ABA Disputes in ASP: Advancing Argument Games through Multi-Shot Solving

Martin Diller[1], Piotr Gorczyca[2]

[1]Logic Programming and Argumentation Group, TU Dresden, Germany
[2]Computational Logic Group, TU Dresden, Germany

Abstract

Argumentation games, which model reasoning as adversarial dialogue, offer intuitive and explainable mechanisms for decision-making in AI. However, their implementation has lagged behind inference-focused approaches, particularly in structured argumentation frameworks like assumption-based argumentation (ABA). This work presents, to our knowledge, the first application of multi-shot answer set programming (ASP) for implementing argument games, focusing on ABA dispute derivations. Leveraging a recent rule-based representation of ABA disputes, our method combines a declarative program with lightweight script-based control of multi-shot aspects, yielding a modular and adaptable system. We extend this core approach to support alternative games and show how it can also be used to implement argument games for Dung's abstract argumentation formalism. Empirical results show that our implementation outperforms existing ABA dispute systems. We also introduce an approximate variant that further improves efficiency – reaching the level of the best current inference-focused ABA system – while maintaining perfect specificity (i.e. true negative rate), demonstrating the practical value of multi-shot ASP, particularly in interactive settings where explainability is key.

Keywords

Assumption-based argumentation, Abstract argumentation, Argument games, Multi-shot answer set programming

1. Introduction

Argumentation plays a crucial role in human decision-making, especially in complex situations without clear-cut answers. *Formal models of argumentation*, rooted in research in non-monotonic reasoning, offer argumentation approaches to knowledge representation and reasoning in AI. These models underpin a wide range of applications including in law, medicine, and e-governance.

Models of argumentation range from highly abstract to more detailed, structured approaches. *Abstract* models focus on the relationships between arguments – most notably, attacks – without considering their internal content. In contrast, *structured* models represent the internal composition of arguments, including premises and inference steps. Structured models are often viewed as concrete instantiations of their abstract counterparts.

On the reasoning side, a key distinction is between approaches that treat argumentation as *inference* – focusing on selecting acceptable arguments under various semantics – and those that also consider the argumentation *process* itself. Among the latter, *argument-game* approaches are prominent, modeling reasoning as an adversarial dialogue between a proponent, who defends a claim, and an opponent, who challenges it [1]. These models are closely linked to broader dialogical frameworks and have been recognized as particularly suitable for explainable AI, given their ability to dialectically justify claims in interactive settings [2].

Research on efficient implementation methods in argumentation has largely focused on abstract models, though structured models – despite their greater complexity – are gaining increasing attention. In contrast, argumentation-game approaches have received relatively little attention in this regard. A likely reason is that the reduction techniques successful in implementing argumentation-as-inference (as evidenced by top-performing systems in the recent main argumentation competition-ICCMA'23 [3])

23rd International Workshop on Nonmonotonic Reasoning, November 11-13, 2025, Melbourne, Australia
martin.diller@tu-dresden.de (M. Diller); piotr.gorczyca@tu-dresden.de (P. Gorczyca)
0000-0001-6342-0756 (M. Diller); 0000-0002-6613-6061 (P. Gorczyca)

are less readily applicable to game-based approaches, due to their iterative and potentially interactive nature.

Among reduction-based methods, those targeting *answer set programming* (*ASP*) are particularly popular. ASP is a declarative programming paradigm for knowledge representation and reasoning, offering a concise, expressive rule-based language and efficient solvers. Its strengths in modularity and succinctness make it a powerful tool for modeling complex problems, with growing use in academia and industry.

In the standard approach to problem solving with ASP, valid solutions are specified through logic rules such that the answer sets of a program correspond to the solutions of the problem. A major computational bottleneck is the *grounding* phase, where variables are instantiated with constants to produce variable-free programs. This becomes especially costly in iterative scenarios, where similar but slightly modified programs must be re-grounded from scratch at each step.

Multi-shot ASP is a recent advancement that enables modifying and re-solving logic programs across iterations, significantly reducing grounding and solving overhead [4]. This approach has proven beneficial in various domains including the implementation of games [5, 6].

This paper presents, to our knowledge, the first study of multi-shot ASP for argumentation, in particular, for implementing argument games. Specifically, we focus on *dispute derivations* in *assumption-based argumentation* (ABA) [7], a key rule-based framework that is also closely related to ASPIC+ [8] and that is the first structured formalism featured at ICCMA (since 2023). Our main contributions are:

- We present a multi-shot ASP-based approach for implementing ABA dispute derivations, targeting the latest rule-based dispute representation [9]. This version is the only one with a participating system at the most recent ICCMA and has outperformed other dispute-based systems in empirical evaluations [10]. Importantly, it nevertheless also clearly builds on earlier argument- and graph-based ABA dispute variants [11, 12].
- Compared to previous implementations of ABA disputes, our multi-shot approach is more declarative. It closely mirrors the formal definitions, delegating execution to the ASP engine, with a lightweight Python script managing multi-shot control. Thanks to its declarative and modular design, the approach is easily adaptable. We demonstrate this by extending it to support alternative games (e.g. complete and stable semantics), and by implementing similar games for Dung's AFs [13], the most fundamental of the many abstract argumentation formalisms. We also provide a simple interactive as well as visual interface to our implementation, proving that such aspects can also easily be accommodated.
- Our systematic empirical evaluation shows that, despite being substantially more concise and interpretable, the multi-shot implementation clearly outperforms the most efficient ABA dispute system on the ABA track ICCMA'23 benchmarks.
- Moreover, we introduce and evaluate an approximate variant that halts after a fixed number of iterations. It guarantees no false positives, yet frequently identifies disputes where the proponent wins. This variant achieves efficiency on par with the best existing inference-focused one-shot ASP-based ABA system. While it sacrifices accuracy to obtain this efficiency, it demonstrates that the multi-shot approach offers a strong alternative – especially when the advantages of game-based methods, such as support for interactivity and explainability, are needed.
- Finally, we propose our approach as a general methodology for implementing and comparing argument games, extending also to formalisms beyond those explicitly considered in this work, namely ABA and Dung's AFs.

2. Background

We begin by very briefly introducing Dung's AFs, ABA, and ASP programs, focusing on the syntax. We explain the aspects most relevant to our work in more detail (and provide examples) when turning to presenting the core ideas behind our approach in Section 3.

An abstract argumentation framework (AF) is a tuple $\mathcal{F} = (A, R)$, where A is a set of (abstract) arguments and $R \subseteq A \times A$ denotes the attack relation. We say that a_1 *attacks* a_2 if $(a_1, a_2) \in R$. A set of arguments $S \subseteq A$ is said to be *conflict-free* if for no pair of arguments $a_1, a_2 \in S$, a_1 attacks a_2. An argument $a_1 \in A$ is said to be *defended* from $a_2 \in A$ by $S \subseteq A$, if a_2 attacks a_1 and there is an argument $a_3 \in S$ that, in turn, attacks a_2. Finally, a set $S \subseteq A$ is *admissible* if S is conflict-free and S defends every argument $a \in S$ from any argument attacking a. All classical semantics introduced in [13] are admissibility-based; i.e. they return sets of arguments that are admissible (commonly also called *extensions*).

