International Journal of Approximate Reasoning 201 (2027) 109857



Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

journal homepage: www.elsevier.com/locate/ijar



# Abstract Dialectical Frameworks with domain assignments

Dimitris Danis [a,b], Giorgos Flouris [b], Antonis Bikakis [c,*], Theodore Patkos [b], Nick Bassiliades [d]

[a] Computer Science Department, University of Crete, Voutes Campus, Vassilika Vouton, Heraklion, GR-70013, Greece
[b] Institute of Computer Science, Foundation for Research and Technology - Hellas, Vassilika Vouton, P.O. Box 1385, Heraklion, GR-70013, Greece
[c] Department of Information Studies, University College London, Gower Street, London, WC1E 6BT, United Kingdom
[d] School of Informatics, Aristotle University of Thessaloniki, University Campus, Thessaloniki, GR-54124, Greece

ARTICLE INFO

*Keywords:*
Computational argumentation
Abstract argumentation frameworks
Abstract Dialectical Frameworks
Domain assignments in argumentation frameworks
ADF
AAFD
ADFD

ABSTRACT

We propose a novel argumentation formalism, called Abstract Dialectical Frameworks with Domain Assignments (ADFDs), which is the fusion of existing related formalisms. ADFDs allow the association of each argument with a domain of application, in the spirit of Abstract Argumentation Frameworks with Domain Assignments (AAFDs), and with an arbitrarily complex acceptance condition, in the spirit of Abstract Dialectical Frameworks (ADFs). The resulting framework combines the best of both worlds, enabling the elegant and simple modelling of complex argumentative scenarios. We define ADFDs, develop their semantics, study their properties and expressive power, and present an implementation along with an experimental evaluation.

## 1. Introduction

Computational argumentation deals with the representational and reasoning aspects that determine how arguments and argumentative processes can be represented in a computer system, and how the outcome of an argumentative process can be automatically determined [1]. The current paper falls in the sub-area of abstract argumentation [2], the objective of which is to determine the acceptability status of arguments in the context of a set of arguments on a topic.

The most well-known formalism for abstract argumentation is the Abstract Argumentation Frameworks (AAFs) [2]. An AAF is a directed graph, in which nodes denote unique atomic arguments and edges represent attacks between arguments. Based on this simple model, Dung [2] defined different conditions for determining the acceptance of arguments in such a graph, based on the idea that a set of arguments is accepted if its members do not attack each other, and, as a whole, shield themselves from external attacks. The exact formal definition given to the term “shield” gives rise to different semantics (admissible, complete, grounded, etc.).

Another formalism for reasoning with arguments is provided by Abstract Dialectical Frameworks (ADFs), originally proposed by Brewka and Woltran [3], and later refined in a series of papers, including [4–6]. ADFs can also be modeled as graphs; however, in ADFs, there is no “attack relation” per se, but each argument is associated with an acceptance condition that determines the exact conditions under which the argument is accepted. These conditions may, in general, require the acceptance (or non-acceptance) of different combinations of associated arguments; as a result, the “attack” relation of AAFs is just one of the conditions that ADFs can model. In this context, each edge in the directed graph of an ADF does not necessarily represent an attack relation, but a dependence of the acceptability of one argument on another. Defining the acceptance of arguments in ADFs is analogous to AAFs, and similar (but more complex) versions of the various AAF semantics have been developed [6].

* Corresponding author.
*E-mail addresses:* jimbo_elesar@yahoo.gr (D. Danis), fgeo@ics.forth.gr (G. Flouris), a.bikakis@ucl.ac.uk (A. Bikakis), patkos@ics.forth.gr (T. Patkos), nbassili@csd.auth.gr (N. Bassiliades).

https://doi.org/10.1016/j.ijar.2026.109857
Received 25 September 2026; Received in revised form 30 October 2026; Accepted 31 October 2026
Available online 4 November 2026
0888-613X/© 2026 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

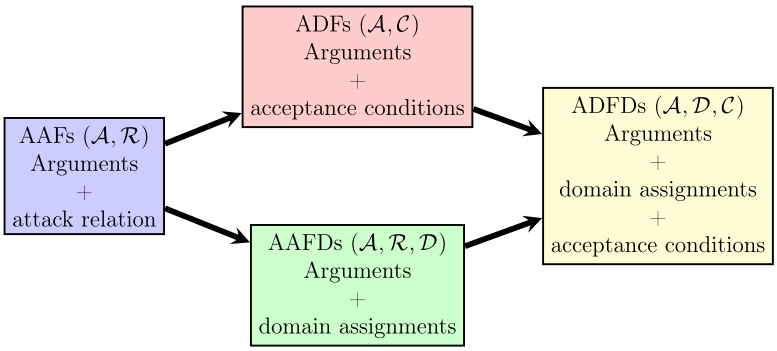



**Fig. 1.** The origins of ADFDs.

Existing abstract argumentation frameworks determine whether an argument is accepted within a given structure, treating arguments as either accepted or rejected as a whole. In many policy-oriented and regulatory settings, however, arguments are not evaluated uniformly, but relative to particular individuals, objects, or situations. An argument may therefore need to be accepted with respect to certain individuals but rejected for others.

As a simple example, consider a traffic rule stating that "a vehicle that violates a red traffic light is subject to a fine" (argument A). Traffic regulations often also specify exemptions, for example that "vehicles operating as ambulances while transporting a patient in critical condition are exempt from such fines" (argument B). Such a scenario can in principle be encoded in standard AAFs or ADFs using a distinct set of arguments for each vehicle under consideration. However, this requires replicating the argument graph for every individual vehicle, including in each instance the arguments representing the rules or exceptions that apply to the particular case. An alternative is to model all policy rules and exemptions within a single framework, such that each rule or exemption is represented as an argument, and the evaluation of the policy distinguishes between different cases (in our example, vehicles), allowing the same argument to be accepted for some cases and rejected for others.

To accommodate this feature, a refinement of AAFs, called Abstract Argumentation Frameworks with Domain Assignments (AAFDs), was proposed by Vassiliades et al. [7]. This framework allows us to (de)activate different arguments and attacks, depending on the specific entity or context under consideration, using a single abstract framework, without having to create a different argument graph for each such entity or context as usually required by existing models. This is achieved by associating arguments with a *domain of application* consisting of the elements (in a given universe) for which each argument is applicable, thereby allowing us to determine when an argument is "active". In the above example, argument A would be associated to all vehicles, whereas argument B (that attacks A) would be associated with ambulances only. This way, the attack from B to A would only materialize for ambulances, but not for other vehicles. Thus, in AAFDs, attacks among arguments serve the purpose of reducing the applicability of arguments, i.e., determining the elements of the domain of application of an argument for which we accept it, and the elements for which we do not. This essentially means that, in general, arguments are *partly accepted*, i.e., accepted for only some of the elements of their original domain. It is easy to see that AAFs can be modeled as AAFDs with a singleton universe.

Based on the above frameworks, we propose here a new abstract argumentation formalism, called Abstract Dialectical Frameworks with Domain Assignments (ADFDs), which combines core features of ADFs and AAFDs (as illustrated in Fig. 1). ADFDs retain the use of acceptance conditions as in ADFs, while incorporating domain assignments as in AAFDs in order to allow arguments to be evaluated relative to specific entities. This requires extending the standard ADF architecture: while classical ADF acceptance conditions are applied to arguments and depend only on the acceptability of associated arguments, in ADFDs acceptance conditions are applied to argument-entity pairs and depend on the acceptability of other arguments with respect to the same or related entities, as well as on the properties and relations of such entities. The framework, therefore, supports situations in which the acceptance of an argument for one entity may influence the acceptance of another argument for a different but related entity. While such situations can also be modeled in other frameworks (AAFs, ADFs or AAFDs), ADFDs enable a more intuitive and compact representation. They also enhance the *elaboration tolerance* [8] of the formalism, meaning that the effort required to add new information is proportional to the complexity of that information, and that small modifications do not cause major changes in the representation.

In the following, we first demonstrate informally the main features of ADFDs (Section 2). In Section 3, we discuss the related work, and in Section 4 we present the related argumentation formalisms (AAFs, AAFDs and ADFs), as well as the Extended Logic Programs (ELPs) that will be used in the following. We formally define ADFDs and their semantics in Section 5, and prove various relevant results in Section 6. In Section 7, we describe how we implemented ADFDs, and, in particular, verification algorithms for their semantics, and present experiments showing how different characteristics of the input affect the runtime of our verification algorithms. Finally, we discuss future work and conclude (Section 8). Two appendices are included, to provide useful information that has been omitted from the main part of the paper. In the first Appendix A, we provide proofs for all our results, whereas the second Appendix B presents the pseudocode for the algorithms described in Section 7.

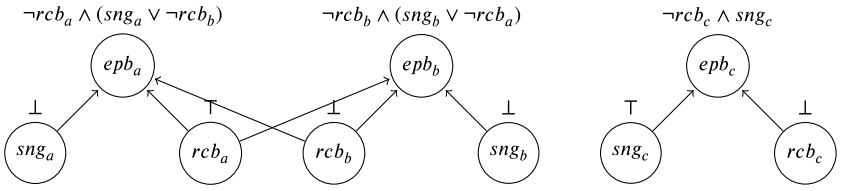



**Fig. 2.** An ADF representing the circumstances of three parents, as for their eligibility for parental benefit according to the policy of the running example. Arguments with subscript $a$ refer to Alice, those with subscript $b$ refer to Bob and those with subscript $c$ refer to Cindy. Alice and Bob are spouses and Alice already receives a cash benefit. Cindy is single and does not receive a benefit. The logical expression "above" each argument corresponds to the ADF's acceptance condition for the respective argument ($\top$ is always true, $\bot$ is always false).

## 2. Demonstrating the main features of ADFDs

To demonstrate the main features of ADFDs, consider the following policy for parental benefits, which will serve as a running example throughout the paper:

> **Running example.** A parent is eligible for parental benefit if they do not receive another cash benefit, and they are single or their spouse does not receive a cash benefit.

Let us note certain important features of this example. From the perspective of whether a certain parent is eligible for the parental benefit, we note that there are certain arguments that block acceptance (such as receiving another cash benefit), while others allow it (such as being single). AAFs and AAFDs, which focus only on attacks among arguments, cannot model such situations, except through the use of additional, artificial arguments, e.g., see [9,10]. On the other hand, ADFs can model any relationship that can be expressed as a propositional formula. However, ADFs cannot be used directly to model this policy because they are designed to model each situation or context individually. This is not convenient here for two reasons.

First, the above arguments are applicable to all parents, while ADFs (or AAFs) do not provide a way to explicitly express that an argument refers to a set of entities, e.g., all parents. Creating an argument for all parents in these models would necessarily treat all parents in the same way, disallowing exceptions or special treatment for certain parents or groups of parents. Therefore, to represent this policy as an ADF while allowing exceptions, we would have to make the assumption that each argument refers to a specific parent, i.e., we have to create one argument for the eligibility of each parent; this can quickly become cumbersome if we want to model a large number of parents.

Second, the acceptance of an argument for a certain parent in the above scenario may depend on the acceptance of some argument for another parent, i.e., their spouse. This introduces the requirement that acceptance of an argument for an entity depends on the acceptance of some (possibly different) argument for another (possibly related) entity. To support this in ADFs, we, again, have to replicate each argument for each different parent that it refers to; but this may lead to an enormous number of arguments for large domains.

These shortcomings are demonstrated in Fig. 2, which depicts an ADF representing the circumstances of three different parents: Alice and Bob, who are spouses, with Alice already receiving a cash benefit; and Cindy, who is single and does not receive a cash benefit. We have used three types of arguments: $epb$, representing that a parent is eligible for parental benefit; $sng$, representing that a parent is single; and $rcb$, representing that a parent receives a cash benefit. The subscript in each argument indicates the person that the argument refers to; for example, $epb_a$ is an argument for Alice's eligibility for parental benefit. This leads to a total of 9 arguments (3 arguments for each person). The propositional formula "above" each argument represents its acceptance condition.

Using ADFDs, we can express this policy more succinctly. We only need three arguments ($epb$, $sng$, $rcb$), each of which is equipped with a domain of application representing the entities to which the argument applies; in particular, the set of all parents for $epb$, and the set of all citizens for $sng$ and $rcb$. Formally, this domain of application can be expressed with a logical formula, e.g., $\delta_{epb}(X) \leftarrow parent(X)$ for the case of $epb$. An argument-entity pair represents that the argument is applicable to the specific entity, for example, $sng(cindy)$ is the argument stating that Cindy is single (note how this corresponds to the argument $sng_c$ in Fig. 2). The acceptance conditions of these arguments represent the general conditions under which an argument should be accepted, as expressed by the policy in the running example. For $epb$ in particular, its acceptance condition is expressed using the following two rules (where the variables $X, Y$ are implicitly universally quantified):

$$\gamma_{epb}(X) \leftarrow acc_{sng}(X), \neg acc_{rcb}(X) \tag{1}$$

$$\gamma_{epb}(X) \leftarrow spouse(X,Y), \neg acc_{rcb}(X), \neg acc_{rcb}(Y) \tag{2}$$

where $\gamma_{epb}(X)$ denotes that "$epb$ should be accepted for entity $X$", $acc_{sng}(X)$ that "$sng$ has been accepted for $X$", $acc_{rcb}(X)$ that "$rcb$ has been accepted for $X$" and $spouse(X,Y)$ that "$X$ and $Y$ are spouses". The above ADFD is depicted as a graph in Fig. 3, where $\gamma_{epb}(X)$ (the acceptance condition of $epb$) refers to the two rules presented above.

We can identify the individual entities for which an argument should be accepted by taking into account the domains and acceptance conditions of all other related (through the acceptance conditions) arguments. Our acceptance decisions are recorded in the *interpretation* of the ADFD, which essentially consists of logical expressions of the form $acc_a(u)$ (or $\neg acc_a(u)$) stating that argument $a$ is

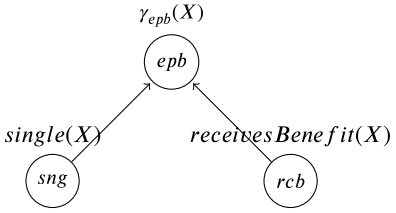



**Fig. 3.** An ADFD representing the policy of the running example. $\gamma_{epb}(X)$ (the acceptance condition of $epb$) refers to the two rules in Eqs. (1) and (2); $single(X)$ and $receivesBenefit(X)$ are the acceptance conditions of $sng$ and $rcb$, respectively. The domain of $epb$ is the set of all parents and the domain of $sng$ and $rcb$ is the set of all citizens (see also Example 1).

(not) accepted for element $u$. Interpretations can be classified as admissible, complete etc, in the usual sense (see Section 5). Therefore, to determine whether a specific parent, e.g., Alice, is eligible for parental benefit, we only need to check whether $acc_{epb}(alice)$ is in the framework's interpretation under the desired semantics, and credulous/skeptical acceptance can be defined in the standard manner.

The above modeling is close to the intuition behind the policy and comes naturally. It is also more succinct compared to a model based on other argumentation formalisms, where we would have to create multiple arguments to model all parents (e.g., see Fig. 2 for the case of ADFs) or artificial arguments to model complex relationships (when using AAFs or AAFDs) [7,9,10]. We do not claim that these situations are impossible to represent using ADFs, or even using simple AAFs; instead, we claim that such a representation would require a more complex model, employing a large number of arguments, some of which artificial. This reduces the compactness and intuitiveness of the model, increases the effort required to maintain it and adapt it in case of changes in the underlying domain, and, ultimately, reduces its *elaboration tolerance*.

It is also interesting to consider the positioning of the ADFD framework in the standard argumentation pipeline. In most such scenarios, modeling a domain starts from a KB, based on which arguments are generated using some structured argumentation model like, e.g., ASPIC$^+$ [11]. Then, these arguments are evaluated using some abstract argumentation model to determine their acceptability. In that pipeline, the argument generation is based on the association of basic facts (e.g., that Alice receives a cash benefit) with rules (e.g., that a parent is eligible for parental benefit if they do not receive another cash benefit and either are single or their spouse does not receive a cash benefit) to determine which arguments apply to any specific individual. Using ADFDs, the same process is followed: arguments are associated to basic facts while rules are encoded in the acceptance conditions. However, this is carried out within an abstract framework, which directly provides the acceptable arguments without requiring a subsequent translation from a structured to an abstract model. Therefore, ADFDs allow "lifting" the domain modeling from the structured to the abstract level, providing an alternative way to perform the entire pipeline (including both the argument generation and the argument evaluation) within a single model.

## 3. Related work

Formal argumentation employs a simple model to capture various types of non-monotonic and commonsense reasoning that cannot be easily captured by standard logics, such as reasoning about trust or preferences. Since Dung's seminal paper on AAFs [2], various extensions or variations have been proposed, including ADFs [3], which generalize AAFs by replacing the attack relation with arbitrary acceptance conditions. This makes ADFs significantly more capable in terms of concisely modeling complex dependencies among arguments. However, ADFs evaluate arguments globally: each argument is either accepted or rejected within a given interpretation. As a result, they cannot directly represent context-dependent or entity-specific acceptability without replicating arguments for each entity. In contrast, ADFDs allow a single argument to be evaluated differently across entities via domain assignments, thus avoiding the replication of argument graphs, and improving the succinctness of the representation.

The most relevant approach to enhance the representational capabilities of ADFs is a framework called GRAPPA [12]. The main aim of GRAPPA is to enable counting or aggregating over a set of arguments in the acceptance conditions. Specifically, in GRAPPA, each link in the graph has an associated label (for example, attack, support, etc.), and the acceptance conditions may contain aggregates over sets of labels, enabling, for example, conditions of the form "accept this node if at least k of its incoming links carry label support". While GRAPPA enhances expressiveness in terms of quantitative reasoning over argument relations, it remains orthogonal to our work: aggregation operates over links, whereas ADFDs operate over domain entities. The development of an extension of GRAPPA with argument domains would be an interesting direction for future work.

In Temporal Argumentation Frameworks (TAFs) [13,14], time is considered at the object language level, and the acceptability of an argument is related to a temporal availability, i.e., an argument may be accepted at some time points but not in others. This is conceptually similar to the partial acceptance supported by ADFDs. However, TAFs are specifically tailored to temporal availability, whereas ADFDs generalize this idea to arbitrary domains (e.g., individuals, objects, contexts). Moreover, TAFs do not provide the same level of expressiveness in modeling complex acceptance conditions across entities.

Frameworks that enrich arguments with additional structure, such as topic-based argumentation [15] or claim-augmented frameworks [16], address different concerns. Topic-based argumentation defines argument acceptability in terms of semantic proximity between arguments, but still treats arguments as globally accepted or rejected. Claim-augmented frameworks shift the focus from arguments to claims, enabling reasoning about supported conclusions rather than argument-level acceptability. In both cases, partial, entity-specific acceptance is not supported, which is a core feature of ADFDs.

Structured argumentation frameworks provide ways to represent and reason with domain knowledge. For example, Hunter [17] investigates various ways to support non-monotonic reasoning in deductive argumentation [18]. This enables constructing arguments based on the (possibly defeasible) domain knowledge expressed in the base logic. However, while some of these logics enable the representation of *general* knowledge using variables, arguments can only support literals as their conclusions. Moreover, the commitment to a specific logic restricts the types of knowledge that we can reason about, which was one of our motivations for basing our work on an abstract framework.

