Technical Report 27 - Extending ASTD with real-time

version 2024-11-22

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 [16, 17]. 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 [8], automata, with timed automata [1], and process algebra, with Timed CSP [13].

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 [15]. 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 [17]. Process algebras support refinement and are also effective for model checking, but lack language features (e.g. shared variables) [14] 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 like 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 [10]. 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 operator .

To show the generality of our time extensions, we made sure that TASTD operators could specify models presented in RoboSim [5], timed automata [11], and an automotive adaptive exterior light and speed control system case study from ABZ-2020 [7].

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 [3, 6], MATLAB Stateflow [8], stateful timed CSP[14], tock-CSP [2], and timed CSP [13].

From timed automata [1, 4, 3], 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, which assign a time stamp from the system clock, and access its value. It simulates 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 [14], timed CSP [13], and *tock*-CSP [12, 2] 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 [14], [13], [8], and [12, 2].

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 . Symbols W, W' respectively denote the values of the parameters of ASTD a, which can be modified during 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 represents 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 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 initial 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 initialisation time is used only in ASTD types flow and synchronisation, because their sub-ASTD 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 initialized 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 current 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 for code generation. Timed-triggered transitions are labelled by the special event Step.

In UPPAAL, clocks increase synchronously at the same rate. The guard of a transition and state invariants restrict the behaviour of the automaton, and a transition can be fired when the clocks values satisfy that guard [4]. 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.

UPPAAL SMC [6], presents an approach to reason about 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.

Additionally, the discrete time semantics from timed automata and Stateful Timed CSP allow Zeno runs in their models. In TASTD, the semantics does not 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. New ASTD types are persistent guard and interrupt ASTDs, and delay, persistent delay, timeout, persistent timeout and timed interrupt 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

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

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 *Vars* 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 types described in [17] need to be adapted.

4.2.1 Automaton

A TASTD automaton adapts ASTD automaton with the addition of time stamps. 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; However, it is does not use invariants like timed automata do.

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 $n_{1\flat} \in \nu(n_1).S$.

In a transition of an automaton, event σ accepts the following: $\sigma(p_1, \ldots, p_n)$, where σ is an event label, each p_i is either a variable, a constant or the wildcard "_", which accepts any value.

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 describes 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 a timed automaton TASTD.

$$\begin{array}{rcl} \mathit{init}(a, \mathsf{ts}, G) & \triangleq & (\mathsf{aut}_\circ, a.n_0, a.E_{init}(\llbracket G \rrbracket), h_{init}, \mathit{init}(a.\nu(n_0), \mathsf{ts}, \\ & & G \lhd a.E_{init})) \\ h_{init} & \triangleq & \{n \mapsto \mathit{init}(a.\nu(n), \mathsf{cst}, a.E) \mid n \in a.S\} \\ \mathit{final}(a, (\mathsf{aut}_\circ, n, E, h, s)) & \triangleq & n \in a.SF \vee (n \in a.DF \wedge \mathit{final}(a.\nu(n), s)) \end{array}$$

Function *init* receives a timestamp ts and an environment G. The environment provides the values of variables declared in the enclosing ASTDs; these variables can be used to initialize the local variables of the automaton The timestamp ts is passed down to its sub-ASTD. The timestamp of a state denote when a state was reached; It is stored in the top level and in synchronisation and flow operators, because they have their own local clock, so to speak. The timestamp is used to determine the behaviour of TASTD types.

Symbol E_{init} denotes the initial values of attributes, as specified in their declaration. $a.E_{init}(\llbracket G \rrbracket)$ means the initial values of the variables of ASTD a using the variables declared in enclosing ASTDs present in environment G. 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 (*i.e.,* $\mathit{init}(\mathsf{elem}, \mathsf{cst}) = \mathsf{elem}$); composite automaton states are mapped to the initial state of their sub-ASTD, recursively. 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.

4.2.1.2 Semantics

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.

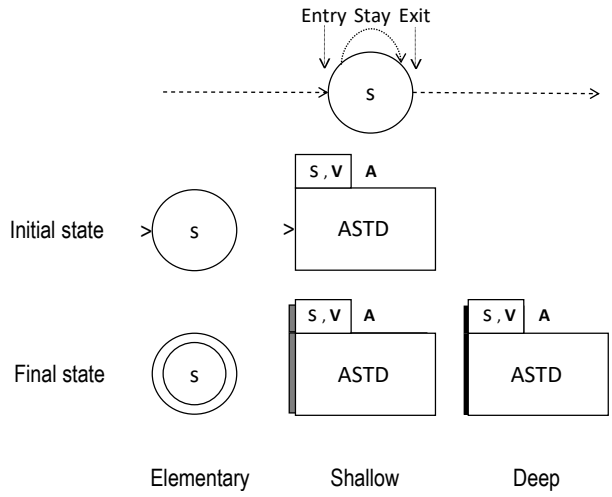

Figure 1. ASTD state

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

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 at cst(*i.e.,* $\mathit{init}(a.\nu(n_2), \mathsf{cst} \subset E'_g)$). The premiss provides that such a transition is possible if there is a matching transition, which is represented by $\delta((\mathsf{loc}, n_1, n_2), \sigma', g, A_{tr}, \mathit{final}?)$. 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.

$$\Psi \triangleq \ (\quad (\mathit{final}? \Rightarrow \mathit{final}(a.\nu(n_1), s)) \quad \wedge \quad g \quad \wedge \quad \sigma' = \sigma \quad)(\![E_g]\!)$$

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.

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

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 $E_e \lhd\!\!\lhd E \lhd\!\!\lhd E_c$, denotes the global list of variables that can be modified by the actions, where E_c are the environment of variable of capture type. Capture variables are placeholders for any possible variable, that variable is value is most likely to be provided by an external component. Their after values E'_g are used to set E' (the local attributes) using the restriction on the attributes (V) and capture variables (C) 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.

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

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

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

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.

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

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

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 $\mathsf{H*}$); 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.

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

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

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.

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

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

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

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.

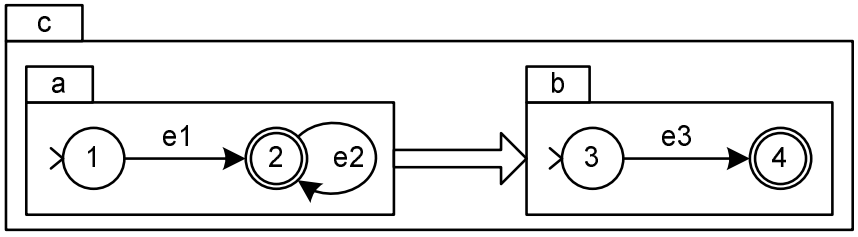

Figure 2. Example - ASTD sequence

4.2.2.1 Syntax

The sequence ASTD subtype has the following structure:

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

where fst, snd are ASTDs denoting respectively the first and second sub-ASTDs of the sequence. A sequence state is of type $\langle \mathrel{⇨}_\circ, E, [\mathsf{fst} \mid \mathsf{snd}], s \rangle$, where $\mathrel{⇨}_\circ$ 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.

$$\begin{aligned}
init(a, \mathsf{ts}, G) &\triangleq (\mathrel{⇨}_\circ, a.E_{init}(\llbracket G \rrbracket), \mathsf{fst}, init(a.fst, \mathsf{ts}, G \lessdot a.E_{init})) \\
final(a, (\mathrel{⇨}_\circ, E, \mathsf{fst}, s)) &\triangleq final(a.fst, s) \wedge final(a.snd, init(a.\mathsf{snd}, \mathsf{cst}, E)) \\
final(a, (\mathrel{⇨}_\circ, E, \mathsf{snd}, s)) &\triangleq final(a.snd, s)
\end{aligned}$$

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 influence evaluation of *final*, so it is filled with cst (any other value could be chosen).

4.2.2.2 Semantics

Three rules are necessary to define the execution of the sequence. Rule $⇨_1$ deals with transitions on the sub-ASTD *fst* only. Rule $⇨_2$ deals with transitions from *fst* to *snd*, when *fst* is in a final state. Rule $⇨_3$ deals with transitions on the sub-ASTD *snd*.

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

$$⇨_2 \frac{final(a.fst, s) \qquad init(a.snd, \mathsf{t}, E_e) \xrightarrow{\sigma,\mathsf{t},E_g,E_g''}_{a.snd} s' \qquad \Theta}{(⇨_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma,\mathsf{t},E_e,E_e'}_{a} (⇨_\circ, E', \mathsf{snd}, s')}$$

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

The initial value of variables of *snd* are determined using the current values of the attributes. The timestamp of the last executed event is the one of the current state.