ABA frameworks add further structure to AFs. Concretely, an ABA framework $\mathcal{F}$ consists of a tuple $(\mathcal{L}, \mathcal{A}, ^-, \mathcal{R})$. Here, $\mathcal{L}$ is the language underlying the framework. As in most work on implementations for ABA we restrict our attention to what is arguably also the most common instance of ABA: those frameworks for which $\mathcal{L}$ is a finite set of propositional atoms. $\mathcal{A} \subseteq \mathcal{L}$ are the set of assumptions. $^-$ is the contrary relation associating to each $\alpha \in \mathcal{A}$ its set of contraries $^-(\alpha) \subseteq \mathcal{L}$. $\mathcal{R}$ is a set of rules of the form $h \leftarrow B$, where $B \subseteq \mathcal{L}$ and $h \in \mathcal{L}$. B is the body, while h is the head of the rule. Again, following most work on implementations for ABA, we also restrict our attention to flat ABA, i.e. frameworks for which assumptions cannot appear in the heads of rules. This is also the instance of ABA which has been featured at the ICCMA competition since 2023.

Arguments in ABA are built by deriving claims from facts via the rules. I.e. the head h of any rule $h \leftarrow \emptyset$ with empty body is an argument with conclusion h (such h can be considered facts). Moreover, if $a_1, \ldots, a_m$ are arguments with conclusions $h_1, \ldots, h_m$ and $r = h \leftarrow \{h_1, \ldots, h_m\} \in \mathcal{R}$, then the composition of $a_1, \ldots, a_m$ with r is an argument with conclusion h. Arguments are, thus, commonly represented as proof trees with nodes labelled by atoms, and the parent-child relation in the tree indicating that the atom labelling the parent node is in the body of a rule used to derive the child node. One argument attacks another if the first argument has as conclusion an atom which is a contrary of an assumption that is used in the second argument.

As explained in the introduction, disputes are a procedural means of deciding acceptance of an atom (the goal claim) modelled after argumentation: the proponent must find an argument deriving the goal claim and defend itself from the opponent by finding jointly consistent arguments attacking the counter-arguments constructed by the opponent. The opponents role, on the other hand, is precisely to find counter-arguments to any argument constructed by the proponent. Whoever has the last word in the dispute wins: i.e. if the opponent builds a counter-argument which the proponent cannot defend against then the opponent wins, while the proponent wins if it finds defending arguments against all counter-arguments of the opponent and the opponent cannot find any further counter-arguments. The goal claim is deemed acceptable whenever there is a dispute for it which the proponent wins; otherwise the goal claim is unacceptable. As we also indicated in the introduction, different versions of disputes exist differing in the way arguments are represented and the semantics that they cover. In this work we make use of the rule-based representation of disputes defined in [9] (which further simplifies the graph-based representation of [12], which in turn builds on [11]), where the proponent and opponent directly exchange rules rather than arguments (although the argument representation can be reconstructed from the rules).

For ASP we make use of the syntax of `clingo` [4]. ASP programs consist in a finite collection of rules of the form a_0 :- $a_1, \ldots, a_m$, *not* $a_{m+1}, \ldots,$ *not* a_{m+l}. where the a_is ($0 \leq i \leq m + l$) are atoms. The latter are expressions of the form $p(t_1, \ldots, t_n)$ where p is a predicate and the t_is are terms. Terms can be either constants or variables. The latter must be written starting with uppercase letters (e.g. X or Var), while constants cannot begin with uppercase letters. Positive (of the form a) and negative (of the form *not* a) atoms are also called literals.

Rules without variables are called ground. Rules with variables are shorthand for the set of ground rules obtained by uniformly replacing the variables in the (non-ground) rule with all constants occurring in the ASP program. If the rule is ground then it can informally be interpreted to mean that a_0 must be derived, i.e. be part of a solution of the program, if $a_1, \ldots, a_m$ are derived and $a_{m+1}, \ldots, a_{m+l}$ are not derived. Rules with empty bodies (of the form a_0 :- .) are facts, while those with empty heads (of the form :- $a_1, \ldots, a_m$, *not* $a_{m+1}, \ldots,$ *not* a_{m+l}.) are integrity constraints. While the first indicate that

a_0 must be a part of all solutions, the second indicates that solutions that satisfy the body of the rule are not allowed. Formally, solutions to ASP programs, so called answer-sets, are distinguished sets of (ground) atoms given by the stable-model semantics.

To ease ASP programming, several extensions of the basic syntax have been defined. For instance, arithmetic expressions can be used in bodies of rules. Also, numbers and strings (enclosed in quotation marks) can be used as constants, while "_" represents an anonymous variable: an unnamed fresh variable whose scope is that of the rule where it is used. More significantly, conditional literals are extensions of literals and are written as $a{:}b_1, \ldots, b_m$ where a and the b_is are positive or negated atoms. These express that a should be included (e.g. in the body of a rule) whenever $b_1, \ldots, b_m$ are derived. They are particularly useful when combined with variables: for instance, $a(X){:}b(X)$ denotes all instances of $a(X)$ for which $b(X)$ is derived. Another construct we make use of are cardinality constraints of the form $\{c_1; \ldots; c_m\} = k$ expressing that exactly k of the c_i conditional literals must be derived. For instance, $\{a(X) : b(X)\} = 1$ means that from all of the instances of $a(X)$ for which $b(X)$ is derived, exactly one must be selected.

In multi-shot ASP [4], the **`#program`** directive is used to partition an ASP program into multiple parametrizable subprograms. Each subprogram is identified by a predicate name and can take parameters, which are constants that influence its instantiation. Multi-shot solving is enabled by integrating ASP with an imperative host language, such as Python, through the `clingo` API. This integration allows dynamic interaction with the ASP solver via an imperative script. Specifically, a `clingo.Control` object is created within the host language, and subprograms are added to it. Through the API, grounding and solving can then be interleaved using dedicated `ground` and `solve` methods. Unlike static, one-shot solving, this interleaving enables grounding only the parts of the program that are relevant at a given stage, based on the evolving problem context. The `solve` method accepts optional arguments, such as assumptions, which are central to our approach. Assumptions are lists of (atom/truth value) pairs that constrain the solver by specifying which literals are to be treated as true or false during a particular solve call, enabling fine-grained control over successive solving phases.

3. Multi-shot ASP encodings of disputes

In this section, we present our proposal for using multi-shot ASP to implement disputes. For ease of understanding and also show the flexibility of our approach, we first consider the simpler case of Dung's AFs in Section 3.1, and then apply the same methodology to implement ABA disputes in Section 3.2. In Section 3.3 we present extensions for the stable semantics and visualisation purposes.

3.1. Abstract argumentation

For Dung AFs we make use of the fact that these can also be captured in ABA and, thus, tackle a specialization of ABA disputes for AFs. Our multi-shot implementation comprises two components shown in Figure 1: the ASP encoding, and the Python script controlling multi-shot solving via the `clingo` API. The latter forms the backbone of our approach and will also be reused in the ABA implementation described in Section 3.2.

The implementation assumes an input AF instance and a designated argument whose acceptance is to be evaluated, both encoded as ASP facts. Given an argumentation framework $\mathcal{F} = (A, R)$, its ASP encoding is:

$$en(\mathcal{F}) := \{\texttt{arg}(a) \mid a \in A\} \cup \{\texttt{att}(a,b) \mid (a,b) \in R\}$$

The designated argument $a \in A$, referred to as the *goal* argument, is encoded as $\texttt{g}(a)$.

