lt2(X1,Y1,V2),V4=V3?V2,V3=1-V1,lt2(X0,Y0,V5),lt2(X1,Y1,V6),
      V7=1-V5,V8=V7?V6,prop(P0,Y0,V4,V8),CX0=1-X0,OR9=P0?CX0,OR9=1,
      lt2(X1,Y1,V10),lt2(X1,Y1,V11),prop(P1,Y1,V10,V11),CX1=1-X1,
      OR12=P1?CX1,OR12=1,lt2(X1,Y1,V13),V15=V14?V13,V14=1-V13,
      lt2(X1,Y1,V16),V17=1-V16,V18=V17?V16,prop(P2,Y2,V15,V18),
      CX2=1-X2,OR19=P2?CX2,OR19=1.
:- cm.
```

An output of an ASP solver given this instance looks as follows:

```
UNSATISFIABLE
```

and indicates that the ADF at hand does not possess any stable model. ◇

Concluding our presentation of dynamic encodings for ADFs, we note that also π_{stb}, in addition to giving us a means of computing the stable interpretations of any ADF, provides us with a complexity-attuned mechanism to decide credulous and skeptical reasoning tasks via the corresponding ASP reasoning tasks (see Table 2).

4 GRAPPA encodings

We now illustrate how to extend the methodology used in our construction of dynamic encodings for ADFs to GRAPPA. For this purpose, we give ASP encodings for the admissible, complete, and preferred semantics. Reflecting the relationship between ADFs and GRAPPA, structurally the encodings are very similar to those for ADFs, the main difference being in the encoding of the evaluation of the acceptance patterns.

Also for our encodings for GRAPPA, we make use of the correspondence $\cong$ between three-valued interpretations (now for GRAPPA instances) and sets of ground atoms (interpretations of ASP programs) defined via ASP atoms $asg(s,x)$ 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 $\sigma(G) \cong \mathscr{AS}(\pi_\sigma(G))$ (see Definition 1 for the formal meaning of the latter overloaded use of $\cong$). We will reuse several of the ASP fragments we defined for ADFs. Formally, this amounts to extending the corresponding encoding functions to also admit GRAPPA instances as arguments.

Throughout this section, let $G = (S, E, L, \lambda, \alpha)$ be a GRAPPA instance with $S = \{s_1, \ldots, s_k\}$. As already hinted at, the main difference between the encodings for GRAPPA and ADFs is in the definition of the set of atoms $\Omega(\phi)$ (first defined for ADFs in Section 3.1) corresponding to the semantic evaluation of the acceptance conditions (now patterns) associated with the statements. The recursive function representing the evaluation of patterns needs a statement s as an additional parameter and for the encoding of the basic patterns is defined as $\Omega_s(\phi) :=$

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

The difference between Ω_s and Ω (note the missing subscript s) as defined in Section 3 is in the last line where $P_s(\tau) \cup \{V_\phi = \#\text{sum}\{1 : V_\tau \bar{R} a\}\}$ encodes the evaluation of a basic pattern $\phi = \tau R a$. Here, we make use of the ASP aggregate #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 #sum, #min, #max, and #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 of the active labels, on the other hand, can be constructed based on the instance G only. We define $T_s(t)$ as

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

For instance, the first atom $V_t = \#\text{sum}\{Z_{r_{i_1}}, r_{i_1}; \ldots ; Z_{r_{i_w}}, r_{i_w}\}$ corresponds to the computation of $val^s_o(\#l)$ when $par(s,l) = \{r_{i_1}, \ldots, r_{i_w}\} \neq \emptyset$. Here, $o = m^z_s$ is the multiset of active labels of s under z and the assignments of truth values to statements of the ADF interpretation z are captured via the variables $Z_{r_{i_j}}$ $(1 \leq j \leq z)$. An assumption of the encoding is that such variables $Z_{r_{i_j}}$ take only values 1 or 0, thus corresponding to two-valued interpretations (e.g. completions). This, as in the encodings for ADFs, is taken care of in the rules in which the atoms Ω_s occur (see the re-definition of the modules π_{sat} and π_{sat2} defined in Section 3 for GRAPPA instances below).

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

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

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

$$\begin{aligned}
\pi'_{sat}(G) := \{& sat(s) {:}\text{-}\, \omega_s, V_{\alpha(s)} = 1.\\
& inv(s) {:}\text{-}\, \omega_s, V_{\alpha(s)} = 0. \mid s \in S\};\\
\pi'_{sat2}(G) := \{& sat2(s) {:}\text{-}\, \omega 2_s, V_{\alpha(s)} = 1.\\
& inv2(s) {:}\text{-}\, \omega 2_s, V_{\alpha(s)} = 0. \mid s \in S\}
\end{aligned}$$

where we make use of the shortcuts ω_s and $\omega 2_s$. In their definitions, we in turn use the function $\Omega_s(\phi)$ returning the atoms for evaluating a GRAPPA acceptance function:

$$\begin{aligned}
\omega_s &:= \{asg(r, Y_r), lt(Y_r, Z_r) \mid r \in par(s)\} \ \cup\ \Omega_s(\alpha(s));\\
\omega 2_s &:= \{asg2(r, Y_r), lt(Y_r, Z_r) \mid r \in par(s)\} \ \cup\ \Omega_s(\alpha(s)).
