Argument & Computation 15 (2024) 21–48
DOI 10.3233/AAC-210027
IOS Press

A concurrent language for modelling agents arguing on a shared argumentation space

Stefano Bistarelli and Carlo Taticchi *
Department of Mathematics and Computer Science, University of Perugia, Italy
E-mails: stefano.bistarelli@unipg.it, carlo.taticchi@unipg.it

Abstract. While agent-based modelling languages naturally implement concurrency, the currently available languages for argumentation do not allow to explicitly model this type of interaction. In this paper we introduce a concurrent language for handling agents arguing and communicating using a shared argumentation space. We also show how to perform high-level operations like persuasion and negotiation through basic belief revision constructs, and present a working implementation of the language and the associated web interface.

Keywords: Argumentation theory, belief revision, concurrency, agents, language

1. Introduction

Many applications in the field of artificial intelligence aim to reproduce the human behaviour and reasoning in order to allow machines to think and act accordingly. One of the main challenges in this sense is to provide tools for expressing a certain kind of knowledge in a formal way so that the machines can use it for reasoning and infer new information. Argumentation Theory provides formal models for representing and evaluating arguments that interact with each other. Consider, for example, two people arguing about whether lowering taxes is good or not. The first person says that a) lowering taxes would increase productivity; the second person replies with b) a study showed that productivity decreases when taxes are lowered; then, the first person adds c) the study is not reliable since it uses data from unverified sources. The dialogue between the two people is conducted through three main arguments (a, b and c) whose internal structure can be represented through different formalisms [40,46], and for which we can identify the relations b attacks a and c attacks b. In this paper, we use the representation for Abstract Argumentation Frameworks (AFs in short) introduced by Dung [32], in which arguments are abstract, that is their internal structure, as well as their origin, is left unspecified. The example dialogue illustrated above can be modelled through an AF as shown in Fig. 1.

AFs have been widely studied from the point of view of the acceptability of arguments and several authors have investigated their dynamics, taking into account both theoretical [21,43] and computational[1] aspects. Logical frameworks for argumentation, like the ones presented in [29,30], have been introduced to fulfil the operational tasks related to the study of dynamics in AFs, such as the description of AFs, the specification of modifications, and the search for sets of "good" arguments. Although some of these

*Corresponding author. E-mail: carlo.taticchi@unipg.it.

[1]For example, a special track on dynamics appeared in the Third International Competition on Computational Models of Argumentation [10,11].

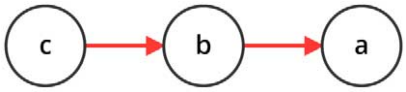

Fig. 1. Example of AF with three arguments.

languages could be exploited to implement applications based on argumentation, for instance, to model debates among political opponents, none of them considers the possibility of intelligent agents arguing with each other in a dynamic, concurrent environment where multiple reasoning and communication processes can take place simultaneously. This lack represents a significant gap between the reasoning capacities of AFs and their possible use in real-life AI-based tools. Consider, for example, the situation where two debating agents share an argumentation space represented by an AF and are capable of reasoning about their beliefs through argumentation semantics. Different kinds of interaction can take place between the two agents (they might want to discuss a certain topic, cooperate to find an optimal solution for a negotiation, or one could try to persuade the other to accept some terms), and since their reasoning processes could take place concurrently, the outcome produced by the interaction may vary according to how synchronisation is handled. Indeed, not only agents can act in different order (for access to memory and information update), but also arguments acceptability can change depending on how the concurrent interaction occurs. We will further elaborate on this in Section 3, where we give examples of how multiple argumentative processes are executed concurrently. The number of issues generated by simultaneous interactions increases with the number of actors (human or virtual) involved in a given process: if it is true that two housemates arguing about who should be the first to use the shower will not run into any problems related to the timing with which they present their arguments, the same cannot be said of larger systems in which hundreds or thousands of virtual agents representing IoT devices enter into negotiations over, for instance, a shared resource. In this case, proper management of the various actions taking place simultaneously, within the reasoning process in which the agents are involved, is essential to the functioning of the system. The possibility of modelling this type of situation falls, in fact, in the sphere of interest of research on computational argumentation, which also aims to provide systems in which virtual agents (also miming the behaviour of human counterparts) can reason automatically.

Motivated by the above considerations, we introduce a concurrent language for argumentation (CLA) that can be used for modelling different types of interaction between agents (such as negotiation and persuasion). In particular, our language allows for modelling concurrent processes, inspired by notions such as the *Ask-and-Tell constraint system* [44], and using an AF as a centralised store. Such AF provides a representation of the state of the world shared by all agents involved in the same process and enables agents to reason through argumentation paradigms. The language is thus endowed with primitives for the specification of interaction between agents through the fundamental operations of adding (or removing) and checking arguments and attacks. Using these primitives to build more sophisticated processes, we can model debating agents (e.g., chatbots) that take part in a conversation, provide arguments and make decisions based on the structure and the semantics of the AF represented in the shared argumentation space. Alchourrón, Gärdenfors, and Makinson (AGM) theory [1] gives operations (like expansion, contraction and revision) for updating and revising beliefs on a knowledge base. Our language is also suitable for implementing AGM-style operations that allow modifying the shared AF and changing the status of its arguments to enable, for instance, negotiation and other forms of dialogue.

The present work summarises and extends a series of previous studies [16–19,45] which led to the conceptualisation and development of CLA. The language was first presented in [16], where we defined an early version of the CLA syntax and operational semantics. Next, we introduced with [17] a prototype

implementation, and in [18] we showed examples of how to obtain belief revision operations via the constructs of our language. A user interface was then provided in [19] together with some examples of execution. A very brief overview of the work was also presented in [45]. In comparison with the works mentioned above, this paper provides new content concerning:

- the introduction of a semantics of failure that allows the execution of a CLA program to proceed and not to suspend waiting for particular conditions to occur;
- an improved operational semantics, for the addition and removal of arguments, that captures previously unmanaged behaviours in case of parallel execution;
- concrete examples of how CLA can be used to model persuasion and negotiation processes;
- a more thorough description of the implementation, with detailed examples covering the use of the interface and insights into how the parsing tree is obtained and visited.

We want to point out that a version of the language also incorporating the notion of time has been studied in [12–14] (see Section 6 for a more careful description). However, such a timed extension will not be addressed in this paper, which instead focuses on the basic formalism of CLA.

The rest of the paper is structured as follows: in Section 2 we recall some notions from Argumentation Theory; in Section 3 we present the syntax and the operational semantics of our concurrent language; Section 4 gives examples of high-level operations like persuasion and negotiation realised through CLA constructs; Section 5 describes how we implemented the language; in Section 6 we discuss existing formalisms from the literature that bring together dynamics in argumentation and multiagent systems, highlighting the contact points and the differences with our work; Section 7 concludes the paper with final remarks and perspectives on future work.

2. Background

Argumentation is an interdisciplinary field that aims to understand and model the human natural fashion of reasoning. In Artificial Intelligence, argumentation theory allows one to deal with uncertainty in non-monotonic (defeasible) reasoning, and it is used to give a qualitative, logical evaluation to sets of interacting arguments, called extensions. In his seminal paper [32], Dung defines the building blocks of abstract argumentation.

Definition 1 (AFs)**.** Let U be a finite set of all possible arguments,[2] which we refer to as the "universe". An Abstract Argumentation Framework is a pair $\langle Arg, R\rangle$ where $Arg \subseteq U$ is a set of adopted arguments and R is a binary relation on Arg.

AFs can be represented through directed graphs, that we depict using the standard conventions. For two arguments $a, b \in Arg$, $(a, b) \in R$ represents an attack directed from a against b. Moreover, we say that an argument b is *defended* by a set $B \subseteq Arg$ if and only if, for every argument $a \in Arg$, if $R(a, b)$ then there is some $c \in B$ such that $R(c, a)$.

The goal is to establish which are the acceptable arguments according to a certain semantics, namely a selection criterion. Non-accepted arguments are rejected. Different kinds of semantics have been introduced [6,32] that reflect qualities which are likely to be desirable for "good" subsets of arguments. In

[2] We introduce both U and $Arg \subseteq U$ (not present in the original definition by Dung) for our convenience, since in the concurrent language that we will define in Section 3 we use an operator to dynamically add arguments from U to Arg. A similar notion of not adopted arguments is also used in [42].

the rest of this paper, we will denote the extension-based semantics (also referred to as Dung semantics), namely admissible, complete, stable, preferred, and grounded, with their respective abbreviation *adm*, *com*, *stb*, *prf* and *gde*, and generically with σ.

Definition 2 (Extension-based semantics)**.** Let $F = \langle Arg, R \rangle$ be an AF. A set $E \subseteq Arg$ is conflict-free in F, denoted $E \in S_{cf}(F)$, if and only if there are no $a, b \in E$ such that $(a, b) \in R$. For $E \in S_{cf}(F)$ we have that:

- $E \in S_{adm}(F)$ if each $a \in E$ is defended by E;
- $E \in S_{com}(F)$ if $E \in S_{adm}(F)$ and $\forall a \in Arg$ defended by E, $a \in E$;
- $E \in S_{stb}(F)$ if $\forall a \in Arg \setminus E, \exists b \in E$ such that $(b, a) \in R$;
- $E \in S_{prf}(F)$ if $E \in S_{adm}(F)$ and $\nexists E' \in S_{adm}(F)$ such that $E \subset E'$;
- $E \in S_{gde}(F)$ if $E \in S_{com}(F)$ and $\forall E' \in S_{com}(F)$, $E \subset E'$.

Moreover, if E satisfies one of the above semantics, we say that E is an extension for that semantics (for example, if $E \in S_{adm}(F)$ we say that E is an admissible extension).

The different semantics described in Definition 2 correspond to different styles of reasoning, each of which may be more appropriate for being applied to a particular application domain. The characterisation of the reasoning requirements for the various domains is still a largely open research problem [7] and can only be based on general criteria rather than on specific cases. The stable semantics can be considered the strongest one: the accepted arguments attack all the others in the framework. Since a stable extension may not exist, one can resort to the semi-stable semantics [23], whose concept was first introduced in [48] under the name of admissible stage. Note that the set U used in Definition 1 is finite precisely because at least one semi-stable extension always exists for AFs with a finite number of arguments [25], while there is no guarantee in the case of infinite AFs. The semi-stable semantics, as well as the preferred one, do not have a unique extension, making the grounded semantics (that always exists and admits exactly one solution) an overall good option for establishing which arguments have to be accepted.

Besides enumerating the extensions for a certain semantics σ, one of the most common tasks performed on AFs is to decide whether an argument a is accepted in some extension of $S_\sigma(F)$ or in all extensions of $S_\sigma(F)$. In the former case, we say that a is *credulously* accepted with respect to σ; in the latter, a is instead *sceptically* accepted with respect to σ. The grounded semantics, in particular, coincides with the set of arguments sceptically accepted by the complete ones. Like the semi-stable semantics, the grounded one always exists and is often used since it is polynomially computable.

