


TASTD: a real-time extension for ASTD [★]

Diego de Azevedo Oliveira¹  and Marc Frappier¹ 

Université de Sherbrooke, Sherbrooke QC J1K 2R1, Canada

Abstract. In ASTD, real-time models are not natively supported. Real-time requirements are pervasive in many systems, like control systems and cybersecurity. Timed Algebraic State Transition Diagrams (TASTD) is an extension of ASTD capable of specifying real-time models. TASTD gives ASTD the capability to handle time with new algebraic operators. This paper describes the syntax and semantics of these new time operators: delay, persistent delay, timeout, persistent timeout, and timed interrupt. These new time operators are specified using two new operators, persistent guard and interrupt. To illustrate our extension, we present a small case study of a sensor where we want to detect potential anomalies.

Keywords: ASTD · real-time model · formal methods · TASTD

1 Introduction

ASTD [22] is a graphical notation that combines process algebra operators and hierarchical state machines. It is particularly well-suited for specifying monitoring systems, like intrusion detection systems and control systems. It has been successfully applied in case studies for intrusion detection [26,27,13] and control systems [5]. ASTD allows for combining state transition diagrams, such as statecharts and automata, and process algebra operators, inspired by CSP. Hence, ASTD takes advantage of the strength of both notations: graphical representation, hierarchy, orthogonality, compositionality, and abstraction.

Real-world specifications frequently depend on quantitative timing. Time extensions have been proposed for several well-known languages like statecharts, with MATLAB Stateflow [19], automata, with timed automata [3], and process algebra, with Timed CSP [23].

Each of these methods has its strengths and weaknesses. Timed automata are efficient for model checking but it remains a challenge to refine them into an implementation [28]. Statecharts offer an explicit representation of the control flow and support a rich notation for data modelling. However, it does not offer the abstraction power of process algebra operators [27]. Process algebras support refinement and are also effective for model checking, but lack language features (e.g., shared variables) [24] and graphical representation. *tock*-CSP [12] is an

[★] Supported by Public Safety Canada’s Cyber Security Cooperation Program (CSCP) and NSERC (Natural Sciences and Engineering Research Council of Canada).

extension of CSP with the definition of a special event named *tock*, which marks the passage of time, but CSP does not have a graphical representation. Our notation tries to overcome these weaknesses and combine some of their strengths.

This paper presents Timed ASTD (TASTD), a real-time extension for ASTD. TASTD includes proposes five new operators to deal with timing constraints: delay, persistent delay, timeout, persistent timeout, and timed interrupt. These new time operators are defined using two new operators not specific to time, interrupt, and persistent guard.

This article is structured as follows: Section 2 briefly presents TASTD. Section 3 illustrates TASTD using a case study that shows the usefulness of TASTD and new operators. Section 4 presents the modifications to the operational semantics of ASTD to deal with time and the new operators. In Section 5, we discuss the tool support for TASTD. Sections 6 discusses related work. Section 7 concludes the paper.

2 An overview of TASTD

In this section, we informally introduce ASTD and its extension, TASTD, and illustrate it with an example. ASTD draws from the statecharts notation the following concepts: hierarchy, OR-states, AND-states, guards, and history states. However, it does not support broadcast communication, or null transitions [22]. Automata constitute the basis for ASTD construction: an elementary ASTD is an automaton. The process algebra operators that may be used to combine elementary ASTDs are sequence, guard, choice, Kleene closure, parallel composition, quantified choice, and quantified parallel composition. Automaton states can be complex ASTDs defined by any of these operators. ASTDs also support state variables, called *attributes*, and actions on transitions, states, and ASTD themselves that can update these attributes and execute arbitrary C++ code.

2.1 Transition System

A TASTD state includes a timestamp that denotes the time at which the state was reached. TASTDs 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 timing constraints and simulate clocks. In ASTD, a transition can be triggered only by external events. In TASTD, we introduce transitions that can be triggered by the passage of time; such a transition is labelled by a special event called **Step**. Automaton transitions can be labelled with **Step**, and some time operators can also execute a **Step** transition. The enabledness of a time-triggered transition is checked on a periodical basis, according to the desired time granularity required to match system timing constraints.

The semantics of TASTDs consists of a labelled transition system (LTS) \mathcal{S} , which is a subset of

$$(\text{State} \times \mathcal{T} \times W) \times \text{Event} \times (\text{State} \times \mathcal{T} \times W)$$

where **State** is the set that contain all states, \mathcal{T} is the set of clock values, and W is the set of all possible attributes and control values. The LTS system represent a set of transitions of the form $(s, t, w) \xrightarrow{\sigma} (s', t', w')$. Such a transition means that a TASTD can execute event σ from state s and move to state s' . Symbols w, w' respectively denote the values of the attributes of ASTD a , which can be modified during execution. Symbols t, t' respectively denote the time at which states s, s' were reached; hence, they denote the timestamp of the *latest executed event*. The value of t for the system's initial state is some timestamp, which represents the time the system starts. The timestamp t of a state s is needed when deciding on various timing operators. For instance, a timeout is evaluated concerning the latest event executed. TASTD timing operators simulate clocks, relying on the time of the latest event executed for that purpose. Thus, when using a TASTD operator, there is no need to define a clock to specify timing constraints. However, clocks can be declared as TASTD attributes and used to specify arbitrary timing constraints. A state s contains attribute and control values that represent the behaviour of various ASTD operators.

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 e of the form $s \xrightarrow{e} s'$ and transitions on the passage of time with d units $s \xrightarrow{d} s'$, which are more suitable for model-checking. A TASTD transition $(s, t, w) \xrightarrow{e} (s', t', w')$ corresponds to two successive transitions $s \xrightarrow{t'-t} s$ and $s \xrightarrow{e} s'$ in timed automata.

