A Streamlined Semantics for TASTD

Diego de Azevedo Oliveira[1[0000-0002-9499-5728]] and Marc Frappier[1[0000-0002-4402-2514]]

CRICUS, Université de Sherbrooke {Diego.De.Azevedo.Oliveira, Marc.Frappier}@USherbrooke.ca

Keywords: ASTD · TASTD · extended hierarchical state machines · process algebra · real-time model

Abstract. Timed Algebraic State-Transition Diagram (TASTD) is a graphical notation that allows for the combination of extended hierarchical state machines with process algebra operators. It supports the specification of real-time constraints using a global clock, timing operators and user-defined clocks. This paper proposes a simplified semantics for TASTD. The operational semantics of TASTD is defined in a tabular representation, identifying commonalities between operators, in order to facilitate its understanding and analysis. The semantics of parallel operators is simplified, removing the obligation of commutativity of actions in a synchronization and a flow; non-deterministic choice between the two possible sequential execution of actions in a parallel composition is used.

Keywords: ASTD · real-time model · case study · TASTD · formal method

1 Introduction

ASTD (Algebraic State-Transition Diagram) [1] is a graphical notation that combines process algebra operators drawn from CSP [2,3] and a variant of extended hierarchical state machines [4,5] inspired from Statecharts [6]. Timed ASTD (TASTD) [7,8] is a real-time extension for ASTD, integrating concepts from time extensions that have been proposed for several well-known languages like Statecharts, with MATLAB Stateflow [9], automata, with timed automata [10], process algebra, with Timed CSP [11,12] and Stateful Timed CSP [13]. TASTD takes advantage of the strengths of these notations: graphical representation, hierarchy, orthogonality, compositionality, and abstraction. It has been successfully applied in case studies for intrusion detection [14,15,16] and control systems [17,18].

This paper introduces a simplified semantics for TASTD. Its operational semantics is defined in a simpler manner, represented by tabular expression, identifying commonalities between operators, in order to facilitate its understanding and analysis. The semantics of parallel operators is simplified, removing the obligation of commutativity of actions in a synchronization and a flow; non-deterministic choice between the two possible sequential execution of actions in a parallel composition is used.

This article is structured as follows. Section 2 presents TASTD and its semantics. Sections 3 discusses related work. Section 4 concludes the paper.

2 TASTD

In this section, we first provide a summary of the ASTD notation, and we then define its time extension, TASTD. The semantics of the extended language is introduced.

2.1 Conventions

When defining a mathematical structure M of type $(a_1 : T_1, \ldots, a_n : T_n)$, we simply write $(a_1, \ldots, a_n)$ and introduce the types $T_1, \ldots, T_n$ and additional constraints in the text following the declaration. We refer to a component a_i of M as $M.a_i$. We use the B notation [19] to write our mathematical descriptions.

- $f \in D_1 \rightarrow D_2$ is a total function from D_1 to D_2.
- $f \in D_1 \nrightarrow D_2$ is a partial function from D_1 to D_2.
- $\mathrm{dom}(r) = \{d \mid \exists d' \cdot (d, d') \in r\}$ is the domain of relation r.
- $r[D] = \{d' \mid \exists d \cdot d \in D \wedge (d, d') \in r\}$ is the set of images of elements of D by relation r.
- $D \lhd r = \{(d, d') \mid (d, d') \in r \wedge d \in D\}$ is the domain restriction of relation r with set D.
- $r \rhd\!\!\!\rhd D = \{(d, d') \mid (d, d') \in r \wedge d' \notin D\}$ is the negative range restriction of relation r with set D.
- $r_1 \circ r_2 = \{(d, d') \mid \exists d'' \cdot (d, d'') \in r_1 \ \wedge \ (d'', d') \in r_2\}$ is the relational composition of relations r_1 and r_2. Note that the symbol ";" is used in B for both relational composition and statement composition. To avoid any confusion, we will use "$\circ$" for relational composition and ";" for statement composition.
- $\beta_1; \beta_2 = \{(d, d') \mid (d, d') \in \beta_1 \circ \beta_2 \wedge \beta_1[\{d\}] \subseteq \mathrm{dom}(\beta_2)\}$ denotes the sequential composition of actions β_1 and β_2. In this paper, actions are represented by relations. Operator ";" is monotonic with respect to B refinement. It is also called demonic composition in relational semantics of programs [20,21].
- "$\frown$" is list concatenation.

$\mathbb{T} = \mathbb{R}_{\geq 0}$ is the set of timestamps (*i.e.,* clock values) given by non-negative real numbers. $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ is the set of Boolean values. Var is a set of variables. We assume in the sequel that each variable declaration in a specification uses a variable distinct from the variables of all the other declarations, in order to avoid shadowing and to simplify the semantic definitions; shadowing is taken into account in [8]. $\mathsf{Env} = \mathsf{Var} \nrightarrow \mathsf{Type}$ is the set of *environments*: An environment is a partial function that maps variables in Var to their values in Type, which denotes the union of all types available. The substitution of free occurrences of variable x by expression v in expression w is noted $w[\![x := v]\!]$, which can be generalized to an arbitrary environment e, and noted $w[\![e]\!]$, and denoting simultaneous substitution. β denotes an action that modifies variables of an ASTD. We assume that a relational semantics is available for the action language, and we write $\beta(e, e')$ for the relation between the initial value e and final values e' induced by the execution of action β.

2.2 ASTD Syntax

ASTD is graphical notation. We provide here its abstract syntax. There are 10 types of primary ASTD, and 6 types of secondary ASTD which are defined in terms of primary ASTD. The primary ASTD types are listed in Table 1: automaton "Aut", sequence "⇨", interrupt "$\triangle$", persistent guard "$\Rightarrow_p$", choice "|", Kleene closure "$\star$", generalized parallel (*i.e.,* synchronization) "$[\![]\!]$", flow "⫛", quantified choice "|:", quantified parallel "$[\![]\!]$:", and ASTD call. The guard ASTD, which was in the original ASTD definition [1], is now a secondary ASTD defined using a persistent guard ASTD. elem is the elementary ASTD type that is used solely to type elementary states of an automaton.

There are three kinds of variables in an ASTD specification: i) ASTD parameters, denoted by P, whose values are specified when an ASTD is called, and can be modified during execution; ii) ASTD attributes, denoted by V, play the same role as attributes in a class definition of a programming language, and define part of the state of an ASTD; iii) quantified variables, which are introduced by quantified ASTD operator quantified choice and quantified generalized parallel; they are read-only.

We do not detail here the syntax of actions. Actions of executable ASTD specifications are written in C++, because this is the language in which cASTD [15], the compiler of the ASTD notation, generates an executable implementation. Actions can also be written using the generalized substitution language of B [19] or the action language of Event-B [22].

All ASTD types share common characteristics, which we define in a general (abstract) type ASTD, from which the primary and secondary ASTD types inherit:

$$\mathsf{ASTD} \triangleq (n, P, V, \beta_{init}, \beta_{astd})$$

Each ASTD has a name n, a list of parameter declarations P, a list of attribute declarations V, an action β_{init} that initializes the attributes of V, and an action β_{astd} that is executed on each transition of the ASTD. Each ASTD type includes a notion of *initial* state and *final* states, which will be defined in Section 2.4.

Syntax of Primary ASTD Types An automaton ASTD $(\mathsf{Aut}, \Sigma, Q, \nu, \delta, F, q_0, \zeta)$ is an extension of traditional automata and also represent OR states of Statecharts. $\Sigma \subseteq \mathsf{Event}$ is the alphabet. An event $\sigma \in \Sigma$ has a label and parameters; $\alpha(\sigma)$ returns the label of σ. The special event step is an element of Σ that can be used to label transitions. It denotes a transition that can be triggered by the passage of time, whereas the other elements of Σ are events submitted by the environment of the ASTD. step is drawn from Stateflow. It is tested for execution on a periodical basis; this period is defined as a parameter of the ASTD specification. $Q \subseteq \mathsf{Name}$ is the set of automaton states. $\nu \in Q \to \mathsf{ASTD}$ maps each state to its sub-ASTD, which can be elementary (noted elem) or complex (*i.e.,* of any ASTD type). δ is the set of transitions. A transition from q_1 to q_2 is labelled with the usual Statecharts notation $\sigma[g]/\beta_{tr}$, which denotes that when guard g holds, event σ can be accepted and action β_{tr} is executed. The abstract syntax of a transition is $\delta(\eta, \sigma, g, \beta_{tr}, \mathit{final}?)$. η denotes an arrow (edge) between $q_1, q_2 \in Q$. A transition (loc, q_1, q_2) denotes a local transition from q_1 to q_2. $\mathit{final}?$ is a Boolean: when $\mathit{final}? = \mathsf{true}$, the source of the transition is decorated with a bullet (i.e, $\bullet$); it indicates that the transition can be fired only if n_1 is final. $F \subseteq Q$ is the set of final states, which is partitioned into SF and DF: SF is the set of *shallow* final states, while DF denotes the set of *deep* final states, with $DF \subseteq \mathrm{dom}(\nu \rhd \{\mathsf{elem}\}$. Function $\mathcal{F}$ defined in Fig. 6 determines how these two types of final states are used. A shallow final is final irrespective of its inner state; a deep final state is final if its inner state is also final. Like in extended state machines, an automaton state can also have action declarations, given by $\zeta \in Q \to (\beta_{in}, \beta_{out}, \beta_{stay})$, which maps each state name to its actions: β_{in} is executed when a transition enters the state; β_{out} is executed when a transition leaves the state; β_{stay} is executed when a transition loops on the state, or is executed within the state.