Example 1. In Fig. 2 we provide an example of an AF F in which: $S_{cf}(F) = \{\{\}, \{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}, \{b, d\}\}$, $S_{adm}(F) = \{\{\}, \{a\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}\}$, $S_{com}(F) = \{\{a\}, \{a, c\}, \{a, d\}\}$, $S_{prf}(F) = \{\{a, c\}, \{a, d\}\}$, $S_{stb}(F) = \{\{a, d\}\}$, and $S_{gde}(F) = \{\{a\}\}$. In detail, the singleton $\{e\}$ is not conflict-free because e attacks itself. The argument b is not contained in any admissible extension because no argument (included itself) defends b from the attack of a. The empty set $\{\}$, and the singletons $\{c\}$ and $\{d\}$ are not complete extensions because a, which is not attacked by any other argument, has to be

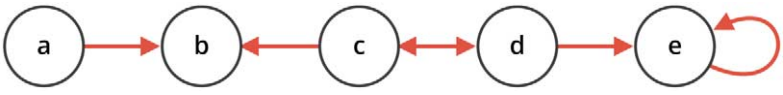

Fig. 2. AF used in Example 1.

contained in all complete extensions. Only the maximal (with respect to set inclusion) admissible extensions $\{a, c\}$ and $\{a, d\}$ are preferred, while the minimal complete $\{a\}$ is the (unique) grounded extension. Then, the arguments in the subset $\{a, d\}$, that conduct attacks against all the other arguments (namely b, d and e), represent a stable extension. To conclude the example, we want to point out that argument a is sceptically accepted with respect to the complete semantics, since it appears in all three subsets of $S_{com}(F)$. On the other hand, arguments c, which is in just one complete extension, is credulously accepted with respect to the complete semantics.

Many of the above-mentioned semantics (such as the admissible and the complete ones) use the notion of defence in order to decide whether an argument is part of an extension or not. The phenomenon for which an argument is accepted in some extension because it is defended by another argument belonging to that extension is known as *reinstatement* [24]. In that paper, Caminada also gives a definition for a reinstatement labelling, a total function that assigns a label to the arguments of an AF:

Definition 3 (Reinstatement labelling)**.** Let $F = \langle Arg, R\rangle$ be an AF and $\mathbb{L} = \{\texttt{IN}, \texttt{OUT}, \texttt{UNDEC}\}$. A labelling of F is a total function $L : Arg \to \mathbb{L}$. We define $\texttt{IN}(L) = \{a \in Arg | L(a) = \texttt{IN}\}$, $\texttt{OUT}(L) = \{a \in Arg | L(a) = \texttt{OUT}\}$ and $\texttt{UNDEC}(L) = \{a \in Arg | L(a) = \texttt{UNDEC}\}$. We say that L is a reinstatement labelling if and only if it satisfies the following:

- $\forall a, b \in Arg$, if $a \in \texttt{IN}(L)$ and $(b, a) \in R$ then $b \in \texttt{OUT}(L)$;
- $\forall a \in Arg$, if $a \in \texttt{OUT}(L)$ then $\exists b \in Arg$ such that $b \in \texttt{IN}(L)$ and $(b, a) \in R$.

An argument is labelled `IN` if all its attackers are labelled `OUT`, and it is labelled `OUT` if at least an `IN` node attacks it; in all other cases, the argument is labelled `UNDEC`. In Fig. 3 we show an example of reinstatement labelling on an AF in which arguments a and c highlighted in green are `IN`, red ones (b and d) are `OUT`, and the yellow argument e (that attacks itself) is `UNDEC`.

A labelling-based semantics [6] associates with an AF a subset of all the possible labellings. There exists a connection between reinstatement labellings and the Dung-style semantics: the set of *in* arguments in any reinstatement labelling constitutes a complete extension, while extensions for other semantics can be obtained through restrictions on the labelling as shown in Table 1. In the following sections, we use the notation L_σ^F to identify a labelling L corresponding to an extension of the semantics σ with respect to the AF F. Note that other definitions of labelling functions (as the one presented in [49]) can also be used to partition arguments of an AF, grasping different nuances in terms of acceptability. In [20], a unifying framework capturing several labelling proposals is described. In the next section, where we present our concurrent language for argumentation, the labelling of Definition 3 is used to implement both primitives and high-level operations that rely on the acceptability state of agent's belief and are able to change the underlying argumentation space accordingly.

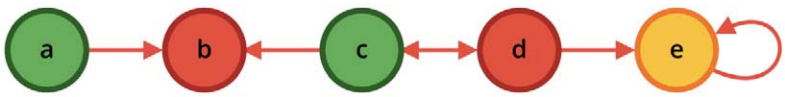

Fig. 3. Example of reinstatement labelling.

Table 1
Reinstatement labelling vs semantics

Labelling restrictions	Semantics
No restrictions	Complete
Empty `UNDEC`	Stable
Maximal `IN`/maximal `OUT`	Preferred
Maximal `UNDEC`/minimal `IN`/minimal `OUT`	Grounded

Table 2
CLA syntax

$$
\begin{array}{l}
P ::= C.A \\
C ::= p(x) :: A | C.C \\
A ::= success | add(Arg, R) \rightarrow A | rmv(Arg, R) \rightarrow A | E | A \| A | \exists_x A | p(a, l, \sigma) \\
E ::= E^w | E^f | E^f +_P E \\
E^w ::= check^w(Arg, R) \longrightarrow A | c\text{-}test^w(a, l, \sigma) \longrightarrow A | s\text{-}test^w(a, l, \sigma) \longrightarrow A | E^w + E^w \\
E^f ::= check^f(Arg, R) \longrightarrow A | c\text{-}test^f(a, l, \sigma) \longrightarrow A | s\text{-}test^f(a, l, \sigma) \longrightarrow A | E^f \|_G E^f | failure
\end{array}
$$

Table 3
CLA operational semantics – addition and removal

$\langle add(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg \cup Arg', R \cup R''\rangle\rangle$ with $R'' = \{(a, b) \in R' \mid a, b \in Arg \cup Arg'\}$	Addition
$\langle rmv(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg \setminus Arg', R \setminus \{R' \cup R''\}\rangle\rangle$ where $R'' = \{(a, b) \in R \mid a \in Arg' \vee b \in Arg'\}$	Removal

3. Syntax and semantics

The syntax of the concurrent language for argumentation (CLA) is presented in Table 2, where P, C, A and E denote a generic process, a sequence of procedure declarations (or clauses), a generic agent and a generic guarded agent, respectively. In a CLA process $P = C.A$, A is the initial agent to be executed in the context of the set of declarations C. To simplify the notation, we write a process $P = C.A$ simply as the corresponding agent A. The operational model of P can be formally described by a transition system $T = (Conf, \rightarrow)$. Configurations (in) $Conf$ are pairs consisting of a process and an AF $F = \langle Arg, R\rangle$ representing the common argumentation space.

In Tables 3–10 we give the definitions for the transition rules. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying those rules, and it characterises the evolution of the system. In particular, $\langle A, F\rangle \longrightarrow \langle A', F'\rangle$ represents a transition from a state in which we have the process A and the AF F to a state in which we have the process A' and the AF F'.

Addition and removal. Suppose to have an agent A whose argumentation space is represented by an AF $F = \langle Arg, R\rangle$. The agents *success* and *failure* represent a successful and a failed termination, respectively, so they may not make any further transition. An $add(Arg', R') \rightarrow A$ action (see Table 3) performed by the agent results in the addition of a set of arguments $Arg' \subseteq U$ (where U is the universe) and a set of relations R' to the AF F. When performing an addition, (possibly) new arguments are taken from $U \setminus Arg$. We want to make clear that the tuple (Arg', R') is not an AF, indeed it is possible to have $Arg' = \emptyset$ and $R' \neq \emptyset$, which allows to perform an addition of only attack relations to the considered AF. It is as well possible to add only arguments to F, or both arguments and attacks. However, the structure

Fig. 4. Example of two AFs. The rightmost is obtained from that on the left after removing argument b' and the attack (b', a).

of the shared store after an *add* operation is guaranteed to be an AF compliant with Definition 1, since only attacks between arguments in $Arg \cup Args'$ are added to R. Intuitively, $rmv(Arg, R) \rightarrow A$ allows to specify arguments and/or attacks to remove from the argumentation space. As illustrated in Table 3, removing an argument from an AF implies to also remove the attack relations involving that argument, while trying to remove an argument (or an attack) which does not exist in F will have no consequences. Other works (e.g., [22,29,31]) have already considered the possibility of retracting arguments to cope with different situations. To give a few examples, an agent might want to hide an already presented argument to comply with a security policy update, or because the audience forces it to retract what was stated previously. In the real world, some statements may be forgotten or simply lose value over time, cases that can be reproduced by deleting an argument. The removal is also used to determine the importance of arguments in an AF: the authors of [2] present a methodology for assessing when an argument a is decisive within a dialogue; if removing a does not produce any change on the set of extensions, then a is not decisive. Note that our language is very permissive: there are no constraints on which arguments or attacks an agent can add/remove.

Example 2. Consider an agent A who wants to add three arguments a, b and b' with the attacks (b, a) and (b', a) into the shared AF. The desired result can be obtained by running the following CLA program.

$$add\big(\{a, b, b'\}, \{(b, a), (b', a)\}\big) \longrightarrow success$$

Assuming that the underlying argumentation space is initially empty, we obtain the AF depicted in Fig. 4 (left). From this situation, we can remove the argument b' with the program below.

$$rmv\big(\{b'\}, \{\}\big) \longrightarrow success$$

The only attack involving b', that is (b', a), will also be removed. We obtain the AF of Fig. 4 (right).

Check. When a non-terminal operation of our language succeeds, the execution proceeds with the subsequent action. Otherwise, two are the possible outcomes: the operation can fail, making the execution to terminate, or it can suspend. Accordingly, we distinguish two categories of expressions that can be written using CLA syntax: E^w is an expression with waiting that suspends the execution in the case that the condition on its head is not satisfied, while E^f is an expression with failure that can only succeed or fail. By allowing expressions to fail, the program can continue the execution even if some of the operation does not succeed. The operation $check^w(Arg', R') \rightarrow A$ in Table 4 is used to verify whether the specified arguments and attack relations are contained in the set of arguments and attacks of the argumentation space, without introducing any further change. If the check is positive, then the operation succeeds, otherwise it suspends. On the other hand, $check^f(Arg, R) \rightarrow A$ fails when its guard is not satisfied.

Table 4
CLA operational semantics – check

$\dfrac{Arg' \subseteq Arg \wedge R' \subseteq R}{\langle check^{w/f}(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg, R\rangle\rangle}$	Check (1)
$\dfrac{Arg' \not\subseteq Arg \vee R' \not\subseteq R}{\langle check^{f}(Arg', R') \rightarrow A, \langle Arg, R\rangle\rangle \longrightarrow failure}$	Check (2)

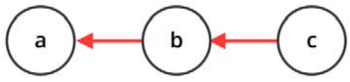

Fig. 5. Example of an AF obtained from that of Fig. 4 (right) by adding argument c and the attack (c, b).

Table 5
CLA operational semantics – credulous and sceptical test

$\dfrac{\exists L_\sigma^F \mid L_\sigma^F(a) = l}{\langle c\text{-}test^{w/f}(a, l, \sigma) \rightarrow A, F\rangle \longrightarrow \langle A, F\rangle}$	$\dfrac{\forall L_\sigma^F . L_\sigma^F(a) \neq l}{\langle c\text{-}test^{f}(a, l, \sigma) \rightarrow A, F\rangle \longrightarrow failure}$	Credulous Test
$\dfrac{\forall L_\sigma^F . L_\sigma^F(a) = l}{\langle s\text{-}test^{w/f}(a, l, \sigma) \rightarrow A, F\rangle \longrightarrow \langle A, F\rangle}$	$\dfrac{\exists L_\sigma^F \mid L_\sigma^F(a) \neq l}{\langle s\text{-}test^{f}(a, l, \sigma) \rightarrow A, F\rangle \longrightarrow failure}$	Sceptical Test

Example 2 (Continued)**.** Starting from the AF of Fig. 4 (right), we want to add a new argument c and the attack (c, b) only if the argument b is already in the shared argumentation space. Opting for a check with failure, we have the subsequent program, leading to the AF of Fig. 5.

$$check^f\big(\{b\}, \{\}\big) \longrightarrow add\big(\{c\}, \big\{(c, b)\big\}\big) \longrightarrow success$$

Note that, in this case, the operation $check^w(\{b\}, \{\})$ would have produced the same result since the condition in the check is satisfied. On the other hand, $check^f(\{d\}, \{\})$ fails since d is not in the AF when the check is performed, and $check^w(\{d\}, \{\})$ suspends for the same reason.