ASTD Operators ASTD operators are automaton, sequence, choice, guard, Kleene closure, flow, parameterised synchronisation, call, and quantified versions of parameterised synchronisation and choice, with the usual process algebra semantics of these operators, adapted to deal with automaton as elementary ASTD. Each ASTD has a function that determines its initial state and a function that determines its final states. A sequence of A_1 and A_2 allows to execute A_1 first; when A_1 is in a final state, A_2 can start its execution. A choice between A_1 and 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 guard P on an ASTD A checks that the condition P is satisfied on the execution of the first event of A ; the guard is ignored after the first event. A Kleene closure on an ASTD A allows for looping on A : it executes A and can restart A from its initial state when A is in a final state. The flow operator Ψ is inspired by the AND state of Statecharts, which executes an event on each sub-ASTD whenever possible. A flow $E_1 \Psi E_2$ can execute an event e iff either E_1 , or E_2 , or both E_1 and E_2 can execute it. The parameterised synchronisation operator $[[\Delta]]$, drawn from CSP, executes two sub-ASTDs in parallel, and they must synchronize on a set of events Δ . If Δ is empty, then the parameterised synchronisation is an interleave, denoted by $|||$. We can draw an analogy between these three operators and Boolean operators. Operator $|||$ acts like a conjunction: $E_1 ||| \{e\} ||| E_2$ can execute an event e iff both E_1 and E_2 can execute it. It expresses a conjunction of ordering constraints on e given by E_1 and E_2 . It is a

hard synchronisation. Operator Ψ acts like an inclusive OR: $E_1 \Psi E_2$ can execute an event e 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 e on either E_1 or E_2 , but on only one of them; if both E_1 and E_2 can execute e , then one of them is chosen nondeterministically.

2.2 TASTD Operators

TASTD introduces five operators to deal with timing constraints: delay, persistent delay, timeout, persistent timeout, and timed interrupt. These time operators are defined using two operators not specific to time, interrupt, and persistent guard. These five time operators are defined in terms of **Step** transitions and two new ASTD operators, persistent guard and interrupt.

A delay d on ASTD A will wait d units of time before accepting the first event of A ; the subsequent events are not subject to the delay. A persistent delay will apply the delay d to each event of A . A timeout is a binary operator with a duration d as parameter; the first ASTD A_1 must execute its first event within d units of time; if no event is executed on A_1 within d units, then A_1 is disabled, and A_2 takes over and executes the subsequent events; if the first event is executed with d , then A_2 is disabled, and A_1 continues the execution. A persistent timeout will apply d to each event of A_1 ; if d is missed, then A_2 takes over, and A_1 is disabled. An interrupt is a binary operator; the second ASTD A_2 has priority on the first ASTD A_1 ; A_1 is executed first, but any event that can be executed on A_2 will interrupt A_1 and disable it. A persistent guard P on an ASTD A checks that the condition P is satisfied on the execution of each event of A .

3 Illustrative Example

In this section, we illustrate TASTD operators with a small case study. The case study consists in a sensor that receives data at a regular interval of 5 seconds. With this case study, we model two potential anomalies. 1) Missing data: when the receiver does not receive data in a 5 seconds interval; it may indicate that an attacker is deleting data. 2) Data overflow: when the receiver receives data in less than 5 seconds from the latest signal; it may indicate that an attacker is injecting data. Figure 1 shows the possible traces for each behaviour. The act of receiving data is labelled by event e .

Figure 2(a) shows the TASTD that models missing data detection. It is a closure ASTD A that loops over a persistent timeout ASTD B . The first ASTD of B is a simple automaton ASTD C that receives sensor events e . A timeout on ASTD C is executed when an event e is not received within $avg + 3 * stddev$ (e.g., 5.5 seconds, the timeout duration), where avg and $stddev$ are the average and standard deviation time between two signals in the regular operation of the sensor. Execution is then transferred to ASTD D , which is an automaton that does nothing; its initial state is also final, which allows the closure A to restart

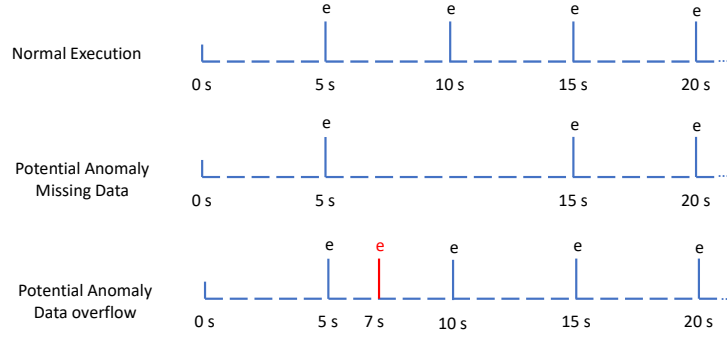


Fig. 1: Sensor traces

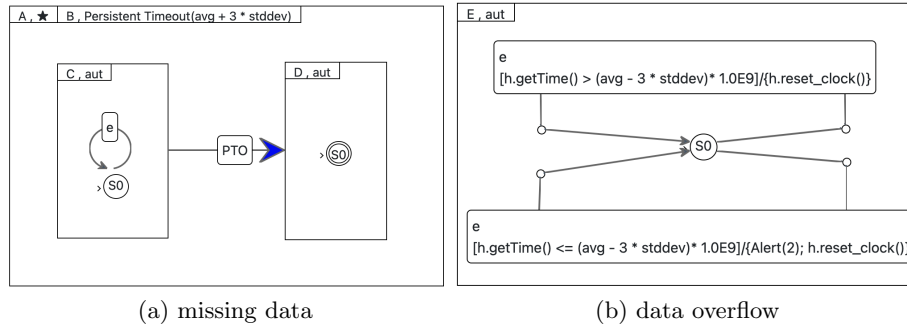


Fig. 2: TASTDs A and E

the persistent timeout B. In this simplified version, nothing else has to be done in D when missing data is detected. In case of a timeout on C, an alert signal is emitted by the action that can be specified on a timeout transition of ASTD B (labelled here by PTO).