Other previous studies on the use of rules with variables in structured argumentation are [19] and [20]. Both the rule schemata in Assumption-Based Argumentation proposed in [19] and the first-order rules in ASPIC$^+$ [20] are rules that contain variables rather than ground constants, whose intended meaning is the set of all their ground instances over some underlying domain. The aims of the two studies are, however, different: the former aims to learn rule schemata from ground data, while the latter focuses on how to ground first-order rules efficiently. While there are some commonalities with our approach, our focus is on abstract argumentation and how not only to represent but also reason with non-grounded elements at the abstract level. It would be interesting to examine how ABA or ASPIC$^+$ with first-order rules could be mapped directly to AAFDs or ADFDs. This would enable reasoning over structured argumentation frameworks without requiring first-order rules to be grounded into propositional arguments.

With the exception of AAFDs, little attention has been paid in computational argumentation to defining a class of entities an argument can be applied to, despite the fact that it has been studied in related areas, such as discourse analysis [21], linguistic argumentation [22] and political argumentation [23]. However, unlike ADFDs (that use acceptance conditions), AAFDs only allow attack relations and they cannot capture the interaction between different domain entities.

## 4. Preliminaries

We provide here a brief formal description of the most relevant frameworks, namely AAFs [2], AAFDs [7] and ADFs [6]. More details can be found in the respective papers.

### 4.1. AAFs

An AAF was defined in [2] as a pair $\mathcal{F}_{AAF} = \langle \mathcal{A}, \mathcal{R} \rangle$ consisting of a (possibly infinite) set of arguments $\mathcal{A}$ and a binary attack relation $\mathcal{R}$ on this set. Acceptability semantics were defined in terms of extensions, i.e., sets of arguments. In particular, a set $E \subseteq \mathcal{A}$ is:

- A *conflict-free* set iff it contains no arguments attacking each other.
- An *admissible* set iff it is conflict-free and *defends* all its elements (i.e., for each argument $b \in \mathcal{A}$ attacking some $c \in E$, there is some $d \in E$ that attacks $b$).
- A *complete* extension iff it is admissible and contains all the arguments it defends.
- A *grounded* extension iff it is minimal (with respect to set inclusion) among the complete extensions.
- A *preferred* extension iff it is maximal among the complete extensions.
- A *stable* extension iff it is conflict-free and attacks all arguments in $\mathcal{A} \setminus E$.

### 4.2. AAFDs

In AAFDs [7], an argument is associated with a domain of application determining the objects, or contexts, where this argument is applicable to. The domain models the fact that certain arguments apply to specific situations, and that counter-arguments may just present exceptions, and not invalidate an argument as a whole.

Formally, given a universe $\mathcal{U}$ (which is just a set of elements), an AAFD is a tuple $\langle \mathcal{A}, \mathcal{R}, D \rangle$, where $\mathcal{A}$ is a set of arguments, $\mathcal{R}$ is a set of attacks, and $D : \mathcal{A} \mapsto 2^{\mathcal{U}}$ is a function associating each argument with a *domain*, i.e., a set of elements of the universe to which it applies.

Semantics are defined using the notion of a *scope*, which is just a mapping $S : \mathcal{A} \mapsto 2^{\mathcal{U}}$. A scope plays the role of extensions in standard AAFs. We want a scope to be "smaller" than the domain, therefore we define a scope to be *compliant* iff for all $a \in \mathcal{A}$, $S(a) \subseteq D(a)$, i.e., if it assigns to each argument (some of) the elements of its domain.

For $a, b \in \mathcal{A}$, $u \in \mathcal{U}$, we say that $a$ attacks $b$ in $u$ if and only if $(a, b) \in \mathcal{R}$ and $u \in D(a) \cap D(b)$, i.e., $a$ attacks $b$, and $u$ belongs to both domains. Defense is defined in a manner analogous to AAFs, i.e., a pair $(a, u) \in \mathcal{A} \times \mathcal{U}$ is defended by a scope $S$ if and only if $u \in D(a)$ and for any $b$ that attacks $a$ in $u$, there is some $c$ that attacks $b$ in $u$, and $u \in S(c)$.

The definitions for a scope to be admissible, complete, stable etc. can now be given using a pattern analogous to the one in AAFs. In particular, given an AAFD $\mathcal{F}_{AAFD} = \langle \mathcal{A}, \mathcal{R}, D \rangle$, a scope $S : \mathcal{A} \mapsto 2^{\mathcal{U}}$ is:

- A *conflict-free* scope iff it is compliant and there exist no arguments attacking each other in an element in their common scope.
- An *admissible* scope iff it is conflict-free and $S$ defends all pairs $(a, u)$ such that $u \in S(a)$.
- A *complete* scope iff it is admissible and $u \in S(a)$ whenever $S$ defends $(a, u)$.
- A *grounded* scope iff it is minimal among complete scopes, in the sense that there is no complete scope $S' \neq S$, such that $S'(a) \subseteq S(a)$ for all $a \in \mathcal{A}$.
- A *preferred* scope iff it is maximal among complete scopes, in the sense that there is no complete scope $S' \neq S$, such that $S'(a) \supseteq S(a)$ for all $a \in \mathcal{A}$.

- A *stable* scope iff it is conflict-free and, for any $a \in \mathcal{A}$, $u \in \mathcal{U}$ such that $u \in \mathcal{D}(a) \setminus S(a)$, there exists some $b \in \mathcal{A}$ that attacks $a$ in $u$, and $u \in S(b)$.

### 4.3. ADFs

ADFs were originally defined in [3], but slight adaptations have appeared in subsequent papers [4–6,24]. Our description here will be roughly based on the terminology and notation used in [6], but adapted to avoid confusion with the notations and terminology to be used for ADFDs.

Formally, an Abstract Dialectical Framework (ADF) is a tuple $\mathcal{F}_{ADF} = \langle \mathcal{A}, C \rangle$, where:

- $\mathcal{A}$ is a set of statements or arguments
- $C$ is a family of functions $C = \{C_a\}_{a \in \mathcal{A}}$, such that $C_a : 2^{\mathcal{A}} \to \{\mathbf{t}, \mathbf{f}\}$. $C_a$ is the *acceptance condition* of $a$.

The semantics of ADFs are given through *ADF interpretations*, which are functions of the form $v : \mathcal{A} \to \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. Note that ADF interpretations should not be confused with logical interpretations, although they use a similar formalism and intent. An ADF interpretation $v$ is called *two-valued* if and only if $\forall a \in \mathcal{A}$, $v(a) \in \{\mathbf{t}, \mathbf{f}\}$. An ADF interpretation determines which arguments are to be considered true (mapped to **t**) and which are to be considered false (mapped to **f**). ADF interpretations (but not two-valued ones) also allow for arguments whose valuation is undecided (mapped to **u**). Thus, intuitively, an ADF interpretation represents a (possibly partial) commitment on the acceptance (or not) of arguments. The semantics of ADFs are based on the idea of checking whether a commitment (ADF interpretation) is "consistent" with regards to $C$, i.e., whether it accepts what it should accept, and rejects what it should reject.

More precisely, given an ADF interpretation $v$ and some argument $a \in \mathcal{A}$, we can compute (using $C$) whether $a$ should be accepted, according to our commitment in $v$, by just computing $C_a(v^t)$, where $v^t = \{b \in \mathcal{A} \mid v(b) = \mathbf{t}\}$. The above formula determines whether the accepted arguments (as per $v$) lead to accepting $a$. For simplicity, we will often abuse notation and write $C_a(v)$ to denote $C_a(\{b \in \mathcal{A} \mid v(b) = \mathbf{t}\})$.

A two-valued ADF interpretation $\hat{v}$ is a *completion* of $v$ if and only if $v(a) = \hat{v}(a)$ whenever $v(a) \neq \mathbf{u}$. We denote by $[v]_2$ the set of all completions of $v$.

An ADF interpretation $v_1$ is *less informative* than $v_2$ if and only if, for all $a \in \mathcal{A}$, $v_1(a) = v_2(a)$ whenever $v_1(a) \neq \mathbf{u}$. We write $v_1 \leq_i v_2$ to indicate that $v_1$ is less informative than $v_2$. It is easy to see that $\leq_i$ is a partial order.

We define an operator $\Gamma$ over ADF interpretations. In particular $\Gamma(v) = v_0$, where:

$$v_0(a) = \begin{cases} \mathbf{t} & \text{, if } \forall \hat{v} \in [v]_2, C_a(\hat{v}) = \mathbf{t} \\ \mathbf{f} & \text{, if } \forall \hat{v} \in [v]_2, C_a(\hat{v}) = \mathbf{f} \\ \mathbf{u} & \text{, otherwise} \end{cases}$$

Now we can define semantics for ADFs. In particular, an ADF interpretation $v$ is called:

- *admissible* if and only if $v \leq_i \Gamma(v)$
- *complete* if and only if $v = \Gamma(v)$
- *preferred* if and only if it is maximally admissible (with respect to $\leq_i$)
- *grounded* if and only if it is minimally complete (with respect to $\leq_i$)
- *model* if and only if it is two-valued and, for all $a \in \mathcal{A}$, $v(a) = C_a(v)$.

To define stable ADF interpretations, we need the extra notion of a *reduct*. In particular, the *reduct of* $\mathcal{F}_{ADF}$ *on an ADF interpretation* $v$ is the ADF, say $\mathcal{F}^v_{ADF} = \langle \mathcal{A}^v, C^v \rangle$, defined as follows:

- $\mathcal{A}^v = \{a \in \mathcal{A} \mid v(a) = \mathbf{t}\}$
- For any $S \subseteq \mathcal{A}$, $C^v_a(S) = C_a(S)$

An ADF interpretation $v$ is *stable* if and only if the ADF interpretation $w : \mathcal{A}^v \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ such that $w(a) = \mathbf{t}$ for all $a \in \mathcal{A}^v$ is a grounded ADF interpretation of $\mathcal{F}^v_{ADF}$.

### 4.4. Extended logic programs

Our definitions in Section 5.1 (and also the example in Section 2 above) are based on the use of Extended Logic Programs (ELPs). The underlying language $\mathcal{L}$ of an ELP consists of constants, variables, and predicate symbols. It supports classical negation ($\neg$), in addition to the default negation (*not*) of standard logic programs. A term is a variable or a constant. An atom is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a predicate symbol of arity $n$, and each $t_i$ is a term. A literal is either an atom $a$ or its classical negation $\neg a$. An Extended Logic Program (ELP), $\Pi$, is a finite set of rules of the form:

$$r : l_1 \leftarrow l_2, \ldots, l_m, \mathit{not}\ l_{m+1}, \ldots, \mathit{not}\ l_n$$

where $r$ is a label, each $l_i$ is a literal over $\mathcal{L}$, and *not* denotes default negation. Rules with empty bodies are also called facts. We assume the answer set semantics of logic programs as defined in [25], and use the symbol $\Vdash$ to denote skeptical entailment under this semantics, i.e., a formula is entailed by a logic program iff it is in all answer sets of the program.

## 5. Formal definitions

We now proceed to define ADFDs, which, as explained above, are essentially the fusion of AAFDs with ADFs. For this reason, most of the ideas presented here have their counterpart in AAFDs [7] or ADFs [3–6].

We begin by explaining how an ADFD can be *specified*, i.e., described. For this, we use logical expressions (based on ELP, see Section 4.4), and our definition closely follows the ideas expressed informally in Section 2. We call this an *ADFD specification* (see Section 5.1). Then, in Section 5.2, we see how the ADFD specification can be translated (grounded, or concretized) into a simpler structure that employs sets and functions rather than logical expressions, and is not affected by the syntactic structure of the logical expressions; we call this *ADFD concretization* (or simply *ADFD*). This is similar in spirit to the way logical interpretations "concretize" the meaning of logical expressions.

### 5.1. The ADFD specifications

As in all argumentation frameworks, the core of an ADFD specification is a set of arguments, $\mathcal{A}$. We also adopt from AAFDs [7] the notion of a *universe* ($\mathcal{U}$), representing the elements that the arguments may refer to (e.g., the set of all citizens, in the parental benefit example).

ADFs [6] employ "links" to associate arguments, and each argument's acceptance depends on the acceptance status of the linked arguments. This dependence is captured by an acceptance condition, represented as a function or a propositional formula. We extend this idea by allowing acceptance conditions to refer to individual elements of the universe (constants in $\mathcal{U}$). This is analogous to how the expressiveness of classical propositional logic, which only allows simple boolean terms in formulas (e.g, $sng$, as a propositional term meaning that a certain, pre-specified, person is single), is enhanced by predicate logic, which allows predicates of various arities, whose truth value depends on the constants to which they are applied (e.g., $sng(Alice)$, $sng(Cindy)$ etc, representing that the given person is single).

Analogously, the concept of the domain of application used in AAFDs is represented using a logical expression that determines the elements of the domain of each argument (see Section 2 for some examples).

To formally represent acceptance conditions and domains, we use an extended logic programming language $\mathcal{L}$, whose constants are the members of $\mathcal{U}$. The predicates in $\mathcal{L}$ are chosen by the modeler to best capture the needs of the domain. However, in addition to these (arbitrarily chosen) domain-specific predicates, the language also includes certain domain-independent predicates; in particular, for each $\alpha \in \mathcal{A}$ we assume the existence of a family of unary predicates $\delta_\alpha(.)$, $\gamma_\alpha(.)$, and $acc_\alpha(.)$.

As also explained informally in the example of Section 2, the $\delta_\alpha(.)$ predicate represents the *domain of application* of an argument $\alpha \in \mathcal{A}$, whereas $\gamma_\alpha(.)$ represents its *acceptance condition*.

Further, $acc_\alpha(.)$ represents the *acceptance status* of $\alpha$ and is called an *interpretation* (corresponding to the notion of an interpretation in ADFs). As with ADFs, an interpretation represents a (possibly arbitrary and possibly partial) choice regarding what we choose to accept and what not, and is expressed using logical formulas determining the truth value of $acc_\alpha(u)$ for different $\alpha \in \mathcal{A}$, $u \in \mathcal{U}$. Importantly, because these are unary predicates (rather than boolean variables), we can reason about the different elements in the universe independently; this constitutes the main departure from the ADF model. For example, $acc_{epb}(Cindy)$ may be true, meaning that we have concluded that Cindy is eligible for a parental benefit, making $epb$ accepted for the individual "Cindy"; whereas $acc_{epb}(Bob)$ may be false, meaning that we have concluded that Bob is not eligible for a parental benefit.

To describe the domains of application and the acceptance conditions, we employ logical expressions, as explained informally in Section 2. Further, we assume the existence of a logic program, $\Pi$, which determines the ground truth, i.e., what is true in the world about the different elements in the universe (e.g., that Cindy is single, or that Alice and Bob are spouses), but may also include rules (e.g., that a parent is a citizen, or that if someone is single then they can have no spouse).

The *ADFD specifications* and their components as well as the *interpretation specifications* are defined formally in Definitions 1 and 2, respectively:

**Definition 1.** A Specification for an Abstract Dialectical Framework with Domain Assignments (ADFD specification for short) is a tuple $\mathcal{F}_{sp} = \langle \mathcal{A}, \mathcal{U}, \mathcal{L}, \Pi, \Delta, \Gamma \rangle$ such that:

- $\mathcal{A}$ is a finite set of *arguments*
- $\mathcal{U}$ is a finite set of *elements*, disjoint from $\mathcal{A}$
- $\mathcal{L}$ is an extended logic programming language, whose set of constants is $\mathcal{U}$, and which contains (among others) one unary predicate $\delta_\alpha$, one unary predicate $\gamma_\alpha$, and one unary predicate $acc_\alpha$ for each $\alpha \in \mathcal{A}$. We denote by $\mathcal{L}^\Gamma$ the subset of $\mathcal{L}$ that does not contain the $\delta_\alpha$ and $\gamma_\alpha$ predicates; and by $\mathcal{L}^\Pi$ the subset of $\mathcal{L}^\Gamma$ that does not contain the $acc_\alpha$ predicates.
- $\Pi$ is a logic program in $\mathcal{L}^\Pi$
- $\Delta = \{r_i^\Delta : \delta_\alpha(X) \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \mid i, m, n \geq 0, l_j \in \mathcal{L}^\Pi, \alpha \in \mathcal{A}\}$
- $\Gamma = \{r_i^\Gamma : \gamma_\alpha(X) \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \mid i, m, n \geq 0, l_j \in \mathcal{L}^\Gamma, \alpha \in \mathcal{A}\}$

**Definition 2.** Consider an ADFD specification $\mathcal{F}_{sp} = \langle \mathcal{A}, \mathcal{U}, \mathcal{L}, \Pi, \Delta, \Gamma \rangle$. An interpretation specification for $\mathcal{F}_{sp}$ is a set of rules

$$E = \{r_i^+ : acc_\alpha(X) \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \mid i, m, n \geq 0, \\ l_j \in \mathcal{L}^\Pi, \alpha \in \mathcal{A}\} \cup \\ \{r_i^- : \neg acc_\alpha(X) \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \mid i, m, n \geq 0, \quad l_j \in \mathcal{L}^\Pi, \alpha \in \mathcal{A}\}$$

such that there is no $\alpha \in \mathcal{A}$, $u \in \mathcal{U}$ for which $\Pi \cup E \Vdash \{acc_\alpha(u), \neg acc_\alpha(u)\}$.

In the above definitions, the rules in $\Delta$, $\Gamma$ and $E$ employ the standard ELP notation of using $X$ as a variable. The sublanguage $\mathcal{L}^\Pi$ is essentially the subset of $\mathcal{L}$ restricted to the domain-specific predicates (and is used to represent $\Pi$, and also for the body of rules in $\Delta$ and $E$), whereas $\mathcal{L}^\Gamma$ is the subset of $\mathcal{L}$ that is used for the body of rules in $\Gamma$ (acceptance conditions), and enhances $\mathcal{L}^\Pi$ with the acceptance status predicates $acc_\alpha$.

The set $\Delta$ contains all the rules that determine the domain of application of each argument; thus, an element $u$ is in the domain of an argument $\alpha$ if and only if $\delta_\alpha(u)$ is implied by the rules in $\Delta$ and the logic program $\Pi$, or, more formally, if and only if $\Pi \cup \Delta \Vdash \delta_\alpha(u)$. Note that $\Delta$ can only contain predicates from $\mathcal{L}^\Pi$, because the domain of application can only be defined on the basis of domain-specific predicates, and should not be affected by the arguments' acceptance conditions or statuses. Further, note that the absence of any rules with head $\delta_\alpha(X)$ in $\Delta$ for some $\alpha \in \mathcal{A}$ implies that $\alpha$ is not applicable for any $u \in \mathcal{U}$ (i.e., its domain is empty), whereas multiple rules with the same head express logical disjunction.

Pertaining to Definition 2, we say that an argument-element pair $(\alpha, u) \in \mathcal{A} \times \mathcal{U}$ is *accepted* under $E$ if and only if $\Pi \cup E \Vdash acc_\alpha(u)$; we say that it is *rejected* under $E$ if and only if $\Pi \cup E \Vdash \neg acc_\alpha(u)$; and we say that it is *undecided* under $E$ if and only if it is neither accepted nor rejected under $E$.