Credulous and sceptical test. We also have two distinct test operations (see Table 5), one credulous and the other sceptical, both requiring the specification of an argument $a \in A$, a label $l \in \{\texttt{IN}, \texttt{OUT}, \texttt{UNDEC}\}$ and a semantics $\sigma \in \{adm, com, stb, prf, gde\}$. The operations $c\text{-}test^w(a, l, \sigma) \rightarrow A$ and $c\text{-}test^f(a, l, \sigma) \rightarrow A$ succeed if *there exists at least* a labelling L_σ^F such that $L_\sigma^F(a) = l$; otherwise (in the case $L_\sigma^F(a) \neq l$ in all labellings) $c\text{-}test^w(a, l, \sigma) \rightarrow A$ suspends and $c\text{-}test^f(a, l, \sigma) \rightarrow A$ fails. Then, $s\text{-}test^w(a, l, \sigma) \rightarrow A$ and $s\text{-}test^f(a, l, \sigma) \rightarrow A$ succeed if a is labelled l in *all* possible labellings L_σ^F and suspend (fail, respectively) in the case $L_\sigma^F(a) \neq l$ in some labelling. Note that since the set of extensions $S_\sigma(F)$ is finite, all test operations are decidable.

Example 2 (Continued)**.** Agent A now wants to test whether the argument a in the shared AF of Fig. 5 is credulously accepted with respect to the admissible semantics and, in case of a positive answer, it wants to introduce a new argument d attacking a. We use the test with failure, but the same operation with waiting would not change the result.

$$c\text{-}test^f(a, \texttt{IN}, adm) \longrightarrow add\big(\{d\}, \big\{(d, a)\big\}\big) \longrightarrow success$$

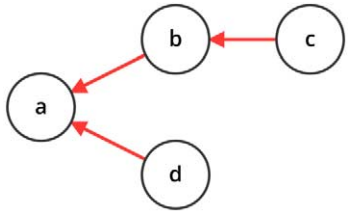

Fig. 6. Example of an AF obtained from that of Fig. 5 by adding argument d and the attack (d, a).

Table 6
CLA operational semantics – maximum parallelism

$\dfrac{\langle A_1, F\rangle \longrightarrow \langle A_1', F'\rangle, \langle A_2, F\rangle \longrightarrow \langle A_2', F''\rangle A_1', A_2' \neq \mathit{success}, \mathit{failure}}{\langle A_1 \| A_2, F\rangle \longrightarrow \langle A_1' \| A_2', *(F, F', F'')\rangle}$	Max-Par (1)
$\dfrac{\langle A_1, F\rangle \longrightarrow \langle \mathit{success}, F'\rangle, \langle A_2, F\rangle \longrightarrow \langle A_2', F''\rangle}{\langle A_1 \| A_2, F\rangle \longrightarrow \langle A_2', *(F, F', F'')\rangle \quad \langle A_2 \| A_1, F\rangle \longrightarrow \langle A_2', *(F, F', F'')\rangle}$ $\qquad \dfrac{\langle A_1, F\rangle \longrightarrow \mathit{failure}}{\langle A_1 \| A_2, F\rangle \longrightarrow \mathit{failure} \quad \langle A_2 \| A_1, F\rangle \longrightarrow \mathit{failure}}$	Max-Par (2)

Since the test succeeds, we obtain an AF as in Fig. 6. If we had chosen to perform a sceptical test instead, $s\text{-}test^f(a, \texttt{IN}, adm)$ would have failed, and $s\text{-}test^w(a, \texttt{IN}, adm)$ would have suspended.

Parallelism. A debate involving many agents that asynchronously provide arguments can be modelled as a parallel composition of *add* operations performed on the argumentation space. The parallel composition in CLA can be declined in two different ways: i) in terms of *maximum parallelism* and ii) with *interleaving*. According to maximum parallelism, all the actions that are composed through the parallel operator $\|$ are executed in one single computational step, while, following the interleaving approach, only one action is executed at a time. The operator $\|$ of Table 6 enables the specification of concurrent argumentation processes in form of maximum parallelism.

Parallel composition of two actions $A_1 \| A_2$, with the assumption of maximum parallelism, can result in three possible behaviours: it succeeds when both actions succeed, suspends when at least one action suspends and fails in the remaining case (i.e., when both actions fail). We use $*(F, F', F'') := (F' \cap F'') \cup ((F' \cup F'') \setminus F)$ to handle parallel additions and removals of arguments.[3] In particular, if an argument b' is added and removed at the same moment (e.g., through the program $add(\{b'\}, \{(b', a)\}) \rightarrow \mathit{success} \| rmv(\{b'\}, \{\}) \rightarrow \mathit{success}$), we have two possible outcomes:

- if b' is not present in the argumentation space (see Fig. 4, right), then the *add* operation prevails over the *rmv* one since $b' \in ((F' \cup F'') \setminus F)$ and we obtain an AF as the leftmost in Fig. 4;
- on the other hand, when b' is already in the shared memory (Fig. 4, left), we have that $b' \notin ((F' \cup F'') \setminus F)$ and b' is removed (we obtain an AF as in Fig. 4, right).

In Table 7, instead, we give the transition rules for the parallel composition operator $\|$ when interleaving is taken into account. In this case, only one agent is handled at a time and the actions of two concurrent agents are executed sequentially in two distinct steps of the computation.

Note that the operations that read and modify the shared AF are atomic (like the ask and tell operators from which they derive [44]) and therefore do not risk producing inconsistent data. Furthermore,

[3] Union, intersection and difference between AFs are intended as the union, intersection and difference of their sets of arguments and attacks, respectively.

Table 7
CLA operational semantics – parallelism with interleaving

$\dfrac{\langle A_1, F\rangle \longrightarrow \langle A_1', F'\rangle A_1' \neq \mathit{success}, \mathit{failure}}{\begin{array}{c}\langle A_1 \| A_2, F\rangle \longrightarrow \langle A_1' \| A_2, F'\rangle \\ \langle A_2 \| A_1, F\rangle \longrightarrow \langle A_1' \| A_2, F'\rangle\end{array}}$		Inter-Par (1)
$\dfrac{\langle A_1, F\rangle \longrightarrow \langle \mathit{success}, F'\rangle}{\begin{array}{c}\langle A_1 \| A_2, F\rangle \longrightarrow \langle A_2, F'\rangle \\ \langle A_2 \| A_1, F\rangle \longrightarrow \langle A_2, F'\rangle\end{array}}$	$\dfrac{\langle A_1, F\rangle \longrightarrow \mathit{failure}}{\begin{array}{c}\langle A_1 \| A_2, F\rangle \longrightarrow \mathit{failure} \\ \langle A_2 \| A_1, F\rangle \longrightarrow \mathit{failure}\end{array}}$	Inter-Par (2)

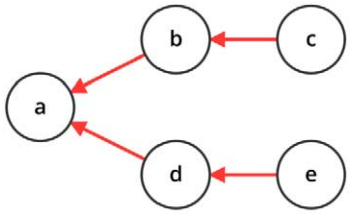

Fig. 7. Example of an AF obtained from that of Fig. 6 by adding argument e and the attack (e, d).

although having one or more agents acting asynchronously does not affect the way in which the shared AF is accessed and modified, the possibility of modelling parallel agents makes it possible to simulate the behaviour of real agents acting individually in a distributed environment.

Example 2 (Continued)**.** Suppose that not one, but two agents A and B want to modify the shared argumentation space given by the AF of Fig. 6. Agent A wants to add an argument e with the attack (e, d), while agent B wants to remove the same argument e. The two agents act simultaneously, so their behaviour can be modelled through the parallel operator.

$$\begin{array}{l} \mathit{add}\big(\{e\}, \big\{(e,d)\big\}\big) \longrightarrow \mathit{success} \\ \| \\ \mathit{rmv}\big(\{e\}, \{\}\big) \longrightarrow \mathit{success} \end{array}$$

Under the assumption of maximum parallelism, operations $\mathit{add}(\{e\}, \{\})$, and $\mathit{rmv}(\{e\}, \{\})$ are executed at the same time during the initial step of the computation. Since argument e is not in the initial argumentation space (see Fig. 6), it is added to the shared AF. At this point, both parallel branches of the program succeed and, consequently, the program itself terminates with success.

On the other hand, when using the interleaving approach, the execution of the above program can proceed in two ways. If the processor first handles the operation $\mathit{add}(\{e\}, \{(e, d)\})$ and then $\mathit{rmv}(\{e\}, \{\})$, the argument e is added into the argumentation space during the first step of the computation, and it is removed in the second step, producing in result an AF shaped like that of Fig. 6. If, instead, $\mathit{rmv}(\{e\}, \{\})$ is executed first and $\mathit{add}(\{e\}, \{(e, d)\})$ second, the resulting AF will be the one of Fig. 7, since argument e will remain in the shared argumentation space.

Guarded parallelism. The operator $\|_G$ for guarded parallelism is designed to execute all the operations for which the guard in the inner expression is satisfied. As shown in Table 8, $E_1 \|_G E_2$ succeeds when either E_1, E_2 or both succeed and all the operations with a satisfiable guard are executed. It only fails

Table 8
CLA operational semantics – guarded parallelism

$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 \|_G E_2, F\rangle \longrightarrow \langle A_1 \| A_2, F\rangle}$	Guarded Parallelism (1)
$\dfrac{\langle E_1, F\rangle \longrightarrow \langle E_1', F\rangle, \langle E_2, F\rangle \longrightarrow \langle E_2', F\rangle}{\langle E_1 \|_G E_2, F\rangle \longrightarrow \langle E_1' \|_G E_2', F\rangle}$	Guarded Parallelism (2)
$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle, \langle E_2, F\rangle \longrightarrow \mathit{failure}}{\langle E_1 \|_G E_2, F\rangle \longrightarrow \langle A_1, F\rangle \quad \langle E_2 \|_G E_1, F\rangle \longrightarrow \langle A_1, F\rangle}$	Guarded Parallelism (3)

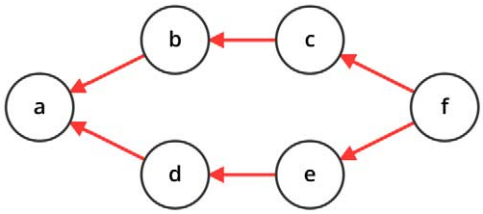

Fig. 8. Example of an AF obtained from that of Fig. 7 by adding argument f and the attacks (f, c) and (f, e).

if both the expressions E_1 and E_2 fail. This behaviour is different both from classical parallelism (for which all the agents have to terminate in order for the procedure to succeed) and from nondeterminism (that only selects one branch). Since only the composition of expressions that can fail are allowed in a guarded parallelism, it cannot suspend under any circumstances.