Figure 2(b) shows the TASTD that models the detection of data overflow. TASTD E has a clock variable *h* that starts when the initial state is reached. When event *e* occurs before or at $(avg - 3 * stddev)$ (e.g., 4.5 seconds, before the expected of 5 seconds and a safe range), the transition emits an alert and resets the clock *h*. When event *e* occurs after $(avg - 3 * stddev)$, the transition resets the clock *h*.

Each of these ASTD models a potential anomaly behaviour and emits an alert when detected. However, these models must run simultaneously to detect both anomalies in a sensor trace. We achieve this parallelism using the flow ASTD F, which calls ASTDs A and E, represented in Figure 3(a).

We can generalise this model by using a learning phase that will compute *avg* and *std* over a certain learning period and then start identifying anomalies using the learned values of *avg* and *std*. These attributes *avg* and *std* are declared in

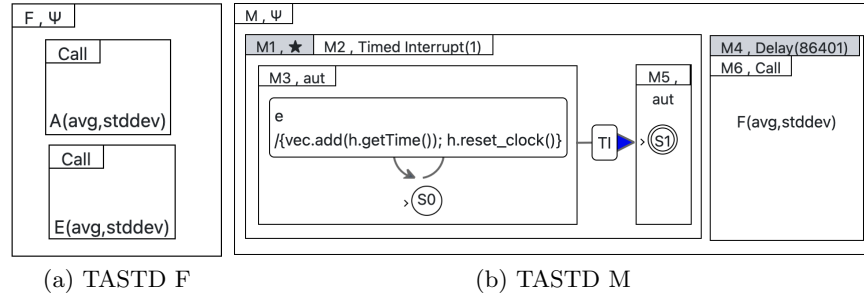


Fig. 3: TASTDs F and M

the main ASTD M We model a receiver that receives data and computes *avg* and *std* at the end of each day. This model is shown in 3(b). ASTD M is a flow between ASTD M1, which calculates *avg* and *std*, and M5, which models the detection. TASTD M2 is a timed interrupt that, at the end of each day, allows M4 to execute and update the values of *avg* and *std* which are used in F. ASTD M3 stores in a vector *vec* the interval between two successive events *e*. TASTD M5 is a delay that delays the detection for one day and one second to allow learning the initial value of *avg* and *std*. After the first day of this model execution, the detection is started. At the end of each day, the value of *avg* and *std* is recalculated with M2.

All these models are found in [8]. A more comprehensive TASTD example, available at [7], models an adaptive exterior light and speed control system of a vehicle, which Mercedes submitted for the ABZ2020 conference case study track. The complete model is composed of 66 automata, 1 closure, 26 synchronisations, 14 flows, 33 calls, 1 persistent guard, 7 persistent delays, and 2 delays, for a total of 150 ASTDs. It can be compiled using the cASTD compiler and translated into C++ for execution in simulation mode or as an actual implementation of the ASTD specification. Figures 2 and 3 were produced from the eASTD editor. The compiler and the editor can be found at [16].

4 TASTD Semantics

In this section, we introduce the modifications made to adapt the operational semantics of ASTDs to deal with time. This semantics is defined using inference rules in the Plotkin style. The complete semantics of TASTD can be found in [6].

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 ASTDs. In the operational semantics, we handle these variables using *environments*. An environment is a function of $\text{Env} \triangleq \text{Var} \rightarrow \text{Term}$ which associates values with variables. The operational semantics of TASTD is defined using an auxiliary transition relation \mathcal{S}_a that deals with environments. A transition of \mathcal{S}_a has the following form:

$$s \xrightarrow{\sigma, t, E_e, E'_e}_a s'$$

where σ is the event executed, E_e, E'_e denote the before and after values of identifiers in the TASTDs enclosing TASTD a . These identifiers could be TASTD attributes, quantified variables introduced by quantified operators like choice, synchronisation, interleave, and attributes. For the main ASTD of a specification, E_e, E'_e denote the attributes values used to call the main ASTD. The time of the latest executed event is denoted by timestamp t .

Recall that a state of the system (given by the main ASTD of the specification) is a triple (s, t, w) . The initial state of the system is

$$(init(Main, cst, P := V), cst, V)$$

Function *init* describes the initial state of an ASTD; it is inductively defined on the ASTD types; it receives the initialisation time of an ASTD and the values V of the main ASTD parameters P . The initialisation time is used in ASTD types that encompass a notion of parallelism, that is, flow and parameterised synchronisation, because each of their sub-ASTDs has its own latest event execution time to determine timing constraints relative to that component of the parallel composition, since the execution of the two sub-ASTDs are independent. A timed interrupt also needs this initialisation time to store it in its initial state. That will be further explained when these ASTD types are defined in the sequel. The timestamp of the latest event executed is stored at the top-level state and initialised with the current system time *cst*.