\end{aligned}$$

Proposition 6
For $\sigma \in \{adm, com, prf\}$, it holds for every GRAPPA instance G that $\sigma(G) \cong \mathscr{AS}(\pi_\sigma(G))$.

Proof
(sketch) The proofs are exactly as those of Propositions 1, 2, and 3; they differ only in the parts in which reference is made to the encoding of the evaluation of the acceptance patterns. □

Example 6
Consider the GRAPPA instance G with $S = \{a, b, c, d\}$, $E = \{(b,b), (a,c), (b,c), (b,d)\}$, $L = \{+, -\}$, $\lambda((b,b)) = +$, $\lambda((a,c)) = +$, $\lambda((b,c)) = +$, $\lambda((b,d)) = -$, $\pi(s) = \#_t(+) - \#(+) = 0 \wedge \#(-) = 0$ for every $s \in S$.

The encoding $\pi_{adm}(G)$ is as follows:

```
arg(a). arg(b). arg(c). arg(d).
lt(u,0). lt(u,1). lt(0,0). lt(1,1).

asg(S,0) :- not asg(S,1), not asg(S,u), arg(S).
asg(S,1) :- not asg(S,u), not asg(S,0), arg(S).
asg(S,u) :- not asg(S,0), not asg(S,1), arg(S).


sat(a) :-  Vbp1s2t = 0,Vbp1s2 = (-1)*Vbp1s2t,
           Vbp1s1t = 0, Vbp1s1 = Vbp1s1t + Vbp1s2,
           Vbp2s1t = 0, Vbp2s1 = Vbp2s1t,
           Vbp1 = #sum{1: Vbp1s1 = 0},
           Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=1.
unsat(a) :-  Vbp1s2t = 0,Vbp1s2 = (-1)*Vbp1s2t,
             Vbp1s1t = 0, Vbp1s1 = Vbp1s1t + Vbp1s2,
             Vbp2s1t = 0, Vbp2s1 = Vbp2s1t,
             Vbp1 = #sum{1: Vbp1s1 = 0},
             Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=0.
sat(b) :- asg(b,Y_b),lt(Y_b,Z_b),
          Vbp1s2t = #sum{Z_b,b},Vbp1s2 = (-1)*Vbp1s2t,
          Vbp1s1t = 1, Vbp1s1 = Vbp1s1t + Vbp1s2,
          Vbp2s1t = 0, Vbp2s1 = Vbp2s1t,
          Vbp1 = #sum{1: Vbp1s1 = 0},
          Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=1.
unsat(b) :- asg(b,Y_b),lt(Y_b,Z_b),
            Vbp1s2t = #sum{Z_b,b},Vbp1s2 = (-1)*Vbp1s2t,
            Vbp1s1t = 1, Vbp1s1 = Vbp1s1t + Vbp1s2,
            Vbp2s1t = 0, Vbp2s1 = Vbp2s1t,
            Vbp1 = #sum{1: Vbp1s1 = 0},
            Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=0.
sat(c) :- asg(a,Y_a),lt(Y_a,Z_a),asg(b,Y_b),
          lt(Y_b,Z_b),Vbp1s2t = #sum{Z_a,a;Z_b,b},
          Vbp1s2 = (-1)*Vbp1s2t,Vbp1s1t = 2,
          Vbp1s1 = Vbp1s1t + Vbp1s2, Vbp2s1t = 0,
          Vbp2s1 = Vbp2s1t, Vbp1 = #sum{1: Vbp1s1 = 0},
          Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=1.
unsat(c) :- asg(a,Y_a),lt(Y_a,Z_a),asg(b,Y_b),
            lt(Y_b,Z_b),Vbp1s2t = #sum{Z_a,a;Z_b,b},
            Vbp1s2 = (-1)*Vbp1s2t,Vbp1s1t = 2,
            Vbp1s1 = Vbp1s1t + Vbp1s2, Vbp2s1t = 0,
            Vbp2s1 = Vbp2s1t, Vbp1 = #sum{1: Vbp1s1 = 0},
            Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=0.
sat(d) :- asg(b,Y_b),lt(Y_b,Z_b),
          Vbp1s2t = 0,Vbp1s2 = (-1)*Vbp1s2t,
          Vbp1s1t = 0, Vbp1s1 = Vbp1s1t + Vbp1s2,
          Vbp2s1t = #sum{Z_b,b}, Vbp2s1 = Vbp2s1t,
          Vbp1 = #sum{1: Vbp1s1 = 0},
          Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=1.
unsat(d) :- asg(b,Y_b),lt(Y_b,Z_b),
            Vbp1s2t = 0,Vbp1s2 = (-1)*Vbp1s2t,
            Vbp1s1t = 0, Vbp1s1 = Vbp1s1t + Vbp1s2,
            Vbp2s1t = #sum{Z_b,b}, Vbp2s1 = Vbp2s1t,
            Vbp1 = #sum{1: Vbp1s1 = 0},
            Vbp2 = #sum{1: Vbp2s1 = 0}, Vp=Vbp1&Vbp2, Vp=0.