Example 2 (Continued)**.** To illustrate the functioning of operator $\|_G$ we use again two agents A and B sharing an argumentation space represented by the AF of Fig. 7. A will perform a credulous test to know if a is `IN` for some admissible labelling, and in case of a positive response, it will add an argument f which attacks both c and e. At the same time, agent B first wants to verify through a sceptical test whether a is `IN` in all admissible labelling, and if so, it will add an argument g with the attack (g, a). Both test operations composed via guarded parallelism are with failure as imposed by the operational semantics in Table 8.

$$\begin{aligned} &\textit{c-test}^f(a, \texttt{IN}, \textit{adm}) \longrightarrow \textit{add}\big(\{f\}, \big\{(f, c), (f, e)\big\}\big) \longrightarrow \textit{success} \\ &\|_G \\ &\textit{s-test}^f(a, \texttt{IN}, \textit{adm}) \longrightarrow \textit{add}\big(\{g\}, \big\{(g, a)\big\}\big) \longrightarrow \textit{success} \end{aligned}$$

As we can see from the resulting AF depicted in Fig. 8, only one of the parallel branches is executed with success. The credulous test performed by agent A succeeds, leading to the addition of argument f and the attacks (f, c) and (f, e). The sceptical test of agent B, however, fails, causing this second branch to terminate with failure. We want to remark that since one of the expressions composed through guarded parallelism succeeds, the whole construct still succeeds.

If then else. The operator $+_P$ is left-associative and realises an if-then-else construct (see Table 9): if we have $E_1 +_P E_2$ and E_1 is successful, than E_1 will be always chosen over E_2, even if also E_2 is successful, so in order for E_2 to be selected, it has to be the only one that succeeds. Moreover E_1 needs to

Table 9
CLA operational semantics – if then else

$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle E_1, F\rangle} \quad \dfrac{\langle E_1, F\rangle \longrightarrow \langle E_1', F\rangle}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle E_1' +_P E_2, F\rangle}$	If Then Else (1)
$\dfrac{\langle E_1, F\rangle \longrightarrow \mathit{failure}, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle E_2, F\rangle}$	If Then Else (2)

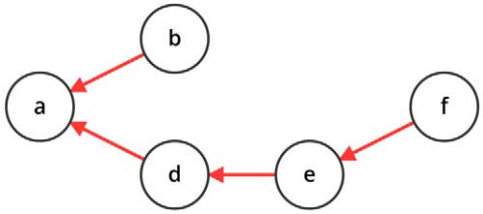

Fig. 9. Example of an AF obtained from that of Fig. 8 by removing argument c.

be an expression with failure, since a waiting expression might never fail, making it impossible to continue the execution with E_2. Differently from nondeterminism, $+_P$ prioritises the execution of a branch when both E_1 and E_2 can be executed. Notice that an if-then-else construct cannot be obtained from nondeterminism since our language is not expressive enough to capture success or failure conditions of each branch (we have angelic nondeterminism, but only with a one-step lookahead obtained through check/test guards).

Example 2 (Continued)**.** This time, our agent wants to choose between two possible executions, prioritising one and leaving the second as a fallback in case the first one fails. For this purpose, the $+_P$ operator is used. In the example, the agent will first try to perform a credulous test to find out whether argument c is `OUT` in at least one labelling of the shared AF of Fig. 8. In the case of a positive response, c will be removed and the execution will successfully terminate. If the test on argument c fails, the agent will try to perform the same test on d and if this latter argument turns out to be `OUT` in at least one labelling of the AF, it will be removed. Again, the execution will end with success.

$$\textit{c-test}^f(c, \texttt{OUT}, \textit{adm}) \longrightarrow \textit{rmv}(\{c\}, \{\}) \longrightarrow \textit{success}$$
$$+_P$$
$$\textit{c-test}^f(d, \texttt{OUT}, \textit{adm}) \longrightarrow \textit{rmv}(\{d\}, \{\}) \longrightarrow \textit{success}$$

The execution of the program above produces the AF represented in Fig. 9. We can see that the first condition to be tested, that is $\textit{c-test}^f(c, \texttt{OUT}, \textit{adm})$, is true, thus the first branch of the $+_P$ operator is executed and the argument c is removed. The second part of the construct, which would have led to the removal of d, is also true but is never executed because the program terminates with success before that. In fact, the program would fail only if both conditions were false and would suspend only if the first condition was false and the second suspended.

Nondeterminism, hidden variable and procedure call. The remaining operators shown in Table 10 are classical concurrency compositions. Any agent composed through $+$ (rule Nondeterminism of Table 10)

Table 10
CLA operational semantics – nondeterminism, hidden variables and procedure call

$\dfrac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle}{\langle E_1 + E_2, F\rangle \longrightarrow \langle A_1, F\rangle}$ $\langle E_2 + E_1, F\rangle \longrightarrow \langle A_1, F\rangle$ $\quad \dfrac{\langle E_1, F\rangle \longrightarrow \langle E_1', F\rangle, \langle E_2, F\rangle \longrightarrow \langle E_2', F\rangle}{\langle E_1 + E_2, F\rangle \longrightarrow \langle E_1' + E_2', F\rangle}$	Nondeterminism
$\dfrac{\langle A[y/x], F\rangle \longrightarrow \langle A', F'\rangle}{\langle \exists_x A, F\rangle \longrightarrow \langle A', F'\rangle}$ with $y \in U \setminus Arg$	Hidden Variables
$\langle p(y), F\rangle \longrightarrow \langle A[y/x], F\rangle$ with $p(x) :: A$ and $x \in \{a, l, \sigma\}$	Procedure Call

is chosen if its guards succeed. The existential quantifier $\exists_x A$ (rule Hidden Variable of Table 10) behaves like agent A where variables in x are local to A, thus hiding the information on x provided by the external environment. Finally, the procedure call has a single parameter which can be an argument, a label among `IN`, `OUT` and `UNDEC`, or a semantics σ. If necessary, the procedure call can be extended for allowing more than one parameter.

Given the transition system defined in Tables 3–10, we can observe the behaviour of a process $P = C.A$ through the trace (list of instructions executed) of terminating computations. The observables of the language provided in the following Definition 4 collect the results of successful or failed computations that an agent A can perform.

Definition 4 (Observables for CLA)**.** Let $P = C.A$ be a CLA process. We define

$$\mathcal{O}_{io}(P) = \lambda F.\Big\{F_1 \cdots F_n \cdot ss \,|\, F = F_1 \text{ and } \langle A, F_1\rangle \longrightarrow^* \langle \mathit{success}, F_n\rangle\Big\}$$
$$\cup \Big\{F_1 \cdots F_n \cdot \mathit{ff} \,|\, F = F_1 \text{ and } \langle A, F_1\rangle \longrightarrow^* \langle \mathit{failure}, F_n\rangle\Big\}$$

where $\longrightarrow^*$ denotes the reflexive and transitive closure of a transition relation $\longrightarrow$.

As we will see in the next section, we aim to use the operators of our language to model the behaviour of agents involved in particular argumentative processes.

4. CLA for persuasion and negotiation dialogues

The basic constructs which compose CLA operational semantics allow realising programs that simulate the interaction between two or more counterparts. The process of exchanging information through such an interaction can assume different nuances, according to the goal of the communication itself. For instance, two agents in conflict to obtain a resource within a distributed system may want to come to a compromise they both can agree on. In that case, we talk about negotiation. Another possibility for one agent is to persuade the other to accept a fact or a condition. Below, we provide examples of how high-level interaction between agents can be obtained through CLA programs.

4.1. Persuasion with CLA

According to [50], persuasion is a particular form of dialogue in which the involved counterparts try to affirm their own thesis. Each participant/agent in the persuasion dialogue holds a thesis which is opposed to the others and needs to be proven "true" in order to be accepted. To persuade its opponents,

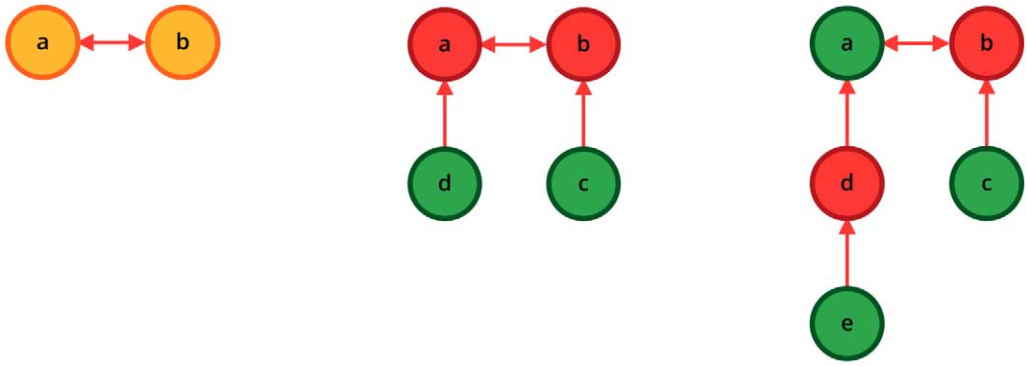

Fig. 10. AFs representing the evolution of a persuasion process from its beginning (on the left) to its conclusion (on the right).

an agent can elaborate different strategies [35] (sequences of actions to perform in the system) both for supporting its own beliefs and for defeating those supported by others. An approach to persuasion through argumentation is given in [39], where agents play a game to solve conflicts of points of view between discordant theses: an agent has to defend its position by replying to every attack against its initial claim. If it fails, the opponent wins the game. Below, we provide an example of how a persuasion dialogue can be enacted by using CLA constructs.

Example 3. Imagine two agents, A and B, discussing about the problem of violence in video games. This topic is often used as an example of an argumentative process where discordant opinions are provided for or against a certain initial thesis (see for instance [51]). Agent A believes that violent video games can make people, especially the young, more aggressive, while agent B has the opposite opinion, that is video games are harmless and safe for all users. This scene can be represented through an AF (like that in Fig. 10, left) with two conflicting arguments: a, which supports video games, and b, which is against them. Attacks in the frameworks are deduced from the arguments themselves: in this example we have that a attacks b and b attacks a.

We use the grounded semantics as criterion for establishing the acceptability of arguments. Other semantics can be considered as well with similar results. In this initial situation, since a and b are attacking each other without being defended, none of them can be part of the grounded extension, and thus no agent will be able to persuade the other. In an attempt to prove the validity of their thesis, the two agents bring forward new arguments: A states that young people could emulate the violent behaviours seen in video games (argument c of Fig. 10, middle), while B points out that violent scenes in video games are not real (argument d of Fig. 10, middle). Neither argument a nor b can still be accepted. The discussion ends, in this example, with agent A adding that people can still be influenced by what they see, regardless of whether it is fact or fiction. This last argument corresponds to e in Fig. 10 (right), and forms, together with a and c, the grounded extension. At this point, assuming agent B does not reply, A has proven its thesis (the argument a) to be acceptable, persuading the other agent. All arguments used in this example are summarized below.

a: "Violent video games can make people, especially the young, more aggressive."
b: "Video games are harmless and safe for all users."
c: "Young people could emulate the violent behaviours seen in video games."
d: "Violent scenes in video games are not real."
e: "People can be influenced by what they see, regardless of whether it is fact or fiction."

Table 11
Example of a CLA program for persuasion

$A : add(\{a\}, \{\}) \longrightarrow check^w(\{b\}, \{\}) \longrightarrow add(\{\}, \{(a, b)\}) \longrightarrow add(\{c\}, \{(c, b)\}) \longrightarrow$ $check^w(\{d\}, \{\}) \longrightarrow add(\{e\}, \{(e, d)\}) \longrightarrow c\text{-}test^f(a, \texttt{IN}, gde) \longrightarrow success$
$B : add(\{b\}, \{\}) \longrightarrow check^w(\{a\}, \{\}) \longrightarrow add(\{\}, \{(b, a)\}) \longrightarrow add(\{d\}, \{(d, a)\}) \longrightarrow$ $c\text{-}test^f(b, \texttt{IN}, gde) \longrightarrow success$
$P : A \parallel B$

The whole dialogue can be formulated through a CLA program. We propose one possible implementation in Table 11, where two agents interact via the transition rules of Tables 3–10 and synchronisation is obtained through the check operator. The first agent A adds argument a into the argumentation space and then waits until also argument b can be found in the underlying AF. Afterwards, A adds argument c together with two attacks, one from a to b and the other one from b to c. Then A waits again for argument d to be present in the AF, after which argument e and the attack (e, d) are added. Finally, A execute a credulous test with failure on argument a: if this latter is labelled `IN` in the grounded labelling, the execution terminates with success. Concurrently, agent B executes its process, which starts with the addition of an argument b. Then B check if argument a is in the argumentation space and, in case of a positive answer, another argument d, and the attacks (b, a) and (d, a) are also added. The last operation executed by B is a credulous test with failure that succeeds if argument b is `IN` in the grounded labelling. Different solutions can also be adopted to model the same dialogue. For instance, agent A could add the argument c into the argumentation space and then use a parallel construct to concurrently add the two attacks (a, b) and (c, b).