The following top transition rule, connects \mathcal{S} to \mathcal{S}_a :

$$\text{env} \frac{s \xrightarrow{\sigma, t, P:=V, P:=V'}_a s'}{(s, t, V) \xrightarrow{\sigma}_a (s', cst, V')}$$

It states that a transition is proved starting with environments providing the initial values V of the top-level ASTD parameters P , and their final values V' , since these parameters can be modified during execution. Current system time *cst* is stored in the global state as the new latest event execution timestamp.

TASTDs are *non-deterministic*: If several transitions on σ are possible from a given state s , then one is non-deterministically chosen. The operational semantics is inductively defined on \mathcal{S}_a for each ASTD type. However, in this paper, we content ourselves with the definition of the modifications to some ASTD types, for illustrative purposes, and the definition of the new types introduced for TASTD. The complete definition of TASTD is provided here [6].

Sequence The sequence ASTD type has the following structure:

$$\text{Sequence} \triangleq \langle \hookrightarrow, fst, snd \rangle$$

where *fst*, *snd* are ASTDs denoting the first and second sub-ASTDs of the sequence, respectively. A sequence state is of type $\langle \hookrightarrow, E, [fst \mid snd], s \rangle$, where \hookrightarrow .

is a constructor of the sequence state, E the values of attributes declared in the sequence, $[\text{fst} \mid \text{snd}]$ is a choice between two markers that respectively indicate whether the sequence is executing the first sub-ASTD or the second sub-ASTD, and s is the state of that sub-ASTD. The initial and final states for a sequence are defined as follows. Let a be a sequence ASTD.

$$\begin{aligned} \text{init}(a, ts, G) &\triangleq (\hookrightarrow_{\circ}, a.E_{\text{init}}(\llbracket G \rrbracket), \text{fst}, \text{init}(a.fst, ts, G \Leftarrow a.E_{\text{init}})) \\ \text{final}(a, (\hookrightarrow_{\circ}, E, \text{fst}, s)) &\triangleq \text{final}(a.fst, s) \wedge \text{final}(a.snd, \text{init}(a.snd, \perp, E)) \\ \text{final}(a, (\hookrightarrow_{\circ}, E, \text{snd}, s)) &\triangleq \text{final}(a.snd, s) \end{aligned}$$

We denote by $u(\llbracket G \rrbracket)$ the application of the environment G as a substitution that replaces environment variables occurring in u by their values given in G . When the first ASTD is being executed, a sequence is in a final state if the current state of the first ASTD is final and if the initial state of the second ASTD is final. The timestamp of the initial state of the second ASTD is not determinant to define if it is final, so it can assume any valid value and is represented with an underscore. When the second ASTD is being executed, a sequence is in a final state if the current state of the second ASTD is final.

Three rules are necessary to define the possible transitions of a sequence ASTD. Rule \hookrightarrow_1 deals with transitions occurring in the first sub-ASTD. Rule \hookrightarrow_2 deals with the transitions that start the second ASTD when the first is in a final state. Rule \hookrightarrow_3 deals with transitions occurring in the second sub-ASTD.

$$\begin{aligned} \hookrightarrow_1 &\frac{s \xrightarrow{\sigma, t, E_g, E_g''} a.fst s' \quad \Theta}{(\hookrightarrow_{\circ}, E, \text{fst}, s) \xrightarrow{\sigma, t, E_e, E_e'} a (\hookrightarrow_{\circ}, E', \text{fst}, s')} \\ \hookrightarrow_2 &\frac{\text{final}(a.fst, s) \quad \text{init}(a.snd, t, E_e) \xrightarrow{\sigma, t, E_g, E_g''} a.snd s' \quad \Theta}{(\hookrightarrow_{\circ}, E, \text{fst}, s) \xrightarrow{\sigma, t, E_e, E_e'} a (\hookrightarrow_{\circ}, E', \text{snd}, s')} \\ \hookrightarrow_3 &\frac{s \xrightarrow{\sigma, t, E_g, E_g''} a.snd s' \quad \Theta}{(\hookrightarrow_{\circ}, E, \text{snd}, s) \xrightarrow{\sigma, t, E_e, E_e'} a (\hookrightarrow_{\circ}, E', \text{snd}, s')} \end{aligned}$$

Predicate Θ used in the premises of these inference rules determines the update of the various environments used in a rule. It is omitted here for the sake of concision. It indicates in which order the various actions of an ASTD are executed and deals with variable shadowing between embedded ASTDs. To summarise, actions can be declared at various places in an ASTD specification, to maximise modularity and avoid repetition of actions. Actions are executed in a bottom-up manner, starting from the automaton transitions. The exit action of the source state of an automaton transition is executed first, followed by the transition action, the entry action of the destination state of the transition, the action declared on the automaton itself, and then all of its enclosing ASTDs, up to the root (main) ASTD.

Parameterised Synchronisation The parameterised synchronisation ASTD subtype has the following structure:

$$\text{Synchronisation} \triangleq \langle |||, \Delta, l, r \rangle$$

where Δ is the synchronisation set of event labels, and $l, r \in \text{ASTD}$ are the synchronised ASTDs. When the label of the event to execute belongs to Δ , the two sub-ASTDs must both execute it; otherwise either the left or the right sub-ASTD can execute it; if both sub-ASTDs can execute it, the choice between them is nondeterministic. When $\Delta = \emptyset$, the synchronisation is called an interleaving and is abbreviated as $|||$.