The syntax of the other ASTD types is inspired from CSP operators, so we provide a short summary of it in the sequel. A sequence ASTD $(\Rightarrow, A_1, A_2)$ starts its execution with ASTD A_1. When A_1 is in a final state, A_2 is enabled to start its execution, but A_1 is still enabled to execute an event (*i.e.,* A_1 is not disabled when it is in a final state). When A_1 is final and both A_1 and A_2 can execute an event, the choice between them is nondeterministic. When A_2 executes its first event, A_1 becomes disabled.

An interrupt $(\triangle, A_1, A_2, \beta_{int})$ is similar to a sequence, but A_2 can start its execution without waiting for A_1 to be in a final state. A_1 can be interrupted at any point (including its initial state). If both A_1 and A_2 can execute an event, then the choice between them is nondeterministic. Action β_{int} is executed on the first event of A_2 that interrupts A_1. A_1 is disabled after being interrupted by A_2. A_2 does not have to be executed; thus, when A_1 is in a final state, then the interrupt is also in a final state, which differs from a sequence, where A_2 has to be executed after A_1. Contrary to CSP's interrupt operator, A_2 is not disabled when A_1 reaches a final state.

A persistent guard $(\Rightarrow_p, g, A_1)$ checks that condition g is satisfied on the execution of each event of A_1. A choice $(|, A_1, A_2)$ allows the first event to choose between executing A_1 or A_2; when the choice is made, the unchosen ASTD is disabled, and the chosen ASTD continues the execution. A Kleene closure $(\star, A_1)$ allows for looping on A_1: it executes A_1, and it can restart A_1 from its initial state when A_1 is in a final state. When A_1 is in a final state, and the next event can be executed both from the initial state of A_1 and its current final state, the choice between them is nondeterministic.

A flow $A = (\pitchfork, A_1, A_2)$ is inspired by the AND state of Statecharts; it executes an event on A_1 and A_2 whenever possible. More precisely, a flow A executes an event σ iff either A_1, A_2, or both A_1 and A_2 can execute it. The order in which A_1 and A_2 are executed is nondeterministic; thus the final values of attributes in A, A_1 and A_2 depend on the order of execution of A_1 and A_2. This semantics of flow differs from its original semantics introduced in [8], where the execution of the actions of A_1 and A_2 needs to be commutative. This turned out to be hard to used in practice. Using a non-deterministic choice between the two ordering allows for an implementation to use one of the execution order. Thus, the compiler cASTD can select one ordering, or the user can specify one in the ASTD editor eASTD, and this one is tested first for execution.

The generalized parallel [3] $(\llbracket\rrbracket, \Delta, A_1, A_2)$ synchronizes the execution of A_1 and A_2 on events σ whose label $\alpha(\sigma)$ belong to Δ. Note that Δ is a set of event labels, thus, contrary to CSP, the synchronization cannot be restricted to a subset of the parameters of σ. To execute an event σ such that $\alpha(\sigma) \in \Delta$, both A_1 and A_2 must be able to execute it; otherwise, the event is refused. As with the flow operator $\pitchfork$, the order in which the actions of A_1 and A_2 are executed is nondeterministic. When $\alpha(\sigma) \notin \Delta$, then either A_1 or A_2 execute σ; if both can execute σ, then the choice between them is nondeterministic. If Δ is empty, then the parameterized synchronisation is an interleave, abbreviated as $|||$. CSP's $\|$ is defined as $(\llbracket\rrbracket, \alpha(A_1) \cap \alpha(A_2), A_1, A_2)$.

We can draw an analogy between these three operators $\pitchfork$, $\llbracket\rrbracket$, $|||$ and Boolean operators. Operator $\llbracket\rrbracket$ acts like a conjunction: $(\llbracket\rrbracket, \{\alpha(\sigma)\}, E_1, E_2)$ can execute an event σ iff both E_1 and E_2 can execute it. It expresses a conjunction of ordering constraints on σ given by E_1 and E_2. It is a *hard* synchronisation. Operator $\pitchfork$ acts like an inclusive "or": $(\pitchfork, E_1, E_2)$ can execute an event σ iff either E_1, or E_2, or both E_1 and E_2 can execute it. It is a *soft* synchronisation. Operator $|||$ looks like an exclusive "or": $(|||, E_1, E_2)$ will execute σ on either E_1 or E_2, but on only one of them; if both E_1 and E_2 can execute σ, then one of them is nondeterministically chosen.

A quantified choice $(|:, x, T, A_1)$ is similar to an existential quantification in first-order logic. It declares a local variable $x \in T$ whose scope is A_1. The value of x is chosen when the first event of A_1 is executed and it cannot be modified afterwards. A quantified generalized parallel $(\llbracket\rrbracket:, x, T, \Delta, A_1)$ instantiates one copy $A_1(\![x := c]\!)$ for each value $c \in T$, and composes them with $\llbracket\Delta\rrbracket$; hence it denotes the n-ary ASTD $\llbracket\Delta\rrbracket_{c \in T} A_1(\![x := c]\!)$.

Termination in CSP is represented by the execution of the special process SKIP, which ends the execution of a process, and enables the execution of its successor when used within a sequential composition. In comparison, an ASTD does not terminate; if it is in a final state, it enables its successor within a sequential composition or a new iteration within a closure. When an ASTD is in a final state, it can still execute events, whereas a CSP process that has executed SKIP can no longer execute events. A final state in an automaton with no outgoing transition can simulate the behaviour of SKIP. Final states are determined by function $\mathcal{F}$ defined in Fig. 6. $\mathcal{F}$ is recursively defined over the structure of an ASTD.

The semantics of TASTD does not contain internal transitions τ and $\checkmark$ used in CSP. This provides for a simpler semantics of observable transitions and easier code generation.

Fig. 1 illustrate the main elements of the graphical representation of an ASTD, where *Type* denote the ASTD type and Z the components specific to an ASTD type (*e.g.*, the guard predicate). s0 is an elementary initial state, while A1 is a complex initial state which contains an ASTD also named A1. s1 is an elementary final state, and shallow by definition. A2 is a complex shallow final state, whereas A3 is a complex deep final state.

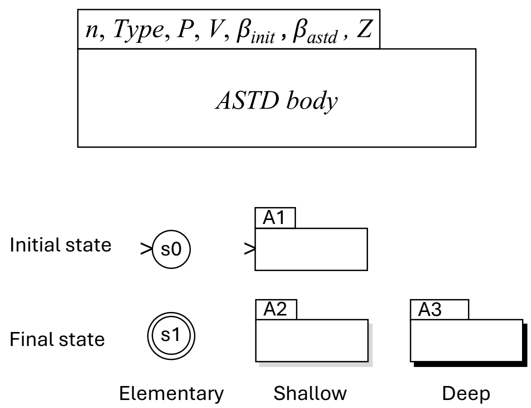

Fig. 1: The main elements of the graphical representation of an ASTD

2.3 Secondary ASTD Type

A guard $(\Rightarrow, g, A)$ is a secondary operator, defined using a persistent guard; it checks that the condition g is satisfied on the execution of the first event of A, like CSP's guard; the guard is ignored for the subsequent events. A guard is defined as follows.

$$\begin{aligned}&(n, \Rightarrow, P, V, \beta_{init}, \beta_{astd}, g, A) \triangleq \\ &\qquad (n, \Rightarrow_p, P, V \frown (b : \mathbb{B}), (\beta_{init}; b := \mathsf{false}), (\beta_{astd}; b := \mathsf{true}), g \vee b, A)\end{aligned}$$

It uses a persistent guard $g \vee b$ on a local Boolean variable b, initialized to false, and set to true after the first event is executed. Hence, the persistent guard always holds after executing the first event.