We want to highlight that the parts involved in the debate used for Example 3 do not take turns as happens for traditional dialogue games, but each agent asynchronously executes CLA procedures defining its behaviour. Some advantages of this approach are that agents neither rely on a synchronised system clock (which may not be available in distributed environments) nor need to wait for specific actions to terminate in order to achieve coordination with other agents. Parallel constructs allow for the execution of multiple actions at the same time, which, for instance, in the context of the Internet of things, may translate to having multiple devices involved in simultaneous argumentation processes.

4.2. *Negotiation with* CLA

Negotiation is a process that aims to solve conflicts arising from the interaction between two or more parties that have different individual goals (for instance, a request of computational resources in a distributed network), and its outcome is an agreement that translates in common benefits for all participants [3]. In order to conduct a negotiation, intelligent agents must be given the capability to change the conditions that meet their goals.

Example 4. We describe an example in which two agents, a client A and a provider B, negotiate over desired parameters for an internet connection. The terms of negotiation are established on the bandwidth (measured in Mbps). The process begins with agent A asking for a connection with at least 100 Mbps of bandwidth to ensure a good streaming quality (argument a in the AF of Fig. 11). The provider has its initial proposal set to 50 Mbps (argument b) as for all new customers. Being the provider's offer incompatible with the client's request, the arguments supporting their proposals are in conflict. In particular, we see in Fig. 11 that a and b attack each other. At this point, both the agents retract their initial

Fig. 11. Two AFs representing the evolution of a negotiation.

Table 12
Example of a CLA program for negotiation

$A : add(\{a\}, \{\}) \longrightarrow check^{w}(\{b\}, \{\}) \longrightarrow add(\{\}, \{(a, b)\}) \longrightarrow c\text{-}test^{f}(a, \texttt{IN}, gde) \longrightarrow success$
$\quad +_P\ c\text{-}test^{f}(a, \texttt{OUT/UNDEC}, gde) \longrightarrow rmv(\{a\}, \{\}) \longrightarrow add(\{a'\}, \{\}) \longrightarrow c\text{-}test^{f}(a', \texttt{IN}, gde) \longrightarrow success$
$B : add(\{b\}, \{\}) \longrightarrow check^{w}(\{a\}, \{\}) \longrightarrow add(\{\}, \{(b, a)\}) \longrightarrow c\text{-}test^{f}(b, \texttt{IN}, gde) \longrightarrow success$
$\quad +_P\ c\text{-}test^{f}(b, \texttt{OUT/UNDEC}, gde) \longrightarrow rmv(\{b\}, \{\}) \longrightarrow add(\{b'\}, \{\}) \longrightarrow c\text{-}test^{f}(b', \texttt{IN}, gde) \longrightarrow success$
$P : A \| B$

bandwidth proposals: A is now willing to accept 70 Mbps (argument a' of Fig. 10), while B decides to concede up to 80 Mbps (argument b'). The two arguments a' and b' are not in conflict, therefore no attack is added between them. Using again the grounded semantics for testing the acceptability, we can see that both a' and b' are acceptable together, so the two agents have reached an agreement. Note that if the two agents cannot find a better agreement than the starting one, the negotiation could be interrupted and end in a stalemate. The conversation between client and provider is summarised as follows.

a: "I will be using streaming services, so I will ask for 100 Mbps of bandwidth to feel comfortable."
b: "I offer 50 Mbps to all new customers, thus I will make you the same offer."
a′: "I do not plan to use the entire 100 Mbps bandwidth, but 50 Mbps is not enough for my needs. Therefore, I will not accept less than 70 Mbps."
b′: "It is in my interest to get new customers, so I can grant up to 80 Mbps."

We show in Table 12 a CLA program realising the negotiation described in this example. Note that we write $c\text{-}test^{f}(a, \texttt{OUT/UNDEC}, gde)$, in which we specify two labels at once, in place of two consecutive credulous tests, one for the label `IN` and one for the label `OUT`.

5. Implementation

To facilitate the use of the tool we develop a web interface exposing the functionalities of our language. In this section we provide a description for the interface, followed by insights on the implementation of CLA itself.

5.1. Web interface

The interface consists of a web page[4] divided into three main areas (shown in Fig. 12): an input form, a text box for the program output and another text box for the shared AF. The output of our tool shows, for each step, the executed operation and the remaining part of the program, together with the results of check and test operations.

The user can either manually input a program in the designated area or select a sample program from those available a the drop down menu. Two buttons below the input area run the program and display the

[4]Web interface available at https://conarg.dmi.unipg.it/cla/.

Fig. 12. Execution of the program in Example 5.

result in different ways. Clicking the button "Run all", the result of the whole program is immediately displayed in the area below and the AF shown on the right represents the final state of the shared store. On the other hand, the button "Run 1 step" shows, as the name suggests, one step at time: each click on the button makes another step of the execution appear in the output area. The AF on the right side is updated after each `add` or `rmv` operation, showing the evolution of the underlying argumentation space. Note that the difference between the two usable modes is only in the visualisation, since both compute the whole result beforehand. Regardless of the chosen method, the executed operation is highlighted in yellow in each line of the output.

Example 5 (Parallel actions)**.** Consider the program below.

```
checkw({c},{}) -> add({a,b},{(a,c)}) -> success ||
add({c},{}) -> success;
```

Running the program produces the results in Fig. 12. Note that the AF representing the argumentation space is always empty at the beginning. In line 1 of the output, the parser recognises a valid program. Two threads (one for each action) are started. In this example, the action that occurred first in the program is also executed first, but in general it can happen in any order. In line 3, the program executes a waiting `checkw`: if the AF contains an argument c then the visit on that branch can continue (and the add operation is executed). Otherwise, the `checkw` is repeated until it (possibly) becomes true. Since the AF is empty by default and no other action has modified it yet, the check on the AF return a negative

answer (line 4). In the meanwhile, the `add` operation of the second thread is executed in line 6. The AF is modified accordingly, introducing an argument c. AF $= \langle\{c\}, \{\}\rangle$. This branch of the execution terminates in line 7 with a success. At this point, the check of the first thread (which had previously given negative results) is repeated, this time giving an affirmative answer (lines 8 and 9). The execution then continues in line 10 with the `add` operation which produces further modifications on the AF. At this point, AF $= \langle\{c, a, b\}, \{(a, c)\}\rangle$. This branch successfully terminates in line 11 and since both the parallel actions of our program succeed, the whole program terminates with a success (line 12).

Example 6 (Nondeterminism)**.** We have the following program with a parallel composition and a nondeterministic operation.

```
add({a,b},{}) -> sum(
checkw({c},{}) -> add({},{(c,a)}) -> success,
testcw({a},in,complete) -> rmv({b},{}) -> success
) ||
add({c},{}) -> success;
```

It is possible to obtain different outcomes according to the order in which the thread handling the parallelism are executed. We show an example in Fig. 13.

After identifying the program in line 1 and the parallel composition in line 2, the visit of the tree proceeds with the execution of the `add` operation of the first thread, which introduces in the AF two new arguments, namely a and b (line 3). AF $= \langle\{a, b\}, \{\}\rangle$. The node corresponding to a nondeterministic choice on the same thread is visited immediately after in line 4. It is important to note that our implementation of the `sum` inspects all the guards on child nodes and selects a verified one (if any) at random. In the program we are analysing, the child of `sum` are `checkw({c},{})` and

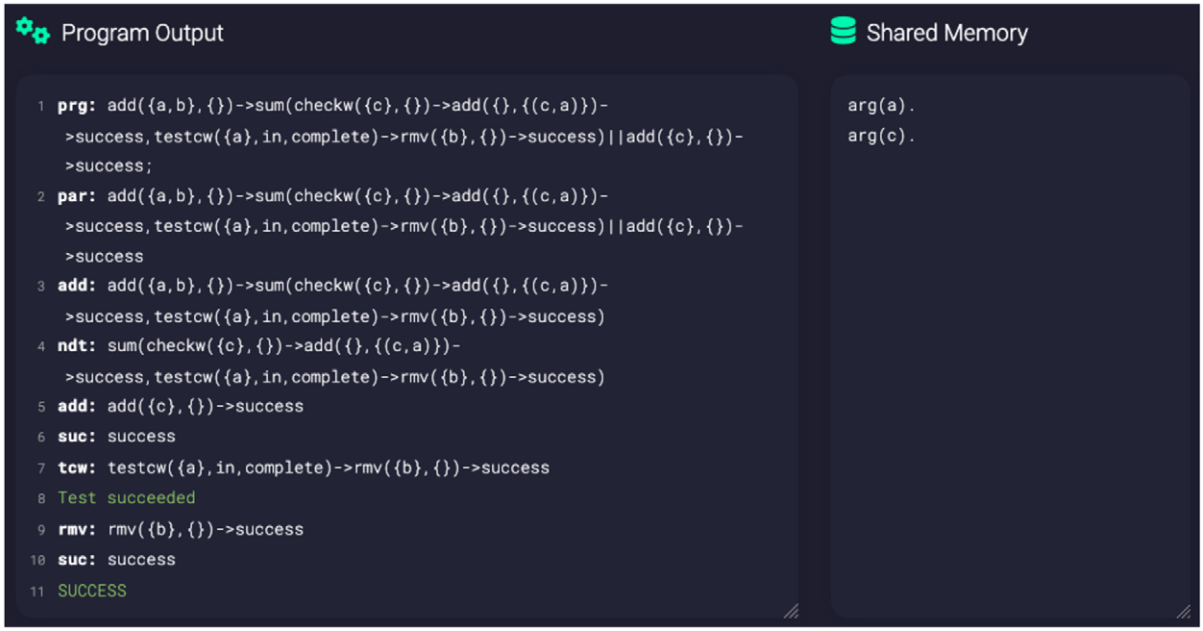

Fig. 13. Execution of a the CLA program in Example 6.

Fig. 14. Execution of the program in Example 7.

`testcw({a},in,complete)`, and only the latter is true at the time of the verification, meaning the former will be ignored. Then the program continues executing the other thread, which adds an argument c to the AF and terminates with a success (lines 5 and 6). AF $= \langle\{a, b, c\}, \{\}\rangle$. At this point, `checkw({c},{})` becomes true, but the choice on which expression will be executed has already been made. The remaining thread resumes its execution performing the `testcw` operation (line 7). The waiting test succeeds on the first try in line 8, leading to the removal of argument b (line 9), as specified by the parse tree. Now we have AF $= \langle\{a, c\}, \{\}\rangle$. The branch and the whole program also succeed (lines 10 and 11).

Example 7 (If-then-else)**.** We run the following program, whose result is shown in Fig. 14.

```
add({a,b},{(a,b)}) ->
checkf({c},{}) -> add({d},{}) -> success +P
testcf({b},in,complete) -> add({e},{}) -> success;
```

After initialising the AF with two arguments and an attack between them in line 3 (AF $= \langle\{a, b\}, \{(a, b)\}\rangle$.), the program executes an if-then-else construct (line 4). The first condition consists of a `checkf` operation, which immediately fails (lines 5 and 6). The program proceeds with the second condition, this time a `testcf`, that also fails (lines 7 and 8). Since both conditions fail, also the program terminates with a failure in line 9. We remark that more than two conditions can be declared by the use of `+P` and only the last one can be a waiting expression.