:- arg(S), asg(S,1), unsat(S).
:- arg(S), asg(S,0), sat(S).
```

◇

5 System and overview of experiments

We have implemented a system which, given an ADF, generates the encodings for the ADF presented in this work. The system, `YADF` (“Y” for “dynamic”), is publicly available (see the link provided in the introduction) and currently (version 0.1.1) supports the admissible, complete, preferred, and stable semantics. It is implemented in `Scala` and can, therefore, be run as a `Java` executable.

The input format for `YADF` is the input format that has become the standard for ADF systems. Each statement x of the input ADF is encoded via the string $s(x)$ (alternatively, for legacy reasons, also $statement(x)$ can be used). The acceptance condition F of x is specified in prefix notation via $ac(x, F)$. For example, the acceptance conditions of the ADF from Example 1 are encoded as follows:

```
s(a).
s(b).
s(c).
ac(a,or(neg(b),b)).
ac(b,b).
ac(c,imp(c,b)).
```

Note the period at the end of each line. Here or, imp, neg stand for $\vee$, $\rightarrow$, $\neg$, respectively. On the other hand and, $c(v)$, and $c(f)$ can be used for $\wedge$, $\top$, and $\bot$.

A typical call of `YADF` (using a `UNIX` command line) looks as follows:

```
java -jar yadf_0.1.1.jar -adm -cred a filename | \
      ./path/to/lpopt | ./path/to/clingo
```

Here, we ask `YADF` for the encoding of credulous reasoning w.r.t. the admissible semantics for the ADF specified in the file specified via $filename$ and the statement a. As hinted at in the introduction to this work, using the rule decomposition tool `lpopt`[2] (Bichler *et al.* 2016a) is recommended for larger ADF instances. We have tested `YADF` using the ASP

[2] https://www.dbai.tuwien.ac.at/research/project/lpopt/

solver clingo (Gebser *et al.* 2018). We provide the complete usage (subject to change in future versions) of YADF:

```
usage: yadf [options] inputfile
with options:
 -h                display this help
 -adm              compute the admissible interpretations
 -com              compute the complete interpretations
 -prf              compute the preferred interpretations
 -stb              compute the stable interpretations
 -cred s           check credulous acceptance of statement s
 -scep s           check sceptical acceptance of statement s
```

A generator for encodings for GRAPPA following the methodology outlined in this work is also available as part of the system GrappaVis[3]. The focus of this system is on providing graphical means of specifying and evaluating GRAPPA instances and it includes means of generating static as well as dynamic encodings to ASP.

We reported on an empirical evaluation of the performance of YADF w.r.t. the main alternative ADF systems available at that time in Brewka *et al.* (2017). The other ADF systems we considered then are the static ASP-based system DIAMOND Ellmauthaler and Strass (2014) (version 0.9) as well as the QBF-based system QADF Diller *et al.* (2014) (version 0.3.2). In these experiments, we found YADF to be competitive for credulous reasoning under the admissible semantics, while outperforming DIAMOND and QADF when carrying out skeptical reasoning under the preferred semantics.

Since our experimental evaluation for AAAI'17, there have been two particularly noteworthy experimental evaluations of ADF systems including our system YADF (Diller *et al.* 2018; Linsbichler *et al.* 2018). All of these evaluations build on the experimental setup of Brewka *et al.* (2017); in particular, they also focus on credulous reasoning for the admissible semantics as well as skeptical reasoning for the preferred semantics. Yet, the experiments consider larger sets of (also larger) ADF instances[4]. Also, the more recent experimental evaluations include the latest version of DIAMOND, goDIAMOND (Strass and Ellmauthaler 2017), as well as a novel system for ADFs based on incremental SAT solving, k++ADF (presented in Linsbichler *et al.* (2018)). We give a brief overview of the setup and results of the mentioned evaluations to then summarize what is currently known about the performance of YADF vs. other existing ADF systems.

As already indicated, the experimental evaluations of Diller *et al.* (2018); Linsbichler *et al.* (2018) build on the setup from Brewka *et al.* (2017). In particular, they make use of the graph-based ADF generator we introduced in Brewka *et al.* (2017)[5]. This generator takes any desired directed graph as input and generates an ADF inheriting the structure of the graph. This means that the edges of the graph become links and the nodes become

[3] https://www.dbai.tuwien.ac.at/proj/adf/grappavis/

[4] The more recent experiments do not consider, on the other hand, ADFs generated via the grid-based generator used in the experiments reported on in Ellmauthaler (2012); Diller *et al.* (2014) and that we also considered in Brewka *et al.* (2017). Note, nevertheless, that the grid-based generator offers less flexibility in generating ADFs than the graph-based generator used in subsequent experiments.