The ASP encoding in Figure 1 is divided into three subprograms via lines 1, 5 and 17: 1) **`base`**: for initialization, 2) **`updateState`**`(t)`: to update the internal state at each step, and 3) **`step`**`(t)`: to select the next move. The **`updateState`**`(t)` and **`step`**`(t)` subprograms take the current dispute step number as a parameter.

```
#program base.
% initialize
m(0,p,G) :- g(G), not att(G,G), arg(G).

#program updateState(t).
defeat(t,C) :- m(_,p,P), att(P,C). % defeated
pm(t,p,P) :- m(_,o,O), att(P,O), not defeat(_,P),
  not m(_,p,P), not att(P,P),
  not att(P,D1) : m(_,p,D1). % possible p. move
pm(t,o,O) :- m(_,p,P), att(O,P), not defeat(_,O),
  not m(_,o,O). % possible o. move
end(t,p) :- g(G), m(_,p,G), not pm(t,o,_),
  defeat(_,O1) : m(_,o,O1). % p. won
end(t,o) :- not pm(t,p,_), m(_,o,O),
  not defeat(_, O). % opp. won

#program step(t).
m(t,o,A) :- pm(t-1,o,A).
{ m(t,p,A) : pm(t-1,p,A) } = 1 :- not pm(t-1,o,_).

#show m/3.
```

```
from clingo import Control, Number as N
                   Function as F
def main(instance, base_code, encoding):
 ctl = Control()
 ctl.load(instance)
 ctl.load(encoding)
 ctl.add("base", [], base_code)
 ctl.ground([("base", ())])
 t = 0
 while True:
  ctl.ground([("updateState",[N(t)])])
  p_win = (F("end",[N(t),F("p")]),True)
  res = ctl.solve(assumptions=[p_win],
                  on_model=print)
  if res.satisfiable:
    return True
  o_win = (F("end",[N(t),F("o")]),False)
  res = ctl.solve(assumptions=[o_win])
  if res.unsatisfiable:
    return False
  t += 1
  ctl.ground([("step",[N(t)])])
```

Figure 1: Multi-shot ASP encoding of AF disputes (left) and main control Python script (right).

At an abstract level, a dispute is a sequence of moves. For AFs, each move consists of a player – proponent or opponent, represented by constants p and o (the colors are for ease of reading) – choosing an argument to play. The key predicate is `m/3`, representing a move with parameters: step number, player, and chosen argument. The predicate `pm/3`, on the other hand, indicates available moves at a given step.

The **`base`** subprogram is grounded at the start, initiating the dispute with the proponent playing the goal argument at step 0, provided it is consistent (i.e., it does not attack itself).

The **`updateState`**`(t)` subprogram updates the internal state at each step `t` using auxiliary predicates. The predicate `defeat/2` collects all arguments attacked by the proponent so far (the "defeated" arguments). In Line 7, possible proponent moves are determined. A valid proponent move must (i) counter an opponent's argument, (ii) not be defeated, (iii) not have been used before by the proponent, (iv) be consistent, and (v) not attack other proponent arguments. In Line 10, opponent moves are derived. These must (i) attack a proponent's argument, (ii) not be defeated, and (iii) not have been used already by the opponent.

The rules in Line 12 and Line 14 define the `end/2` predicate, indicating whether the game has been won. The proponent wins if (i) the goal is among their arguments, (ii) the opponent has no more available moves, and (iii) all opponent arguments have been defeated. It is easy to see that if the proponent wins, their argument set corresponds to an admissible extension as per Section 2. The opponent wins if (i) the proponent has no available moves and (ii) at least one undefeated opponent argument remains.

The **`step`**`(t)` subprogram selects the next move. Disputes branch only at proponent moves, as admissibility requires the opponent to play all possible counter-attacks to a proponent's argument. Conversely, for each opponent move, the proponent must select one counter-argument. For efficiency (though this is easy to adapt), the encoding prioritizes opponent moves, performing all available opponent moves at each step (Line 18). A proponent move is selected only when no opponent moves are possible (Line 19).

The **`#show`** directive at the end of Figure 1 simply indicates which atoms are shown when printing the answer-sets. Concretely, the atoms indicating the moves made at each step are shown.

We now turn to the Python control script presented in Figure 1 (right). The `main` function (Line 3) takes three arguments: the ASP encoding of the input AF, a base program for initialization, and the main encoding for the argumentation formalism (e.g., the ASP code in Figure 1, left). The base program offers additional flexibility and control. For Dung's AFs, for instance, it is used to specify the goal

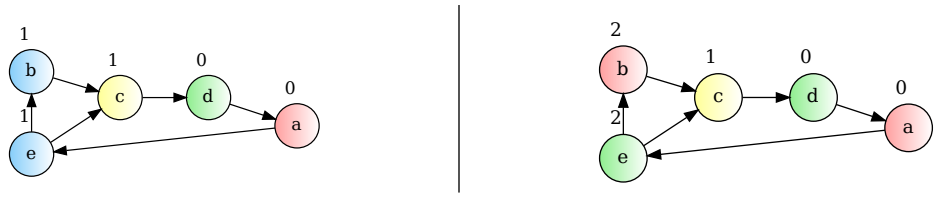

Figure 2: Representation of two steps of a dispute for the argumentation framework $\mathcal{F} = (A, R)$ and goal d, where $A = \{a, b, c, d, e\}$ and $R = \{(a, e), (d, a), (c, d), (b, c), (e, b), (e, c)\}$. The left figure depicts the dispute state after a proponent move `m(0,p,d)` and opponent move `m(1,o,c)`; the right figure depicts the state in which the proponent has won after making a `m(2,p,e)` move. The numbers above the arguments indicate the step at which their status is determined. The status is shown via colors: green – put forward by the proponent (e.g., e at step 2 via `m(2,p,e)`), yellow – put forward by the opponent, red – defeated unplayed arguments (e.g., a via `defeat(0,a)`), blue – possible moves at a step (e.g., `pm(1,p,b)`, `pm(1,p,e)`).

argument of the dispute in a short ASP program, such as that containing the fact `g(`a`)` for a given goal argument a.

Once the encodings are loaded, the **base** subprogram is grounded – this includes grounding the input framework as well as the goal directive. The step counter `t` is then initialized to 0.

The main loop begins in Line 10, where the subprogram **`updateState`**`(t)` is grounded for the current step `t`. In Line 12, an *assumption* is constructed – this is a pair consisting of a literal and a truth value. For a given step `t` $= n$, the assumption (`end(`n`,p)`, `True`) is created, asserting that the proponent has won at step n. This assumption is passed to the solver to constrain answer sets accordingly. If the solve is successful, this indicates that a dispute of length n exists in which the proponent wins.

If the solve fails, the script proceeds to Line 17, where the assumption (`end(`n`, o)`, `False`) is created. This assumption asserts that the opponent has *not* won at step n (i.e., there is at least one answer set which does not contain `end(`n`, o)`, meaning also that the proponent has not yet lost and the dispute may continue). If this second solve also fails, it implies that - regardless of the move the proponent makes at step n - they will loose to the opponent and, hence, the procedure can terminate.

If neither condition holds, the search proceeds: `t` is incremented, a new move is selected via the **`step`**`(t)` subprogram (Line 22), and the next iteration begins. Figure 2 provides the visualisation of a dispute for an exemplary AF generated by our system `MS-DIS`[1] (using the listings from Figure 1).

3.2. Assumption-based argumentation

To implement ABA disputes our approach is the same as that presented in Section 3.1. In particular, Figure 1 (right) continues to serve as the Python script controlling the multi-shot execution. The only difference lies in the inputs provided to this script.