A parameterised synchronisation state is of type $\langle |||_o, E, s_l, c_l, s_r, c_r \rangle$, where s_l, s_r are the states of the left and right sub-ASTDs and $c_l, c_r \in C$ are the timestamp of the latest executed event on the left and right sub-ASTDs. These timestamps are updated when their respective sub-ASTD is executed. Initial and final states are defined as follows. Let a be a parameterised synchronised ASTD.

$$\begin{aligned} \text{init}(a, ts, G) &\triangleq (|||_o, a.E_{\text{init}}(G), \text{init}(a.l, ts, G \Leftarrow a.E_{\text{init}}), ts, \\ &\quad \text{init}(a.r, ts, G \Leftarrow a.E_{\text{init}}), ts) \\ \text{final}(a, (|||_o, E, s_l, t_l, s_r, t_r)) &\triangleq \text{final}(a.l, s_l) \wedge \text{final}(a.r, s_r) \end{aligned}$$

The initial state of a parameterised synchronisation initialises both of its sub-ASTDs with the timestamp received as parameter. A parameterised synchronisation is final when both of its sub-ASTDs are final.

We define the semantics of a parameterised synchronisation with three rules. Rules $|||_1$ and $|||_2$ respectively describe execution of events, with no synchronisation required, either on the left or the right sub-ASTDs. Rule $|||_1$ below caters for execution on the left sub-ASTD. The function $\alpha(e)$ returns the label of event e .

$$|||_1 \frac{\alpha(\sigma) \notin \Delta \quad s_l \xrightarrow{\sigma, t_l, E_g, E_g''}_{a.l} s'_l \quad \Theta}{(|||_o, E, s_l, t_l, s_r, t_r) \xrightarrow{\sigma, t, E_e, E_e'}_a (|||_o, E', s'_l, \text{cst}, s_r, t_r)}$$

It is important to notice that only the left clock is updated with $|||_1$. Rule $|||_2$ is symmetric to $|||_1$ and indicates the behaviour when the right side execute the action. In the case of a synchronisation, the timestamps of both sub-ASTDs are updated with cst . The rule for synchronisation is omitted for the sake of concision.

Persistent Guard Persistent Guard ASTD is a new operator from the TASTD extension that guards the execution of *each* event execution of its sub-ASTD, whereas a “regular” guard only affects the first event, as in CSP. The persistent guard ASTD type has the following structure:

$$\text{PGuard} \triangleq \langle \Rightarrow_p, g, b \rangle$$

where ASTD b is the body of the persistent guard, and g is the guard condition. The type of a persistent guard state is $\langle \Rightarrow_p, E, s \rangle$ where s is the state of b and E the attribute values of the persistent guard ASTD. Initial and final states are defined as follows. Let a be a persistent guard ASTD.

$$\begin{aligned} \text{init}(a, ts, G) &\triangleq (\Rightarrow_p, a.E_{\text{init}}([G]), \text{init}(a.b, ts, G \Leftarrow a.E_{\text{init}})) \\ \text{final}(a, (\Rightarrow_p, E, s)) &\triangleq \text{final}(a, s) \end{aligned}$$

The initial and final states of a persistent guard ASTD are straightforward. A guard is in a final state if the persistent guard's body is final.

Persistent guard is defined using a single inference rule \Rightarrow_{p1} that executes any transition from b if the guard predicate g holds in the current environment E_e .

$$\Rightarrow_{p1} \frac{g([E_e]) \quad s \xrightarrow{\sigma, t, E_g, E'_g} a.b \ s' \quad \Theta}{(\Rightarrow_p, E, s) \xrightarrow{\sigma, t, E_e, E'_e} (\Rightarrow_p, E', s')}$$

Interruption Interruption ASTD is a new operator from the TASTD extension with the following structure

$$\text{Interrupt} \triangleq \langle \text{Intpt}, fst, snd, A_{Int} \rangle$$

where fst, snd are the sub-ASTDs and A_{Int} is an action executed when the interruption occurs. The second ASTD snd has priority on the first ASTD fst and can interrupt it at any point. An interruption state is of type $\langle \text{Intpt}_o, E, [\text{fst}|snd], s \rangle$, where $[\text{fst}|snd]$ is a choice between the two markers that indicate which ASTD is being executed, and s is the state of that ASTD. Its initial state and final state are defined as follows

$$\begin{aligned} \text{init}(a, ts, G) &\triangleq (\text{Intpt}_o, a.E_{\text{init}}([G]), \text{fst}, \text{init}(a.fst, ts, G \Leftarrow a.E_{\text{init}})) \\ \text{final}(a, (\text{Intpt}_o, E, \text{fst}, s)) &\triangleq \text{final}(a.fst, s) \\ \text{final}(a, (\text{Intpt}_o, E, \text{snd}, s)) &\triangleq \text{final}(a.snd, s) \end{aligned}$$

We define the semantics of interruption execution with three rules. Intpt_1 allows for the execution of the first sub-ASTD. Intpt_2 allows for the interruption execution when the event σ from the second astd happen. Intpt_3 allows for the execution of the second sub-ASTD after the interruption.

$$\begin{aligned} \text{Intpt}_1 &\frac{s \xrightarrow{\sigma, t, E_g, E'_g} a.fst \ s' \quad \Theta}{(\text{Intpt}_o, E, \text{fst}, s) \xrightarrow{\sigma, t, E_e, E'_e} a (\text{Intpt}_o, E', \text{fst}, s')} \\ \text{Intpt}_2 &\frac{\Omega_{\text{Interrupt}} \quad \text{init}(a.snd, t, E_e) \xrightarrow{\sigma, t, E'_g, E''_g} a.snd \ s' \quad \Theta}{(\text{Intpt}_o, E, \text{fst}, s) \xrightarrow{\sigma, t, E_e, E'_e} a (\text{Intpt}_o, E', \text{snd}, s')} \\ \Omega_{\text{Interrupt}} &\triangleq a.A_{Int}(E_g, E'''_g) \end{aligned}$$

$$\text{Intpt}_3 \frac{s \xrightarrow{\sigma, t, E_g, E_g''}_{a.snd} s' \quad \Theta}{(\text{Intpt}_o, E, \text{snd}, s) \xrightarrow{\sigma, t, E_e, E_e'}_a (\text{Intpt}_o, E', \text{snd}, s')}$$

The predicate $\Omega_{\text{Interrupt}}$ used in the premise of the second inference rule determines that the update of different environments must take into account the interrupt action prior to any changes related to the event σ .