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. Figure 3 provides an example of a choice ASTD, which includes two automata 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.

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 initial state of its 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.

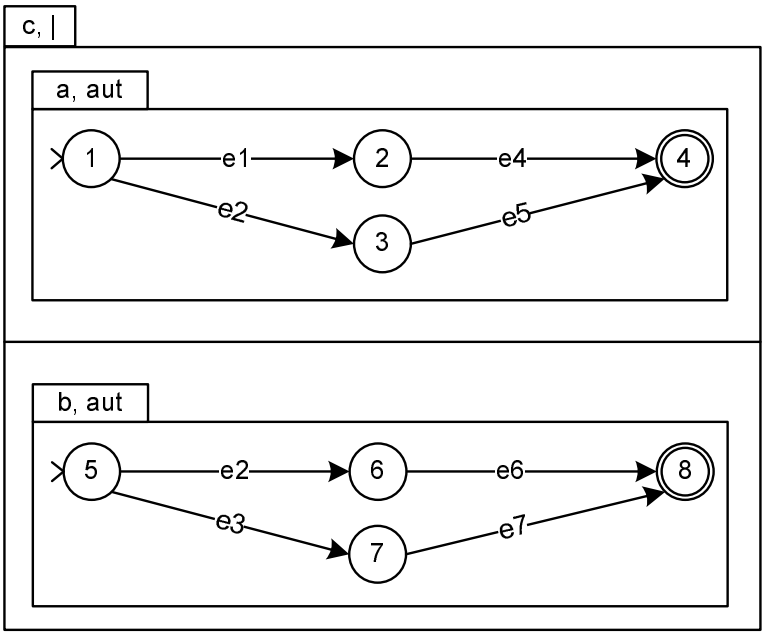

Figure 3. Example - ASTD choice

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

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.

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

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

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

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

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 compound ASTDs. Each contains respectively two ASTD automata a and b. The LHS closure ASTD c must be reach its final state before the next one (d) starts its execution. ASTD f can start its execution with *e1* or *e3*, since c is a closure, its initial state is final, and thus the second element of the sequence can execute *e3* from its initial state. After *e1*, it can execute *e1*, *e2* or *e3*. It can execute *e1* again since d is a closure and its initial state is thus final.

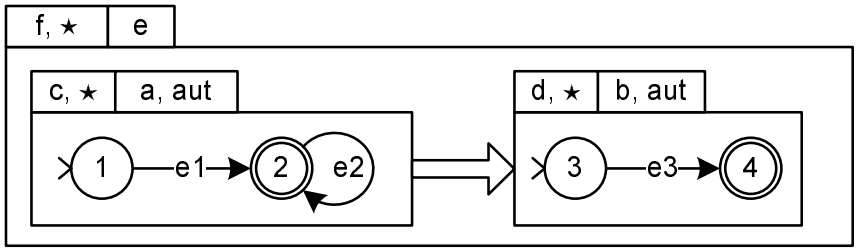

Figure 4. Example - closure ASTD

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 $\langle \star_\circ, E, \mathit{started}?, s \rangle$ where $s \in \mathsf{State}$, E the attribute values of the closure ASTD, $\mathit{started}? \in \mathsf{Boolean}$ indicates whether the first iteration has been started. It is essentially used to determine if the closure can immediately exit (*i.e.,* it is in a final state) without any iteration. Initial and final states are defined as follows. Let a be a closure ASTD.

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

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.

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

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

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.

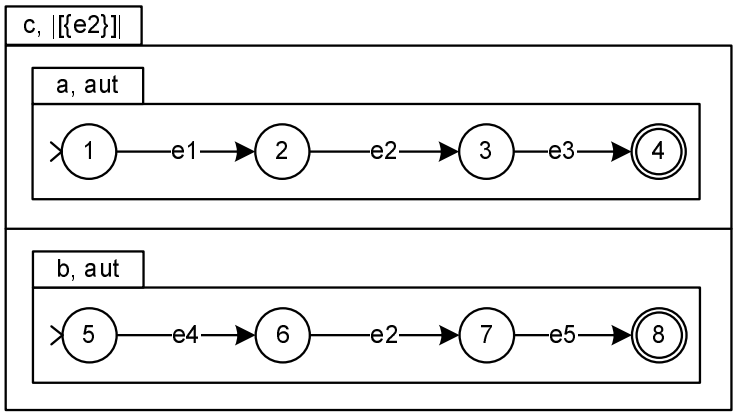

Figure 5. Example - synchronization ASTD

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.

$$\begin{aligned} \mathit{init}(a, \mathsf{ts}, G) &\triangleq (|[]|_\circ, a.E_{init}(\llbracket G \rrbracket), \mathit{init}(a.l, \mathsf{ts}, G \lhd a.E_{init}), \mathsf{ts}, \\ &\qquad \mathit{init}(a.r, \mathsf{ts}, G \lhd a.E_{init}), \mathsf{ts}) \\ \mathit{final}(a, (|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r)) &\triangleq \mathit{final}(a.l, s_l) \wedge \mathit{final}(a.r, s_r) \end{aligned}$$

4.2.5.2 Semantics

There are two inference rules. Rule $|[]|_1$ describe execution of events, with no synchronization required, either on the left or the right sub-ASTDs. Rule $|[]|_1$ below caters for execution of one of the sub-ASTDs.

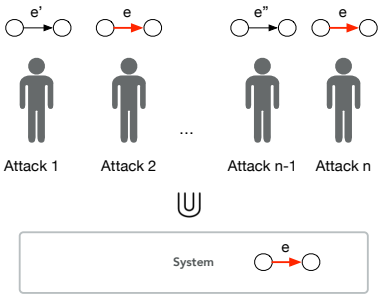

Figure 6. Using the Flow operator to synchronize multiple attack models

The function $\alpha(e)$ returns the label of event e.

$$|[]|_1 \frac{\begin{array}{ll}(1) & = \{(1, s_1, t_1, s'_l, t'_l), (r, s_r, t_r, s'_r, t'_r)\} \\ & \{(ok, s_{ok}, t_{ok}, s'_{ok}, \mathsf{cst}), (ko, s_{ko}, t_{ko}, s_{ko}, t_{ko})\} \\ (2) & \alpha(\sigma) \notin \Delta \quad \wedge \quad s_{ok} \xrightarrow{\sigma, t_{ok}, E_g, E''_g}_{a.A_{ok}} s'_{ok} \\ (3) & \Theta \end{array}}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma, \_, E_e, E'_e}_a (|[]|_\circ, E', s'_l, \mathsf{t}'_l, s'_r, \mathsf{t}'_r)}$$

Where (1) matches sub-ASTDs $\{l, r\}$ with $\{ko, ko\}$, allowing to write a single rule for either l or r executing the event. (2) ensures that only one of sub-ASTDs can be executed.

Rule $|[]|_2$ indicates behaviour when the two sub-ASTDs synchronises in the event σ.

$$|[]|_2 \frac{\begin{array}{ll}(1) & = \{(1, s_l, t_l, s'_l), (r, s_r, t_r, s'_r)\} \\ & \{i, s_i, t_i, s'_i), (j, s_j, t_j, s'_j)\} \\ (2) & s_i \xrightarrow{\sigma, t_i, E_g, \_}_{a.i} \_ \quad \wedge \quad s_j \xrightarrow{\sigma, t_j, E_g, \_}_{a.j} \_ \\ (3) & \alpha(\sigma) \in \Delta \quad \wedge \quad s_i \xrightarrow{\sigma, t_i, E_g, E'''_g}_{a.i} s'_i \quad \wedge \quad s_j \xrightarrow{\sigma, t_j, E'''_g, E''_g}_{a.j} s'_j \\ (4) & \Theta \end{array}}{(|[]|_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma, \_, E_e, E'_e}_a (|[]|_\circ, E', s'_l, \mathsf{cst}, s'_r, \mathsf{cst})}$$

Where (1) matches sub-ASTDs $\{l, r\}$ with $\{i, j\}$, allowing to write a single rule for either l or r executing the event. (2) ensures that both sub-ASTDs can execute the event from the current state. (3) ensures that the two sub-ASTDs are executed. Additionally, (1) with (3) ensures that the first to execute σ is chosen non-deterministically.