5.2. CLA *parser and synchronisation*

We implemented our language using python and ANTLR[5] [38], a parser generator for reading, processing, executing, and translating structured text. Starting from a grammar, ANTLR generates a parser that can build and walk parse trees. ANTLR provides two ways of traversing the syntax tree: either

[5] ANTLR website: https://www.antlr.org.

through a listener (the default option) or a visitor. The biggest difference between the listener and visitor mechanisms is that listener methods are called independently, whereas visitor methods must walk their children with explicit visit calls. Not invoking visitor methods on the children of a node means those subtrees are not visited. Since we want to implement guards in our language, we need the possibility to decide which part of the tree will be visited, making our choice fall on the visitor approach. Our project consists of a grammar file and seven python classes, the most interesting being the *CustomVisitor*, in which we define the behaviour of the parser, and the class *ArgFun* containing all the auxiliary argumentation-related functions used to process the argumentation space of the agents (that is, indeed, an AF). We define our grammar using the syntax given in Table 2 and we obtain a .g4 file of which we show the main part in Table 13. Capitalized words are placeholder for terminals specifying syntactic elements of the language: for instance, `ARROW` stands for the symbol `->`, `PAR` corresponds to ∥, and `ARGS` is any list of literals enclosed in curly brackets.

Starting from the grammar, ANTLR automatically generates all the components we will use for parsing the language, the most remarkable being the list of used tokens, the interpreter containing names for literals and rules and symbolic names for the tokens, a lexer which recognises input symbols from a character stream, the parser itself (endowed with all the necessary support code) and the visitor class. Then, we need to manually override the default methods provided in the visitor to customise the behaviour of the parser. The visit of the parse tree always starts with the execution of the function *visitPrg*, which recursively visits all its children. The parser recognises twenty types of node (the non-terminal elements in the grammar), identified through a three-letter code preceded by # (see Table 13). These codes are then used as a shortcut to recall nodes for which we want to specify a desired behaviour. Below, we provide details on the implementation of visiting functions.

- *visitPrg*: calls the visit on its children, collects the results and, in case of termination, returns the output of the whole program.
- *visitPar*: starts two separated threads to execute (visit) two actions in parallel, returning true if both succeeds, false if at least one action fails, and suspends if an action is waiting for its guard to become true.
- *visitAdd* and *visitRmv*: modify the AF by either adding or removing part of the AF, respectively. Always succeeds and continues on the children. Note that *visitRmv* succeeds also if the specified arguments and/or attacks are not in the AF. In that case, the AF is left unchanged.
- *visitSuc* and *visitFlr*: correspond to visits to terminal nodes and return true (success) and false (failure), respectively.
- *visitNdt*: implements a concatenation of + operators, inspecting the guards of all its children and randomly selecting a branch to execute among the possible ones. A guard can be a waiting check or either of the waiting tests. If no guards are found with satisfiable conditions, *visitNdt* waits for changes in the AF until some child can be executed.
- *visitGpa*: implements a concatenation of $\|_G$ operators and execute all its children in separated threads. Contrary to *visitNdt*, *visitGpa* only works with expressions that can fail (and do not suspend), thus allowing for two possible outcomes, that is success if at least one expression succeeds, and failure if all expressions fail.
- *visitIte*: behaves like an if-then-else construct. The first child must be an expression with guaranteed termination (either success or failure). The children are executed in the same order in which they are specified and, as soon as a satisfiable guard is found, the corresponding branch is executed. Since some child can be a waiting expression, *visitIte* is not guaranteed to terminate.

Table 13
Part of .g4 file specifying the CLA grammar

```
grammar CA;

program
:   par_action SEMICOLON                                          #prg
;

par_action
:   action (PAR action)*                                          #par
;

action
:   '(' action ')'                                                #pac
|   'add(' (EMP | ARGS) ',' (EMP | ATKS) ')' ARROW action        #add
|   'rmv(' (EMP | ARGS) ',' (EMP | ATKS) ')' ARROW action        #rmv
|   expression                                                    #exp
|   SUCCESS                                                       #suc
|   FAILURE                                                       #flr
;

expression
:   '(' expression ')'                                            #pex
|   expression_w                                                  #exw
|   expression_f                                                  #exf
|   'sum(' expression_w (',' expression_w)* ')'                   #ndt
|   'gpar(' expression_f (',' expression_f)* ')'                  #gpa
|   expression_f (PPLUS expression)*                              #ite
;

expression_w
:   'checkw(' (EMP | ARGS) ',' (EMP | ATKS) ')' ARROW action     #ckw
|   'testcw(' (EMP | ARGS) ',' LABEL ',' SEM ')' ARROW action    #tcw
|   'testsw(' (EMP | ARGS) ',' LABEL ',' SEM ')' ARROW action    #tsw
;

expression_f
:   'checkf(' (EMP | ARGS) ',' (EMP | ATKS) ')' ARROW action     #ckf
|   'testcf(' (EMP | ARGS) ',' LABEL ',' SEM ')' ARROW action    #tcf
|   'testsf(' (EMP | ARGS) ',' LABEL ',' SEM ')' ARROW action    #tsf
;

ARROW
:   '->'
;

...
```

- *visitCkw* and *visitCkf*: check if a given set of arguments and/or attacks is present in the argumentation space. In case of success, both nodes visit the consequent action. On the other hand, when the argumentation space does not contain the specified parts of AF, *visitCkw* waits for the condition to become true, while *visitCkf* immediately returns false and leads to branch failure.

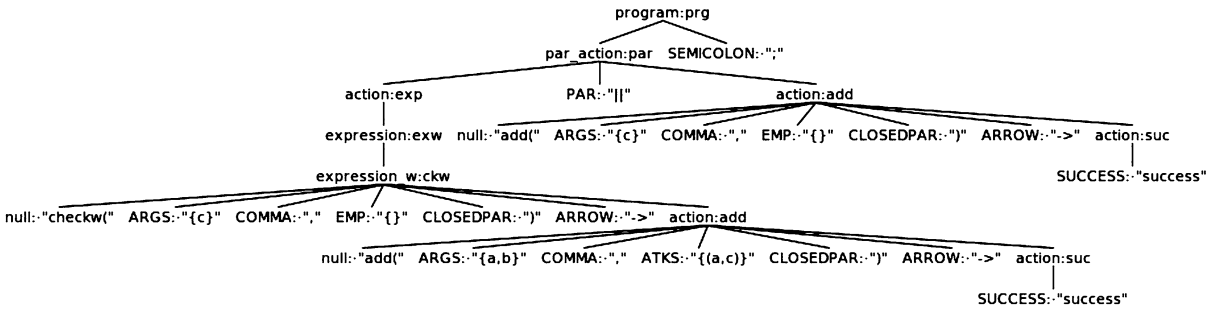

Fig. 15. Parse tree of the CLA program in Example 5.

- *visitTcw*, *visitTcf*, *visitTsw* and *visitTsf*: call a solver[6] to execute credulous and sceptical tests on the acceptability of a given set of arguments. As with the checks, the test functions are also available in two versions, one that always terminates (with either a success or a failure) and the other that possibly suspends and waits for the condition to become true.

In addition to the visiting functions, we have a set of core functions responsible for managing auxiliary tasks, like starting new threads when a parallel composition is detected, making changes to the shared AF and computing the semantics for the test operations. All the components are put together in the *Main* class, which takes in input and runs the user-defined program. First of all, the input stream (a string containing the definition of the program to run) is passed to the lexer, which extracts the tokens and sends them to the parser. Then, the parser uses the tokens to generate a tree ready to be traversed (see Fig. 15 for an example.). Finally, the visitor walks the tree and executes the program.

The synchronisation between parallel CLA agents is obtained in form of *interleaving*, i.e., only one agent is handled by the processor at a time. To implement the interleaving approach in CLA we use the functionality provided by Python's multiprocessing package. In particular, we rely on two fundamental components to manage the synchronisation of parallel processes: *threads* and *shared variables*. First, when the parser detects a construct that requires the parallel execution of two branches, separate threads are started whose management and scheduling are then automatically delegated to the processor. In each thread, the execution of the CLA program continues independently, with the possibility of starting further parallel processes. Secondly, access to the shared AF is also managed so as not to cause the generation of inconsistent information. The AF itself is stored in a shared variable, access to which can be managed with a lock-and-unlock system: when any thread wants to read or write the contents of this variable, it must first request control over it, which is granted to only one thread at a time. Control is then released at the end of the operation. Read and write operations on the AF thus become atomic.

The implementation of CLA can be used for both research purposes and practical applications. Although in this paper we are not going to address the issues arising from the computational complexity of argumentation problems [33], we want to point out that efficient implementations of CLA programs can be achieved by using, for example, the grounded semantics, for which finding and verifying extensions is a task that can be performed in polynomial time [33].

[6]ConArg website: https://conarg.dmi.unipg.it.

6. Related work

A formalism for expressing dynamics in AFs is defined in [43] as a *Dynamic Argumentation Framework* (DAF). The aim of that paper is to provide a method for instantiating Dung-style AFs by considering a universal set of arguments U. A DAF consists of an AF $\langle U, R\rangle$ and a set of evidence, which has the role of restricting $\langle U, R\rangle$ to possible arguments and relations, so to obtain a static instance of the framework. DAFs are built starting from argumental structures, in which a tree of arguments supports a claim (corresponding to the root of the tree), and then adding attacks between argumental structures. The dynamic component of a DAF is thus the set of evidence. The introduced approach allows for generalising AFs, adding the possibility of modelling changes, but, contrary to our study, it does not consider how such modifications affect the semantics and does not allow to model the behaviour of concurrent agents.

The impact of modifications on an AF in terms of sets of extensions is studied in [26]. Different kinds of revision are introduced, in which a new argument interacts with an already existing one. The authors describe different kinds of revision differing in the number of extensions that appear in the outcome, with respect to a semantics: a *decisive* revision allows to obtain a unique non-empty extension, a *selective* revision reduces the number of extensions (to a minimum of two), while a *questioning* one increases that number; a destructive *revision* eliminates all extensions, an *expansive* revision maintain the number of extensions and increases the number of accepted arguments; a *conservative* revision does not introduce changes on the semantics level, and an *altering* revision adds and deletes arguments in the extensions. All these revisions are obtained through the addition of a single argument, together with a single attack relation either towards or from the original AF, and can be implemented as procedures of our language. The review operator we define in the syntax of our language (as the other two operator for expansion and contraction), instead, does not consider whole extensions, but just an argument at a time, allowing communicating agents to modify their beliefs in a finer grain.

Focusing on syntactic expansion of an AF (the mere addition of arguments and attacks), [8] show under which conditions a set of arguments can be enforced (to become accepted) for a specific semantics. Moreover, since adding new arguments and attacks may lead to a decrease in term of extensions and accepted arguments, the authors also investigate whether an expansion behaves in a monotonic fashion, thus preserving the status of all originally accepted arguments. The study is only conducted on the case of weak expansion (that adds further arguments which do not attack previous arguments). The notion of expansion we use in the presented work is very different from that in [8]. First of all, we take into account semantics when defining the expansion, making it more similar to an enforcement itself: we can increment the labels of an argument so to match a desired acceptance status. Therefore, our expansion turns out to be more general, being able to change the status of a given topic not only accepted but also rejected, indecisive or indeterminate. This is useful, for instance, when we want to diminish the beliefs of an opponent agent.

