Technical Report 27 - Extending ASTD with real-time

version 1.02 2019-05-24

Département d'informatique

Faculté des sciences

UNIVERSITÉ DE SHERBROOKE

Timed Algebraic State-Transition Diagrams

Diego de Azevedo Oliveira[1] and Marc Frappier[1]
[1] GRIL, Département d'informatique, Université de Sherbrooke
Sherbrooke (Quebec), J1K 2R1, Canada

Abstract

Timed Algebraic State Transition Diagrams (TASTD) is an extension of ASTD capable of specifying real-time models. ASTD 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. In the previous version of ASTD, we identified the need for dealing with real time and thus the need for adding time operators to the ASTD language. In this paper, we describe the syntax and semantics of these new time operators: delay, persistent delay, timeout, persistent timeout, and timed interrupt. These operators are specified using a new persistent guard operator and an interrupt operator. To show the generality of our time extensions, we simulate using TASTD the time operators of Stateful Timed CSP, a version of CSP capable to model real-time systems, and MATLAB Stateflow & Simulink, a well-known state-machine language widely used in the industry for control systems.

Keywords. *Formal methods, Algebraic-state transition diagrams, Real-time models, Process algebra, State machines.*

Contents

1 Introduction

ASTD is a graphical notation that combines process algebra operators and hierarchical state machines. It 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 [1, 2]. ASTD allows for the combination of state transition diagrams, such as statecharts, and automata, and process algebra operators, inspired from 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 [3], automata, with timed automata [4], and process algebra, with Timed CSP [5].

Each of these methods has its strengths and weaknesses. Timed automata is efficient for model checking, but lack high-level compositional patterns for hierarchical design [6]. Statecharts offers explicit representation of the control flow and support a rich notation for data modeling, however it does not offer the abstraction power of process algebra operators [2]. Process algebras support refinement and are also effective for model checking, but lack language features (e.g. shared variables) [7] and graphical representation. *tock*-CSP is an extension of CSP with the definition of a special event named *tock*, which marks the passage of time, but as CSP does not have graphical representation. TASTD tries to overcome their respective weaknesses and combine some of their strengths.

ASTDs draws from the statecharts notation the following concepts: hierarchy, OR-states, AND-states, guards, and history states. But it does not support broadcast communication or null transitions [8]. Automata constitute the base for ASTD construction: an elementary ASTD is an automaton. The process 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 operator. ASTDs also supports state variables and actions on transitions, states and ASTD structure.

TASTD extends ASTDs with real time. A TASTD state includes a timestamp which denote 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 ASTDs, a transition can be triggered only by external events. In contrast, a transition can also be triggered by the passage of time in a TASTD; such a transition is labelled by a special event called Step. 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.

TASTDs includes 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, *i.e.,*interrupt and persistent guard. In this technical report we also define these two new ASTD operators .

To show the generality of our time extensions, we made sure that TASTD operators could specify models presented in RoboSim [9], timed automata [10], and a automotive adaptive exterior light and speed control system case study from ABZ-2020 [11].

2 Related Work

We studied the different approaches for dealing with real time and developed one for ASTD. Our solution is inspired from timed automata from UPPAAL [12, 13], MATLAB Stateflow [3], stateful timed CSP[7], tock-CSP [14], and timed CSP [5].

From timed automata [4, 15, 12], we developed our notion for syntax, transitions, clocks, and invariants. Although, some concepts are not directly implemented or mentioned in TASTD (e.g invariants), it is possible to simulate their behavior. Another concept that is different are the clocks. With timed automata a user can have different clocks. In TASTD there is one system clock, which TASTD can't modify. Additionally, we introduce variables of type clock, that a user can reset, what assign a punctual time stamp from system clock, and access its value. It simulate the clocks in timed automata. In timed automata, the use of invariants and guards over the transitions are the approach to express timing constraints on transitions.

Stateful timed CSP [7], timed CSP [5], and *tock*-CSP [16, 14] contain operators that inspired TASTD operators delay, timeout, and timed interrupt. In TASTD, the operators delay, timeout and guard are provided in two versions: regular and persistent. The regular version of the operator applies to the first transition of its operand. The persistent version of the operator applies to every transition of its operand. With our operators, we can simulate all time operators of [7], [5], and [16, 14].

3 Semantics

3.1 States and transitions

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

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

representing a set of transitions of the form $(s, \mathsf{t}, w) \xrightarrow{\sigma} (s', \mathsf{t}', w')$. Such a transition means that a TASTD can execute event σ from state s and move to state s'. Symbols t, t' respectively denote the time at which states s, s' were reached. Symbols W, W' respectively denote the values of the parameters of ASTD a,

which can be modified durign execution. A state s contains attribute values and control values that are used to represent behavior of various ASTD operators. Note that t is a timestamp, and not a time duration. The value of t for the initial state of the system is some timestamp, which represent the time at which the system is started. The timestamp t of a state s is needed when making decision about various timing operators. For instance, a timeout is evaluated with respect to the last event executed. TASTD timing operators simulate clocks, and they rely 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 timing constraints. However, clocks can be declared as TASTD attributes and used to specify arbitrary timing constraints.

Suppose that A_2 is a sub-TASTD of A_1. A_1 may declare attributes that A_2 can use and modify. Thus, the behaviour of A_2 depends on the attributes declared in its enclosing ASTDs. In the operational semantics, we handle these attributes using *environments*. An environment is a function of $\mathsf{Env} \triangleq \mathsf{Var} \nrightarrow \mathsf{Term}$ which associates values to variables. The operational semantics of TASTDs is defined using an auxiliary transition relation $\mathcal{S}_a$ that deals with environments. A transition has the following form:

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

where E_e, E'_e denote the before and after values of identifiers in the TASTDs enclosing TASTD a. These identifiers could be TASTD parameters, quantified variables introduced by quantified operators like choice, synchronization and interleave, and attributes. Note that this transition relation does not include the timestamp t of the last executed event as part of the before state or the after state; instead, it is included in the transition information.

The initial state of the system is

$$(\mathit{init}(a, \mathsf{cst}, P := V), \mathsf{cst}, V)$$

Function *init* describes the initial state of an ASTD; it is inductively defined in the sequel on the ASTD types; it receives the initialisation time of an ASTD and the current value of the variables in the environment enclosing an ASTD, because the intial value of an attribute may depend on the values of some symbols in its environment. For a top-level ASTD a, the environment consists solely of the parameter values of a. The intialisation time is used only in ASTD types flow and synchronisation, because their sub-ASTDS each have their own clock, since their execution is independent. That will be further explained when these ASTD types are defined in the sequel. The timestamp of the last event executed is stored at the top-level state, and it is initialised with the current system time.

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

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

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'.

TASTDs are *non-deterministic*. If several transitions on σ are possible from a given state s, then one of them is non deterministically chosen. The operational semantics is inductively defined on $\mathcal{S}_a$ in Section 4 for each ASTD type.

3.2 Step Event

A concept introduced with TASTD is the special event Step. This event denotes the passage of time. It is sent by the system clock at some regular interval, chosen by the specifier. In other words, Step defines the desired time granularity chosen by the specifier. A Step is a default control event for TASTD, responsible

for checking if a time triggered transition may be fired from the actual state. Step is an event that can be used to annotate automata transitions, to denote time-triggered transitions. It is also implicitly used in time operators timeout, persistent timeout, and timed interrupt to trigger the timeout or the interrupt.

3.3 Event driven transitions versus time driven transitions

The semantics of Stateful Timed CSP and timed automata differs from the one we have chosen for TASTD. In their semantics, transition denotes either the passage of d units of time while staying in the same state (ie transitions of the form (s, d, s)), or a transition triggered by the reception of an event e (*i.e.,* transitions of the form (s, e, s')). This format is more amenable to model checking by computing time zones between states.

The semantics of TASTD is an advantage during code execution and deployment. In TASTD, time cannot fire a transition because we want to build a graphic language that produces executable code. Timed-triggered transitions are labelled by the special event Step.

In UPPAAL, clocks increase synchronously with the same rate, the guard of a transition restricts the behaviour of the automaton, and a transition can be fired when the clocks values satisfy that guard [15]. Since clocks increase synchronously at the same rate, that indicates that there is a value for that rate. It is the UPPAAL definition of time granularity, whichcomparable to our Step. The distinction between Step and rate is that Step is stated as submitted by the environment at some regular interval and UPPAAL does not state when a timed transition is triggered. It is up to the implementation to determine a reasonable rate interval when implementing a timed automata with UPPAAL.

With UPPAAL SMC [13], UPPAAL presents an approach to reason real-time systems with stochastic semantics. Moreover, it introduces the idea of clocks that evolve at different rates. That approach is particularly good when used with Statistical Model Checking (SMC). In TASTD, a specifier can model that the value of Step changes with an action over a transition. That is similar to varying the rate.

In Stateflow, we see transitions that occur in a given time. It is, in fact, triggered by Stateflow step. What is like UPPAAL rate and our definition of Step.

While having discrete time is an advantage during model verification, it deviates from the continuous time from our natural world. Additionally, the semantic discrete time from timed automata and Stateful Timed CSP allow Zeno runs in their models. That is, infinitely many steps may take a finite time.

TASTD compiler is also able to produce executable code for simulation, where time does not follow the system clock. With simulation, TASTD is set to model verification, where executable code will have a discrete time but will allow for Zeno runs.

4 Syntax and Semantics of TASTD Types

This section describes the syntax and semantics of TASTD, including the modifications to previously defined ASTD types, definition of TASTD types and the definition of two new ASTD types. The new persistent guard and interrupt types are ASTDs, while delay, persistent delay, timeout, persistent timeout and timed interrupt are TASTDs.

The syntax of TASTD is defined in terms of what we call *types*, which is a concept more comprehensive than just an operator. An TASTD type includes several characteristics; some types are based on well-know process algebra operators, others are based on automaton characteristics. Since several TASTD types share the same characteristics, we introduce them in a *generic* type, from which all the other types inherits; we denote this generic type by TASTD.

4.1 Type TASTD

Type TASTD is the most generic type of TASTD.

4.1.1 Properties

The common properties of TASTDs are the name, the parameters, the attributes, and the action, which we formally denote by

$$\mathsf{ASTD} \triangleq \langle n, P, V, A_{astd} \rangle$$

where $n \in \mathsf{Name}$ is the name of the TASTD, P is a list of parameters, V is a list of attributes, and $A_{astd} \in \mathcal{A}$ is an action. Parameters P are used to receive values passed by a calling TASTD; they can be read-only or read-write. Attributes V are state variables that can be modified by actions within the scope of the TASTD. Actions can also modify attributes received as parameters of the TASTD. A_{astd} is an action that is executed for every transition of the TASTD.

4.2 Previous ASTD Types

Since we are adding timestamps to TASTD's syntax and semantics, the previous existing ASTD described in [2] also need to be modified.

4.2.1 Automaton

A TASTD automaton substitute ASTD automaton, while it adequate the later to accept timestamps their composition is very similar. It is also similar to a timed automaton except that its states can be of any TASTD type, and its transition function can refer to or from substates of automaton states, as in statecharts.

4.2.1.1 Syntax

The timed automaton TASTD subtype has the following structure:

$$\text{Timed Automaton} \triangleq \langle \mathsf{aut}, \Sigma, S, \zeta, \nu, \delta, SF, DF, n_0 \rangle$$

$\Sigma \subseteq \mathsf{Event}$ is the alphabet. $S \subseteq \mathsf{Name}$ is the set of state names. An automaton state can also have action declarations. $\zeta \in S \to \langle A_{in}, A_{out}, A_{stay} \rangle$ maps each state name to its actions: A_{in} is executed when a transition enters the state; A_{out} is executed when a transition leaves the state; A_{stay} is executed when a transition loops on the state or is executed within the state. $\nu \in S \to \mathsf{ASTD}$ maps each state to its sub-ASTD, which can be elementary (noted elem) or complex. An automaton transition from n_1 to n_2 labelled with $\sigma[g]/A_{tr}$ is represented in the transition relation δ as follows:

$$\delta(\eta, \sigma, g, A_{tr}, \mathit{final}?)$$

Symbol η denotes the arrow. There are three types of arrows: $\langle \mathsf{loc}, n_1, n_2 \rangle$ denotes a local transition from n_1 to n_2, $\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat} \rangle$ denotes a transition from n_1 to substate $n_{2_\flat}$ of n_2 such that $n_{2\flat} \in \nu(n_2).S$, and $\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$ denotes a transition from substate $n_{1_\flat}$ of n_1 to n_2 such that $n_{1\flat} \in \nu(n_1).S$.

Symbol *final*? is a Boolean: when *final*? = 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. $SF \subseteq S$ is the set of shallow final states, while $DF \subseteq S$ denotes the set of deep final states, with $DF \cap SF = \varnothing$ and $DF \subseteq dom(\nu \rhd \{\mathsf{elem}\}$. $n_0 \in S$ is the name of the initial state. The definition of $final$ provided in the sequel will describe the distinction between the two types of final states.

The state of a timed automaton is a more complex structure of type $\langle \mathsf{aut}_\circ, n, E, h, s) \rangle$. $\mathsf{aut}_\circ$ is the constructor of the automaton state. $n \in S$ denotes the current state of the automaton. E contains the values of the automaton attributes. $h \in S \nrightarrow \mathsf{State}$ is the history function that implements the notion of *history state* used in statecharts; it records the last visited sub-state of a state. $s \in \mathsf{State}$ is state of the sub-ASTD of n, when n is a complex state; $s = \mathsf{elem}$ when n is elementary.

Functions *init* and *final* are defined as follows. Let a be an timed automaton TASTD.

$$\begin{aligned} init(a, \mathsf{ts}, E) &\triangleq (\mathsf{aut}_\circ, a.n_0, a.E_{init}([\![E]\!]), h_{init}, init(a.\nu(n_0), \mathsf{ts}, a.E_{init}([\![E]\!]))) \\ h_{init} &\triangleq \{n \mapsto init(a.\nu(n), \mathsf{cst}, a.E) \mid n \in a.S\} \\ final(a, (\mathsf{aut}_\circ, n, E, h, s)) &\triangleq n \in a.SF \vee (n \in a.DF \wedge final(a.\nu(n), s)) \end{aligned}$$

Function *init* receives a timestamp ts and an environment E. The timestamp ts is passed down to its sub-ASTD. The timestamp of a state denote when a state was reached.

Symbol E_{init} denotes the initial values of attributes, as specified in their declaration. Symbol h_{init} is the initial value of the history function; it maps each state name to the initial state of its internal structure: elementary states are mapped to the constants elem and cst (*i.e.,* $init(\mathsf{elem}, \mathsf{cst}) = \mathsf{elem}, \mathsf{cst}$); composite automaton states are mapped to the initial state of their sub-ASTD, recursively. Symbol ts is the time stamp of the current system time. A deep final state is final only when its sub-ASTD is also final, whereas a shallow final state is final irrespective of the state of its sub-ASTD.

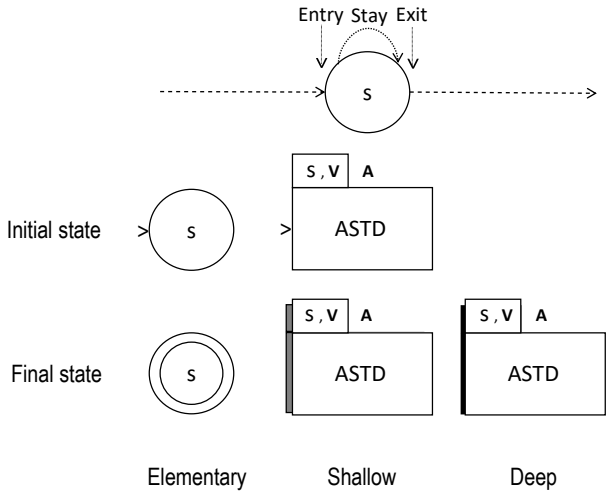

Figure 1. ASTD state

4.2.1.2 Semantics

For all transitions, there are six rules of inference, written in the usual form $\frac{premiss}{conclusion}$. Each rule describes the semantic of the transition between states.

4.2.1.2.1 Transition Between Local States : loc

The first rule, aut_1, describe a transition between local states.

$$\mathsf{aut}_1 \ \frac{a.\delta((\mathsf{loc}, n_1, n_2), \sigma', g, A_{tr}, final?) \quad \Psi \quad \Omega_{loc}}{(\mathsf{aut}_\circ, n_1, E, h, s_1) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', h', init(a.\nu(n_2), \mathsf{cst}))}$$

The conclusion of this rule states that a transition on event σ can occur from n_1 to n_2 with before and after automaton attributes values E, E', and before and after values h, h' for the history state of statecharts. The state of the sub-ASTD of n_2 is its initial state and the initialization occurs in cst(*i.e.,* $init(a.\nu(n_2), \mathsf{cst})$). The premiss provides that such a transition is possible if there is a matching transition, which is represented by $\delta((\mathsf{loc}, n_1, n_2), \sigma', g, A_{tr}, final?)$. σ' is the event labelling the transition, and it may contain variables. The value of these variables is given by the environment E_e and local attributes values E, which can be applied as a substitution to a formula using operator $(\![\ ]\!)$. This match on the transition is provided by premiss Ψ defined as follows.

$$\Psi \triangleq \quad (\quad (final? \Rightarrow final(a.\nu(n_1), s)) \quad \wedge \quad g \quad \wedge \quad \sigma' = \sigma \quad)(\!|E_g|\!)$$

Ψ can be understood as follows. If the transition is final (*i.e., final*? = true), then the current state must be final. The transition guard g holds. The event received, noted σ, must match the event pattern σ' which labels the automaton transition, after applying the environment E_g as a substitution. Pattern matching rules defined at the beginning of Section 4.2.1 are abbreviated here by the equality $\sigma = \sigma'$. Environment E_g is defined below. The premiss Ω_{loc} determines how the attributes, the transition environment and the history function are computed.

$$\Omega_{loc} \triangleq \left\{ \begin{array}{l} \textbf{if } n_1 = n_2 \textbf{ then} \\ \quad A = A_{tr} \ ; \ a.\zeta(n_1).A_{stay} \ ; \ a.A_{astd} \\ \textbf{else} \\ \quad A = a.\zeta(n_1).A_{out} \ ; \ A_{tr} \ ; \ a.\zeta(n_2).A_{in} \ ; \ a.A_{astd} \\ \textbf{end} \\ E_g = E_e \lhd\!\!\lhd E \\ A(E_g, E'_g) \\ E'_e = E_e \lhd\!\!\lhd (a.V \lhd\!\!\lhd E'_g) \\ E' = a.V \lhd E'_g \\ h' = h \lhd\!\!\lhd \{n_1 \mapsto s_1\} \end{array} \right\}$$

Premiss Ω_{loc} can be understood as follows. Let $A(E, E')$ denote that E' is a possible after value of executing action A on E (*i.e., A* is a predicate that relates the before and after values of the attributes of the ASTD a). When the transition is a loop (*i.e.,* $n_1 = n_2$), the actions executed are the transition action A_{tr}, followed by the stay action $\zeta(n_1).A_{stay}$ of state n_1 and finally the ASTD action $a.A_{astd}$, which is declared in the heading of the automaton. The ASTD action is useful to factor out state modifications that must be done on every transition of the ASTD. State actions (entry, exit and stay) are useful to factor out modifications that must done on all transitions upon entry, exit or loop, of a given state. When the transition is not a loop (*i.e.,* $n_1 \neq n_2$), the actions executed are the exit code of n_1, the transition action, the entry code of n_2 and finally the ASTD action. Symbol E_g, defined as as $E_e \lhd\!\!\lhd E$, denotes the global list of variables that can be modified by the actions. Their after values E'_g are used to set E' (the local attributes) using the restriction on the attributes V declared in the ASTD and the values E'_e (the attributes declared in enclosing ASTDs). The history function in the target state, noted h', is updated by storing the last visited sub-state of n_1.

4.2.1.2.2 Transitions to a Substate, But Not a History State

Rule aut_2, handles transitions to substates, in the particular case where the substate is not an history

state.

$$\Omega_{tsub\_base} \triangleq \left\{ \begin{array}{l} \textbf{if } n_1 = n_2 \textbf{ then} \\ \quad A = A_{tr} \ ; \ a.\zeta(n_1).A_{stay} \\ \textbf{else} \\ \quad A = a.\zeta(n_1).A_{out} \ ; \ A_{tr} \ ; \ a.\zeta(n_2).A_{in} \\ \textbf{end} \\ E_g = E_e \lhd E \\ A(E_g, E'_g) \\ a.\nu(n_2).\zeta(n_{2_\flat}).A_{in}(E_{g\flat}, E'_{g\flat}) \\ E'_\flat = a.\nu(n_2).V \lhd E'_{g\flat} \\ E''_g = E'_g \lhd (a.\nu(n_2).V \lhd E'_{g\flat}) \\ a.A_{astd}(E''_g, E'''_g) \\ E'_e = E_e \lhd (a.V \lhd E'''_g) \\ E' = a.V \lhd E'''_g \\ h' = h \lhd \{n_1 \mapsto s_1\} \end{array} \right\}$$

$$\Omega_{tsub} \triangleq \left\{ \begin{array}{l} \Omega_{tsub\_base} \\ E_{g\flat} = E'_g \lhd a.\nu(n_2).V_{init} \end{array} \right\}$$

$$\mathsf{aut}_2 \ \frac{a.\delta((\mathsf{tsub}, n_1, n_2, n_{2_\flat}), \sigma', g, A_{tr}, \mathit{final}?) \quad n_{2_\flat} \notin \{\mathsf{H}, \mathsf{H}^*\} \quad \Omega_{tsub} \quad \Psi}{(\mathsf{aut}_\circ, n_1, E, h, s_1) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', h', (\mathsf{aut}_\circ, n_{2_\flat}, E'_\flat, a.\nu(n_2).h_{init}, \mathit{init}(a.\nu(n_{2_\flat}), \mathsf{cst}, E')))}$$

The target state is n_2, with $n_{2_\flat}$ as its substate. The initial state of the substate is targeted (since this substate could also be an ASTD). A transition exits the local state n_1 and moves to the substate $n_{2_\flat}$ of the automaton state n_2. The premiss states that there must be a matching transition from n_1 to substate $n_{2_\flat}$, i.e $a.\delta((\mathsf{tsub}, n_1, n_2, n_{2_\flat}), \sigma', g, A_{tr}, \mathit{final}?)$, and that the substate $n_{2_\flat}$ is not a history state (the rules aut_3 and aut_4 is used for this).