Given an ABA instance $\mathcal{F} = (\mathcal{L}, \mathcal{A}, ^-, \mathcal{R})$, its encoding as a set of ASP facts is defined as follows:

$$en(\mathcal{F}) := \{\texttt{assumption(}a\texttt{)} \mid a \in \mathcal{A}\} \cup \{\texttt{contrary(}a,b\texttt{)} \mid (a,b) \in {}^-\}$$
$$\cup \bigcup_{r=h \leftarrow B \in \mathcal{R}} \{\texttt{head(}r,h\texttt{)}\} \cup \{\texttt{body(}r,b\texttt{)} \mid b \in B\}$$

The goal of the dispute is a statement $s \in \mathcal{L}$ that is to be justified dialectically. This is encoded as `g(`s`)`. For ABA, further inputs (which we will explain shortly) are facts encoding the termination condition and the advancement type. For the dispute for the admissible semantics these are `"tt(ta)."` and `"at(dabf)."` respectively.

The multi-shot ASP encoding for ABA disputes is shown in Figure 3. As with the encoding for AF disputes, the program is divided into three sub-programs: **`base`**, **`updateState`**`(t)`, and **`step`**`(t)`, each fulfilling the same roles as in the AF encoding.

At the most abstract level, ABA disputes also consist of a sequence of moves. The main difference compared to disputes for AFs is that each move now involves one of the players – the proponent or

[1] https://github.com/gorczyca/MS-DIS

the opponent (represented by the constants p and o as in the AF encoding) – putting forward either an assumption or a rule from the ABA framework. Thus, what is explicitly constructed is a *rule set* for each player, consisting of the assumptions and rules they have played. When a rule is played by a player, each statement in the rule's body and head is also considered as played by that player. For the proponent, this amounts to committing to the rule and its associated statements; for the opponent, it reflects an exploration of a possible line of attack against the proponent's commitments. To avoid redundancy, the proponent's rule set is considered a subset of the opponent's rule set.

The rule sets determine, in an implicit manner, the arguments available to each player: these are exactly the arguments that can be constructed from the rules and claims contained in their rule set. Hence, the proponent's arguments are those derivable from the proponent's rule set, while the opponent's arguments include all arguments derivable from their own rule set, which subsumes that of the proponent.

The fact that players in ABA disputes put forward both rules and assumptions allows us to distinguish between different types of moves. In the encoding given in Figure 3, we distinguish between eight move types, represented by the constants `pb1`, `pb2`, `pf1`, `pf2`, `ob1`, `ob2`, `of1` and `of2`. Proponent move types begin with `p`, and opponent move types begin with `o`. The letters `b` and `f` stand for "backward" and "forward", respectively.

The backward moves `pb1` and `ob1` are used to justify a claim s already in the player's rule set by introducing a rule $h \leftarrow B$ such that $h = s$. Conversely, the forward moves `pf1` and `of1` add a rule $h \leftarrow B$ when its body B is in the player's current claim set.

The backward moves `pb2` and `ob2` are used to attack an assumption a of the opposing player by introducing a rule $h \leftarrow B$ such that $h \in {}^{-}(a)$. Finally, the forward moves `pf2` and `of2` introduce an assumption $a_1 \in {}^{-}(a_2)$, provided that a_2 is an assumption in the opposing players claim set.

We refer to all move types that involve rules (i.e., all except `pf2` and `of2`) as *rule move types*, and they are declared using the predicate `rMT/1`. Among the proponent's moves, all except `pf1` are considered *branching*, and are denoted using the `branchMT/1` predicate. As in the case of AF disputes, only the proponent can introduce branching in the search for a winning dispute. The reason why `pf1` is not branching is that it simply derives consequences from claims the proponent has already committed to. Therefore, `pf1` does not introduce a choice point, but rather derives consequences that follow from previous choices.

Moves in ABA disputes are encoded via the predicate `m/4`, which now takes four arguments to represent (i) the dispute step, (ii) the player, (iii) the move type, and (iv) the rule or assumption involved in the move. Specifically, `m(t,P,T,X)` encodes a move made at turn `t`, by player `P`, of type `T`. If `T` is a rule move type (i.e., `rMT(T)` holds), then `X` refers to the rule's identifier; otherwise, it refers to the assumption introduced.

As in the AF encoding, possible moves are defined via the predicate `pm/4`, which also now has arity 4. To track the evolving rule sets of each player, we use the auxiliary predicates `stS/3` and `stR/4`. The predicate `stS(t,S,P)` denotes that statement `S` has been used by player `P` at step `t`. Similarly, `stR(t,P,h,r)` records that a rule $r = h \leftarrow B$ has been used by player `P` at step `t`.

Turning now from the predicates used to the encoding itself, as in the encoding for AFs, the **`base`** and **`updateState`**`(t)` subprograms define various auxiliary predicates that are used to constrain the possible moves available to the proponent and opponent at each step. The actual move to perform is selected by the **`step`**`(t)` subprogram.

The **`base`** subprogram, which is grounded at the start, defines in lines 2 to 4 the rule move types and branching move types, as well as the players (via the `plr/1` predicate). In lines 5 and 6, all statements appearing in rules are collected using the `rS/2` predicate. In Line 7, the dispute is initialised by adding the goal statement to both the proponent and opponent rule sets, provided the goal is not an inconsistent assumption. In Line 8, the `remBloR/4` predicate – encoding rules that are initially blocked for the proponent – is populated with those rules that are inconsistent.