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. 6. 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 Ψ, 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:

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

A flow state is of type $\langle \Psi_\circ, E, s_l, t_l, s_r, t_r \rangle$, 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.

$$\begin{array}{rcl} \mathit{init}(a, \mathsf{ts}, G) & \triangleq & (\Psi_\circ, a.E_{init}(\llbracket G \rrbracket), \mathit{init}(a.l, \mathsf{ts}, G \lessdot a.E_{init}), \mathsf{ts}, \\ & & \mathit{init}(a.r, \mathsf{ts}, G \lessdot a.E_{init}), \mathsf{ts}) \\ \mathit{final}(a, (\Psi_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r)) & \triangleq & \mathit{final}(a.l, s_l) \wedge \mathit{final}(a.r, s_r) \end{array}$$

4.2.6.2 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 rule when only one of the two sub-ASTDs can execute the event.

$$\Psi_1 \; \frac{\begin{array}{ll} (1) & = \{(l, s_l, t_l, s'_l, t'_l), (r, s_r, t_r, s'_r, t'_r)\} \\ & \quad \{(ok, s_{ok}, t_{ok}, s'_{ok}, \mathsf{cst}), (ko, s_{ko}, t_{ko}, s_{ko}, t_{ko})\} \\ (2) & s_{ok} \xrightarrow{\sigma, t_{ok}, E_g, E''_g}_{a.A_{ok}} s'_{ok} \quad \wedge \quad s_{ko} \xrightarrow{\sigma, t_{ko}, E_g, -}\!\!\!\!\!\!/\;\; _{a.A_{ko}} \\ (3) & \Theta \end{array}}{\langle \Psi_\circ, v, s_1, t_1, s_2, t_2 \rangle \xrightarrow{\sigma, -, E_e, E'_e}_a \langle \Psi_\circ, v', s'_1, t'_1, s'_2, t'_2 \rangle}$$

Where (1) matches sub-ASTDs $\{l, r\}$ with $\{ko, ko\}$, allowing to write a single rule for either l or r executing the event. (2) ensures that only one of sub-ASTDs can be executed.

The second rule describes the case where both sub-ASTDs can execute the event; it is almost the same as $|[]|_2$, as it requires commutativity.

$$\Psi_2 \; \frac{\begin{array}{ll} (1) & = \{(l, s_l, t_l, s'_l), (r, s_r, t_r, s'_r)\} \\ & \quad \{(i, s_i, t_i, s'_i), (j, s_j, t_j, s'_j)\} \\ (2) & s_i \xrightarrow{\sigma, t_i, E_g, -}_{a.i} - \quad \wedge \quad s_j \xrightarrow{\sigma, t_j, E_g, -}_{a.j} - \\ (3) & s_i \xrightarrow{\sigma, t_i, E_g, E'''_g}_{a.i} s'_i \quad \wedge \quad s_j \xrightarrow{\sigma, t_j, E'''_g, E''_g}_{a.j} s'_j \\ (4) & \Theta \end{array}}{(\Psi_\circ, E, s_l, \mathsf{t}_l, s_r, \mathsf{t}_r) \xrightarrow{\sigma, -, E_e, E'_e}_a (\Psi_\circ, E', s_l, \mathsf{cst}, s'_r, \mathsf{cst})}$$

Where (1) matches sub-ASTDs $\{l, r\}$ with $\{i, j\}$, allowing to write a single rule for either l or r executing the event. (2) ensures that both sub-ASTDs can execute the event from the current state. (3) ensures that the two sub-ASTDs are executed. Additionally, (1) with (3) ensures that the first to execute σ is chosen non-deterministically.

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 7, 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.

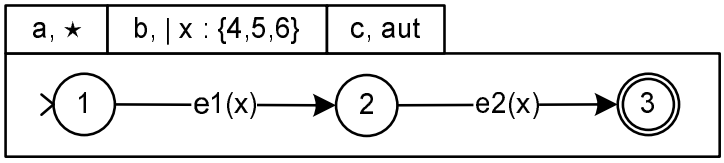

Figure 7. Example - ASTD quantified choice

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.

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

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.

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

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

4.2.8 Quantified Synchronization

The quantified synchronization allows for the modelling 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 behaviour 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 8 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.

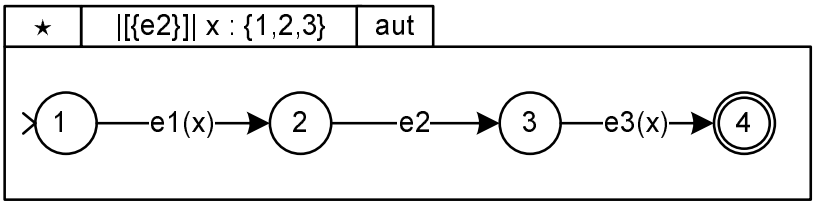

Figure 8. Example - quantified synchronization

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 $\langle |[]|:_\circ, E, t, f, u \rangle$ 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, t is the last event execution time which is stored in the state of the quantified synchronization to initialize the last event execution time of each instance of b, $u \in T \to R_{0+}$ 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.

$$\begin{array}{rcl} \mathit{init}(a, \mathsf{ts}, G) & \triangleq & (|[]|:_\circ, a.E_{\mathit{init}}([\![G]\!]), \mathsf{ts}, \varnothing, \varnothing\}) \\ \mathit{final}(a, (|[]|:_\circ, E, \mathsf{t}, f, u)) & \triangleq & \forall\, d : T \cdot \mathit{final}(a.b, f'(d)) \\ & & \text{where} \\ f' & = & f \cup (T - \mathit{dom}(f)) \times (\mathit{init}(a.b,\ t,\ E \lhd\!\!\!\!- a.E_{\mathit{init}})) \end{array}$$

4.2.8.2 Semantics

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

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

where u' is:

$$u' = u \cup (T - \mathsf{dom}(u)) \times t$$

Rule $|[]|:_2$ describe execution of an event with synchronization. All elements of T must execute σ, in any order.

$$|[]|:_2 \frac{\begin{array}{ll}(1) & \alpha(\sigma) \in \Delta \quad \wedge \quad k = |Tp| \quad \wedge \quad p \in \pi(Tp) \\ (2) & \forall\, d \in Tp \cdot f'(d) \xrightarrow{\sigma, u'(d), E_g\{x \mapsto d\}, \_}_{a.b\ \_} \\ (3) & g \in 0..k \nrightarrow \mathsf{Env} \quad \wedge \quad g(0) = E_g \quad \wedge \quad g(k) = E''_g \\ (4) & \forall\, i \in 1..k \cdot f'(p(i)) \xrightarrow{\sigma, u(p(i)), g(i-1) \lhd\!\!+ \{x \mapsto p(i)\}, g(i)}_{a.b}\ f''(p(i)) \\ (5) & \Theta \end{array}}{(|[]|:_\circ, E, t, f, u) \xrightarrow{\sigma, \_, E_e, E'_e}_a (|[]|:_\circ, E', t, f'', Tp \times \mathsf{cst})}$$

where in equation (1), it is asserted that the order is non-deterministic due to the random selection of permutation p from Tp, representing the sequence in which the instances of b are executed. (2) guarantees that each instance can execute the event. (3) keeps track of the intermediate values of the global environment during the execution of the k instances, which g(0) denotes the initial global environment and g(k) their final values. (4) resolves the state of the instances after their execution.

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.

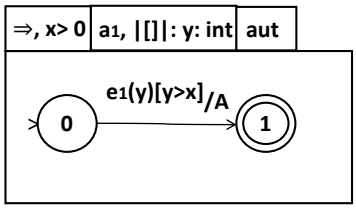

Figure 9. Example - guard

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 $\langle \Rightarrow_\circ, E, \mathit{started}?, s \rangle$ where $\mathit{started}?$ 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.

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

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.

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

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

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 10 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.

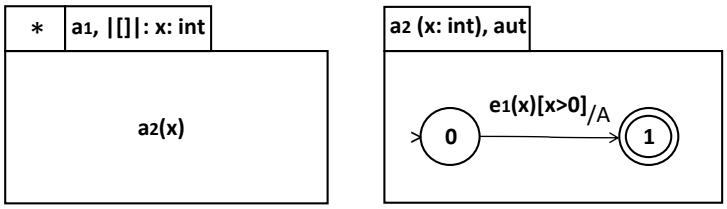

Figure 10. Example - call

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.

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

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 $E_{cal} = \{P \mapsto \vec{c}\}$.

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

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

4.3 New ASTD Types

This section presents two new ASTD types, persistent guard and interrupt. Additionally we also define types delay, timeout, persistent timeout, timed interrupt which are specific behavior of the composition of different ASTDs. 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.

All new ASTD types and 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.

4.3.1.1 Syntax

The persistent guard ASTD 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.