Ω_{tsub} executes multiple actions : A, $A_{in_\flat}$ and $a.A_{astd}$. When the transition is a loop toward a substate $n_{2_\flat}$ (i.e., $n_1 = n_2$), then A executes the transition action A_{tr}, followed by the stay action $\zeta(n_1).A_{stay}$ of state n_1.

When the transition is not a loop (i.e., $n_1 \neq n_2$), then A executes the exit code $\zeta(n_1).A_{out}$ of state n_1, the transition action A_{tr} and the entry code $\zeta(n_2).A_{in}$ of state n_2. The global list of variables E_g is used to execute A since it contains the attributes E of the ASTD a. Before executing the ASTD action A_{astd}, we must execute the actions inside the state n_2.

The environment $E_{g\flat}$ contains the initial attribute values of n_2, given by $a.\nu(n_2).V_{init}$, as well as the attributes of the ASTD a. It is used for the execution of the entry code of $n_{2_\flat}$, which needs to be executed on the environment of the state n_2. It produces the updated $E'_{g\flat}$, from which we can extract the attribute values $E'_\flat$ of n_2.

Then the ASTD action A_{astd} is executed with the updated global list of variables E''_g which accounts for updates to attributes of the ASTD a (made by A and $A_{in_\flat}$), but does not have the values of the attributes of the automaton in state n_2. The resulting environment E'''_g is used to update the environment E'_e and the attribute values E' of ASTD a. In summary, here is the sequence of actions execution when there is no loop (*i.e.,* $n_1 \neq n_2$).

1. $a.\zeta(n_1).A_{out}$: the exit action of state n_1
2. A_{tr} : the transition action from n_1 to n_2
3. $a.\zeta(n_2).A_{in}$: the entry action of state n_2

4. $a.\nu(n_2).\zeta(n_{2_\flat}).A_{in}$: the entry action of substate $n_{2_\flat}$
5. $a.A_{astd}$: the action of ASTD a, which contains the transition from n_1 to n_2

Thus, the convention is that actions are executed from source to destination, and bottom-up for ASTD actions.

When the transition is a loop (*i.e.,* $n_1 = n_2$), the first three steps are replaced by the following two steps.

1. A_{tr} : the transition action from n_1 to n_2
2. $a.\zeta(n_2).A_{stay}$: the stay action of state n_2

4.2.1.2.3 Transition to a Shallow History State

Rule aut_3 handles transitions to a *shallow* history state (noted H), following the behaviour prescribed by statecharts.

$$\Omega_{tsubh} \triangleq \left\{ \begin{array}{l} \Omega_{tsub\_base} \\ E_{g\flat} = E'_g \lhd h(n_2).E \end{array} \right\}$$

$$\mathsf{aut}_3 \ \frac{a.\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}), \sigma', g, A_{tr}, \mathit{final?}) \quad n_{2_\flat} = \mathit{name}(h(n_2)) \quad \Omega_{tsubh} \quad \Psi}{(\mathsf{aut}_\circ, n_1, E, h, s_1) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', h', (\mathsf{aut}_\circ, n_{2_\flat}, E'_\flat, a.\nu(n_2).h_{init}, \mathit{init}(\nu(n_{2_\flat}), \mathsf{cst}, E'_\flat)))}$$

This case is similar to the previous one. Function *name* returns the name of an automaton state: $\mathit{name}(\mathsf{aut}_\circ, n, \ldots) = n$. In the case of shallow history, the target state is the *initial state* of the ASTD referenced by $h(n_2)$. The premiss Ω_{tsubh} differs only from Ω_{tsub} in that it must use the stored values of the attributes declared in n_2 instead of their initial values, when executing the entry code of $n_{2_\flat}$.

4.2.1.2.4 Transition to a Deep History State

Rule aut_4 handles transitions to a *deep* history state (noted H^*); in that case, the target state is the full state recorded in $h(n_2)$, but with the attributes of n_2 updated by the entry code of $n_{2_\flat}$.

$$\mathsf{aut}_4 \ \frac{a.\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}^*), \sigma', g, A_{tr}, \mathit{final?}) \quad n_{2_\flat} = \mathit{name}(h(n_2)) \quad \Omega_{tsubh} \quad \Psi}{(\mathsf{aut}_\circ, n_1, E, h, s_1) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', h', (\mathsf{aut}_\circ, n_{2_\flat}, E'_\flat, h(n_2).h, h(n_2).s))}$$

4.2.1.2.5 Transition From a Substate

Rule aut_5 handles transitions from a substate.

$$\Omega_{fsub} \triangleq \left\{ \begin{array}{l} E_g = E_e \lhd E \\ a.\nu(n_1).\zeta(n_{1_\flat}).A_{out}(E_g \lhd E_\flat, E_0) \\ \textbf{if } n_1 = n_2 \textbf{ then} \\ \quad A = A_{tr} \ ; \ a.\zeta(n_1).A_{stay} \\ \textbf{else} \\ \quad A = a.\zeta(n_1).A_{out} \ ; \ A_{tr} \ ; \ a.\zeta(n_2).A_{in} \\ \textbf{end} \\ A(E_g \lhd (\nu(n_1).V \lhd E_0), E'_g) \\ E'_e = E_e \lhd (a.V \lhd E'_g) \\ E' = a.V \lhd E'_g \\ E'_\flat = a.\nu(n_1).V_\flat \lhd E_0 \\ h' = h \lhd \{n_1 \mapsto (\mathsf{aut}_\circ, n_{1_\flat}, E'_\flat, h_\flat, s'')\} \end{array} \right\}$$

$$\mathsf{aut}_5 \ \frac{a.\delta((\mathsf{fsub}, n_1, n_{1_\flat}, n_2), \sigma', g, A_{tr}, \mathit{final?}) \quad \Psi_{fsub} \quad \Omega_{fsub}}{(\mathsf{aut}_\circ, n_1, E, h, (\mathsf{aut}_\circ, n_{1_\flat}, E_\flat, h_\flat, s'')) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e} (\mathsf{aut}_\circ, n_2, E', h', \mathit{init}(\nu(n_2), \mathsf{cst}))}$$

This rule uses a slightly different version of Ψ, called Ψ_{fsub} which takes into account the substate $n_{1_\flat}$ for the final condition, instead of n_1.

$$\Psi_{fsub} \triangleq \ (\quad (\mathit{final?} \Rightarrow \mathit{final}(a.\nu(n_1).\nu(n_{1_\flat}), s'')) \quad \wedge \quad g \quad \wedge \quad \sigma' = \sigma \quad)([\![E_g]\!])$$

A transition is executed from the substate $n_{1_\flat}$ of composite state n_1 to state n_2 when there is a matching transition $a.\delta((\mathsf{fsub}, n_1, n_{1_\flat}, n_2), \sigma',\ g, A_{tr}, \mathit{final?})$. When exiting substate $n_{1_\flat}$, the exit action $a.\nu(n_1).\zeta(n_{1_\flat}).A_{out}$ of $n_{1_\flat}$ is executed on the global list of variables E_g and the attribute values $E_\flat$ of n_1. When moving from n_1 to n_2, action A is executed on the updated attributes E_g. The history function h' is computed by replacing $E_\flat$ with $E'_\flat$, which is computed from E_0, to reflect the updates done to the attributes of n_1 before exiting it.

4.2.1.2.6 Transition Within a State

Rule aut_6, handles transitions within the sub-ASTD $a.\nu(n)$ of state n.

$$\Omega_{sub} \ \triangleq \ a.\zeta(n).A_{stay}(E'''_g, E''_g)$$

$$\Theta \ \triangleq \ \Big(E_g = E_e \lhd E \quad a.A_{astd}(E''_g, E'_g) \quad E'_e = E_e \lhd (V \lhd E'_g) \quad E' = V \lhd E'_g\Big)$$

$$\mathsf{aut}_6 \ \frac{s \xrightarrow{\sigma,E_g,E'''_g}_{a.\nu(n)} s' \qquad \Omega_{sub} \qquad \Theta}{(\mathsf{aut}_\circ, n, E, h, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{aut}_\circ, n, E', h, s')}$$

The transition starts from a sub-state s and moves to the sub-state s' of state n. Actions are executed bottom-up. E'''_g denotes the values computed by the ASTD of state n. Premiss Ω_{sub} defines the computation of E''_g from E'''_g using the stay code of state n and the computation of E'_g from E''_g by executing the ASTD action A_{astd}. E'_e and E' are extracted by partitioning E'_g using V. Premiss Θ is reused in all subsequent rules where a sub-ASTD transition is involved.

One can easily observe that the semantics of transitions to, and from, substates (*i.e.,* rules aut_2, aut_3, aut_4 and aut_5) is significantly more complex than the semantics of transitions between local states and within a state. These features are inherited from statecharts and are less elegant than the pure compositional and algebraic style of rules aut_1 and aut_6. However, these features seem to be very well appreciated by practitioners, so we decided to introduce them in the ASTD notation.

4.2.2 Sequence

The sequence ASTD allows for the sequential composition of two ASTDs. When the first item reaches a final state, the second one can start its execution. This enables decomposing problems into a set of tasks that have to be executed in sequence.

In Figure 2, the enclosing ASTD sequence c has two sub-ASTDs a and b that acts sequentially one after the other. The automaton ASTD a must reach its final state 2 before b can start its execution. Then, b starts at the initial state 3 and stop its execution at the final state 4. So, when the system is in state 2, it can accept event e2 and stay in state 2, or it can accept event e3 and move to state 4. Note that reaching a final state of an automaton does not mean that this automaton stops its execution; it means that it enables the next ASTD in the sequence.

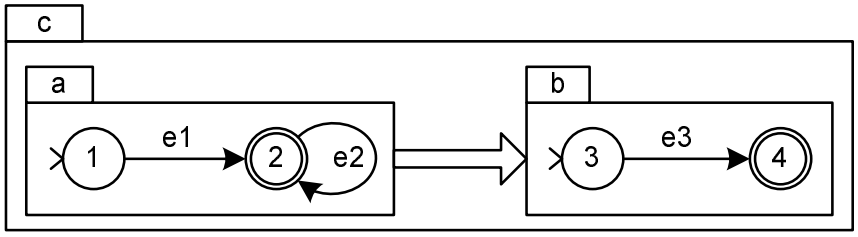

Figure 2. Example - ASTD sequence

4.2.2.1 Syntax

The sequence ASTD subtype has the following structure:

$$\mathsf{Sequence} \triangleq \langle \leftthreetimes, \mathsf{fst}, \mathsf{snd} \rangle$$

where fst, snd are ASTDs denoting respectively the first and second sub-ASTDs of the sequence. A sequence state is of type $\langle \leftthreetimes_\circ, E, [\mathsf{fst} \mid \mathsf{snd}], s \rangle$, where $\leftthreetimes_\circ$ is a constructor of the sequence state, E the values of attributes declared in the sequence, $[\mathsf{fst} \mid \mathsf{snd}]$ is a choice between two markers that respectively indicate whether the sequence is in the first sub-ASTD or the second sub-ASTD and $s \in \mathsf{State}$. Functions *init* and *final* are defined as follows. Let a be a sequence ASTD.

$$\begin{aligned} init(a, \mathsf{ts}, E) &\triangleq (\leftthreetimes_\circ, a.E_{init}([\![E]\!]), \mathsf{fst}, init(a.fst, \mathsf{ts}, a.E_{init}([\![E]\!]))) \\ final(a, (\leftthreetimes_\circ, E, \mathsf{fst}, s)) &\triangleq final(a.fst, s) \wedge final(a.snd, init(a.snd, \bot, E)) \\ final(a, (\leftthreetimes_\circ, E, \mathsf{snd}, s)) &\triangleq final(a.snd, s) \end{aligned}$$

The initial state of a sequence is the initial state of its first sub-ASTD. A sequence state is final when either i) it is executing its first sub-ASTD and this one is in a final state, and the initial state of the second sub-ASTD is also a final state; ii) it is executing the second sub-ASTD which is in a final state. The timestamp does not change the fact if a state is final, so it is hidden with a $\bot$.

4.2.2.2 Semantics

Three rules are necessary to define the execution of the sequence. Rule $\leftthreetimes_1$ deals with transitions on the sub-ASTD fst only. Rule $\leftthreetimes_2$ deals with transitions from fst to snd, when fst is in a final state. Rule $\leftthreetimes_3$ deals with transitions on the sub-ASTD *snd*.

$$\leftthreetimes_1 \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.fst} s' \qquad \Theta}{(\leftthreetimes_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\leftthreetimes_\circ, E', \mathsf{fst}, s')}$$

$$\leftthreetimes_2 \frac{final(a.fst, s) \qquad init(a.snd, \mathsf{t}, E_e) \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.snd} s' \qquad \Theta}{(\leftthreetimes_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\leftthreetimes_\circ, E', \mathsf{snd}, s')}$$

$$\leftthreetimes_3 \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.snd} s' \qquad \Theta}{(\leftthreetimes_\circ, E, \mathsf{snd}, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\leftthreetimes_\circ, E', \mathsf{snd}, s')}$$

4.2.3 Choice

A choice ASTD allows a choice between two sub-ASTDs. Once a sub-ASTD has been chosen, the other sub-ASTD is ignored. It is essentially the same as a choice operator in a process algebra. The choice is nondeterministic if each sub-ASTD can execute the requested event. An example. Figure 3 provides an example of a choice ASTD, which includes two automaton sub-ASTDs. If e1 is received, then a is chosen to execute it. The subsequent events will be accepted by a only. Dually, if e3 is received, then b is chosen to execute it. If e2 is received, then a nondeterministic choice is made between a and b to execute it.

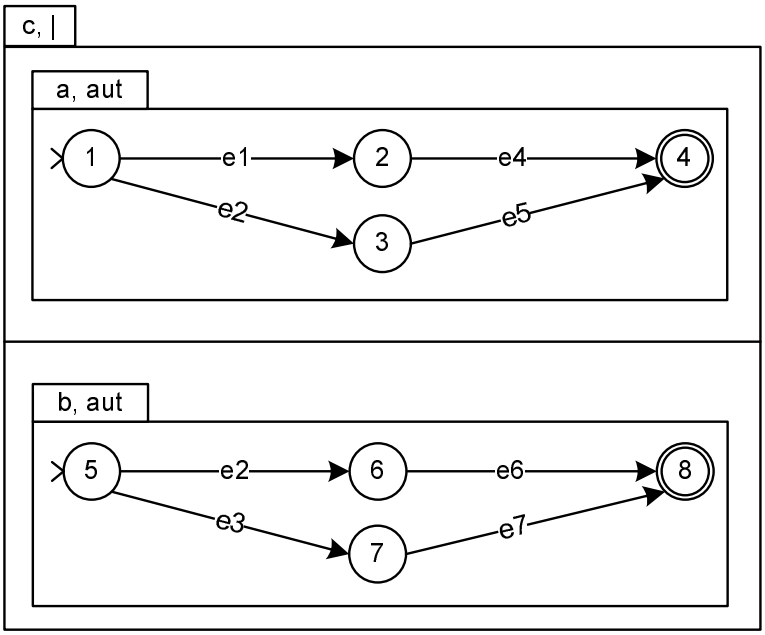

Figure 3. Example - ASTD choice

4.2.3.1 Syntax

The choice ASTD subtype has the following structure:

$$\mathsf{Choice} \triangleq \langle |, l, r \rangle$$

where $l, r \in \mathsf{ASTD}$ are respectively the first and second element of the choice. The type of a choice state is $\langle |_\circ, E, side, s \rangle$ where $side \in (\bot \mid \langle \mathsf{left} \rangle \mid \langle \mathsf{right} \rangle)$ denotes the sub-ASTD which has been chosen, $s \in (\mathsf{State} \mid \bot)$ denotes the state of the sub-ASTD which has been chosen and E the values of attributes declared in the choice ASTD. A choice state is final if i) it hasn't started yet and the initial state of each sub-ASTD is final, or ii) the chosen sub-ASTD is in a final state. Here are the formal definitions of the initial state and the final states. Let a be a choice ASTD.

$$\begin{aligned}
init(a, \mathsf{ts}, E) &\triangleq (|_\circ, a.E_{init}([\![E]\!]), \bot, \bot) \\
final(a, (|_\circ, E_{init}, \bot, \bot)) &\triangleq final(init(a.l, \bot, E_{init})) \vee final(init(a.r, \bot, E_{init})) \\
final(a, (|_\circ, E, \mathsf{left}, s)) &\triangleq final(a.l, s) \\
final(a, (|_\circ, E, \mathsf{right}, s)) &\triangleq final(a.r, s)
\end{aligned}$$

4.2.3.2 Semantics

There are four rules of inference. The first two deal with the execution of the first event from the initial state. The other two deal with execution of the subsequent events from the chosen sub-ASTD.

$$|_1 \frac{init(a.l, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.l} s' \qquad \Theta}{(|_\circ, E_{init}, \bot, \bot) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|_\circ, E', \mathsf{left}, s')}$$

$$|_2 \frac{init(a.r, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.r} s' \qquad \Theta}{(|_\circ, E_{init}, \bot, \bot) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|_\circ, E', \mathsf{right}, s')}$$

$$|_3 \frac{s \xrightarrow{\sigma,\mathsf{t},\Gamma,E_g,E''_g}_{a.l} s' \qquad \Theta}{(|_\circ, E, \mathsf{left}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|_\circ, E', \mathsf{left}, s')}$$

$$|_4 \frac{s \xrightarrow{\sigma,\mathsf{t},\Gamma,E_g,E''_g}_{a.r} s' \qquad \Theta}{(|_\circ, E, \mathsf{right}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|_\circ, E', \mathsf{right}, s')}$$

4.2.4 Kleene closure

This operator comes from regular expressions. It allows for iteration on an ASTD an arbitrary number of times (including zero). When the sub-ASTD is in a final state, it enables to start a new iteration. A Kleene closure is in a final state when it has not started or when its sub-ASTD is in a final state. Figure 4 presents the case of two nested ASTD f and e, where f is an closure ASTD and e is an ASTD sequence. During the execution, e has the precedence on f, i.e, f starts its execution after e terminates to execute. The ASTD sequence e contains two sub-ASTDs c and d, which are both two nested ASTDs. Each contains respectively two ASTD automaton a and b. The LHS closure ASTD c must be reach its final state before the next one (d) starts its execution.

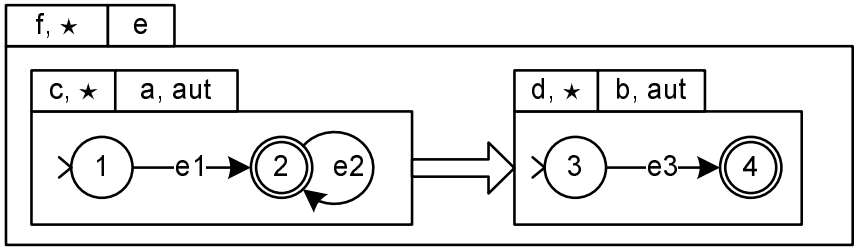

Figure 4. Example - closure ASTD

4.2.4.1 Syntax

The closure ASTD subtype has the following structure:

$$\mathsf{Closure} \triangleq \langle \star, b \rangle$$

where $b \in \mathsf{ASTD}$ is the body of the closure. The type of a closure state is $\langle \star_\circ, E, started?, s \rangle$ where $s \in \mathsf{State}$, E the attribute values of the closure ASTD, $started? \in \mathsf{Boolean}$ indicates whether the first

iteration has been started. It is essentially used to determine if the closure can immediately exit (*i.e.,* is in a final state) without any iteration. Initial and final states are defined as follows. Let a be a closure ASTD.

$$\begin{aligned} init(a, \mathsf{ts}, E) &\triangleq (\star_\circ, a.E_{init}(\llbracket E \rrbracket), \mathsf{false}, \bot) \\ final(a, (\star_\circ, E, started?, s)) &\triangleq final(a.b, s) \vee \neg started? \end{aligned}$$

4.2.4.2 Semantics

There are two inference rules: $\star_1$ allows for restarting from the initial state of the sub-ASTD when a final state has been reached; $\star_2$ allows for execution on the sub-ASTD.

$$\star_1 \frac{final(a.b, s) \qquad init(a.b, \mathsf{t}, E_e) \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.b} s' \qquad \Theta}{(\star_\circ, E, \_, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\star_\circ, E', \mathsf{true}, s')}$$

$$\star_2 \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.b} s' \qquad \Theta}{(\star_\circ, E, \_, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\star_\circ, E', \mathsf{true}, s')}$$

4.2.5 Parameterized Synchronization

A parameterized synchronization ASTD executes two sub-ASTDs in parallel; they must synchronize on a set Δ of event. In Figure 5, the synchronization ASTD c has two sub-ASTDs a and b, which are two automaton ASTDs. At the beginning of the execution, a and b are respectively in the state 1 and 5. When a receives an event e1, a transition is executed from 1 to 2. Another transition is triggered from 5 to 6 upon the reception of the event e4. Events e1 and e4 can be executed in interleave. Next, transitions from 2 to 3 and 6 to 7 are executed simultaneously on the reception of the event e2, since a and b must synchronize on events of Δ. Finally, the transitions from 3 and 7 interleave on the reception of e3 and e5.