TASTD Types TASTD introduces five types to deal with time constraints: delay, persistent delay, timeout, persistent timeout, and timed interrupt. They are defined in terms of primary ASTDs. A delay (Delay, d, A) will wait d units of time before accepting the first event of ASTD A. The execution of subsequent events are not subject to the delay. A delay is represented by a guard that checks that at least d units of time have elapsed since the last event has been executed. The elapsed time is represented by the difference between the current system time cst and the last event execution time t. These two variables are predefined in the ASTD notation.

$$(n, \mathsf{Delay}, P, V, \beta_{init}, \beta_{astd}, d, A) \triangleq (n, \Rightarrow, P, V, \beta_{init}, \beta_{astd}, \mathsf{cst} - \mathsf{t} > d, A)$$

A persistent delay (PDelay, d, A) will apply the delay d to the execution of each event of A. A persistent delay is represented by a persistent guard in a manner similar to the delay.

$$(n, \mathsf{PDelay}, P, V, \beta_{init}, \beta_{astd}, d, A) \triangleq (n, \Rightarrow_p, P, V, \beta_{init}, \beta_{astd}, \mathsf{cst} - \mathsf{t} > d, A)$$

A timeout $(\mathsf{TO}, d, \beta_{to}, A_1, A_2)$ must execute its first event on A_1 within d units of time, otherwise A_1 is disabled, and A_2 takes over and executes the subsequent events. If A_1 succeeds in executing the first event within d, then A_2 is disabled, and A_1 continues the execution without any further time constraint. Action β_{to} is executed when a timeout occurs. A timeout is easier to define using its graphical representation provided in Fig 2. The timeout n is represented by an interrupt n on n_1 and n_2. n_1 is a guard on A_1 checking that no more than d units of time have elapsed since the last executed event. This ensure that the first event of A_1 must occurs before d units of time; otherwise the guard is false and n_1 is disabled. A boolean b is added to list of variable declarations of n and initialized to false. The execution of the first event of A_1 sets b to true by the ASTD action of A_1. This disables the execution of n_2, because n_2 is a guard checking that d units of time have elapsed and that A_1 has not executed its first event. n_2 is applied to a choice n_3 between a step event in automaton n_4 and A_2. So either step or the first event of A_2 can interrupt A_1. After executing step, n_4 enables A_2. Action β_{to} is executed when the interrupt occurs.

A persistent timeout $(\mathsf{PTO}, d, \beta_{to}, A_1, A_2)$ will apply d to each event of A_1; if d is missed, then A_2 takes over, and A_1 is disabled. A persistent timeout is defined in a similar fashion in Fig 3 as the timeout. A persistent guard is used in n_1, so that the time out is checked on each event execution of A_1. There is no need to declare a boolean in that case.

A timed interrupt $(\mathsf{TI}, d, \beta_{to}, A_1, A_2)$ executes A_1 for up to d units of time; execution is transferred to A_2 after d, and A_1 is disabled. Action β_{to} is executed when the interrupt occurs. A time interrupt is also defined in a similar fashion in Fig 4 as a timeout. n declares a timestamp variable ts which is initialized to the last event execution time t. n_1 is a persistent guard that allows A_1 to execute for d units of time. When that time is up, the guard of n_2 is enabled, allowing a step event or the first event of A_2 to execute.

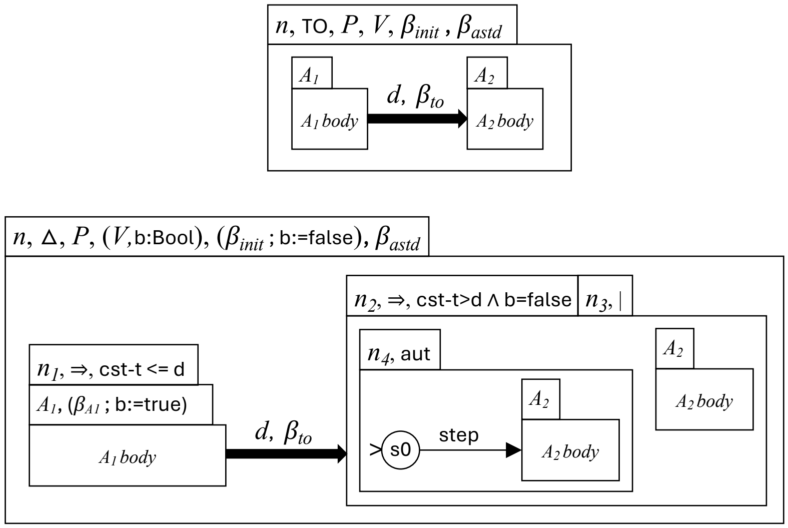

Fig. 2: At the bottom, the ASTD defining a timeout $(n, \mathsf{TO}, P, V, \beta_{init}, \beta_{astd}, d, \beta_{to}, A_1, A_2)$ illustrated at the top

2.4 TASTD Semantics

In this section, we introduce the modifications made to adapt the operational semantics of ASTDs [1] to deal with time. This semantics is defined using inference rules in the Plotkin style.

Transition System The semantics of a TASTD $a(P)$, where P are the parameters of a, consists of a labelled transition system (LTS) $\mathcal{L}$, which is a subset of $\mathcal{S} \times \Sigma \times \mathcal{S}$, where $\mathcal{S} \triangleq \mathsf{S}_a \times \mathbb{T} \times \mathsf{Env}$, and S_a is the set of states of ASTD a that shall be inductively defined in the sequel (see Table 1). The LTS represents a set of transitions of the form $(s, \mathtt{t}, p) \xrightarrow{\sigma} (s', \mathtt{t}', p')$ denoting that a TASTD can execute event σ from state s and move to state s'. Symbols p, p' respectively denote the before (p) and after (p') values of the ASTD parameters P, meaning that ASTD parameters can be modified during execution. Symbols $\mathtt{t}, \mathtt{t}'$ respectively denote the time at which states s, s' were reached; hence, they denote the timestamp of the *last executed event*. The value of $\mathtt{t}$ for the system's initial state is some timestamp, which represents the system start time. The timestamp $\mathtt{t}$ of a state s is needed when deciding on various time operators. For instance, a timeout is evaluated with respect to the last event executed. TASTD time operators simulate clocks, relying on the time of the last event executed for that purpose. Thus, when using a TASTD operator, there is no need to define a clock to specify time constraints. However, clocks can be declared as TASTD attributes and used to specify arbitrary time constraints. A state s contains its own attributes declared within the ASTD and control values that represent the behaviour of various ASTD operators.

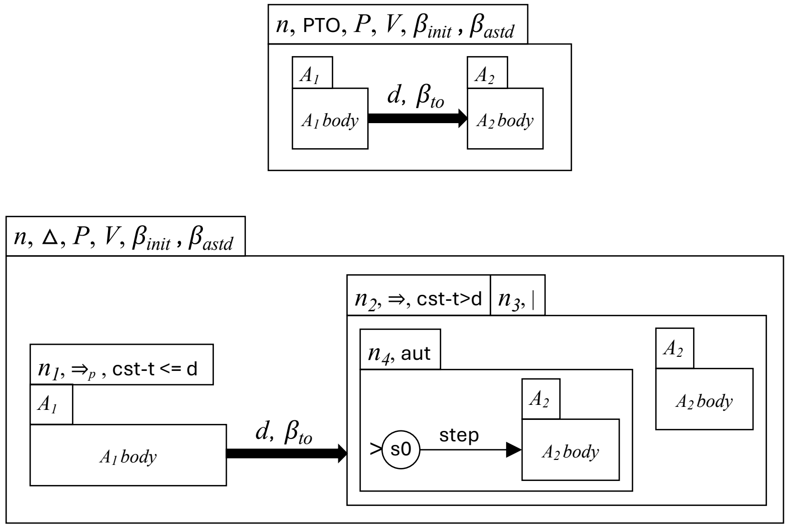

Fig. 3: At the bottom, the ASTD defining a persistent timeout $(n, \mathsf{PTO}, P, V, \beta_{init}, \beta_{astd}, d, \beta_{to}, A_1, A_2)$ illustrated at the top

A TASTD state includes a timestamp that denotes the time at which the state was reached. TASTD rely on the availability of a global clock called cst, which stands for *current system time*. It is used in various time operators to represent time constraints and simulate clocks. In ASTD, a transition can be triggered only by external events. In TASTD, a transition can be triggered by the passage of time; such a transition is labelled by a special event called step, drawn from Stateflow; it also corresponds to time transition in timed automaton. Automaton transitions can be labelled with step, and some time operators implicitly include step transitions. When implementing a TASTD, the executability of a time-triggered transition is checked on a periodical basis, according to the desired time granularity required to match system time constraints.