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

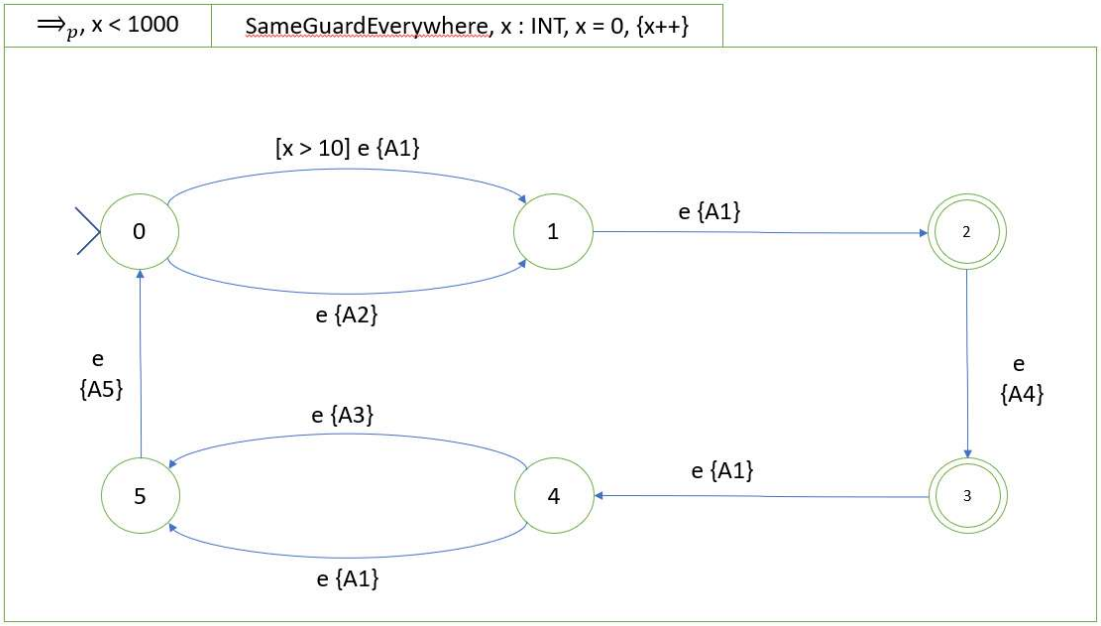

Figure 11. Example - persistent guard

The initial and final states of a persistent guard ASTD are straightforward. For the initial state, it initializes 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: $\Rightarrow_{p_1}$ execute any transition from the ASTD that always shall hold on g.

$$\Rightarrow_{p_1} \frac{g([\![E_g]\!]) \qquad s \xrightarrow{\sigma,\mathrm{t},E_g,E_g''}_{a.b} s' \qquad \Theta}{(\Rightarrow_p, E, s) \xrightarrow{\sigma,\mathrm{t},E_e,E_e'} (\Rightarrow_p, E', s')}$$

4.3.2 Interruption

The interruption ASTD 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. When both first and the initial state of second sub-ASTDs can execute event e, the choice between the two options is non deterministic. One can remove this non-determinism by adding guards around fst and snd to determine which one should be executed.

4.3.2.1 Syntax

The interruption ASTD has the following structure:

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

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

An interruption state is of type $\langle \mathsf{Intpt}_\circ, E, [\mathsf{fst}|\mathsf{snd}], s \rangle$, where $\mathsf{Intpt}_\circ$ 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 $s \in State$.

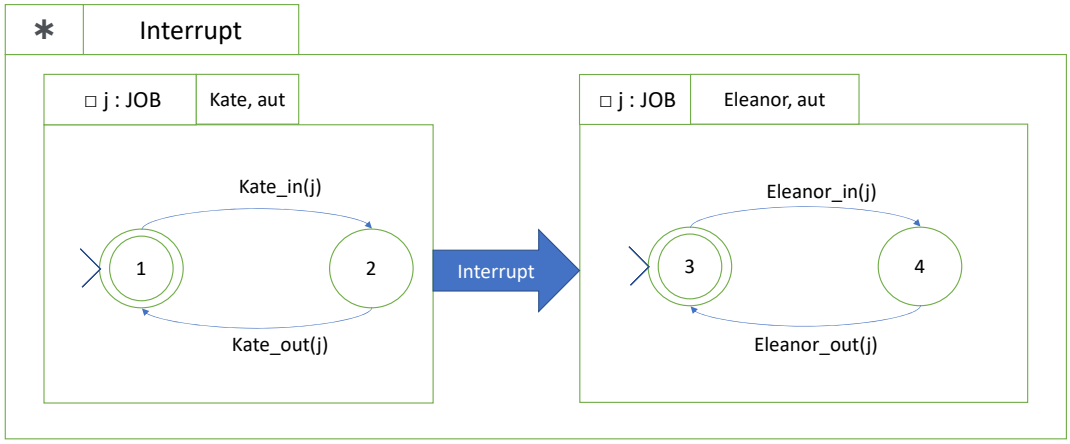

Figure 12. Example - interruption

Let a be an interruption TASTD.

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

The initial state of an interruption initializes 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.

4.3.2.2 Semantics

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

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

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

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

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

4.3.2.3 Example

The copier from 12 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.

4.3.3 Delay

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

4.3.3.1 Syntax

Delay TASTD has the following structure:

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

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 $\langle \mathsf{Delay}_\circ, E, started?, s \rangle$ where $s \in State$, E the attribute values of the delay TASTD, and $started?$ is a boolean which indicates if the first event occurred. Initial and final states are defined as follows. Let a be a TASTD.

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

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

4.3.3.2 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.

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

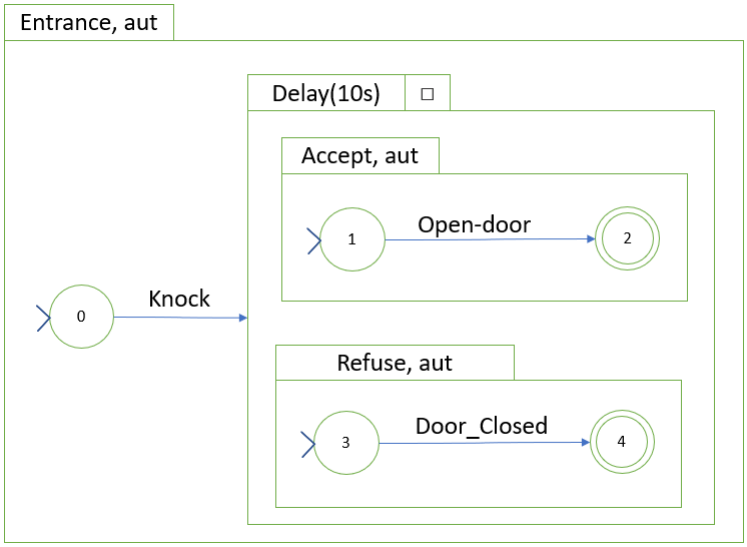

Figure 13. Example - Open door

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

4.3.3.3 Example

Figure 13 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*.

4.3.3.4 Defining Delay using a Guard

A delay TASTD can be equivalently represented by a Guard ASTD over the variables cst and t. The equivalence is shown in Figure 14.

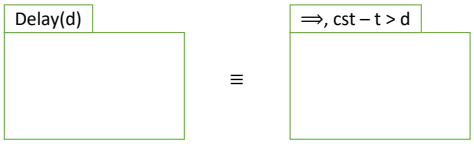

Figure 14. Equivalence between delay TASTD and guard ASTD

4.3.4 Persistent Delay

A persistent delay TASTD allows for idling for at least d time units before each event.

4.3.4.1 Syntax

Persistent delay TASTD has the following structure:

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

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 $\langle \mathsf{Delay}_p, E, s \rangle$ where $s \in State$, E the attribute values of the delay TASTD. Initial and final states are defined as follows. Let a be a TASTD.

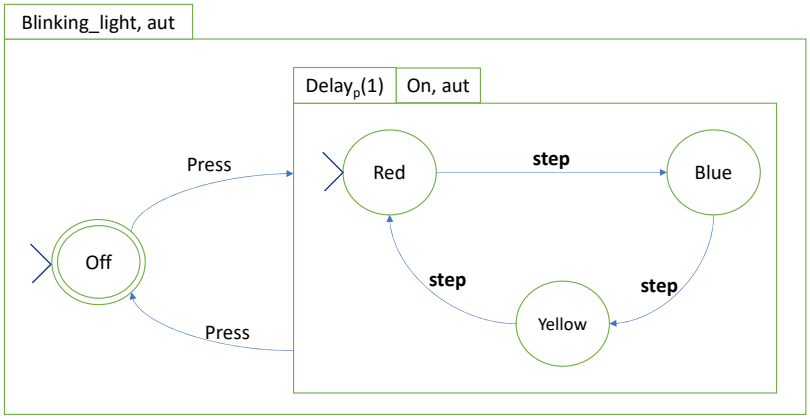

Figure 15. Example - Blinking lights

$$\begin{aligned} init(a, \mathsf{ts}, G) &\triangleq (\mathsf{Delay}_p, a.E_{init}([\![G]\!]), init(a.b, \mathsf{ts}, G \lhd a.E_{init})) \\ final(a, (\mathsf{Delay}_p, E, s)) &\triangleq final(a.b, s) \end{aligned}$$

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

4.3.4.2 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.