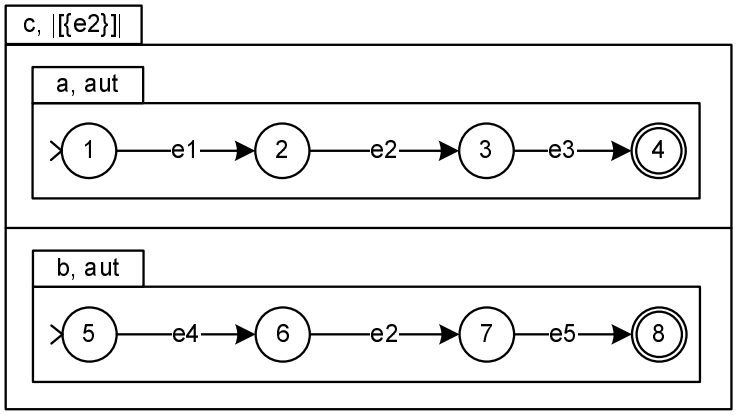

Figure 5. Example - synchronization ASTD

4.2.5.1 Syntax

The parameterized synchronization ASTD subtype has the following structure:

$$\mathsf{Synchronization} \triangleq \langle |[]|, \Delta, l, r \rangle$$

where Δ is the synchronization set of event labels, $l, r \in \mathsf{ASTD}$ are the synchronized 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 $\Delta = \varnothing$, the synchronization is called an interleaving and is abbreviated as $|||$. Symbol $\|$ is used to denote a synchronization on $\Delta = \alpha(l) \cap \alpha(r)$, the events that are common to both sub-ASTDs.

A parameterized synchronization state is of type $\langle |[]|_\circ, E, s_l, c_l, s_r, c_r \rangle$, where s_l, s_r are the states of the left and right sub-ASTDs and $c_l, c_r \in C$. Initial and final states are defined as follows. Let a be a parameterized synchronized ASTD.

$$\begin{array}{rcl} \mathit{init}(a, \mathsf{ts}, E) & \triangleq & (|[]|_\circ, a.E_{init}(\![E]\!), \mathit{init}(a.l, \mathsf{ts}, a.E_{init}(\![E]\!)), \mathsf{ts}, \mathit{init}(a.r, \mathsf{ts}, a.E_{init}(\![E]\!)), \mathsf{ts}) \\ \mathit{final}(a, (|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r)) & \triangleq & \mathit{final}(a.l, s_l) \wedge \mathit{final}(a.r, s_r) \end{array}$$

4.2.5.2 Semantics

There are three inference rules. Rules $|[]|_1$ and $|[]|_2$ respectively describe execution of events, with no synchronization 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.

$$|[]|_1 \frac{\alpha(\sigma) \notin \Delta \qquad s_l \xrightarrow{\sigma, \mathsf{t}_l, E_g, E_g''}_{a.l} s_l' \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma, \mathsf{t}, E_e, E_e'}_{a} (|[]|_\circ, E', s_l', \mathsf{cst}, s_r, \mathsf{t}_r)}$$

Rule $|[]|_2$ is symmetric to $|[]|_1$ and indicates behaviour when the right side execute the action.

$$|[]|_2 \frac{\alpha(\sigma) \notin \Delta \qquad s_r \xrightarrow{\sigma, \mathsf{t}_r, E_g, E_g''}_{a.r} s_r' \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma, \mathsf{t}, E_e, E_e'}_{a} (|[]|_\circ, E', s_l, \mathsf{t}_l, s_r', \mathsf{cst})}$$

The most interesting case is when the left and right sub-ASTDs must synchronize on an event (*i.e.,* when $\alpha(\sigma) \in \Delta$). Consider the transitions on the left and right sub-ASTDs when each of them is executed independently of the other.

$$\Omega_{ilr} \triangleq \left(s_l \xrightarrow{\sigma, \mathsf{t}, E_g, E_{gl}'}_{a.l} s_l' \qquad s_r \xrightarrow{\sigma, \mathsf{t}, E_g, E_{gr}'}_{a.r} s_r' \right)$$

Since shared variables (E_g) can be modified by both sub-ASTDs, their modifications could be inconsistent, which should forbid the synchronization transition. This requires to check that $E_{gl}' = E_{gr}'$. However, when one variable is modified by both sub-ASTDs, the natural intent is typically that the compound result of both sides is desired, that is, to assume that one side executes on the values returned by the other. For instance, assume that both sub-ASTDs increment shared variable x by 1 and assume that the before value of x is 0. The above semantics gives $x = 1$, whereas the compound result is $x = 2$. If one sub-ASTD increments by 1 and the other by 2, the above semantics forbids execution because the result is inconsistent, whereas the compound execution returns 3. Executing the left sub-ASTD before the right sub-ASTD is specified as follows.

$$\Omega_{lr} \triangleq \left(s_l \xrightarrow{\sigma, \mathsf{t}, E_g, E_{g1}'}_{a.l} s_l' \qquad s_r \xrightarrow{\sigma, \mathsf{t}, E_{g1}', E_g''}_{a.r} s_r' \right)$$

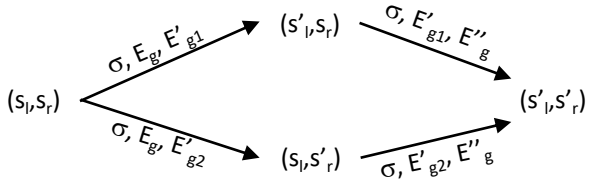

Figure 6. Commutativity of actions execution in a parameterized synchronization on σ

Executing the right sub-ASTD before the left sub-ASTD is specified as follows.

$$\Omega_{rl} \triangleq \left(s_r \xrightarrow{\sigma,\mathsf{t},E_g,E'_{g2}}_{a.r} s'_r \qquad s_l \xrightarrow{\sigma,\mathsf{t},E'_{g2},E''_g}_{a.l} s'_l \right)$$

Since synchronization should be commutative, *i.e.,* both execution orders should return the same states. Figure 6 illustrates this property.

Checking this commutativity at each transition is expensive. To improve performance, it could be statically checked using proof obligations, or by analysis of the variables read and written by each sub-ASTD, to ensure that the left and right sub-ASTDs are independent (*i.e.,* they do not modify the same variables and one sub-ASTD does not modify the variables read by the other). When the synchronization operands are nondeterministic, this is still expensive to execute, because a commutative combination must be found, which means enumerating combinations of possibilities from both sides. Still, this is our preferred semantics for a synchronization, because it works well for deterministic specifications. Here is the rule for synchronization.

$$|[]|_3 \frac{\alpha(\sigma) \in \Delta \qquad \Omega_{lr} \qquad \Omega_{rl} \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|[]|_\circ, E', s'_l, \mathsf{cst}, s'_r, \mathsf{cst})}$$

4.2.6 Flow

It is quite common that the same event e can be part of several cyber attack specifications, as illustrated in Fig. 7. An intrusion detection system needs to execute such an event on each attack specification that can execute it; this behaviour is not fulfilled by either interleaving or synchronization of attack specifications. This raises the need of a new operator $\Cup$, called flow, inspired from AND states of statecharts, which executes an event on each sub-ASTD whenever possible. In contrast to other ASTD operators, the rules for this operator involve negation, *i.e.,* one has to determine whether an event can be executed in the sub-ASTDs.

4.2.6.1 Syntax

The flow ASTD subtype has the following structure:

$$\mathsf{Flow} \triangleq \langle \Cup, l, r \rangle$$

A flow state is of type $\langle \Cup_\circ, E, s_l, t_l, s_r, t_r \rangle$, where s_l, s_r are the states of the left and right sub-ASTDs and t_l, t_r are real values greater than 0. Initial and final states are defined as follows. Let a be a flow ASTD.

$$\begin{aligned} \mathit{init}(a, \mathsf{ts}, E) &\triangleq (\Cup_\circ, a.E_{init}([\![E]\!]), \mathit{init}(a.l, \mathsf{ts}, a.E_{init}([\![E]\!])), \mathsf{ts}, \mathit{init}(a.r, \mathsf{ts}, a.E_{init}([\![E]\!])), \mathsf{ts}) \\ \mathit{final}(a, (\Cup_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r)) &\triangleq \mathit{final}(a.l, s_l) \wedge \mathit{final}(a.r, s_r) \end{aligned}$$

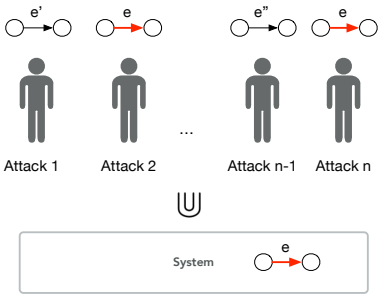

Figure 7. Using the Flow operator to synchronize multiple attack models

4.2.6.2 Semantics

We use the following abbreviation to denote that an ASTD cannot execute a transition from a state s and global attributes E_g.

$$s \xrightarrow{\sigma,\mathsf{t},E_g} \!\!\!\!\!/\;\; a \;\triangleq\; \neg \exists\, E'_g, s' \cdot s \xrightarrow{\sigma,\mathsf{t},E_g,E'_g}_a s'$$

The negation of a transition predicate is computed using the usual negation as failure approach. Here are the first two rules when only one of the two sub-ASTDs can execute the event.

$$\mathbb{U}_1 \frac{s_l \xrightarrow{\sigma,\mathsf{t}_l,E_g,E''_g}_{a.l} s'_l \qquad s_r \xrightarrow{\sigma,\mathsf{t}_r,E''_g}\!\!\!\!\!/\;\;_{a.r} \qquad \Omega_{lr} \Leftrightarrow \Omega_{rl} \qquad \Theta}{(\mathbb{U}_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_a (\mathbb{U}_\circ, E', s'_l, \mathsf{cst}, s_r, \mathsf{t}_r)}$$

$$\mathbb{U}_2 \frac{s_r \xrightarrow{\sigma,\mathsf{t}_r,E_g,E''_g}_{a.r} s'_r \qquad s_l \xrightarrow{\sigma,\mathsf{t}_l,E''_g}\!\!\!\!\!/\;\;_{a.l} \qquad \Omega_{lr} \Leftrightarrow \Omega_{rl} \qquad \Theta}{(\mathbb{U}_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_a (\mathbb{U}_\circ, E', s_l, \mathsf{t}_l, s'_r, \mathsf{t}_r)}$$

The premiss $\Omega_{lr} \Leftrightarrow \Omega_{rl}$ ensures that if one execution order succeeds, then the other must also succeed. It ensures the determinacy of the flow operator.

The third rule describes the case where both sub-ASTDs can execute the event; it is almost the same as $||[]||_3$, as it requires commutativity.

$$\mathbb{U}_3 \frac{\Omega_{lr} \qquad \Omega_{rl} \qquad \Theta}{(\mathbb{U}_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_a (\mathbb{U}_\circ, E', s'_l, \mathsf{cst}, s'_r, \mathsf{cst})}$$

4.2.7 Quantified choice

The quantified choice is very similar to an existential quantification in first-order logic. It allows for picking a value from a set and execute a sub-ASTD with that value. The scope of the quantified variable is the sub-ASTD. In Figure 8, the closure ASTD a has two nested ASTDs b and c, where b is an ASTD choice and c an ASTD automaton. In its initial state, this ASTD can execute either $\mathtt{e1}(4)$, $\mathtt{e1}(5)$ or $\mathtt{e1}(6)$. If $\mathtt{e1}(4)$ is received, x is instantiated with 4, and a transition from state 1 to 2 is executed; the next event that can be received is $\mathtt{e2}(4)$. At each iteration, a new value for x can be chosen.

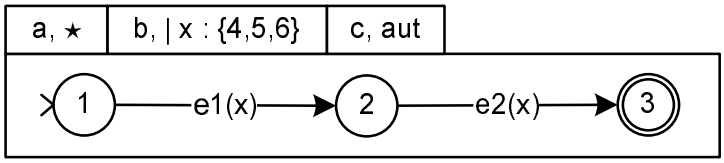

Figure 8. Example - ASTD quantified choice

4.2.7.1 Syntax

A quantified choice ASTD subtype has the following structure:

$$\mathsf{QChoice} \triangleq \langle |:, x, T, b \rangle$$

where $x \in \mathsf{Var}$ denotes a quantification variable, T is a type and $b \in \mathsf{ASTD}$ is the quantified ASTD. The type of a quantification choice state is $\langle |:_\circ, [\bot \mid c], E, [\bot \mid s] \rangle$ where $|:_\circ$ is the constructor of the quantification choice state, $\bot$ is a constant indicating that the choice has not been made yet, $c \in \mathsf{Term}$ denotes the current value of the choice quantified variable once the ASTD choice has been made, E the values of attributes, $s \in \mathsf{State}$. Initial and final states are defined as follows. Let a be a quantified choice ASTD.

$$\begin{array}{rcl} \mathit{init}(a, \mathsf{ts}, E) & \triangleq & (|:_\circ, \bot, a.E_{\mathit{init}}([\![E]\!]), \bot) \\ \mathit{final}(a, (|:_\circ, \bot, E_{\mathit{init}}, \bot)) & \triangleq & \exists\, x : T \cdot \mathit{final}(a.b, \mathit{init}(a.b, \bot, E_{\mathit{init}})) \\ c \neq \bot \Rightarrow (\mathit{final}(a, (|:_\circ, c, E, s)) & \triangleq & \mathit{final}(a.b, s)\{x \mapsto c\}) \end{array}$$

4.2.7.2 Semantics

There is two inference rules. They use the notion of environment to manage the quantification. When a transition is computed using rules, the value c bound to the quantification variable x is added to the execution environment (the one appearing on the transition arrow) and can be used to make the proof, in particular to check that the event received σ matches the transition event σ', after the environment has been applied as a substitution. This behavior is expressed hereafter.

$$|:_1 \frac{\mathit{init}(a.b, \mathsf{ts}, E_e) \xrightarrow{\sigma, \mathsf{t}, E_g \lhd \{x \mapsto d\}, E''_g}_{a.b} s' \qquad \Theta \qquad d \in T}{(|:_\circ, \bot, E_{\mathit{init}}, \bot) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (|:_\circ, d, E', s')}$$

$$|:_2 \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g \lhd \{x \mapsto d\}, E''_g}_{a.b} s' \qquad \Theta \qquad d \neq \bot}{(|:_\circ, d, E, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (|:_\circ, d, E', s')}$$

4.2.8 Quantified Synchronization

The quantified synchronization allows for the modeling of an arbitrary number of instances of an ASTD which are executing in parallel, synchronizing on events from Δ. For IS modeling, it allows one to concisely and explicitly represent the behavior of each instances of an entity type or an association. The quantified synchronized ASTD consists of the synchronization set Δ, the synchronization variable with its type T and a sub-ASTD b. Figure 9 is similar to a parameterized quantification but now includes the quantified variable x. The enclosing closure ASTD enables looping on the nested quantified

synchronization ASTD. For each instance of x, the automaton sub-ASTD of the quantified ASTD is executed. These automata can execute in interleave e1 and e3, but they must synchronize on event e2 of Δ.

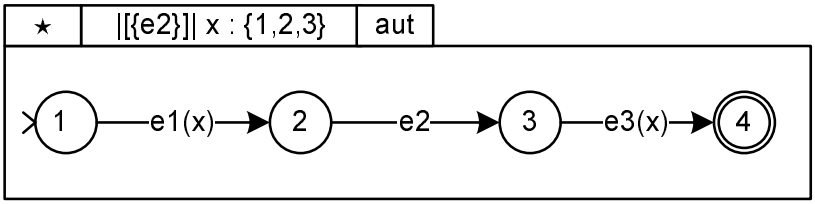

Figure 9. Example - quantified synchronization

4.2.8.1 Syntax

The quantified synchronization ASTD subtype has the following structure:

$$\mathsf{QSynchronization} \triangleq \langle |[]|_{:}, x, T, \Delta, b \rangle$$

where $x \in \mathsf{Var}$ a quantified variable that can be only accessed in read-only mode, T the type of x, $\Delta \subseteq \mathsf{Label}$ a synchronization set of event labels and $b \in \mathsf{ASTD}$ the body of the synchronization. The state of a quantified synchronization is of type $\langle |[]|_{:\circ}, E, f, u \rangle$ where $|[]|_{:\circ}$ is the constructor, E the values of attributes, $f \in T \to \mathsf{State}$ is a function which associates a state of b to each value of T, $u \in T \to R_{0+}$ is a function which associates a time-stamp u for each value of T. Initial and final states are defined as follows. Let a be a quantified synchronized ASTD.

$$\begin{array}{rcl} \mathit{init}(a, \mathsf{ts}, E) & \triangleq & (|[]|_{:\circ}, a.E_{\mathit{init}}(\llbracket E \rrbracket), T \times \{\mathit{init}(a.b, \mathsf{ts}, a.E_{\mathit{init}}(\llbracket E \rrbracket))\}, T \times \mathsf{ts}\}) \\ \mathit{final}(a, (|[]|_{:\circ}, E, f, u)) & \triangleq & \forall\, d : T \cdot \mathit{final}(a.b, f(d)) \end{array}$$

4.2.8.2 Semantics

Rule $|[]|_{:1}$ describe execution of events with no synchronization. Symbol c denotes the element of T chosen for the execution.

$$|[]|_{:1}\ \frac{\alpha(\sigma) \notin \Delta \qquad f(d) \xrightarrow{\sigma, u(d), E_g \lhd\!\!+ \{x \mapsto d\}, E''_g}_{a.b} s' \qquad \Theta}{(|[]|_{:\circ}, E, f, u) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (|[]|_{:\circ}, E', f \lhd\!\!+ \{d \mapsto s'\}, u \lhd\!\!+ \{d \mapsto \mathsf{cst}\})}$$

Rule $|[]|_{:2}$ describe execution of an event with synchronization. All elements of T must execute σ, in any order. It generalizes rule $|[]|_3$ and requires commutativity.

$$|[]|_{:2}\ \frac{\alpha(\sigma) \in \Delta \qquad \Omega_{qsync} \qquad \Theta}{(|[]|_{:\circ}, E, f, u) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (|[]|_{:\circ}, E', f', u \lhd\!\!+ (T * \mathsf{cst}))}$$

Premiss Ω_{qsync} formalizes commutativity by universally quantifying over all permutations p of T (noted $p \in \pi(T)$) and using Es as a sequence of environments storing the intermediate results of the computation of E''_g from E_g by iterating over the elements $p(i)$ of p. Let $k = |T|$.

$$\Omega_{qsync} \triangleq \left(\begin{array}{c} \forall\, p \in \pi(T) \cdot \exists\, Es \in 0..k \rightarrowtail \mathsf{Env} \wedge Es(0) = E_g \wedge E(k) = E''_g \wedge \\ \forall\, i \in 1..k \cdot f(p(i)) \xrightarrow{\sigma, u(p(i)), Es(i-1) \lhd\!\!+ \{x \mapsto p(i)\}, Es(i)}_{a.b} f'(p(i)) \end{array} \right)$$

4.2.9 Guard

A guard ASTD guards the execution of its sub-ASTD using a predicate or a function declaration of its parent ASTD. The first event received must satisfy the guard predicate. Once the guard has been satisfied by the first event, the sub-ASTD execute the subsequent events without further constraints from its enclosing guard ASTD. The predicate may refer to variables whose scope include the guard.

The guard ASTD is a generalization of the guard specified on an automaton transition. It is especially useful when the sub-ASTD is a complex structure, avoiding the duplication of the guard predicate on all the possible first transitions of that structure. For example, we assume that the guard ASTD below is executed for $x := 1$. The nested ASTD a_1 can start its execution, but for $x := -1$ no execution is possible.