[5] This generator underlies the subsequent version available at https://www.dbai.tuwien.ac.at/proj/grappa/subadfgen/.

statements of the resulting ADF. In the experiments in Diller *et al.* (2018); Linsbichler *et al.* (2018), the generator was only modified to take an undirected rather than directed graph as input (providing more flexibility). A probability controls whether an edge in the input graph will result in a symmetric link in the ADF (in the experiments a probability of 0.5 is used); in case of non-symmetric links, the direction of the link is chosen at random.

For constructing the acceptance conditions of the ADFs, the graph-based generator assigns each of the parents of a statement to one of five different groups (with equal probability in the experiments). This assignment determines 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 the exclusive-or connective ($\oplus$; this, in order to capture the full complexity of ADFs). More precisely, if for a statement s_0, the parents $s_1, \ldots, s_n$ are assigned to the group for attack, the corresponding subformula for these parents in the acceptance condition of s_0 has the form $\neg s_1 \wedge \ldots \wedge \neg s_n$. The subformulas for group-attack, support, group-support, and the exclusive-or group on the other hand have the form $\neg s_1 \vee \ldots \vee \neg s_n$, $s_1 \vee \ldots \vee s_n$, $s_1 \wedge \ldots \wedge s_n$, and $l_1 \oplus \ldots \oplus l_n$, respectively. In the last suformula, l_i $(1 \leq i \leq n)$ is either s_i or $\neg s_i$ with equal probability. Also, for groups to which no parents are assigned, the corresponding subformulas are $\top$ or $\bot$ with identical probability. To generate the final acceptance condition of a statement, the subformulas for the different groups of parents of the statement are connected via $\wedge$ or $\vee$; again, with equal probability.

In Diller *et al.* (2018) (extending the evaluations from Keshavarzi Zafarghandi (2017)), the authors compare the performance of ADF systems on acyclic, being those whose underlying graph is acyclic vs. non-acyclic ADFs (i.e. ADFs whose underlying graph is not guaranteed to be acyclic). The combinations of ADF and back-end systems used are as in Brewka *et al.* (2017), except that now also the system `goDIAMOND` (version 0.6.6) is considered[6]. Thus, `QADF` is version 0.3.2 with the preprocessing tool `bloqqer` 035 (Heule *et al.* 2015) and the QSAT solver `DepQBF` 4.0 (Lonsing and Biere 2010; Lonsing and Egly 2017). `YADF` is version 0.1.0 with the rule decomposition tool `lpopt` version 2.0 and the ASP solver `clingo` 4.4.0 (Gebser *et al.* 2018). Version 0.1.0 of `YADF` is identical to version 0.1.1 except that it does not generate the encodings for the stable semantics. The time-out set for the experiments reported on in Diller *et al.* (2018) is 600 s.

The main difference of the experimental setup used in Diller *et al.* (2018) w.r.t. that of Brewka *et al.* (2017) is that now the benchmark set is generated from Dung AFs interpreted as (undirected) graphs obtained from benchmarks used at the second international competition of argumentation (ICCMA'17) (Gaggl *et al.* 2018). These result from encoding assumption-based argumentation problems into AFs ("ABA") (Lehtonen *et al.* 2017), encoding planning problems as AFs ("Planning") (Cerutti *et al.* 2017), and a data set of AFs generated from traffic networks ("Traffic") (Diller 2017). Specifically, based on preliminary experiments, for the experiments in Diller *et al.* (2018) 100 AFs were selected at random from a subset of AFs having up to 150 arguments in the very

[6] The reason for not using the other more recent versions of `DIAMOND`, versions 3.0.x implemented in `C++` (Ellmauthaler and Strass 2016), is the decrease in performance to previous versions of `DIAMOND` documented in Strass and Ellmauthaler (2017).

dense AFs in the “ABA” data set, and 100 AFs at random from a subset of AFs having up to 300 arguments in each of the “Planning” and “Traffic” benchmarks. From the resulting 300 AFs interpreted as undirected graphs, 300 acyclic and 300 non-acyclic ADFs were generated using the graph-based ADF generator.

In the study reported on in Linsbichler *et al.* (2018), the authors generate reasoning problems from the same set of ADFs used in Diller *et al.* (2018)[7] but also consider the new ADF system `k++ADF` they implement. The authors also make use of novel versions for the back-end systems w.r.t. previous experiments. Thus, for `goDIAMOND` version 0.6.6 is still used but now with `clingo` 5.2.1. For `QADF` version 0.3.2[8] with `bloqqer` 037 and `DepQBF` 6.03 is considered. Finally, for `YADF` version 0.1.0 is taken in account, but now with `lpopt` 2.2 and `clingo` 5.2.1. The version of `k++ADF` was version 2018-07-06; the SAT solver used is `MiniSAT` (Eén and Sörensson 2003) version 2.2.0. The variation of back-end systems used in Linsbichler *et al.* (2018) w.r.t. previous experiments leads to some variation in the results obtained, in particular for `YADF`[9]. For the experiments reported on in Linsbichler *et al.* (2018), the time-out is also larger than for previous experiments: 1800 s.