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

4.3.4.3 Example

A LED lamp, shown in Figure 15, 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 is no press event, 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.

4.3.4.4 Defining Persistent Delay using a Persistent Guard

Similarly to the equivalence between Delay TASTD and Guard ASTD, a Persistent Delay can be equivalently represented by a Persistent Guard. The equivalence is shown in Figure 16.

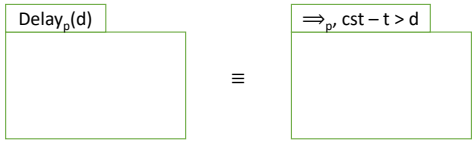

Figure 16. Equivalence between persistent delay TASTD and persistent guard ASTD

4.3.5 Timeout

A timeout TASTD allows for verifying if a TASTD receives an event within a time period of d units, 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.

4.3.5.1 Syntax

A timeout TASTD has the following structure:

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

where $fst, snd \in$ 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, [fst|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.

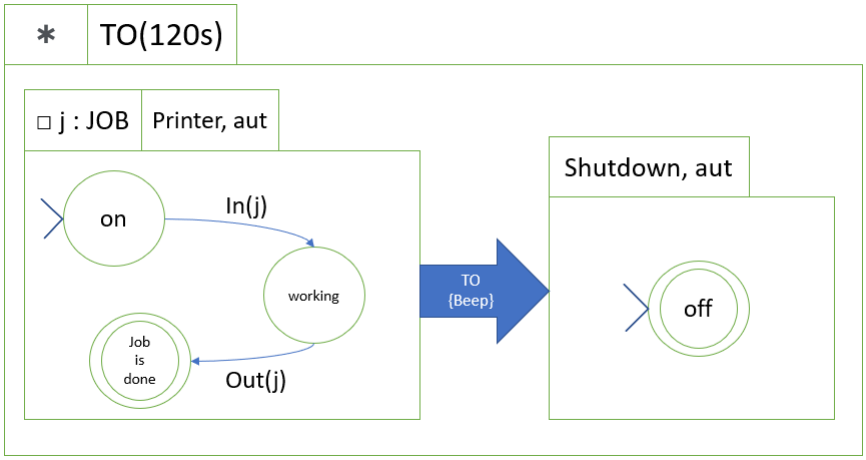

Figure 17. Example - Timeout

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

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

The initial state of a timeout initializes 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.

4.3.5.2 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.

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

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

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

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

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

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

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

4.3.5.3 Example

The printer example, shown in Figure 17, 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: $(\mathsf{aut}_\circ, \text{“job is done”}, E, h, \text{“job is done”})$
- Quantified Choice state: $(|:_\circ, j, E, \text{“job is done”})$
- Timeout state: $(\mathsf{Timeout}_\circ, j, E, \text{“job is done”})$
- 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 ($\star_1$). $init(KC, c) = (\star_\circ, E, \mathsf{true}, \text{“job is done”})$

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

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

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

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 ($\mathsf{Timeout}_3$).

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

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 not active state where the Timed automaton is not reinitialized by the quantified choice, which shall remain in it initial state. It can be executed again by starting a new iteration of the closure.

4.3.5.4 Defining Timeout using a combination of other ASTDs

A timeout TASTD, as shown in Figure 18, can be equivalently represented by a combination of other ASTDs, as shown in Figure 19.

The two ASTDs are equivalent since the ASTD in Figure 19 can simulate every rule of the timeout TASTDof Figure 18.

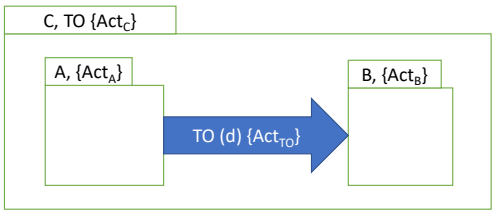

Figure 18. Timeout TASTD

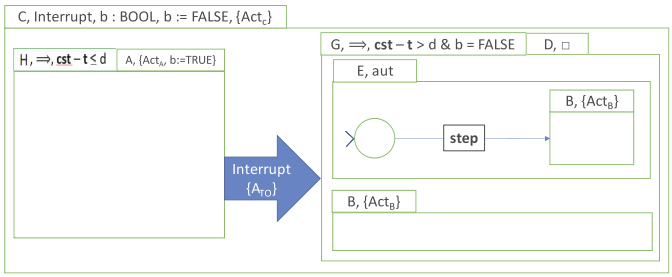

Figure 19. Equivalent representation of the timeout ASTD**of 24**

- Rules $\mathsf{Timeout}_1$ and $\mathsf{Timeout}_2$ is simulated by executing ASTD H and A, which is equivalent to executing ASTD A of Figure 18 ASTD. ASTD H guarantees that $\mathsf{cst} - \mathsf{t} \leq d$.
  - Rule $\mathsf{Timeout}_1$ allows for the execution of the first event of ASTD H and A, which will trigger the ASTD A action that assigns b to true. ASTD H guarantees that $\mathsf{cst} - \mathsf{t} \leq d$.
  - 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 and the timeout action, while $\mathsf{Timeout}_4$ also execute ASTD B. This behaviour is captured by the interruption C, the guard G, and the choice D.
  - Rule $\mathsf{Timeout}_3$ is represented 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 in the first execution of ASTD B the guard from ASTD G has to be true.
- Rule $\mathsf{Timeout}_5$ allows for the execution of ASTD B.

4.3.6 Persistent Timeout

A persistent timeout TASTD allows for verifying if the active state of a TASTD receives an event within a period of time, if not, then another TASTD has the right to execute. In the persistent timeout TASTD, the active state of the first TASTD must execute the next event within d time units.

4.3.6.1 Syntax

A persistent timeout TASTD has the following structure:

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

where $fst, snd \in \mathsf{TASTD}$ 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, $s \in State$.

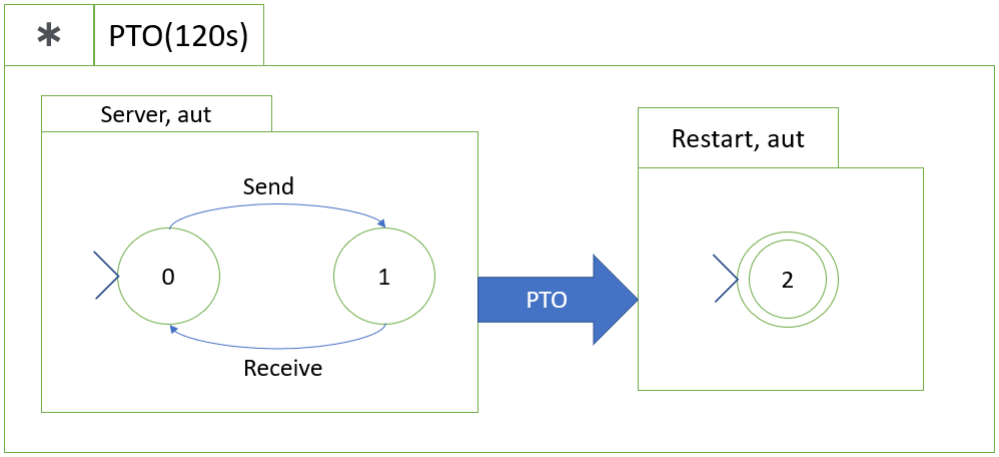

Figure 20. Example - Persistent timeout

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

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

The initial state of a persistent timeout initializes 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.

4.3.6.2 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.

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

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

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

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

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

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

4.3.6.3 Example

The jammed server example, shown in 20, 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.

4.3.6.4 Defining Persistent Timeout using combination of other ASTDs

The Persistent Timeout TASTD shown in Figure 21, is equivalent to the ASTD, shown in Figure 22.

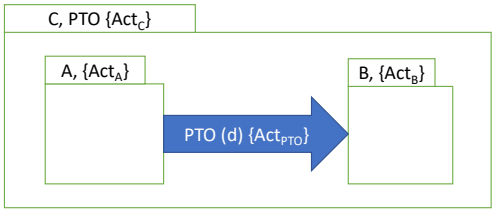

Figure 21. Persistent timeout TASTD

The two ASTDs are equivalent since the ASTD of Figure 22 can simulate every rule of the Persistent Timeout TASTD of Figure 21.