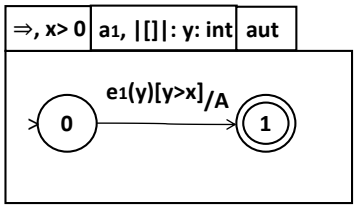

Figure 10. Example - guard

4.2.9.1 Syntax

The guard ASTD subtype has the following structure:

$$\mathsf{Guard} \triangleq \langle \Rightarrow, g, b \rangle$$

where $b \in \mathsf{ASTD}$ is the body of the guard. The type of a guard state is $\langle \Rightarrow_\circ, E, started?, s \rangle$ where $started?$ denotes when the guard has been satisfied, $s \in \mathsf{State}$, E the attribute values of the guard ASTD. Initial and final states are defined as follows. Let a be a guard ASTD.

$$\begin{array}{rcl} init(a, \mathsf{ts}, E) & \triangleq & (\Rightarrow_\circ, a.E_{init}([\![E]\!]), \mathsf{false}, init(a.b, \mathsf{ts}, a.E_{init}([\![E]\!]))) \\ final(a, (\Rightarrow_\circ, E_{init}, \mathsf{false}, init(a.b, \mathsf{ts}, E_{init}))) & \triangleq & final(a, init(a.b, \bot, E_{init})) \\ final(a, (\Rightarrow_\circ, E, \mathsf{true}, s)) & \triangleq & final(a, s) \end{array}$$

4.2.9.2 Semantics

There are two inference rules: $\Rightarrow_1$ deals with the first transition and the satisfaction of the guard predicate; $\Rightarrow_2$ deals with subsequent transitions.

$$\Rightarrow_1 \frac{g([\![E_e]\!]) \qquad init(a.b, \mathsf{t}, E_e) \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.b} s'}{(\Rightarrow_\circ, E_{init}, \mathsf{false}, init(a.b, \mathsf{t}, E_e)) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e} (\Rightarrow_\circ, E', \mathsf{true}, s')}$$

$$\Rightarrow_2 \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.b} s'}{(\Rightarrow_\circ, E, \mathsf{true}, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e} (\Rightarrow_\circ, E', \mathsf{true}, s')}$$

4.2.10 Call

It is possible to call an ASTD which is defined in another diagram. A call is graphically represented by the called ASTD name and its actual parameter values. Calls can be recursive.

Figure 11 presents a quantified ASTD a1 which is enclosed by a closure ASTD. When the quantified variable x is instantiated, the call ASTD $a2(x)$ enables referring to an ASTD automaton $a2$. The value of x is used in $a2$ to compute the transition from 0 to 1.

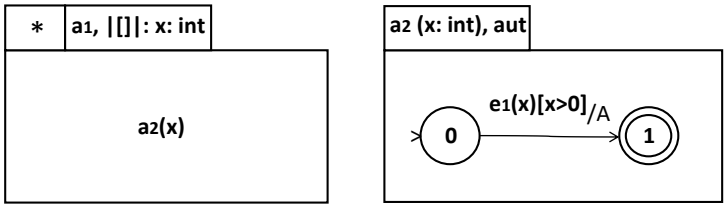

Figure 11. Example - call

4.2.10.1 Syntax

A call ASTD is of the subtype $\mathsf{ASTDCall} \triangleq \langle \mathsf{cal}, n(\vec{c}) \rangle$ where n is the name of an ASTD $q = \langle n, P, V, A_{astd} \rangle$. Let $P = \vec{x} : \vec{T}$. For each $c_i \in \vec{c}$, we have $c_i \in T_i$. The type of an ASTD call state is $\langle \mathsf{cal}_\circ, E, [\bot \mid s] \rangle$, where $\mathsf{cal}_\circ$ is the constructor of the call state, E the values of attributes, $\bot$ denotes that the call has not been made yet, $s \in \mathsf{State}$ is the state of the called ASTD q once the called has been made. The initial and final states are as follows. Let a be an ASTD call.

$$
\begin{array}{rcl}
\mathit{init}(a, \mathsf{ts}, E) & \triangleq & (\mathsf{cal}_\circ, a.E_{init}([\![E]\!]), \bot) \\
\mathit{final}(a, (\mathsf{cal}_\circ, E_{init}, \bot)) & \triangleq & \mathit{final}(q, \mathit{init}(q, \bot, E_{init}))([\vec{x} := \vec{d}]) \\
s \neq \bot \Rightarrow \mathit{final}(a, (\mathsf{cal}_\circ, E, s)) & \triangleq & \mathit{final}(q, s)([\vec{x} := \vec{d}])
\end{array}
$$

4.2.10.2 Semantics

There are two rules of inference. Rule cal_1 deals with the initial call execution, while cal_2 deals with subsequent executions. Let $E_{cal} = \{P \mapsto \vec{c}\}$.

$$
\mathsf{cal}_1 \ \frac{\mathit{init}(a.b, \mathsf{t}, E_e) \xrightarrow{\sigma, \mathsf{t}, E_g \lhd E_{cal}, E''_g}_{a.b} s' \qquad \Theta}{(\mathsf{cal}_\circ, E_{init}, \bot) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\mathsf{cal}_\circ, E', s')}
$$

$$
\mathsf{cal}_2 \ \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g \lhd E_{cal}, E''_g}_{a.b} s' \qquad \Theta}{(\mathsf{cal}_\circ, E, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\mathsf{cal}_\circ, E', s')}
$$

4.3 New ASTD Types

This section presents two new ASTD types, persistent guard and interrupt. Their syntax and semantics described here use the TASTD syntax and semantics presented in the section 4 but could also use the ASTD semantics presented in technical report 25.

Persistent guard declares a guard that shall hold through all the execution of the sub-ASTD.

Interrupt allows ASTD to have precedence. With interrupt an ASTD has the preference to execute over another.

These new types are formally defined below:

4.3.1 Persistent Guard

A persistent guard ASTD guards the execution of its sub-ASTD using a predicate or a function declaration of its parent ASTD. The guard predicate must be satisfied to allow the ASTD execution. The predicate may refer to variables whose scope include the guard.

The persistent guard ASTD is especially useful when the sub-ASTD is a complex structure, avoiding the duplication of the guard predicate on the transitions of that structure.

For example, we assume that the persistent guard ASTD below is executed for $x < 1000$. Once $x >= 1000$ no execution is possible.

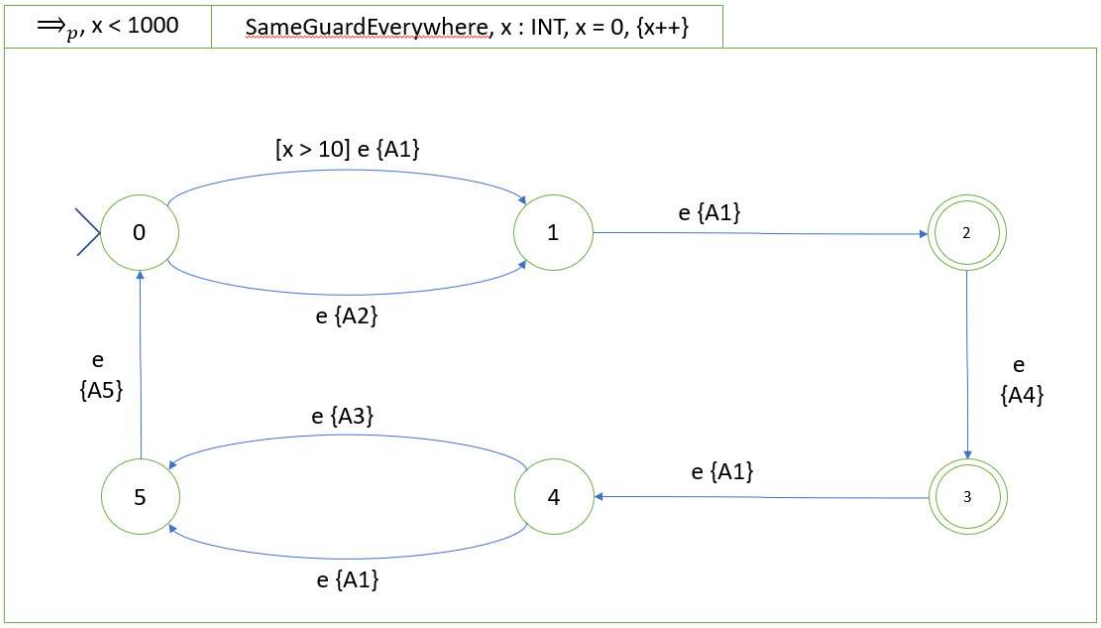

Figure 12. Example - persistent guard

4.3.1.1 Syntax

The persistent guard ASTD subtype has the following structure:

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

where $b \in \mathsf{ASTD}$ is the body of the persistent guard. The type of a persistent guard state is $\langle \Rightarrow_p, E, s \rangle$ where $s \in \mathsf{State}$, E the attribute values of the persistent guard ASTD. Initial and final states are defined

as follows. Let a be a persistent guard ASTD.

$$\begin{aligned} \mathit{init}(a, \mathsf{ts}, E) &\triangleq (\Rightarrow_p, a.E_{\mathit{init}}(\llbracket E \rrbracket), \mathit{init}(a.b, \mathsf{ts}, a.E_{\mathit{init}}(\llbracket E \rrbracket))) \\ \mathit{final}(a, (\Rightarrow_p, E, s)) &\triangleq \mathit{final}(a, s) \end{aligned}$$

The initial and final states of a persistent guard ASTD are straightforward. For the initial state, it initialises the persistent guard's body with the variables of the persistent guard and the environment. The final state is final if the persistent guard's body is final.

4.3.1.2 Semantics

There is one inference rules: $\Rightarrow_{p1}$ execute any transition from the ASTD that always shall hold on g.

$$\Rightarrow_{p1} \frac{g(\llbracket E_e \rrbracket) \qquad s \xrightarrow{\sigma, \mathsf{t}, E_g, E_g''}_{a.b} s' \qquad \Theta}{(\Rightarrow_p, E, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E_e'} (\Rightarrow_p, E', s')}$$

4.3.2 Interruption

The interruption ASTD subtype declares an precedence of execution between two sub-ASTD's. With the interruption ASTD the second ASTD has a precedence over the first ASTD. That means, when an event of the second ASTD occurs and the first ASTD is the one with the right of execution the second ASTD may take the right of execution.

4.3.2.1 Syntax

The interruption ASTD subtype has the following structure:

$$\mathsf{Interrupt} \triangleq \langle \mathsf{Interruption}, \mathsf{fst}, \mathsf{snd}, A_{Int} \rangle$$

where $\mathsf{fst}, \mathsf{snd} \in \mathsf{ASTD}$ are the sub-ASTDs and A_{Int} is an action for when the interruption occurs.

An interruption state is of type $\langle \mathsf{Interruption}_\circ, E, [\mathsf{fst}|\mathsf{snd}], s \rangle$, where $\mathsf{Interruption}_\circ$ represent a constructor of the interrupt state, E the value of the attributes declared in the interrupt, $[\mathsf{fst}|\mathsf{snd}]$ is a choice between the two markers that respectively indicate whether the interruption is in the first sub-ASTD or the second sub-ASTD, and $s \in \mathit{State}$.

Let a be an interruption TASTD.

$$\begin{aligned} \mathit{init}(a, \mathsf{ts}, E) &\triangleq (\mathsf{Interruption}_\circ, a.E_{\mathit{init}}(\llbracket E \rrbracket), \mathsf{fst}, \mathit{init}(a.fst, \mathsf{ts}, a.E_{\mathit{init}}(\llbracket E \rrbracket))) \\ \mathit{final}(a, (\mathsf{Interruption}_\circ, E, \mathsf{fst}, s)) &\triangleq \mathit{final}(a.fst, s) \\ \mathit{final}(a, (\mathsf{Interruption}_\circ, E, \mathsf{snd}, s)) &\triangleq \mathit{final}(a.\mathsf{snd}, s) \end{aligned}$$

The initial state of an interruption initialises its body with the variables of the interrupt and the environment. The final state of a interrupt depends on its interrupt state: if its interrupt state is in the first sub-ASTD, then it is final if the first sub-ASTD is final; if its interrupt state is in the second sub-ASTD, then it is final if the second sub-ASTD is final.

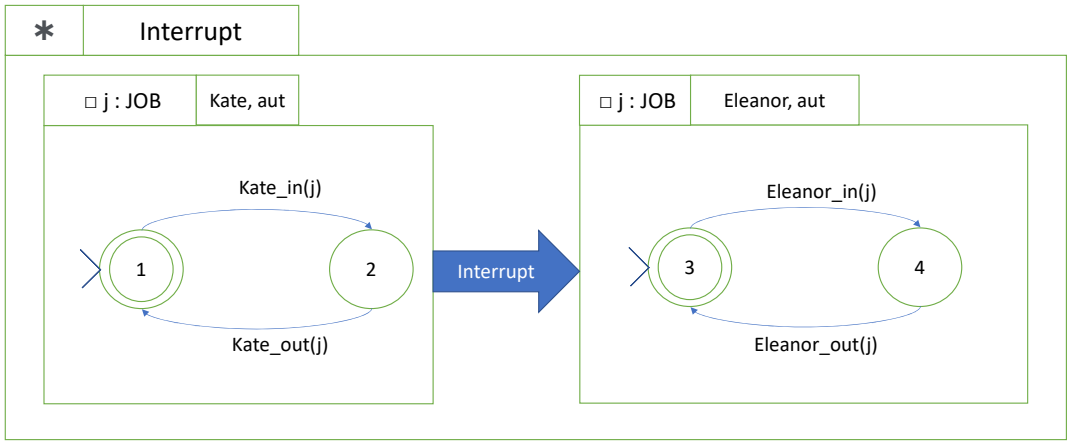

Figure 13. Example - interruption

4.3.2.2 Semantics

We define the semantics of interruption execution with three rules. $\mathsf{Interruption}_1$ allows for the execution of the first sub-ASTD. $\mathsf{Interruption}_2$ allows for the interruption execution when the event σ from the second astd happen. $\mathsf{Interruption}_3$ allows for the execution of the second sub-ASTD after the interruption.

$$\mathsf{Interruption}_1 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.fst} s' \qquad \Theta}{(\mathsf{Interruption}_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{Interruption}_\circ, E', \mathsf{fst}, s')}$$

$$\mathsf{Interruption}_2 \frac{\Omega_{Interrupt} \qquad init(a.\mathsf{snd}, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E'''_g,E''_g}_{a.\mathsf{snd}} s' \qquad \Theta}{(\mathsf{Interruption}_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{Interruption}_\circ, E', \mathsf{snd}, s')}$$

$$\mathsf{Interruption}_3 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.\mathsf{snd}} s' \qquad \Theta}{(\mathsf{Interruption}_\circ, E, \mathsf{snd}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{Interruption}_\circ, E', \mathsf{snd}, s')}$$

$$\Omega_{Interrupt} \triangleq a.A_{Int}(E_g, E'''_g)$$

4.3.2.3 Example

The copier from 13 has two user profiles, Kate and Eleanor.

Eleanor usually prints important documents and shall have immediate access to the copier. If Kate is using the copier and it receives a document from Eleanor, the copier shall print Eleanor's document and forget about Kate's. Kate can resend the document once Eleanor has finished to use the copier.

The interruption ASTD allows the copier to manage the precedence behaviour between Kate and Eleanor profiles.

4.4 Delay

Delay TASTD allows for the idling for at least an arbitrary time d before the first event. Once the first event occurred, the TASTD may continue its execution without further delay.

4.4.1 Syntax

Delay TASTD subtype has the following structure:

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

where $b \in \mathsf{TASTD}$ is the body of the delay and d is the delay value in time units.

The type of a delay state is $\langle \mathsf{Delay}_\circ, E, \mathit{started}?, s \rangle$ where $s \in \mathit{State}$, E the attribute values of the delay TASTD, and $\mathit{started}?$ is a boolean which indicates if the first event occurred.

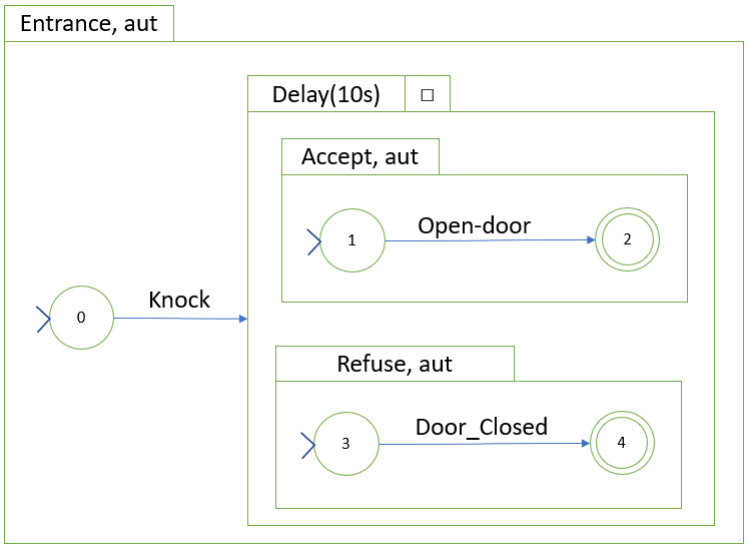

Figure 14. Example - Open door

Initial and final states are defined as follows. Let a be a TASTD.

$$\begin{aligned} \mathit{init}(a, \mathsf{ts}, E) &\triangleq (\mathsf{Delay}_\circ, a.E_{\mathit{init}}([\![E]\!]), \mathsf{false}, \mathit{init}(a.b, \mathsf{ts}, a.E_{\mathit{init}}([\![E]\!]))) \\ \mathit{final}(a, (\mathsf{Delay}_p, E, \mathit{started}?, s)) &\triangleq \mathit{final}(a.b, s) \end{aligned}$$

Delay initial state initialises its body with the variables of the delay and the environment. A delay is final if the current state is final.

4.4.2 Semantics

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

$$\mathsf{Delay}_1 \frac{\mathsf{cst} - \mathsf{t} > d \qquad \mathit{init}(a.b, \mathsf{t}, E_e) \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.b} s \qquad \Theta}{(\mathsf{Delay}_\circ, E, \mathsf{false}, \mathit{init}(a.b, \mathsf{t}, E)) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\mathsf{Delay}_\circ, E', \mathsf{true}, s)}$$

$$\mathsf{Delay}_2 \frac{s \xrightarrow{\sigma, \mathsf{t}, E_g, E''_g}_{a.b} s' \qquad \Theta}{(\mathsf{Delay}_\circ, E, \mathsf{true}, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E'_e}_{a} (\mathsf{Delay}_\circ, E', \mathsf{true}, s')}$$

4.4.3 Example

Figure 14 bring the example of delay TASTD usage. Entrance TASTD models the knock in a door and then the response of the butler with events *Open_Door* or *Door_Closed* Once the door is knocked, with the use event *Knock*, the butler has at least 10 seconds to decide if the person may come in, simulated with *Open_Door*, or the door should remain shut, simulated with *Door_Closed*.

4.4.4 Equivalence between Delay and Guard ASTDs

Delay TASTD is similar to Guard ASTD, while delay is a short-term for a guard over the variables cst and t. The equivalence is shown in Figure 15.

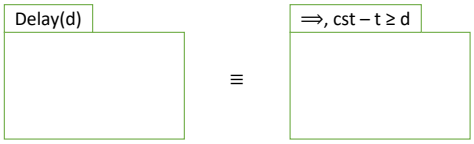

Figure 15. Equivalence between delay TASTD and guard ASTD

4.5 Persistent Delay

Persistent delay TASTD allows for the idling for at least an arbitrary time d after each event.

4.5.1 Syntax

Persistent delay TASTD subtype has the following structure:

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

where $b \in$ TASTD is the body of the delay and d is the delay value in time units.

The type of a delay state is $\langle \mathsf{Delay}_p, E, s \rangle$ where $s \in State$, E the attribute values of the delay TASTD.

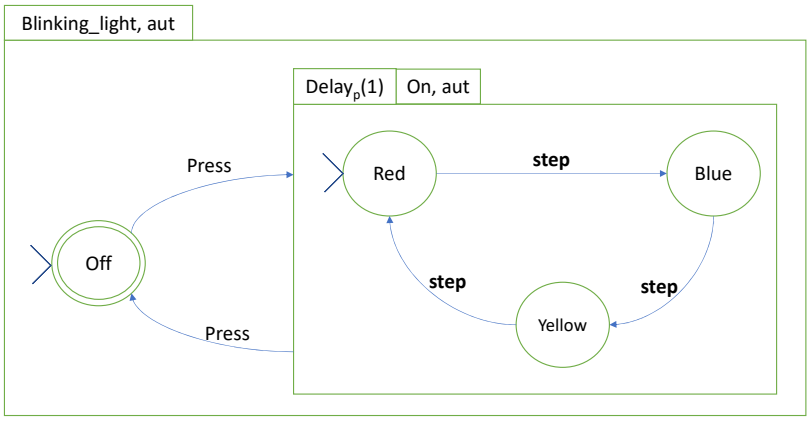

Figure 16. Example - Blinking lights

Initial and final states are defined as follows. Let a be a TASTD.

$$
\begin{aligned}
\mathit{init}(a, \mathsf{ts}, E) &\triangleq (\mathsf{Delay}_p, a.E_{\mathit{init}}(\llbracket E \rrbracket), \mathit{init}(a.b, \mathsf{ts}, a.E_{\mathit{init}}(\llbracket E \rrbracket))) \\
\mathit{final}(a, (\mathsf{Delay}_p, E, s)) &\triangleq \mathit{final}(a.b, s)
\end{aligned}
$$

Persistent delay initial state initialises its body with the variables of the persistent delay and the environment. A persistent delay is final if the current state is final.

4.5.2 Semantics

There is one inference rule for Persistent Delay: Delay_{p_1} allows for the transition on the sub-TASTD after idling for at least d time steps.

$$
\mathsf{Delay}_{p_1} \frac{\mathsf{cst} - \mathsf{t} > d \qquad s \xrightarrow{\sigma, \mathsf{t}, E_g, E_g''}_{a.b} s' \qquad \Theta}{(\mathsf{Delay}_p, E, s) \xrightarrow{\sigma, \mathsf{t}, E_e, E_e'}_{a} (\mathsf{Delay}_p, E', s')}
$$

4.5.3 Example

A LED lamp, shown in Figure 16, which changes its color between three different colors, red, blue and yellow, with the delay of 1 second represented on persistent delay TASTD.