We summarize the results obtained in the studies from Diller *et al.* (2018) and Linsbichler *et al.* (2018) in bullet-point fashion (we of course refer to the alluded works for details). As already hinted at, these studies build on (and largely confirm the results obtained in) previous studies, mainly that of Brewka *et al.* (2017). When making reference to Diller *et al.* (2018), we refer to the studies on the (possibly) non-acyclic ADF instances since these can be compared to the study of Linsbichler *et al.* (2018) which does not consider acyclic ADFs (nevertheless, some comments on the performance of especially `YADF` on the acyclic instances follow). In the summary, when mentioning results for a particular solver, we use the best of the results for that solver obtained in the different studies. Also, as a reminder to the reader, in this discussion when we refer to the solvers `k++ADF`, `goDIAMOND`, `YADF`, and `QADF`, except if stated otherwise, these are versions 2018-07-06, 0.6.6, 0.1.0, and 0.3.2, respectively. In particular, we again note that `YADF` version 0.1.0 is identical to the newer version detailed in this work (0.1.1) for the encodings considered in the experiments we allude to[10].

- For credulous reasoning w.r.t. the admissible semantics each of `k++ADF` (when using the link-information-sensitive variant `ADM-K-BIP`, rather than `ADM-2`), `goDIAMOND`,

[7] The exact encodings used differ nevertheless because the choice of the statements whose acceptability is checked may differ.

[8] The paper mistakenly reports use of version 2.9.3 which does not exist (`QADF` version 0.3.2 is implemented in version 2.9.3 of the programming language `Scala`).

[9] Especially for the admissible semantics. For example, `YADF` has 47 and 21 time-outs on the “Traffic” and “Planning” data sets, while in the study reported on in Diller *et al.* (2018) the time-outs were 2 and 0, respectively. We have determined (via tests carried out using the different versions of `lpopt` on instances for which there were time-outs in the study from Linsbichler *et al.* (2018)) the cause of this to be the use of `lpopt` version 2.2, which seems to have problems in generating the rule decompositions of some of the encodings obtained via `YADF` in a timely manner; while versions previous to 2.2. (we also tried 2.0 and 2.1) don’t have this issue.

[10] We note also that there is meanwhile a newer version of `QADF` (see `https://www.dbai.tuwien.ac.at/proj/adf/qadf/`) (version 0.4.0) which includes link-information-sensitive encodings (see Section 3.1.3 of Diller (2019)), but is otherwise (i.e. when not using the link-information-sensitive variants of the encodings) identical to version 0.3.2 considered in the experiments reported on in Diller *et al.* (2018); Linsbichler *et al.* (2018).

and `YADF` (making use of `lpopt` version 2.0 as in the experiments from Brewka *et al.* (2017); Diller *et al.* (2018)) have rather acceptable performance on the "Traffic" and "Planning" data sets. (The same holds for `DIAMOND` version 0.9 on a small set of ADFs generated from metro-networks (Brewka *et al.* 2017; Keshavarzi Zafarghandi 2017).) The order in which we mention the solvers reflects the improvement in performance, with `k++ADF` being the clear "winner". The system `QADF` (even in the more advantageous configuration with `bloqqer` version 035 and `DepQBF` version 4.0 from Brewka *et al.* (2017); Diller *et al.* (2018)) on the other hand already has quite a few time-outs on the "Traffic" and "Planning" instances. We remind the reader that the "Traffic" and "Planning" data sets include ADFs with 10 to 300 statements resulting from the underlying graphs obtained from representing transportation networks as AFs (Diller 2017) and encoding planning problems into AFs (Cerutti *et al.* 2017), respectively.

- Thus, `k++ADF` (in the link-information-sensitive variant `ADM-K-BIP`) had 0 time-outs (1800 s) and 0.05 s mean running time, `goDIAMOND` 0 time-outs and 6.42 s mean running time in the experiments reported on in Linsbichler *et al.* (2018) on the "Traffic" data set. `YADF` had two time-outs (600 s) and 5.68 s mean running time (disregarding time-outs) in the experiments reported on in Diller *et al.* (2018). The system `k++ADF` (implementing `ADM-K-BIP`) had 0 time-outs and 0.14 s mean running time, `goDIAMOND` 0 time-outs and 6.72 s mean running time in the experiments reported on in Linsbichler *et al.* (2018) on the "Planning" data set. `YADF` had 0 time-outs and 13.20 s mean running time in the experiments reported on in Diller *et al.* (2018). `QADF` had 25 time-outs and 2.15 s mean running time on the "Traffic" instances and 59 time-outs and 14.63 s mean running time on the "Planning" instances (Diller *et al.* 2018).