Further, the latter condition in Definition 2 (i.e., that there is no $\alpha \in \mathcal{A}$, $u \in \mathcal{U}$ for which $\Pi \cup E \Vdash \{acc_\alpha(u), \neg acc_\alpha(u)\}$), ensures that no interpretation specification will assign a dual status (both accepted and rejected) for the same argument-element pair; on the contrary, it is possible that for a certain argument-element pair an *undecided* status is assigned (i.e., neither accepted, nor rejected). Note also that the body of rules in $E$ is formulated using terms from $\mathcal{L}^\Pi$, as the acceptance status of an argument in an interpretation is determined independently of the domain assignments ($\delta_\alpha$), acceptance conditions ($\gamma_\alpha$), and acceptance statuses ($acc_\alpha$) of other arguments.

Importantly, one should note the difference in the intended meaning of the $acc_\alpha(.)$ and $\gamma_\alpha(.)$ predicates. The former ($acc_\alpha(.)$) encodes the *acceptance status*, i.e., what has already been decided as being accepted (through the interpretation specification). Intuitively, if $acc_\alpha(u)$ is true, then this means that "it has been decided that element $u$ is accepted for argument $\alpha$". An interpretation specification formalizes a (possibly arbitrary) collective decision as to what we choose to accept and what not.

The latter ($\gamma_\alpha(.)$), as well as the associated rules in $\Gamma$ in Definition 1, determine what *should be* accepted, given the interpretation specification, i.e., given our choices regarding the acceptance status of the various argument-element pairs represented by the truth values of $acc_\alpha(.)$. This determines whether our choices (in the interpretation specification) "make sense"; the formal definition of what it means for our choices to "make sense" will be given later, when we describe the semantics (Section 5.3).

The rules in $\Gamma$ are formed using the language $\mathcal{L}^\Gamma$, which includes the $acc_\alpha(.)$ predicates, but not the $\delta_\alpha(.)$ and $\gamma_\alpha(.)$ predicates, as it does not make sense for the domain of application or acceptance condition of an argument to affect the acceptance of another (or the same) argument. Intuitively, an element $u$ of an argument $\alpha$ should be accepted if and only if $\gamma_\alpha(u)$ is implied by the world knowledge $\Pi$, the rules in $\Gamma$ and the interpretation specification, or, more formally if and only if $\Pi \cup \Gamma \cup E \Vdash \gamma_\alpha(u)$. As with $\Delta$, the absence of rules with head $\gamma_\alpha(X)$ in $\Gamma$ means that $\alpha$ is not acceptable for any $u \in \mathcal{U}$, whereas multiple rules with the same head express logical disjunction.

Note how this definition of acceptance conditions allows the modeler to tweak separately the accepted elements for each argument and for each possible set of accepted argument-element pairs. This allows interesting features, such as to base the acceptance of an element (solely) on the acceptance of other elements of other arguments (e.g., that the benefit eligibility for a parent depends on the status of their spouse), to support scenarios where the acceptance of an element for an argument depends on the acceptance of all, some, or any specific set of elements for any argument (e.g., that a parent is eligible for parental benefit only for children who are less than five years old), or to define special cases (e.g., elements that should be treated differently than others in terms of acceptance, such as parents with certain disabilities who should be treated in a special manner). Some of these features are illustrated in the example of Section 2 and presented more formally in the example below:

**Example 1.** Following Definition 1, the policy in our running example can be modeled using the ADFD specification $\mathcal{F}_{sp} = \langle \mathcal{A}, \mathcal{U}, \mathcal{L}, \Pi, \Delta, \Gamma \rangle$ with:

$\mathcal{A} = \{epb, sng, rcb\}$

$\mathcal{U} = \{alice, bob, cindy\}$

$\mathcal{L} = (\{alice, bob, cindy\}, \{X, Y\}, \mathcal{P})$ where

the constants of $\mathcal{L}$ are $\{alice, bob, cindy\}$, the variables of $\mathcal{L}$ are $\{X, Y\}$,

and the predicates of $\mathcal{L}$ are

$\mathcal{P} = \{citizen/1, parent/1, receivesBenefit/1, single/1, spouse/2\} \cup \bigcup_{\alpha \in \mathcal{A}} \{\delta_\alpha/1, \gamma_\alpha/1, acc_\alpha/1\}$

$\Pi = \{spouse(X, Y) \leftarrow spouse(Y, X),$

$parent(alice).\ citizen(alice).\ receivesBenefit(alice).\ spouse(alice, bob).$

$parent(bob).\ citizen(bob).$

$parent(cindy).\ citizen(cindy).\ single(cindy).\}$

$\Delta = \{r_1^\Delta : \delta_{epb}(X) \leftarrow parent(X),$

$$
\begin{aligned}
& r_2^{\Delta}: \delta_{sng}(X) \leftarrow citizen(X), \\
& r_3^{\Delta}: \delta_{rcb}(X) \leftarrow citizen(X)\}, \\
\Gamma = \{ & r_1^{\Gamma}: \gamma_{epb}(X) \leftarrow acc_{sng}(X), \neg acc_{rcb}(X), \\
& r_2^{\Gamma}: \gamma_{epb}(X) \leftarrow spouse(X,Y), \neg acc_{rcb}(Y), \neg acc_{rcb}(X), \\
& r_3^{\Gamma}: \gamma_{sng}(X) \leftarrow single(X), \\
& r_4^{\Gamma}: \gamma_{rcb}(X) \leftarrow receivesBenefit(X) \}
\end{aligned}
$$

Note that $\mathcal{U}$ and $\Pi$ can be extended to include all citizens of the country and facts about them. The following is an interpretation specification for $\mathcal{F}_{sp}$:

$$
\begin{aligned}
E_1 = \{ & r_1^{+}: acc_{epb}(X) \leftarrow parent(X), single(X), not\ receivesBenefit(X), \\
& r_2^{+}: acc_{epb}(X) \leftarrow parent(X), spouse(X,Y), not\ receivesBenefit(X), \\
& \qquad\qquad not\ receivesBenefit(Y), \\
& r_3^{+}: acc_{sng}(X) \leftarrow single(X), \\
& r_4^{+}: acc_{rcb}(X) \leftarrow receivesBenefit(X), \\
& r_1^{-}: \neg acc_{epb}(X) \leftarrow citizen(X), not\ parent(X), \\
& r_2^{-}: \neg acc_{epb}(X) \leftarrow receivesBenefit(X), \\
& r_3^{-}: \neg acc_{epb}(X) \leftarrow spouse(X,Y), receivesBenefit(Y), \\
& r_4^{-}: \neg acc_{sng}(X) \leftarrow citizen(X), not\ single(X), \\
& r_5^{-}: \neg acc_{rcb}(X) \leftarrow citizen(X), not\ receivesBenefit(X)\}
\end{aligned}
$$

Based on the specifications above, the argument-element pairs accepted under $E_1$ include $(epb, cindy)$, as $\Pi \cup E_1 \Vdash acc_{epb}(cindy)$. This means that according to $E_1$, Cindy is eligible for parental benefit. On the other hand, $(epb, alice)$ and $(epb, bob)$ are rejected under $E_1$ (as $\Pi \cup E_1 \Vdash \neg acc_{epb}(alice), \neg acc_{epb}(bob)$), which means that, according to $E_1$, neither Alice nor Bob are eligible for parental benefit. For these three pairs, the following statements also hold: $\Pi \cup \Gamma \cup E_1 \Vdash \gamma_{epb}(cindy)$, $\Pi \cup \Gamma \cup E_1 \nVdash \gamma_{epb}(alice)$, $\Pi \cup \Gamma \cup E_1 \nVdash \gamma_{epb}(bob)$. This means that $E_1$ is "consistent" with the acceptance conditions of $epb$. It is easy to verify that the same holds for $sng$ and $rcb$ too, making $E_1$ a "meaningful" interpretation. In Section 5.3, we formally define such interpretations (i.e. where the acceptance atoms that they entail are consistent with the arguments' acceptance conditions), as conflict-free.

Consider now an alternative interpretation $E_2$, which is derived from $E_1$, by swapping the heads of rules $r_1^{+} - r_2^{+}$ with those of $r_1^{-} - r_3^{-}$. Under $E_2$, $(epb, cindy)$ is rejected, while $(epb, alice)$ and $(epb, bob)$ are accepted. This interpretation, however, is not consistent with the acceptance conditions of $epb$, since $\Pi \cup \Gamma \cup E_2 \Vdash \gamma_{epb}(alice), \gamma_{epb}(bob)$, and $\Pi \cup \Gamma \cup E_2 \nVdash \gamma_{epb}(cindy)$, and, therefore, $E_2$ is not conflict-free (see Section 5.3).

### 5.2. Deriving the ADFD concretization

The ADFD specification, as defined above, facilitates modeling because it allows the modeler to model the domain of interest in a declarative way, thereby improving succinctness, compactness of the descriptions, as well as modeling intuitiveness and ease of debugging. Despite the merits of the ADFD specification formalism, we provide in this subsection an alternative formulation of ADFDs, which, in a sense, "distills" the actual meaning that the ADFD specification provides, by grounding it into a simpler structure that uses sets and functions. We call this the *concretization of the ADFD specification*, or simply the *ADFD concretization*.

The ADFD concretization will be used later (Section 5.3 to define the semantics of ADFDs in a set-theoretic (rather than a proof-theoretic) manner, which we believe is more appropriate to the problem at hand and is the method being used also in ADFs and other formalisms. Although the ADFD concretization is less intuitive and more cumbersome to describe by a modeler, it facilitates formal treatment and proofs because it is unaffected by the syntactic structure of logical rules. These two ADFD variants (specification and concretization) combine the best of both worlds: a modeler can use the ADFD specification to describe an ADFD, which can then be "translated" into a *concrete ADFD* (i.e., its ADFD concretization) to perform reasoning.

We start our formal description with the definition of concrete interpretations.

**Definition 3.** Consider an ADFD specification $\mathcal{F}_{sp} = \langle \mathcal{A}, \mathcal{U}, \mathcal{L}, \Pi, \Delta, \Gamma \rangle$, and some interpretation specification $E$ for $\mathcal{F}_{sp}$. The concrete interpretation (or simply interpretation) of $E$ over $\mathcal{F}_{sp}$, denoted by $SI(E, \mathcal{F}_{sp})$ is a pair $(S_{\text{ACC}}, S_{\text{REJ}})$ such that:

- $S_{\text{ACC}} = \{(\alpha, u) \mid \alpha \in \mathcal{A}, u \in \mathcal{U}, \Pi \cup E \Vdash acc_{\alpha}(u)\}$
- $S_{\text{REJ}} = \{(\alpha, u) \mid \alpha \in \mathcal{A}, u \in \mathcal{U}, \Pi \cup E \Vdash \neg acc_{\alpha}(u)\}$

**Example 2.** Consider the ADFD specification $\mathcal{F}_{sp}$ and its two interpretation specifications, $E_1$ and $E_2$, given in Example 1. The concrete interpretation of $E_1$ over $\mathcal{F}_{sp}$ ($SI(E_1, \mathcal{F}_{sp})$) is the pair $(S^1_{\text{ACC}}, S^1_{\text{REJ}})$, where:

- $S^1_{\text{ACC}} = \{(sng, cindy), (rcb, alice), (epb, cindy)\}$

- $S^1_{\text{REJ}} = \{(sng, alice), (sng, bob), (rcb, bob), (rcb, cindy), (epb, alice), (epb, bob)\}$

On the other hand, $SI(E_2, \mathcal{F}_{sp}) = (S^2_{\text{ACC}}, S^2_{\text{REJ}})$, where

- $S^2_{\text{ACC}} = \{(sng, cindy), (rcb, alice), (epb, alice), (epb, bob)\}$
- $S^2_{\text{REJ}} = \{(sng, alice), (sng, bob), (rcb, bob), (rcb, cindy), (epb, cindy)\}$

Observe that, for any ADFD specification $\mathcal{F}_{sp}$ and interpretation specification $E$, if $(S_{\text{ACC}}, S_{\text{REJ}}) = SI(E, \mathcal{F}_{sp})$, then $S_{\text{ACC}} \cap S_{\text{REJ}} = \emptyset$. This follows from the requirement (in Definition 2) that it cannot be the case that $\Pi \cup E \Vdash \{acc_\alpha(u), \neg acc_\alpha(u)\}$ for any $\alpha \in \mathcal{A}$, $u \in \mathcal{U}$. Also, obviously, $S_{\text{ACC}} \cup S_{\text{REJ}} \subseteq \mathcal{A} \times \mathcal{U}$; in the case of equality, i.e., if $S_{\text{ACC}} \cup S_{\text{REJ}} = \mathcal{A} \times \mathcal{U}$, then $S$ is called *total*, otherwise it is called *partial*. The latter distinction will prove crucial when describing the ADFD semantics in Section 5.3.

Definition 3 illustrates the principle mentioned above about "distilling" the crucial part of the interpretation specification. In particular, the exact logical form of the rules in $E$ is not important; instead, what is really important is what these rules imply in terms of the truth values of $acc_\alpha(u)$ and $\neg acc_\alpha(u)$, which is exactly the information encoded in the concrete interpretation $SI(E, \mathcal{F}_{sp})$. One could argue that the logical form may also incorporate the "meaning" or "intent" of the modeler, but, as already mentioned, the concrete versions will be used for the mathematical manipulation of these structures, where the modeler's intent is of no concern. These ideas give rise to the notion of equivalent interpretation specifications:

**Definition 4.** Given an ADFD specification $\mathcal{F}_{sp}$, and two interpretation specifications $E_1$ and $E_2$ (for $\mathcal{F}_{sp}$), we say that $E_1$ is equivalent to $E_2$ for $\mathcal{F}_{sp}$, denoted by $E_1 \equiv^{\mathcal{F}_{sp}} E_2$, if and only if $SI(E_1, \mathcal{F}_{sp}) = SI(E_2, \mathcal{F}_{sp})$.

Note that the equivalence relation is defined with respect to some ADFD specification, because the concrete interpretation is also defined with respect to an ADFD specification; as a matter of fact, two interpretation specifications may be equivalent for some $\mathcal{F}_{sp}$, but not be equivalent for some other. Nevertheless, the ADFD specification will in most cases be fixed and obvious from the context, so the superscript in the equivalence relation will be omitted.

Clearly, the relation of Definition 4 is indeed an equivalence relation (i.e., reflexive, symmetric and transitive) for any given $\mathcal{F}_{sp}$. Also, it is clear that equivalent interpretation specifications are indeed "the same", in the sense that they dictate the same truth values for $acc_\alpha(u)$, $\neg acc_\alpha(u)$.

Another obvious corollary of the above definitions is that, for any given ADFD specification $\mathcal{F}_{sp}$, each interpretation specification $E$ is mapped to a single concrete interpretation ($S = SI(E, \mathcal{F}_{sp})$), but each concrete interpretation $S$ can be generated by many different interpretation specifications; i.e., it is possible that $SI(E, \mathcal{F}_{sp}) = SI(E', \mathcal{F}_{sp})$ for $E \neq E'$. In this case, $E \equiv E'$ by definition. For ease of presentation, we will define the inverse of $SI$, denoted by $SI^{-1}$, to pick a specific interpretation specification from the various equivalent ones. In particular, for a given concrete interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$, we define: $SI^{-1}(S) = \{acc_\alpha(u) \leftarrow \mid (\alpha, u) \in S_{\text{ACC}}\} \cup \{\neg acc_\alpha(u) \leftarrow \mid (\alpha, u) \in S_{\text{REJ}}\}$, i.e., an interpretation specification whose rules have an empty body and whose head include the accepted/rejected argument-element pairs, as dictated by $S$ (i.e., $SI^{-1}(S)$ is an ELP consisting entirely of facts).

Now we can define the concrete ADFD (or ADFD concretization):

**Definition 5.** Given an ADFD specification $\mathcal{F}_{sp} = \langle \mathcal{A}, \mathcal{U}, \mathcal{L}, \Pi, \Delta, \Gamma \rangle$, the ADFD concretization of $\mathcal{F}_{sp}$ (or simply ADFD) is a tuple $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, \mathcal{D}, \mathcal{C} \rangle$, where:

- $\mathcal{A}$ is the set of arguments of $\mathcal{F}$, equal to the set of arguments in $\mathcal{F}_{sp}$
- $\mathcal{U}$ is the set of elements of $\mathcal{F}$, equal to the set of elements in $\mathcal{F}_{sp}$
- $\mathcal{D}$ is a function $\mathcal{D} : \mathcal{A} \to 2^{\mathcal{U}}$ such that, for any given $\alpha \in \mathcal{A}$, $\mathcal{D}(\alpha) = \{u \in \mathcal{U} \mid \Pi \cup \Delta \Vdash \delta_\alpha(u)\}$
- $\mathcal{C}$ is a function $\mathcal{C} : \mathcal{A} \times 2^{\mathcal{A} \times \mathcal{U}} \to 2^{\mathcal{U}}$ such that, for any given $\alpha \in \mathcal{A}$ and $S^* \subseteq \mathcal{A} \times \mathcal{U}$, $\mathcal{C}(\alpha, S^*) = \{u \in \mathcal{U} \mid \Pi \cup \Gamma \cup E \Vdash \gamma_\alpha(u)\}$, where $E = SI^{-1}(S)$, and $S = (S^*, (\mathcal{A} \times \mathcal{U}) \setminus S^*)$.

Note again how the ADFD concretization "distills" what the ADFD specification dictates, through the use of the two functions $\mathcal{D}, \mathcal{C}$, which determine what is in the domain of each argument and what should be accepted, respectively.

In particular, $u$ is in the domain of $\alpha$ if and only if $u \in \mathcal{D}(\alpha)$. Analogously, $u$ should be accepted in $\alpha$, for a given set of accepted argument-element pairs $S^*$, if and only if $u \in \mathcal{C}(\alpha, S^*)$. It is important to note that $\mathcal{C}$ receives as input one set only (the set of accepted argument-element pairs) rather than two sets (accepted plus rejected, i.e., a concrete interpretation). This follows the pattern used in ADFs (e.g., see Definition 3.1 in [6], or Definition 4.1 in [5] and analogous definitions in other papers), where two-valued interpretations are considered, rather than three-valued. In these definitions, it is assumed that whatever is not included in that set ($S^*$ in our notation) is rejected; this is the intuition behind the definition of $S$ and $E$ in Definition 5.

**Example 3.** The concretization of the ADFD specification $\mathcal{F}_{sp} = \langle \mathcal{A}, \mathcal{U}, \mathcal{L}, \Pi, \Delta, \Gamma \rangle$ given in Example 1, is $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, \mathcal{D}, \mathcal{C} \rangle$, where

$$\mathcal{D}(sng) = \mathcal{D}(rcb) = \mathcal{D}(epb) = \{alice, bob, cindy\}$$

and $\mathcal{C}$ is defined as follows, for any given $S^* \subseteq \mathcal{A} \times \mathcal{U}$:

$$\mathcal{C}(sng, S^*) = \{cindy\}$$

$$\mathcal{C}(rcb, S^*) = \{alice\}$$

$$\mathcal{C}(epb, S^*) = \{u_i \mid ((sng, u_i) \in S^* \wedge (rcb, u_i) \notin S^*) \vee \quad (u_i \in \{alice, bob\} \wedge (rcb, alice) \notin S^* \wedge (rcb, bob) \notin S^*)\}$$

We can also define equivalent ADFD specifications, in the spirit of equivalent interpretation specifications (Definition 4):

**Definition 6.** Consider two ADFD specifications $\mathcal{F}^1_{sp}$, $\mathcal{F}^2_{sp}$, and their concretizations $\mathcal{F}^1$, $\mathcal{F}^2$, respectively. We say that $\mathcal{F}^1_{sp}$ is equivalent to $\mathcal{F}^2_{sp}$, denoted by $\mathcal{F}^1_{sp} \equiv \mathcal{F}^2_{sp}$, if and only if $\mathcal{F}^1 = \mathcal{F}^2$.