Enforcing is also studied in [28], where the authors consider an expansion of the AF that only allows the addition of new attack relations, while the set of arguments remains the same (differently from [8]). It is shown, indeed, that if no new argument is introduced, it is always possible to guarantee the success of enforcement for any classical semantics. Also in this case, we want to highlight the differences with our work. Starting from the modifications allowed into the framework, we are not limited to only change the set of relations, since we implement procedures that also add and remove arguments. Moreover, the operators we define are not just enforcement operators, since they allow to modify the acceptability status of a single argument of an AF.

In our model, AFs are equipped with a universe of arguments that agents use to add new information in the knowledge base. The problem of combining AFs is addressed in [9], that study the computational complexity of verifying if a subset of argument is an extension for a certain semantics in incomplete argumentation frameworks obtained by merging different beliefs. The incompleteness is considered both for arguments and attack relation. Similarly to our approach, arguments (and attacks) can be brought forward by agents and used to build new acceptable extensions. On the other hand, the scope of [9] is focused on a complexity analysis and does not provide implementations for the merging.

The authors of [36] introduce a model for representing the mental states of agents. In the proposed setting, argumentation is used to express changes in agents' intentions produced as a consequence of interaction processes, with a particular focus on negotiation. The mental model of an agent is defined via a specifically designed logical model, which is then used to evaluate the beliefs through argumentation semantics. This logic is intended to allow one to examine single agents rather than the interaction between multiple agents. In this aspect, and in the fact that only negotiation is considered as a possible communicative process, the work in [36] diverges from ours, which, instead, aims at providing a flexible framework for modelling any kind of (possibly concurrent) interaction, using argumentation to handle beliefs belonging to all the involved agents.

The integration of new agents (and therefore new beliefs) in an existing system is a challenging problem that needs to be addressed in order to enable dynamics in open multi-agent systems. The concurrent programming language of [47] builds upon the *Ask-and-Tell* paradigm [44] to allow a form of communication between agents which can resort to a parametric belief revision operator to adjust their beliefs and integrate additional knowledge. Differently from ours, such a language focuses on the exchange of information between agents without providing any mechanism for reasoning on shared information, therefore precluding the possibility of modelling protocols like negotiation and persuasion.

Different frameworks have been adopted to model communication processes in multi-agent systems. Among those frameworks, Petri Nets offer the capability of representing concurrent interactions, besides verifying particular properties (like reachability and liveness) related to agents' behaviour. In [27], Coloured Petri Nets constitute the basis of a language for conversation specification. A conversation is intended here as a sequence of actions that can also happen simultaneously, and that can realise protocols like negotiation. However, the authors do not consider the notion of beliefs belonging to agents in the system and they only aim at modelling the series of actions that correspond to a certain form of interaction.

The possibility of concurrent actions performed by agents is also contemplated in [5], where Dung-style AFs are extended to directly integrate a notion of persuasion. In the resulting formalism, called Abstract Persuasion Argumentation Framework (APA), the classical notion of defence is reinterpreted to accommodate a broader meaning, including the ability to persuade as well as defend. In particular, arguments in an APA can be converted (and thus transformed) into other arguments. A subsequent work [4] further extends APAs with numerical values which also make it possible to represent resource allocations and conditional relations between arguments. Even if persuasion is presented in these works as dynamic relation, APAs (and their extension) cannot represent dynamic interactions between agents, which we can model instead through the constructs of our language. Moreover, in our general setting, not only negotiation as in [4,5], but any process between multiple agents can be modelled.

A timed version of CLA has been studied in [12,13] with the introduction of constructs allowing for the specification of temporal intervals in which actions occur. Expressions, for instance, are endowed with timeouts that, once expired, make the execution terminate with failure. This behaviour could better represents real-world situations in which timed applications cannot indefinitely wait for an event to

happen. Concurrent operations, then, are modelled following a maximum parallelism approach (i.e., it is assumed that there are infinite processors, and all parallel operations can be performed simultaneously). An interleaving model on a single processor is adopted instead in [14] for basic CLA computation steps. Contrary to maximum parallelism, the interleaving approach limits the number of enabled agents executed at a time, mimicking the limited number of available processors as in the real world: only one of the enabled agents is executed at each instant of time, while all the other agents may have to wait for the processor to be free.

7. Conclusion and future work

We introduced a concurrent language for argumentation, that can be used by (intelligent) agents to implement different forms of communication. The agents involved in the process share an AF that serves as an argumentation space and where arguments represent the agreed beliefs. The shared AF can be modified via a set of primitives that allow the addition and removal of arguments and attacks. All agents have at their disposal a universe of arguments to choose from when they need to introduce new information. Besides operations at a syntactic level, we also defined semantic operations that verify the acceptability of the arguments in the store. The functioning of all CLA operations was described in detail and shown through explanatory examples, emphasising how the parallel execution of multiple processes takes place. Finally, we presented a tool (also endowed with a web interface) for modelling concurrent argumentation processes written in CLA, giving insights on the implementation choices and describing the main components of the tool.

For the future, we plan to extend this work in many directions. First of all, given the known issues of abstract argumentation [41], we want to consider (semi-)structured AFs, e.g., CAFs [34], and provide an implementation for our expansion, contraction and revision operators, for which a different store (structured and not abstract, indeed) need to be considered. The concurrent primitives are already general enough and do not require substantial changes.

On the operations level, we are currently only able to modify the acceptance status of the arguments, without further considerations on the obtained semantics. To gain control also over changes on the set of extensions, we want to introduce operators able to obtain a specified semantics (when possible) or to leave it unchanged (this can be done relying on the notion of robustness [15]).

Then, we would like to investigate the relation between the revision operations that can be implemented in CLA and the AGM postulates for belief revision [1]. Following this direction, we could devise a set of AGM-style operations that allow for modifying an AF (the shared memory our agents access to communicate) and changing the status of its arguments so as to allow negotiation and the other forms of dialogues. Chatbots using argumentation techniques to interact with the users could benefit from this approach.

As a final consideration, whereas in real-life cases it is always clear which part involved in a debate is stating a particular argument, AFs do not hold any notion of "ownership" for arguments or attacks, that is, any bond with the one making the assertion is lost. To overcome this problem, we want to implement the possibility of attaching labels on (groups of) arguments and attacks of AFs, in order to preserve the information related to who added a certain argument or attack, extending and taking into account the work in [37]. Consequently, we can also obtain a notion of locality (or scope) of the belief in the argumentation space: arguments owned by a given agents can be placed into a local store and used in the implementation of specific operators through hidden variables.

Acknowledgement

We thank the anonymous reviewers for their insightful comments and valuable suggestions. Stefano Bistarelli and Carlo Taticchi are members of the INdAM Research group GNCS and of Consorzio CINI. This work has been partially supported by: GNCS-INdAM, CUP E55F22000270001; Project RACRA – funded by Ricerca di Base 2018-2019, University of Perugia; Project BLOCKCHAIN4FOODCHAIN: funded by Ricerca di Base 2020, University of Perugia; Project DopUP – REGIONE UMBRIA PSR 2014-2020; Project GIUSTIZIA AGILE, CUP: J89J22000900005.

References

[1] C.E. Alchourrón, P. Gärdenfors and D. Makinson, On the logic of theory change: Partial meet contraction and revision functions, *The Journal of Symbolic Logic* **50**(02) (1985), 510–530. doi:10.2307/2274239.

[2] L. Amgoud and F.D. de Saint-Cyr, Extracting the core of a persuasion dialog to evaluate its quality, in: *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, C. Sossai and G. Chemello, eds, Springer, Berlin Heidelberg, 2009, pp. 59–70. ISBN 978-3-642-02906-6. doi:10.1007/978-3-642-02906-6_7.

[3] L. Amgoud and S. Vesic, A formal analysis of the role of argumentation in negotiation dialogues, *J. Log. Comput.* **22**(5) (2012), 957–978. doi:10.1093/logcom/exr037.

[4] R. Arisaka and T. Ito, Numerical abstract persuasion argumentation for expressing concurrent multi-agent negotiations, in: *Artificial Intelligence. IJCAI 2019 International Workshops – Macao, Revised Selected Best Papers*, China, August 10–12, 2019, A.E.F. Seghrouchni and D. Sarne, eds, Lecture Notes in Computer Science, Vol. 12158, Springer, 2019, pp. 131–149. doi:10.1007/978-3-030-56150-5_7.

[5] R. Arisaka and K. Satoh, Abstract argumentation/persuasion/dynamics, in: *PRIMA 2018: Principles and Practice of Multi-Agent Systems – 21st International Conference, Proceedings*, Tokyo, Japan, October 29–November 2, 2018, T. Miller, N. Oren, Y. Sakurai, I. Noda, B.T.R. Savarimuthu and T.C. Son, eds, Lecture Notes in Computer Science, Vol. 11224, Springer, 2018, pp. 331–343. doi:10.1007/978-3-030-03098-8_20.

[6] P. Baroni, M. Caminada and M. Giacomin, An introduction to argumentation semantics, *Knowledge Eng. Review* **26**(4) (2011), 365–410. doi:10.1017/S0269888911000166.

[7] P. Baroni and M. Giacomin, On principle-based evaluation of extension-based argumentation semantics, *Artif. Intell.* **171**(10–15) (2007), 675–700. doi:10.1016/j.artint.2007.04.004.

[8] R. Baumann and G. Brewka, Expanding argumentation frameworks: Enforcing and monotonicity results, in: *Computational Models of Argument: Proceedings of COMMA 2010, Desenzano del Garda*, Italy, September 8–10, 2010, P. Baroni, F. Cerutti, M. Giacomin and G.R. Simari, eds, Frontiers in Artificial Intelligence and Applications, Vol. 216, IOS Press, 2010, pp. 75–86.

[9] D. Baumeister, D. Neugebauer, J. Rothe and H. Schadrack, Verification in incomplete argumentation frameworks, *Artif. Intell.* **264** (2018), 1–26. doi:10.1016/j.artint.2018.08.001.

[10] S. Bistarelli, L. Kotthoff, F. Santini and C. Taticchi, A first overview of ICCMA'19, in: *Proceedings of the Workshop on Advances In Argumentation In Artificial Intelligence 2020 co-located with the 19th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2020)*, Online, November 25–26, 2020, B. Fazzinga, F. Furfaro and F. Parisi, eds, CEUR Workshop Proceedings, Vol. 2777, CEUR-WS.org, 2020, pp. 90–102.

[11] S. Bistarelli, L. Kotthoff, F. Santini and C. Taticchi, Summary report for the third international competition on computational models of argumentation, *AI Mag.* **42**(3) (2021), 70–73. doi:10.1609/aimag.v42i3.15109.

[12] S. Bistarelli, M.C. Meo and C. Taticchi, Timed concurrent language for argumentation, in: *Proceedings of the 36th Italian Conference on Computational Logic*, Parma, Italy, September 7–9, 2021, S. Monica and F. Bergenti, eds, CEUR Workshop Proceedings, Vol. 3002, CEUR-WS.org, 2021, pp. 1–15, http://ceur-ws.org/Vol-3002/paper11.pdf.

[13] S. Bistarelli, M.C. Meo and C. Taticchi, Concurrent argumentation with time: An overview, in: *Proceedings of the 5th Workshop on Advances in Argumentation in Artificial Intelligence 2021 Co-Located with the 20th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2021)*, Milan, Italy, November 29th, 2021, M. D'Agostino, F.A. D'Asaro and C. Larese, eds, CEUR Workshop Proceedings, Vols 3086, CEUR-WS.org, 2021, http://ceur-ws.org/Vol-3086/short3.pdf.