- For credulous reasoning w.r.t. the admissible semantics, but now on the "ABA" data set; here all ADF systems have some time-outs, yet again the results for `k++ADF` are the most promising. We remind the reader that the "ABA" data set consists in 100 very dense ADFs having between 10 to 150 statements resulting from the underlying graphs of encoding problems for assumption-based AFs to AF reasoning problems (Lehtonen *et al.* 2017).

  - Thus, `k++ADF` (now in the `ADM-2` variant) had 12 time-outs (1800 s) and mean running time of 16.12 s in the experiments of Linsbichler *et al.* (2018). Interestingly, for the "ABA" data set `QADF` (with `bloqqer` version 035 and `DepQBF` version 4.0) gets "second-place" having 30 time-outs (600 s) and 8.15 s mean running time in the experiments from Diller *et al.* (2018). The system `goDIAMOND` has 52 time-outs and `YADF` 56 time-outs in the experiments from Diller *et al.* (2018).

- For the preferred semantics, the performance of `YADF` and `QADF` (as well as version 0.9 of `DIAMOND` on ADFs resulting from traffic networks (Brewka *et al.* 2017; Keshavarzi Zafarghandi 2017)) worsens considerably on the "Traffic" and "Planning" problems (w.r.t. results for the admissible semantics), while `k++ADF` (particularly in the link-information-sensitive variant `PRF-K-BIB-OPT`, but not in the variant `PRF-3`) and `goDIAMOND` have much better performance.

— Thus, `YADF` (`lpopt` version 2.0) has 36 time-outs on the "Traffic" instances and 71 time-outs on the "Planning" instances in the study from Diller *et al.* (2018). `QADF` has 80 and 100 time-outs on the "Traffic" and "Planning" benchmarks (again, study from Diller *et al.* (2018)). On the other hand, `goDIAMOND` has 0 time-outs on both data sets with 28.42 and 17.52 s mean running times on the "Traffic" and "Planning" instances, respectively (Linsbichler *et al.* 2018). The system `k++ADF` (in the `PRF-K-BIB-OPT` variant) manages having only one time-out on the "Traffic" instances and three on the "Planning" instances with 17.18 and 11.14 s mean running time, respectively (Linsbichler *et al.* 2018).

- All ADF systems also have time-outs when solving skeptical acceptance w.r.t the preferred semantics on the "ABA" data set, with `k++ADF` in the `PRF-K-BIB-OPT` variant having the least (16 time-outs (Linsbichler *et al.* 2018)) and `QADF` the most (81 time-outs (Diller *et al.* 2018)).

— Thus, `k++ADF` in the `PRF-K-BIB-OPT` variant has 16 time-outs and 25.90 s mean running time (Linsbichler *et al.* 2018), while `QADF` has 81 time-outs (with 32.73 s running time on the remaining instances) (Diller *et al.* 2018). `YADF` has 57 time-outs and 39.46 s mean running time, while `goDIAMOND` has 52 time-outs and 27.67 s mean running time (Diller *et al.* 2018).

To conclude, while our experiments from Brewka *et al.* (2017) (on the instances obtained via the grid-based generator first used in Ellmauthaler (2012) and ADFs constructed from a limited set of traffic networks also used in subsequent experiments) suggested `YADF` to be the best performing of the then considered systems (including `DIAMOND` version 0.9 and `QADF` 0.3.2), this picture has changed with subsequent experiments (Diller *et al.* 2018; Linsbichler *et al.* 2018) involving the new systems `goDIAMOND` and `k++ADF` as well as more (and larger) data sets. In particular, the clearly overall best performing approach for ADF systems seems to be, at current moment, the incremental SAT-based approach implemented in the system `k++ADF` (despite the fact that even this system still has quite a few time-outs for the preferred semantics on the ABA data set). But even just considering ASP-based systems, while competitive for the admissible semantics, `YADF` is clearly behind in performance w.r.t. `goDIAMOND` for the preferred semantics on the "Traffic" and "Planning" data sets.

Some reason for nevertheless sticking to the dynamic ASP-based approach presented in this work (vs. static encodings) is provided by the results on the performance of `YADF` on the ABA data set (in the configurations from Diller *et al.* (2018)). Here, the constraint built into `goDIAMOND` of not supporting ADFs with statements having more than 31 parents is reflected in the constant number of time-outs (52; and similar mean running times: ca. 21 s for admissible, 27 s for preferred) on all reasoning tasks (admissible and preferred) and for acyclic as well as non-acyclic instances (the latter in the experiments from Diller *et al.* (2018)). Indeed, the constraint built into `goDIAMOND` of not supporting ADFs with statements having more than 31 parents is due to the fact that this system (as previous versions of DIAMOND) needs to convert acceptance conditions of ADFs into a Boolean function representation (with a potential exponential explosion), which our dynamic encoding strategy allows to circumvent. Thus, while YADF still has many time-outs (in fact, a few more than goDIAMOND) there is some (slight) improvement on the acyclic instances: 54 time-outs with 7.38 s mean running time vs. 56 time-outs with 31.39 s mean running time for the admissible semantics and 54 time-outs with 16.77 s mean running time vs. 57 time-outs with 39.46 s mean running time for the preferred semantics. These results suggest room for improvement as well as, in accordance with the theoretical considerations motivating our dynamic ASP-based approach, a potential niche for the use of (a further optimized) YADF vs. for example, goDIAMOND.

