Timed concurrent language for argumentation with maximum parallelism

STEFANO BISTARELLI, *Department of Mathematics and Computer Science, University of Perugia, Via Luigi Vanvitelli, 1 - 06123 Perugia, Italy.*

MARIA CHIARA MEO, *Department of Economics, University G. d'Annunzio of Chieti-Pescara, Viale Pindaro, 42 - 65127 Pescara, Italy.*

CARLO TATICCHI, *Department of Mathematics and Computer Science, University of Perugia, Via Luigi Vanvitelli, 1 - 06123 Perugia, Italy.*

Abstract

The timed concurrent language for argumentation (tcla) is a framework to model concurrent interactions between communicating agents that reason and take decisions through argumentation processes, also taking into account the temporal duration of the performed actions. Time is a crucial factor when dealing with dynamic environments in real-world applications, where agents must act in a coordinated fashion to reach their own goals. However, modelling complex interactions and concurrent processes may be challenging without the help of proper languages and tools. In this paper, we discuss the use of tcla for practical purposes and provide a working implementation of the language, endowed with a user interface available online, that serves the dual purpose of aiding the research in this field and facilitating the development of multi-agent systems based applications.

Keywords: argumentation theory, timed concurrent language, semantics

1 Introduction

Argumentation theory aims to study how conclusions can be reached starting from a set of assumptions through a process of logical reasoning. This process is very similar to the human way of thinking and involves features which can be traced to a form of dialogue between two (or more) people. Abstract argumentation frameworks [20] (AFs) are used to study the acceptability of arguments according to given selection criteria. Formalisms like timed abstract argumentation frameworks (TAFs) [12, 15, 16] have been proposed to meet the need for including the notion of time into argumentation processes. Time is an essential aspect of cooperative environments: in many 'real-life' applications, the activities have a temporal duration (that can even be interrupted); furthermore, the coordination of such activities has to consider this timeliness property. The interacting actors are mutually influenced by their actions, meaning that agent A reacts accordingly to the timing and quantitative aspects related to the behaviour of agent B and vice versa. Moreover, the information brought forward by debating agents that interact in a dynamic environment can be affected by time constraints, limiting, for instance, the influence of some arguments in the system to a specific time-lapse. Therefore, a mechanism for handling time is required to better model the behaviour of intelligent agents involved in argumentation processes.

This paper deals with the timed concurrent language for argumentation (*tcla*), a formalism able to model dynamic interactions between agents by using notions from argumentation theory for

Advance Access Publication on 25 March 2023 https://doi.org/10.1093/logcom/exad009

reasoning about shared knowledge. In particular, we extend the work in [7–9] with the further addition of three main contributions. First, we carefully discuss how to use *tcla* constructs for modelling TAFs (extended AFs in which arguments are only available for given temporal intervals). An example illustrates step-by-step the procedure to obtain the *tcla* translation of a TAF. Second, we present a working implementation for *tcla* that enables users to write and execute *tcla* programs through a web interface. The description of the implementation includes comments on the chosen architecture and the functioning of the main components. Finally, we offer some practical examples in which we use our system to construct solutions to real-world problems (such as resolving disputes between opposing parties).

tcla is based on the hypothesis of *bounded asynchrony* [31]: any computation takes a bounded period of time rather than being instantaneous as in the concurrent synchronous languages `esterel` [5], `lustre` [22], `signal` [25] and `Statecharts` [23]. Time itself is measured by a discrete global clock, i.e. the internal clock of the *tcla* process. In *tcla*, we directly introduce a timed interpretation of the programming constructs by identifying a time-unit with the time needed for the execution of a primary action (that can be an addition, removal or a check of arguments/attacks or a semantical test of arguments) and by interpreting action prefixing ($\rightarrow$) as the next-time operator. Explicit timing primitives are also introduced to allow for the specification of timeouts. We consider a paradigm in which parallel operations are expressed in terms of maximal parallelism. Following this approach (also adopted in works like [6, 17, 31]), at each moment, every enabled agent of the system is activated, so that the processor simultaneously executes all parallel threads. This is the opposite of the interleaving paradigm, for which at each moment only one of the enabled agents is executed instead.

The rest of this paper is organized as follows: in Section 2, we summarize essential background notions and the frameworks from which *tcla* derives. In Section 3, we present *tcla* and the operational semantics of its agents. Section 4 exemplifies the use of timed paradigms in *tcla* and shows an application example of how a TAF can be dynamically instantiated, highlighting the relation between *tcla* and TAFs. Section 5 describes and shows practical examples of a tool implementing *tcla*. In Section 6, we discuss some related work. Finally, Section 7 concludes the paper by also indicating future research lines.

2 Background

Argumentation theory aims to understand and model the natural human fashion of reasoning, allowing one to deal with uncertainty in non-monotonic (defeasible) reasoning. In his seminal paper [20], Dung defines the building blocks of abstract argumentation.

DEFINITION 2.1 (AFs).
Let U be the set of all available arguments[1] , 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, which denotes a set of attacks between arguments in Arg.

For a pair consisting of a set of arguments and a set of attacks, we define the associated AF as follows.

[1] The set U is not present in the original definition by Dung and we introduce it for our convenience.

DEFINITION 2.2 (Associated AF).
Let F$=\langle Arg, R\rangle$, where Arg and R are a set of arguments and a set of attacks, respectively. We define the AF associated with F as F$\downarrow=\langle Arg, R'\rangle$, where $R' = \{(a,b) \mid (a,b) \in Arg \times Arg \text{ and} (a,b) \in R\}$ is obtained by projecting beforehand the set of attacks to couples of arguments in Arg.

By the previous definition, given a set of arguments Arg and a set of attacks R, $\langle Arg, R\rangle \downarrow$ is an AF, and we say that it is the AF associated with the pair $\langle Arg, R\rangle$. We will need the notion of associated AF in the context of TAFs (where arguments are only available in specific time intervals) to obtain an AF at each given instant. For two arguments $a, b \in Arg$, the notation $(a,b) \in R$ represents an attack directed from a against b. Given an AF, we define the sets of arguments that are attacked by another argument or by a set of arguments as follows.

DEFINITION 2.3 (Attacks).
Let F $= \langle Arg, R\rangle$ be an AF, $a \in Arg$ and $S \subseteq Arg$. We define the sets $a_F^+ = \{b \in Arg \mid (a,b) \in R\}$ and $S_F^+ = \bigcup_{a\in S} a_F^+$ (we will omit the subscript F when it is clear from the context).

DEFINITION 2.4 (Acceptable argument).
Given an AF $F = \langle Arg, R\rangle$, an argument $a \in Arg$ is acceptable with respect to $D \subseteq Arg$ if and only if $\forall b \in Arg$ such that $(b,a) \in R$, $\exists c \in D$ such that $(c,b) \in R$, and we say that a is *defended* from D.

Using the notion of defence as a criterion for distinguishing acceptable sets of arguments (extensions) in the framework, one can further refine the set of selected 'good' arguments. The goal is to establish acceptable arguments according to a certain semantics, namely a selection criterion. Non-accepted arguments are rejected. Different kinds of semantics have been introduced [2, 20] that reflect qualities which are likely to be desirable. We first recall definitions for the extension-based semantics [20], namely admissible, complete, stable, semi-stable, preferred and grounded semantics (denoted with *adm*, *com*, *stb*, *sst*, *prf* and *gde*, respectively, and generically with σ).

DEFINITION 2.5 (Extension-based semantics).
Let $F = \langle Arg, R\rangle$ be an AF. A set $E \subseteq Arg$ is conflict-free in F, denoted $E \in S_{cf}(F)$, when there are no $a, b \in E$ such that $(a,b) \in R$. For $E \in S_{cf}(F)$, we have that

- $E \in S_{adm}(F)$ when each $a \in E$ is defended by E[2] ;
- $E \in S_{com}(F)$ when $E \in S_{adm}(F)$ and $\forall a \in Arg$ defended by E, $a \in E$;
- $E \in S_{stb}(F)$ when $\forall a \in Arg \setminus E$, $\exists b \in E$ such that $(b,a) \in R$;
- $E \in S_{sst}(F)$ when $E \in S_{com}(F)$ and $E \cup E^+$ is maximal (with respect to subset inclusion);
- $E \in S_{prf}(F)$ when $E \in S_{adm}(F)$ and E is maximal (with respect to subset inclusion);
- $E \in S_{gde}(F)$ when $E \in S_{com}(F)$ and E is minimal (with respect to subset inclusion).

Besides enumerating the extensions for a specific 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.

[2] If $E \in S_{adm}(F)$, we say that E is an admissible extension of F. Analogously for the other semantics.

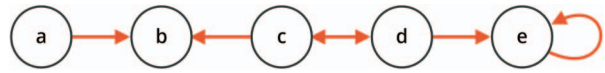

FIGURE 1. Example of an abstract argumentation framework.

EXAMPLE 2.6
In Figure 1, we provide an example of AF where sets of extensions are given for all the mentioned semantics: $S_{cf}(F) = \{\{\}, \{a\}, \{b\}, \{c\}, \{d\}, \{a,c\}, \{a,d\}, \{b,d\}\}$, $S_{adm}(F) = \{\{\}, \{a\}, \{c\}, \{d\}, \{a,c\}, \{a,d\}\}$, $S_{com}(F) = \{\{a\}, \{a,c\}, \{a,d\}\}$, $S_{prf}(F) = \{\{a,c\}, \{a,d\}\}$, $S_{stb}(F) = S_{sst}(F) = \{\{a,d\}\}$ 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 other 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 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, c 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, argument c, which is in just one complete extension, is credulously accepted by the complete semantics.

Many of the semantics mentioned above (such as the admissible and the complete ones) exploit the notion of defence to decide whether an argument is part of an extension. 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* [13]. In the same paper, Caminada also gives a definition for reinstatement labelling.