[14] S. Bistarelli, M.C. Meo and C. Taticchi, Timed concurrent language for argumentation: An interleaving approach, in: *Practical Aspects of Declarative Languages – 24th International Symposium, PADL 2022, Proceedings*, Philadelphia, PA, USA, January 17–18, 2022, J. Cheney and S. Perri, eds, Lecture Notes in Computer Science, Vol. 13165, Springer, 2022, pp. 101–116. doi:10.1007/978-3-030-94479-7_7.

[15] S. Bistarelli, F. Santini and C. Taticchi, On looking for invariant operators in argumentation semantics, in: *Proceedings of the Thirty-First International Florida Artificial Intelligence Research Society Conference, FLAIRS 2018*, Melbourne, Florida, USA, May 21–23, 2018, 2018, pp. 537–540.
[16] S. Bistarelli and C. Taticchi, A concurrent language for argumentation, in: *Proceedings of the Workshop on Advances in Argumentation in Artificial Intelligence 2020 Co-Located with the 19th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2020)*, Online, November 25–26, 2020, B. Fazzinga, F. Furfaro and F. Parisi, eds, CEUR Workshop Proceedings, Vol. 2777, CEUR-WS.org, 2020, pp. 75–89.
[17] S. Bistarelli and C. Taticchi, Towards an implementation of a concurrent language for argumentation, in: *AIxIA 2020 – Advances in Artificial Intelligence – XIXth International Conference of the Italian Association for Artificial Intelligence, Virtual Event, November 25–27, 2020, Revised Selected Papers*, M. Baldoni and S. Bandini, eds, Lecture Notes in Computer Science, Vol. 12414, Springer, 2020, pp. 154–171. doi:10.1007/978-3-030-77091-4_10.
[18] S. Bistarelli and C. Taticchi, A concurrent language for argumentation: Preliminary notes, in: *Recent Developments in the Design and Implementation of Programming Languages, Gabbrielli's Festschrift, November 27, 2020*, F.S. de Boer and J. Mauro, eds, OASIcs, Vol. 86, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Bologna, Italy, 2020, pp. 9:1–9:22. doi:10.4230/OASIcs.Gabbrielli.9.
[19] S. Bistarelli and C. Taticchi, Introducing a tool for concurrent argumentation, in: *Logics in Artificial Intelligence – 17th European Conference, JELIA 2021, Virtual Event, May 17-20, 2021, Proceedings*, W. Faber, G. Friedrich, M. Gebser and M. Morak, eds, Lecture Notes in Computer Science, Vol. 12678, Springer, 2021, pp. 18–24. doi:10.1007/978-3-030-75775-5_2.
[20] S. Bistarelli and C. Taticchi, A unifying four-state labelling semantics for bridging abstract argumentation frameworks and belief revision, in: *Proceedings of the 22nd Italian Conference on Theoretical Computer Science*, Bologna, Italy, September 13–15, 2021, C.S. Coen and I. Salvo, eds, CEUR Workshop Proceedings, Vol. 3072, CEUR-WS.org, 2021, pp. 93–106, http://ceur-ws.org/Vol-3072/paper8.pdf.
[21] G. Boella, S. Kaci and L.W.N. van der Torre, Dynamics in argumentation with single extensions: Attack refinement and the grounded extension (extended version), in: *Argumentation in Multi-Agent Systems, 6th International Workshop, ArgMAS 2009. Revised Selected and Invited Papers*, Lecture Notes in Computer Science, Vol. 6057, Springer, 2009, pp. 150–159. ISBN 978-3-642-12804-2.
[22] G. Boella, S. Kaci and L.W.N. van der Torre, Dynamics in argumentation with single extensions: Abstraction principles and the grounded extension, in: *Proceedings*, Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 10th European Conference, ECSQARU 2009, Verona, Italy, July 1–3, C. Sossai and G. Chemello, eds, Lecture Notes in Computer Science, Vol. 5590, Springer, 2009, pp. 107–118. doi:10.1007/978-3-642-02906-6_11.
[23] M. Caminada, *Semi-Stable Semantics, in: Computational Models of Argument: Proceedings of COMMA 2006, September 11–12, 2006*, P.E. Dunne and T.J.M. Bench-Capon, eds, Frontiers in Artificial Intelligence and Applications, Vol. 144, IOS Press, Liverpool, UK, 2006, pp. 121–130, http://www.booksonline.iospress.nl/Content/View.aspx?piid=1932.
[24] M. Caminada, On the issue of reinstatement in argumentation, in: *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Proceedings, Lecture Notes in Computer Science*, Liverpool, UK, September 13–15, 2006, Vol. 4160, Springer, 2006, pp. 111–123. ISBN 978-3-540-39625-3.
[25] M. Caminada and B. Verheij, On the existence of semi-stable extensions, in: *Proceedings of the 22nd Benelux Conference on Artificial Intelligence (BNAIC 2010)*, 2010.
[26] C. Cayrol, F.D. de Saint-Cyr and M.-C. Lagasquie-Schiex, Revision of an argumentation system, in: *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference, KR 2008*, Sydney, Australia, September 16–19, 2008, AAAI Press, 2008, pp. 124–134. ISBN 978-1-57735-384-3.
[27] R.S. Cost, Y. Chen, T.W. Finin, Y. Labrou and Y. Peng, Using colored Petri nets for conversation modeling, in: *Issues in Agent Communication*, F. Dignum and M. Greaves, eds, Lecture Notes in Computer Science, Vol. 1916, Springer, 2000, pp. 178–192. doi:10.1007/10722777_12.
[28] S. Coste-Marquis, S. Konieczny, J. Mailly and P. Marquis, Extension enforcement in abstract argumentation as an optimization problem, in: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires*, Argentina, July 25–31, 2015, Q. Yang and M.J. Wooldridge, eds, AAAI Press, 2015, pp. 2876–2882.
[29] F.D. de Saint-Cyr, P. Bisquert, C. Cayrol and M. Lagasquie-Schiex, Argumentation update in YALLA (yet another logic language for argumentation), *Int. J. Approx. Reason.* **75** (2016), 57–92. doi:10.1016/j.ijar.2016.04.003.
[30] S. Doutre, A. Herzig and L. Perrussel, A dynamic logic framework for abstract argumentation, in: *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014*, Vienna, Austria, July 20–24, 2014, 2014.
[31] S. Doutre, F. Maffre and P. McBurney, A dynamic logic framework for abstract argumentation: Adding and removing arguments, in: *Advances in Artificial Intelligence: From Theory to Practice – 30th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2017, Proceedings, Part II*, Arras, France, June 27–30, 2017, S. Benferhat, K. Tabia and M. Ali, eds, Lecture Notes in Computer Science, Vol. 10351, Springer, 2017, pp. 295–305. doi:10.1007/978-3-319-60045-1_32.

[32] P.M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, *Artificial Intelligence* **77**(2) (1995), 321–357. doi:10.1016/0004-3702(94)00041-X.
[33] W. Dvořák and P.E. Dunne, Computational problems in formal argumentation and their complexity, *FLAP* **4**(8) (2017).
[34] W. Dvořák and S. Woltran, Complexity of abstract argumentation under a claim-centric view, *Artif. Intell.* **285** (2020), 103290. doi:10.1016/j.artint.2020.103290.
[35] M. Kacprzak, K. Budzynska and O. Yaskorska, A logic for strategies in persuasion dialogue games, in: *Advances in Knowledge-Based and Intelligent Information and Engineering Systems – 16th Annual KES Conference*, San Sebastian, Spain, 10–12 September 2012, Frontiers in Artificial Intelligence and Applications, Vol. 243, IOS Press, 2012, pp. 98–107. ISBN 978-1-61499-104-5. doi:10.3233/978-1-61499-105-2-98.
[36] S. Kraus, K.P. Sycara and A. Evenchik, Reaching agreements through argumentation: A logical model and implementation, *Artif. Intell.* **104**(1–2) (1998), 1–69. doi:10.1016/S0004-3702(98)00078-2.
[37] N. Maudet, S. Parsons and I. Rahwan, Argumentation in Multi-Agent Systems: Context and Recent Developments, in: *Argumentation in Multi-Agent Systems, Third International Workshop, ArgMAS 2006*, Hakodate, Japan, May 8, 2006, Revised Selected and Invited Papers, 2006, pp. 1–16.
[38] T. Parr, *The Definitive ANTLR 4 Reference, the Pragmatic Bookshelf*, 2013. ISBN 9781934356999.
[39] H. Prakken, Models of persuasion dialogue, in: *Argumentation in Artificial Intelligence*, Springer, 2009, pp. 281–300. ISBN 978-0-387-98196-3. doi:10.1007/978-0-387-98197-0_14.
[40] H. Prakken, An abstract framework for argumentation with structured arguments, *Argument & Computation* **1**(2) (2010), 93–124. doi:10.1080/19462160903564592.
[41] H. Prakken and M.D. Winter, Abstraction in argumentation: Necessary but dangerous, in: *Computational Models of Argument – Proceedings of COMMA 2018*, Warsaw, Poland, 12–14 September 2018, S. Modgil, K. Budzynska and J. Lawrence, eds, Frontiers in Artificial Intelligence and Applications, Vol. 305, IOS Press, 2018, pp. 85–96.
[42] R. Riveret, N. Oren and G. Sartor, A probabilistic deontic argumentation framework, *Int. J. Approx. Reason.* **126** (2020), 249–271. doi:10.1016/j.ijar.2020.08.012.
[43] N.D. Rotstein, M.O. Moguillansky, A.J. Garcia and G.R. Simari, An abstract argumentation framework for handling dynamics, in: *Proceedings of the Argument, Dialogue and Decision Workshop in NMR 2008*, Sydney, Australia, 2008, pp. 131–139.
[44] V.A. Saraswat and M. Rinard, Concurrent constraint programming, in: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages – POPL '90*, ACM Press, San Francisco, California, United States, 1990, pp. 232–245. ISBN 978-0-89791-343-0.
[45] C. Taticchi, A concurrent language for negotiation and debate with argumentation, in: *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event*, United Kingdom, May 3–7, 2021, F. Dignum, A. Lomuscio, U. Endriss and A. Nowé, eds, ACM, 2021, pp. 1840–1841, https://dl.acm.org/doi/10.5555/3463952.3464258.
[46] F. Toni, A tutorial on assumption-based argumentation, *Argument & Computation* **5**(1) (2014), 89–117. doi:10.1080/19462166.2013.869878.
[47] R.M. van Eijk, F.S. de Boer, W. van der Hoek and J.C. Meyer, in: *Open Multi-Agent Systems: Agent Communication and Integration, in: Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 6th International Workshop, ATAL '99, Proceedings*, Orlando, Florida, USA, July 15–17, 1999, N.R. Jennings and Y. Lespérance, eds, Lecture Notes in Computer Science, Vol. 1757, Springer, 1999, pp. 218–232. doi:10.1007/10719619_16.
[48] B. Verheij, Two approaches to dialectical argumentation: Admissible sets and argumentation stages, in: *Proceedings of the Eighth Dutch Conference on Artificial Intelligence (NAIC'96), Utrecht, 1996. Utrecht University*, J.-J.C. Meyer and L.C. van der Gaag, eds, 1996, pp. 357–368.
[49] B. Verheij, A labeling approach to the computation of credulous acceptance in argumentation, in: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence*, Hyderabad, India, January 6–12, 2007, M.M. Veloso, ed., 2007, pp. 623–628, http://ijcai.org/Proceedings/07/Papers/099.pdf.
[50] D. Walton, Types of dialogue, dialectical shifts and fallacies, in: *Argumentation Illuminated*, Proceedings of the International Society for the Study of Argumentation in Amsterdam, SICSAT, 1992, 1992, pp. 133–147.
[51] D. Walton and T.F. Gordon, Argument invention with the Carneades argumentation system, *SCRIPTed* **14** (2017), 168. doi:10.2966/scrip.140217.168.