Delay The Delay TASTD type has the following structure:

$$\text{Delay} \triangleq \langle \text{Delay}, b, d \rangle$$

where ASTD b is the body of the delay, and d is the delay value in time units. Initial and final states are defined as follows.

$$\begin{aligned} \text{init}(a, ts, G) &\triangleq (\text{Delay}_o, a.E_{\text{init}}([G]), \text{false}, \\ &\quad \text{init}(a.b, ts, G \Leftarrow a.E_{\text{init}})) \\ \text{final}(a, (\text{Delay}_p, E, \text{started?}, s)) &\triangleq \text{final}(a.b, s) \end{aligned}$$

There are two inference rules for Delay: Delay_1 allows for the transition on its body b after idling for at least d time step on the initial state. Delay_2 allows for the execution after the first event.

$$\begin{aligned} \text{Delay}_1 &\frac{\text{cst} - t > d \quad \text{init}(a.b, t, E_e) \xrightarrow{\sigma, t, E_g, E_g''}_{a.b} s \quad \Theta}{(\text{Delay}_o, E, \text{false}, \text{init}(a.b, t, E)) \xrightarrow{\sigma, t, E_e, E_e'}_a (\text{Delay}_o, E', \text{true}, s)} \\ \text{Delay}_2 &\frac{s \xrightarrow{\sigma, t, E_g, E_g''}_{a.b} s' \quad \Theta}{(\text{Delay}_o, E, \text{true}, s) \xrightarrow{\sigma, t, E_e, E_e'}_a (\text{Delay}_o, E', \text{true}, s')} \end{aligned}$$

The condition $\text{cst} - t > d$ in rule Delay_1 states that the first event of the delay body can be executed iff the difference between the current system time (cst) and the timestamp of the latest executed event (t) is greater than d . Recall that t is stored in the top level state and passed on to the proof rules of the ASTDs using rule env . If a delay occurs as the second operand of a sequence, then the delay will start from the timestamp of the latest executed event of the first ASTD of the sequence, according to rule \Rightarrow_2 .

A delay TASTD can also be defined using a combination of ordinary ASTDs, as illustrated in Figure 4.

The cASTD compiler uses these equivalences to implement the delay operator. All the other TASTD operators are also expressed in terms of existing ASTD operators. In the sequel, we define the TASTD operators using these equivalences and omit the inference rules.



Fig. 4: Equivalence between delay TASTD and a guard ASTD

Persistent Delay The persistent delay TASTD type has the following structure:

$$\text{PDelay} \triangleq \langle \text{Delay}_p, b, d \rangle$$

where b is the ASTD body of the delay, and d is the delay value in time units. The persistent delay is defined using a persistent guard as illustrated in Figure 5. The persistent guard ensures that each event is delayed.



Fig. 5: Equivalence between a persistent delay and a persistent guard

Timeout The timeout TASTD type has the following structure:

$$\text{Timeout} \triangleq \langle \text{Timeout}, fst, snd, d, A_{TO} \rangle$$

where fst denotes the ASTD that is executed if its first event occurs within d time units; ASTD snd takes over if not. Action A_{TO} is executed when the timeout occurs. A timeout ASTD $\langle \text{Timeout}, A, B, d, A_{to} \rangle$ is implemented using the equivalence illustrated in Figure 6. It uses an interrupt C , which declares a Boolean b initialised to FALSE. If ASTD A can execute its first event within d units of time, then b is set to TRUE by the ASTD action of A , which has been suffixed with the assignment $b := \text{TRUE}$. ASTD A is interrupted by B if b is still FALSE and the condition $\text{cst} - t > d$ holds (the timeout time has been reached), and either B is capable of executing an event, or the **Step** event is executed. ASTD D is a choice (represented by the CSP operator \square) between an automaton E and ASTD B . ASTD E can execute a **Step** event and then enter the complex automaton state defined with B . This choice is needed because two things can occur to trigger a timeout. Recall that **Step** is tested for execution at some regular interval defined by the specifier. The timeout time can be reached within two successive **Step** events. In that case, if B can execute an event e that occurs between these two **Step** events and after the timeout time, then this will trigger the timeout transition from A to B . If e does not occur, then the next

Step event following the timeout time is executed, and the ASTD moves to state B, and it resumes execution from there. ASTD A is disabled in any case.