Again, it is easy to see that the relation of Definition 6 is indeed an equivalence relation (i.e., reflexive, symmetric and transitive) and that equivalent ADFD specifications are indeed "the same", in the sense that they carry the same semantics and differ only in the syntactic formulation of the rules that determine those semantics.

The definition of ADFD specifications as a logic program (Definition 1) facilitates the modeling of the domain of interest, thereby improving succinctness, compactness of the descriptions, as well as modeling intuitiveness and ease of debugging. This is due to the declarative nature of the definition, as opposed to the ADFD concretizations which require explicitly defining the relevant functions. On the other hand, the concrete ADFDs (Definition 5) are "clearer" from a formal perspective, in the sense that they are not affected by the syntactic form of the logical rules, and will be used to define the semantics of ADFDs in a set-theoretic manner. Thus, the rest of the formal treatment in the following sections considers the concrete ADFDs only, and we use the term "ADFD" to refer to the concrete ADFD (analogously, "interpretation" refers to the concrete interpretation). The results obviously apply for the ADFD/interpretation specifications as well, and, in fact, we implicitly assume that there is some ADFD/interpretation specification from which the concrete ADFD/interpretation is derived. On the contrary, our implementation (Section 7) assumes an ADFD specification as input, as the declarative way of determining an ADFD/interpretation (through the specifications) is more suitable for practical purposes. For the same reason, we expect modelers to adopt the declarative variant.

### 5.3. ADFD semantics

A critical concept related to the definition of ADFD semantics is the comparison of interpretations on the basis of their level of commitment (or informativeness) that they impose. Intuitively, a "less informative" interpretation commits as to the truth value (acceptance or rejection) of fewer items (argument-element pairs). However, an important prerequisite for two interpretations to be comparable in terms of informativeness is to agree on the items that they commit to. This is better clarified in the following formal definition (which is analogous to the definition of $\leq_i$ for ADF interpretations):

**Definition 7.** Given an ADFD $\mathcal{F}$, and two interpretations $S = (S_{\text{ACC}}, S_{\text{REJ}})$, $S' = (S'_{\text{ACC}}, S'_{\text{REJ}})$ of $\mathcal{F}$, we say that $S$ is less informative than $S'$, denoted by $S \sqsubseteq S'$, if and only if $S_{\text{ACC}} \subseteq S'_{\text{ACC}}$ and $S_{\text{REJ}} \subseteq S'_{\text{REJ}}$.

It is easy to show that $\sqsubseteq$ is a partial order (i.e., reflexive, antisymmetric, and transitive). Moreover, by Definition 7, it is clear that total interpretations are always maximal with respect to $\sqsubseteq$. Given a partial interpretation $S$, the total interpretations that are more informative than $S$ are those for which we take some stance (either accept or reject) toward the argument-element pairs for which $S$ is uncommitted. Those total interpretations are important for our analysis later, which is why we give them a special name:

**Definition 8.** Given an ADFD $\mathcal{F}$, and two interpretations $S, S'$ of $\mathcal{F}$, we say that $S'$ is a completion of $S$ if and only if $S'$ is a total interpretation and $S \sqsubseteq S'$. We denote by $Compl(S)$ the set of all completions of $S$.

**Example 4.** Consider the following partial concrete interpretation of the ADFD $\mathcal{F}$ given in Example 3: $S^3 = (S^3_{\text{ACC}}, S^3_{\text{REJ}})$ with

$S^3_{\text{ACC}} = \{(sng, cindy), (rcb, alice)\}$

$S^3_{\text{REJ}} = \{(sng, alice), (sng, bob), (rcb, bob), (rcb, cindy)\}$

Both concrete interpretations of $\mathcal{F}_{sp}$ given in Example 2, $S^1 = (S^1_{\text{ACC}}, S^1_{\text{REJ}})$ and $S^2 = (S^2_{\text{ACC}}, S^2_{\text{REJ}})$, are completions of $S^3$.

As seen in Definition 5, the acceptance condition of an argument assumes a full commitment as to the accepted argument-element pairs. Indeed, $C$ takes as input a set $S^*$ of accepted argument-element pairs, implicitly assuming that all other argument-element pairs are rejected. However, in many cases, even a partial commitment as to the accepted/rejected argument-element pairs may be enough to determine whether an argument will be accepted for a certain element. This leads to the definition of decisive acceptance/rejection (similar to the idea of decisively in/out in [5]):

**Definition 9.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then:

- We say that $(\alpha, u) \in \mathcal{A} \times \mathcal{U}$ is *decisively accepted* by $S$ if and only if $u \in D(\alpha)$, and for any $S' = (S'_{\text{ACC}}, S'_{\text{REJ}}) \in Compl(S)$ it holds that $u \in C(\alpha, S'_{\text{ACC}})$.
- We say that $(\alpha, u) \in \mathcal{A} \times \mathcal{U}$ is *decisively rejected* by $S$ if and only if $u \notin D(\alpha)$, or for any $S' = (S'_{\text{ACC}}, S'_{\text{REJ}}) \in Compl(S)$ it holds that $u \notin C(\alpha, S'_{\text{ACC}})$.

We denote by $DEC_{\text{ACC}}(S)$, $DEC_{\text{REJ}}(S)$ the elements of $\mathcal{A} \times \mathcal{U}$ that are decisively accepted/rejected (respectively) by $S$, and by $DEC(S)$ the interpretation $(DEC_{\text{ACC}}(S), DEC_{\text{REJ}}(S))$.

**Example 5.** $(sng, cindy)$ is decisively accepted by $S^3$ (given in Example 4) as $cindy \in D(sng)$ and $cindy \in C(sng, S^3_{\text{ACC}})$ for any completion of $S^3$. On the other hand, $(epb, alice)$ is decisively rejected by $S^2$ (given in Example 2) as $S^2$ is a total interpretation and $alice \notin C(epb, S^2_{\text{ACC}})$, i.e., Alice does not satisfy the acceptance conditions of $epb$ (because she receives a cash benefit).

To define ADFD semantics, we identify "interesting" interpretations, i.e., interpretations that satisfy certain intuitive properties. As explained above, an "interesting" interpretation is one in which our acceptance/rejection choices "make sense", with respect to the $D$ function (i.e., the domains of the various arguments), and, more importantly, with respect to the $C$ function of the ADFD (acceptance conditions, i.e., what should be accepted).

We start with *compliant interpretations*, which are those that are faithful to the domain specification of each argument. For example, if a certain argument is about parents, an interpretation that accepts this argument for someone who is not a parent is not compliant. Unlike AAFDs [7], here we have two notions of compliance: the first requires that an element that is not in an argument's domain should not be accepted for this argument; the second requires that such an element should be rejected. The reason for this additional compliance notion is that in AAFDs [7] only accepted and rejected argument-element pairs are allowed, i.e., there is no third scenario (undecided).

**Definition 10.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then:

- $S$ is *compliant* with $\mathcal{F}$ if and only if $u \in D(\alpha)$ whenever $(\alpha, u) \in S_{\text{ACC}}$.
- $S$ is *fully compliant* with $\mathcal{F}$ if and only if $(\alpha, u) \in S_{\text{REJ}}$ whenever $u \notin D(\alpha)$.

A fully compliant interpretation requires that an element that is not in an argument's domain should be rejected for this argument; a compliant interpretation requires that such an element should not be accepted. Thus, a fully compliant interpretation is also compliant (see also Proposition 3).

**Example 6.** $S^1$ (of Example 2) is compliant because $cindy \in D(sng)$, $alice \in D(rcb)$ and $cindy \in D(epb)$ (see Example 3). It is also fully compliant because there are no $\alpha \in \mathcal{A}$, $u \in \mathcal{U}$ for which $u \notin D(\alpha)$. Similarly, $S^2$ (see Example 2) and $S^3$ (see Example 4) are also fully compliant (and, therefore, compliant).

We continue with the conflict-free interpretations, which are those whose commitments are not "outright irrational", in the sense that they do not accept argument-element pairs they ought to reject or vice versa:

**Definition 11.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then, $S$ is *conflict-free* in $\mathcal{F}$ if and only if $(S_{\text{ACC}} \cap DEC_{\text{REJ}}(S)) \cup (S_{\text{REJ}} \cap DEC_{\text{ACC}}(S)) = \emptyset$.

**Example 7.** $S^1$, given in Example 2, is conflict-free because none of the argument-element pairs in $S^1_{\text{ACC}}$ is decisively rejected, and none of the argument-element pairs in $S_{\text{REJ}}$ is decisively accepted. On the other hand, $S^2$ is not conflict-free, as $(epb, alice)$, which belongs to $S^2_{\text{ACC}}$, is decisively rejected by $S^2$ as shown in Example 5.

We should note that in most ADF papers (e.g., Brewka and Woltran [3], Polberg [5,24]) conflict-freeness is defined for sets (i.e., two-valued structures, where elements are either accepted or rejected), not interpretations (three-valued structures, which also admit undecided elements), as we do here. Three-valued conflict-free interpretations in ADFs were defined in [26], and (with minor variations) in [27]. The intuition behind Definition 11 above is drawn from the respective three-valued definition found in [26].

Model interpretations are similarly inspired by the respective definition in [3]. The idea behind a model interpretation is that it provides one possible way to "interpret" the entire framework (i.e., it is a total interpretation), and it is also consistent with respect to the acceptance conditions. Thus, a model interpretation is self-justifying: the arguments that it accepts are exactly those whose acceptance conditions are satisfied for the acceptance choices made by the interpretation. Formally:

**Definition 12.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then $S$ is a *model* of $\mathcal{F}$ if and only if $S$ is total, and $(\alpha, u) \in S_{\text{ACC}}$ if and only if $u \in D(\alpha) \cap C(\alpha, S_{\text{ACC}})$.

We next define admissible, complete, grounded and preferred interpretations, which are based on the notion of decisive acceptance/rejection:

**Definition 13.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then:

- $S$ is called *admissible* if and only if $S \sqsubseteq DEC(S)$.
- $S$ is called *complete* if and only if $S = DEC(S)$.
- $S$ is called *grounded* if and only if $S$ is a minimal complete interpretation (under $\sqsubseteq$).
- $S$ is called *preferred* if and only if $S$ is a maximal admissible interpretation (under $\sqsubseteq$).

Stable interpretations are defined in the spirit of Brewka et al. [6] and require the idea of the *reduct* of an ADFD. Intuitively, a stable interpretation must be a model, in the sense that it must be self-justifying. Moreover, a stable interpretation should not include any cyclic justifications. For example, if $(a, u)$ is accepted "just because" $(b, u)$ is accepted (and vice-versa), then an interpretation accepting both should not be a stable interpretation. This idea is formalized by considering the reduct of an ADFD for an interpretation $S$, which is an ADFD that contains only the argument-element pairs that are accepted by $S$. If $S$ is a grounded interpretation of the reduct, then it does not contain cyclic supports, so it is stable. The above ideas are formalized in the following pair of definitions:

**Definition 14.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and a total interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then, the *reduct* of $\mathcal{F}$ for $S$, denoted by $\mathcal{F}^S$, is an ADFD $\mathcal{F}^S = \langle \mathcal{A}^S, \mathcal{U}^S, D^S, C^S \rangle$, defined as follows:

- $\mathcal{A}^S = \mathcal{A}$

- $\mathcal{U}^S = \mathcal{U}$
- For any $\alpha \in \mathcal{A}$, $u \in \mathcal{U}$, $u \in D^S(\alpha)$ if and only if $u \in D(\alpha)$ and $(\alpha, u) \in S_{\text{ACC}}$
- $\mathcal{C}^S = \mathcal{C}$

**Definition 15.** Consider an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, and an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$. Then $S$ is called *stable* if and only if $S$ is a model of $\mathcal{F}$ and a grounded interpretation of $\mathcal{F}^S$.

**Example 8.** $S^1$, given in Example 2, is admissible, complete, grounded, preferred and stable. It can be verified that there is no other admissible, complete, grounded, preferred or stable interpretation. We can therefore conclude that, under any of the semantics, Cindy is eligible for parental benefit, while Alice and Bob are not.

## 6. Results

In this section, we present various useful results for ADFDs and interpretations, namely: reduction results (formally relating ADFDs with ADFs and AAFDs); hierarchy and existence results (that determine the multiplicity, existence and relationships between different types of interpretations); and results on the computational complexity of various problems.

### 6.1. Reductions to other frameworks

We can easily observe that AAFDs are a special case of ADFDs. Indeed, an AAFD is an ADFD whose acceptance conditions are of a special form: accept $u$ for $\alpha$ if and only if none of the attackers of $\alpha$ accept $u$. Another obvious result is that an ADFD with a singleton universe is essentially an ADF, therefore ADFs are a special case of ADFDs.

A more difficult question is whether we can show the opposite, i.e., that ADFDs can be reduced to ADFs and/or AAFDs. It turns out that the answer is positive, but only if we allow for additional (artificial) arguments. Therefore, ADFDs provide a more convenient and compact way to describe arguments (and their relationships) about objects or contexts, but do not add to the expressive power of ADFs or AAFDs (or AAFs, for this matter).

To show this result, we observe that, although ADFs do not support domains, one could create an ADF whose arguments are the argument-element pairs of an ADFD (this idea was illustrated in our running example in Section 2). This process would result in an equivalent, albeit cumbersome, way to represent ADFDs using ADFs. Note that the same idea was employed to map AAFDs to AAFs in [7].

Before proceeding to the mapping itself, we formalize the analogy between ADF and ADFD (concrete) interpretations:

**Definition 16.** Given an interpretation $S = (S_{\text{ACC}}, S_{\text{REJ}})$, we denote by $v^S$ the mapping $v^S : \mathcal{A} \times \mathcal{U} \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$, which is defined as follows:

$$v^S(\alpha, u) = \begin{cases} \mathbf{t} & \text{, when } (\alpha, u) \in S_{\text{ACC}} \\ \mathbf{f} & \text{, when } (\alpha, u) \in S_{\text{REJ}} \\ \mathbf{u} & \text{, otherwise} \end{cases}$$

It is clear by the above definition that the mapping between ADFD and ADF interpretations is an isomorphism. I.e., for every ADFD interpretation $S$ there exists a single ADF interpretation $v$ for which $v = v^S$; and vice-versa, for every ADF interpretation $v$ there exists a single ADFD interpretation $S$ for which $v = v^S$.

We now define the mapping between ADFs and ADFDs:

**Definition 17.** For an ADFD $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$, let $\mathcal{F}^{\Phi}_{ADF}$ be the ADF $\mathcal{F}^{\Phi}_{ADF} = \langle \mathcal{A}^{\Phi}, C^{\Phi} \rangle$, where:

- $\mathcal{A}^{\Phi} = \mathcal{A} \times \mathcal{U}$
- For any $(\alpha, u) \in \mathcal{A}^{\Phi}$, $S^* \subseteq \mathcal{A}^{\Phi}$, we define $C^{\Phi}_{(\alpha,u)}(S^*) = \mathbf{t}$ if and only if $u \in C(\alpha, S^*)$ and $u \in D(\alpha)$

We call $\mathcal{F}^{\Phi}_{ADF}$ the *corresponding ADF of* $\mathcal{F}$.

We now show that the mapping respects the semantics of the respective ADF and ADFD:

**Proposition 1.** *Take $\mathcal{F} = \langle \mathcal{A}, \mathcal{U}, D, C \rangle$ and its corresponding ADF, $\mathcal{F}^{\Phi}_{ADF}$. Then, $S$ is a $\sigma$ interpretation for $\mathcal{F}$ if and only if $v^S$ is a $\sigma$ ADF interpretation for $\mathcal{F}^{\Phi}_{ADF}$, for $\sigma$ being admissible, complete, grounded, preferred, model, stable.*

Proposition 1 shows that ADFDs can be mapped to appropriate ADFs, so the reduction between ADFs and ADFDs is bidirectional; however, to realize this mapping, each argument in the ADFD must be replicated $|\mathcal{U}|$ times. Note also that a similar reduction has been shown between AAFs and ADFs [9,10], again requiring the use of additional arguments. Also, a similar result was shown between AAFDs and AAFs in [7]. Combining the above results, we conclude that all four frameworks (ADFDs, ADFs, AAFDs and AAFs) can be reduced to each other, if we allow for additional (artificial) arguments to be created in each case (see Definition 17, Proposition 1, and Vassiliades et al. [7], Strass [10], Polberg [28] for details). As a result, the main advantage of the more advanced variants (AAFDs, ADFs, and, especially, ADFDs) is not expressive power, but the fact that they allow a more compact and natural way to model a domain of application. The following proposition makes this claim more precise:

**Proposition 2.** *Consider an ADFD $\mathcal{F}$, and its corresponding ADF, $\mathcal{F}^{\Phi}_{ADF}$, as per Definition 17. It holds that $|\mathcal{F}^{\Phi}_{ADF}| = O(|\mathcal{F}|^2)$.*

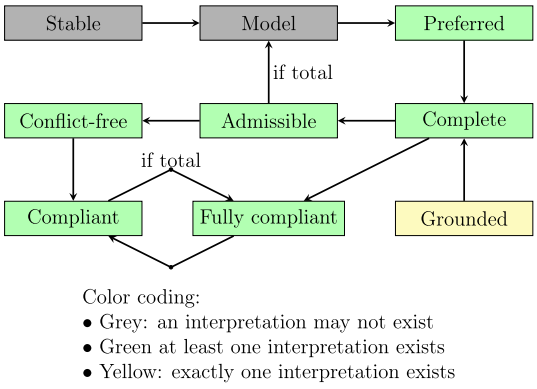



**Fig. 4.** Hierarchy results for semantics.

Note that Proposition 2 assumes that both the ADFD concretization and the corresponding ADF are represented by explicitly enumerating their components. If we consider the ADFD specification as the starting point, then the induced ADFD concretization can be at most exponentially larger than the specification; the same holds between ADFs whose acceptance conditions have the form of propositional formulas and their corresponding enumerated representations.

We conclude that, for the enumerated representations, the size of the ADF that can be used to "emulate" an ADFD is *polynomially larger*, in general, compared to the ADFD. This observation is related to the discussion on realizability, found in [10,27,29] and other works, which essentially gives a technical, formal meaning to the terms "expressiveness" and "succinctness" and applies to representation formalisms in general. Based on Proposition 2, and the definitions of Strass [10], we conclude that ADFDs and ADFs are equally expressive and equally succinct. Still, the polynomially smaller size and the more "natural" way of expressing arguments in ADFDs is a tangible and clear advantage of the ADFD formalism.