At the starting state, the lamp is off. When someone press the lamp button, the TASTD executes the press event and the lamp turn on, beginning with the red color.

While the light is on and there isn't no event press, the lamp changes between the colors red, blue and yellow. The persistent delay TASTD allows the change of state when a step event occurs after 1 second of the last event.

4.5.4 Equivalence between Persistent Delay and Persistent Guard ASTDs

Similarly to the equivalence between Delay TASTD and Guard ASTD, their persistent forms are equivalent, while persistent delay is a short-term for a persistent guard over the variables cst and t. The equivalence is shown in Figure 17.

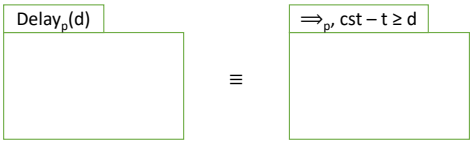

Figure 17. Equivalence between persistent delay TASTD and persistent guard ASTD

4.6 Timeout

Timeout TASTD subtype allows for verifying if a TASTD receives an event in a period of time, if not, then another TASTD has the right to execute. In the timeout TASTD subtype, the first transition of the first TASTD has to occur before d time units, otherwise the timeout is executed.

4.6.1 Syntax

Timeout TASTD subtype has the following structure:

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

where $\mathsf{fst}, \mathsf{snd} \in \mathsf{TASTD}$ are the sub-TASTDs, d represent the time units necessary for a timeout, and A_{TO} is an optional action triggered by the timeout.

A timeout state is of type $\langle \mathsf{Timeout}_\circ, E, [\mathsf{fst}|\mathsf{snd}], s, first? \rangle$ where $\mathsf{Timeout}_\circ$ is a constructor of the timeout state, E represent the value of attributes declared in the timeout, $[\mathsf{fst}|\mathsf{snd}]$ represent a choice between two markers which respectively indicate whether the timeout is in the first sub-TASTD or the second sub-TASTD, $s \in State$, and $first? \in BOOL$ is a boolean value to indicate if the first event has been executed.

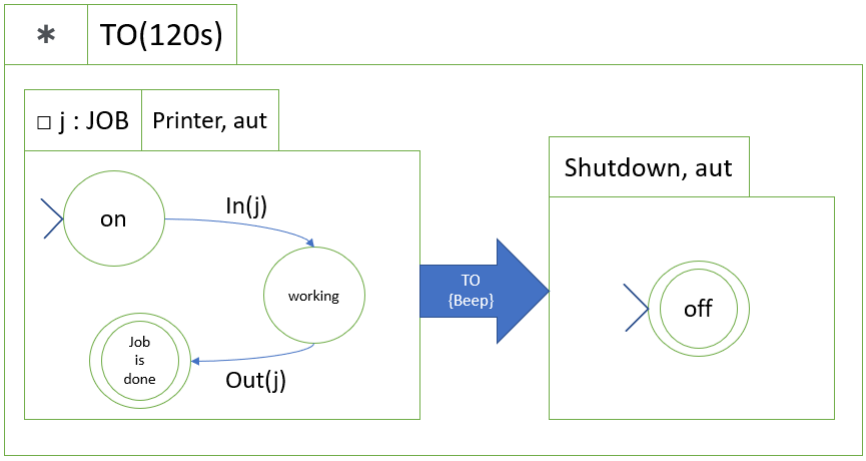

Figure 18. Example - Timeout

The initial and the final states are defined as follows. Let a be a timeout TASTD.

$$\begin{aligned} init(a, \mathsf{ts}, E) &\triangleq (\mathsf{Timeout}_\circ, a.E_{init}(\llbracket E \rrbracket), \mathsf{fst}, init(a.fst, \mathsf{ts}, a.E_{init}(\llbracket E \rrbracket)), \mathsf{false}) \\ final(a, (\mathsf{Timeout}_\circ, E, \mathsf{fst}, s, \_)) &\triangleq final(a.fst, s) \\ final(a, (\mathsf{Timeout}_\circ, E, \mathsf{snd}, s, \_)) &\triangleq final(a.\mathsf{snd}, s) \end{aligned}$$

The initial state of a timeout initialises its body with the variables of the timeout and the environment. The final state of a timeout depends on its timeout state: if its timeout state is in the first sub-ASTD, then it is final if the first sub-ASTD is final; if its timeout state is in the second sub-ASTD, then it is final if the second sub-ASTD is final.

4.6.2 Semantics

We define the semantics of timeout execution with five rules. $\mathsf{Timeout}_1$ allows for the execution of the first event in the first sub-TASTD. $\mathsf{Timeout}_2$ allows for the execution of the first sub-TASTD after the first event. $\mathsf{Timeout}_3$ allows for the timeout to happen when the event Step occurs after d time units. $\mathsf{Timeout}_4$ allows for the timeout to happen when the event σ from the second TASTD happen after d time units. $\mathsf{Timeout}_5$ allows for the execution of the second sub-TASTD after the timeout.

$$\textsf{Timeout}_1 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E_g''}_{a.fst} s' \qquad \Theta \qquad \mathsf{cst}-\mathsf{t} \leq d}{(\textsf{Timeout}_\circ, E, \mathsf{fst}, s, \mathsf{false}) \xrightarrow{\sigma,\mathsf{t},E_e,E_e'}_{a} (\textsf{Timeout}_\circ, E', \mathsf{fst}, s', \mathsf{true})}$$

$$\textsf{Timeout}_2 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E_g''}_{a.fst} s' \qquad \Theta}{(\textsf{Timeout}_\circ, E, \mathsf{fst}, s, \mathsf{true}) \xrightarrow{\sigma,\mathsf{t},E_e,E_e'}_{a} (\textsf{Timeout}_\circ, E', \mathsf{fst}, s', \mathsf{true})}$$

$$\textsf{Timeout}_3 \frac{\mathsf{cst}-\mathsf{t} > d \qquad \Theta_{TO}}{(\textsf{Timeout}_\circ, E, \mathsf{fst}, s, \mathsf{false}) \xrightarrow{\mathsf{Step},\mathsf{t},E_e,E_e'}_{a} (\textsf{Timeout}_\circ, E', \mathsf{snd}, init(a.\mathsf{snd}, \mathsf{cst}), \mathsf{true})}$$

$$\textsf{Timeout}_4 \frac{\mathsf{cst}-\mathsf{t} > d \qquad \Omega_{Timeout} \qquad init(a.\mathsf{snd}, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E_g''',E_g''}_{a.\mathsf{snd}} s' \qquad \Theta}{(\textsf{Timeout}_\circ, E, \mathsf{fst}, s, \mathsf{false}) \xrightarrow{\sigma,\mathsf{t},E_e,E_e'}_{a} (\textsf{Timeout}_\circ, E', \mathsf{snd}, s', \mathsf{true})}$$

$$\textsf{Timeout}_5 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E_g''}_{a.\mathsf{snd}} s' \qquad \Theta}{(\textsf{Timeout}_\circ, E, \mathsf{snd}, s, \mathsf{true}) \xrightarrow{\sigma,\mathsf{t},E_e,E_e'}_{a} (\textsf{Timeout}_\circ, E', \mathsf{snd}, s', \mathsf{true})}$$

$$\Theta_{TO} \triangleq \left\{ \begin{array}{l} E_g = E_e \lhd E \\ a.A_{TO}(E_g, E_g'') \\ a.A_{astd}(E_g'', E_g') \\ E_e' = E_e \lhd (V \lhd E_g') \\ E' = V \lhd E_g' \end{array} \right\}$$

$$\Omega_{Timeout} \triangleq a.A_{TO}(E_g, E_g''')$$

4.6.3 Example

The printer example, shown in Figure 18, describes a printer that the initial state is *on* inside the *Printer* automaton.

When the printer receives a work *j*, the event *in* is executed and the active state becomes *working*. Once the printer finishes the work, the *out* event happens and the active state becomes *job is done*.

After 300 seconds with *job is done* as the active state, the timeout shall occur when a *step* event arrives. Which also may occurs when the Printer TASTD is initialized and don't receive an *in* event. When a timeout occurs, the printer emits a *beep* and turns off.

The interaction of Kleene closure ASTD and timeout TASTD allows for the printer to turn off in the state *job is done*, even if it means that an event was executed in the first sub-TASTD of the timeout. Such interaction happens because *step* event can fire the Kleene closure rule to restart the Printer TASTD and timeout the Printer.

From a semantic point of view, when the TASTD is on *job is done*, it shall have the following configuration:

- Timed Automaton (Printer) state: ($\textsf{aut}_\circ$, "job is done", E, h, "job is done")
- Quantified Choice state: ($|:_\circ, j, E$, "job is done")
- Timeout state: ($\textsf{Timeout}_\circ, j, E$, "job is done")

- Kleene closure state: $(\star_\circ, E, \mathsf{true}, \text{“job is done”})$

At this point, receiving *step* event after 300 seconds will fire the first Kleene closure rule ($\star_1$). $init(KC, c) = (\star_\circ, E, \mathsf{true}, \text{“job is done”})$

$$\star_1 \frac{\mathsf{true} \qquad init(\mathrm{TO}, c) \xrightarrow{\mathsf{Step},\mathsf{t},E_g,E''_g}_{TO} off \qquad \Theta}{(\star_\circ, E, \mathsf{true}, \text{“job is done”}) \xrightarrow{\mathsf{Step},\mathsf{t},E_e,E'_e}_{a} (\star_\circ, E', \mathsf{true}, off)}$$

The Kleene closure is going to initialize it sub-TASTD that is the Timeout TASTD. The initialization of the Timeout TASTDhas the following configuration:

$$init(TO, \mathsf{t}) = (\mathsf{Timeout}_\circ, \bot, \mathsf{fst}, init(QChoice, \mathsf{t}), \mathsf{false})$$

The Timeout TASTD initialization also initializes the quantified choice ASTD.

$$init(QC, \mathsf{t}) = (|:_\circ, \bot, \bot, \bot)$$

At this point, we can fire the third rule of the timeout TASTD ($\mathsf{Timeout}_3$).

$$\mathsf{Timeout}_3 \frac{\mathsf{cst} - \mathsf{t} > 300 \qquad \Theta_{TO}}{(\mathsf{Timeout}_\circ, \bot, \mathsf{fst}, (|:_\circ, \bot, \bot, \bot), \mathsf{false}) \xrightarrow{\mathsf{Step},\mathsf{t},E_e,E'_e}_{a} (\mathsf{Timeout}_\circ, \bot, \mathsf{snd}, init(\mathrm{Shutdown}, \mathsf{cst}), \mathsf{true})}$$

By the third rule the *Shutdown* TASTD is initialized. So the timeout happens and we have the *beep* from Θ_{TO}. One may ask, what happens with the quantified choice and the timed automaton Printer. They are going to be in a deadlock state where the Timed automaton is not reinitialized by the quantified choice, which shall remain in it initial state.

4.6.4 Equivalence between generic Timeout TASTD and generic ASTD

Generic Timeout TASTD shown in Figure 19, is equivalent to the structure of a generic ASTD with an interruption and a choice ASTD types, shown in Figure 20.

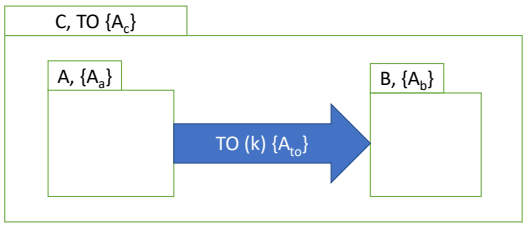

Figure 19. Generic Timeout TASTD

The two ASTDs are equivalent since the generic ASTD can simulate every rule of the Timeout TASTD. ASTDs A and B are abstractions and does not have their states, transitions, and actions represented, it does not change the equivalence.

- Rules $\mathsf{Timeout}_1$ and $\mathsf{Timeout}_2$ is going to execute ASTD A, what is equivalent to executing ASTD A of the generic ASTD.

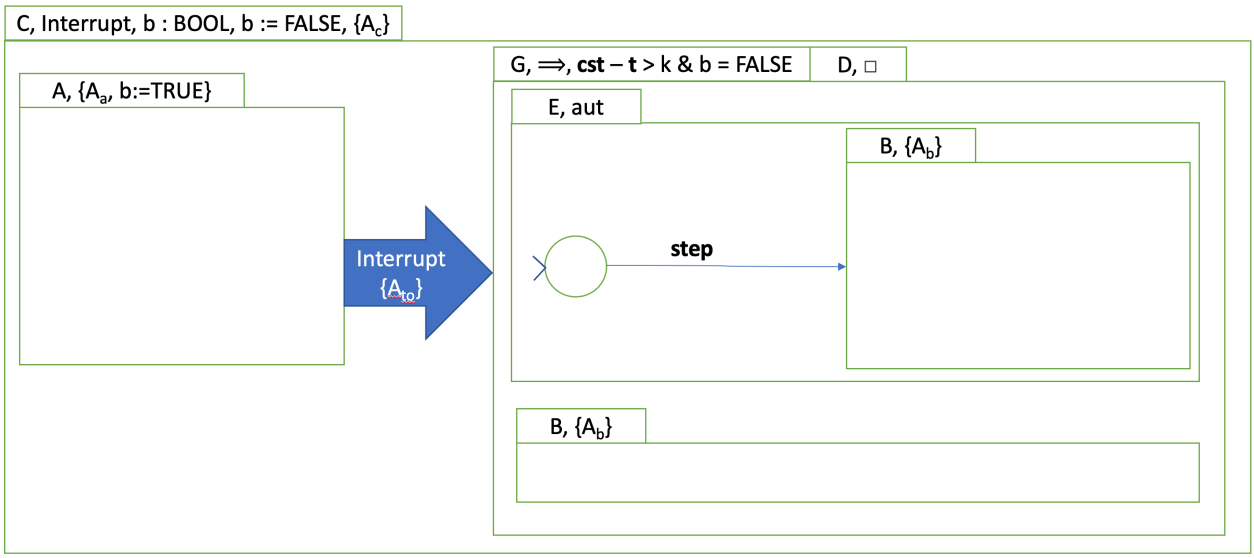

Figure 20. Equivalent generic ASTD

- Rule $\mathsf{Timeout}_1$ allows for the execution of the first event of ASTD A, which in the generic will trigger the ASTD A action that assigns b to true.
- Rule $\mathsf{Timeout}_2$ allows for the execution of the subsequent events from the ASTD A.

- Rules $\mathsf{Timeout}_3$ and $\mathsf{Timeout}_4$ are going to execute the timeout, the timeout action, while $\mathsf{Timeout}_4$ also execute ASTD B. This behaviour is captured by the interruption, the guard, and the choice, ASTDs C, G, and D.
  - Rule $\mathsf{Timeout}_3$ is interpreted by the choice of ASTD E. When *step* on E is executed it is similar to executing the *step*, the timeout action and the initialization of ASTD B.
  - Rule $\mathsf{Timeout}_4$ is interpreted by the choice of ASTD B. It is possible due the guard, so the first execution of ASTD B has to hold the guard from ASTD G.

- Rule $\mathsf{Timeout}_5$ allows for the execution of ASTD B.

4.7 Persistent Timeout

Persistent timeout TASTD subtype allows for verifying if the active state of a TASTD receives an event in a period of time, if not, then another TASTD has the right to execute. In the persistent timeout TASTD subtype, the active state of the first TASTD has d time units to receive an event for each event.

4.7.1 Syntax

Persistent timeout TASTD subtype has the following structure:

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

where $\mathsf{fst}, \mathsf{snd} \in \mathsf{TASTD}$ are the sub-TASTDs, d represent the time units necessary for a timeout, and A_{PTO} is an optional action executed when the timeout is triggered.

A persistent timeout state is of type $\langle \mathsf{PTimeout}_\circ, E, [\mathsf{fst}|\mathsf{snd}], s\rangle$ where $\mathsf{PTimeout}_\circ$ is a constructor of the persistent timeout state, E represent the value of attributes declared in the persistent timeout, $[\mathsf{fst}|\mathsf{snd}]$ represent a choice between two markers which respectively indicate whether the persistent timeout is in the first sub-TASTD or the second sub-TASTD, $s \in State$.

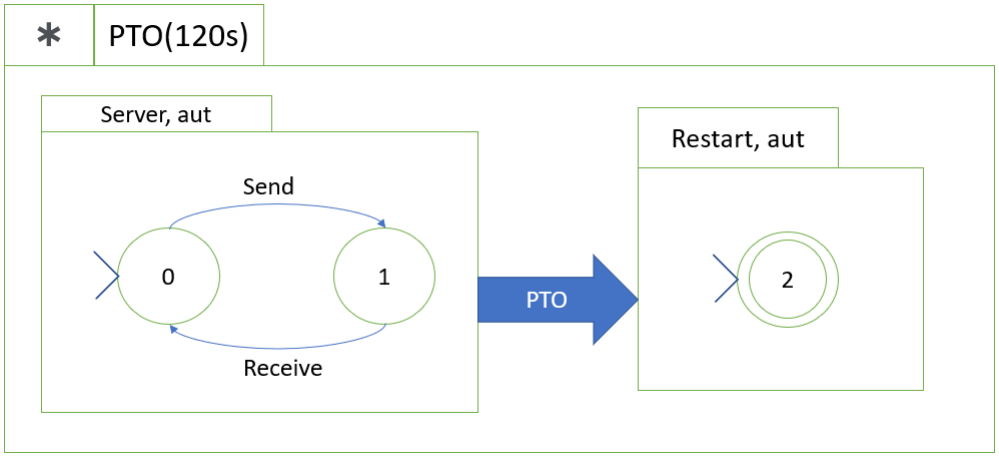

Figure 21. Example - Persistent timeout

The initial and the final states are defined as follows. Let a be a persistent timeout TASTD.

$$\begin{aligned} init(a, \mathsf{ts}, E) &\triangleq (\mathsf{PTimeout}_\circ, a.E_{init}([\![E]\!]), \mathsf{fst}, init(a.fst, \mathsf{ts}, a.E_{init}([\![E]\!]))) \\ final(a, (\mathsf{PTimeout}_\circ, E, \mathsf{fst}, s)) &\triangleq final(a.fst, s) \\ final(a, (\mathsf{PTimeout}_\circ, E, \mathsf{snd}, s)) &\triangleq final(a.\mathsf{snd}, s) \end{aligned}$$

The initial state of a persistent timeout initialises its body with the variables of the persistent timeout and the environment. The final state of a persistent timeout depends on its persistent timeout state: if its persistent timeout state is in the first sub-ASTD, then it is final if the first sub-ASTD is final; if its persistent timeout state is in the second sub-ASTD, then it is final if the second sub-ASTD is final.

4.7.2 Semantics

We define the semantics of persistent timeout execution with four rules. $\mathsf{PTimeout}_1$ allows for the execution of the first sub-TASTD. $\mathsf{PTimeout}_2$ allows for the timeout execution when the event Step occurs after d time units. $\mathsf{PTimeout}_3$ allows for the timeout execution when the event σ from the second sub-TASTD happen after d time units. $\mathsf{PTimeout}_4$ allows for the execution of the second sub-TASTD after the timeout.

$$\mathsf{PTimeout}_1 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.fst} s' \qquad \Theta \qquad \mathsf{cst} - \mathsf{t} \leq d}{(\mathsf{PTimeout}_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{PTimeout}_\circ, E', \mathsf{fst}, s')}$$

$$\mathsf{PTimeout}_2 \frac{\mathsf{cst} - \mathsf{t} > d \qquad \Theta_{PTO}}{(\mathsf{PTimeout}_\circ, E, \mathsf{fst}, s) \xrightarrow{\mathsf{Step},\mathsf{t},E_e,E'_e}_{a} (\mathsf{PTimeout}_\circ, E', \mathsf{snd}, init(a.\mathsf{snd}, \mathsf{cst}))}$$

$$\mathsf{PTimeout}_3 \frac{\mathsf{cst} - \mathsf{t} > d \qquad \Omega_{PTimeout} \qquad init(a.\mathsf{snd}, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E'''_g,E''_g}_{a.\mathsf{snd}} s' \qquad \Theta}{(\mathsf{PTimeout}_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{PTimeout}_\circ, E', \mathsf{snd}, s')}$$

$$\mathsf{PTimeout}_4 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.\mathsf{snd}} s' \qquad \Theta}{(\mathsf{PTimeout}_\circ, E, \mathsf{snd}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{PTimeout}_\circ, E', \mathsf{snd}, s')}$$

$$\Theta_{PTO} \triangleq \left\{ \begin{array}{l} E_g = E_e \lhd E \\ a.A_{PTO}(E_g, E''_g) \\ a.A_{astd}(E''_g, E'_g) \\ E'_e = E_e \lhd (V \lhd E'_g) \\ E' = V \lhd E'_g \end{array} \right\}$$

$$\Omega_{PTimeout} \triangleq a.A_{PTO}(E_g, E'''_g)$$

4.7.3 Example

The jammed server example, shown in 21, describes a server that can only restart when it stops for more than 300 seconds without an event.

When the server starts, it shall communicate through *send* and then wait for a response to come with *receive*.

If the server sent or received a message and after 300 seconds there is no communication, the server may be jammed and it may restart once a *step* event occurs. The *step* events allows for the timeout to happens. After a timeout, the server can retry to communicate with a send due the Kleene closure.