The semantics of TASTDs is designed for generating executable code. It differs from the semantics of timed automata and timed CSP, where there are transitions on external events σ of the form $s \xrightarrow{\sigma} s'$ and transitions on the passage of time with d units $s \xrightarrow{d} s'$, which are more suitable for model-checking. A TASTD transition $(s, t, p) \xrightarrow{\sigma} (s', t', p')$ corresponds to two successive transitions $s \xrightarrow{t'-t} s$ and $s \xrightarrow{\sigma} s'$ in timed automata.

Because ASTD can declare local variables, and that a nested ASTD can use variables declared in its enclosing ASTDs, we use environments in an auxiliary transition relation called $\mathcal{L}_a$ which is used to define $\mathcal{L}$. Suppose that a_2 is a sub-TASTD of a_1. a_1 may declare variables that a_2 can use and modify. Thus, the behaviour of a_2 depends on the variables declared in its enclosing ASTD a_1.

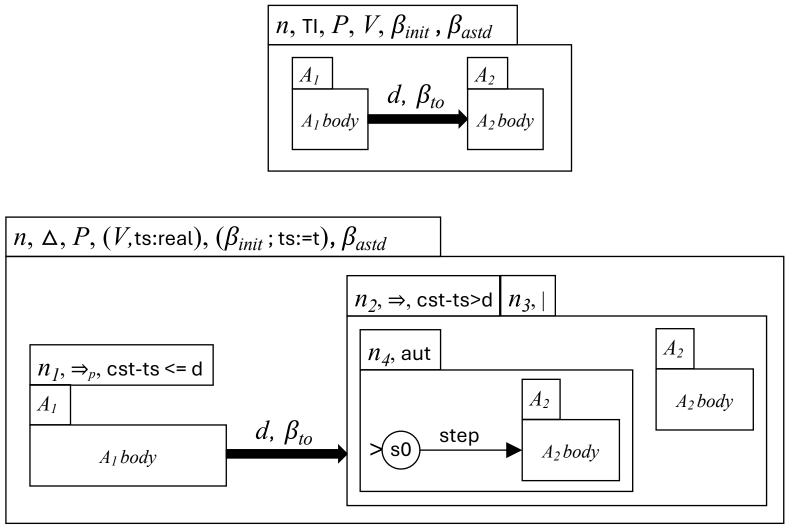

Fig. 4: At the bottom, the ASTD defining a timed interrupt $(n, \mathsf{TI}, P, V, \beta_{init}, \beta_{astd}, d, \beta_{to}, A_1, A_2)$ illustrated at the top

A transition of $\mathcal{L}_a$ has the following form:

$$s \xrightarrow{\sigma, \mathsf{t}, e, e'}_a s'$$

where σ is the event executed, e, e' are environments respectively containing the before and after values of variables in the ASTDs enclosing ASTD a. These variables could be ASTD parameters, attributes, or quantified variables introduced by quantified operators like choice and generalized parallel. The time of the last executed event is denoted by timestamp t.

The initial state of a system whose main ASTD is a, called with parameter values p is

$$(\mathcal{I}(a, \mathsf{cst}, p), \mathsf{cst}, p)$$

We distinguish between a system state and an ASTD state. A *system state* (s, t, p) is made of the state of its main ASTD a, the last event execution time t and the values p of the parameters of a. $\mathcal{I}$ is a function that returns the initial state of an ASTD (see Fig. 6). $\mathcal{I}$ receives the current system time cst to initialize the local clocks of sub-ASTDs of a that must deal with local time constraints, *i.e.*,, generalized parallel and flow.

The following main inference rule connects $\mathcal{L}$ to $\mathcal{L}_a$:

$$\mathsf{env} \; \frac{s \xrightarrow{\sigma, \mathsf{t}, p, p'}_a s'}{(s, \mathsf{t}, p) \xrightarrow{\sigma}_a (s', \mathsf{cst}, p')}$$

It states that a transition in $\mathcal{L}$ is proved by proving a transition in $\mathcal{L}_a$ using the before values p of the parameters P of a, their final values p', and the last event execution time t. The current system time cst is stored in the global state as the new last event execution time. The operational semantics of ASTD is inductively defined on $\mathcal{L}_a$ for each ASTD type.

ASTD States The type S_a of a state s of an ASTD a depends on the type of a. This differs from the process algebra style, where a state is given by a process term. In the sequel, for the sake of convenience and concision, we define $\mathsf{S}_{A_0} \triangleq \{\bot\}$, such that $s \in \mathsf{S}_{A_0}$ denotes an undefined state $s = \bot$, which means that the corresponding ASTD is not initialized, and thus has not started its execution. In Table 1, we define a state structure for each ASTD type. Each state contains the values v of its ASTD attributes. One must not confuse a state $q \in Q$ of an automaton with the state of an ASTD. The state $s \in \mathsf{S}_a$ of an automaton a contains more information than q; q is just a label for a node in an automaton, as in traditional automata. s constain all the information necessary to define $\mathcal{L}_a$. In particular, each state s contains the values $v \in \mathsf{Env}$ of attributes V declared in a, as well as other control information to define operator behaviour. The state of an automaton is given by both q and the state s of its sub-ASTD $\nu(q)$; when q is elementary, then $\nu(q) = \mathsf{elem}$ and $s = \mathsf{elem}_\circ$. We omit in Table 1 the definition of history states h, drawn from Statecharts; details can be found in [8]. State component i in a sequence and an interrupt indicates which ASTD A_i it is currently executing. The sub-state s of choice and closure is undefined ($s \in \mathsf{S}_{A_0}$) in their initial state. Flow and general parallel contain the states s_i and last event execution time t_i of each of their sub-ASTD A_i, which means that their operands have their own local clock to deal with time constraints. Quantified choice stores the value v_x of its quantified variable x, which is undefined (*i.e.,* $v_x = \varnothing \wedge s \in A_0$) in its initial state. Functions f and u of a quantified general parallel respectively contain the current states and the last event execution time of their instances, while t records the time where the instances become available for execution, and is used to initialize $u(d)$ when instance $d \in T$ is started.

ASTD Initial States and Final States Function $\mathcal{I}(a, t, e)$ defined in Fig. 5 returns the initial state of ASTD a using last event execution time t and environment e that provides the current values of variables declared in ASTDs enclosing a. Function $\mathcal{F}(a, s)$ defined in Fig. 6 returns true if state s of ASTD a is final. Function final does not depend on the values of the variables or the last event execution time; it depends essentially on the state of the leaf ASTDs in a, which are automata. When a sub-ASTD A_i of an ASTD a is not initialized, then its initial state $\mathcal{I}(a.A_i, \_, \_)$ is used to determine if a is final.