### 6.2. Hierarchy results

Using the reduction of ADFDs to ADFs shown above, several results from ADFs, AAFDs and AAFs migrate trivially to ADFDs. We outline these results here, along with some additional ones that are specific for ADFDs (see also Fig. 4 for a visualization):

**Proposition 3.** *Consider an ADFD $\mathcal{F}$ and some interpretation $S = (S_{ACC}, S_{REJ})$. Then:*

1. *If $S$ is fully compliant, then it is compliant.*
2. *If $S$ is compliant and total, then it is fully compliant.*
3. *If $S$ is conflict-free, then it is compliant.*
4. *If $S$ is admissible, then it is conflict-free.*
5. *If $S$ is complete, then it is admissible.*
6. *If $S$ is grounded, then it is complete.*
7. *If $S$ is preferred, then it is complete.*
8. *If $S$ is model, then it is preferred.*
9. *If $S$ is admissible and total, then it is model.*
10. *If $S$ is stable, then it is model.*
11. *If $S$ is complete, then it is fully compliant.*

**Proposition 4.** *For any ADFD $\mathcal{F}$, the following hold:*

1. *There exists at least one preferred interpretation*
2. *There exists exactly one grounded interpretation*

Note how Propositions 3 and 4 combined, imply the existence of several other types of interpretations, namely conflict-free, admissible, complete, fully compliant and compliant.

### 6.3. Computational complexity

We now analyze the computational complexity of ADFDs. The idea is to use the mapping developed in Section 6.1 (Definition 17) to map an ADFD (specification) into an ADF, and then use known complexity results from the ADF literature [27] to determine the ADFD complexity for various computational tasks.

The following proposition will prove useful:

**Proposition 5.** *Consider an ADFD $\mathcal{F}$, such that $|\mathcal{F}| = N$. Then:*

1. *Computing the respective ADF, $\mathcal{F}^{\Phi}_{ADF}$, requires $O(N^2)$ time.*
2. *Consider an interpretation $S$ of $\mathcal{F}$. It holds that $|S| = O(N^2)$.*
3. *Consider an interpretation $S$ of $\mathcal{F}$, and its respective ADF interpretation $v^S$. Then, $|v^S| = O(N^2)$.*
4. *Given an interpretation $S$ of $\mathcal{F}$, computing $v^S$ requires $O(N^2)$ time.*
5. *Given an interpretation $v$ of $\mathcal{F}^{\Phi}_{ADF}$, computing the interpretation $S$ of $\mathcal{F}$ for which $v = v^S$ requires $O(N^2)$ time.*

The results in Propositions 2 and 5 show that it takes polynomial time to translate an ADFD to an ADF, and that the size increase during the translation is at most polynomial. Moreover, they tell us that the same holds for interpretations: an ADF interpretation can be translated into an ADFD interpretation in polynomial time (and vice-versa), without size increase (at least asymptotically). Combining these results, we can infer that the complexity of the computational tasks regarding ADFs that were analyzed in [4,27] (and elsewhere) for the different semantics remain in the same complexity class for ADFs and ADFDs. In particular, the results shown in Table 2 of [27] (for ultimate semantics) apply equally for ADFDs, for all the semantics considered in Proposition 1 and for all the problems studied in [27].

## 7. Implementation and experiments

In the following, we present a simple implementation[1] of ADFDs (see Section 7.1), using Python combined with the Answer Set Programming Language (ASP), a popular programming tool oriented toward combinatorial search problems [30]. Our implementation focuses on the verification problem, i.e., given an ADFD $\mathcal{F}_{sp}$, an interpretation $E$ and some semantics $\sigma$, determine whether $E$ is a $\sigma$-interpretation of $\mathcal{F}_{sp}$. The verification problem is one of the main computational problems in argumentation, as evidenced by its inclusion in competitions like ICCMA[2] Note that our objective is not to provide efficient and optimized algorithms for ADFD reasoning, but rather to provide a proof-of-concept implementation that will help us experimentally explore the main bottlenecks of reasoning with ADFDs, as well as the effects of different parameters (e.g., number of arguments, number of elements, complexity of rules, etc.) on the performance of the related algorithms (see Section 7.2 for details).

Throughout this section, for brevity and to avoid repetition, we use the shorthand "*AE*" to refer to an "argument-element pair".

### 7.1. Implementation

Our implementation follows a philosophy compatible with existing ASP-based ADF solvers (e.g., Ellmauthaler and Strass [31], Brewka et al. [32]). The input has the form of an ASP program, and the grounding step performed by our system is conceptually equivalent to the standard ASP grounding of first-order predicates over the Herbrand universe, with the universe and the domain assignments controlling which argument-element pairs are instantiated. The main difference is that, unlike existing ADF solvers which focus on computing or enumerating interpretations, our system targets the verification problem, determining whether a given interpretation satisfies a specified semantics.

The general pipeline of the system is simple: the user input consists of an ADFD $\mathcal{F}_{sp}$, an interpretation $E$, and a type of semantics $\sigma$, and the output is either `true` or `false`, signifying whether $E$ is a $\sigma$-interpretation for $\mathcal{F}_{sp}$.

In more detail, the ADFD is provided in the form of an ASP program that describes the arguments in $\mathcal{A}$ (e.g., `arg(epb)`), the elements of the universe $\mathcal{U}$ (e.g., `elem(cindy)`), the logic program $\Pi$ (e.g., `spouse(X,Y) :- spouse(Y,X)`), the domain assignment in $\Delta$ (e.g., `dom(epb,X):- parent(X)`) and the acceptance conditions in $\Gamma$ (e.g., `cond(epb,X) :- acc(sng,X), -acc(rcb,X))` [3] ; note that the language $\mathcal{L}$ is not explicitly provided, as it is implicit by the above program. This information is initially parsed and stored in appropriate Python data structures.

Analogously, the interpretation is given in the form of an ASP program (e.g., `acc(epb,cindy)`, `-acc(epb,alice)`), and is parsed to produce a globally accessible dictionary called 'three_valued'.

The above structures, along with the semantics chosen by the user, are fed to our verification algorithms, which are briefly described below and whose pseudocode is given in Appendix B.

As a preprocessing of all verification algorithms, Pseudocode 1 determines whether the input interpretation is compliant, as per Definition 10. Note that if it is not compliant, then the answer to the original verification (for any of the other semantics) would be `false` (see also Proposition 3); this allows a quick exit point that speeds up the algorithms in some cases. Pseudocode 2 checks whether the interpretation is fully compliant.

To verify whether $E$ is conflict-free (Pseudocode 3), the system verifies that there are no pairs $(\alpha, u) \in \mathcal{A} \times \mathcal{U}$ that are both accepted and decisively rejected, or both rejected and decisively accepted. Note that, during verification, the implementation evaluates each grounded acceptance condition with respect to the given interpretation. We use this operation as a primitive in all verification procedures.

For the verification of admissible interpretations (Pseudocode 4), the system verifies that each accepted AE is decisively accepted and each rejected pair is decisively rejected.

[1] https://github.com/jimboelessar/adfd-semantics-verification

[2] International Competition on Computational Models of Argumentation (ICCMA), https://argumentationcompetition.org/index.html, accessed: July 23, 2026.

[3] Note that, in ASP, the '-' prefix before a predicate indicates strong negation of the predicate.

Similarly, for verifying complete interpretations (Pseudocode 5) the system verifies that each accepted, rejected, and undecided AE is decisively accepted, decisively rejected, or neither of the two, respectively.

The algorithm for grounded interpretations first computes the grounded interpretation of $\mathcal{F}_{sp}$ and then verifies that it is the same as $E$ (see Pseudocode 6). The creation of the grounded interpretation (Pseudocode 7) consists of starting from an empty interpretation, identifying the decisively accepted and decisively rejected pairs for the interpretation, and then adding them iteratively to the interpretation (as accepted and rejected, respectively). This constructive approach is faster than verifying that $E$ is minimal (with respect to $\sqsubseteq$) among the complete interpretations of $\mathcal{F}_{sp}$, and works correctly, since the grounded interpretation is unique.

To verify a preferred interpretation, the system searches for a "larger" (with respect to $\sqsubseteq$) admissible interpretation (see Pseudocode 8). If it succeeds, then the verification fails. If it exhausts all possibilities without finding one, then $E$ is maximal, and therefore preferred. Doing so in an efficient manner required a series of optimisations that allow pruning the search space by removing certain hopeless options, i.e., skipping the search for maximal admissible interpretations along branches in the search tree that cannot possibly return such an interpretation (see Pseudocodes 9-12 for details).

The algorithm for verifying model interpretations (Pseudocode 13) verifies that $E$ is total; that for each accepted AE $(\alpha, u)$, its acceptance condition evaluates to `true` and $u$ is in the domain of $\alpha$; and that each AE $(\alpha, u)$ which has a `true` acceptance condition and where $u$ is in the domain of $\alpha$, is accepted. If all these conditions hold, it determines that $E$ is a model interpretation.

For the verification of stable interpretations (Pseudocode 14), the algorithm first verifies that $E$ is a model interpretation. Then, it produces the reduct ADFD and verifies that $E$ is the reduct's grounded interpretation.

### 7.2. Experiments

We conducted a set of experiments in order to evaluate the performance of our implementation and to investigate and quantify the effect of different parameters on the total runtime of our algorithms.

#### 7.2.1. Benchmark data creation

Since the ADFD formalism is a new proposal, no ADFD datasets exist in order to perform our evaluation. Therefore, we created the datasets that we needed to evaluate our implementation by adapting the ADFs used for the benchmarks of ICCMA-21 [4] In particular, we randomly selected one ADF from each of the ABA2AF, ABA2AF_acyc, grid, metro, and traffic ADF categories. Those five ADFs differ significantly in their size (i.e., the number of arguments they have) and in the complexity of their conditions.

To make these datasets usable for our experiments, we made two main pre-processing steps. First, we transformed the argument definitions and the corresponding conditions into the format accepted by our implementation. Second, we created a universe (set of elements) and added an arbitrarily chosen subset of it to the domain of each argument, in order to produce an ADFD from the underlying ADF. In particular, we considered three different universe sizes, namely 3, 8 and 15. To equip each argument with a domain, we created a probabilistic method that adds a given element $u$ in the domain of a given argument $a$ with a probability $p$, and applied this method using four different values for $p$, namely 10%, 50%, 80%, and 100%. Ultimately, this process generated a total of 12 different ADFDs from each of the five original ADFs, one for each different combination of values for the size of the universe (3, 8, or 15) and for the probability $p$ (10%, 50%, 80%, and 100%).

#### 7.2.2. Generating interpretations

To evaluate our verification algorithms on the created ADFDs, we also generated a number of interpretations for each ADFD. This process is detailed below.

First, we created some initial interpretations in a straightforward manner by selecting $AEs$ and assigning values to them. To ensure that the created interpretations were compliant (and would, therefore, pass the first check of the verification algorithms), our interpretations classified as $rej$ (rejected) every pair $(a, u)$ for which $u$ is *not* in the domain of $a$. For the rest of the $AEs$, we used the following procedures, which led to 13 different interpretations: (a) one interpretation in which all other pairs $(a, u)$ were accepted ($acc$); (b) one in which all other pairs $(a, u)$ were rejected ($rej$); (c) one in which all other pairs $(a, u)$ were undecided ($undec$); and (d) ten more interpretations in which each of the other pairs $(a, u)$ was assigned a random value from the set $\{acc, rej, undec\}$, with equal probability.

The above interpretations were not created with a specific semantics in mind, and therefore, most of them would not satisfy the conditions for being admissible, complete, etc. In turn, this would mean that our verification tests would almost exclusively return a negative answer. For this reason, we decided to create some positive examples for each ADFD and verification algorithm, i.e., one or more interpretations that are conflict-free, admissible, complete, grounded, preferred, model and stable.

The task of finding an extension (resp. interpretation) for a given semantics $\sigma$ is also, in fact, an interesting computational problem in argumentation for its own sake (usually referred to as "**SE**-$\sigma$", or "give some extension", in competitions like ICCMA). Our work, therefore, tackles this problem as well.

We aimed to produce 3 interpretations for each ADFD and for each semantics, although in some cases that number was not attainable for various reasons (timeouts in the respective routines, or lack of further interpretations, e.g., for grounded).

We provide an overview of all the interpretations we created for each ADFD in Table 1. Note that, for brevity and clarity, we group up the ADFDs by ADFD "class" (i.e., the ADF they originate from), instead of listing all 60 different ADFDs (12 per ADFD

[4] The 4th International Competition on Computational Models of Argumentation (ICCMA-21), https://argumentationcompetition.org/2021/index.html, accessed: July 23, 2026.

**Table 1**
Number of created interpretations for each ADFD "class".

| interpretation type | ABA2AF | ABA2AF_acyc | grid | metro | traffic |
|---|---|---|---|---|---|
| all acc | 12 | 12 | 12 | 12 | 12 |
| all rej | 12 | 12 | 12 | 12 | 12 |
| all undec | 12 | 12 | 12 | 12 | 12 |
| random | 120 | 120 | 120 | 120 | 120 |
| conflict-free (cf) | 52 | 72 | 157 | 192 | 161 |
| admissible (ad) | 16 | 36 | 121 | 156 | 125 |
| complete (co) | 16 | 36 | 60 | 105 | 69 |
| grounded (gr) | 12 | 12 | 12 | 12 | 12 |
| preferred (pr) | 4 | 24 | 21 | 70 | 33 |
| model (mo) | 4 | 24 | 6 | 58 | 21 |
| stable (st) | 1 | 12 | 3 | 29 | 7 |
| total distinct | 208 | 228 | 310 | 348 | 317 |

class). As explained above, we were not always able to generate 3 interpretations for each ADFD and each semantics, hence the variety in the number of interpretations generated per semantics and ADFD class. Additionally, we have added up the numbers of interpretations of each semantics to that of the semantics that precede them, hierarchically (as per Proposition 3); for example, the 121 admissible interpretations of the "grid" ADFD class include the 60 complete ones. The bottom row displays the total number of distinct interpretations that we created for each ADFD class.

*Conflict-free.* To compute the conflict-free interpretations, we start from the "empty" interpretation (i.e., all $AEs$ are undec) and then iteratively add, one by one, random $AEs$ as $rej$ (by adding them as $rej$ instead of $acc$ we avoid the need to check each time whether compliance is violated). At the end of each iteration, we check if the interpretation is conflict-free. We end the process once we identify the desired number of conflict-free interpretations.

*Admissible.* To compute the admissible interpretations, we start from either the "empty" (all AEs are undec) or the grounded interpretation (depending on whether we want to identify admissible interpretations that are less or more informative than the grounded interpretation). We then apply a method that finds more informative interpretations that are admissible (Pseudocode 9), producing a new admissible interpretation with each iteration.

*Complete.* Similarly to the admissible interpretations, we start from the grounded interpretation, but this time we use a slightly modified version of Pseudocode 9 (which finds more informative admissible interpretations), in order to identify more informative complete ones.

*Grounded.* We computed the grounded interpretation for each ADFD using Pseudocode 7.

*Preferred.* To compute the preferred interpretations, we followed the same process as for complete interpretations, but instead of storing the complete interpretations in each iteration, we run the process iteratively until there exist no other more informative interpretations that are complete, thus reaching a preferred interpretation.

*Model.* Since the definition of model interpretations (Definition 12) does not depend on the notions of decisively accepting or rejecting $AEs$, it was easy to encode it using rules in a declarative language like ASP and then pass them, alongside the ADFD, to an ASP solver like Clingo.[5] The solver grounds and executes the rules, producing all answer sets for the given input. These answer sets correspond to the model interpretations.

*Stable.* To compute the stable interpretations, we go through all the model interpretations for each ADFD and check whether they are stable using Pseudocode 14.

### 7.2.3. Experimental setup

To experimentally evaluate our implementation, we ran the verification algorithms for each semantics over all the generated ADFDs and interpretations. For each case, we made five runs and averaged the results. We do not discriminate between "hot" and "cold" runs, as there was no significant runtime difference in the two cases. We conducted the experiments on a 64bit Windows 10 Pro-computer with an Intel(R) Core(TM) i5-10600 3.30GHz CPU and 32GB of RAM, and set an enforced timeout of 240 seconds per run. The clingo version we used was 5.4.0.

[5] Potassco, the Potsdam Answer Set Solving Collection, University of Potsdam, https://potassco.org/clingo, accessed: August 12, 2025.

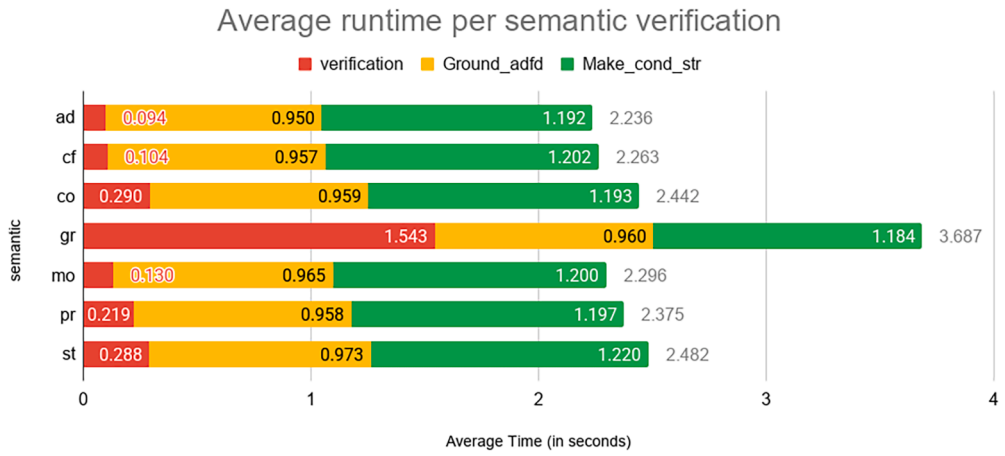



**Fig. 5.** Average runtime per semantics verification.

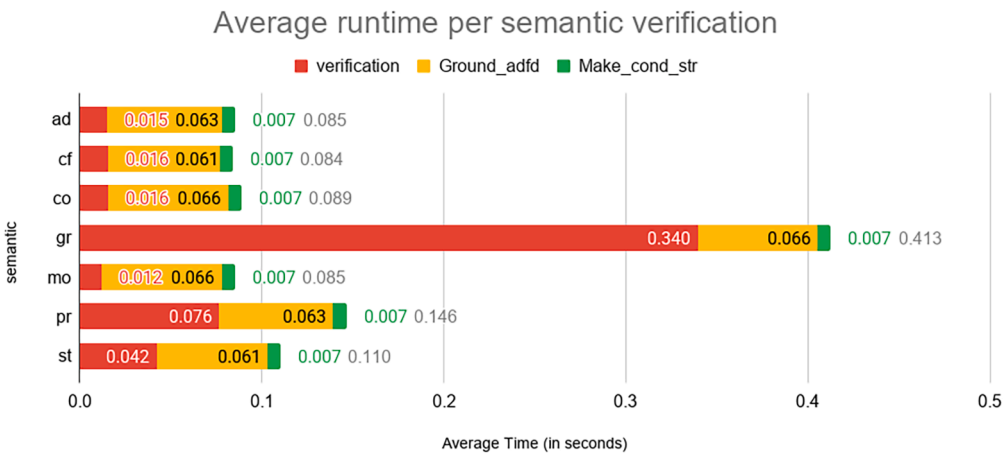



**Fig. 6.** Average runtime per semantics verification (small only).

### *7.2.4. Experimental results*

Our experiments quantified the effect of different parameters (namely, the semantics, the number of elements, the domain assignment probability, the ADFD classes, and the number of AEs in the domain) on the total runtime of our verification algorithms. The main conclusion from our experiments is that the most important factor with respect to runtime is the complexity of the acceptance conditions (as determined by the ADFD class, i.e., metro, grid, ABA2AF etc.).