The **`updateState`**`(t)` subprogram, as in the AF case, updates the internal state of the dispute. Lines 11 and 12 define the so-called defences and culprits: defences are assumptions to which the proponent is committed, while culprits are assumptions contrary to some claim in the proponent's rule set (i.e. these

```
#program base.
rMT(pb1;pb2;pf1;ob1;ob2;of1). % rule move types
branchMT(pb1;pb2;pf2). % branching (br.) move types
plr(p;o). % players; proponent (p.) and opponent (o.)
rS(R,S) :- head(R,S). % rule's statements from rule heads
rS(R,S) :- body(R,S). % rule's statements from rule bodies
stS(0,S,P) :- g(S), not contrary(S,S), plr(P). % goal - initial statement
remBloR(0,R,H,p) :- head(R,H), rS(R,S1), rS(R,S2), contrary(S1,S2). % blocked, inconsistent rules

#program updateState(t).
def(t,D) :- stS(_,D,p), assumption(D). % defence
cul(t,C) :- stS(_,S,p), contrary(C,S). % culprit
defCtr(t,DC) :- def(_,D), contrary(D,DC). % defence contrary
culCtr(t,CC) :- cul(_,C), contrary(C,CC). % culprit contrary

remR(t,R,H,P) :- not stR(_,R,H,P), head(R,H), plr(P). % remaining player's rule
remBloR(t,R,H,P) :- remR(t,R,H,P), body(R,B), cul(_,B), plr(P). % blocked remaining player P.'s rule
remBloR(t,R,H,p) :- remR(t,R,H,p), rS(R,S), defCtr(_,S). % blocked remaining p.'s rule

unexpS(t,H,p) :- stS(_,H,p), not stR(_,_,H,p). % unexpanded statement
stExpS(t,H,o) :- stS(_,H,o), remBloR(_,R,H,o) : remR(t,R,H,o). % fully expanded statement
stBloS(t,S,o) :- stS(_,S,o), cul(_,S). % state blocked statement
stBloS(t,S,o) :- stExpS(t,S,o), not assumption(S), stBloR(t,R,S,o) : stR(_,R,S,o).
stBloR(t,R,H,o) :- stR(_,R,H,o), body(R,B), stBloS(t,B,o). % state blocked rule

comS(t,S,p) :- def(_,D).  % complete statement
comS(t,H,p) :- stR(_,R,H,p), comS(t,S,p) : body(R, S).
unbloComS(t,S,o) :- stS(_,S,o), assumption(S), not cul(_,S). % unblocked complete (unb. com.) statement
unbloComS(t,H,o) :- stS(_,H,o), not stBloS(t,H,o), unbloComR(t,_,H,o).
unbloComR(t,R,H,o) :-  stR(_,R,H,o), not stBloR(t,R,H,o), unbloComS(t,B,o) : body(R, B).% unb. com. rule
unbloSupSS(t,S,o) :- stS(_,S,o), contrary(D,S), def(t,D), not stBloS(t,S,o).% unb. statements support. S
unbloSupSS(t,S,o) :- stS(_,S,o), not stBloS(t,S,o), unbloSupSR(t,R,_,o), body(R, S).
unbloSupSR(t,R,H,o) :- stR(_,R,H,o), not stBloR(t,R,H,o), unbloSupSS(t,H,o).% unb. rules supporting S
culCan(t,C) :- assumption(C), unbloSupSS(t,C,o). % culprit candidate

pm(t,p,pb1,R) :- remR(t,R,H,p), not remBloR(_,R,H,p), unexpS(t,H,p). % possible move (pm) "PB1"
pm(t,p,pb2,R) :- remR(t,R,H,p), not remBloR(_,R,H,p), culCan(t,C), contrary(C, H), not stS(_,H,p),
  not contrary(D, H) : def(_,D). % pm "PB2"
pm(t,p,pf1,R) :- remR(t,R,H,p), not remBloR(_,R,H,p), comS(t,B,p) : body(R, B). % pm "PF1"
pm(t,p,pf2,A) :- culCan(t,C), contrary(C,A), assumption(A), not stS(_,A,p), not contrary(A,A),
  not cul(_,A),  not contrary(D,A) : def(_,D). % pm "PF2"
pm(t,o,ob1,R) :- remR(t,R,H,o), not remBloR(_,R,H,o), unbloSupSS(t,H,o). % pm "OB1"
pm(t,o,ob2,R) :- remR(t,R,H,o), not remBloR(_,R,H,o), contrary(D, H), def(_,D). % pm "OB2"
pm(t,o,of1,R) :- remR(t,R,H,o), not remBloR(_,R,H,o), unbloComS(t,B,o) : body(R, B). % pm "OF1"
pm(t,o,of2,A) :- contrary(D,A), def(_,D), assumption(A), not stS(_,A,o). % pm "OF2"

stS(t,S,P) :- m(_,p,T,R), rMT(T), rS(R,S), plr(P). % new state statement
stS(t,A,P) :- m(_,p,T,A), not rMT(T), plr(P).
stS(t,S,o) :- m(_,o,T,R), rMT(T), rS(R,S).
stS(t,A,o) :- m(_,o,T,A), not rMT(T).
stR(t,R,H,P) :- m(_,p,T,R), head(R,H), rMT(T), plr(P). % new state rule
stR(t,R,H,o) :- m(_,o,T,R), head(R,H), rMT(T).

term(t) :-  tt(ta), g(G), comS(_,G,p), comS(_,CC,p) : culCtr(_,CC);
  not unbloComS(t,DC,o) : defCtr(_,DC).
end(t,p) :- term(t), not pm(t,o,_,_). % p. won; termination condition satisfied and o. cannot move
end(t,o) :- not term(t), not pm(t,p,_,_). % o won; term. cond. not satisfied and p. cannot move

#program step(t).
m(t,P,T,X) :- pm(t-1,P,T,X), not branchMT(T), t > 0. % proceed with non-br. move type
{ m(t,P,T,X) : pm(t-1,P,T,X) } = 1 :- t > 0, branchMT(T) : pm(t-1,_,T,_). % choose one br. move

#show m/4.
```

Figure 3: Multi-shot ASP encoding of ABA disputes for the admissible semantics.

are attacked by the proponent). Lines 13 and 14 then define the contraries of defences and culprits.

In Line 16, the predicate `remR/4` gathers the remaining rules for each player, i.e., rules not yet used. The subsequent lines define the subset of these that are *blocked*. For both players (Line 17), rules are