Syntax ASTD Type	Semantics ASTD State	Description
elem	$\mathsf{elem}_\circ$	state of an elementary automaton state
$(\mathsf{Aut}, \Sigma, Q, \delta, F, q_0, \zeta, \nu)$	$(\mathsf{Aut}_\circ, v, q, s)$	$q \in Q,\ s \in \mathsf{S}_{a.\nu(q)}$
$(⇨, A_1, A_2)$	$(⇨_\circ, v, i, s)$	$i \in 1..2 \wedge s \in \mathsf{S}_{A_i}$
$(\triangle, \beta_{int}, A_1, A_2)$	$(\triangle_\circ, v, i, s)$	$i \in 1..2 \wedge s \in \mathsf{S}_{A_i}$
$(\Rightarrow_p, g, A_1)$	$(\Rightarrow_{p\circ}, v, s)$	$s \in \mathsf{S}_{A_1}$
$(\mid, A_1, A_2)$	$(\mid_\circ, v, i, s)$	$i \in 0..2 \wedge s \in \mathsf{S}_{A_i}$
$(\star, A_1)$	$(\star_\circ, v, s)$	$i \in 0..1 \wedge s \in \mathsf{S}_{A_i}$
$(\pitchfork, A_1, A_2)$	$(\pitchfork_\circ, v, s_1, t_1, s_2, t_2)$	$s_i \in \mathsf{S}_{A_i} \wedge t_i \in \mathbb{T}$
$(\|[]\|, \varDelta, A_1, A_2)$	$(\|[]\|_\circ, v, s_1, t_1, s_2, t_2)$	$s_i \in \mathsf{S}_{A_i} \wedge t_i \in \mathbb{T}$
$(\|:, x, T, A_1)$	$(\|:_\circ, v, v_x, s)$	$v_x \in \{x\} \rightarrow\!\!\!\!\!\!+ T$ $i \in 0..1 \wedge s \in \mathsf{S}_{A_i}$
$(\|[]\|:, x, T, \varDelta, A_1)$	$(\|[]\|:_\circ, v, t, f, u)$	$t \in \mathbb{T}$ $f \in T \rightarrow\!\!\!\!\!\!+ \mathsf{S}_{A_1}$ $u \in T \rightarrow\!\!\!\!\!\!+ \mathbb{T}$

Table 1: Primary ASTD types and their state structure for defining $\mathcal{L}_a$

$$
\begin{array}{l}
\mathcal{I}(a : \mathsf{ASTD}, t : \mathbb{T}, e : \mathsf{Env}) : \mathsf{S}_a \triangleq \\
\quad \textbf{let}\ \ v \text{ be such that } a.\beta_{init}(e, v), \\
\qquad\ e' = v \cup e \ \textbf{in} \\
\quad \textbf{match}\ a\ \textbf{with} \\
\qquad \begin{array}{r@{\,:\,}l}
\mathsf{elem} & \mathsf{elem}_\circ \\
(\mathsf{Aut}, \Sigma, Q, \delta, F, q_0, \zeta, \nu) & (\mathsf{Aut}_\circ, v, q_0, \mathcal{I}(\nu(q_0), t, e')) \\
(⇨, A_1, A_2) & (⇨_\circ, v, 1, \mathcal{I}(a.A_1, t, e')) \\
(\triangle, A_1, A_2, \beta_{int}) & (\triangle_\circ, v, 1, \mathcal{I}(a.A_1, t, e')) \\
(\Rightarrow_p, g, A_1) & (\Rightarrow_{p\circ}, v, \mathcal{I}(a.A_1, t, e')) \\
(\mid, A_1, A_2) & (\mid_\circ, v, 0, \bot) \\
(\star, A_1) & (\star_\circ, v, \bot) \\
(\pitchfork, A_1, A_2) & (\pitchfork_\circ, v, \mathcal{I}(a.A_1, t, e'), t, \mathcal{I}(a.A_2, t, e'), t) \\
(|[]|, \varDelta, A_1, A_2) & (|[]|_\circ, v, \mathcal{I}(a.A_1, t, e'), t, \mathcal{I}(a.A_2, t, e'), t) \\
(|:, x, T, A_1) & (|:_\circ, v, \varnothing, \bot) \\
(|[]|:, x, T, \varDelta, A_1) & (|[]|:_\circ, v, t, \varnothing, \varnothing)
\end{array}
\end{array}
$$

Fig. 5: Function $\mathcal{I}$ that returns the initial state of an ASTD

$$
\begin{array}{l}
\mathcal{F}(a : \mathsf{ASTD}, s' : \mathsf{S}_a) : \mathbb{B} \triangleq \\
\textbf{match } s' \textbf{ with} \\
\begin{array}{rl}
(\mathsf{Aut}_\circ, v, q, h, s) : & q \in a.SF \vee (q \in a.DF \wedge \mathcal{F}(a.\nu(q), s)) \\
(\Rightarrow_\circ, v, i, s) : & \textbf{if } i = 1 \textbf{ then} \\
& \quad \mathcal{F}(a.A_1, s) \wedge \mathcal{F}(a.A_2, \mathcal{I}(a.A_2, \_, \_)) \\
& \textbf{else } \mathcal{F}(a.A_2, s) \\
(\triangle_\circ, v, i, s) : & \mathcal{F}(a.A_i, s) \\
(\Rightarrow_{p\circ}, v, s) : & \mathcal{F}(a.A_1, s) \\
(|_\circ, v, i, s) : & \textbf{if } i = 0 \textbf{ then} \\
& \quad \mathcal{F}(a.A_1, \mathcal{I}(a.A_1, \_, \_)) \vee \mathcal{F}(a.A_2, \mathcal{I}(a.A_2, \_, \_)) \\
& \textbf{else } \mathcal{F}(a.A_i, s) \\
(\star_\circ, v, s) : & s = \bot \vee \mathcal{F}(a.A_1, s) \\
(\pitchfork_\circ, v, s_1, t_1, s_2, t_2) : & \mathcal{F}(a.A_1, s_1) \wedge \mathcal{F}(a.A_2, s_2) \\
(\llbracket\rrbracket_\circ, v, s_1, t_1, s_2, t_2) : & \mathcal{F}(a.A_1, s_1) \wedge \mathcal{F}(a.A_2, s_2) \\
(|:_\circ, v, v_x, s) : & \textbf{if } s = \bot \textbf{ then} \\
& \quad \mathcal{F}(a.A_1, \mathcal{I}(a.A_1, \_, \_)) \\
& \textbf{else } \mathcal{F}(a.A_i, s) \\
(\llbracket\rrbracket:_\circ, v, t, f, u) : & (\forall c \in \mathrm{dom}(f) \cdot \mathcal{F}(a.A_1, f(c))) \wedge \\
& (\mathrm{dom}(f) \neq T \Rightarrow \mathcal{F}(a.A_1, \mathcal{I}(a.A_1, \_, \_)))
\end{array}
\end{array}
$$

Fig. 6: Boolean function $\mathcal{F}$ that determines if an ASTD state is final

Inference Rules Rule r_1 describes a transition between local states of an automaton.

$$
\mathsf{r}_1 \; \frac{\begin{array}{l}
(1)\ a.\delta((\mathsf{loc}, q_1, q_2), \sigma', g, \beta_{tr}, \mathit{final?}) \\
(2)\ \big(\sigma' = \sigma \ \wedge\ (\mathit{final?} \Rightarrow \mathcal{F}(a.\nu(q_1), s)) \ \wedge\ g\big)(\![e]\!) \\
(3)\ \textbf{if } q_1 = q_2 \\
(4)\quad \textbf{then } \beta = \beta_{tr} \ ;\ a.\zeta(q_1).\beta_{stay} \ ;\ a.\beta_{astd} \\
(5)\quad \textbf{else } \beta = a.\zeta(q_1).\beta_{out} \ ;\ \beta_{tr} \ ;\ a.\zeta(q_2).\beta_{in} \ ;\ a.\beta_{astd} \\
(6)\ \beta(e \cup v, e' \cup v')
\end{array}}{(\mathsf{Aut}_\circ, q_1, v, s_1) \xrightarrow{\sigma, t, e, e'}_a (\mathsf{Aut}_\circ, q_2, v', \mathcal{I}(a.\nu(q_2), t, e'))}
$$

Hypothesis (1) indicates that there is a transition from q_1 to q_2 labeled with σ' in automaton a. Hypothesis (2) checks that the transition label σ', which may contain variables, matches the event σ to be executed, after applying the environment e as a substitution; it also checks that q_1 is in a final state if the transition is marked as final, and that its guard holds. Hypotheses (4) to (6) determine the action to execute. The state and transition actions are executed before the astd action β_{astd}. The traditional semi-colon ";" is used to indicate sequential composition of actions. If $q_1 = q_2$, the stay action β_{stay} is executed, followed by the transition action β_{tr}; otherwise, the exit, transition and entry actions are executed.