DEFINITION 2.7 (Reinstatement labelling).
Let $F = \langle Arg, R\rangle$ be an AF and $\mathbb{L} = \{\mathsf{in}, \mathsf{out}, \mathsf{undec}\}$. A labelling of F is a total function $L : Arg \to \mathbb{L}$. We define $in(L) = \{a \in Arg \mid L(a) = \mathsf{in}\}$, $out(L) = \{a \in Arg \mid L(a) = \mathsf{out}\}$ and $undec(L) = \{a \in Arg \mid L(a) = \mathsf{undec}\}$. We say that L is a reinstatement labelling if and only if it satisfies the following:

- $\forall a \in Arg.a \in in(L) \iff \forall b \in Arg \mid (b,a) \in R.b \in out(L)$;
- $\forall a \in Arg.a \in out(L) \iff \exists b \in Arg \mid (b,a) \in R \wedge b \in in(L)$.

In other words, 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. A labelling-based semantics [2] associates a subset of all the possible labellings with an AF. In Figure 2, 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 outand the yellow argument e (that attacks itself) is undec.

Given a labelling L, it is possible to identify a correspondence with the extension-based semantics [2]. In particular, the set of in arguments coincides with a complete extension, while other semantics can be obtained through restrictions on the labelling as shown in Table 1.

The notion of time can be included in the reasoning model of AFs by considering temporal intervals [16] defined as follows.

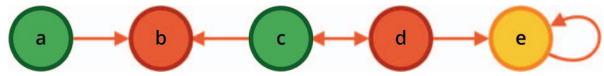

FIGURE 2. Example of reinstatement labelling.

TABLE 1. Reinstatement labelling vs semantics.

Labelling restrictions	Semantics
No restrictions	Complete
Empty undec	Stable
Minimal undec	Semi-stable
Maximal in	Preferred
Maximal out	Preferred
Maximal undec	Grounded
Minimal in	Grounded
Minimal out	Grounded

DEFINITION 2.8 (Temporal interval).
A temporal interval is a pair built from $t_1, t_2 \in \mathbb{Z}$ in one of the following ways:

- $[t_1, t_1]$, denoting the moment t_1;
- $[t_1, +\infty)$, a set of moments formed by all the numbers in $\mathbb{Z}$ greater than or equal to t_1;
- $(-\infty, t_2]$, a set of moments formed by all the numbers in $\mathbb{Z}$ smaller than or equal to t_2;
- $[t_1, t_2]$, a set of moments formed by all the numbers in $\mathbb{Z}$ from t_1 to t_2 (both included);
- $(-\infty, +\infty)$, denoting a set of moments formed by all the numbers in $\mathbb{Z}$.

The set of all the intervals defined over $\mathbb{Z} \cup \{-\infty, +\infty\}$ is denoted Υ.

In TAFs [15, 16], each argument is associated with temporal intervals that express the period of time in which the argument is available.

DEFINITION 2.9 (TAFs).
A TAF is a tuple $T = \langle Arg, R, Av \rangle$ where $Arg \subseteq U$ is a set of adopted arguments belonging to the universe U, R[3] is a binary relation on Arg and $Av : Arg \rightarrow \wp(\Upsilon)$ is the availability function for timed arguments.

In the following, we denote by $i \in I$ ($I \in \wp(\Upsilon)$), possibly subscript, a generic temporal interval (set of temporal intervals). More in detail, considering a temporal interval i and a time instant t, we say that $t \in i$ if one of the following holds:

- $i = [t_1, t_2]$ and $t_1 \leq t \leq t_2$;
- $i = [t_1, +\infty)$ and $t_1 \leq t$;
- $i = (-\infty, t_2]$ and $t \leq t_2$;
- $i = (-\infty, +\infty)$.

[3]Note that in certain time intervals, attacks could be defined between arguments not currently available in the TAF.

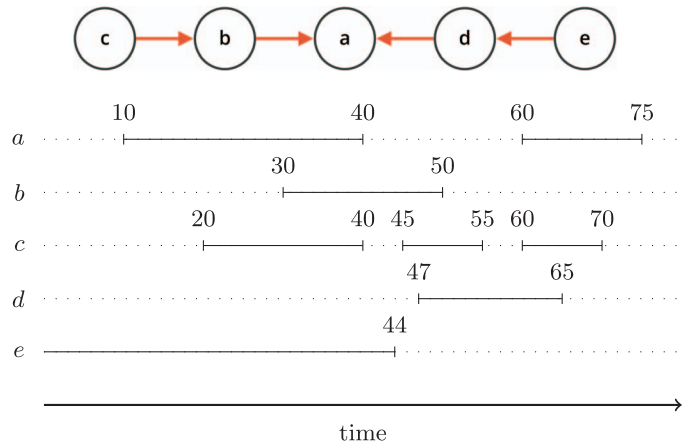

FIGURE 3. Example of a TAF.

Moreover, given $I \in \wp(\Upsilon)$, we say that $t \in I$ if there exists $i \in I$ such that $t \in i$.

We illustrate in Figure 3 an example of the behaviour of timed arguments taken from [16], with availability function defined as $Av(a) = \{[10, 40], [60, 75]\}$, $Av(b) = \{[30, 50]\}$, $Av(c) = \{[20, 40], [45, 55], [60, 70]\}$, $Av(d) = \{[47, 65]\}$ and $Av(e) = \{(-\infty, 44]\}$. Note that attacks in a TAF never change but are only considered when the availability of both the attacker and the attacked arguments overlaps (alternatively, we can consider attacks to blink together with the arguments they connect). We can imagine the modifications at a given instant t taking place simultaneously as if every argument of the TAF was handled (made available/not available) by a different agent. We use the following notation.

DEFINITION 2.10 (Attacks starting from a set of arguments).
Let Arg and R be a set of arguments and a set of attacks, respectively, and let a be an argument. We denote by $R_{|Arg} = \{(a, b) \in R \mid a \in Arg\}$ the set of attacks in R starting from the arguments in Arg, $R_{|a} = R_{|\{a\}}$ and by $\langle Arg, R\rangle_{|a} = \langle Arg', R_{|Arg'}\rangle$, where $Arg' = Arg \cap \{a\}$.

Given a TAF and a time instant t, we define a pair whose components are the set of arguments available at t and the set of attacks starting from it.

DEFINITION 2.11
Let T$= \langle Arg, R, Av\rangle$ be a TAF, and let t a time instant. We denote by $\alpha(T, t)$ the pair $\langle Arg', R_{|Arg'}\rangle$, where $Arg' = \{a \in Arg \mid t \in Av(a)\}$.

By the previous definition, $\alpha(T, t) \downarrow$ is the AF associated with the TAF T, which is available at the instant t. Given a semantics σ, the set of acceptable arguments in a TAF can be computed with respect to a specific moment t by only considering arguments (and attacks) available at t. For example, in the TAF of Figure 3, the singleton $\{b\}$ is an admissible extension for $t \in [41, 44]$, i.e. when b itself is available and its attacker c is not.

TABLE 2. *tcla* syntax.

$$
\begin{aligned}
&P ::= C.A \\
&C ::= p(x) :: A \mid C.C \\
&A ::= success \mid failure \mid add(Arg, R) \rightarrow A \mid rmv(Arg, R) \rightarrow A \mid E \mid A \| A \mid \exists_x A \mid p(x) \\
&E ::= check_t(Arg, R) \rightarrow A \mid c - test_t(a, l, \sigma) \rightarrow A \mid s - test_t(a, l, \sigma) \rightarrow A \mid \\
&\qquad E + E \mid E +_P E \mid E \|_G E
\end{aligned}
$$

3 *tcla* Syntax and Semantics

The syntax of our *tcla* is presented in Table 2, where, as usual, 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. Moreover, $t \in \mathbb{N}$. In Tables 3 and 4, then, we give the definitions for the transition rules.

Communication between tcla agents is implemented via shared memory, similarly to *cla* [8] and CC [32] and opposed to other languages (e.g., CSP [24] and CCS [28]) based on message passing. In the following, we denote by $\mathcal{E}$ the class of guarded agents and by $\mathcal{E}_0$ the class of guarded agents such that all outermost guards have $t = 0$ (note that a Boolean syntactic category could be introduced in replacement of $\mathcal{E}_0$ to better handle guards and allow for finer distinctions). In a *tcla process* $P = C.A$, A is the initial agent to be executed in the context of the set of declarations C.

The operational model of *tcla* processes can be formally described by a transition system $T = (Conf, \longrightarrow)$, where we assume that each transition step takes exactly one time-unit. Configurations (in) *Conf* are pairs consisting of a process and a framework $F = \langle Arg, R \rangle$ representing the common knowledge base, where *Arg* is a set of arguments and R is a set of attacks. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules in Tables 3 and 4, and it characterizes the (temporal) evolution of the system. So, $\langle A, F \rangle \longrightarrow \langle A', F' \rangle$ means that, if at time t, we have the process A and the framework F, then at time $t + 1$ we have the process A' and the framework F'.

In Tables 3 and 4, we have omitted the symmetric rules for the choice operator $+$ and for the two parallel composition operators $\|$ and $\|_G$. Indeed, $+$ is commutative, so $E_1 + E_2$ produces the same result as (i.e. is congruent to) $E_2 + E_1$. The same is true for $\|$ and $\|_G$. Note that $+$ $\|$ and $\|_G$ are also associative. Moreover, *success* and *failure* are the identity and the absorbing elements under the parallel composition $\|$, respectively (namely, for each agent A, we have that $A \| success$ and $A \| failure$ are the agents A and *failure*, respectively). The agents *success* and *failure* represent a successful and a failed termination, respectively, so they may not make any further transition.

Note that in the configurations set *Conf*, the framework $F = \langle Arg, R \rangle$ is not guaranteed to be an AF since R may contain attacks between arguments not in *Arg*.

In the following, we will usually write a *tcla* process $P = C.A$ as the corresponding agent A, omitting C when not required by the context.