In the figures below, the runtime is split into 3 different parts, each corresponding to a different stage of the process, to demonstrate the contribution of each of these stages in the final runtime. In particular, the runtime of the verification algorithms themselves is shown in red, the ADFD grounding process (happening during the input parsing) is shown in yellow, whereas the time needed for the transformation of the ASP condition rules into strings is shown in green. The total runtime (sum of the above) appears in gray to the right of each bar.

*Comparing semantics verification.* Among the verification algorithms, our expectation was that the algorithm for the preferred semantics would exhibit the worst performance, because of the inherent complexity of the definition of preferred interpretations. However, the results (in Figs. 5 and 6) show that it is the grounded semantics verification that takes longer than the rest.

The explanation is two-fold. First, for the bigger ADFDs (ABA2AF and ABA2AF_acyc) some timeouts occurred during the experiments for the preferred verification, which would otherwise have raised the average runtime for this semantics significantly. Second, in our $is_grounded()$ method (Pseudocode 6) we always (i.e., for every input interpretation) construct the grounded interpretation before comparing it with the input interpretation. This means that the time required is almost the same, regardless of whether the answer to the verification problem is positive or negative. In contrast, the verification algorithms for all other semantics have exit points that cause an early termination when it is already known that the input interpretation is not of the specified semantics (i.e., when the verification fails). Given that many of the input interpretations are created randomly, or do not satisfy the respective semantics, this case happens often, and lowers the average time. In other words, in grounded semantics the worst-case scenario essentially coincides

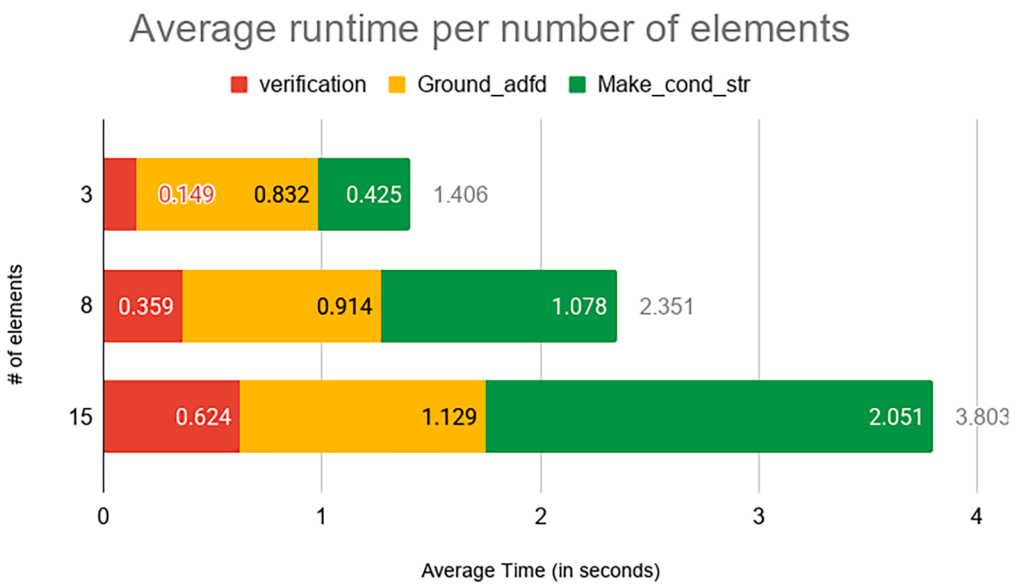



**Fig. 7.** Average runtime per number of elements.

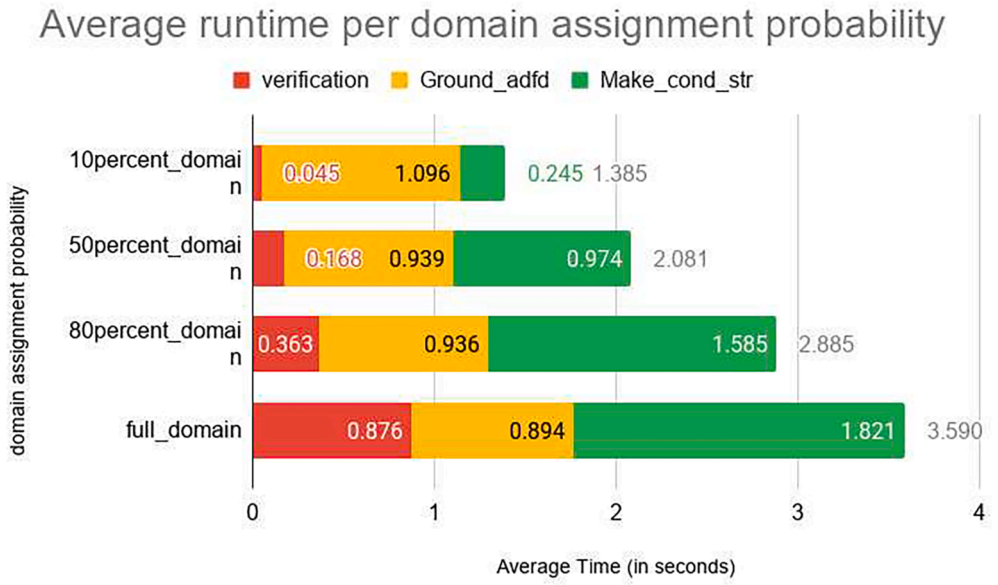



**Fig. 8.** Average runtime per domain assignment probability.

with the best-case scenario, whereas in other semantics the best-case scenario is significantly faster than the worst-case one, lowering the average.

*Comparing number of elements and domain assignment probability.* In Figs. 7 and 8 we see the average runtime in relation to the size of the universe and the domain assignment probability, respectively. Recall that the domain assignment probability is described in Section 7.2.1.

As one would expect, the runtime is affected by both parameters. Specifically, the runtime is increasing with the number of elements in the universe, and with the number of $AEs$ in the domain.

*Comparing ADFDs.* In order to compare the complexity of the conditions of ADFDs we use a simple metric, which quantifies this complexity using a single number, i.e., the total number of atoms in the bodies of all conditions in the ADFD. Fig. 9 shows the average runtime per ADFD class (with the number of arguments and the condition complexity appearing inside parentheses for each class). Evidently, the runtime increases with the complexity of the conditions of the ADFDs.

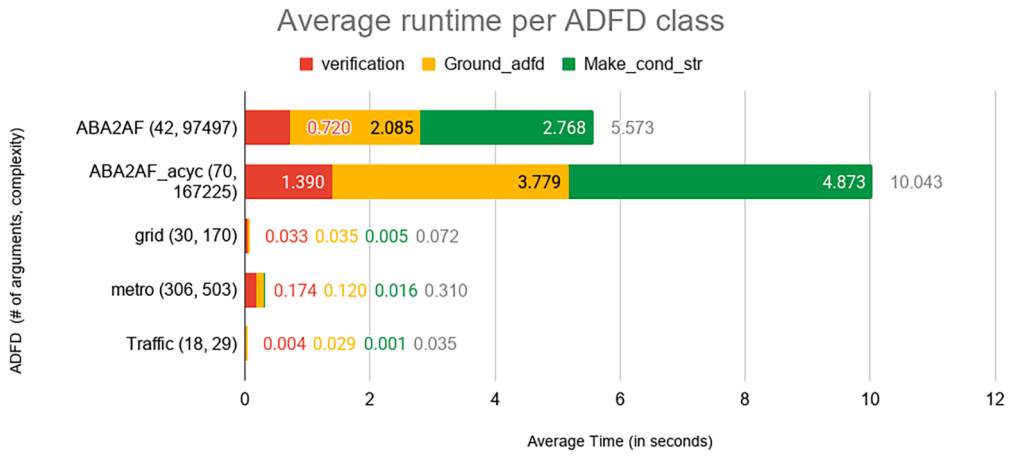



**Fig. 9.** Average runtime per ADFD class.

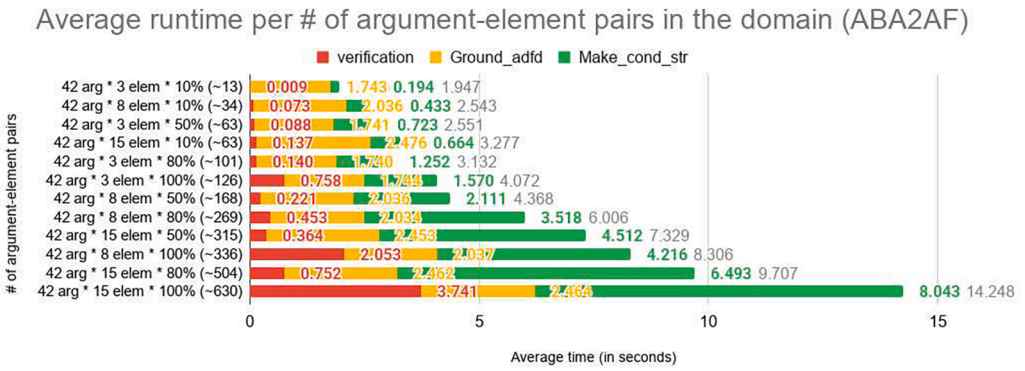



**Fig. 10.** Average runtime per # of $AEs$ in the domain (ABA2AF).

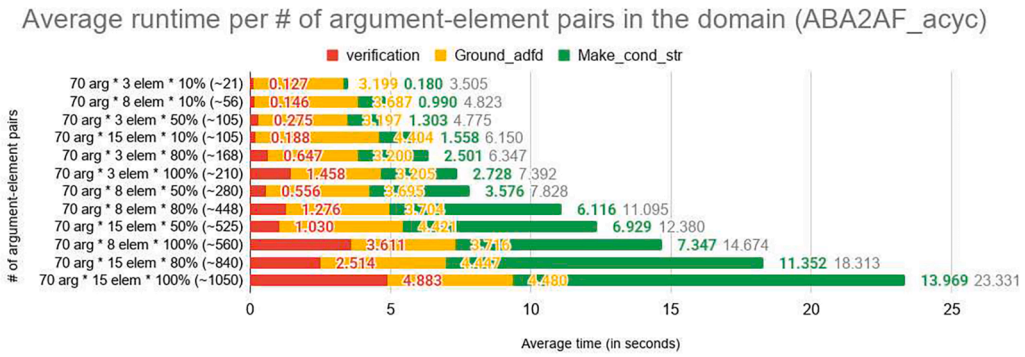



**Fig. 11.** Average runtime per # of $AEs$ in the domain (ABA2AF_acyc).

Another interesting observation is that the number of arguments in the ADFDs does not seem to affect the runtime to the same degree as the complexity, as is apparent by the *metro* ADFD class, which has many more arguments than all the other ADFD classes (namely, 308) but still takes substantially less time than the more complex *ABA2AF* and *ABA2AF_acyc* (with 42 and 70 arguments respectively).

*Comparing number of $AEs$ in domain.* The following figures (Figs. 10–14) show the average runtime per number of $AEs$ that are in the domain, with each figure corresponding to one of our five ADFD classes.

We see, in each figure, that just the total number of $AEs$ (number of arguments × number of elements) is not as important as the number of $AEs$ that are in the domain (number of arguments × number of elements × domain assignment probability). As the number of $AEs$ in the domain increases, so does the average runtime, quite consistently. Just like in Fig. 9, where we showed that

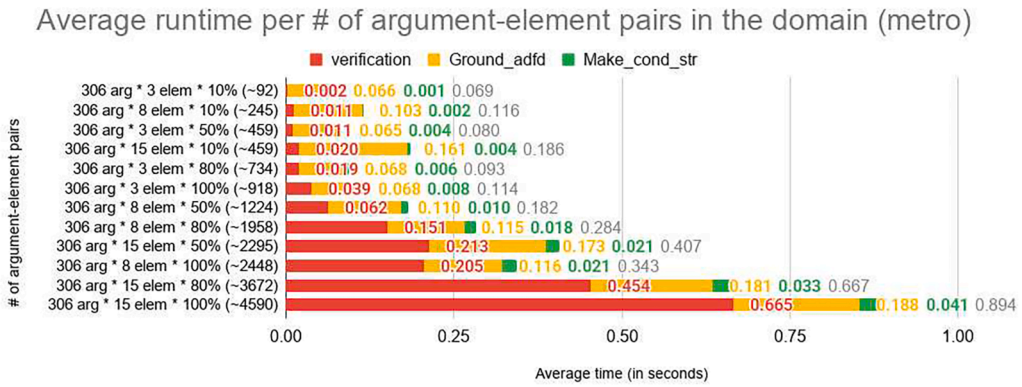



**Fig. 12.** Average runtime per # of $AEs$ in the domain (metro).

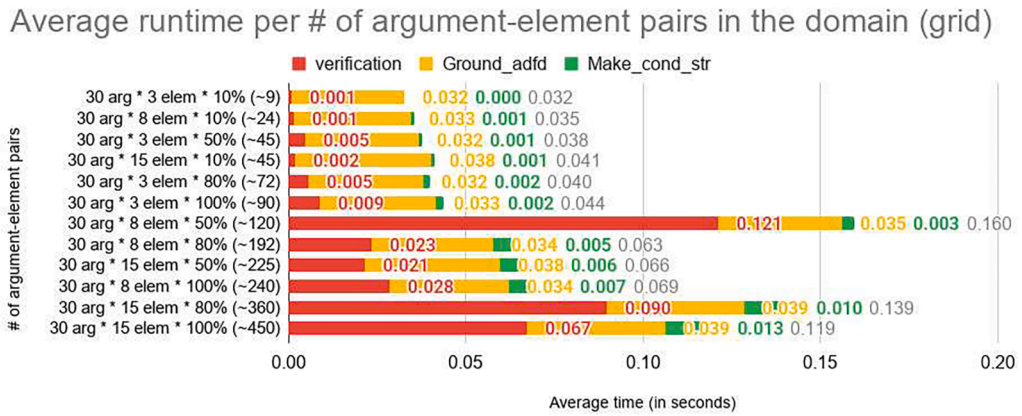



**Fig. 13.** Average runtime per # of $AEs$ in the domain (grid).

the total number of $AEs$ in an ADFD is less important compared to the complexity of the ADFD's conditions, here we show that the total number of $AEs$ is similarly less important compared to the number of $AEs$ that are in the domain.

Another interesting observation regarding Figs. 10 and 11 is that, due to the high complexity of the conditions in ABA2AF and ABA2AF_acyc, the transformation of the conditions into strings (*make_cond_str*) takes up a significant part of the total runtime.

As one can observe in Fig. 13, there is an outlier that corresponds to the ADFD *grid_8elem_50percent*. The outlier occurs when verifying whether or not the grounded, the stable and three complete interpretations are preferred. Certainly, the preferred verification would take longer for interpretations that are complete, as it wouldn't simply return `false` at the start, when checking for completeness (see Pseudocode 8), but that doesn't explain why it happens in this particular ADFD instance over others. Upon closer examination of the experiments' output, it turned out that the main bulk of that excess time in this particular case is the deepcopy method by which we store the data and reset them at the end of each loop (see Pseudocode 10). We, therefore, conclude that the outlier occurred due to the structure of those interpretations combined with the randomness by which this particular ADFD's domain was created.

Last but not least, in Fig. 14, we observe that due to the *traffic* ADFD class being very small and very simple (only 18 arguments, and a condition complexity of 29, as shown in Fig. 9), the average total runtime is very small overall (around 0.03-0.04 seconds) and doesn't fluctuate much. Furthermore, the ADFD grounding process takes up the predominant share of the runtime. However, if we turn our focus to the runtime of the verification method (red bar), we can see that the effects we observed earlier hold in this case as well.

In summary, the experiments confirm that the verification algorithms behave as expected with respect to the structural parameters of the input (universe size, domain probability, condition complexity), and they provide useful intuition about which factors dominate runtime in practice; specifically, that the most important factor is the complexity of the acceptance conditions. We note that all benchmarks were derived by augmenting existing ADF instances with synthetically generated domains; since no independent ADFD datasets currently exist, the generalisability of these findings to real-world ADFD deployments remains to be established. We also note that the main aim of our implementation and experiments was to provide a first empirical evaluation of ADFDs. A comparison with alternative approaches, e.g., translating ADFDs to ADFs and invoking an existing ADF solver, would give clearer insights into the efficiency of our implementation, and is an important direction for future work.

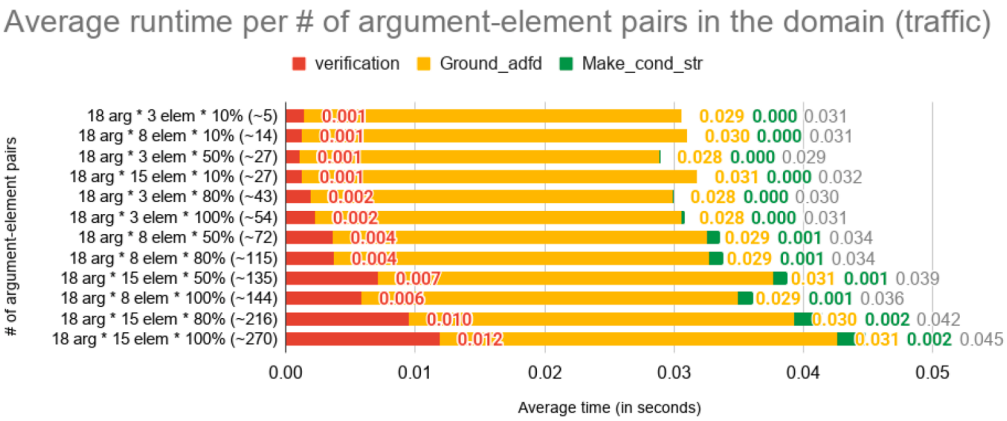



**Fig. 14.** Average runtime per # of $AEs$ in the domain (traffic).

## 8. Conclusion and outlook

We proposed a new argumentation formalism, called *Abstract Dialectical Frameworks with Domain Assignments (ADFDs)*, which allows the association of each argument with a domain of application (in the spirit of AAFDs [7]) and with an arbitrarily complex acceptance condition (in the spirit of ADFs [6]). Essentially, ADFDs are the fusion of AAFDs and ADFs, resulting in a representationally richer formalism that enables a more intuitive representation of complex argumentative scenarios that require entity-dependent reasoning .

We defined ADFDs in two alternative and equivalent ways. The first (ADFD specifications) employs logical expressions to model the domain, and thus it is intuitively clearer, more succinct and easier to understand and debug by the modeler. The second (ADFD concretizations) is essentially derived from the ADFD specifications by "distilling" (or grounding) the meaning of the logical expressions into a more concrete structure consisting of sets and functions. This alternative formalization allows the definition of ADFD semantics in a clear set-theoretic manner. Using our set-theoretic definition of semantics, we showed several results related to the properties of ADFD interpretations, formalized the relation of ADFDs with other frameworks to determine their expressive power, and showed some complexity results.

We argue that ADFDs have a lot of potential for modeling complex situations and can be viewed as a highly expressive addition in the arsenal of a modeler who is faced with complex argumentation scenarios. They can even play the role of a "unifying framework for argumentation semantics", as many of the argumentation frameworks proposed in the literature can be modeled as special cases of ADFDs. Many of these reductions stem directly from the fact that ADFs themselves are a special case of ADFDs; however, we claim that ADFDs can incorporate also other frameworks, such as Incomplete Abstract Argumentation Frameworks [33], Value-based Argumentation Frameworks [34] and others. A formal analysis of these relationships is among our plans for future work.