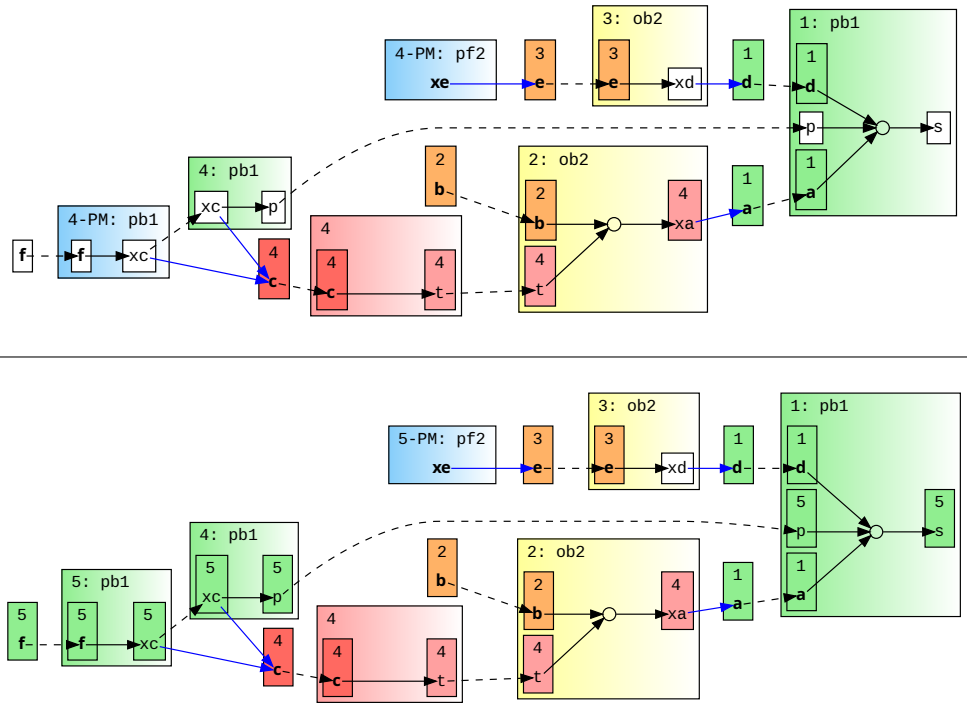

Figure 4: Two dispute states - 4 (top) and 5 (bottom) - for the ABA framework $\mathcal{F} = (\mathcal{L}, \mathcal{A}, ^-, \mathcal{R})$, where $\mathcal{A} = \{a, b, c, d, e, xe\}$, $^-(\tau) = \{x\tau\}$ for $\tau \in \{a, b, c, d, e\}$ (there are no further contraries), and $\mathcal{R} = s \leftarrow d, p, a; p \leftarrow xc; xc \leftarrow f; xd \leftarrow e; xa \leftarrow b, t; t \leftarrow c$. The dispute state on the top has been obtained via the sequence of moves: `m(1,p,pb1,`$s \leftarrow p, d, a$`)`, `m(2,o,ob2,`$xa \leftarrow b, t$`)`, `m(3,o,ob2,`$xd \leftarrow e$`)`, `m(4,p,pb1,`$p \leftarrow xc$`)`. At step 4, the proponent has two possible moves: `pm(4,p,pf2,`xe`)` and `pm(4,p,pb1,`$xc \leftarrow f$`)`. Performing the latter gives rise to the move `m(5,p,pb1,`$xc \leftarrow f$`)` and the dispute state shown at the bottom. Green pieces are introduced by the proponent, yellow ones by the opponent. Black arrows depict rules, while blue arrows represent attacks. Dashed arrows show dependencies between rules as well as between assumptions and rules. Assumptions are shown in boldface. Red rules are blocked, dark red statements are culprits and light red blocked opponents statements. Blue indicates possible moves. Numbers in squares next to a rule or assumption indicate the step at which they obtain their current status. For example, a "4" next to $p \leftarrow xc$ indicates that this rule was introduced by the proponent at step 4, which is also when $t \leftarrow c$ became blocked (shown in red) due to c becoming a culprit (because attacked by xc). Additional information can be retrieved: e.g. statement s becomes a proponent's complete piece at step 5 (indicated by the number above it), represented by `comS(5,`s`,p)`; c becomes a culprit at step 4 (`cul(4,`c`)`); or e and b become culprit candidates at steps 3 and 2, respectively (`culCan(3,`e`)` and `culCan(2,`b`)`).

blocked if they contain a culprit in the body – i.e., assumptions that are attacked by the proponent. Additionally, for the proponent, rules that contain the contrary of a defence in their body are also blocked, as they would render the proponent's rule set inconsistent.

In Line 20, the predicate `unexpS/3` identifies unexpanded statements within the proponent's rule set: statements for which no rule has yet been introduced that justifies them. Conversely, in Line 21, the predicate `stExpS/3` identifies fully expanded statements in the opponent's rule set – those for which all matching-head rules are blocked. The next lines then define the opponent's *blocked statements*: either culprits (Line 22), or non-assumption statements for which all rules with a matching head are blocked (Line 23). A *blocked rule* (`stBloS/4`), defined in Line 24, is any rule that contains a blocked statement in its body.

Lines 26 and 27 define the *complete statements* of the proponent: every defence is a complete statement, as is any statement derivable via rules whose body is composed entirely of other complete statements. These capture the statements for which the proponent has complete arguments. The opponent's analogous notions – *unblocked complete statements* (`unbloComS/3`) and *unblocked complete rules* (`unbloComR/4`) – are defined in Line 28-30. These represent statements and rules that are part of

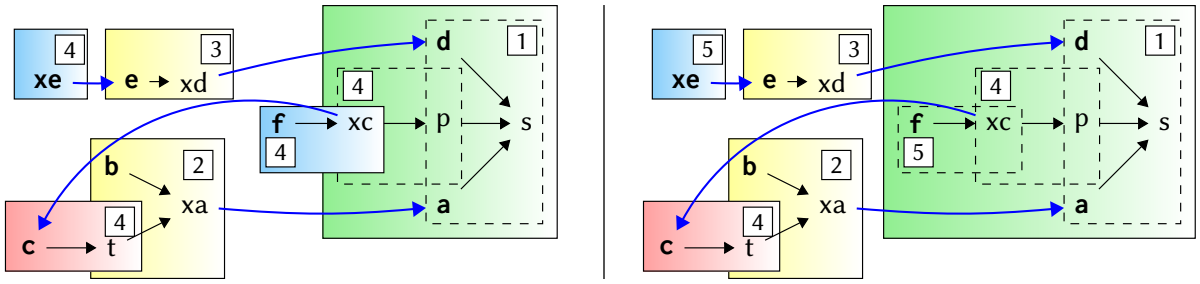

Figure 5: Argument-based representation of the dispute states from Figure 4 (left corresponding to top, right to bottom), with similar symbols and coloring scheme. The green rectangles represent the proponent's arguments; the yellow the opponent's.

complete arguments of the opponent and that are not attacked by the proponent.

Lines 31-33 identify *unblocked supporting statements* and *rules* of the opponent (`unbloSupSS/3`, `unbloSupSR/4`), i.e. those that contribute to justifying a contrary of a defence. These support the identification of *culprit candidates* in Line 34 – assumptions that appear in such justifications. Since these assumptions are part of a potential attack on the proponent, they become potential targets for counter-attack.

Lines 36-45 define the possible moves to choose from at step `t`. These are:

- `pb1` – the proponent introduces a rule $h \leftarrow B$ with head $h = s$ to justify a currently unexpanded statement s in their rule set, provided $h \leftarrow B$ is non-blocked and unused.
- `pb2` – the proponent introduces a rule $h \leftarrow B$ whose head is contrary to a culprit candidate, such that $h \leftarrow B$ is consistent, non-blocked, unused, and does not attack any defences.
- `pf1` – the proponent introduces an unblocked rule $h \leftarrow B$ whose body consists of complete statements, thus allowing the derivation of the new claim h.
- `pf2` – the proponent introduces an assumption $\bar{a}$ that is the contrary of a culprit candidate a and $\bar{a}$ is consistent, unused, not itself a culprit, and does not attack any defences.
- `ob1` – the opponent introduces a rule $h \leftarrow B$ with head $h = s$, where s is a statement contributing to an argument attacking a defence, and $h \leftarrow B$ is non-blocked and unused.
- `ob2` – the opponent introduces a rule $h \leftarrow B$ whose head is contrary to a defence, provided $h \leftarrow B$ is non-blocked and unused.
- `of1` – the opponent introduces a rule $h \leftarrow B$ whose body is composed solely of unblocked complete statements, thereby reinforcing or extending attacks on the proponent.
- `of2` – the opponent introduces an assumption $\bar{a}$, provided it is the contrary of a defence a and has not yet been used.

Lines 47-52 then extract the statements and rules used in the selected move and add them to the respective player's rule set, ensuring that the opponent also has access to any rule or statement used by the proponent.

Finally, Line 54 defines the termination condition for admissible semantics (`ta`): the dispute terminates successfully if (i) the goal is a complete statement of the proponent (i.e. the proponent has a complete argument for the goal), (ii) all culprits are complete statements (the proponent has complete arguments for all statements used to attack the opponent), and (iii) no contrary of a defence is an unblocked complete statement (all arguments of the opponent attacking the proponent are blocked, i.e. in turn attacked by the proponent). If this condition holds and no further opponent moves are possible, the proponent wins (Line 56); if the condition fails and the proponent has no remaining moves, the opponent wins (Line 57).