Suppose we have an agent A whose knowledge base is represented by a framework $F = \langle Arg, R \rangle$. An $add(Arg', R')$ action 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 attacks R' to the framework F. Note that if an $add(Arg', R')$ action is performed at time t, then the updated store will be visible only from time $t + 1$ onward since an addition takes one time unit to be completed. 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 $R' \neq \emptyset$. Moreover, the structure of the shared store after

TABLE 3. *tcla* operational semantics (part 1/2).

$$\langle add(Arg', R') \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg \cup Arg', R \cup R'\rangle\rangle \quad \text{Ad}$$

$$\langle rmv(Arg', R') \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg \setminus Arg', R \setminus R'\rangle\rangle \quad \text{Re}$$

$$\frac{Arg' \subseteq Arg \wedge R' \subseteq R \quad t > 0}{\langle check_t(Arg', R') \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg, R\rangle\rangle} \quad \text{Ch (1)}$$

$$\frac{Arg' \not\subseteq Arg \vee R' \not\subseteq R \quad t > 0}{\langle check_t(Arg', R') \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle check_{t-1}(Arg', R') \to A, \langle Arg, R\rangle\rangle} \quad \text{Ch (2)}$$

$$\langle check_0(Arg', R') \to A, F\rangle \longrightarrow \langle failure, F\rangle \quad \text{Ch (3)}$$

$$\frac{\exists L \in S_\sigma(\langle Arg, R\rangle \downarrow) \mid l = L(a) \quad t > 0}{\langle c - test_t(a, l, \sigma) \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg, R\rangle\rangle} \quad \text{CT (1)}$$

$$\frac{\forall L \in S_\sigma(\langle Arg, R\rangle \downarrow).l \neq L(a) \quad t > 0}{\langle c - test_t(a, l, \sigma) \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle c - test_{t-1}(a, l, \sigma) \to A, \langle Arg, R\rangle\rangle} \quad \text{CT (2)}$$

$$\langle c - test_0(a, l, \sigma) \to A, F\rangle \longrightarrow \langle failure, F\rangle \quad \text{CT (3)}$$

$$\frac{\forall L \in S_\sigma(\langle Arg, R\rangle \downarrow).l = L(a) \quad t > 0}{\langle s - test_t(a, l, \sigma) \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle A, \langle Arg, R\rangle\rangle} \quad \text{ST (1)}$$

$$\frac{\exists L \in S_\sigma(\langle Arg, R\rangle \downarrow) \mid l \neq L(a) \quad t > 0}{\langle s - test_t(a, l, \sigma) \to A, \langle Arg, R\rangle\rangle \longrightarrow \langle s - test_{t-1}(a, l, \sigma) \to A, \langle Arg, R\rangle\rangle} \quad \text{ST (2)}$$

$$\langle s - test_0(a, l, \sigma) \longrightarrow A, F\rangle \longrightarrow \langle failure, F\rangle \quad \text{ST (3)}$$

$$\frac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle, \quad E_1 \notin \mathcal{E}_0, \quad A_1 \notin \mathcal{E}}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle A_1, F\rangle} \quad \text{If (1)}$$

$$\frac{\langle E_1, F\rangle \longrightarrow \langle E_1', F\rangle, \quad E_1 \notin \mathcal{E}_0, \quad E_1' \in \mathcal{E}}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle E_1' +_P E_2, F\rangle} \qquad \frac{E_1 \in \mathcal{E}_0, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 +_P E_2, F\rangle \longrightarrow \langle A_2, F\rangle} \quad \text{If (2)}$$

an *add* operation is not guaranteed to be an AF since attacks involving arguments in $U \setminus \{Arg \cup Args'\}$ can be added to R.

Intuitively, $rmv(Arg', R')$ allows specifying arguments and attacks to be removed from the knowledge base. The store is again not guaranteed to be an AF: after the removal of an argument, possible attacks outgoing from and incoming to that argument are still maintained. Removing an argument (or an attack) that does not exist in F will have no consequences.

The $c - test_t(a, l, \sigma) \to A$, $s - test_t(a, l, \sigma) \to A$ and $check_t(Arg, R) \to A$ constructs are explicit timing primitives that allow for the specification of timeouts. In fact, in timed applications often one cannot wait indefinitely for an event. Thus, a timed language should allow us to specify that in case a given time bound is exceeded (i.e. a *timeout* occurs), the wait is interrupted and an alternative action is taken.

TABLE 4. *tcla* operational semantics (part 2/2).

$$\frac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle,\ \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle,\ \ E_1, E_2 \notin \mathcal{E}_0,\ \ A_1 \notin \mathcal{E}}{\langle E_1 \|_G E_2, F\rangle \longrightarrow \langle A_1 \| A_2, F\rangle} \quad \text{GP (1)}$$

$$\frac{\langle E_1, F\rangle \longrightarrow \langle E_1', F\rangle,\ \langle E_2, F\rangle \longrightarrow \langle E_2', F\rangle,\ \ E_1, E_2 \notin \mathcal{E}_0,\ \ E_1', E_2' \in \mathcal{E}}{\langle E_1 \|_G E_2, F\rangle \longrightarrow \langle E_1' \|_G E_2', F\rangle} \quad \text{GP (2)}$$

$$\frac{E_1 \in \mathcal{E}_0, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 \|_G E_2, F\rangle \longrightarrow \langle A_2, F\rangle} \quad \text{GP (3)}$$

$$\frac{\langle A_1, F\rangle \longrightarrow \langle A_1', F'\rangle,\ \ \langle A_2, F\rangle \longrightarrow \langle A_2', F''\rangle}{\langle A_1 \| A_2, F\rangle \longrightarrow \langle A_1' \| A_2', *(F, F', F'')\rangle} \quad \text{Pa}$$

$$\frac{\langle E_1, F\rangle \longrightarrow \langle A_1, F\rangle,\ \ E_1 \notin \mathcal{E}_0,\ \ A_1 \notin \mathcal{E}}{\langle E_1 + E_2, F\rangle \longrightarrow \langle A_1, F\rangle} \qquad \frac{E_1 \in \mathcal{E}_0, \langle E_2, F\rangle \longrightarrow \langle A_2, F\rangle}{\langle E_1 + E_2, F\rangle \longrightarrow \langle A_2, F\rangle} \quad \text{ND (1)}$$

$$\frac{\langle E_1, F\rangle \longrightarrow \langle E_1', F\rangle,\ \langle E_2, F\rangle \longrightarrow \langle E_2', F\rangle,\ \ E_1, E_2 \notin \mathcal{E}_0,\ \ E_1', E_2' \in \mathcal{E}}{\langle E_1 + E_2, F\rangle \longrightarrow \langle E_1' + E_2', F\rangle} \quad \text{ND (2)}$$

$$\frac{\langle A[y/x], \langle Arg, R\rangle\rangle \longrightarrow \langle A', F'\rangle}{\langle \exists_x A, \langle Arg, R\rangle\rangle \longrightarrow \langle A', F'\rangle} \text{ with } y \in U \setminus Arg \quad \text{HA}$$

$$\langle p(y), F\rangle \longrightarrow \langle A[y/x], F\rangle \text{ with } p(x) :: A \text{ and } x \in \{a, l, \sigma, t\} \quad \text{PC}$$

The operator $check_t(Arg', R')$ (rules Ch (1)–(3) in Table 3) realizes a timed construct and is used to verify whether, in a given time interval, the specified arguments and attacks are contained in the set of arguments and attacks of the knowledge base, without introducing any further change. If $t > 0$ and the check is positive, the guard succeeds and the agent $check_t(Arg', R') \rightarrow A$ behaves like A at the next time instant (rule Ch (1)). If $t > 0$ but the check is not satisfied, then the control is repeated at the next time instant, and the value of the counter t is decreased (rule Ch (2)). Axiom Ch (3) shows that if the timeout is exceeded, i.e. the counter t has reached the value of 0, then the process $check_t(Arg', R') \rightarrow A$ fails.

The rules for credulous tests CT (1)–(3) and sceptical tests ST (1)–(3) in Table 3 are similar to rules Ch (1)–(3) described before. Observe that we have two distinct test operations, both requiring the specification of an argument $a \in Arg$, a label $l \in \{\mathsf{in}, \mathsf{out}, \mathsf{undec}\}$[4] and a semantics $\sigma \in \{adm, com, stb, sst, prf, gde\}$. We know that $\langle Arg, R\rangle$ is not guaranteed to be an AF since R may contain attacks between arguments not in Arg. Therefore, to execute a test operation, we project beforehand the set of attacks to couples of arguments in Arg. Hence, $c - test_t(a, l, \sigma)$ succeeds if there exists at least one extension of $S_\sigma(\langle Arg, R\rangle \downarrow)$ whose corresponding labelling L is such that $L(a) = l$. Similarly, $s - test_t(a, l, \sigma)$ succeeds if a is labelled l in all possible labellings $L \in S_\sigma(\langle Arg, R\rangle \downarrow)$.

The operator $+_P$ (if (1)–(2) in Table 3) is left-associative and realizes an if-then-else construct: if we have $E_1 +_P E_2$ (with $E_1, E_2 \in \mathcal{E}$) and the guards of E_1 succeed, then E_1 will be always chosen

[4]Other labelling semantics could be considered where undec arguments are further divided into 'don't know' and 'don't care' [10].

over E_2, even if also the guards of E_2 succeed. So, for E_2 to be selected, it has to be the only one such that its guards succeed and will be selected only after E_1 fails. If the guards of E_1 do not fail, the execution can either move to any consequent agent A_1 which does not belong to $\mathcal{E}$ or proceed with $E_1' +_P E_2 \in \mathcal{E}$.

The guarded parallelism GP (1)–(3) in Table 4 is designed to allow all the operations for which the guards in the inner expression are satisfied. In more detail, the guards of $E_1 \|_G E_2$ succeed when either the guards of E_1, E_2, or both succeed, and all the operations that can be executed are executed. This behaviour differs from classical parallelism (for which all the agents have to succeed in order for the parallel agent to succeed) and from nondeterminism (that only selects one branch).

The remaining operators are classical in concurrency. The rule parallelism Pa in Table 4 models the parallel composition operator in terms of *maximal parallelism*. By transition rules, an agent in a parallel composition obtained through $\|$ succeeds only if all the agents succeed.

In general, the parallelism operator denotes the distributed execution of processes; therefore, it is not possible to have a global view of the entire parallel composition. Each agent only has access to the shared store and not the processes running in the other agents.

In our implementation of rule Pa, we use $*(F, F', F'') := (F' \cap F'') \cup ((F' \cup F'') \setminus F)$ to handle parallel additions and removals of arguments[5] . In particular, if an argument a is added and removed at the same moment (e.g. through the process $add(\{a\}, \{\}) \rightarrow success \parallel rmv(\{a\}, \{\}) \rightarrow success$), we have two possible outcomes: if a was not present in the knowledge base, then the *add* operation gains priority over the *rmv* one, since $a \in ((F' \cup F'') \setminus F)$; on the other hand, when a was already in the shared memory, we have that $a \notin ((F' \cup F'') \setminus F)$ and a is removed.