- Rule PTimeout_1 is simulated by ASTD H and A, which is equivalent to executing ASTD A of the ASTD in Figure 21. ASTD H guarantees that $\mathsf{cst} - \mathsf{t} \leq d$.
- Rules PTimeout_2 and PTimeout_3 are going to execute the timeout, and the persistent timeout action. This behaviour is captured by the interruption C, the guard G and the choice D, ASTDs.
  - Rule PTimeout_2 is simulated 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.

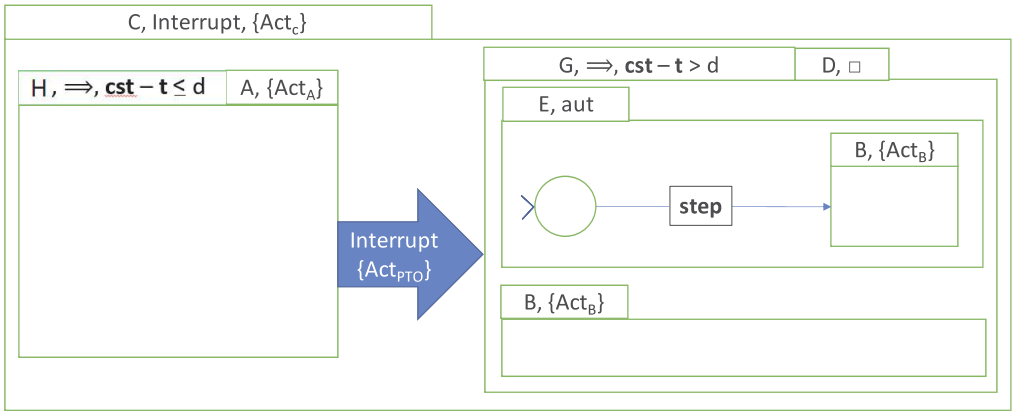

Figure 22. Equivalent representation of the persistent timeout TASTD of Figure 21

  - Rule $\mathsf{PTimeout}_3$ is simulated by the choice of ASTD B. It is possible due the guard, so the first execution of ASTD B has to satisfy the guard from ASTD G.

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

4.3.7 Timed Interrupt

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

4.3.7.1 Syntax

The timed interrupt TASTD subtype has the following structure:

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

where $fst, snd \in \mathsf{TASTD}$ 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.

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

The initial state of a timed interrupt initializes 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 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.

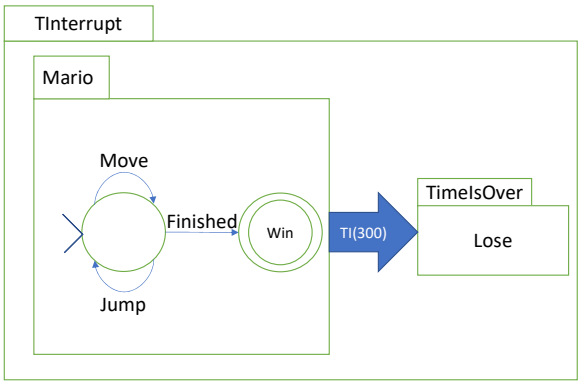

Figure 23. Example - timed interrupt

4.3.7.2 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.

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

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

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

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

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

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

4.3.7.3 Example

Figure 23 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.

4.3.7.4 Defining Timed Interrupt using combination of other ASTDs

The timed interrupt TASTD, shown in Figure 24, is equivalent to the ASTD, shown in Figure 25.

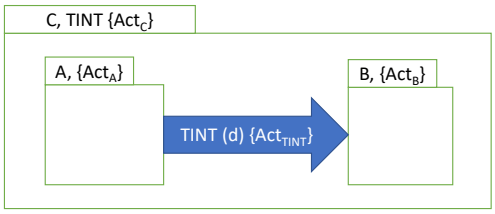

Figure 24. Timed Interrupt TASTD

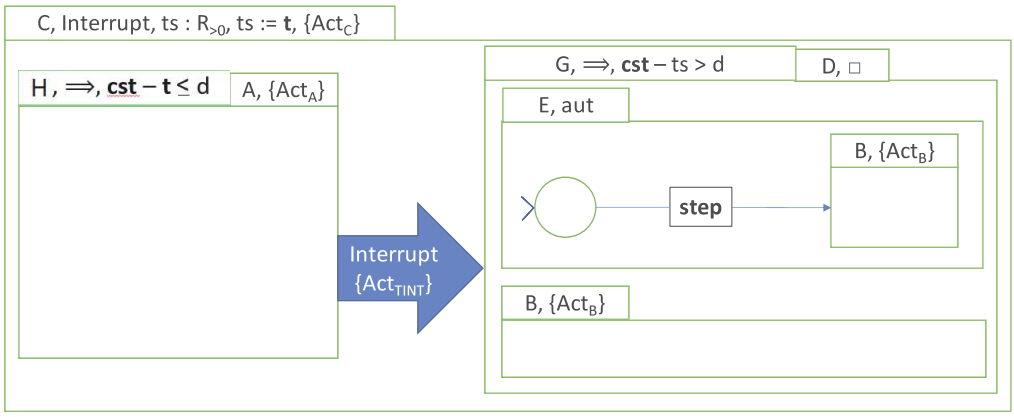

Figure 25. Equivalent representation of the timed interrupt TASTD **of Figure 24**

The two ASTDs are equivalent since the ASTD of Figure 25 can simulate every rule of the Timed Interrupt TASTD24.

- Rule $\mathsf{TInterrupt}_1$ is simulated by ASTD H and 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 C, the guard G and the choice D ASTDs
  - Rule $\mathsf{TInterrupt}_2$ is simulated 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 simulated 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.

4.4 Some observations on TASTD behavior

4.4.1 Interruption and an automaton with history state.

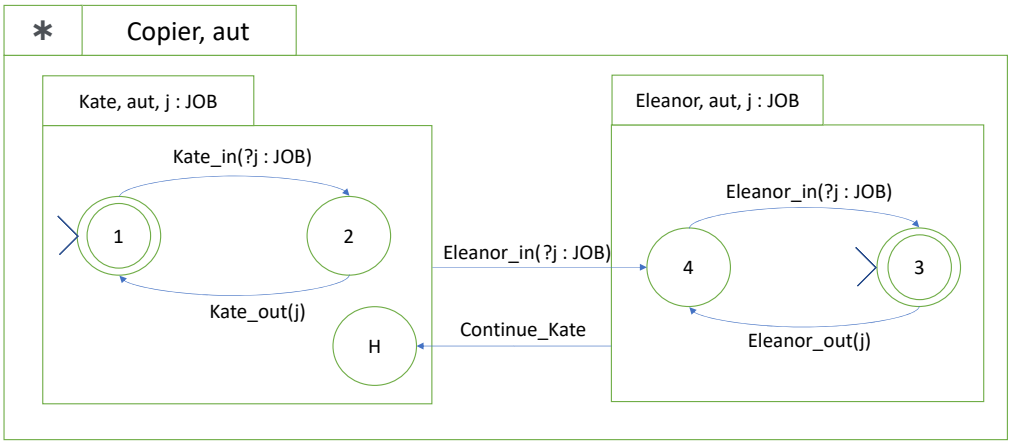

Figure 26. Example - Copier Timed Automata

An interruption ASTD does not return to the interrupted ASTD by any means other than reinitializing the interruption ASTD. In contrast, an automaton with history state, shown in Figure 26, is capable of returning to ASTD Kate. However, such transitions are only allowed in an automaton ASTD, thus this behavior cannot be generalized to arbitrary ASTDs.

4.4.2 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 executing the first sub-ASTD does not mean that the second sub-ASTD will also execute. That way, timing operators that refer to the time of execution of the last event (t) refer to the last event executed within this sub-ASTD; it is not influenced by the execution of event in the other synchronised ASTD. An example of that case is shown in Figure 27.
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. 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 event for the first sub-ASTD shall not assign cstto 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 occurs 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 the last the action of the transition.