6 Discussion

In this work, we 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)); see also Bichler *et al.* (2016b) who advocated this idea as a programming technique in the world of ASP.

We implemented our approach for ADF and GRAPPA. Our experiments show the potential of our approach. Still, the number of statements we can handle is somewhat limited. Our encodings thus might also be interesting benchmarks for ASP competitions. Nonetheless, there are certain aspects which have to be considered in future versions of our system. In fact, a crucial aspect for the programming technique due to Bichler *et al.* (2016b) is the possible decomposition of long rules, since grounders have severe problem with such rules. As reported in a recent paper (Bichler *et al.* 2018) that also employs this technique, the actual design of long rules can strongly influence the runtime. We shall thus analyze our encodings in the light of the findings in Bichler *et al.* (2018) in order to allow for better decomposition whenever possible.

Beyond boosting performance of our system, future work is to apply our approach to alternative ADF semantics (Polberg 2014) as well as more recent generalizations of ADFs and GRAPPA such as weighted ADFs (Brewka *et al.* 2018) and multi-valued GRAPPA (Brewka *et al.* 2019); for dealing with possibly infinitely many values in this context, recent advances in ASP (Janhunen *et al.* 2017) might prove useful. Also the application of other recent ASP techniques (e.g. Bogaerts *et al.* (2016); Redl (2017)) that allow for circumventing the problem of an exponential blowup when problems beyond the second level of the polynomial are treated is of interest.

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.

Atkinson, K. and Bench-Capon, T. J. M. 2018. Relating the ANGELIC methodology and ASPIC+. In *Proc. COMMA*. Frontiers in Artificial Intelligence and Applications, vol. 305. IOS Press, 109–116.

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. 2016a. lpopt: A rule optimization tool for answer set programming. In *LOPSTR*. Lecture Notes in Computer Science, vol. 10184. Springer, 114–130.

Bichler, M., Morak, M. and Woltran, S. 2016b. The power of non-ground rules in answer set programming. *TPLP 16,* 5–6, 552–569.

Bichler, M., Morak, M. and Woltran, S. 2018. Single-shot epistemic logic program solving. In *Proc. IJCAI.* ijcai.org, 1714–1720.

Bogaerts, B., Janhunen, T. and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP 16,* 5–6, 570–586.

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

Brewka, G., Diller, M., Heissenberger, G., Linsbichler, T. and Woltran, S. 2017. Solving advanced argumentation problems with answer-set programming. In *Proc. AAAI.* AAAI Press, 1077–1083.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Brewka, G., Ellmauthaler, S., Strass, H., Wallner, J. P. and Woltran, S. 2018. Abstract Dialectical Frameworks: An Overview. In *Handbook of Formal Argumentation*, P. Baroni, D. Gabbay, M. Giacomin, and L. van der Torre, Eds. College Publications, Chapter 5, 236–286.

Brewka, G., Polberg, S. and Woltran, S. 2014. Generalizations of dung frameworks and their role in formal argumentation. *IEEE Intell. Syst. 29,* 1, 30–38.

Brewka, G., Pührer, J. and Woltran, S. 2019. Multi-valued GRAPPA. In *Proc. JELIA.* Lecture Notes in Computer Science, vol. 11468. Springer, 85–101.

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

Brewka, G., Strass, H., Wallner, J. P. and Woltran, S. 2018. Weighted abstract dialectical frameworks. In *Proc. AAAI.* AAAI Press, 1779–1786.

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

Brewka, G. and Woltran, S. 2014. GRAPPA: A semantical framework for graph-based argument processing. In *Proc. ECAI.* Frontiers in Artificial Intelligence and Applications, vol. 263. IOS Press, 153–158.

Cabrio, E. and Villata, S. 2016. Abstract dialectical frameworks for text exploration. In *Proc. ICAART'16.* SciTePress, 85–95.

Cartwright, D. and Atkinson, K. 2009. Using computational argumentation to support E-participation. *IEEE Intell. Syst. 24,* 5, 42–52.