Any agent composed through $+$ (rules ND (1)-(2)) is chosen if its guards succeed.

The existential agent $\exists_x A$ in rule HA in Table 4 implements a notion of locality: the agent $\exists_x A$ behaves like A, with the argument x considered as local to it, thus hiding the information on x provided by the external environment. This is obtained by substituting argument x for argument y, which we assume to be new, not in the framework $\langle Arg, R \rangle$ and not used by any other process. Standard renaming techniques can be used to ensure this. Only the agent executing $\exists_x$ can read the hidden variable x. This operator can be used, for instance, to introduce an argument a into the knowledge base without the other agents knowing about it. If a attacks some other argument, say b, processes in which a is not local can still 'see' that b is defeated just by executing a credulous/sceptical test, but they can neither directly verify the existence of a nor attack it back. Finally, the procedure call (rule PC) has a single parameter: an argument, a label among in, out and undec, a semantics σ or an instant of time. The procedure call can be extended to allow more than one parameter if necessary.

In rules HA and PC, $A[x/y]$ denotes the process obtained from A by replacing variable x for variable y.

Using the transition system described in (the rules in) Tables 3 and 4, we can now define our notion of observables, which considers the traces, whose components are AFs, of successful or failed terminating computations that the agent A can perform for each *tcla* process $P = C.A$.

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

DEFINITION 3.1 (Observables for *tcla*).
Let $P = C.A$ be a *tcla* process. We define

$$\begin{aligned}\mathcal{O}_{io}(P) = \quad & \{(F_1 \downarrow) \cdots (F_n \downarrow) \cdot ss \mid \langle A, F_1\rangle \longrightarrow^* \langle success, F_n\rangle\} \cup \\ & \{(F_1 \downarrow) \cdots (F_n \downarrow) \cdot ff \mid \langle A, F_1\rangle \longrightarrow^* \langle failure, F_n\rangle\},\end{aligned}$$

where $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

4 Modelling TAFs

In the context of argumentation, the possibility to include time in the reasoning processes conducted by intelligent agents allows for modelling dynamic behaviours, such as, for instance, the addition and the removal of information at precise moments from a knowledge base. Agents using *tcla* constructs and availability functions in TAFs regulate the existence of arguments in a similar way; in particular, we can imagine each agent having control over an argument of the TAF. This result is shown by providing an explicit translation from a TAF into a *tcla* process.

In the following, given $I \in \wp(\Upsilon)$, I is ordered if there is no $[t_1, t_2] \in I$ such that $t_2 < t_1$ and for each $i_1, i_2 \in I$, we have that i_1 and i_2 are disjoint and non-consecutive. Moreover, I is positive if for each $i \in I$, we have that $i = (-\infty, +\infty)$ or $i = [t, +\infty)$ or $i = [t, t']$ or $i = (-\infty, t']$ with $t' \geq 0$. Without loss of generality, since we want define the translation of a TAF for positive time instants, we consider only positive ordered sets of intervals. Moreover, given $I \neq \emptyset$, we denote by

$$first(I) = \begin{cases} (-\infty, +\infty) & \text{if } (-\infty, +\infty) \in I \\ (-\infty, t] & \text{if } (-\infty, t] \in I \\ [t, +\infty) & \text{if } I = \{[t, +\infty)\} \\ [t_1, t_2] & \text{otherwise,} \end{cases}$$

where $[t_1, t_2] \in I$ is such that for each $i \in I \setminus [t_1, t_2]$, we have that $i = [t'_1, t'_2]$ or $i = [t'_1, +\infty)$, with $t_2 < t'_1$. Finally, given an ordered set of intervals $I \neq \emptyset$, we define $continue(I) = I \setminus first(I)$.

To ease the translation, we use $sleep(t) \rightarrow A$ as a shortcut for

$$\begin{cases} A & \text{if } t \leq 0 \\ check_1(\{\}, \{\}) \rightarrow (sleep(t-1) \rightarrow A) & \text{otherwise.} \end{cases}$$

The agent $sleep(t) \rightarrow A$, with $t > 0$, executes the check operation for exactly t times and then the agent A is executed. Practically, $sleep(t)$ lets the agent A wait for t instants of time. When $t \leq 0$, the agent A is immediately executed.

Finally, we use $add(Arg, R)$ and $rmv(Arg, R)$ as shortcuts for $add(Arg, R) \rightarrow success$ and $rmv(Arg, R) \rightarrow success$, respectively.

DEFINITION 4.1 (Translation).
The translation of a TAF $\langle \{a_1, \ldots, a_n\}, R, Av\rangle$ is

$$\mathcal{T}(\langle \{a_1, \ldots, a_n\}, R, Av\rangle) = \mathcal{T}_{add}(a_1, R_{|a_1}, Av(a_1)) \| \cdots \| \mathcal{T}_{add}(a_n, R_{|a_n}, Av(a_n)),$$

where

$$\mathcal{T}_{add}(a,R,S) = \begin{cases} success & \text{if } S = \emptyset \text{ or} \\ & S = \{(-\infty,+\infty)\} \text{ or} \\ & S = \{[t,+\infty)\} \text{ with } t \leq 0 \\ \mathcal{T}_{rmv}(a,R,S) & \text{if there is } (-\infty,t] \in S \\ (sleep(t_{in}-1) \rightarrow add(\{a\},R)) \parallel & \\ \mathcal{T}_{rmv}(a,R,S)) & \text{otherwise} \end{cases}$$

and

$$\mathcal{T}_{rmv}(a,R,S) = \begin{cases} success & \text{if } S = \{[t,+\infty)\} \\ (sleep(t_{fin}) \rightarrow rmv(\{a\},R)) \parallel & \\ \mathcal{T}_{add}(a,R,continue(S))) & \text{otherwise,} \end{cases}$$

where we assume that $first(S) = [t_{in}, t_{fin}]$ or $first(S) = (-\infty, t_{fin}]$ if $S \neq \emptyset$, $S \neq \{(-\infty,+\infty)\}$ and $S \neq \{[t,+\infty)\}$ with $t \leq 0$.

EXAMPLE 4.2

To illustrate how the translation given above can be used to model a TAF, we consider a scenario presented in [11], where an agent (a person called Anna) is looking for an apartment to rent. In this example, we use the following arguments to model Anna's reasoning about whether the chosen apartment will be a good option for the next 12 months. The deriving TAF is depicted in Figure 4.

a: Anna should rent the apartment she found
b: the apartment seems to have humidity problems
c: the owner is committed to solving structural problems in the apartment
d: a nearby nightclub will open in May
e: laws valid until August forbid the opening of nightclubs in the area

By translation $\mathcal{T}$, the availability of timed arguments for each time instant $t \geq 0$ can be simulated by a *tcla* process that modifies the store by adding and removing part of the underlying framework. The TAF considered in this example can be modelled through a translation $\mathcal{T}(\langle Arg, R, Av\rangle)$, with $Arg = \{a,b,c,d,e\}$, $R = \{(b,a),(c,b),(d,a),(e,d)\}$ and the availability function Av defined as shown in Figure 4. In this example, the translation $\mathcal{T}$ executes five parallel $\mathcal{T}_{Add}$, one for each argument in Arg. The part of the translation in charge of handling argument a in our example is

$$\mathcal{T}_{Add}(a,\{\},\{[1,12]\}) = sleep(0) \rightarrow add(\{a\},\{\}) \rightarrow success \parallel sleep(12) \rightarrow rmv(\{a\},\{\}) \rightarrow success.$$

For argument d we have, instead,

$$\mathcal{T}_{Add}(d,\{(d,a)\},\{[5,12]\}) = sleep(4) \rightarrow add(\{d\},\{(d,a)\}) \rightarrow success \parallel sleep(12) \rightarrow rmv(\{d\},\{(d,a)\}) \rightarrow success.$$

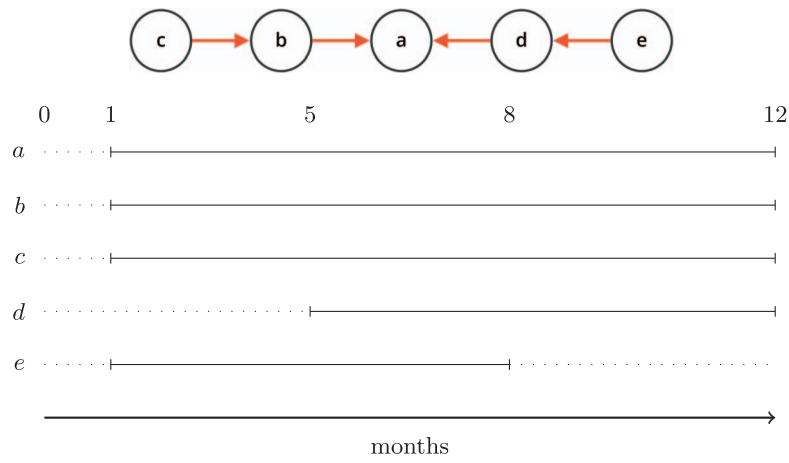

FIGURE 4. Example of a TAF. Time intervals are specified in months (we match 1 in January, 2 in February, 5 in May and so on).

In practice, to emulate the behaviour of argument d, starting from the time instant $t = 0$, the process sleeps for 4 instants of time (i.e. it executes four consecutive dummy $check_1(\{\}, \{\})$ operations), until $add(\{d\}, \{(d, a)\})$ is executed, so that at $t = 5$ the store will contain the argument d and the attack from d towards a. At the same time, in a parallel branch, the process sleeps for 12 instants of time and then executes $rmv(\{d\}, \{\})$. This way, $t = 12$ is the last instant in which d is available in the store. Note that the translation also considers open intervals with lower limit $-\infty$ and upper limit $+\infty$.

THEOREM 4.3
The translation $\mathcal{T}$ of a TAF $T = \langle Arg, R, Av\rangle$ produces a *tcla* process P such that, starting from the store $F_0 = \alpha(T, 0)$, at any instant of time $t \geq 0$, the AF associated to the computed store by P is the AF available at t in the original TAF T.