Moreover, as explained in the introduction, ADFDs have the potential to constitute a formalism that "bridges" work on structured and abstract argumentation, as they use a single model that supports both argument construction from domain knowledge and argument evaluation: the background knowledge ($\Pi$) plays the role of the knowledge base, the domain rules ($\Delta$) control which argument-entity pairs are instantiated (mimicking the argument-construction stage), and the acceptance conditions ($\Gamma$) encode the abstract evaluation, all within one uniform ASP program.

Aiming to empirically evaluate ADFDs, we also presented an ASP-Python implementation focusing on the verification problem. In the future, we plan to compare our current implementation with a pure ASP alternative. Given the ELP-based definition of ADFDs (ADFD specifications), an existing ASP encoding of ADFs (e.g., Ellmauthaler and Strass [31], Brewka et al. [32]) could be reused or lightly adapted for ADFDs, e.g., by adding the ADFD concretization as a preprocessing step to generate the relevant argument-element pairs, and replacing the propositional argument identifiers of standard ADF encodings with the grounded argument-element pairs of ADFDs.

Given the growing interest that ADFs have attracted not only as a theoretical framework but also as a practical tool for real-world reasoning tasks in various domains such as legal reasoning [35,36], product review analysis [37], and text exploration [38], we believe that the representational advantages of ADFDs open further opportunities for real-world impact. In particular, the ability to associate arguments with domains of application makes ADFDs a natural fit for policy and regulatory reasoning, as illustrated by the running example throughout this paper.

## CRediT authorship contribution statement

**Dimitris Danis:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation; **Giorgos Flouris:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Investigation, Formal analysis, Conceptualization; **Antonis Bikakis:** Writing – review & editing, Writing – original

draft, Project administration, Methodology, Investigation, Formal analysis, Conceptualization; **Theodore Patkos:** Writing – review & editing, Investigation, Formal analysis; **Nick Bassiliades:** Writing – review & editing, Investigation, Formal analysis.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Research Link Provided.

## Appendix A. Proofs of results

In this appendix, we provide the proofs of all propositions that appear in the main text.
We start with a lemma, which is useful for proving Proposition 1:

**Lemma 1.** *Consider two ADFs* $\mathcal{F}^1_{ADF} = \langle \mathcal{A}^1, C^1 \rangle$, $\mathcal{F}^2_{ADF} = \langle \mathcal{A}^2, C^2 \rangle$ *such that:*

- $\mathcal{A}^1 \subseteq \mathcal{A}^2$
- *For all* $a \in \mathcal{A}^1$, $S \subseteq \mathcal{A}^1$, $C^1_a(S) = C^2_a(S)$
- *For all* $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$, $S \subseteq \mathcal{A}^2$, $C^2_a(S) = \mathbf{f}$

*Consider also two ADF interpretations,* $v_1 : \mathcal{A}^1 \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$, $v_2 : \mathcal{A}^2 \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ *such that* $v_1(a) = v_2(a)$ *for all* $a \in \mathcal{A}^1$ *and* $v_2(a) = \mathbf{f}$ *for all* $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$. *Then:*

1. $v_1$ *is complete in* $\mathcal{F}^1_{ADF}$ *if and only if* $v_2$ *is complete in* $\mathcal{F}^2_{ADF}$.
2. $v_1$ *is grounded in* $\mathcal{F}^1_{ADF}$ *if and only if* $v_2$ *is grounded in* $\mathcal{F}^2_{ADF}$.

**Proof of Lemma 1.** We will start with some notations and observations that will be useful in the following.
Consider an ADF interpretation $w_1$ for $\mathcal{F}^1_{ADF}$ and an ADF interpretation $w_2$ for $\mathcal{F}^2_{ADF}$. We write $w_1 \leftrightarrow w_2$ if and only if $w_1(a) = w_2(a)$ for all $a \in \mathcal{A}^1$ and $w_2(a) = \mathbf{f}$ for all $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$.

It is easy to note that for each ADF interpretation $w_1$ of $\mathcal{F}^1_{ADF}$ there is a unique ADF interpretation $w_2$ of $\mathcal{F}^2_{ADF}$ such that $w_1 \leftrightarrow w_2$; and vice-versa i.e., for each ADF interpretation $w_2$ of $\mathcal{F}^2_{ADF}$ there exists a unique ADF interpretation $w_1$ of $\mathcal{F}^1_{ADF}$ such that $w_1 \leftrightarrow w_2$.

Moreover, we observe that, if $w_1 \leftrightarrow w_2$, and $\hat{w}_1$ is a completion of $w_1$, then there exists a unique completion of $w_2$, say $\hat{w}_2$ such that $\hat{w}_1 \leftrightarrow \hat{w}_2$; symmetrically, for every completion $\hat{w}_2$ of $w_2$ there exists a completion $\hat{w}_2$ of $w_1$ such that $\hat{w}_1 \leftrightarrow \hat{w}_2$.

Last but not least, if $w_1 \leftrightarrow w_2$, then $C^1_a(w_1) = C^2_a(w_2)$ for all $a \in \mathcal{A}^1$. This follows from our hypothesis regarding $C^1$ and $C^2$ and the fact that $w_2(a) = \mathbf{f}$ for all $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$ (i.e., there is no argument in $\mathcal{A}^2 \setminus \mathcal{A}^1$ that is mapped to $\mathbf{t}$.
Now for the actual problem at hand, we observe that $v_1 \leftrightarrow v_2$. Suppose initially that $v_1$ is complete. We will show that $v_2$ is complete, i.e., that $v_2 = \Gamma(v_2)$. Let us take some $a \in \mathcal{A}^2$. We consider the following cases:

- First, suppose that $a \in \mathcal{A}^1$ and $v_1(a) = \mathbf{t}$. Then $v_2(a) = \mathbf{t}$ (since $v_1 \leftrightarrow v_2$). Take any arbitrary completion $\hat{v}_2$ of $v_2$ and the respective (unique) completion $\hat{v}_1$ of $v_1$ such that $\hat{v}_1 \leftrightarrow \hat{v}_2$. Since $v_1$ is complete $C^1_a(\hat{v}_1) = v_1(a) = \mathbf{t}$; thus, per the observations above, $C^2_a(\hat{v}_2) = C^1_a(\hat{v}_1) = \mathbf{t} = v_2(a)$. Therefore, for any completion $\hat{v}_2$ of $v_2$ it follows that $v_2 = \Gamma(v_2)$, in this case.
- Now suppose that $a \in \mathcal{A}^1$ and $v_1(a) = \mathbf{f}$. We can use similar arguments to show that $C^2_a(\hat{v}_2) = v_2(a)$, i.e., $v_2 = \Gamma(v_2)$ in this case as well.
- Now consider the case that $a \in \mathcal{A}^1$ and $v_1(a) = \mathbf{u}$. Since $v_1$ is complete, there exist completions $\hat{v}^t_1, \hat{v}^f_1$ such that $C^1_a(\hat{v}^t_1) = \mathbf{t}$ and $C^1_a(\hat{v}^f_1) = \mathbf{f}$. We take the respective completions of $v_2$, namely $\hat{v}^t_2, \hat{v}^f_2$ such that $\hat{v}^t_1 \leftrightarrow \hat{v}^t_2$ and $\hat{v}^f_1 \leftrightarrow \hat{v}^f_2$. Using the above observations, we conclude that $C^2_a(\hat{v}^t_2) = \mathbf{t}$ and $C^2_a(\hat{v}^f_2) = \mathbf{f}$, thus $v_2 = \Gamma(v_2)$ in this case as well.
- Finally, we consider the case where $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$. In this case, by our hypotheses, $v_2(a) = \mathbf{f} = C^2_a(S)$ for all $S \subseteq \mathcal{A}^2$, and thus $v_2(a) = C^2_a(\hat{v}_2) = \mathbf{f}$ for all completions $\hat{v}_2$ of $v_2$. We conclude that $v_2 = \Gamma(v_2)$ in this case as well.

Summing up, it follows that $v_2 = \Gamma(v_2)$ in all cases, thus $v_2$ is complete.
For the opposite, suppose that $v_2$ is complete. We need to show that $v_1$ is complete. This can be done in a totally analogous manner (details are omitted).

For the second result, suppose that $v_1$ is grounded in $\mathcal{F}^1_{ADF}$. Then $v_1$ is complete in $\mathcal{F}^1_{ADF}$, so $v_2$ is complete in $\mathcal{F}^2_{ADF}$. Suppose that $v_2$ is not grounded. Then, there exists $v'_2$ such that $v'_2$ is complete in $\mathcal{F}^2_{ADF}$ and $v'_2 <_i v_2$, i.e., $v'_2(a) = v_2(a)$ whenever $v'_2(a) \neq \mathbf{u}$ and $v'_2 \neq v_2$, so there exists some $a \in \mathcal{A}^2$ such that $v'_2(a) = \mathbf{u}$ and $v_2(a) \neq \mathbf{u}$. So, suppose initially that there exists some $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$ such that $v'_2(a) = \mathbf{u}$. However, for any completion $\hat{v}'_2$ of $v'_2$ it holds that $C^2_a(\hat{v}'_2) = \mathbf{f} \neq v'_2(a)$, which means that $v'_2$ is not complete, a contradiction. Therefore, for all $a \in \mathcal{A}^2 \setminus \mathcal{A}^1$, $v'_2(a) = \mathbf{f}$. We conclude that there exists some $a \in \mathcal{A}^1$ such that $v'_2(a) = \mathbf{u}$ and $v_2(a) \neq \mathbf{u}$. So now take the interpretation $v'_1$ of $\mathcal{F}^1_{ADF}$ such that $v'_1 \leftrightarrow v'_2$. By the previous result, $v'_1$ is complete in $\mathcal{F}^1_{ADF}$ (given that $v'_2$ is complete in $\mathcal{F}^2_{ADF}$), and $v'_1 <_i v_1$, a contradiction by the fact that $v_1$ is grounded in $\mathcal{F}^1_{ADF}$. Thus, $v_2$ is grounded in $\mathcal{F}^2_{ADF}$.

For the opposite result we proceed analogously: suppose that $v_2$ is grounded in $\mathcal{F}^2_{ADF}$. Then, $v_1$ is complete in $\mathcal{F}^1_{ADF}$. Suppose that it is not grounded. Then, there exists $v'_1$ such that $v'_1$ is complete in $\mathcal{F}^1_{ADF}$ and $v'_1 <_i v_1$, i.e., there exists some $a \in \mathcal{A}^1$ such that $v'_1(a) = \mathbf{u}$ and $v_1(a) \neq \mathbf{u}$. Take the ADF interpretation $v'_2$ of $\mathcal{F}^2_{ADF}$ such that $v'_1 \leftrightarrow v'_2$. By the previous result, since $v'_1$ is complete in $\mathcal{F}^1_{ADF}$, it follows that $v'_2$ is complete in $\mathcal{F}^2_{ADF}$, and $v'_2 <_i v_2$, a contradiction by the fact that $v_2$ is grounded in $\mathcal{F}^2_{ADF}$. Thus, $v_1$ is grounded in $\mathcal{F}^1_{ADF}$. □

**Proof of Proposition 1.** We first observe that, for any two interpretations $S_1$, $S_2$, $S_1 \sqsubseteq S_2$ if and only if $v^{S_1} \leq_i v^{S_2}$.
Secondly, $S' \in Compl(S)$ if and only if $v^{S'}$ is a completion of $v^S$ in the respective ADF.
Now take some $S$ and set $\overline{v} = \Gamma(v^S)$. Then, it is easy to see that, for any $(\alpha, u) \in \mathcal{A} \times \mathcal{U}$:

- $(\alpha, u)$ is decisively-accepted by $S$ if and only if $\overline{v}(\alpha, u) = \mathbf{t}$.
- $(\alpha, u)$ is decisively-rejected by $S$ if and only if $\overline{v}(\alpha, u) = \mathbf{f}$.

A direct consequence of this fact is that, for any interpretation $S$, $v^{DEC(S)} = \Gamma(v^S)$. This proves the claim for $\sigma$ admissible and complete. The claim for grounded and preferred is a direct consequence of the above (on admissible and complete).

For $\sigma$ being model, note that $S$ is total if and only if $v^S$ is total. Therefore, given Definitions 12 and 17, it follows that:
$S$ is a model for $\mathcal{F} \Leftrightarrow$
$S$ is total and $(\alpha, u) \in S_{\text{ACC}}$ if and only if $u \in D(\alpha) \cap C(\alpha, S_{\text{ACC}}) \Leftrightarrow$
$v^S$ is total and $v^S(\alpha, u) = C^{\Phi}_{(\alpha,u)}(S_{\text{ACC}}) \Leftrightarrow$
$v^S$ is model for $\mathcal{F}^{\Phi}_{ADF}$.

Finally, for $\sigma$ being stable, we observe that, from the previous result, $S$ is a model for $\mathcal{F}$, if and only if $v^S$ is a model for $\mathcal{F}^{\Phi}_{ADF}$. Now consider the reduct $\mathcal{F}^S$, and its corresponding ADF, $\mathcal{F}^{S\Phi}_{ADF}$. Consider also the reduct of $\mathcal{F}^{\Phi}_{ADF}$ for $v^S$, namely $\mathcal{F}^{\Phi v^S}_{ADF}$. It is easy to see, by the respective definitions, that for $\mathcal{F}^{S\Phi}_{ADF} = \langle \mathcal{A}^{S\Phi}, \mathcal{C}^{S\Phi} \rangle$ and $\mathcal{F}^{\Phi v^S}_{ADF} = \langle \mathcal{A}^{\Phi v^S}, \mathcal{C}^{\Phi v^S} \rangle$, Lemma 1 applies. Thus: by the above results, $S$ is grounded in $\mathcal{F}^S$ if and only if $v^S$ is grounded in $\mathcal{F}^{S\Phi}_{ADF}$; by Lemma 1, and the fact that $v^S(a) = \mathbf{t}$ for all $a \in \mathcal{A}^{S\Phi}$ (by the construction of $\mathcal{F}^{S\Phi}_{ADF}$), we conclude that $v^S$ is grounded in $\mathcal{F}^{S\Phi}_{ADF}$ if and only if $w : \mathcal{A}^{\Phi v^S} \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ such that $w(a) = \mathbf{t}$ for all $a \in \mathcal{A}^{\Phi v^S}$ is grounded in $\mathcal{F}^{S\Phi}_{ADF}$. Combining these facts, the claim follows easily. □

**Proof of Proposition 2.** By Definition 5, we have that: $|\mathcal{F}| = |\mathcal{A}| + |\mathcal{U}| + |D| + |\mathcal{C}|$. By the definition of ADFs in Section 4.3, it follows that: $|\mathcal{F}^{\Phi}_{ADF}| = |\mathcal{A}^{\Phi}| + |\mathcal{C}^{\Phi}|$. Moreover, by Definition 17, it follows that: $|\mathcal{A}^{\Phi}| = |\mathcal{A}| \cdot |\mathcal{U}|$ and $|\mathcal{C}^{\Phi}| = |\mathcal{C}| + |D|$. Combining the above relations, the result follows trivially. □

**Proof of Proposition 3.** (1): Suppose that $S$ is fully compliant but not compliant, and take some $(\alpha, u) \in S_{\text{ACC}}$ such that $u \notin D(\alpha)$. Then, since $S$ is fully compliant, $(\alpha, u) \in S_{\text{REJ}}$ a contradiction because $(\alpha, u) \in S_{\text{ACC}}$.
(2): Take some $a \in \mathcal{A}$, $u \in \mathcal{U}$, such that $u \notin D(\alpha)$. Since $S$ is compliant, it is the case that $(\alpha, u) \notin S_{\text{ACC}}$, and thus, since $S$ is total, $(\alpha, u) \in S_{\text{REJ}}$, which implies that $S$ is fully compliant.
(3): Suppose that $S$ is conflict-free, and take some $(\alpha, u) \in S_{\text{ACC}}$. By Definition 11, it follows that $(\alpha, u) \notin DEC_{\text{REJ}}(S)$, so, by Definition 9, $u \in dom(\alpha)$, so $S$ is compliant.
(4): Take any $(\alpha, u) \in S_{\text{ACC}}$. Then, since $S$ is admissible, it follows that $(\alpha, u) \in DEC_{\text{ACC}}(S)$ and thus $(\alpha, u) \notin DEC_{\text{REJ}}(S)$, so $S_{\text{ACC}} \cap DEC_{\text{REJ}}(S) = \emptyset$. Analogously, if $(\alpha, u) \in S_{\text{REJ}}$, then $(\alpha, u) \in DEC_{\text{REJ}}(S)$ and thus $(\alpha, u) \notin DEC_{\text{ACC}}(S)$, so $S_{\text{REJ}} \cap DEC_{\text{ACC}}(S) = \emptyset$. We conclude that $S$ is conflict-free.
(5): Direct from Definition 13.
(6): Direct from Definition 13.
(7): Corollary of Proposition 1 and the results of Brewka and Woltran [3], Polberg [5], Brewka et al. [6].
(8): Corollary of Proposition 1 and the results of Brewka and Woltran [3], Polberg [5], Brewka et al. [6].
(9): Since $S$ is total, it follows that $Compl(S) = \{S\}$. Take $(\alpha, u) \in \mathcal{A} \times \mathcal{U}$.

If $(\alpha, u) \in S_{\text{ACC}}$ and $(\alpha, u) \in D(\alpha)$, then, since $S$ is admissible, $(\alpha, u)$ is decisively accepted. Since $S \in Compl(S)$, it follows that $u \in C(\alpha, S_{ACC})$.

If $(\alpha, u) \notin S_{\text{ACC}}$ or $(\alpha, u) \notin D(\alpha)$, then $(\alpha, u) \in S_{\text{REJ}}$ (because $S$ is total) or $(\alpha, u) \notin D(\alpha)$, and, since $S$ is admissible, $(\alpha, u)$ is decisively rejected. Since $S \in Compl(S)$, it follows that $u \notin C(\alpha, S_{\text{ACC}})$.
Therefore, $S$ is model.
(10): Direct from Definition 15.
(11): Take some $u \notin D(\alpha)$. By Definition 9, it follows that $(\alpha, u) \in DEC_{\text{REJ}}$. But $DEC_{\text{REJ}} = S_{\text{REJ}}$ by the fact that $S$ is complete and Definition 13, so $S$ is fully compliant. □

**Proof of Proposition 4.** Both results follow using Proposition 1 and previous results on ADFs ([3,5,6]). □

**Proof of Proposition 5.**
1. Direct from Definition 17.
2. Direct from Definition 3.
3. Direct from Definitions 16 and 3.
4. Direct from Definition 16.
5. Direct from Definition 16. □

## Appendix B. Algorithms

In this appendix we provide the pseudocode used for the implementation of the algorithms described in Section 7.1.