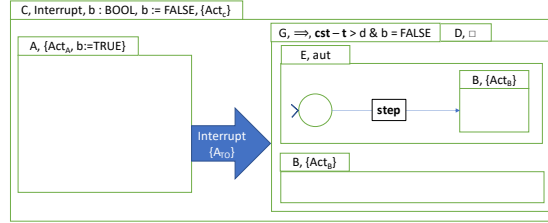


Fig. 6: A combination of ASTDs implementing a timeout

Persistent Timeout The persistent timeout TASTD type has the following structure:

$$\text{PTimeout} \triangleq \langle \text{PTimeout}, fst, snd, d, A_{PTO} \rangle$$

Its distinction with the timeout ASTD is that the execution of each event of *fst* is subject to the timeout *d*. Similarly to timeout, it is implemented with a guard, but no Boolean *b* is needed, since each event executed by *A* is subject to a timeout; it is illustrated in Figure 7

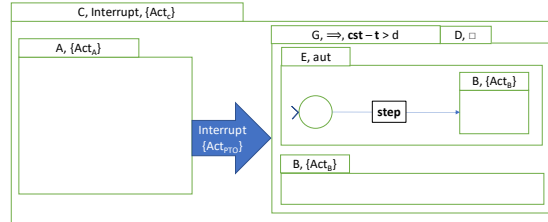


Fig. 7: A combination of ASTDs implementing a persistent timeout

Timed Interrupt The timed interrupt TASTD subtype has the following structure:

$$\text{TInterrupt} \triangleq \langle \text{TInterrupt}, fst, snd, d, A_{TI} \rangle$$

where *fst* is the ASTD whose execution is interrupted by *snd* after *d* units of time. Action *A_{TI}* is executed when the interruption is triggered. It is implemented using a composition similar to persistent timeout, in Figure 8. The only

difference is that the interrupt C stores the latest event execution time t upon its initialisation in a local variable ts , and the current system time is compared with ts , in order to determine how much has elapsed since the start of the ASTD. This is another reason that requires passing t as a parameter to the initialisation function of the ASTDs.

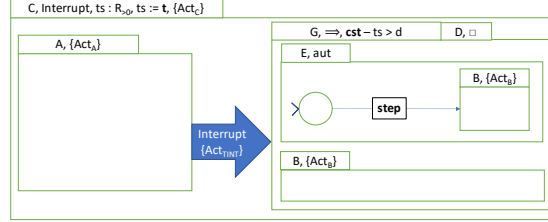


Fig. 8: A combination of ASTDs implementing a timed interrupt

5 Tool support

TASTD uses the same tools as ASTD, which consists of the editor eASTD and the compiler cASTD.

eASTD is a graphical editor for ASTD and TASTD. It supports editing and verification of the well-formedness of TASTD specifications.

cASTD is used for automatic code generation. It can be used through the editor as a plugin, which allows compilation and code production from eASTD interface. cASTD has options for code optimization, parameters definition, and execution of the TASTD in simulation mode. The simulation mode allows one to control the value of cst . As default, cASTD produces executable C++ code that can be deployed in a system and uses the system clock.

6 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 behaviors using operators. CSP does not allow for state variables, but stateful timed CSP (STCSP) [9] does. TASTD supports all the timing operators of STCSP. TASTD offers a more modular approach to specify actions and attributes than STCSP. On the other hand, CSP and STCSP are well supported by model checking tools like FDR [18] 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 behavior 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 [12] as its semantics. *tock*-CSP does not provide quantified operators such as quantified synchronisation, quantified choice or flow, present in ASTDs. Timed automata are graphical notations that offer limited support for specification composition, like in process algebra. However, they are more amenable to model checking and well supported by sophisticated tools like UPPAAL [10]. In several works, including [4,20,14], pattern diagrams for timed automata have been proposed to aid in the modelling of high-level system designs. [20] uses patterns based on UML Statecharts, [14] proposes patterns from time-proven compositional constructs in Timed CSP/TCOZ, and [4] uses UML activity diagrams. Alternatively, TASTD offers those patterns as TASTD types, thus making them algebraic and compositional, and the use of statechart-like boxes makes their application modular and transparent. TASTD supports all the basic features of Stateflow [19], 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 represents 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 [26]. Thanks to shared variables, correlation can be done between attacks spread on several machines, and better top-level decisions can be easily specified [26,22].

The graphical and algebraic approach of TASTDs allows for more modularity than in model-based notations like B [1], Event-B [2] and ASM [11]. However, these 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 [15,17].

7 Conclusion

This paper proposes a real-time extension for the the ASTD notation and seven new operators for ASTD notation: Two for ASTD and five for TASTD, which are defined as a combination of ASTD operators. ASTD was designed to provide a rich, abstract, compositional modeling notation that can be used to specify control systems and monitoring systems and generate their implementation automatically. Several case studies [26,27,13] have shown that it is well-suited to model cybersecurity attacks and control systems. TASTD is supported by a graphical editor and a compiler.

In [7,5], an automotive vehicle's adaptive light and cruise control system is modelled with TASTD. It is found that the algebraic approach allowed for the decomposition of a specification into smaller components that were easy to analyze and understand. The behavior of an event that affected several components

could be separately specified in each component. The synchronisation and flow operators could be used to indicate how these components interacted over such events, either through hard or soft synchronisation. Communication via shared attributes simplified the automata of a specification and reduced the number of automaton states. The graphical nature of TASTD facilitated the understanding of a specification, where automata and process algebra operators made it simple to understand the ordering relationship between events. Additionally, TASTD provided a simple, modular approach to deal with timing requirements, and its compiler cASTD could generate C++ code that could be deployed into an embedded system. It is also capable of generating code for simulation, which is helpful in simulation and system validation.

Our future work will address the formal verification of TASTD specifications. We are currently working on proving invariants in ASTD specifications. A new tool called **pASTD** is under development; it will offer the possibility to specify TASTD attributes and actions using the Event-B language and generate proof obligations for invariants declared on automata states and TASTDs. This will hopefully allow for decomposing the proof of invariants into smaller parts that will be easier to prove, compared to model-based notations like B, Event-B, and ASM. These proof obligations are represented as theorems of a synthetic Event-B context that can be proved using the Rodin platform. Such an (Event-)B-annotated ASTD specification could then be refined into an implementation by transforming actions into B0 actions, proving their refinement, and translating them into C using the Atelier B tools. A translation from ASTD to B was initially proposed in [21]. However, ASTD has evolved with more operators, attributes, and actions. This translation must be reviewed and updated to handle TASTDs correctly. Moreover, the proof obligations generated from the B translation were complex and hard to discharge.