PROOF. Let us consider the TAF $T = \langle Arg, R, Av\rangle$, and let $\mathcal{T}$ its translation. We prove that $F_0F_1 \cdots F_n \cdot ss \in \mathcal{O}_{io}(\mathcal{T})$, where $F_0 = \alpha(T, 0)$ if and only if for each $\tilde{t} \geq 0$,

$$\alpha(T, \tilde{t}) \downarrow = \begin{cases} F_{\tilde{t}} & \text{if } \tilde{t} \leq n \\ F_n & \text{otherwise.} \end{cases}$$

We prove that for each $a \in Arg$,

$$\langle \mathcal{T}_{add}(a, R_{|a}, Av(a)), \alpha(T, 0)_{|a}\rangle \longrightarrow^* \langle success, F'_m\rangle$$

if and only if $m <= n$ and for each $\tilde{t} \geq 0$

$$\alpha(T, \tilde{t})_{|a} = \alpha(\langle\{a\}, R_{|a}, Av(a)\rangle, \tilde{t}) = \begin{cases} F'_{\tilde{t}} & \text{if } \tilde{t} \leq m \\ F'_m & \text{otherwise} \end{cases}$$

Therefore, the proof is straightforward by definition of $\|$.

The proof is by induction on the cardinality l of $Av(a)$. In the following, let $\tilde{F}_0 = \langle\emptyset, \emptyset\rangle$, and let $\tilde{F}_1 = \langle\{a\}, R_{|a}\rangle$.

- $l = 0$) In this case, $Av(a) = \emptyset$ and $\mathcal{T}_{add}(a, R_{|a}, Av(a)) = success$. By definition for each $\tilde{t} \geq 0$ $\alpha(T, \tilde{t})_{|a} = \tilde{F}_0$, $m = 0$, $F'_0 = \alpha(T, 0)_{|a} = \tilde{F}_0$ and then the thesis.
- $l > 0$) We have the following possibilities.
  - e: $Av(a) = \{(-\infty, +\infty)\}$ or $Av(a) = \{[t, +\infty)\}$, with $t \leq 0$.) In this case, $\mathcal{T}_{add}(a, R_{|a}, Av(a)) = success$. By definition for each $\tilde{t} \geq 0$ $\alpha(T, \tilde{t})_{|a} = \tilde{F}_1$, $m = 0$, $F'_0 = \alpha(T, 0)_{|a} = \tilde{F}_1$ and then the thesis.
  - e: $Av(a) = \{[t, +\infty)\}$, with $t > 0$. By definition for each $\tilde{t} \geq 0$,

$$\alpha(T, \tilde{t})_{|a} = \begin{cases} \tilde{F}_0 & \text{if } 0 \leq \tilde{t} < t \\ \tilde{F}_1 & \text{otherwise} \end{cases}$$

    and

$$\mathcal{T}_{add}(a, R_{|a}, Av(a)) = (sleep(t-1) \rightarrow add(\{a\}, R_{|a})) \| success$$

    By definition, $sleep(t-1) \rightarrow add(\{a\}, R_{|a})$ tries to execute the operation check $t-1$ times starting from the time instant 0 and therefore at the instant $t-1$ the action $add(\{a\}, R_{|a})$ is executed. By previous observations, for each $0 \leq \tilde{t} < t$, $F'_{\tilde{t}} = \tilde{F}_0$ and for each $\tilde{t} \geq t$, $F'_{\tilde{t}} = \tilde{F}_1$ and then the thesis.
  - e: $i = (-\infty, t] = first(Av(a))$ In this case,

$$\alpha(T, \tilde{t})_{|a} = \begin{cases} \tilde{F}_1 & \text{if } 0 \leq \tilde{t} \leq t \\ \alpha(\langle \{a\}, R_{|a}, Av(a) \setminus i \rangle, \tilde{t}) & \text{otherwise} \end{cases}$$

    and

$$\mathcal{T}_{add}(a, R_{|a}, Av(a)) = (sleep(t) \rightarrow rmv(\{a\}, R_{|a})) \parallel \mathcal{T}_{add}(a, R_{|a}, Av(a) \setminus i).$$

    By definition, $sleep(t) \rightarrow rmv(\{a\}, R_{|a})$ tries to execute the operation check t times, starting from the time instant 0, and therefore, analogously to the previous case, at the instant t the action $rmv(\{a\}, R_{|a})$ is executed. Therefore,

$$\langle sleep(t) \rightarrow rmv(\{a\}, R_{|a}), F'_0 \rangle \longrightarrow^* \langle rmv(\{a\}, R_{|a}), F'_t \rangle \longrightarrow \langle success, \tilde{F}_0 \rangle,$$

    where for each $0 \leq \tilde{t} \leq t$, $F'_{\tilde{t}} = \tilde{F}_1$.
  - e: Moreover, let $Av'(a) = Av(a) \setminus i$. By inductive hypothesis and by definition of $\|$,

$$\langle \mathcal{T}_{add}(a, R_{|a}, Av'(a)), \alpha(T, 0)_{|a} \rangle \longrightarrow^* \langle success, F''_p \rangle$$

    if and only if $p <= m$ and for each $\tilde{t} \geq 0$

$$\alpha(\langle \{a\}, R_{|a}, Av'(a) \rangle, \tilde{t}) = \begin{cases} F''_{\tilde{t}} & \text{if } \tilde{t} \leq p \\ F''_p & \text{otherwise.} \end{cases}$$

    By definition of $Av'(a)$, for each $[t', t''] \in Av'(a)$ or $[t', +\infty) \in Av'(a)$, we have that $t' > t + 1$. Therefore, for each $0 \leq \tilde{t} \leq t + 1$, $F''_{\tilde{t}} = \tilde{F}_0$ and for each $\tilde{t} > t + 1$, $F''_{\tilde{t}} = F'_{\tilde{t}}$. Now, the proof follows by definition of $Av'(a)$ and $\|$.

TABLE 5. Implementation of *tcla* operations.

tcla syntax	Implementation
$add(Arg, R)$	add(Arg,R)
$rmv(Arg, R)$	rmv(Arg,R)
$check_t(Arg, R)$	check(t,Arg,R)
$c - test_t(a, l, \sigma)$	ctest(t,a,l,σ)
$s - test_t(a, l, \sigma)$	stest(t,a,l,σ)
$E + \cdots + E$	sum(E,...,E)
$E \|\|_G \ldots \|\|_G E$	gpar(E,...,E)
$E +_P E$	(E)+P(E)

e: $i = [t, t'] = first(Av(a))$ with $t' \geq 0$) In this case,

$$\alpha(T, \tilde{t})_{|a} = \begin{cases} \tilde{F}_0 & \text{if } 0 \leq \tilde{t} < t \\ \tilde{F}_1 & \text{if } t \leq \tilde{t} \leq t' \\ \alpha(\langle \{a\}, R_{|a}, Av(a) \setminus i \rangle, \tilde{t}) & \text{otherwise} \end{cases}$$

and

$$\begin{aligned} \mathcal{T}_{add}(a, R_{|a}, Av(a)) = \quad & (sleep(t-1) \rightarrow add(\{a\}, R_{|a})) \parallel \\ & (sleep(t') \rightarrow rmv(\{a\}, R_{|a})) \parallel \\ & \mathcal{T}_{add}(a, R_{|a}, Av(a) \setminus i). \end{aligned}$$

Now, the proof is analogous to the previous case.

□

COROLLARY 4.4
Given a *tcla* process produced by a translation $\mathcal{T}$ of a TAF, the set of arguments in the store accepted with respect to a semantics σ at a given instant of time $t \geq 0$ (starting from the initial store $F_0 = \alpha(T, 0)$) coincides with the set of extensions identified by σ at moment t on the original TAF.

5 *tcla* Simulator

We developed a working implementation for *tcla*. Some of the operations had their syntax translated (see Table 5) to enable users to specify *tcla* programs manually.

As shown in Figure 5, the main components of our tool are (i) the ConArg solver, (ii) an engine written in Python that works as an interpreter and (iii) a web interface (see Figure 7) exposing the functionalities of the language. The web interface communicates with the Python engine, which in turn serves a dual purpose: first, it reads the input programme and forwards it to ConArg for solving the argumentation problems posed by the test operations of itshape tcla; then, once the result of the execution has been processed, the Python engine returns it to the web interface where it is displayed for the user's convenience. Below, we describe each component, focusing on the implementation choices made during the development of the tool.

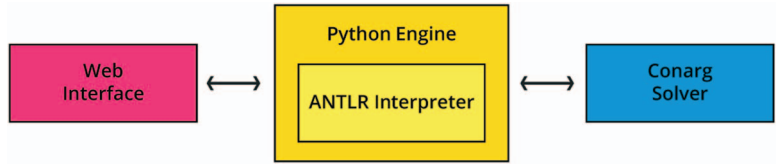

FIGURE 5. Components of the *tcla* architecture.

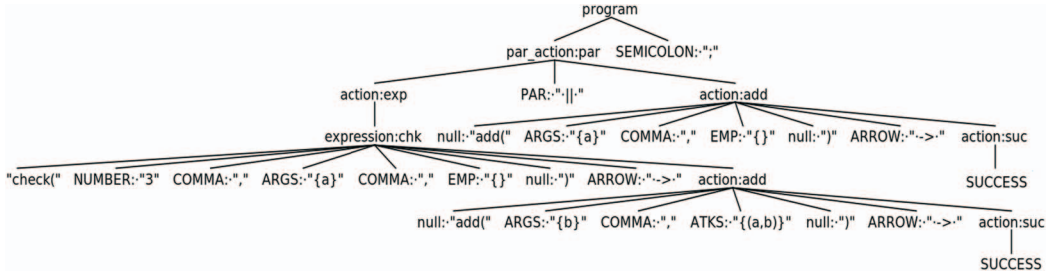

FIGURE 7. Example of a *tcla* program executed form the web interface.