4.7.4 Equivalence between generic Persistent Timeout and generic ASTD

Generic Persistent Timeout TASTD shown in Figure 22, is equivalent to the structure of a generic ASTD with an interruption and a choice ASTD types, shown in Figure 23. ASTDs A and B are abstractions and does not have their states, transitions, and actions represented, it does not change the equivalence.

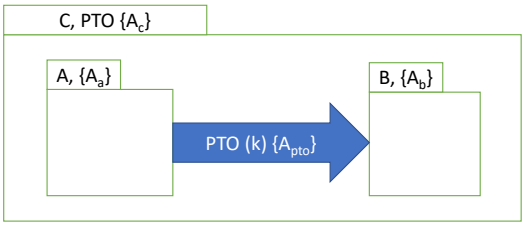

Figure 22. Generic Persistent Timeout TASTD

The two ASTDs are equivalent since the generic ASTD can simulate every rule of the Persistent Timeout TASTD. ASTDs A and B are abstractions and does not have their states, transitions, and actions represented, it does not change the equivalence.

- Rule $\mathsf{PTimeout}_1$ is going to execute ASTD A, what is equivalent to executing ASTD A of the generic ASTD.
  - Rule $\mathsf{PTimeout}_1$ allows for the execution of ASTD A.
- Rules $\mathsf{PTimeout}_2$ and $\mathsf{PTimeout}_3$ are going to execute the timeout, and the persistent timeout action. This behaviour is captured by the interruption, the guard and the choice, ASTDs C, G, and D.

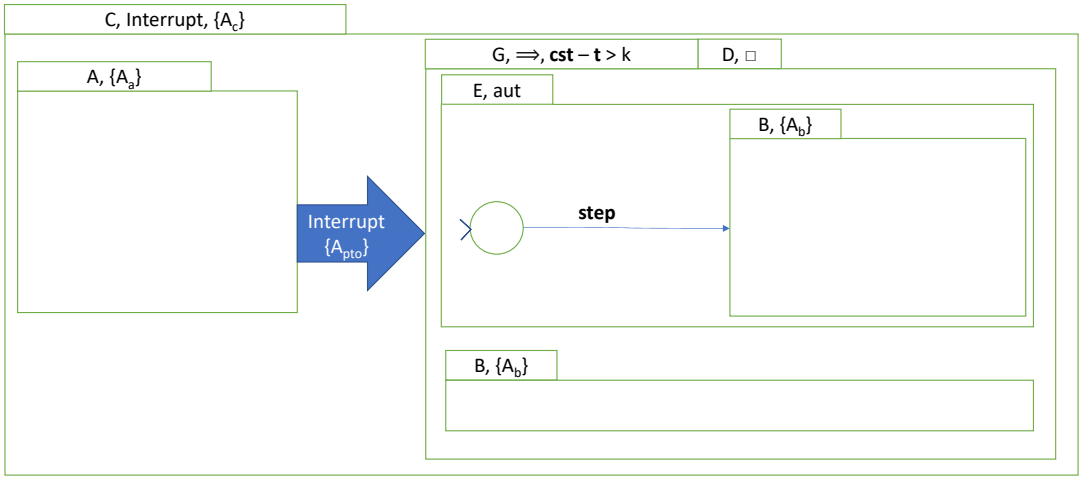

Figure 23. Equivalent generic ASTD

- Rule $\mathsf{PTimeout}_2$ is interpreted by the choice of ASTD E. When *step* on E is executed it is similar to executing the *step*, the timeout action and the initialization of ASTD B.
- Rule $\mathsf{PTimeout}_3$ is interpreted by the choice of ASTD B. It is possible due the guard, so the first execution of ASTD B has to hold the guard from ASTD G.

- Rule $\mathsf{PTimeout}_4$ allows for the execution of ASTD B.

4.8 Timed Interrupt

The timed interrupt TASTD subtype allows for an TASTD to execute until a determined time is reached, then another TASTD has the right to execute.

4.8.1 Syntax

The timed interrupt TASTD subtype has the following structure:

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

where $\mathsf{fst}, \mathsf{snd} \in$ TASTD are the sub-TASTDs, d represent the time units necessary for an interruption, and A_{TI} is an optional action executed when the interruption is triggered.

A timed interrupt state is of type $\langle \mathsf{TInterrupt}_\circ, E, [\mathsf{fst}|\mathsf{snd}], s, di \rangle$, where $\mathsf{TInterrupt}_\circ$ represent a constructor of the timed interrupt state, E the value of the attributes declared in the timed interrupt, $[\mathsf{fst}|\mathsf{snd}]$ is a choice between the two markers that respectively indicate whether the interruption is in the first sub-TASTD or the second sub-TASTD, $s \in State$ and di is a real value greater than 0.

Let a be a timed interrupt TASTD.

$$\begin{aligned}
init(a, \mathsf{ts}, E) &\triangleq (\mathsf{TInterrupt}_\circ, a.E_{init}([\![E]\!]), \mathsf{fst}, init(a.\mathsf{fst}, \mathsf{ts}, a.E_{init}([\![E]\!])), \mathsf{ts}) \\
final(a, (\mathsf{TInterrupt}_\circ, E, \mathsf{fst}, s, \mathsf{ts})) &\triangleq final(a.\mathsf{fst}, s) \\
final(a, (\mathsf{TInterrupt}_\circ, E, \mathsf{snd}, s, \mathsf{ts})) &\triangleq final(a.\mathsf{snd}, s)
\end{aligned}$$

The initial state of a timed interrupt initialises its body with the variables of the timed interrupt and the environment. The final state of a timed interrupt depends on its timed interrupt state: if its timed

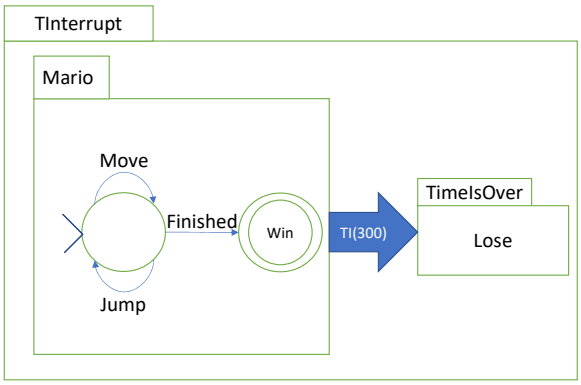

Figure 24. Example - timed interrupt

interrupt state is in the first sub-ASTD, then it is final if the first sub-ASTD is final; if its timed interrupt state is in the second sub-ASTD, then it is final if the second sub-ASTD is final.

4.8.2 Semantics

We define the semantics of timed interruption execution with four rules. $\mathsf{TInterrupt}_1$ allows for the execution of the first sub-TASTD. $\mathsf{TInterrupt}_2$ allows for the interruption execution when the event Step occurs after d time units. $\mathsf{TInterrupt}_3$ allows for the interruption execution when the event σ from the second sub-TASTD happen after d time units. $\mathsf{TInterrupt}_4$ allows for the execution of the second sub-TASTD after the interruption.

$$\mathsf{TInterrupt}_1 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.\mathsf{fst}} s' \qquad \Theta \qquad \mathsf{cst} - \mathsf{ts} \leq d}{(\mathsf{TInterrupt}_\circ, E, \mathsf{fst}, s, \mathsf{ts}) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{TInterrupt}_\circ, E', \mathsf{fst}, s', \mathsf{ts})}$$

$$\mathsf{TInterrupt}_2 \frac{\mathsf{cst} - \mathsf{ts} > d \qquad \Theta_{TInterrupt}}{(\mathsf{TInterrupt}_\circ, E, \mathsf{fst}, s, \mathsf{ts}) \xrightarrow{\mathsf{Step},\mathsf{t},,E_e,E'_e}_{a} (\mathsf{TInterrupt}_\circ, E', \mathsf{snd}, init(a.\mathsf{snd}, \mathsf{cst}), \mathsf{cst})}$$

$$\mathsf{TInterrupt}_3 \frac{\mathsf{cst} - \mathsf{ts} > d \qquad \Omega_{TInterrupt} \qquad init(a.\mathsf{snd}, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E'_g,E''_g}_{a.\mathsf{snd}} s' \qquad \Theta}{(\mathsf{TInterrupt}_\circ, E, \mathsf{fst}, s, \mathsf{ts}) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{TInterrupt}_\circ, E', \mathsf{snd}, s', \mathsf{cst})}$$

$$\mathsf{TInterrupt}_4 \frac{s \xrightarrow{\sigma,\mathsf{t},E_g,E''_g}_{a.\mathsf{snd}} s' \qquad \Theta}{(\mathsf{TInterrupt}_\circ, E, \mathsf{snd}, s, \mathsf{ts}) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (\mathsf{TInterrupt}_\circ, E', \mathsf{snd}, s', \mathsf{cst})}$$

$$\Theta_{TInterrupt} \triangleq \left\{ \begin{array}{l} E_g = E_e \lhd\!\!+ E \\ a.A_{TInt}(E_g, E''_g) \\ a.A_{astd}(E''_g, E'_g) \\ E'_e = E_e \lhd\!\!+ (V \lhd E'_g) \\ E' = V \lhd E'_g \end{array} \right\}$$

$$\Omega_{TInterrupt} \triangleq a.A_{TO}(E_g, E'''_g)$$

4.8.3 Example

Figure 24 presents an illustration of a timed interrupt. On the TASTD, Mario has 300 time units to finish and reach the *win* state. If 300 seconds has passed, the time is over and the active TASTD becomes Lose, which is abstract.

4.8.4 Equivalence between generic Timed Interrupt and generic ASTD

Generic Timed Interrupt TASTD shown in Figure 25, is equivalent to the structure of a generic ASTD with an interruption and a choice ASTD types, shown in Figure 26.

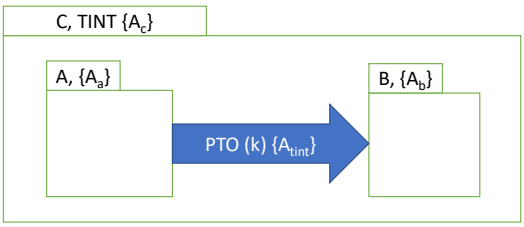

Figure 25. Generic Timed Interrupt TASTD

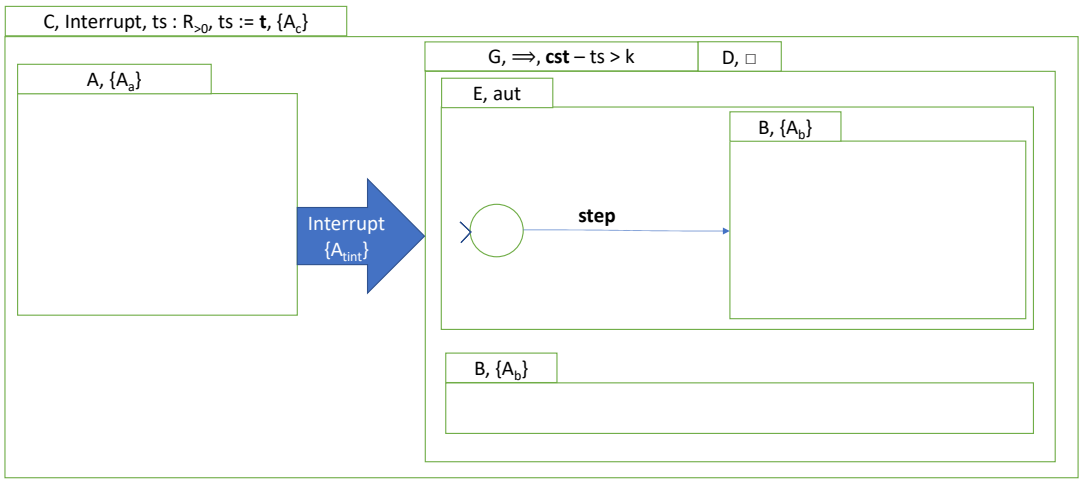

Figure 26. Equivalent generic ASTD

The two ASTDs are equivalent since the generic ASTD can simulate every rule of the Timed Interrupt TASTD. ASTDs A and B are abstractions and does not have their states, transitions, and actions represented, it does not change the equivalence.

- Rule $\mathsf{TInterrupt}_1$ is going to execute ASTD A.
  - Rule $\mathsf{TInterrupt}_1$ allows for the execution of ASTD A.
- Rules $\mathsf{TInterrupt}_2$ and $\mathsf{TInterrupt}_3$ are going interrupt ASTD A and execute the timed interruption action. This behaviour is captured by the interruption, the guard and the choice, ASTDs C, G, and D.

– Rule $\mathsf{TInterrupt}_2$ is interpreted by the choice of ASTD E. When *step* on E is executed it is similar to executing the *step* event, the timed interruption action and the initialization of ASTD B.

– Rule $\mathsf{TInterrupt}_3$ is interpreted by the choice of ASTD B. It is possible due the guard, so the first execution of ASTD B has to hold the guard from ASTD G.

- Rule $\mathsf{TInterrupt}_4$ allows for the execution of ASTD B.

4.9 General information

4.9.1 Interruption and an automaton with history state.

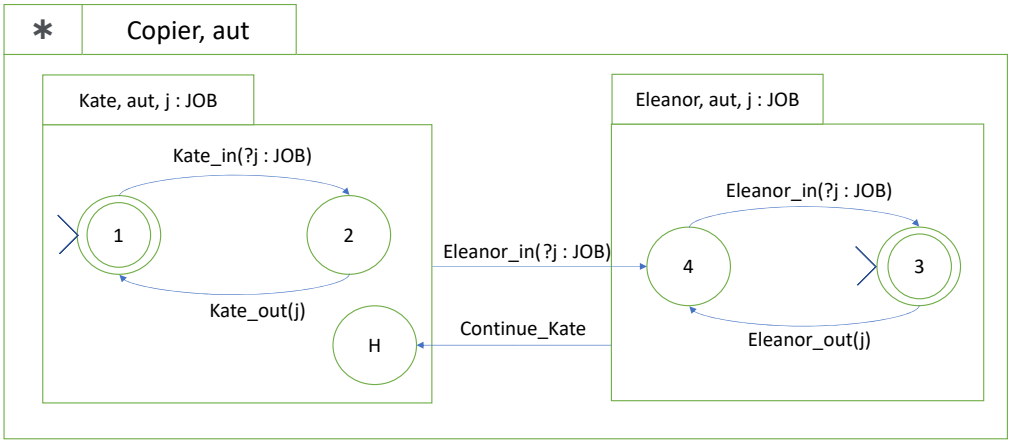

Figure 27. Example - Copier Timed Automata

An automaton with history state, shown in Figure 27, is capable of returning to a Kate ASTD. While interruption ASTD does not return to the interrupted ASTD by any means other than reinitializing the interruption ASTD.

The *alternative* Copier ASTD, presented in 27, is capable of returning to *Kate* automaton and continue to operate where it was before going to Eleanor ASTD because of the history state and *Continue_Kate* event. This behaviour is not applied in 13.

4.9.2 Two timestamps in parametrized synchronization, quantified synchronization and flow.

Parameterized synchronization, quantified synchronization and flow ASTDs, use two timestamps that are unsynchronised, which means they can have different values. Two timestamps are necessary because acting in the first sub-ASTD does not mean an act to the second sub-ASTD. That way, each have two separated timestamps that tracks the last action of their sub-ASTD.

An exemplification of that matter is shown in Figure 28.

In this example there is a parametrized synchronization, synchronized over *e3*. The two sub-ASTDs are two persistent timeout. If 10 seconds has passed without any other event execution and a *step* occurs, TASTD *A* will timeout and the right of execution goes to TASTD *B*. Similarly, if 10 seconds has passed without any event execution and a *step* occurs, TASTD *C* will timeout and the right of execution goes to TASTD *D*.

To execute event *e3*, TASTD *C* and *B* need to be active. To achieve that state, the first sub-ASTD of the synchronization *Sync* needs to timeout.

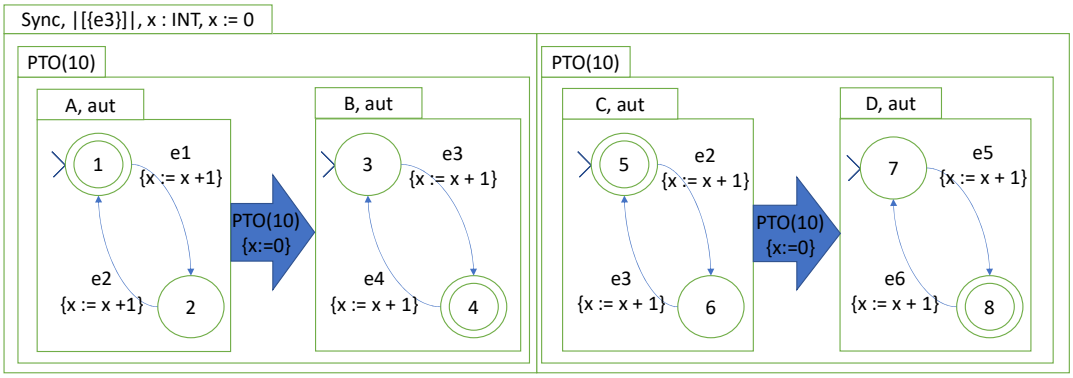

Figure 28. Example - Timeout synchronization

If the parametrized synchronization had only one timestamp, then it would be impossible to timeout A without timing out C. In result, *e3* would not be executed.

Additionally, receiving an instruction for the first sub-ASTD shall not assign *cst* to the second sub-ASTD timestamp since it didn't received an event.

To finish, the order of executed actions of a timeout depend on which rule is applied. If $\mathsf{PTimeout}_2$ happens, then the timeout action occur and the initialization of the second timeout TASTD. When $\mathsf{PTimeout}_3$ happens, then the timeout action occur, the initialization of the second timeout TASTD and for last the action of the transition.

Both rules are available with the following trace:

1. With the initialization of the TASTD on Figure 28, the active states are *1* and *5*.
2. After 5 seconds event *e2* occurs, the state of C becomes *6* and the variable x becomes 1.
3. After more 5 seconds, two rules are available. If *step* occurs, then $\mathsf{PTimeout}_2$ is executed and x becomes 0. If *e3* happens, then two possibilities arise: to execute *e3* on C and then *e3* on B through the timeout or to execute B and then C.
    - With B then C: x becomes 2.
    - With C then B: x becomes 1.

Another interesting interaction is when there **isn't** a synchronization on *e3*, then following the previous trace until the third point. Once *e3* occur, or the timeout may be executed and then *e3*, or C is executed.

- With $\mathsf{PTimeout}_3$, the timeout happens, x becomes 0, *e3* executes and x become 1 with active states *4* and *5*.
- Executing *e3* on C, x becomes 2 with *1* and *5*. What means that even with 10 seconds the timeout still not occured, and shall occur with a *step* or another *e3* event.

4.9.3 *Step* acting as a flow

As an alternative behaviour to event *Step*, we propose that *Step* act as a flow in TASTDs with parallelism. That means, every available *Step* is executed once a *Step* is received. To make this be possible, parallel ASTD have included inference rules that fire only with *Step* and exclude *Step* from the other rules. A summary of the changes are presented below:

1. For a *Step* inside an unary ASTD, such as Kleene closure, Call, Guard, Delay, and Persistent Delay, there is no change.

2. For a *Step* inside a binary ASTD without parallelism, such as Sequence, Choice, Interrupt, Timeout, Persistent Timeout, and Timed Interrupt, there is no change.

3. For a *Step* inside a binary ASTD with parallelism, such as Parametrized synchronisation, and Flow, it acts as a flow excepts for a synchronisation over *Step*. If there is a synchronisation over *Step* than it acts as a synchronisation.

4. For a *Step* inside a quantified ASTD without parallelism, such as Quantified choice, there is no change.

5. For a step inside a quantified ASTD with parallelism, such as Quantified synchronisation, there are three scenarios:

   (a) Inside an enumerated domain *Step* acts as a flow over the whole domain. Except if there is a synchronisation over *Step*.

   (b) Inside an unbounded domain *Step* acts as a flow over the instantiated values. Except if there is a synchronisation over *Step*.

   (c) Inside a quantified ASTD with synchronisation over *Step*, *Step* acts as a synchronisation.

The rules for the ASTD types present in 1, 2, 4 does not change.