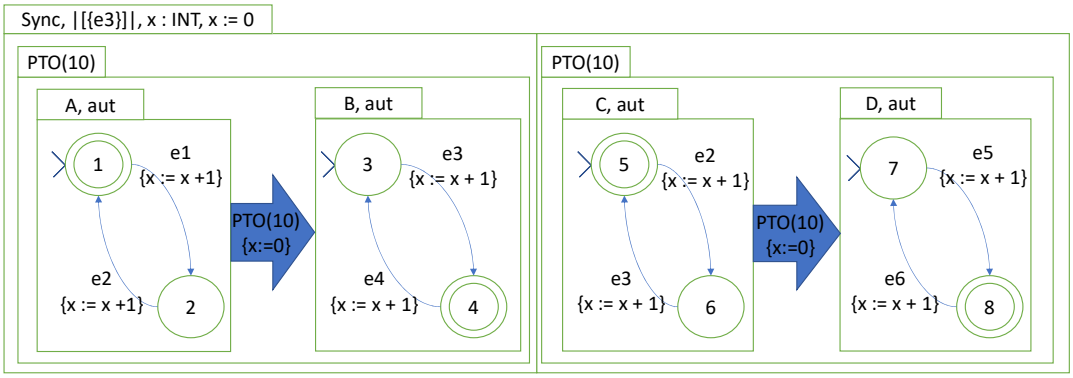

Figure 27. Example - Timeout synchronization

Both rules are available with the following execution:

1. With the initialization of the TASTD on Figure 27, 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 5 more seconds, two rules are available. If Step occurs, then 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.

4.4.3 Commutativity

In [17] we are introduced to formal definitions for flow and synchronization ASTDs and the commutativity in their actions. However, the definition of commutativity could not be checked by the ASTD compiler (cASTD). Commutativity has been replaced by a non deterministic choice between the two possible execution order for the two sub-ASTDs in a flow and synchronization. It is important to mention that in the newer versions of the ASTD editor (eASTD) it is possible to determine the order of execution.

4.4.4 Handling Step events

In the earlier discussions of Step introduction, different ways of handling Step events were discussed. However, the final consensus is that Step has to be treated as any other event, without distinction. If one wants to synchronization in Step, then one shall use an synchronization ASTD with Step. If one wants to have a flow in Step, the one shall use a flow ASTD.

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 [5]. Second case study is a production line [11].

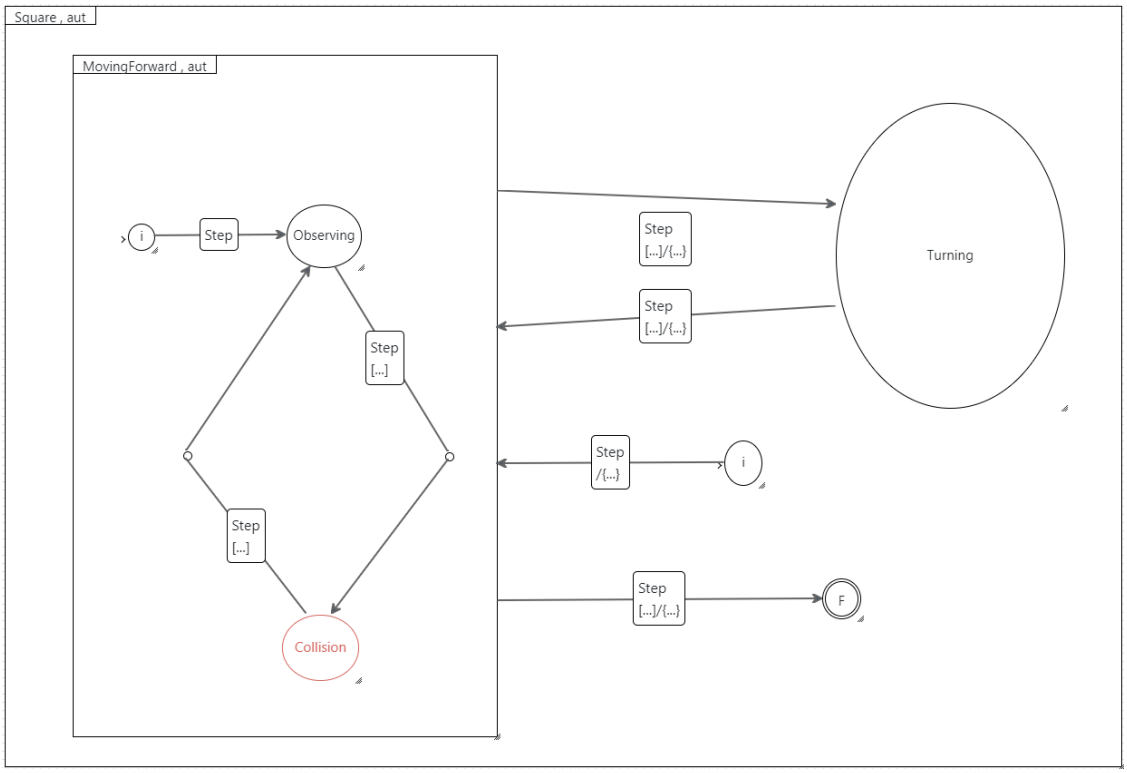

Figure 28. Case study - Square from RoboChart

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 [5]. 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 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 at Step intervals.

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 28.

The initial state of *Square* is i and its transition leads to state *MovingForward*. In this transition the timer 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.
State *MovingForward* entry code executes a function so the robot starts moving forward. The initial state of *MovingForward* is *i2*. 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*, it means that the robot is moving forward and

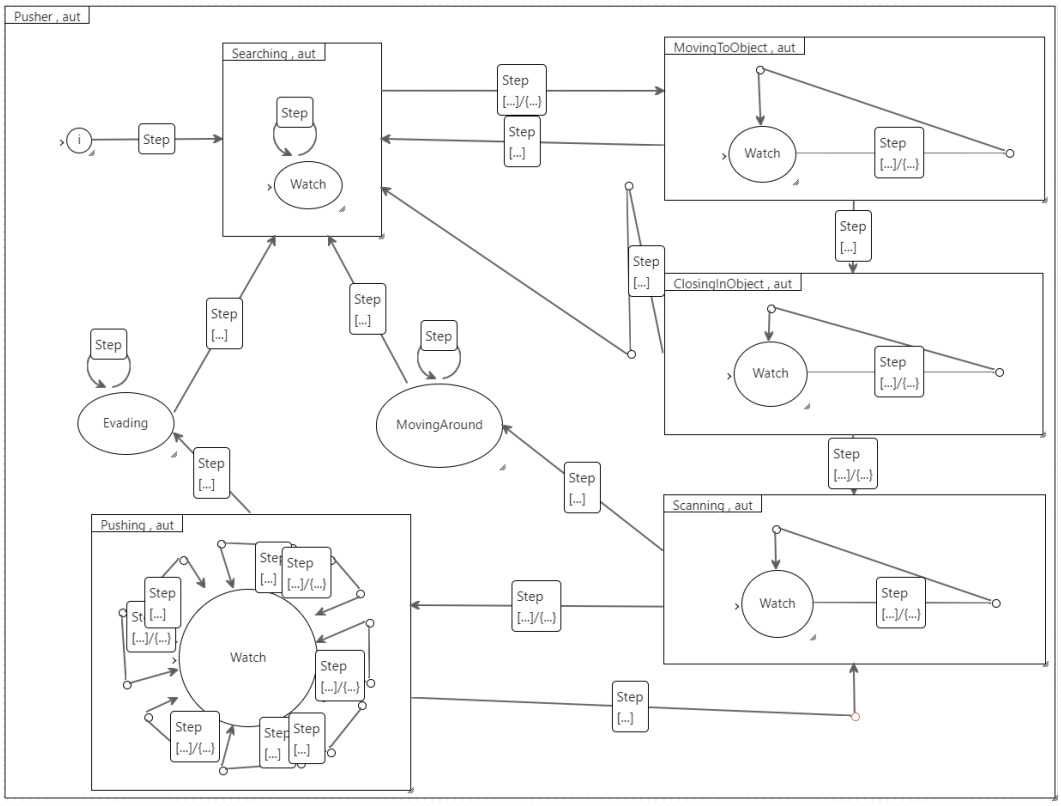

Figure 29. Case study - Transporter from RoboChart

observes its 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*, it means that the robot moved all the distance for this segment of the square, and it 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.

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, and make the ASTD deterministic. The other Step transitions does not compete with those because they are also guarded.

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 29.

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 *ClosingI-*

nObject. 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 30.

The initial state is *i* inside *Movement* automaton. The Step moves the active state to *i* inside *MovementAndAvoidance* automaton. The *MovementAndAvoidance* automaton is responsible for moving the robot forward and avoiding collisions. *MovementAndAvoidance* has an entry code that resets clock *MBD*. From *i* inside *MovementAndAvoidance* there are two allowed transitions, one Step that transits to *Move*, and one Step that changes the active state to *i* in *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*.

The automaton *Turning* 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.