5.1 ConArg suite

Based on constraint programming, ConArg[6] consists of a suite of tools aimed at solving argumentation-related problems, like computing the set of extensions of a given AFs or testing the acceptability of a particular argument. In the *tcla* implementation, we use ConArg to realize credulous and sceptical test operations. For example, the execution of `ctest(t,{a},in,com)` generates a call to the ConArg solver to determine whether argument a is credulously accepted with respect to the complete semantics. Note that the reference AFs used to solve the problem are all those within the subsequent t time units. Therefore, the AF that is passed to ConArg is the one that constitutes the shared memory at the instant the test operation is executed. The execution of a sceptical test involves a similar procedure. To establish whether a given argument a is sceptically accepted with respect to a semantics σ, ConArg verifies if every extension of σ contains a. Then, the computational complexity of a sceptical test is the same as computing all extensions of σ. Concerning the credulous test, ConArg looks for only one extension of σ containing a; although, in the worst case, all extensions must still be calculated, making the computational complexity of a credulous test equal to that of a sceptical test. Note that, although we decided to use ConArg as the argumentation solver for the implementation discussed here, the tool's architecture allows complete generalization of the various components and thus the adoption of other solvers. ConArg is called from the Python script described in the following.

5.2 The Python engine

The core of our implementation consists of a Python script that covers three fundamental tasks: it serves as an interpreter for the *tcla* syntax, it executes programs taken in input, and it communicates

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

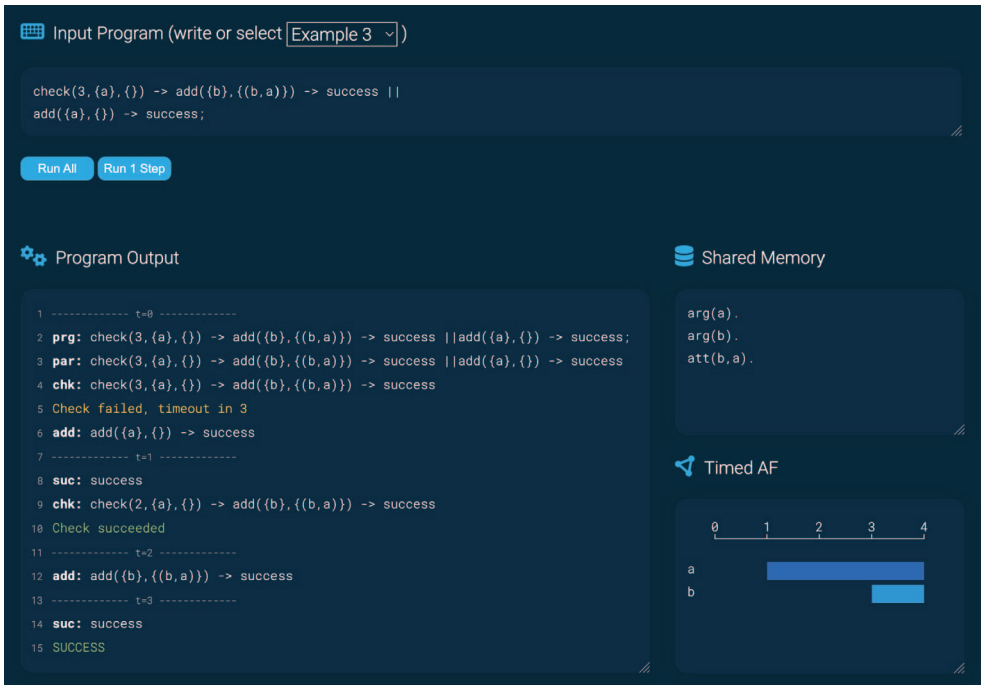

FIGURE 6. Example of parse tree for a *tcla* program.

with the web interface. The interpreter is built using ANTLR[7] , a parser generator for reading, processing, executing and translating structured text. We start with a grammar file defining the constructs in Table 2. Any source program, then, is parsed according to the grammar, and a parse tree is generated. Nodes of this three correspond to syntactic elements of the grammar whose behaviour is defined in a dedicated Python class. Visiting the parse three is equivalent to executing the corresponding program. Figure 6 shows an example of a parse tree for the program below.

```
check(3,{a},{}) -> add({b},{(b,a)}) -> success ||
add({a},{}) -> success;
```

The parser. The root of a parse tree for *tcla* programs is always a *visitPrg* node, which calls the visit on its children, collects the results and returns the final output. *visitSuc* and *visitFlr* are leafs corresponding to terminal nodes of `success` and `failure` agents, respectively. *visitAdd* and *visitRmv* implement `add(Arg,R)` and `rmv(Arg,R)` operations, respectively. They modify the knowledge base by adding/removing part of the framework, always succeeding and continuing on their children. Note that, for adding an attack (a, b), arguments a and b must be contained in the shared memory when *visitAdd* is performed. The two arguments can be introduced in the same step in which the attack between them is added. *visitRmv* also succeeds if the specified arguments and attacks are not in the AF (in that case, the AF is left unchanged). Suppose the same argument (or attack) is added and removed during the same step through a *visitAdd* and a

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

visitRmv executed in parallel. In that case, the resulting shared memory state is computed by the $*$ operator. *visitChk* checks if a given set of arguments and attacks belongs to the knowledge base as per the `check(t,Arg,R)` operator. In case of success, the visit proceeds to the consequent action. On the other hand, when the knowledge base does not contain the specified AF parts, the timeout decreases, and the check is repeated. When the timeout reaches zero, *visitChk* fails. *visitTcr* (`ctest(t,{a},l,`σ`)`) and *visitTsk* (`stest(t,{a},l,`σ`)`) call the ConArg solver to credulously/sceptically test the acceptability of a given argument a, with respect to a semantics σ. The functions repeat the verification until either the test succeeds or the timeout reaches zero. In the latter case, both constructs return failure. A node *visitNdt* implements `sum(E,...,E)`, that is, a concatenation of $+$ operators, inspecting the guards of all its children expressions and randomly selecting one branch to execute among the possible ones. If no guards are found with satisfiable conditions, all timeouts are decreased and the conditions are rechecked in the next step. Expressions with expired timeouts are discarded, and if no expression can be executed before the last timeout expires, the construct fails. *visitIte* behaves like the if-then-else construct `(E)+P(E)`. The expressions are handled in the same order in which they are specified. If the first succeeds, *visitIte* succeeds without executing the second one. If the first expression fails, then the second one is executed. If also the second expression fails, the construct fails. Parallel agents are implemented through the use of threads. A node *visitPar* starts separated threads to execute parallel agents composed through $\|$, returning true if both succeed or false as soon as one action fails. *visitGpa* implements `gpar(E,...,E)`, namely a concatenation of $\|_G$ operators. It cycles through its children, starting a new thread for every expression found with a satisfiable guard. If all the executed expressions succeed, the construct succeeds. If no expression can be executed, all timeouts are decreased and expressions with expired timeouts are discarded. The process is repeated until all the expressions have been executed or discarded for a timeout or when an expression terminates with failure. Procedure calls are handled by *visitPcl* nodes. Beforehand, a procedure must be defined, at the beginning of the program, with the construct `def p(`x_1`,...,`x_n`): A`, where p is the name of the procedure, $x_1, \ldots, x_n$ a set of optional parameters and A the body of the procedure containing the action to execute. Variables in A are specified by the name of a parameter preceded by the $ symbol. When the procedure is called inside the program, all variables in the body are instantiated with the values given in the call, and agent A is executed. Consider, for instance, the following program.

```
def my_proc(x,y): check($y,{$x},{}) -> add({a},{}) -> success;
my_proc(b,4) -> success || add({b},{}) -> success;
```

It begins with defining the *my_proc* procedure, which takes x and y as parameters. Subsequently, when *my_proc*$(b, 4)$ is called, the variables in its body are instantiated, and *check*$(4, \{b\}, \{\}) \rightarrow$ *add*$(\{a\}, \{\}) \rightarrow$ *success* is executed.

Maximal parallelism. To obtain maximal parallelism for the operations, we synchronize the threads that implement parallel agents by keeping track of time elapsing in each parallel branch of the execution. We use a global counter, shared by all threads, to simulate clock cycles of a processor. The counter is defined as a shared variable via the Python *multiprocessing* package and thus can be safely managed by parallel threads without the risk of generating inconsistent data. At each step (which corresponds to a clock cycle), one operation of each parallel agent is executed. The last thread to execute an operation in the current step is in charge of increasing the clock counter, marking the beginning of the subsequent cycle. Failed attempts of execution also consume a unit of time: when the condition of a guarded agent is not satisfied, its timeout is decreased and the execution is postponed by one step. To obtain such a behaviour, we rewrite the program's parse tree so that each

node representing a non-satisfiable guarded agent A is assigned a single child node that is a clone of A, except for the timeout, which is decreased by one.

5.3 tcla web interface

The input program is provided to the Python script through a web interface (shown in Figure 7) developed in HTML and JavaScript, which can be tested at the following link: https://conarg.dmi.unipg.it/tcla-m. After the program has been executed, its output is also shown in the interface.

We have two main areas, one for the input and the other for the output. First, the user enters a program in the dedicated text box (either manually or by selecting one of the provided examples), after which there are two ways to proceed: by clicking on the 'Run All' button, the whole program is executed at once, and the final result is displayed in the output area; alternatively, by clicking on the 'Run 1 Step' button, it is possible to monitor the execution step by step. The interface communicates with the underlying Python engine through an Ajax call which passes the program as a parameter and asynchronously retrieves the output. After the execution of (a step of) the program, three different components are simultaneously visualized in the output area: the program output, the state of the shared memory and a timeline representing the availability of arguments over time. The program output box shows the results of the execution, divided by steps. The beginning of each step is marked by a separating line explicitly showing the step number (i.e. the elapsed time in terms of clock cycles). The shared memory box is updated after each execution step and shows the AF used as the knowledge base. Finally, the bottom-right box contains the visual representation of timed arguments (which resembles the illustration of a TAF provided in Figure 3) and shows the temporal evolution of the AF used by the *tcla* program. Time is reported on the x axis, and each bar of the timeline shows the intervals of time during which an argument is contained in the shared memory, miming the availability functions of arguments in a TAF.

5.4 Execution examples

We now illustrate two examples showing the execution of *tcla* programs obtained as translations of TAFs. Each of the used TAFs represents, in its turn, a dialogue between different counterparts that disagree on a certain choice.