4.9.3.1 Parametrized Synchronisation

For a Parametrized Synchronisation, there are four inference rules. Rules $|[]|_1$ and $|[]|_2$ respectively describe execution of events, with no synchronization 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.

$$|[]|_1 \frac{\alpha(\sigma) \notin \Delta \qquad \sigma \neq \mathsf{Step} \qquad s_l \xrightarrow{\sigma,\mathsf{t}_l,E_g,E''_g}_{a.l} s'_l \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|[]|_\circ, E', s'_l, \mathsf{cst}, s_r, \mathsf{t}_r)}$$

Rule $|[]|_2$ is symmetric to $|[]|_1$ and indicates behaviour when the right side execute the action.

$$|[]|_2 \frac{\alpha(\sigma) \notin \Delta \qquad \sigma \neq \mathsf{Step} \qquad s_r \xrightarrow{\sigma,\mathsf{t}_r,E_g,E''_g}_{a.r} s'_r \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_{a} (|[]|_\circ, E', s_l, \mathsf{t}_l, s'_r, \mathsf{cst})}$$

The most interesting case is when the left and right sub-ASTDs must synchronize on an event (*i.e.,* when $\alpha(\sigma) \in \Delta$). Consider the transitions on the left and right sub-ASTDs when each of them is executed independently of the other.

$$\Omega_{ilr} \triangleq \left(s_l \xrightarrow{\sigma,\mathsf{t},E_g,E'_{gl}}_{a.l} s'_l \qquad s_r \xrightarrow{\sigma,\mathsf{t},E_g,E'_{gr}}_{a.r} s'_r \right)$$

Since shared variables (E_g) can be modified by both sub-ASTDs, their modifications could be inconsistent, which should forbid the synchronization transition. This requires to check that $E'_{gl} = E'_{gr}$. However, when one variable is modified by both sub-ASTDs, the natural intent is typically that the compound

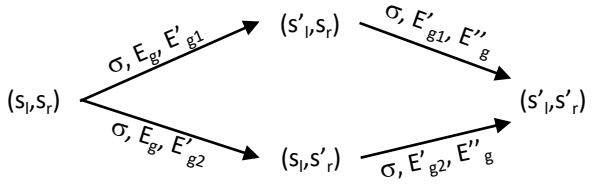

Figure 29. Commutativity of actions execution in a parameterized synchronization on σ

result of both sides is desired, that is, to assume that one side executes on the values returned by the other. For instance, assume that both sub-ASTDs increment shared variable x by 1 and assume that the before value of x is 0. The above semantics gives $x = 1$, whereas the compound result is $x = 2$. If one sub-ASTD increments by 1 and the other by 2, the above semantics forbids execution because the result is inconsistent, whereas the compound execution returns 3. Executing the left sub-ASTD before the right sub-ASTD is specified as follows.

$$\Omega_{lr} \triangleq \left(s_l \xrightarrow{\sigma,\mathsf{t},E_g,E'_{g1}}_{a.l} s'_l \qquad s_r \xrightarrow{\sigma,\mathsf{t},E'_{g1},E''_g}_{a.r} s'_r \right)$$

Executing the right sub-ASTD before the left sub-ASTD is specified as follows.

$$\Omega_{rl} \triangleq \left(s_r \xrightarrow{\sigma,\mathsf{t},E_g,E'_{g2}}_{a.r} s'_r \qquad s_l \xrightarrow{\sigma,\mathsf{t},E'_{g2},E''_g}_{a.l} s'_l \right)$$

Since synchronization should be commutative, *i.e.,* both execution orders should return the same states. Figure 29 illustrates this property.

Checking this commutativity at each transition is expensive. To improve performance, it could be statically checked using proof obligations, or by analysis of the variables read and written by each sub-ASTD, to ensure that the left and right sub-ASTDs are independent (*i.e.,* they do not modify the same variables and one sub-ASTD does not modify the variables read by the other). When the synchronization operands are nondeterministic, this is still expensive to execute, because a commutative combination must be found, which means enumerating combinations of possibilities from both sides. Still, this is our preferred semantics for a synchronization, because it works well for deterministic specifications. Here is the rule for synchronization.

$$|[]|_3 \frac{\alpha(\sigma) \in \Delta \qquad \Omega_{lr} \qquad \Omega_{rl} \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_a (|[]|_\circ, E', s'_l, \mathsf{cst}, s'_r, \mathsf{cst})}$$

When a Steparrives and there is no synchronisation over Step, every available ASTD shall execute Step, which is described in $|[]|_4$:

$$|[]|_4 \frac{\alpha(\mathsf{Step}) \notin \Delta \qquad \Omega_{lr} \qquad \Omega_{rl} \qquad \Theta}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\mathsf{Step},\mathsf{t},E_e,E'_e}_a (|[]|_\circ, E', s'_l, \mathsf{cst}, s'_r, \mathsf{cst})}$$

4.9.3.2 Quantified Synchronisation

Rule $|[]|:_1$ describe execution of events with no synchronization. Symbol c denotes the element of T chosen for the execution.

$$|[]|:_1 \frac{\alpha(\sigma) \notin \Delta \qquad \sigma \neq \mathsf{Step} \qquad f(d) \xrightarrow{\sigma,u(d),E_g \lhd \{x \mapsto d\},E''_g}_{a.b} s' \qquad \Theta}{(|[]|:_\circ, E, f, u) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_a (|[]|:_\circ, E', f \lhd \{d \mapsto s'\}, u \lhd \{d \mapsto \mathsf{cst}\})} 0$$

Rule $|[]|:_2$ describe execution of an event with synchronization. All elements of T must execute σ, in any order. It generalizes rule $|[]|_3$ and requires commutativity.

$$|[]|:_2 \frac{\alpha(\sigma) \in \Delta \qquad \Omega_{qsync} \qquad \Theta}{(|[]|:_\circ, E, f, u) \xrightarrow{\sigma,\mathsf{t},E_e,E'_e}_a (|[]|:_\circ, E', f', u \lhd (T * \mathsf{cst}))}$$

Premiss Ω_{qsync} formalizes commutativity by universally quantifying over all permutations p of T (noted $p \in \pi(T)$) and using Es as a sequence of environments storing the intermediate results of the computation of E''_g from E_g by iterating over the elements $p(i)$ of p. Let $k = |T|$.

$$\Omega_{qsync} \triangleq \left(\begin{array}{c} \forall p \in \pi(T) \cdot \exists Es \in 0..k \nrightarrow \mathsf{Env} \land Es(0) = E_g \land E(k) = E''_g \land \\ \forall i \in 1..k \cdot f(p(i)) \xrightarrow{\sigma,u(p(i)),Es(i-1) \lhd \{x \mapsto p(i)\},Es(i)}_{a.b} f'(p(i)) \end{array} \right)$$

Rule $|[]|:_3$ describe execution of an *step* that is not synchronised. All elements of T must execute Step, in any order. It generalizes rule $|[]|_4$ and requires commutativity.

$$|[]|:_3 \frac{\alpha(\mathsf{Step}) \notin \Delta \qquad \Omega_{qsync} \qquad \Theta}{(|[]|:_\circ, E, f, u) \xrightarrow{\mathsf{Step},\mathsf{t},E_e,E'_e}_a (|[]|:_\circ, E', f', u \lhd (T * \mathsf{cst}))}$$

5 Case Study

In this section, we present three case studies to demonstrate the usefulness, limitations and capabilities of TASTD. First case study is three RoboChart models, those models are translated into *tock-csp* in [9]. Second case study is a production line [10]. Third case study is Adaptive Exterior Light and Speed Control System from ABZ-2020 conference [11].

5.1 Robosim

Robosim is a language to model simulations of robotic systems by state machines combined to define concurrent and distributed designs that use specific services of a platform [9]. The goal of Robosim case study is to demonstrate that TASTD is capable of modeling another timed notation.

Robosim cannot be directly translated into TASTD. Two differences between Robosim and TASTD are junctions and transitions without a label. In TASTD junctions are not defined, although we can simulate them as a minimal state. In TASTD urgent transitions or transitions without a label cannot be simulated. To simulate urgent transitions present in Robosim models, we use event *Step*. So, in our TASTD model, states with urgent transitions will be active for the duration of a *Step*.

5.1.1 Square

This case study is a robot that performs a square trajectory, it avoids obstacles in his course. The TASTD model is presented in Figure 30.

The initial state is i and its transition leads to state *MovingForward*, in this transition the time stamp C is reset, it receives the value of the current system time, and the variable *segment* is assigned to 0; *segment* counts the number of sides that have been traversed.

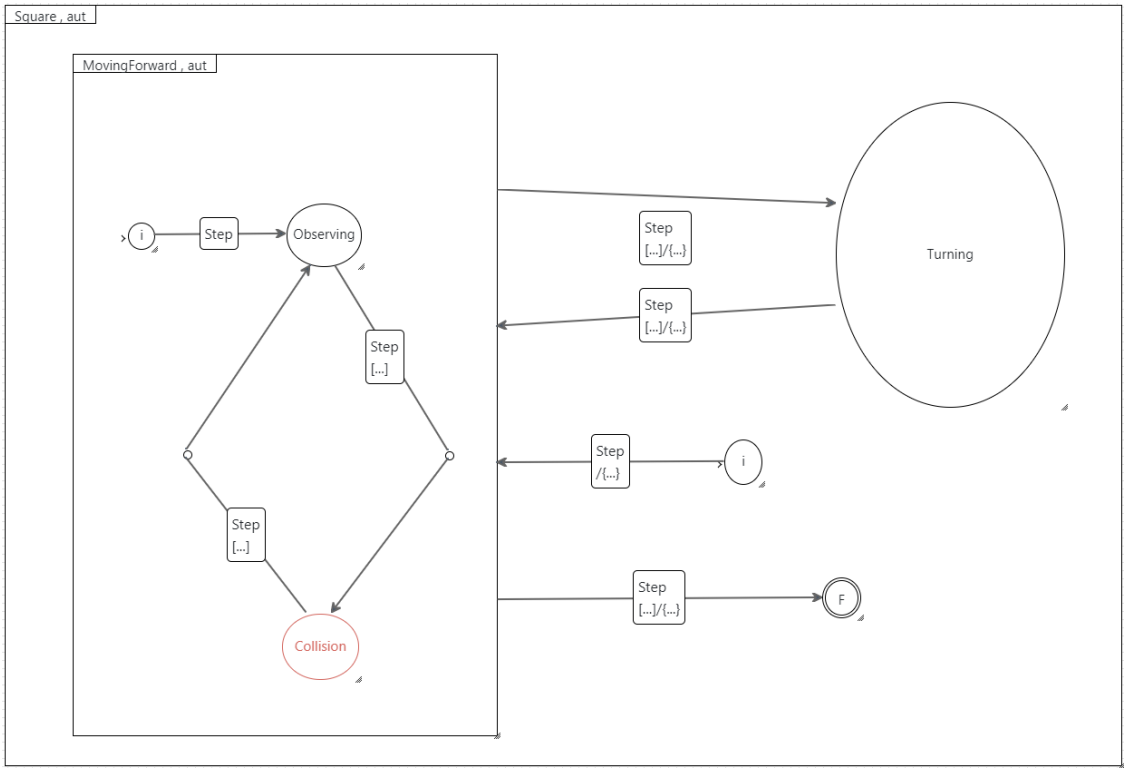

Figure 30. Case study - Square from RoboChart

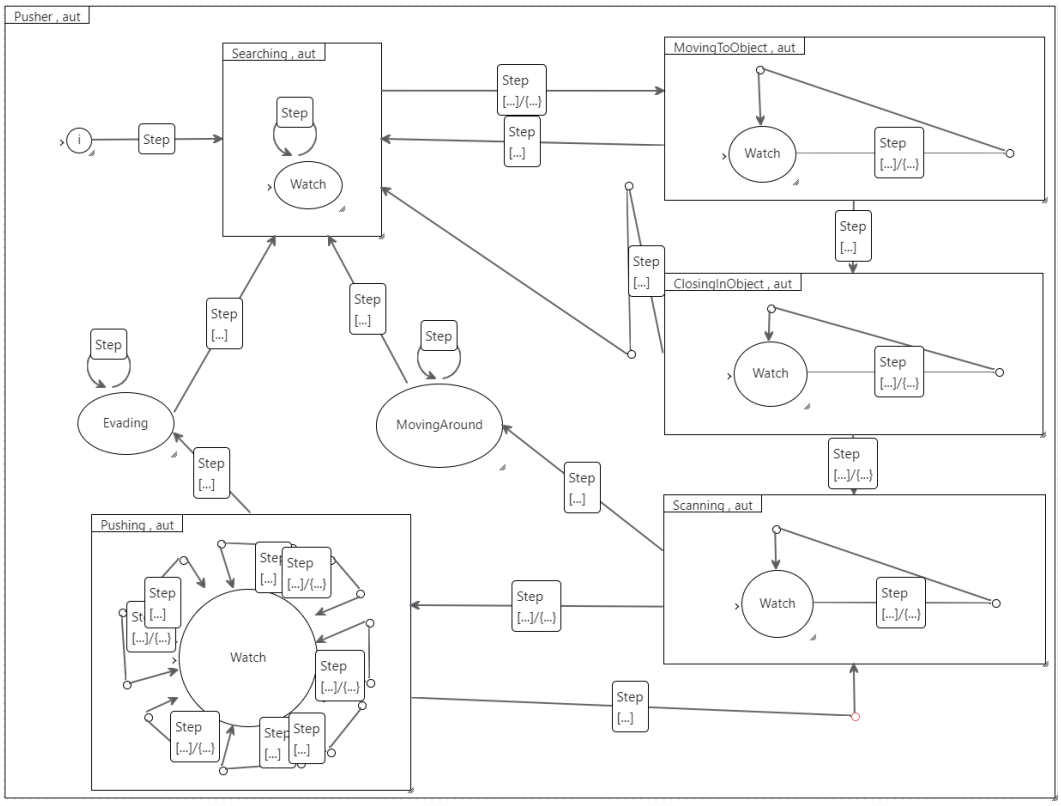

Figure 31. Case study - Transporter from RoboChart

MovingForward entry code executes a function so the robot starts moving forward. The initial state of *MovingForward* is *i*. From that state, three transitions are available, one to *Observing*, one to *F*, and one to *Turning*. Those transitions means three different behaviors of the robot.

- If the active state continues as *MovingForward* means that the robot is moving forward and observes it path to verify if it can find an obstacle. Once it finds an obstacle, it checks if the obstacle is farther than the end of segment, if not, state *Collision* avoids the collision.
- If the active state goes to *Turning* means that the robot moved all the distance for this segment of the square, and should turn to cover the next segment of the square.
- If the active state goes to F means that the robot moved all the four segments and stops.

TASTD is non deterministic, but the transition to *Turning* has the guard $C > 5$ and $segment < 4$, and the transition to F has guard $C > 5$ and $segment = 4$. So, these transitions can only happen in a specific scenario.

5.1.2 Transporter

The second case study from RoboSim is a swarm of robots for moving a large object to a specified goal. The robots moves the object by pushing it. The TASTD model is presented in Figure 31.

This TASTD models seven behaviors of a robot from the swarm:

1. *Searching* state searches for an object to be moved. Once the scan finds an object, event *Step* fires the transition to *MovingToObject*.

2. *MovingToObject* makes the robot rapidly move to the object, but has a threshold range where it should approach slower. When the robot reaches the range threshold, event *Step* fires to *ClosingInObject*. A time threshold is also defined for a case when the robot takes too much time to reach the object, then a *Step* launches a transition to *Searching*.

3. *ClosingInObject* is responsible to close the gap between object and robot, it handles that the robot does not crash into the object. When the robot is touching the object, a *Step* fires the transition to *Scanning*. A time threshold is also defined for a case when the robot takes too much time to reach the object, then a *Step* launches a transition to *Searching*.

4. *Scanning* scan the object to check if it reached the goal. If the object is not in the goal, then *Step* launches the transition to *Pushing*. Otherwise, *Step* launches a transition to *MovingAround*.

5. *Pushing* is responsible for pushing the object until the goal. After a time threshold, *Step* fires a transition to *Scanning* where the robot will check if the object reached the goal. With a second time threshold, *Step* launches a transition to *Evading* where the robot will evade the object.

6. *MovingAround* makes the robot move around the object after it reached the goal. After the robot moved around the object, *Step* fires a transition to *Searching*.

7. *Evading* makes the robot evade the object after it didn't reach the goal but reached a time threshold. After evading the object, *Step* launches a transition to *Searching*.

5.1.3 Alpha Algorithm

The third case study from RoboSim is a single robot in a swarm acting under Alpha Algorithm. The TASTD model is presented in Figure 32.

The initial state is *i* inside *Movement* automaton, with a *Step* the active state becomes state *i* inside *MovementAndAvoidance* automaton.
MovementAndAvoidance automaton is responsible for moving the robot forward and avoid collisions. *MovementAndAvoidance* has a entry code that resets clock *MBD*. From *i* there are two allowed transitions, one *Step* that transits to *Move*, and one *Step* that changes the active state to *i* from *Turning* automaton, when more than *MB* time units is passed since entry in *MovementAndAvoidance* automaton, and the active state is final.
Move has an entry action that makes the robot move forward. A delay of one time unit over event *Step* guarantees that this state is active for at least one second before moving to *Turning* or *Avoid*.
Avoid is responsible for avoiding obstacles on the robot's route. *Avoid* has an entry code that makes the robot avoid an obstacle. This action takes at least $\text{floor}(\text{rand}() * 360/\text{av}))$ time units. Once the obstacle is avoided, then the active state can change to *Move* or *Turning*.
Turning automaton is responsible for turning the robot, once the robot has turned the active state goes to *MovementAndAvoidance*. *Turning* has an entry code which assigns *false* to variable *turned*. *Turning* initial state is *i*, with a *Step* it can move to *RandomTurn*, where the robot turns randomly, or *Turn180*, where the robot turns 180 degrees. Once the turn is finished, a *Step* assigns *true* to *turned* and *Turning* reaches its final state *F*. From *F*, the robot can start moving forward again once *MovementAndAvoidance* becomes active.

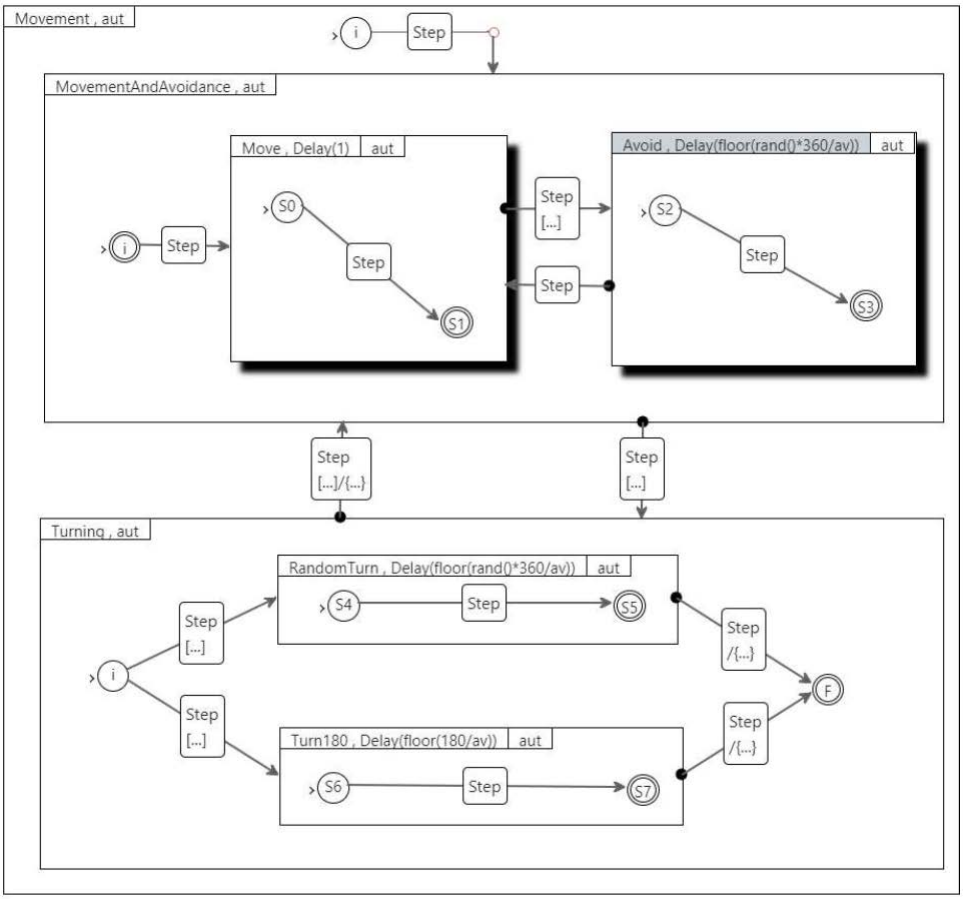

Figure 32. Case study - Alpha Algorithm from RoboChart

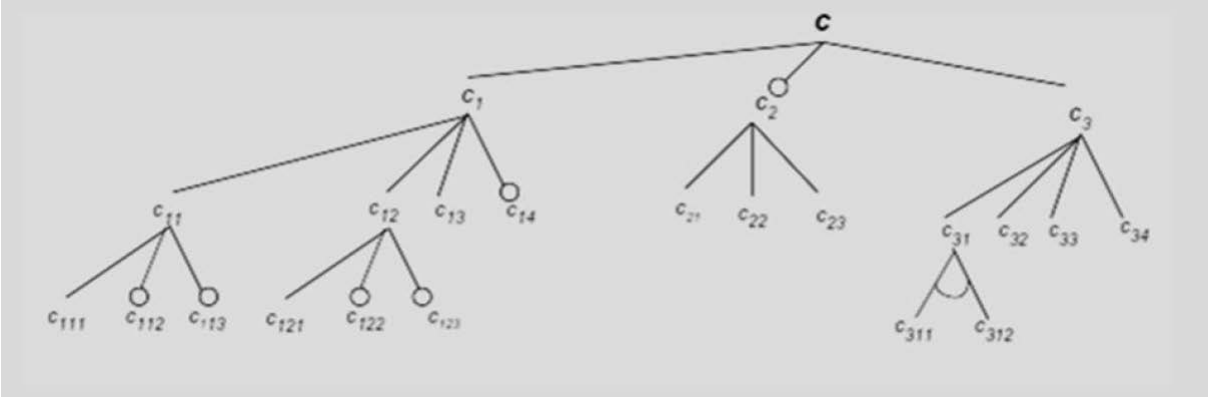

Figure 33. Case study - Functional perspective of real estate services and urban sanitizing model in FORM/BCS retired from [10]

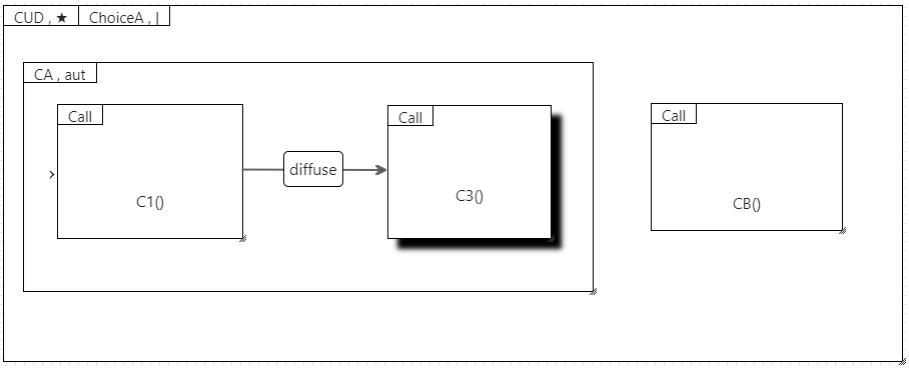

Figure 34. Case study FORM/BCS - CUD (Main ASTD)

5.2 Feature-Oriented Reuse Method with Business Component Semantics

Second case study is a functional perspective of real estate services and urban sanitizing model in FORM/BCS (Feature-Oriented Reuse Method with Business Component Semantics) [10]. FORM is a systematic method that looks for and captures commonalities and variability of a product line in terms of "features" [17].