**Pseudocode 1** $is_compliant()$.

**Input:** an interpretation $S$
**Output:** if $S$ is compliant **true** , otherwise **false**

```
for all AEs in S do
    if AE is acc and AE ∉ domain then
        return false
    end if
end for
return true
```

**Pseudocode 2** $is_fully_compliant()$.

**Input:** an interpretation $S$
**Output:** if $S$ is compliant **true** , otherwise **false**

```
for all AEs in S do
    if AE ∉ domain and AE is not rej then
        return false
    end if
end for
return true
```

**Pseudocode 3** $is_conflictfree()$.

**Input:** a compliant interpretation $S$
**Output:** if $S$ is conflict-free **true** , otherwise **false**

```
for all AEs in S do
    if AE is undec or AE ∉ domain then
        continue
    end if
    E ← evaluate AE's acceptance condition under S //determine whether AE is dec. acc/rej
    if AE is rej and E = true then
        return false //AE is rej but dec. acc
    end if
    if AE is acc and E = false then
        return false //AE is acc but dec. rej
    end if
end for
return true
```

**Pseudocode 4** *is_admissible*().

**Input:** a compliant interpretation $S$
**Output:** if $S$ is admissible **true** , otherwise **false**

**for all** $AEs$ in $S$ **do**
  **if** $AE$ is $undec$ **or** $AE \notin domain$ **then**
    **continue**
  **end if**
  $E \leftarrow$ evaluate AE's acceptance condition under S //determine whether $AE$ is dec. acc/rej
  **if** $AE$ is $rej$ **and** $E = true$ **then**
    **return false** //$AE$ is rej but dec. acc
  **end if**
  **if** $AE$ is $acc$ **and** $E = false$ **then**
    **return false** //$AE$ is acc but dec. rej
  **end if**
  **if** $E$ is $unknown$ **then**
    **return false** //$AE$ is **not** undec, but is neither dec. acc nor dec. rej
  **end if**
**end for**
**return true**

**Pseudocode 5** *is_complete*().

**Input:** a compliant interpretation $S$
**Output:** if $S$ is complete **true** , otherwise **false**

**for all** $AEs$ in $S$ **do**
  **if** $AE \notin domain$ **then**
    **if** $AE$ is $undec$ **then**
      **return false** //pair is undec but not in the domain, thus dec. rej
    **else**
      **continue** //$AE$ is rej (since if it were acc it would fail the compliancy check earlier on)
    **end if**
  **end if**
  $E \leftarrow$ evaluate AE's acceptance condition under S //determine whether $AE$ is dec. acc/rej
  **if** $AE$ is **not** $acc$ **and** $E = true$ **then**
    **return false** //$AE$ is not acc but dec. acc
  **end if**
  **if** $AE$ is **not** $rej$ **and** $E = false$ **then**
    **return false** //$AE$ is not rej but dec. rej
  **end if**
  **if** $AE$ is **not** $undec$ **and** $E$ is $unknown$ **then**
    **return false** //$AE$ is not undec, but is neither dec. acc nor dec. rej
  **end if**
**end for**
**return true**

**Pseudocode 6** *is_grounded*().

**Input:** a compliant interpretation $S$
**Output:** if $S$ is grounded **true** , otherwise **false**

**if not** *is_complete*(*interpretation*) **then**
  **return false**
**end if**
let $S'$ be the grounded interpretation we get from *compute_grounded*()
**if** $S = S'$ **then**
  **return true**
**else**
  **return false**
**end if**

**Pseudocode 7** *compute_grounded*().

**Input:** -
**Output:** the grounded interpretation $S'$

let $S'$ be the empty interpretation (i.e., all $AEs$ are $undec$)
**for all** $AEs$ in $S'$ **do**
  **if** $AE \notin domain$ **then**
    $AE \leftarrow rej$
  **end if**
**end for**
**repeat**
  find the dec. acc and dec. rej $AEs$, for the current $S'$, by evaluating their condition
  **for all** dec. acc $AEs$ **do**
    $AE \leftarrow acc$
  **end for**
  **for all** dec. rej $AEs$ **do**
    $AE \leftarrow rej$
  **end for**
**until** $S'$ does not change
**return** $S'$

**Pseudocode 8** *is_preferred*().

**Input:** a compliant interpretation $S$
**Output:** if $S$ is preferred **true** , otherwise **false**

**if not** *is_complete*(*interpretation*) **then**
  **return false**
**end if**
mark all $AEs$ in $S$ that are acc or rej as `fixed`, and the undec as **not** `fixed`
create the dependency graph of the $AEs$, to find their ancestors and descendants.
find the cycles in the dependency graph
**for all** $AEs$ in $S$ that are in a cycle and are **not** `fixed` **do**
  mark $AE$ as `fixed`
  $AE \leftarrow acc$
  **if** *exists_more_inf_ad*($AE$) **then**
    **return false**
  **end if**
  $AE \leftarrow rej$
  **if** *exists_more_inf_ad*($AE$) **then**
    **return false**
  **end if**
  $AE \leftarrow undec$
**end for**
**return true**

**Pseudocode 9** *exists_more_inf_ad*().

**Input:** an $AE$ $V$
**Output:** **true** or **false**

```
1: let TDL be an empty queue, and savestate_collection be an empty stack
2: vec ← null
3: queue V into TDL
4: repeat
5:    let W be the first AE in TDL
6:    E ← evaluate W's acceptance condition under the current partial interpretation //determine whether W is dec. acc/rej
7:    if (W is acc and E = false) or (W is rej and E = true) then
8:       if savestate_collection is empty then
9:          return false
10:      else
11:         S ← savestate_collection.pop()
12:         "load" the savestate S
13:         goto line 38
14:      end if
15:   end if
16:   if (W is acc and E = true) or (W is rej and E = false) then
17:      dequeue W from TDL
18:      vec ← null
19:      goto line 38
20:   end if
21:   if E is unknown then
22:      if check_ancestors(W, TDL, vec) then
23:         save the ancestors combination vec, the queue TDL, and the values of AEs into a savestate S'
24:         insert S' into savestate_collection
25:         dequeue W from TDL
26:         vec ← null
27:         goto line 38
28:      else
29:         if savestate_collection is empty then
30:            return false
31:         else
32:            S ← savestate_collection.pop()
33:            "load" the savestate S
34:            goto line 38
35:         end if
36:      end if
37:   end if
38: until TDL is empty
39: return true
```

**Pseudocode 10** *check_ancestors*().

**Input:** an $AEV$, a queue $TDL$, a vector $in_vec$, a list of vectors $good_combs_for_V_ancestors$
**Output:** **true** or **false**

1: $N \leftarrow length(ancestors(V))$
2: **if** $in_vec = [rej, rej, \dots, rej, rej]$ **then**
3: **return false**
4: **end if**
5: **if** $in_vec = null$ **then**
6: $vec \leftarrow [undec, undec, \dots, undec, undec]$ //of size $N$
7: **else**
8: $vec \leftarrow step1(in_vec)$
9: **end if**
10: **repeat**
11: **for** $i = 0$ to $N - 1$ **do**
12: $V_i \leftarrow ancestors[V][i]$
13: **if** $vec[i] \neq V_i$ **and** $V_i$ is `fixed` **then**
14: **goto** line 35
15: **end if**
16: **if** $vec[i] = acc$ **and** $V_i \notin domain$ **then**
17: **goto** line 35
18: **end if**
19: $V_i \leftarrow vec[i]$
20: **end for**
21: $E \leftarrow$ evaluate V's acceptance condition under the current partial interpretation //determine whether $V$ is dec. acc/rej
22: **if** ($V$ is $acc$ **and** $E = true$) **or** ($V$ is $rej$ **and** $E = false$) **then**
23: **if** $vec$ is **not** a superset of a vector in $good_combs_for_V_ancestors$ **then**
24: **for all** ancestors $W$ of $V$ **do**
25: **if** $W$ is **not** $undec$ **and** $W$ is **not** `fixed` **then**
26: mark $W$ as `fixed`
27: queue $W$ into $TDL$
28: **end if**
29: **end for**
30: $in_vec \leftarrow vec$
31: **return true**
32: **end if**
33: **end if**
34: reset `fixed`marks and values for $AEs$ to those before the loop
35: $vec \leftarrow step1(vec)$
36: **until** $vec = [undec, undec, \dots, undec, undec]$
37: $vec \leftarrow null$
38: **return false**

**Pseudocode 11** *step*1().

**Input:** a vector *in_vec*
**Output:** the next vector in line, or the $[undec, undec, ., undec, undec]$ vector, if there is no next

$N \leftarrow length(in_vec)$
$non_undec_ids \leftarrow$ the indexes of *in_vec* where the value is **not** *undec*
$d \leftarrow length(non_undec_ids)$ // current depth
$vec \leftarrow mini_step1(in_vec, N-1)$
**if** $vec \neq null$ **then**
  **return** *vec*
**end if**
$combinations_gen \leftarrow combinations(N, d)$
get to the combination *comb* in *combinations_gen* that is the same as *non_undec_ids*
$next \leftarrow next(comb)$
**if** *next* exists **then**
  $vec \leftarrow [undec, undec, \ldots, undec, undec]$
  **for** the indices $j$ in *next* **do**
    $vec[j] \leftarrow acc$
  **end for**
  **return** *vec*
**end if**
$d \leftarrow d + 1$
**if** $d > N$ **then**
  $vec \leftarrow [undec, undec, ., undec, undec]$
  **return** *vec*
**end if**
$combinations_gen \leftarrow combinations(N, d)$
$next_comb \leftarrow next(combinations_gen)$
$vec \leftarrow [undec, undec, \ldots, undec, undec]$
**for** the indices $j$ in *next_comb* **do**
  $vec[j] \leftarrow acc$
**end for**
**return** *vec*

**Pseudocode 12** *mini_step*1().

**Input:** a vector *in_vec*, an integer *pos*
**Output:** the next vector in line, or *null* , if there is no next

**if** $pos < 0$ **then**
  **return** *null*
**end if**
**if** $in_vec[pos] = undec$ **then**
  **return** $mini_step1(in_vec, pos-1)$
**end if**
**if** $in_vec[pos] = acc$ **then**
  $in_vec[pos] \leftarrow rej$
  **return** *in_vec*
**end if**
**if** $in_vec[pos] = rej$ **then**
  $in_vec[pos] \leftarrow acc$
  **return** $mini_step1(in_vec, pos-1)$
**end if**

**Pseudocode 13** *is_model*().

**Input:** an interpretation $S$
**Output:** if the interpretation is model **true** , otherwise **false**

**for all** $AEs$ in the $S$ **do**
  **if** $AE$ is $undec$ **then**
    **return false** //$S$ is not total
  **end if**
  **if** $AE$ is $acc$ **then**
    **if** $AE \notin domain$ **then**
      **return false**
    **end if**
    $E \leftarrow$ evaluate AE's acceptance condition under S
    **if** $E \neq true$ **then**
      **return false**
    **end if**
  **end if**
  **if** $AE \in domain$ **then**
    $E \leftarrow$ evaluate AE's acceptance condition under S
    **if** $E = true$ **then**
      **if** $AE$ is **not** acc **then**
        **return false**
      **end if**
    **end if**
  **end if**
**end for**
**return true**

**Pseudocode 14** *is_stable*().

**Input:** an interpretation $S$
**Output:** if the interpretation is stable **true** , otherwise **false**

**if not** $is_model(S)$ **then**
  **return false**
**end if**
let *reduct_domain* be an initially empty list for the domain of the reduct ADFD
**for all** $AEs$ in the $S$ **do**
  **if** $AE$ is acc **then**
    **if** $AE \in domain$ **then**
      add $AE$ into *reduct_domain*
    **end if**
  **end if**
**end for**
use *reduct_domain* as the domain
**if** $is_grounded(S)$ **then**
  **return true**
**else**
  **return false**
**end if**

## References


[1] P. Baroni, D. Gabbay, M. Giacomin, L. van der Torre (Eds.), Handbook of Formal Argumentation, vol. 1, College Publications, 2018.
[2] P.M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and N-person games, Artif. Intell. 77 (2) (1995) 321–357.
[3] G. Brewka, S. Woltran, Abstract Dialectical Frameworks, in: Twelfth International Conference on the Principles of Knowledge Representation and Reasoning, 2010.
[4] G. Brewka, H. Strass, S. Ellmauthaler, J.P. Wallner, S. Woltran, Abstract Dialectical Frameworks revisited, in: Twenty-Third International Joint Conference on Artificial Intelligence, 2013.
[5] S. Polberg, Revisiting Extension–Based Semantics of Abstract Dialectical Frameworks, Technical Report DBAI-TR-2015-88, DBAI, TUWIEN, 2015.
[6] G. Brewka, S. Ellmauthaler, H. Strass, J.P. Wallner, S. Woltran, Abstract Dialectical Frameworks, in: P. Baroni, D. Gabbay, M. Giacomin, L. van der Torre (Eds.), Handbook of Formal Argumentation, 2018, pp. 237–286.
[7] A. Vassiliades, T. Patkos, G. Flouris, A. Bikakis, N. Bassiliades, D. Plexousakis, Abstract argumentation frameworks with domain assignments, in: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI-21), 2021.

[8] M. Gelfond, H. Przymusinska, Towards a theory of elaboration tolerance: logic programming approach, Int. J. Softw. Eng. Knowl. Eng. 6 (1996) 89–112. https://api.semanticscholar.org/CorpusID:15214488.
[9] G. Brewka, P. Dunne, S. Woltran, Relating the semantics of Abstract Dialectical Frameworks and standard AFs, in: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), 2011, pp. 780–785.
[10] H. Strass, Expressiveness of two-valued semantics for Abstract Dialectical Frameworks, J. Artif. Intell. Res. 54 (2015) 193–231.
[11] H. Prakken, An abstract framework for argumentation with structured arguments, Argum. Comput. 1 (2010) 93–124.
[12] G. Brewka, S. Woltran, GRAPPA: a semantical framework for graph-based argument processing, in: T. Schaub, G. Friedrich, B. O'Sullivan (Eds.), ECAI 2014 - 21st European Conference on Artificial Intelligence, 18–22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014), vol. 263 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2014, pp. 153–158. https://doi.org/10.3233/978-1-61499-419-0-153
[13] M.C.D. Budán, M.J. Gomez Lucero, G.R. Simari, An approach for temporal argumentation using labeled defeasible logic programming (l-DeLP), J. Comput. Sci. Technol. 12 (2012).
[14] M.C.D. Budán, M.L. Cobo, D.C. Martinez, G.R. Simari, Bipolarity in temporal argumentation frameworks, Int. J. Approx. Reason. 84 (2017) 1–22.
[15] M.C.D. Budán, M.L. Cobo, D.C. Martinez, G.R. Simari, Proximity semantics for topic-based abstract argumentation, Inf. Sci. 508 (2020) 135–153. https://doi.org/10.1016/j.ins.2019.08.037
[16] W. Dvorák, S. Woltran, Complexity of abstract argumentation under a claim-centric view, Artif. Intell. 285 (2020) 103290. https://doi.org/10.1016/j.artint.2020.103290
[17] A. Hunter, Non-monotonic reasoning in deductive argumentation, CoRR abs/1809.00858 (2018). http://arxiv.org/abs/1809.00858.
[18] P. Besnard, A. Hunter, Elements of Argumentation, MIT Press, 2008. https://mitpress.mit.edu/books/elements-argumentation.
[19] E. De Angelis, M. Proietti, F. Toni, ABA learning via ASP, in: E. Pontelli, S. Costantini, C. Dodaro, S.A. Gaggl, R. Calegari, A.S.d. Garcez, F. Fabiano, A. Mileo, A. Russo, F. Toni (Eds.), Proceedings 39th International Conference on Logic Programming, ICLP 2023, Imperial College London, UK, 9th July 2023–15th July 2023, vol. 385 of EPTCS, 2023, pp. 1–8. https://doi.org/10.4204/EPTCS.385.1
[20] M. Diller, S.A. Gaggl, P. Hanisch, G. Monterosso, F. Rauschenbach, Grounding rule-based argumentation using datalog, in: M. Ortiz, R. Wassermann, T. Schaub (Eds.), Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning, KR 2025, Melbourne, Australia, November 1–17, 2025, 2025. https://doi.org/10.24963/KR.2025/28
[21] J.A.A. Sillince, Shifts in focus and scope during argumentation, J. Pragmat. 24 (4) (1995) 413–431.
[22] M.L. Zubizarreta, The lexical encoding of scope relations among arguments, in: Syntax and the Lexicon, Brill, 1992, pp. 211–258.
[23] D. Zarefsky, Strategic maneuvering in political argumentation, Argumentation 22 (3) (2008) 317.
[24] S. Polberg, Extension-based semantics of Abstract Dialectical Frameworks, 2014, arXiv:1405.0406 [cs.AI]
[25] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (3/4) (1991) 365–386. https://doi.org/10.1007/BF03037169
[26] H. Strass, Approximating operators and semantics for Abstract Dialectical Frameworks, Artif. Intell. 205 (2013) 39–70.
[27] H. Strass, J.P. Wallner, Analyzing the computational complexity of Abstract Dialectical Frameworks via approximation fixpoint theory, Artif. Intell. (2015) 34–74.
[28] S. Polberg, Understanding the Abstract Dialectical Frameworks, in: 15th European Conference on Logics in Artificial Intelligence (JELIA-16), 2016.
[29] J. Pührer, Realizability of three-valued semantics for Abstract Dialectical Frameworks, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.
[30] V. Lifschitz, Answer Set Programming, Springer Cham, 2019.
[31] S. Ellmauthaler, H. Strass, The DIAMOND system for computing with Abstract Dialectical Frameworks, in: S. Parsons, N. Oren, C. Reed, F. Cerutti (Eds.), Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9–12, 2014, Frontiers in Artificial Intelligence and Applications, IOS Press, 2014, pp. 233–240. https://doi.org/10.3233/978-1-61499-436-7-233
[32] G. Brewka, M. Diller, G. Heissenberger, T. Linsbichler, S. Woltran, Solving advanced argumentation problems with answer set programming, Theory Pract. Log. Program. 20 (3) (2020) 391–431. https://doi.org/10.1017/S1471068419000474
[33] D. Baumeister, D. Neugebauer, J. Rothe, H. Schadrack, Verification in incomplete argumentation frameworks, Artif. Intell. J. 264 (2018) 1–26.
[34] T.J.M. Bench-Capon, Persuasion in practical argument using value-based argumentation frameworks, J. Log. Comput. 13 (3) (2003) 429–448.
[35] L. Al-Abdulkarim, K. Atkinson, T.J.M. Bench-Capon, A methodology for designing systems to reason with legal cases using Abstract Dialectical Frameworks, Artif. Intell. Law 24 (1) (2016) 1–49. https://doi.org/10.1007/S10506-016-9178-1
[36] J. Mumford, K. Atkinson, T.J.M. Bench-Capon, Combining a legal knowledge model with machine learning for reasoning with legal cases, in: M. Grabmair, F. Andrade, P. Novais (Eds.), Proceedings of the Nineteenth International Conference on Artificial Intelligence and Law, ICAIL 2023, Braga, Portugal, June 19–23, 2023, ACM, 2023, pp. 167–176. https://doi.org/10.1145/3594536.3595158
[37] A. Keshavarzi Zafarghandi, D. Ceolin, Categorizing review helpfulness using Abstract Dialectical Frameworks, in: T. Muller, C.F. Gago, D. Ceolin, E. Gudes, N. Gal-Oz (Eds.), Trust Management XIV - 14th IFIP WG 11.11 International Conference on Trust Management, IFIPTM 2023, Amsterdam, the Netherlands, October 18–20, 2023, Proceedings, vol. 694 of IFIP Advances in Information and Communication Technology, Springer, 2023, pp. 89–104. https://doi.org/10.1007/978-3-031-76714-2_6
[38] E. Cabrio, S. Villata, Abstract Dialectical Frameworks for text exploration, in: H.J. van den Herik, J. Filipe (Eds.), Proceedings of the 8th International Conference on Agents and Artificial Intelligence (ICAART 2016), Volume 2, Rome, Italy, February 24–26, 2016, SciTePress, 2016, pp. 85–95. https://doi.org/10.5220/0005699100850095