EXAMPLE 5.1
Consider the TAF whose arguments and attacks are represented in Figure 8 and with availability of timed arguments $Av(a) = Av(b) = \{(-\infty, +\infty)\}$, $Av(c) = \{(-\infty, 3]\}$ and $Av(d) = \{[2, +\infty)\}$. In this case, time instants represent days, with $t = 0$ meaning 'today'. Such a TAF can be seen as an abstract formalization of the dialogue between Mark and Lisa, who want to decide what to do during the Christmas holidays. Four arguments are presented in the discussion, as listed below, two of which have limited time availability.

- a : Mark wants to watch a movie at the cinema
- b : Lisa wants to take advantage of the sales and go shopping
- c : sales have yet to start and will not be available for the next three days
- d : the movie Mark wants to watch will be dropped from the schedule in two days

We use the translation of Definition 4.1 to generate a *tcla* program corresponding to the considered TAF. Arguments a, b and c are available since the beginning; hence, they will be immediately added in parallel into the shared memory. For conciseness and consistency, our translation function will put a $sleep(0)$ before the add operation. The system will ignore this sleep operation, and the subsequent

FIGURE 8. Arguments and attacks of the TAF used in Example 5.1.

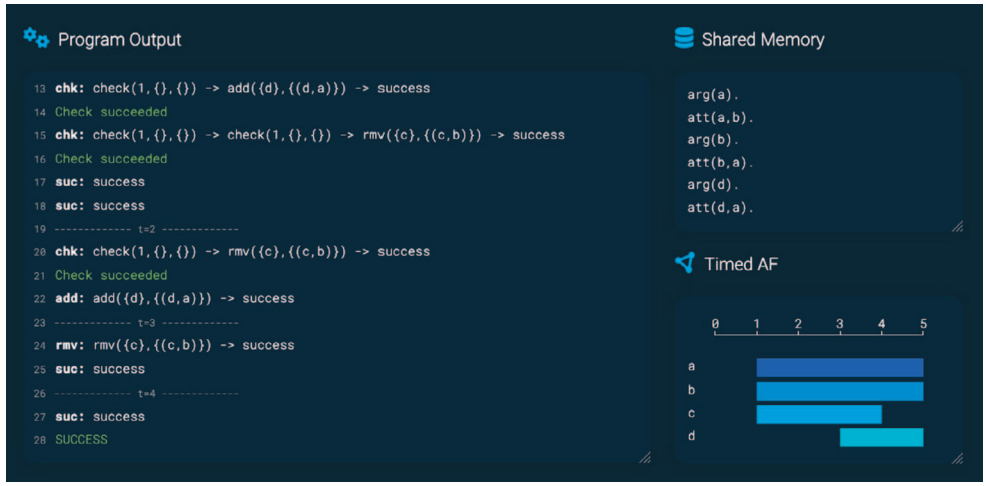

FIGURE 9. The program of Example 5.1 executed form the web interface.

action (in this case, the addition) will be executed in the same time instant. Argument c is only available in the interval $(-\infty, 3]$, and so it needs to be removed three instants after being added. This results in a remove operation on the argument c after a *sleep*(3). Finally, argument d, available only after two instants of time, is added in the last parallel branch of the program after a *sleep*(2) operation. The complete program returned by the translation is reported below.

```
sleep(0) -> add({a},{(a,b)}) -> success ||
sleep(0) -> add({b},{(b,a)}) -> success ||
sleep(0) -> add({c},{(c,b)}) -> success ||
sleep(3) -> rmv({c},{(c,b)}) -> success ||
sleep(2) -> add({d},{(d,a)}) -> success;
```

We can now run this program by passing it as input into our tool. Figure 9 shows the web interface at the end of the execution. In detail, the output panel reveals that the execution successfully terminated at time $t = 4$, while on the right side of the interface, we can see the contents of the shared memory at the end of execution and the availability of the various arguments during runtime. Since the arguments a, b and d remain available forever after being added (until $t = +\infty$), they are not removed from the shared memory at the end of execution. From the program output, we can also see how sleep shortcuts are translated into a sequence of check operations when the program is executed (like in lines 13, 15 and 20 of Figure 9).

One possible use of *tcla* concerns the evaluation of the acceptability status of arguments at a given instant of time. For example, starting with the program under consideration, we can add in parallel the process $sleep(1) \rightarrow ctest(1, \{b\}, in, complete) \rightarrow success$ to check whether the argument b is contained in some complete extension at time instant $t = 1$. In our case, the test fails, meaning that

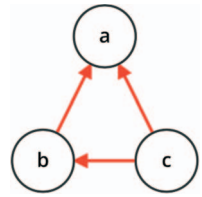

FIGURE 10. Arguments and attacks of the TAF used in Example 5.2.

going shopping one day after 'today' to take advantage of the sales is not a valid option. On the other hand, running $sleep(1) \rightarrow ctest(1, \{a\}, in, complete) \rightarrow success$, one can verify that the test succeeds. The situation is reversed when $t = 3$, for which b is accepted and a rejected according to the credulous test for the complete semantics. In practice, a possible logical conclusion to the dispute between Lisa and Mark could be reached by going to the cinema tomorrow and shopping in three days.

In the next example, we show the functioning of arguments with intermittent availability, which are added into and removed from the shared memory multiple times. This particular kind of argument can model situations where a given event occurs periodically, or a claim is reinstated under specific conditions.

EXAMPLE 5.2
Paul was planning to go to the swimming pool the next week. However, some factors limit the availability of the facility as schematized in the TAF of Figure 10, where the three arguments a, b and c have the following meanings.

- a : Paul wants to go swimming next week
- b : there will be competitions on Monday and Friday, and no free swimming hours are scheduled for those days
- c : the swimming pool will be closed every other day due to maintenance work for the next week

We associate integers from 0 to 6 with the various days of the week. 0 corresponds to Sunday, 1 to Monday and so on, until 6 which represents Saturday. Setting the start of the week on Monday, the availability function of our TAF is $Av(a) = \{[1, 6]\}$, $Av(b) = \{[1, 1], [5, 5]\}$ and $Av(c) = \{[2, 2], [4, 4], [6, 6]\}$. The translation of the given TAF produces a *tcla* program where a is added and removed only once, while b and c are repeatedly withdrawn and put back into the shared memory. We add one last operation to evaluate the acceptability of argument a and answer the question 'When is the first suitable day for Paul to go to the pool?' Therefore, the program we want to execute is as follows.

```
sleep(0) -> add({a},{}) -> success ||
sleep(6) -> rmv({a},{}) -> success ||
sleep(0) -> add({b},{(b,a)}) -> success ||
sleep(1) -> rmv({b},{(b,a)}) -> success ||
sleep(4) -> add({b},{(b,a)}) -> success ||
sleep(5) -> rmv({b},{(b,a)}) -> success ||
sleep(1) -> add({c},{(c,a)}) -> success ||
sleep(2) -> rmv({c},{(c,a)}) -> success ||
```

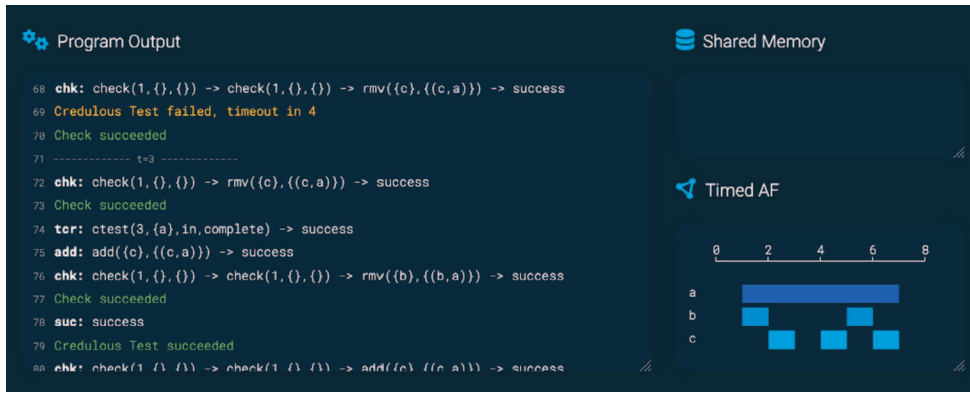

FIGURE 11. The program of Example 5.2 executed form the web interface.

```
sleep(3) -> add({c},{(c,a)}) -> success ||
sleep(4) -> rmv({c},{(c,a)}) -> success ||
sleep(5) -> add({c},{(c,a)}) -> success ||
sleep(6) -> rmv({c},{(c,a)}) -> success ||
ctest(6,{a},in,complete) -> success;
```

We see in Figure 11 the produced output, together with the status of the shared memory (which is empty since all arguments are removed by the end of the execution) and a graphical representation of the availability function. The rightmost panel contains the answer to our question: the credulous test on argument a (we use complete semantics again for illustrative purposes) succeeds on line 73 during the third step; thus, $t = 3$, i.e. Wednesday, is the first day the pool allows free swimming and is a good day for Paul to go.

6 Related Work

Due to their dynamic nature, AFs are often used to model the interaction between debating counterparts, especially in those situations where shared information is subject to changes in structure and acceptability status. As (t)cla is the first concurrent language that uses argumentation-based operations to realize interactions in multi-agent systems, there are no other simulator implementations that can be compared with ours. Therefore, in this section, we discuss relevant works in the literature that use argumentation to provide theoretical foundations and deliver applications in the context of dynamic agent interaction.

A formalism for expressing dynamics in AFs is defined in [30] as a *dynamic argumentation framework* (DAF), structures consisting of AF with attached a set of evidence, which has the role of restricting the possible arguments and attacks. This approach allows for generalizing AFs by adding the possibility of modelling changes, although the justification status of arguments is not considered in the process. Focusing on the mere addition of arguments and attacks, [3] shows under which conditions a set of arguments can be enforced (to become accepted) for specific semantics. The authors also investigate whether such modifications behave in a monotonic fashion, thus preserving the status of all initially accepted arguments. Similar kinds of frameworks have also been studied from the point of view of dynamics in works like [14, 29], where the authors investigate the

consequences brought by modifications on AFs. Differently from our study, the works mentioned above do not consider the role of time in performing changes to knowledge basis and therefore are not suitable to model the behaviour of concurrent agents.