As in the encoding for AFs, the **`step(t)`** subprogram selects the next move, giving precedence to non-branching moves. Only if no such move is applicable will a single branching move among the available branching types be selected. Finally, the **`#show`** directive indicates that when printing the answer sets only the atoms encoding selected moves at each step are shown.

```
#program updateState(t).
% add another option to perform "PF2" move
pm(t,p,pf2,A) :- at(ds), assumption(A),
  not cul(_,A), not stS(_,A,p),
  not contrary(A,A),
  not contrary(D,A) : def(_,D).

% define: remaining assumptions
%  - neither defences, nor culprits
remA(t,A) :- tt(ts), assumption(A),
  not def(_,A), not cul(_,A).

% modify termination criteria:
term(t) :- tt(ts), g(G), comS(_,G,p),
  comS(_,CC,p) : culCtr(_,CC);
  not unbloComS(t,DC,o) : defCtr(_,DC);
  not remA(t,A) : assumption(A).
  % require no assumption be remaining
```

```
graph(m).
graph((A,l),m) :- arg(A).
node(A,(A,l)) :- arg(A).
edge((A,B),m) :- att(A,B).

attr(node,A,fillcolor,"green") :- m(_,p,A).
attr(node,A,fillcolor,"yellow") :- m(_,o,A).
attr(graph,(A,l),label,T) :- m(T,_,A).

ldefeat(Tm,A) :- defeat(Tm,A), not m(_,_,A),
  Tm = #min { T : defeat(T,A) }.
attr(graph,(A,l),label,Tm) :- ldefeat(Tm,A).
attr(node,A,fillcolor,"red") :- ldefeat(_,A).

lpm(Tm,P,A) :- pm(Tm,P,A), not m(_,_,A), not defeat(_,A),
  Tm = #max { T : pm(T,_,A) }.
attr(graph,(A,l),label,T) :- lpm(T,_,A).
attr(node,A,fillcolor,"lightblue") :- lpm(_,_,A).
```

Figure 6: Left: extension of the encoding in Figure 3 for the stable semantics. Right: code for the graphical representation of AF disputes (Figure 1) with `clingraph`.

Two steps of a dispute as generated by our system `MS-DIS` for an exemplary ABA framework is shown in Figure 4. This shows the rule-based representation which consists in the graph of dependencies and attacks among rules, together with labels (via colors) indicating the status of statements and rules at the dispute state. The corresponding argument-based representation is shown in Figure 5.

3.3. Extensibility

One of the main advantages of the ASP-based approach is its modularity: adding new functionalities often requires only small, local modifications. This section illustrates this extensibility with two examples: first, we show how to adapt the ABA dispute encoding to support stable semantics, and second: we demonstrate how to obtain a graphical representation of AF disputes, utilizing `clingraph`.

Stable semantics. Figure 6 (left) shows a code fragment extending Figure 3 for the stable semantics. For a goal statement s, the proponent wins the dispute iff a stable extension exists containing an argument for s. The proponent's defence set in such a case corresponds to a set of stable assumptions [7]. Figure 6 specifically extends the `updateState` subprogram from Figure 3 with two small additions. First (Line 3), it introduces a new forward move type (`pf2`) that allows the proponent to propose an assumption even if it does not attack a culprit (unlike `pf1`). The conditions are: the assumption (i) is not a culprit, (ii) is not already in the claim set, (iii) is consistent, and (iv) does not attack any current defences. Second, the termination condition is extended for stable semantics (Line 14), triggered by `tt(ts)`. In addition to the admissible criteria, it requires that all assumptions be either culprits or defences, i.e., all non-proponent arguments are attacked by the proponent. Remaining assumptions are identified via the `remA` predicate (Line 10). To enable the stable semantics, the advancement and termination types `at(ds)` and `tt(ts)` are passed as the base code parameter to the `main` solve call in Figure 1 (right).

Graphical representation. Figure 6 (right) shows how a graphical representation of disputes – here for AFs – can be generated with just a few lines of ASP code using `clingraph` [14]. In Line 1, the main graph `m` is declared, followed by the creation of subgraphs (A, l) for each argument A. Each subgraph contains a single node representing A, allowing step number labels to be attached individually. Edges are then added between arguments according to the attack relation. In Line 6 and the following lines, arguments used by the proponent (opponent) are colored green (yellow) and labeled with the step in which they were introduced. Defeated arguments are marked in red starting at Line 10, and labeled with the first step in which they became defeated. Finally, from Line 15, possible next moves are highlighted in light blue and annotated similarly. The visualization in Figure 2 is generated directly from the encoding (plus a few additional facts to enhance aesthetics) shown above. A similar encoding

was used for ABA disputes, with an example shown in Figure 4. These visualizations make dispute derivations easier to follow and are straightforward to produce within MS-DIS.

4. Implementation and evaluation

4.1. System

The code listings in Section 3 are part of our system MS-DIS, which implements dispute derivations for both AFs and ABA. The system supports both automatic and interactive modes and includes a visualization component. In automatic mode, given a claim and an ABA framework (or an argument and an AF), the system attempts to construct a winning dispute for the proponent. In interactive mode, the user is guided through the dispute process, with the system presenting available moves at each step and updating the dispute state based on the selected move. The visualization component is implemented using `clingraph` [14] as explained in Section 3.3. Further details can be found on the GitHub page.

4.2. Experimental setup

In our experiments, we focus on the implementation of MS-DIS (version 1.0) for ABA and, specifically, on the efficiency of its automatic mode. Since the automatic and interactive modes can also be interleaved, the results are relevant beyond the purely automatic setting.

We consider four aspects: (1) the performance benefit of the multi-shot approach; (2) comparison with previous implementations of ABA disputes; (3) comparison with the most efficient inference-oriented (i.e. also not dispute-based) system for ABA; and (4) the use of approximation within MS-DIS.

As to (1), we compare the multi-shot variant of MS-DIS – our main approach – with a naïve one-shot iterative version that restarts grounding and solving from scratch at each step. For a given step number n, the first solver call checks whether the proponent can win within n steps, and the second whether the search should continue. These correspond to lines 13 and 18 in Figure 1 (right).

As to (2), we compare MS-DIS with `flexABle` (version 1.0), which implements rule-based dispute derivations for ABA as proposed in [10]. The system `flexABle` has been shown to outperform earlier ABA systems [11, 12] and was the only ABA dispute solver to participate in the latest ICCMA competition [3].

As to (3), we compare MS-DIS to `aspforaba` [15] (downloaded on 4.6.25), which uses static one-shot ASP encodings. As the top-performing ABA solver in ICCMA'23, `aspforaba` offers an upper bound for performance. While we consider the comparison of MS-DIS and `aspforaba` informative, we did not expect MS-DIS to outperform `aspforaba`, given that the latter is optimized for decision problems, whereas MS-DIS aims to simulate a dispute justifying a claim. In particular, to decide acceptance of claims `aspforaba` requires a single call to an ASP solver, while MS-DIS requires multiple calls (albeit making use of multi-shot capabilities).

As to (4), we evaluate a step-bounded approximation mode of MS-DIS, where a winning dispute is returned only if found within a fixed step limit; otherwise, the instance is deemed unsatisfiable. We test bounds of 5, 10, and 25 steps. This ensures 100% specificity (true negative rate), allowing a fair comparison with an approximate mode of `flexABle` offering similar guarantees. In `flexABle`, approximation is achieved by restricting the opponent to a randomly selected subframework [10]. In our setup, the proponent sees the full framework, while the opponent is limited to 5%, 25%, or 50% of it, maintaining the specificity guarantee.