Cerutti, F., Giacomin, M. and Vallati, M. 2017. Exploiting Planning Problems for Generating Challenging Abstract Argumentation Frameworks. URL: http://argumentationcompetition.org/2017/Planning2AF.pdf

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

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. 2017. Traffic Networks Become Argumentation Frameworks. URL: http://argumentationcompetition.org/2017/Traffic.pdf

Diller, M. 2019. Realising argumentation using answer set programming and quantified boolean formulas. Ph.D. thesis, TU Wien.

Diller, M., Keshavarzi Zafarghandi, A., Linsbichler, T. and Woltran, S. 2018. Investigating subclasses of abstract dialectical frameworks. In *Proc. COMMA*. Frontiers in Artificial Intelligence and Applications, vol. 305. IOS Press, 61–72.

Diller, M., Wallner, J. P. and Woltran, S. 2014. Reasoning in abstract dialectical frameworks using quantified boolean formulas. In *Proc. COMMA*. Frontiers in Artificial Intelligence and Applications, vol. 266. IOS Press, 241–252.

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.

Eén, N. and Sörensson, N. 2003. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science, vol. 2919. Springer, 502–518.

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. 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., Gottlob, G. and Mannila, H. 1997. Disjunctive datalog. *ACM Trans. Database Syst. 22,* 3, 364–418.

Ellmauthaler, S. 2012. Abstract Dialectical Frameworks: Properties, Complexity, and Implementation. M.S. thesis, TU Wien.

Ellmauthaler, S. and Strass, H. 2014. The DIAMOND system for computing with abstract dialectical frameworks. In *Proc. COMMA*. Frontiers in Artificial Intelligence and Applications, vol. 266. IOS Press, 233–240.

Ellmauthaler, S. and Strass, H. 2016. DIAMOND 3.0 – A native C++ implementation of DIAMOND. In *Proc. COMMA*. Frontiers in Artificial Intelligence and Applications, vol. 287. IOS Press, 471–472.

Gaggl, S., Linsbichler, T., Maratea, M. and Woltran, S. 2018. Summary report of the second international competition on computational models of argumentation. *AI Mag. 39*, 77–79.

Gaggl, S. A. and Strass, H. 2014. Decomposing abstract dialectical frameworks. In *Proc. COMMA*. Frontiers in Artificial Intelligence and Applications, vol. 266. IOS Press, 281–292.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T. and Thiele, S. 2015. Potassco User Guide. URL: https://sourceforge.net/projects/potassco/files/guide/. [Accessed on November 22, 2016].

Gebser, M., Kaminski, R., Kaufmann, B., Lühne, P., Obermeier, P., Ostrowski, M., Romero, J., Schaub, T., Schellhorn, S. and Wanko, P. 2018. The potsdam answer set solving collection 5.0. *KI 32,* 2–3, 181–182.

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

Heule, M., Järvisalo, M., Lonsing, F., Seidl, M. and Biere, A. 2015. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res. 53*, 127–168.

Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P. and Schaub, T. 2017. Clingo goes linear constraints over reals and integers. *TPLP 17,* 5–6, 872–888.

Janhunen, T. and Niemelä, I. 2016. The answer set programming paradigm. *AI Magazine 37,* 3, 13–24.

Keshavarzi Zafarghandi, A. 2017. Investigating Subclasses of Abstract Dialectical Frameworks. M.S. thesis, Vienna University of Technology, Institute of Information Systems.

Lehtonen, T., Wallner, J. P. and Järvisalo, M. 2017. Assumption-Based Argumentation Translated to Argumentation Frameworks. URL: http://argumentationcompetition.org/2017/ABA2AF.pdf

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.

Linsbichler, T., Maratea, M., Niskanen, A., Wallner, J. P. and Woltran, S. 2018. Novel algorithms for abstract dialectical frameworks based on complexity analysis of subclasses and SAT solving. In *Proc. IJCAI.* ijcai.org, 1905–1911.

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

Lonsing, F. and Egly, U. 2017. Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In *Proc. CADE.* Lecture Notes in Computer Science, vol. 10395. Springer, 371–384.

McBurney, P., Parsons, S. and Rahwan, I., Eds. 2012. *Proc. ArgMAS.* Lecture Notes in Computer Science, vol. 7543. Springer.

Neugebauer, D. 2018. DABASCO: Generating AF, ADF, and ASPIC$^+$ instances from real-world discussions. In *Proc. COMMA.* Frontiers in Artificial Intelligence and Applications, vol. 305. IOS Press, 469–470.

Polberg, S. 2014. Extension-based semantics of abstract dialectical frameworks. In *Proc. STAIRS.* Frontiers in Artificial Intelligence and Applications, vol. 264. IOS Press, 240–249.

Redl, C. 2017. Answer set programs with queries over subprograms. In *Proc. LPNMR.* Lecture Notes in Computer Science, vol. 10377. Springer, 160–175.

Strass, H. and Ellmauthaler, S. 2017. goDIAMOND 0.6.6 – ICCMA 2017 System Description. URL: http://argumentationcompetition.org/2017/goDIAMOND.pdf.

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