It is reasonable to assume that the interactions between interacting entities (being them people or synthetic agents) are regulated by the passing of time [26, 27]. Timed abstract dialectical frameworks (tADFs) are introduced in [4] to cope with acceptance conditions changing over time. However, unlike TAFs, which extend classical AFs with the notion of time intervals, tADFs use generic statements that subsume arguments and relations (not only including attacks but also, for instance, supports). Therefore, tADFs are not suitable for being modelled through *tcla* processes, which, instead, rely on AFs.

A logical language for handling dynamics in AFs is presented in [19], where arguments and attacks are represented by means of propositional variables, and acceptability criteria for arguments are defined through logical formulas. Furthermore, dynamic evolution is modelled by changing the truth values of variables expressed in the *dynamic logic of propositional assignments* [1]. Two kinds of modifications can be performed in the framework: adding/removing attacks and changing the extensions set according to the desired outcome. At the same time, a straightforward mechanism for dealing with the addition (and deletion) of arguments is not implemented.

The revision of agents' beliefs is studied in [21], where the authors provide a language for representing beliefs as *defeasible logic programs*. A reasoning mechanism serves the purpose of changing the belief state of the agents: as new data are acquired, a set of rules revise the knowledge base by either incorporating or rejecting the new information. However, no proposal is made about handling possible conflicts between agents that could arise in the case of concurrency.

YALLA is an agent-based language for argumentation proposed in [18]. Similarly to the work in [19], it can be used to represent AFs, characterize extensions and modify the framework. The major difference with our language lies in the fact that, since YALLA does not consider concurrency between processes, all the operations are done sequentially, as if the agents were taking turns.

The work presented here extends three previous papers [7–9] that we discuss below. First, [8] introduces syntax and semantics of the concurrent language for argumentation (*cla*) on which *tcla* is also based. An implementation of *cla* is then given in [9]. Both *cla* and *tcla* share the same fundamental operations and allow intelligent agents to interact through an argumentation-based reasoning engine. However, *cla* does not include a notion of time, which is the prerogative of *tcla*'s constructs. Concerning [7], it introduces *tcla* (specifically, with maximum parallelism for parallel operations) and comments on the possibility of handling TAFs. The extension we provide here includes a detailed discussion of TAFs modelling and the implementation of *tcla*.

7 Conclusion and Future Work

In this paper, we discussed the practical aspects of using *tcla*, a concurrent language based on maximum parallelism that realizes the simultaneous execution of actions by parallel agents. We showed that the constructs of our language can be used to model TAFs. The transformation in *tcla* was formally presented and exemplified with TAF instances taken from the real world. We also presented an implementation of the language available online via a dedicated interface.

In future work, we would like to study time-dependent notions of acceptability. For example, we could introduce operators to check if a given argument is acceptable in all (or some) instants of time. Quantitative temporal operators could be defined to check acceptability only in given intervals. More complex dynamics could also be considered: for instance, an argument could be made available only

if specific conditions are met, and agents could be given priorities in introducing arguments for specific time intervals.

Acknowledgements

The authors are members of the INdAM Research group GNCS and 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 FICO - funded by Ricerca di Base 2021, University of Perugia; Project GIUSTIZIA AGILE, CUP_J89J22000900005.

References

[1] P. Balbiani, A. Herzig and N. Troquard. Dynamic logic of propositional assignments: A well-behaved variant of PDL. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013*, 25–28 June 2013. pp. 143–152. IEEE Computer Society, New Orleans, LA, USA, 2013.

[2] P. Baroni, M. Caminada and M. Giacomin. An introduction to argumentation semantics. *Knowl. Eng. Rev.*, **26**, 365–410, 2011.

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

[4] R. Baumann and M. Heinrich. Timed abstract dialectical frameworks: A simple translation-based approach. In *Computational Models of Argument - Proceedings of COMMA 2020*, 4–11 September 2020, H. Prakken, S. Bistarelli, F. Santini and C. Taticchi, eds. Vol. 326 of *Frontiers in Artificial Intelligence and Applications*, pp. 103–110. IOS Press, Perugia, Italy, 2020.

[5] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, **19**, 87–152, 1992.

[6] S. Bistarelli, M. Gabbrielli, M. C. Meo and F. Santini. Timed soft concurrent constraint programs: An interleaved and a parallel approach. *Theory Pract. Log. Program*, **15**, 743–782, 2015.

[7] 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, 7–9 September 2021, S. Monica and F. Bergenti, eds. Vol. 3002 of *CEUR Workshop Proceedings*, pp. 1–15. CEUR-WS.org, 2021.

[8] 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)*, 25–26 November 2020, B. Fazzinga, F. Furfaro and F. Parisi, eds. Vol. 2777 of *CEUR Workshop Proceedings*, pp. 75–89. CEUR-WS.org, 2020.

[9] S. Bistarelli and C. Taticchi. Introducing a tool for concurrent argumentation. In *Logics in Artificial Intelligence - 17th European Conference, JELIA 2021, Virtual Event*, 17–20 May 2021, W. Faber, G. Friedrich, M. Gebser and M. Morak, eds. Vol. 12678 of *Lecture Notes in Computer Science*, pp. 18–24. Springer, 2021.

[10] 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, 13–15 September 2021, C. S. Coen and I. Salvo, eds. Vol. 3072 of *CEUR Workshop Proceedings*, pp. 93–106. CEUR-WS.org, 2021.

[11] M. C. Budán, M. L. Cobo, D. C. Martínez and G. R. Simari. Bipolarity in temporal argumentation frameworks. *Int. J. Approx. Reason.*, **84**, 1–22, 2017.

[12] M. C. Budán, M. J. G. Lucero, C. I. Chesñevar and G. R. Simari. Modeling time and valuation in structured argumentation frameworks. *Inf. Sci.*, **290**, 22–44, 2015.

[13] M. Caminada. On the issue of reinstatement in argumentation. In *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006*, Liverpool, UK, 13–15 September 2006, M. Fisher, W. van der Hoek, B. Konev and A. Lisitsa, eds. Vol. 4160 of *Lecture Notes in Computer Science*, pp. 111–123. Springer, 2006.

[14] C. Cayrol, F. D. de Saint-Cyr and M.-C. Lagasquie-Schiex. Change in abstract argumentation frameworks: Adding an argument. *J. Artif. Intell. Res.*, **38**, 49–84, 2010.

[15] M. L. Cobo, D. C. Martínez and G. R. Simari. On admissibility in timed abstract argumentation frameworks. In *ECAI 2010 - 19th European Conference on Artificial Intelligence*, Lisbon, Portugal, 16–20 August 2010, H. Coelho, R. Studer and M. J. Wooldridge, eds. Vol. 215 of *Frontiers in Artificial Intelligence and Applications*, pp. 1007–1008. IOS Press, 2010.

[16] M. L. Cobo, D. C. Martínez and G. R. Simari. On semantics in dynamic argumentation frameworks. In *XVIII Congreso Argentino de Ciencias de la Computación*. SEDICI, 2013.

[17] F. S. de Boer, M. Gabbrielli and M. C. Meo. A timed concurrent constraint language. *Inf. Comput.*, **161**, 45–83, 2000.

[18] F. Dupin, P. de Saint-Cyr, C. C. Bisquert and M.-C. Lagasquie-Schiex. Argumentation update in YALLA (yet another logic language for argumentation). *Int. J. Approx. Reason.*, **75**, 57–92, 2016.

[19] 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, 20–24 July 2014, C. Baral, G. De Giacomo and T. Eiter, eds. AAAI Press, 2014.

[20] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, **77**, 321–357, 1995.

[21] M. A. Falappa, A. J. García and G. R. Simari. Belief dynamics and defeasible argumentation in rational agents. In *10th International Workshop on Non-Monotonic Reasoning (NMR 2004)*, Whistler, Canada, 6–8 June 2004, J. P. Delgrande and T. Schaub, eds, pp. 164–170, 2004.

[22] N. Halbwachs, F. Lagnier and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Software Eng.*, **18**, 785–793, 1992.

[23] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, **8**, 231–274, 1987.

[24] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, **21**, 666–677, 1978.

[25] P. LeGuernic, T. Gautier, M. Le Borgne and C. Le Maire. Programming real-time applications with signal. *Proc. IEEE*, **79**, 1321–1336, 1991.

[26] N. Mann and A. Hunter. Argumentation using temporal knowledge. In *Computational Models of Argument: Proceedings of COMMA 2008*, Toulouse, France, 28–30 May 2008, P. Besnard, S. Doutre and A. Hunter, eds. Vol. 172 of *Frontiers in Artificial Intelligence and Applications*, pp. 204–215. IOS Press, 2008.

[27] M. Julieta Marcos, M. A. Falappa and G. R. Simari. Dynamic argumentation in abstract dialogue frameworks. In *Argumentation in Multi-Agent Systems - 7th International Workshop,*

ArgMAS 2010, Toronto, ON, Canada, 10 May 2010, Revised, Selected and Invited Papers, P. McBurney, I. Rahwan and S. Parsons, eds. Vol. 6614 of *Lecture Notes in Computer Science*, pp. 228–247. Springer, 2010.

[28] R. Milner. *A Calculus of Communicating Systems*. Vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[29] N. D. Rotstein, M. O. Moguillansky, M. A. Falappa, A. J. García and G. R. Simari. Argument theory change: Revision upon warrant. In *Computational Models of Argument: Proceedings of COMMA 2008*, Toulouse, France, 28–30 May 2008, P. Besnard, S. Doutre and A. Hunter, eds. Vol. 172 of *Frontiers in Artificial Intelligence and Applications*, pp. 336–347. IOS Press, 2008.

[30] 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, pp. 131–139. UNSW Press, 2008.

[31] V. A. Saraswat, R. Jagadeesan and V. Gupta. Timed default concurrent constraint programming. *J. Symb. Comput.*, **22**, 475–520, 1996.

[32] V. A. Saraswat and M. C. Rinard. Concurrent constraint programming. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, San Francisco, California, USA, January 1990, F. E. Allen, ed., pp. 232–245. ACM Press, 1990.

Received 23 May 2022