The structure of the model is represented by Figure 33. Figure 33 presents a tree structure where the context C is divided in three other contexts C_1, C_2 and C_3. As shown in Figure 33, C_2 has a empty circle that connects it to C, what means that C_2 is optional. C_1 is divided in four contexts, C_{11}, C_{12}, C_{13}, and the optional C_{14}. C_{11} is further divided in C_{111}, and optional C_{112}, and C_{113}. Each different context means a different action taken for the real estate services.

Since ASTD is modular, we decided to use a combination of choice ASTD type and call ASTD type to simulate the tree presented in Figure 33. Figure 34 is the main ASTD CUD. CA represents the run where C_2 does not act, and CB, Figure 35, represent the run where the optional C_2 is executed.

The other ASTDs follow the same structure, they use a combination of call, choice, kleene closure, automata ASTD types. The complete ASTD model may be found in this technical report repository

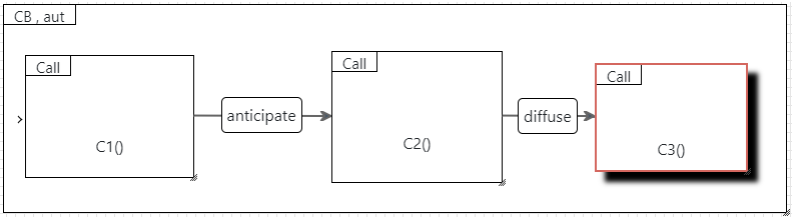

Figure 35. Case study FORM/BCS - CB

`https://depot.gril.usherbrooke.ca/diego/astd-tech-report-27` .

Although this model does not use time constraints, it was important to define cASTD (ASTD compiler) limitations. With the use of several call and choice ASTDs, the executable $c++$ file produced by cASTD reaches 155 MB. We intend to adopt measures to reduce the size of executable files, such as identify and remove non-reachable branches from the translation.

5.3 Automotive Adaptive Exterior Light and Speed Control System

The third case study is a adaptive exterior light and speed control system from an automotive.

In comparison with the event-B specification, the TASTD model is less abstract and more modular.

5.3.1 Modeling strategy

This section describes our modelling strategy, how the model is structured and provide insight how we approached the formalization of the requirements.

5.3.1.1 Model structure

We divide our model in the elements that can be manipulated by the user and environment, such as buttons, switches, and sensors, and the response on the actuators for the manipulation of those elements. We call the first group, physical structure, and the later group, actuators.

For the physical structure, each button, sensor and switch has an independent ASTD. They are key state, light rotary switch, pitman arm, hazard switch, reverse gear, battery, steering angle, cruise control lever, cruise control mode, brake pedal, gas pedal, traffic sign detection, doors, brightness, darkness mode switch, camera, and proximity sensor. All the physical structure is connected through an interleave ASTD type. Two, or more, ASTD inside an interleave type can act independently.

For the actuators, our specification has two modules, separated in speed actuators and light actuators. Further, we divide the light actuators in cornering lights, high beams, reverse light, brake light, low beams, and direction indication and hazard lights. Additionally, we also partitioned speed actuators in smaller modules, which are cruise controller, speed limit control, emergency break, and speed gauge. All the actuators are inside the same flow ASTD type. ASTDs inside an flow execute an event whenever possible.

Figure 36 shows the flow ASTD, Control, composed of physical structure and actuators. As one can observe, TASTD specification is modular, we can separate each part of the vehicle and specify its functions in each module. A modular structure allows us to define each group of requirements separately. As well, ASTD is a graphic notation, in which, one can visually identify the behavior of a module. ASTD also allows shared variables. To our specification this is an important aspect, since we use physical structure state to define the response in an actuator.

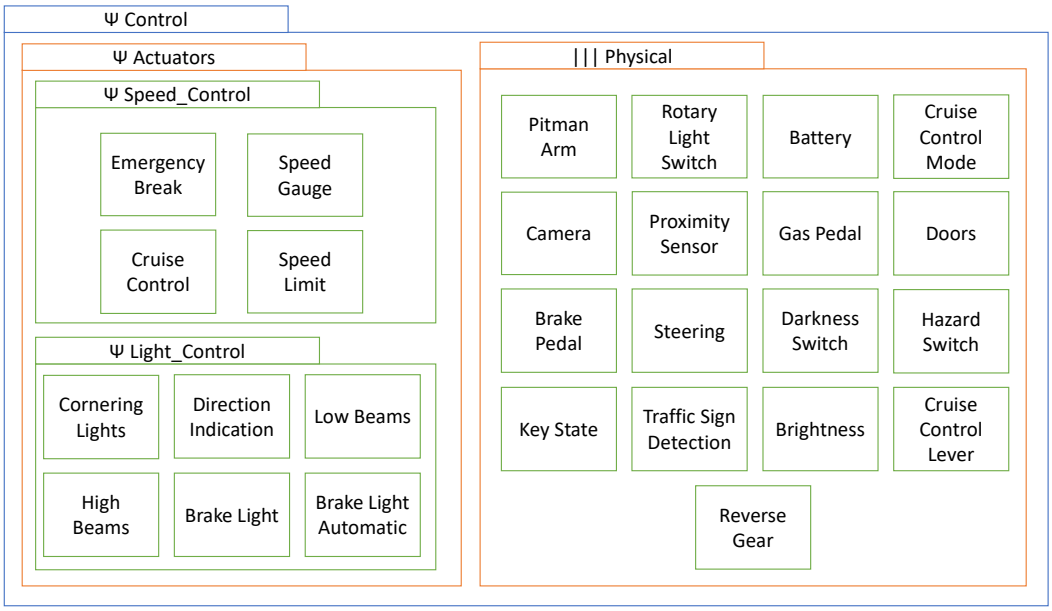

Figure 36. Case study - Automotive adaptive exterior light and speed control system - ASTD control flow

Figure 37. Case study - Automotive adaptive exterior light and speed control system - Main ASTD

ASTD Control is not the root. Although it may seems sufficient for complete modeling, to ensure requirements with ASTD Control means that we need to model each event at each state. Otherwise, an event present in both ASTD may be refused by ASTD Physical and accepted by ASTD Actuators. Whereas an event present in both ASTD has to always be accepted by Physical. So, we use a synchronization ASTD type with a synchronization over each event in ASTD Physical, what makes ASTD Control refuse a synchronized event when ASTD Physical cannot accept. Figure 37 shows the synchronization ASTD CAR, which is the root ASTD.

5.3.1.2 Formalization of the requirements

Tables 1 and 2 relates ASTDs and requirements listed in [11]. Some requirements are present in several ASTDs as they are related to different actuators. Time requirements, such as ELS-1 and SCS-8, are covered with the use of event *step*. SCS-30 is the only requirement not covered since it is related to interface appearance.

ASTD	Requirements
directionIndication	ELS-1, ELS-2, ELS-3, ELS4, ELS-5, ELS-6, ELS-7, ELS-8, ELS-9, ELS-10, ELS-11, ELS-12, ELS-13, ELS-23, ELS-29, ELS-47
lowBeams	ELS-14, ELS-15, ELS-16, ELS-17, ELS-18, ELS-19, ELS-21, ELS-22, ELS-28, ELS-29, ELS-47
corneringLights	ELS-24, ELS-25, ELS-26, ELS-27, ELS-29, ELS-45, ELS-47
highBeams	ELS-30, ELS-31, ELS-32, ELS-33, ELS-34, ELS-35, ELS-36, ELS-37, ELS-38, ELS-42, ELS-43, ELS-44, ELS-46, ELS-47, ELS-48, ELS-49
brakeLightAut	ELS-29, ELS-39, ELS-40, ELS-47
reverseLightAut	ELS-29, ELS-41, ELS-47

Table 1. Cross-reference between ASTDs and requirements for adaptive exterior light system of [11]

ASTD	Requirements
cruiseControl	SCS-1, SCS-2, SCS-3, SCS-4, SCS-5, SCS-6, SCS-7, SCS-8, SCS-9, SCS-10, SCS-11, SCS-12, SCS-13, SCS-14, SCS-15, SCS-16, SCS-17, SCS-18, SCS-19
automatedControlVehicleAhead	SCS-20, SCS-21, SCS-22, SCS-23, SCS-24, SCS-25, SCS-26
emergencyBreakSignals	SCS-27, SCS28
speedLimitControl	SCS-29, SCS-31, SCS-32, SCS-33, SCS-34, SCS-35
trafficSignDetection	SCS-36, SCS-37, SCS-38, SCS-39
cameraAndProximity	SCS-40, SCS-41
brakePedal	SCS-42
brakeLightAutomatic	SCS-43
Not implemented	SCS-30

Table 2. Cross-reference between ASTDs and requirements for speed control system of [11]

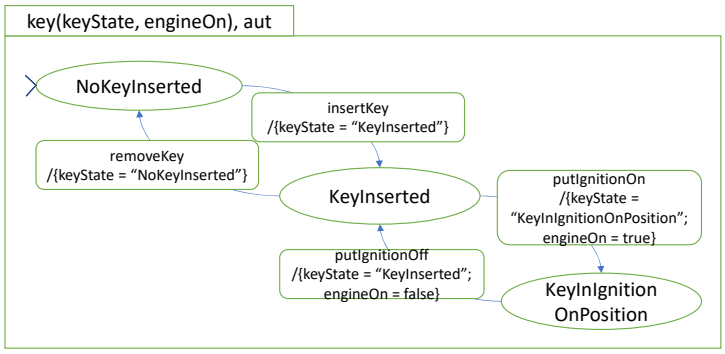

Figure 38. ASTD key

5.3.2 Model details

This section shortly describes the main modeling elements of our specification. The complete specification is found in

5.3.2.1 Physical structure ASTDs

Elements that can be manipulated by the user and environment are each an ASTD. They contain the valid states that the physical object can attain. For example ASTD key, that is shown in Figure 38. This ASTD is an automaton and its initial state is *NoKeyInserted*. The only valid states for ASTD key are *NoKeyInserted, KeyInserted, KeyInIgnitionOnPosition*, which transitions are valid movements for the key. With each transition, the shared variable *keyState* is updated. Event *putIgnitionOn* turns the engine on and shared variable *engineOn* becomes true. Event *putIgnitionOff* turns the shared variable *engineOn* false, as the engine turns off.

5.3.2.2 Actuators ASTDs

Actuators depends on the physical structure to act. Shared variables, that contain the state of the physical structure, affect how the actuators can be executed.

Let us take as example ASTD directionIndication, which is a flow between ASTD BlinkControl and ASTD BlinkBulb. For the sake of simplicity, lets extract the transitions between states *off* and *tip* from sub-ASTD BlinkControl, shown in Figure 39. State *off* indicates that blinking shall stop after completing the previous signal, whereas *tip* indicates that tip blinking shall be executed. Those two states have five transitions among then that depend on pitman arm, hazard switch, key state and time. The transition from *off* to *tip* through event *movePitmanArmUD*, is conditioned on the position which pitman arm is moving, and on the key state. The guard is to conform to requirements ELS-1, ELS-5 and the statement that direction blinking is only available when ignition is on. Executing the transition changes the value of variables *pitmanArmUDP*, *tip_timer*, and *NbrCycles*. Where *pitmanArmUDP*, stores the value of the last pitman arm position and later used to define which side shall blink. Variable *tip_timer* is used to acknowledge for how long the user holds pitman arm. *NbrCycles* is a counter to determine how many blinking are necessary. These variables are related to ELS-2, ELS-3, ELS-4, and ELS-7.

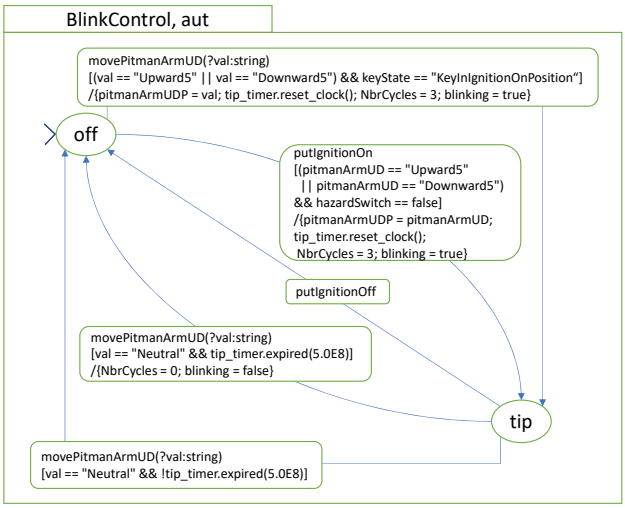

Figure 39. ASTD **BlinkControl, extract with states** *tip* **and** *off*

Similarly, Figure 40 is an extract from sub-ASTD BlinkBulb. State *off* indicates that the light is off, due to dark cycle or no blinking. State *on* means that the light is on. These two states alone have six transitions between them. Transition from *off* to *on* through event *movePitmanArmUD*, is intentionally related ELS-1, ELS-10, ELS-11, ELS-13. It has a guard on the position which pitman arm is moving, on the key state, on the hazard switch, and on cycle timer. Variable *cycle_timer* is used to acknowledge for how long the bulb is bright or dark, and *hazardSwitch* indicates if hazard switch is on. Executing this transition resets *cycle_timer* and perform function *blinkLightsOn*, that respecting other requirements turns on the blinking lights.

Moving pitman arm from *Neutral* to *Upward5*, at a state where user only turned the engine on, will execute transitions present in ASTD pitmanArm, ASTD BlinkBulb, and ASTD BlinkControl. What result in the activation of the right direction blinking light.

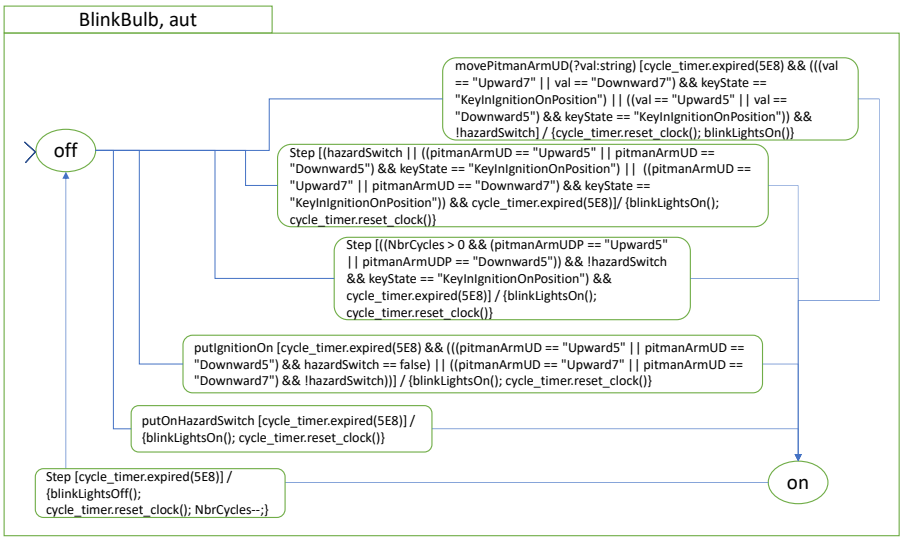

Figure 40. ASTD **BlinkBulb, extract with states** *on* **and** *off*

5.3.2.3 Modeling time requirements

Some requirements (e.g. ELS-1, and SCS-7) determine specific behavior for distinct components during specific time interval. In TASTD, with each event *Step* the system acknowledge the passage of time. So, choosing a *Step* value that successfully achieve all requirements is mandatory. For this case study, we chose the value of step as 0.05 seconds.

One may argue that with this step value, ELS-40 or ELS-8 may not be accomplished. ELS-40 asks a pulse ratio of 360 ± 60 flashes per minute, in other words 1/12 of second bright and 1/12 of second dark. We argue that for ELS-40, the chosen step value accomplish the requirement because we can have less 60 flashes per minute. At the worst case scenario, with a step of 1/20 second, there is 300 flashes per minute. ELS-8 demands a fixed pulse ratio of 1:2, what means 1/3 of second bright and 2/3 of second dark, without a safe range. With our chosen step we complete each cycle at 1.05 seconds, missing the requirement. Our solution is to reduce the value of a step, which shall meet the requirement.

At each *Step*, every transition labeled *Step* is executed, if its guard hold. In Figure 40 we have a transition *Step* from state *on* to state *off*. This transition is responsible to finish a bright cycle and turn off the direction blinking light. During execution, at each step value time units, a signal step is produced and then treated. In our simulations, at each 0.05 seconds the system would react to the passage of time. However, only after 0.5 seconds (or 10 steps) in state *on*, transition *Step* from state *on* to *off* is executed.

5.3.3 Validation and verification

This section we explain how we verify and valid our specification.

To validate our model we execute produced code for simulation and compare the results with the provided scenarios.

To start, more scenarios are necessary to produce a complete test. For instance, no provided scenario changes the value of the reverse light or both cornering lights. Additionally, some scenarios have misleading information or errors. For example, the first scenario of validation sequence speed, that illustrate requirements SCS-5 to SCS-9, changes the value of target speed. Based in other scenarios, target speed

refers to another vehicle speed detected by the proximity sensor, and in this scenario the proximity sensor is deactivated. To finish, no information in how to calculate speed is provided in the case study. We tried our best to calculate the car speed, but we were unable to reach the same value present in validation trace scenarios.

Compilation is automatic and no human modification is necessary after production. But the execution of compiled code is manual and verification needs a human to read the results, then compare with provided trace. Our model follows the provided scenarios in all instances, with minor differences in current speed due to no sufficient information in how to calculate it. To overcome this difference, we added to our model a function that changes current speed to a chosen value. This function is used when simulation arrive at row "target speed reached" for each scenario, mainly to continue the simulation with the same speed than provided the trace.

References

[1] L. N. Tidjon, M. Frappier, and A. Mammar, "Intrusion detection using astds," in *International Conference on Advanced Information Networking and Applications*. Springer, 2020, pp. 1397–1411.

[2] L. N. Tidjon, "Formal modeling of intrusion detection systems," Ph.D. dissertation, Institut Polytechnique de Paris; Université de Sherbrooke (Québec, Canada), 2020.

[3] MATLAB, "Stateflow," https://www.mathworks.com/products/stateflow.html, MATLAB Simulink, 2020.

[4] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[5] S. Schneider, *Concurrent and Real-time systems*. John Wiley and Sons, 2000.

[6] J. Sun, Y. Liu, J. S. Dong, and X. Zhang, "Verifying stateful timed csp using implicit clocks and zone abstraction," in *International Conference on Formal Engineering Methods*. Springer, 2009, pp. 581–600.

[7] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and É. André, "Modeling and verifying hierarchical real-time systems using stateful timed csp," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 1–29, 2013.

[8] L. Nganyewou Tidjon, M. Frappier, M. Leuschel, and 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.

[9] A. Cavalcanti, A. Sampaio, A. Miyazawa, P. Ribeiro, M. Conserva Filho, A. Didier, W. Li, and J. Timmis, "Verified simulation for robotics," *Science of Computer Programming*, vol. 174, pp. 1–37, 2019.

[10] M. Ngassam, "Conception et realisation d'un editeur des composants metiers caracteristiques de la methode form/bcs : Real," Master's thesis, Université de Douala, 2017.

[11] F. Houdek and A. Raschke, "Adaptive exterior light and speed control system," in *International Conference on Rigorous State-Based Methods*. Springer, 2020, pp. 281–301.

[12] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236.

[13] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal smc tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[14] J. Baxter, P. Ribeiro, and A. Cavalcanti, "Sound reasoning in tock-csp," *Acta Informatica*, 2021.

[15] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Advanced Course on Petri Nets*. Springer, 2003, pp. 87–124.

[16] P. Ribeiro, J. Baxter, and A. Cavalcanti, "Priorities in tock-csp," *arXiv preprint arXiv:1907.07974*, 2019.

[17] M. F. Ndjodo and A. Ngoumou, "The feature oriented reuse method with business component semantics." *Int. J. Comput. Sci. Appl.*, vol. 6, no. 4, pp. 63–83, 2009.