The other rules for Aut, $\Rightarrow$, $\triangle$, $\Rightarrow_p$, $|$ and $\star$ are defined following general form below, and compactly represented in tabular format in Table 2.

$$
\mathsf{r}_2..\mathsf{r}_{11}\ \frac{\begin{array}{l}(1)\ s_1 \xrightarrow{\sigma,t,e\cup v,e_2}_{a.A_i} s' \\ (2)\ \beta(e_2,e_3) \\ (3)\ e' = \mathrm{dom}(e) \lhd e_3 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_3 \\ (4)\ \Theta\end{array}}{(\mathsf{A}_\circ, v, \gamma, s_1) \xrightarrow{\sigma,t,e,e'}_a (\mathsf{A}_\circ, v', \gamma', s')}
$$

Hypothesis (1) indicates that the new state s' of sub-ASTD $a.A_i$ is determined by the execution of $a.A_i$ from state s_1 under environment $e \cup v$ on event σ with last event execution time t. e_2 contains the new (intermediate) values of e and v. Hypothesis (2) indicates that action β is executed on e_2, producing the final values e', v' stored in e_3; thus, this means that actions are executed bottom-up in an ASTD hierarchy. Θ is a condition that depends on the particular case that the rule is dealing with. γ and γ' denote the current state q in an automaton, or i in ASTD types ⇨, $\vartriangle$ and $|$. Table 2 provides a compact representation of the rules that follow this pattern and defines the values of γ, γ', A_i, s_1, β and Θ of this rule pattern, and it allows for an easy comparison of the rules.

r	Type	γ	γ'	A_i	s_1	β	Θ
r_2	Aut	q	q	$a.\nu(q)$	s	$a.\zeta(q).\beta_{stay};$ $a.\beta_{astd}$	
r_3	⇨	i	i	$a.A_i$	s	$a.\beta_{astd}$	$i : 1..2$
r_4	⇨	1	2	$a.A_2$	$\mathcal{I}(a.A_2, t, e_1)$	$a.\beta_{astd}$	$\mathcal{F}(a.A_1, s)$
r_5	$\vartriangle$	i	i	$a.A_i$	s	$a.\beta_{astd}$	$i : 1..2$
r_6	$\vartriangle$	1	2	$a.A_2$	$\mathcal{I}(a.A_2, t, e_1)$	$a.\beta_{int};$ $a.\beta_{astd}$	
r_7	$|$	0	i	$a.A_i$	$\mathcal{I}(a.A_i, t, e_1)$	$a.\beta_{astd}$	$i \in 1..2$
r_8	$|$	i	i	$a.A_i$	s	$a.\beta_{astd}$	$i \in 1..2$
r_9	$\Rightarrow_p$			$a.A_1$	s	$a.\beta_{astd}$	$g(\!\lbrack e \rbrack\!)$
r_{10}	$\star$			$a.A_1$	$\mathcal{I}(a.A_1, t, e_1)$	$a.\beta_{astd}$	$\mathcal{F}(a, (\star_\circ, v, s))$
r_{11}	$\star$			$a.A_1$	s	$a.\beta_{astd}$	

Table 2: Inference rules for Aut, ⇨, $\vartriangle$, $\Rightarrow_p$, $|$ and $\star$

Inference Rules for Flow A flow ASTD $(\pitchfork, A_1, A_2)$ tries to execute both A_1 and A_2. Rule r_{12} deals with the case where one ASTD, noted A_{ok}, can execute the event, but not the other (A_{ko}), which is denoted by $s_{ko} \overset{\sigma,t_{ko},e_1,-}{\not\longrightarrow}_{a.A_{ko}}$. Hypothesis (1) matches $\{A_1, A_2\}$ with $\{A_{ok}, A_{ko}\}$, allowing to write a single rule for either A_1 or A_2 executing the event. Rule r_{13} deals with the case where both A_1 and A_2 can execute the event. Hypothesis (1) states that the order in which they are executed is non-deterministic, by allowing either A_1 or A_2 to execute first; i denotes the first ASTD to execute and j denotes the second ASTD. Hypothesis (2) executes the first ASTD i, followed by the second ASTD j

using the values of the variables e_1 obtained after executing the first ASTD. Note that the last event execution times used to compute a transition in each sub-ASTD A_1 and A_2 are the ones stored in the flow state, represented respectively by t_1 and t_2. These values are updated with cst when they execute.

$$\mathsf{r}_{12}\ \frac{\begin{array}{l}(1)\ \{(A_1, s_1, t_1, s_1', t_1'), (A_2, s_2, t_2, s_2', t_2')\} = \{(A_{ok}, s_{ok}, t_{ok}, s_{ok}', \mathsf{cst}), (A_{ko}, s_{ko}, t_{ko}, s_{ko}, t_{ko})\} \\ (2)\ s_{ok} \xrightarrow{\sigma, t_{ok}, e \cup v, e_1}_{a.A_{ok}} s_{ok}' \quad \wedge \quad s_{ko} \not\xrightarrow{\sigma, t_{ko}, e_1, -}_{a.A_{ko}} \\ (3)\ \beta_{astd}(e_1, e_2) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_2 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_2\end{array}}{(\pitchfork_\circ, v, s_1, t_1, s_2, t_2) \xrightarrow{\sigma, -, e, e'}_a (\pitchfork_\circ, v', s_1', t_1', s_2', t_2')}$$

$$\mathsf{r}_{13}\ \frac{\begin{array}{l}(1)\ \{(A_1, s_1, t_1, s_1'), (A_2, s_2, t_2, s_2')\} = \{(A_i, s_i, t_i, s_i'), (A_j, s_j, t_j, s_j')\} \\ (2)\ s_i \xrightarrow{\sigma, t_i, e \cup v, e_1}_{a.A_i} s_i' \quad \wedge \quad s_j \xrightarrow{\sigma, t_j, e_1, e_2}_{a.A_j} s_j' \\ (3)\ \beta_{astd}(e_2, e_3) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_3 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_3\end{array}}{(\pitchfork_\circ, v, s_1, t_1, s_2, t_2) \xrightarrow{\sigma, -, e, e'}_a (\pitchfork_\circ, v', s_1', \mathsf{cst}, s_2', \mathsf{cst})}$$

Inference Rules for Generalized Parallel The rules for a generalized parallel $(|[]|, \Delta, A_1, A_2)$ are very similar to a flow, except that events for which synchronization is required are specified using Δ. Thus, if the label $\alpha(\sigma)$ of σ is not in Δ (rule r_{14}), then only one of A_1, A_2 is allowed to execute; if both can execute σ, then one is chosen non-deterministically. If $\alpha(\sigma) \in \Delta$ (rule r_{15}), then a transition occurs iff both A_1 and A_2 can execute it. A_i denotes the first to execute, while A_j executes after A_i from the environment e_1 computed by A_i (hypothesis (2)). The order in which A_1 and A_2 are executed is non-deterministic (hypothesis (1) and (2)).

$$\mathsf{r}_{14}\ \frac{\begin{array}{l}(1)\ \{(A_1, s_1, t_1, s_1', t_1'), (A_2, s_2, t_2, s_2', t_2')\} = \{(A_{ok}, s_{ok}, t_{ok}, s_{ok}', \mathsf{cst}), (A_{ko}, s_{ko}, t_{ko}, s_{ko}, t_{ko})\} \\ (2)\ \alpha(\sigma) \notin \Delta \quad \wedge \quad s_{ok} \xrightarrow{\sigma, t_{ok}, e \cup v, e_1}_{a.A_{ok}} s_{ok}' \\ (3)\ \beta_{astd}(e_1, e_2) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_2 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_2\end{array}}{(|[]|_\circ, v, s_1, t_1, s_2, t_2) \xrightarrow{\sigma, -, e, e'}_a (|[]|_\circ, v', s_1', t_1', s_2', t_2')}$$

$$\mathsf{r}_{15}\ \frac{\begin{array}{l}(1)\ \{(A_1, s_1, t_1, s_1'), (A_2, s_2, t_2, s_2')\} = \{(A_i, s_i, t_i, s_i'), (A_j, s_j, t_j, s_j')\} \\ (2)\ \alpha(\sigma) \in \Delta \quad \wedge \quad s_i \xrightarrow{\sigma, t_i, e \cup v, e_1}_{a.A_i} s_i' \quad \wedge \quad s_j \xrightarrow{\sigma, t_j, e_1, e_2}_{a.A_j} s_j' \\ (3)\ \beta_{astd}(e_2, e_3) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_3 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_3\end{array}}{(|[]|_\circ, v, s_1, t_1, s_2, t_2) \xrightarrow{\sigma, -, e, e'}_a (|[]|_\circ, v', s_1', \mathsf{cst}, s_2', \mathsf{cst})}$$

Inference Rules for Quantified Choice Rule r_{16} deals with the case where a quantified choice $(|:, x, T, A_1)$ has not started its execution, and it allows for the non-deterministic choice of a value d for x such that its sub-ASTD A_1 can execute a transition with d. Rule 17 caters for the other transitions once the value of x is chosen.