It will also be interesting to address the proof of properties that involves the clocks of a specification and to prove invariants related to these clocks or temporal properties.

On a more practical side, we have noticed in our modeling experiments that it is often desirable to allow the specifier to order the execution of some binary operators. When dealing with nondeterministic specifications in a choice or an interleave, it is handy to indicate which side should be tested first for execution. Parallelism operators, synchronisation and flow, could also be ordered, so this order can be used to update shared variables in a specific order. Associative and commutative binary operators could be extended to become n -ary. This would avoid the creation of superfluous intermediate binary operators. For instance, an interleave $E_1 \parallel E_2 \parallel E_3$ is represented by 5 ASTDs $(E_{123}, E_{12}, E_1, E_2, E_3)$, because E_{12} represents the interleave ASTD composing E_1 and E_2 , and E_{123} composing E_{12} with E_3 . n -ary operators would also allow us to generate more compact C++ code.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *STTT* **12**(6), 447–466 (2010)
3. Alur, R., Dill, D.L.: A theory of timed automata. *Theoretical computer science* **126**(2), 183–235 (1994)
4. André, É., Choppy, C., Reggio, G.: Activity diagrams patterns for modeling business processes. In: *Software Engineering Research, Management and Applications*. pp. 197–213. Springer (2014)
5. de Azevedo Oliveira, D., Frappier, M.: Modelling an automotive software system with TASTD. <https://github.com/DiegoOliveiraUDES/casestudyABZ2020-tastdmodel> (2023)
6. de Azevedo Oliveira, D., Frappier, M.: Technical report 27 - extending ASTD with real-time. <https://github.com/DiegoOliveiraUDES/astd-tech-report-27> (2023), [Online; accessed 28-January-2023]
7. Diego de Azevedo Oliveira, M.F.: Modelling an automotive software system with tastd. In: *International Conference on Rigorous State-Based Methods*. LNCS, vol. xxxx. Springer-Verlag (2023), *to appear*
8. de Azevedo Oliveira, D., Frappier, M.: *tastd-models-abz2023*. <https://github.com/DiegoOliveiraUDES/tastd-models-abz2023> (2023), [Online; accessed 26-January-2023]
9. Balaban, M., Rosen, T.: STCSP—structured temporal constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence* **25**, 35–67 (1999)
10. Behrmann, G., David, A., Larsen, K.G.: 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 pp. 200–236 (2004)
11. Börger, E., Stärk, R.: Abstract state machines: a method for high-level system design and analysis. Springer Science & Business Media (2012)
12. Cavalcanti, A., Sampaio, A., Miyazawa, A., Ribeiro, P., Conserva Filho, M., Didier, A., Li, W., Timmis, J.: Verified simulation for robotics. *Science of Computer Programming* **174**, 1–37 (2019)
13. Chaymae, E.J., Marc, F., Thibaud, E., Pierre-Martin, T.: Development of monitoring systems for anomaly detection using ASTD specifications. In: *International Symposium on Theoretical Aspects of Software Engineering*. pp. 274–289. Springer (2022)
14. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed automata patterns. *IEEE Transactions on Software Engineering* **34**(6), 844–859 (2008)
15. Fayolle, T., Frappier, M., Laleau, R., Gervais, F.: Formal refinement of extended state machines. *arXiv preprint arXiv:1606.02016* (2016)
16. Frappier, M.: ASTD support tools repo. <https://github.com/DiegoOliveiraUDES/ASTD-tools> (2023), [Online; accessed 26-January-2023]
17. Frappier, M., Gervais, F., Laleau, R., Milhau, J.: Refinement patterns for ASTDs. *Formal Aspects of Computing* **26**, 919–941 (2014)
18. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: 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. pp. 187–201. Springer (2014)
19. MATLAB: Stateflow. <https://www.mathworks.com/products/stateflow.html> (2020)
 20. Mekki, A., Ghazel, M., Toguyeni, A.: 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. pp. 1–13 (2009)
 21. Milhau, J., Frappier, M., Gervais, F., Laleau, R.: Systematic translation rules from astd to event-b. In: International Conference on Integrated Formal Methods. pp. 245–259. Springer (2010)
 22. Nganyewou Tidjon, L., Frappier, M., Leuschel, M., Mammar, A.: Extended algebraic state-transition diagrams. In: 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 146–155. Melbourne, Australia (2018)
 23. Schneider, S.: Concurrent and Real-time systems. John Wiley and Sons (2000)
 24. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, É.: Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **22**(1), 1–29 (2013)
 25. Sun, J., Liu, Y., Dong, J.S., Pang, J.: Pat: Towards flexible verification under fairness. *Lecture Notes in Computer Science*, vol. 5643, pp. 709–714. Springer (2009)
 26. Tidjon, L.N., Frappier, M., Mammar, A.: Intrusion detection using ASTDs. In: International Conference on Advanced Information Networking and Applications. pp. 1397–1411. Springer (2020)
 27. Tidjon, L.N.: Formal modeling of intrusion detection systems. Ph.D. thesis, Institut Polytechnique de Paris; Université de Sherbrooke (Québec, Canada) (2020), <https://theses.hal.science/tel-03137661>
 28. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. *Computer Science Review* **9**, 1–26 (2013)