For our experiments, we use the ICCMA'23 ABA benchmarks [3], which comprise 400 instances containing between 25 and 5000 atoms. Each instance includes a query requiring solvers to determine credulous acceptance under admissible semantics. The benchmarks span all combinations of the following parameters: assumptions set at 10% or 30% of atoms; rule counts of up to 5 or 10 per atom; and rule body sizes capped at 5 or 10. Solvers were given a 600-second timeout per instance, with any run exceeding this limit recorded as a timeout. Notably, the correct outcome for 19 instances remains unknown, as no participant in ICCMA'23 produced a result for them.

	aspforaba	MS-DIS					flexABle			
			naïve	approx.				approx.		
	–	–	–	5	10	25	–	.05	.25	.5
# t-out.	22	233	260	0	91	164	269	47	160	217
time [h]	4	41	45	3	21	31	46	8	27	37
# t-out.	3	214	241	0	81	145	250	43	146	203
# inc.	0	0	0	153	30	0	0	114	69	34
time [h]	1	38	42	2	19	28	43	8	25	35
% acc.	100	100	100	60	90	100	100	66	71	81
% acc. t	99	44	37	60	71	62	34	59	44	38

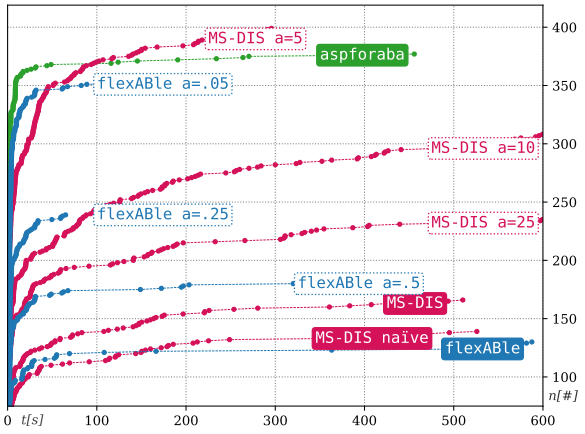

Figure 7: Table on the left shows solving results for all 400 instances (top part) and for the 381 with known correct answers (bottom part). "–" indicates the exact solver; "naïve" denotes MS-DIS in the naïve iterative mode; "approx." denotes approximate setup. Approximate setups compare MS-DIS with step limits 5, 10, 25 and flexABle with opponent sampling at 5%, 25%, and 50%. Metrics: number of timeouts (#t-out.), incorrect results (#inc.), total time in hours (time [h]), accuracy excluding timeouts (% acc.), and accuracy treating timeouts as incorrect (% acc. t). All values are rounded to the nearest integer. Plot on the right shows solving times for all instances, ordered by time (x-axis: solving time in seconds, y-axis: instance index). Solid label background indicates exact solver; white background: approximate.

In our experiments both MS-DIS and aspforaba make use of the ASP solver clingo [4] version 5.6.2. We made use of a high-performance computing cluster, with 64 GB of RAM allocated per task.

4.3. Results

The results of our experiments are summarized in Figure 7. MS-DIS, in the exact (i.e. non-approximate), multi-shot variant solved 36 more instances than flexABle and took 5 hours less of total solving time. Interestingly, even the naïve iterative MS-DIS variant performs slightly better than flexABle.

Regarding approximate methods, we find MS-DIS to outperform flexABle across nearly all metrics. For instance, an approximation with an upper bound of 10 steps in MS-DIS results in significantly fewer timeouts compared to flexABle with 25% sampling (81 timeouts for MS-DIS versus 146 for flexABle in the instances in the bottom half of the table). Additionally, MS-DIS achieves greater accuracy (90% compared to 71%, excluding timeouts; 71% compared to 44%, including timeouts) and a shorter total solving time (19 hours versus 25 hours). Notably, setting the upper bound to 25 steps guarantees 100% accuracy (excluding timeouts), suggesting that successful disputes rarely require more than 25 steps, regardless of the framework size.

By sacrificing accuracy, MS-DIS approaches the performance of aspforaba, and with a 5-step bound, it can even surpass aspforaba – albeit with an approximate accuracy of 60%. This highlights a potential niche for dispute-based systems, even when the primary goal is merely to determine the acceptance of claims. In particularly hard instances, executing a bounded dispute (especially with human-in-the-loop guidance) may yield more insight than receiving a time-out from systems such as aspforaba.

5. Conclusion

To the best of our knowledge, this is the first study of multi-shot solving for argument games, focusing on disputes in ABA and AFs. Our prototype, MS-DIS, outperforms existing dispute ABA systems, and its approximate variant matches the efficiency of the leading inference-focused ABA system – showing the benefits of the approach, especially when interaction and explanation are needed. More generally, our modular encodings and modern ASP systems suggest that multi-shot solving can serve as a common basis for implementing and comparing argument games. As shown for AFs and ABA, the

control components (Figure 1-right) are reusable across two-player games, while the ASP modules for initialization (**base**), state updates (**updateState**(t)), and move selection (**step**(t)) must be adapted. We envision developing a library of encodings for argument games in diverse formalisms.

Acknowledgments

This work was supported by funding from BMFTR within projects SEMECO (grant no. 03ZU1210B), KIMEDS (grant no. GW0552B), and MEDGE (grant no. 16ME0529).

Declaration on Generative AI

The authors used ChatGPT in order to: paraphrase and reword. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Caminada, Argumentation semantics as formal discussion, in: P. Baroni, D. Gabbay, M. Giacomin (Eds.), Handbook of Formal Argumentation, 2018, pp. 487–518.
[2] K. Cyras, A. Rago, E. Albini, P. Baroni, F. Toni, Argumentative XAI: A survey, in: IJCAI, 2021, pp. 4392–4399.
[3] M. Järvisalo, T. Lehtonen, A. Niskanen, ICCMA 2023: 5th international competition on computational models of argumentation, Artificial Intelligence 342 (2025) 104–311.
[4] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82.
[5] M. Gebser, R. Kaminski, P. Obermeier, T. Schaub, Ricochet robots reloaded: A case-study in multi-shot ASP solving, in: Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation, volume 9060 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 17–32.
[6] E. Böhl, S. Ellmauthaler, S. A. Gaggl, Winning snake: Design choices in multi-shot ASP, Theory Pract. Log. Program. 24 (2024) 772–789.
[7] K. Cyras, X. Fan, C. Schulz, F. Toni, Assumption-based argumentation: Disputes, explanations, preferences, in: P. Baroni, D. Gabbay, M. Giacomin (Eds.), Handbook of Formal Argumentation, 2018, pp. 365–408.
[8] S. Modgil, H. Prakken, Abstract rule-based argumentation, in: P. Baroni, D. Gabbay, M. Giacomin (Eds.), Handbook of Formal Argumentation, 2018, pp. 287–364.
[9] M. Diller, S. A. Gaggl, P. Gorczyca, Flexible dispute derivations with forward and backward arguments for assumption-based argumentation, in: CLAR, volume 13040 of *LNCS*, 2021, pp. 147–168.
[10] M. Diller, S. A. Gaggl, P. Gorczyca, Strategies in flexible dispute derivations for assumption-based argumentation, in: SAFA@COMMA, volume 3236 of *CEUR Workshop Proceedings*, 2022, pp. 59–72.
[11] F. Toni, A generalised framework for dispute derivations in assumption-based argumentation, Artif. Intell. 195 (2013) 1–43.
[12] R. Craven, F. Toni, Argument graphs and assumption-based argumentation, Artif. Intell. 233 (2016) 1–59.
[13] P. M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, Artif. Intell. 77 (1995) 321–358.
[14] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, *Clingraph*: A system for ASP-based visualization, Theory Pract. Log. Program. 24 (2024) 533–559.
[15] T. Lehtonen, J. P. Wallner, M. Järvisalo, Declarative algorithms and complexity results for assumption-based argumentation, J. Artif. Int. Res. 71 (2021) 265–318.