$$\mathsf{r}_{16}\ \frac{\begin{array}{l}(1)\ d \in a.T \quad \wedge \quad \mathcal{I}(a.A_1, t, e \cup v) \xrightarrow{\sigma,t,e\cup v\cup\{x\mapsto d\},e_1}_{a.A_1} s' \\ (2)\ \beta_{astd}(e_1,e_2) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_2 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_2\end{array}}{(|\!:_\circ, v, v_x, s) \xrightarrow{\sigma,t,e,e'}_a (|\!:_\circ, v', \{x \mapsto d\}, s')}$$

$$\mathsf{r}_{17}\ \frac{\begin{array}{l}(1)\ s \xrightarrow{\sigma,t,e\cup v\cup v_x,e_1}_{a.A_1} s' \\ (2)\ \beta_{astd}(e_1,e_2) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_2 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_2\end{array}}{(|\!:_\circ, v, v_x, s) \xrightarrow{\sigma,t,e,e'}_a (|\!:_\circ, v', v_x, s')}$$

Inference Rules for Quantified Generalized Parallel A quantified generalized parallel $(|[\,]|\!:, x, T, \Delta, A_1)$ executes one copy of A_1 for each value d of T. Since f and u are initialized to $\varnothing$, f' and u' denote their extension with initialization values for elements of T not started yet. They are used in rules r_{18} and r_{19} and defined as follows:

$$\begin{array}{l} f' = f \ \cup\ (T - dom(f)) \times \mathcal{I}(a.A_1, t, e \cup v) \\ u' = u \ \cup\ (T - dom(u)) \times \{t\} \end{array}$$

The last event execution to start each instance of A_1 is the one used to initialized the quantified ASTD. Rule r_{18} deals with the case where there is no synchronization and only one instance executes the event; the last event execution time t used is the one stored in the state of the quantified generalized parallel $(|[\,]|\!:_\circ, v, t, f, u)$, not the one given in $\xrightarrow{\sigma,\_,e,e'}_a$, represented by an "_".

$$\mathsf{r}_{18}\ \frac{\begin{array}{l}(1)\ \alpha(\sigma) \notin \Delta \quad \wedge \quad d \in T \quad \wedge \quad f'(d) \xrightarrow{\sigma,u'(d),e\cup v\cup\{x\mapsto d\},e_1}_{a.A_1} s' \\ (2)\ \beta_{astd}(e_1,e_2) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_2 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_2\end{array}}{(|[\,]|\!:_\circ, v, t, f, u) \xrightarrow{\sigma,\_,e,e'}_a (|[\,]|\!:_\circ, v', t, f \mathbin{\lhd\!\!+} \{d \mapsto s'\}, u \mathbin{\lhd\!\!+} \{d \mapsto \mathsf{cst}\})}$$

Rule r_{19} deals with the case of synchronization, thus, each instance must be able to execute the event (hyp. (2)). It non-deterministically picks a permutation p of T which denotes the order in which the instances of A_1 are executed (hyp. (1)). g is a function that records the intermediate values of e and v during the execution of the k instances, with $g(0)$ denoting their initial values, and $g(k)$ their final values (hyp. (3)). f'' denote the states of the instances after their execution.

$$\mathsf{r}_{19}\ \frac{\begin{array}{l}(1)\ \alpha(\sigma) \in \Delta \quad \wedge \quad k = |T| \quad \wedge \quad p \in \pi(T) \\ (2)\ \forall d \in T \cdot f'(d) \xrightarrow{\sigma,u'(d),e\cup v\cup\{x\mapsto d\},\_}_{a.A_1} \_ \\ (3)\ g \in 0..k \rightarrow\!\!\!\!\!+ \mathsf{Env} \quad \wedge \quad g(0) = e \cup v \quad \wedge \quad g(k) = e_1 \\ (4)\ \forall i \in 1..k \cdot f'(p(i)) \xrightarrow{\sigma,u(p(i)),g(i-1)\lhd\!\!+\{x\mapsto p(i)\},g(i)}_{a.A_1} f''(p(i)) \\ (5)\ \beta_{astd}(e_1,e_2) \quad \wedge \quad e' = \mathrm{dom}(e) \lhd e_2 \quad \wedge \quad v' = \mathrm{dom}(v) \lhd e_2\end{array}}{(|[\,]|\!:_\circ, v, t, f, u) \xrightarrow{\sigma,\_,e,e'}_a (|[\,]|\!:_\circ, v', t, f'', T \times \{\mathsf{cst}\})}$$

3 Related Work

The ASTD notation was designed to specify control and monitoring systems in an abstract, compositional manner and to automatically generate an efficient implementation from a specification. It is inspired by process algebras like CSP to freely compose behaviours using operators. CSP does not allow for state variables, but Stateful Timed CSP (STCSP) [23] does. TASTD supports all the time operators of STCSP. TASTD offers a more modular approach to specify actions and attributes than STCSP: variables are global in STCSP, while they are local in an ASTD declaration. On the other hand, CSP and STCSP are well supported by model checking tools like FDR [24] and PAT [25]. Given its rich language, we still have to evaluate how easy it will be to develop model checking tools for TASTD specifications. Using automata to specify the basic behaviour of systems is an advantage over textual notations like CSP and STCSP. RoboSim is a graphical tool for modelling and verifying software simulation of robots. It uses *tock*-CSP [26] as its semantics. *tock*-CSP does not provide quantified operators such as quantified synchronisation, quantified choice or flow, as provided in ASTDs.

Timed automata are graphical notations that offer limited support for specification composition. However, they are more amenable to model checking and well supported by sophisticated tools like UPPAAL [27]. In several works, including [28,29,30], pattern diagrams for timed automata have been proposed to aid in the modelling of high-level system designs. The work in [29] uses patterns based on UML Statecharts, while [30] proposes patterns from time-proven compositional constructs in Timed CSP/TCOZ, and UML activity diagrams are proposed in [28]. Alternatively, TASTD offers those patterns as TASTD types, thus making them algebraic and compositional, and the use of Statecharts-like boxes makes their application modular and transparent. TASTD supports all the basic features of Stateflow [9], and it can simulate all Stateflow operators. It is also efficiently executable like Stateflow models. Stateflow does not support compositional specifications like TASTD does through its algebraic approach. In particular, Stateflow does not support quantified operators like interleaving and choice, which are very useful to model systems where there are an arbitrary number of instances of a given state machine that represent a component. These operators are handy in cybersecurity when modeling attacks that can target, for instance, all the computers of a network. One can model an attack on a machine and quantify over the IP address of the machines to recognise attacks on a whole network all at once [14]. Thanks to shared variables, correlation can be done between attacks spread on several machines, and better top-level decisions can be easily specified [1,14].

The algebraic approach of TASTDs allows for more modularity than in model-based notations like B [19], Event-B [31] and ASM [32]. On the other hand, this syntactic richness entails a more complex semantics, hence the need to have a simple description of it, which is the main objective of this paper. Moreover, these model-based notations offer rich refinement and proof theories, well supported by tools, that TASTD should draw from in the future. Some refinement patterns exist for ASTD [33]. In [34], an approach for proving invariants for a subset of the ASTD notation is defined. Attributes can be defined using the mathematical language of classical B or Event-B, and actions can be written using the rich generalized substitution language of classical B. Proof obligations are generated and can be discharged using Event-B provers. These proof obligations can also be discharged using an Event-B metamodel of the ASTD language introduced in [35] using Rodin [36] and its theory plugin [37,38]. In this paper, we omit the declaration of invariants, as they are properties of a specification, and do not affect its behaviour. A translation from ASTD to B has been proposed in [39,40], but it produces monolithic, complex POs.

4 Conclusion

This paper proposes a simplified description of TASTD, a time extension of the ASTD notation. Its tabular presentation is more concise and easier to understand than its previous version. Non-deterministic choice is used for the choice of an execution order between actions in a flow and a synchronization. TASTD operators are defined in terms of primary ASTDs.

The main advantages of modelling with TASTD in comparison with other comparable methods are the following. The algebraic approach allows for the decomposition of a specification into very small components which are easier to analyse and understand. In particular, the behaviour of an event that affects several components can be separately specified in each component. The synchronisation and flow operators can be used to indicate how these components interact over these events (i.e., hard or soft synchronisation). Communication by shared attributes permits to simplify automata of a specification and reduce the number of automaton states. The graphical nature of TASTD allows for an easier understanding of a specification. Automata and process algebra operators make it easier to understand the ordering relationship between events. TASTD provides a simple, modular approach to deal with time requirements. TASTD, with its compiler cASTD, can generate C++ code that can be deployed into an embedded system. It is also capable of generating code for simulation, in order to check scenarios.