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) [11]. FORM is a systematic method that looks for and captures commonalities and variability of a product line in terms of "features" [9].

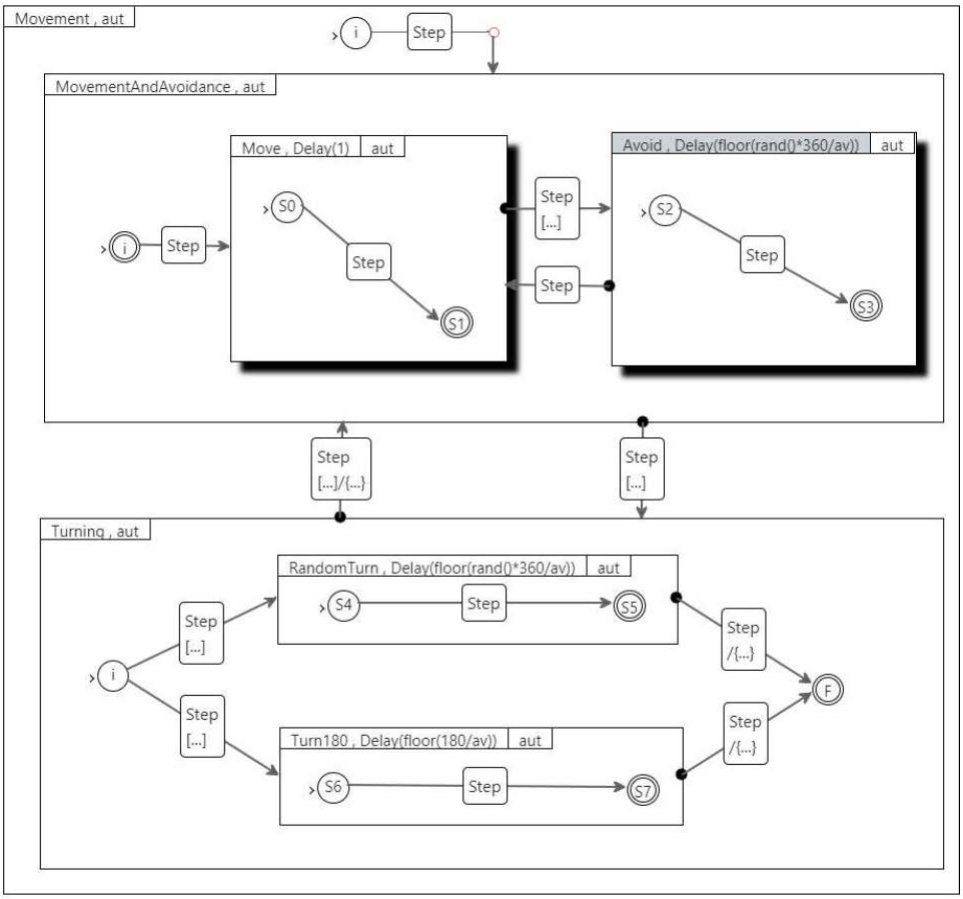

Figure 30. Case study - Alpha Algorithm from RoboChart

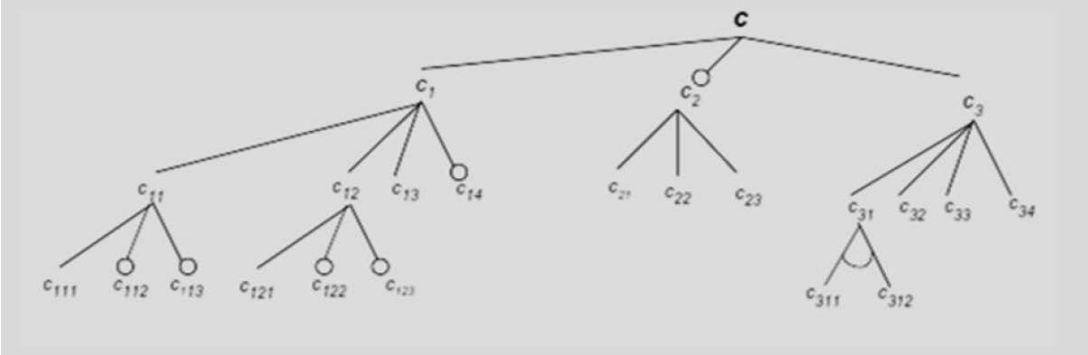

Figure 31. Case study - Functional perspective of real estate services and urban sanitizing model in FORM/BCS draw from [11]

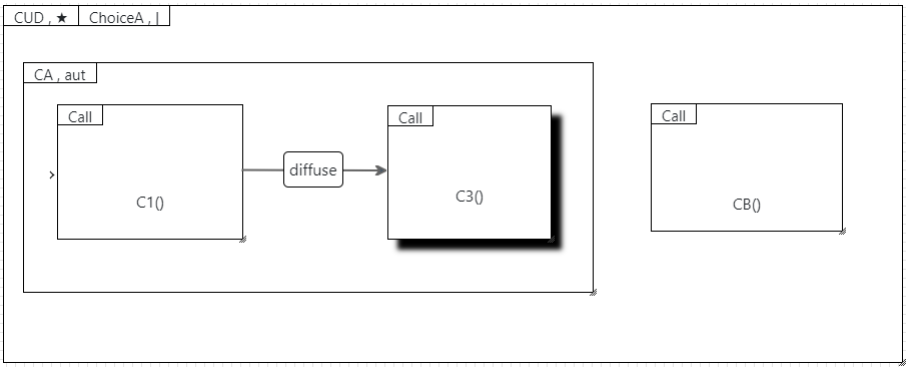

Figure 32. Case study FORM/BCS - CUD (Main ASTD)

The structure of the model is represented by Figure 31. Figure 31 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 31, C_2 has an empty circle that connects it to C, which 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 supports modularity, we decided to use a combination of a choice ASTD and call ASTD to simulate the tree presented in Figure 31. Figure 32 is the main ASTD, CUD. CA represents the run where C_2 does not act, and CB, Figure 33, 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 `https://depot-rech.fsci.usherbrooke.ca/gril/fram1801/castd_tests` .

Although this model does not use time constraints, it was important to test 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, and better modularise generated code.

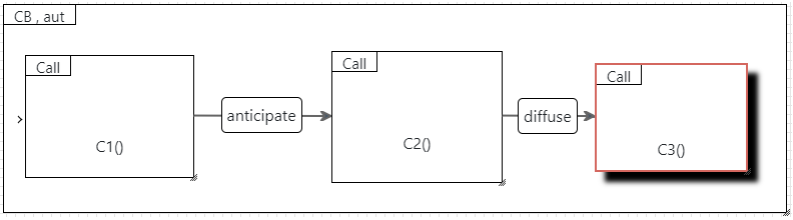

Figure 33. Case study FORM/BCS - CB

References

[1] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[2] James Baxter, Pedro Ribeiro, and Ana Cavalcanti. Sound reasoning in tock-csp. *Acta Informatica*, 2021.

[3] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[4] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*, pages 87–124. Springer, 2003.

[5] Ana Cavalcanti, Augusto Sampaio, Alvaro Miyazawa, Pedro Ribeiro, Madiel Conserva Filho, André Didier, Wei Li, and Jon Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019.

[6] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[7] Frank Houdek and Alexander Raschke. Adaptive exterior light and speed control system. In *International Conference on Rigorous State-Based Methods*, pages 281–301. Springer, 2020.

[8] MATLAB. Stateflow. `https://www.mathworks.com/products/stateflow.html`, 2020.

[9] Marcel Fouda Ndjodo and Amougou Ngoumou. The feature oriented reuse method with business component semantics. *Int. J. Comput. Sci. Appl.*, 6(4):63–83, 2009.

[10] 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)*, pages 146–155, Melbourne, Australia, 2018.

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

[12] Pedro Ribeiro, James Baxter, and Ana Cavalcanti. Priorities in tock-csp. *arXiv preprint arXiv:1907.07974*, 2019.

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

[14] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):1–29, 2013.

[15] Jun Sun, Yang Liu, Jin Song Dong, and Xian Zhang. Verifying stateful timed csp using implicit clocks and zone abstraction. In *International Conference on Formal Engineering Methods*, pages 581–600. Springer, 2009.

[16] Lionel N Tidjon, Marc Frappier, and Amel Mammar. Intrusion detection using astds. In *International Conference on Advanced Information Networking and Applications*, pages 1397–1411. Springer, 2020.

[17] Lionel Nganyewou Tidjon. *Formal modeling of intrusion detection systems.* PhD thesis, Institut Polytechnique de Paris; Université de Sherbrooke (Québec, Canada), 2020.