TASTD currently lacks supports for verification. Some work in that direction is in progress [34,35], where ASTD are extended with invariants that can be attached to automaton states and three other ASTD types (sequence, closure and guard), thus allowing to decompose the verification of properties into small parts. We hope that this new approach will help to simplify proof obligations. The refinement of ASTD actions into executable code must also be studied, in conjunction with ASTD refinement and invariant preservation.

References

1. L. Nganyewou Tidjon, M. Frappier, M. Leuschel, A. Mammar, Extended algebraic state-transition diagrams, in: 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS), Melbourne, Australia, 2018, pp. 146–155.
2. C. A. R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677. doi:10.1145/359576.359585.
URL http://doi.acm.org/10.1145/359576.359585

3. B. Roscoe, The theory and practice of concurrency, Prentice Hall, 1998.
4. E. Mikk, Y. Lakhnechi, M. Siegel, Hierarchical automata as model for statecharts, in: R. K. Shyamasundar, K. Ueda (Eds.), Advances in Computing Science — ASIAN'97, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 181–196.
5. D. Latella, I. Majzik, M. Massink, Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker, Formal Aspects Comput. 11 (6) (1999) 637–664. `doi:10.1007/S001659970003`.
URL https://doi.org/10.1007/s001659970003
6. D. Harel, Statecharts: A visual formalism for complex systems, Science of computer programming 8 (3) (1987) 231–274.
7. D. de Azevedo Oliveira, M. Frappier, Tastd: A real-time extension for astd, in: International Conference on Rigorous State-Based Methods, Springer, 2023, pp. 142–159.
8. D. de Azevedo Oliveira, M. Frappier, Technical report 27 - extending astd with real-time, https://github.com/DiegoOliveiraUDES/astd-tech-report-27, [Online; accessed 28-January-2023] (2023).
9. MATLAB, Stateflow, https://www.mathworks.com/products/stateflow.html (2020).
10. R. Alur, D. L. Dill, A theory of timed automata, Theoretical computer science 126 (2) (1994) 183–235.
11. S. Schneider, Concurrent and Real-time systems, John Wiley and Sons, 2000.
12. J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, É. André, Modeling and verifying hierarchical real-time systems using stateful timed CSP, ACM Trans. Softw. Eng. Methodol. 22 (1) (2013) 3:1–3:29. `doi:10.1145/2430536.2430537`.
URL https://doi.org/10.1145/2430536.2430537
13. J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, É. André, Modeling and verifying hierarchical real-time systems using stateful timed csp, ACM Transactions on Software Engineering and Methodology (TOSEM) 22 (1) (2013) 1–29.
14. L. N. Tidjon, M. Frappier, A. Mammar, Intrusion detection using astds, in: International Conference on Advanced Information Networking and Applications, Springer, 2020, pp. 1397–1411.
15. L. N. Tidjon, Formal modeling of intrusion detection systems, Ph.D. thesis, Institut Polytechnique de Paris; Université de Sherbrooke (Québec, Canada), https://theses.hal.science/tel-03137661 (2020).
16. E. J. Chaymae, F. Marc, E. Thibaud, T. Pierre-Martin, Development of monitoring systems for anomaly detection using astd specifications, in: International Symposium on Theoretical Aspects of Software Engineering, Springer, 2022, pp. 274–289.
17. D. de Azevedo Oliveira, M. Frappier, Modelling an automotive software system with TASTD, in: U. Glässer, J. C. Campos, D. Méry, P. A. Palanque (Eds.), Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings, Vol. 14010 of Lecture Notes in Computer Science, Springer, 2023, pp. 124–141. `doi:10.1007/978-3-031-33163-3\_10`.
URL https://doi.org/10.1007/978-3-031-33163-3_10
18. A. R. Ndouna, M. Frappier, Modelling a mechanical lung ventilation system using TASTD, in: S. Bonfanti, A. Gargantini, M. Leuschel, E. Riccobene, P. Scandurra (Eds.), Rigorous State-Based Methods - 10th International Conference, ABZ 2024, Bergamo, Italy, June 25-28, 2024, Proceedings, Vol. 14759 of Lecture Notes in Computer Science, Springer, 2024, pp. 324–340. `doi:10.1007/978-3-031-63790-2\_26`.
URL https://doi.org/10.1007/978-3-031-63790-2_26

19. J.-R. Abrial, The B-book: Assigning Programs to Meanings, Cambridge University Press, New York, NY, USA, 1996.
20. R. Berghammer, Relational specification of data types and programs, Univ. der Bundeswehr München, Fak. für Informatik, 1991.
21. R. C. Backhouse, J. van der Woude, Demonic operators and monotype factors, Math. Struct. Comput. Sci. 3 (4) (1993) 417–433. `doi:10.1017/S096012950000030X`.
URL https://doi.org/10.1017/S096012950000030X
22. J. Abrial, Modeling in Event-B, Cambridge University Press, 2010.
23. M. Balaban, T. Rosen, Stcsp—structured temporal constraint satisfaction problems, Annals of Mathematics and Artificial Intelligence 25 (1999) 35–67.
24. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, Fdr3—a modern refinement checker for csp, in: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20, Springer, 2014, pp. 187–201.
25. J. Sun, Y. Liu, J. S. Dong, J. Pang, Pat: Towards flexible verification under fairness, in: Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09), Vol. 5643 of Lecture Notes in Computer Science, Springer, 2009, pp. 709–714.
26. A. Cavalcanti, A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, J. Timmis, Verified simulation for robotics, Science of Computer Programming 174 (2019) 1–37.
27. G. Behrmann, A. David, K. G. Larsen, A tutorial on uppaal, Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures (2004) 200–236.
28. É. André, C. Choppy, G. Reggio, Activity diagrams patterns for modeling business processes, in: Software Engineering Research, Management and Applications, Springer, 2014, pp. 197–213.
29. A. Mekki, M. Ghazel, A. Toguyeni, Validating time-constrained systems using uml statecharts patterns and timed automata observers, in: Third International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2009) 3, 2009, pp. 1–13.
30. J. S. Dong, P. Hao, S. Qin, J. Sun, W. Yi, Timed automata patterns, IEEE Transactions on Software Engineering 34 (6) (2008) 844–859.
31. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, STTT 12 (6) (2010) 447–466.
32. E. Börger, R. Stärk, Abstract state machines: a method for high-level system design and analysis, Springer Science & Business Media, 2012.
33. M. Frappier, F. Gervais, R. Laleau, J. Milhau, Refinement patterns for astds, Formal Aspects of Computing 26 (2014) 919–941.
34. Q. Cartellier, M. Frappier, A. Mammar, Proving local invariants in astds, in: Y. Li, S. Tahar (Eds.), Formal Methods and Software Engineering - 24th International Conference on Formal Engineering Methods, ICFEM 2023, Brisbane, QLD, Australia, November 21-24, 2023, Proceedings, Vol. 14308 of Lecture Notes in Computer Science, Springer, 2023, pp. 228–246. `doi:10.1007/978-981-99-7584-6\_14`.
URL https://doi.org/10.1007/978-981-99-7584-6_14

35. C. Chen, P. Rivière, N. K. Singh, G. Dupont, Y. Aït-Ameur, M. Frappier, A proof-based ground algebraic meta-model for reasoning on ASTD in event-b, in: 13th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025, IEEE, 2025, pp. 46–57. doi:10.1109/FORMALISE66629.2025.00011.
URL https://doi.org/10.1109/FormaliSE66629.2025.00011
36. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in event-b, Int. J. Softw. Tools Technol. Transf. 12 (6) (2010) 447–466.
37. M. Butler, I. Maamria, Practical theory extension in event-b, in: Z. Liu, J. Woodcock, H. Zhu (Eds.), Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 67–81. doi:10.1007/978-3-642-39698-4_5.
URL https://doi.org/10.1007/978-3-642-39698-4_5
38. T. S. Hoang, L. Voisin, A. Salehi, M. J. Butler, T. Wilkinson, N. Beauger, Theory plug-in for rodin 3.x, CoRR abs/1701.08625 (2017). arXiv:1701.08625.
URL http://arxiv.org/abs/1701.08625
39. T. Fayolle, Combinaison de méthodes formelles pour la spécification de systèmes industriels, Theses, Université Paris-Est ; Université de Sherbrooke (Québec, Canada) (Jun. 2017).
URL https://theses.hal.science/tel-01743832
40. J. Milhau, M. Frappier, F. Gervais, R. Laleau, Systematic translation rules from astd to event-b, in: International Conference on Integrated Formal Methods, Springer, 2010, pp. 245–259.