Proving Local Invariants in ASTDs*

Quelen Cartellier[1], Marc Frappier[1][0000−0002−4402−2514], and Amel Mammar[2][0000−0003−0016−6898]

[1] GRIC, Université de Sherbrooke, Sherbrooke, J1K2R1 Quebec, Canada
{Quelen.Cartellier,Marc.Frappier}@USherbrooke.ca
[2] SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris
91120 Palaiseau, France
amel.mammar@telecom-sudparis.eu

Abstract. This paper proposes a formal approach for generating proof obligations to verify local invariants in an Algebraic State Transition Diagram (ASTD). ASTD is a graphical specification language that allows for the combination of extended hierarchical state machines using CSP-like process algebra operators. Invariants can be declared at any level in a specification (state, ASTD), fostering the decomposition of system invariants into modular local invariants which are easier to prove, because proof obligations are smaller. The proof obligations take advantage of the structure of an ASTD to use local invariants as hypotheses. ASTD operators covered are automaton, sequence, closure and guard. Proof obligations are discharged using Rodin. When proof obligations cannot be proved, ProB can be used to identify counter-examples to help in correcting/reinforcing the invariant or the specification.

Keywords: ASTD, invariant, proof obligation, Rodin, ProB

1 Introduction and Related Work

ASTD [6,12] 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 [4,18] and control systems [1,5]. ASTD allows for the combination of state transition diagrams (Statecharts-like) with process algebra operators, drawn from CSP. Hence, ASTD takes advantage of the strengths of both notations: graphical representation, hierarchy, orthogonality, compositionality, and abstraction. Statecharts-like notations offer only two operators for decomposing behavior, OR and AND states. ASTDs support these two operators (OR is represented by ASTD automaton; AND is represented by the flow operator), and it supports most of CSP's operators. ASTDs differ from these notations by using simpler communication mechanisms. ASTDs can communicate through shared state variables or through synchronisation. Statecharts' broadcast communication is not supported.

In order to promote the use of ASTDs for modeling safety critical systems, it is crucial to have the ability to prove safety properties like invariants. Indeed, designing an ASTD specification is an error-prone task, *e.g.*, invariants of states can be incorrect.

* This work was supported by the ANR projet DISCONT, Public Safety Canada and NSERC

Model-based methods like B and Event-B offer powerful environments for proving invariants on formal specifications using refinement. But still, proving global invariants on these systems is hard [9,10]. A translation of ASTDs to B and Event-B has been proposed [11,5]. Global invariants which are associated to all states of the system can be declared and proof obligations are then generated to ensure that each event preserves the invariants. These proof obligations are hard to discharge for large specifications, due to the encoding of ASTD operators that introduces several control variables. Moreover, it does not support local invariants which are associated to some states in the system. In [1], it has been shown that the algebraic approach of ASTDs streamlines the modularisation of a large specification.

Several works have addressed the specification and verification of invariants in Statecharts-like notations (*e.g.*, [7,13,14]), but only [17,16] have addressed the proof of invariants; others are targeting model checking or assertions for run-time verification. Model-checking is often limited by state explosion (*e.g.*, [9,10]) for large specifications, whereas run-time verification is not satisfactory for safety-critical system, as it offers no insight on system correctness before deployment.

To overcome the limitations of [11,5], we propose in this paper to generate proof obligations for invariant preservation directly from an ASTD specification, in order to reduce proof complexity and make the traceability between the ASTD invariants and the produced proof obligations straightforward. An invariant can be declared by the user at any level in the specification (from complex ASTDs to elementary states of an automaton). Our work differs from [17] by permitting invariants on elementary automaton states, an important feature for critical control systems, and by supporting invariants for complex ASTDs, which amounts to supporting invariant for complex process expressions, since complex ASTDs are defined using process algebra operators. UML-B [16] supports invariants on classes and traditional states machines; a UML-B specification is translated into Event-B, and thus invariants are represented globally in the resulting Event-B machine. Our POs are local to a state, and thus simpler. They are represented as theorems of an Event-B context, and can be discharged using Rodin[3] and debugged using ProB [8], which are two industrial strengths tools supporting the B and Event-B methods.

The rest of this paper is structured as follows. Section 2 introduces a subset of the ASTD notation and its semantics. Section 3 defines the proof obligations and illustrates them on a small example. Section 4 concludes the paper.

2 Overview of the ASTD notation

The ASTD notation includes several ASTD *types*, which are Elem, Automaton, Sequence, Kleene closure, Guard, Choice, Parameterized Synchronization, Flow, Quantified choice, Quantified synchronization and Call. In this paper, we consider only the first five types. Figure 1 illustrates a simple but representative ASTD specification that is used to illustrate our approach throughout this paper. ASTDs A and C are of type Kleene closure. ASTD B is of type Sequence, and it executes ASTDs C and E in sequence. ASTD E is of type Guard. ASTDs D and F are of type Automaton. Automaton

[3] http://www.event-b.org/

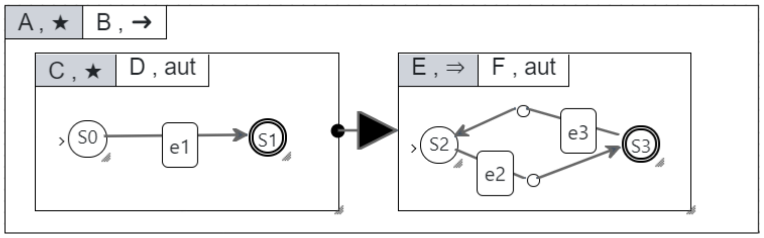

Fig. 1. ASTD Case study

states *S0*, *S1*, *S2*, *S3* are ASTDs of type Elem. ASTD *A* is a Kleene closure that allows for iteration on ASTD *B*. *B* executes *C* and *D* in sequence: when *C* reaches the final state *S1*, *E* is enabled to execute the event e2, to move from *S2* to *S3*. Since *C* is a Kleene closure, it can trigger a new iteration of *D* when it is in the state *S1*, and execute the event e1 from the initial state *S0*. When *F* is in the final state *S3*, *A* can also trigger a new iteration of *B* and execute the event e1 from *S0*. Here is a possible trace of this specification:

$$[\mathtt{e1}, \mathtt{e1}, \mathtt{e1}, \mathtt{e2}, \mathtt{e3}, \mathtt{e2}, \mathtt{e1}, \ldots]$$

ASTD types are organised into a type hierarchy. Specific ASTD types inherit from the top-level type ASTD, which introduces three general fields $\langle n, V, I\rangle$, where n is the name of the ASTD, V is a set of attributes and I an invariant associated to all states of the ASTD. These properties are inherited by all ASTD types. We refer to a field p of an ASTD $a \in$ ASTD using the notation $a.p$. Attributes $a.V$ are variables that are initialised a and modified in a or in its sub-ASTDs. For instance, an attribute declared in *A* can be modified in *B*, *C*, *D*, *E*, *F*. The invariant $a.I$ is a first-order logic formula on $a.V$ and on the attributes of its super-ASTDs. For instance, the invariant $F.I$ can refer to attributes $A.V$, $B.V$, $E.V$ and $F.V$. Table 1 provides the main elements of these ASTDs. If a is an elementary state, then $a.I$ applies only to this state. But if a is a complex ASTD, then $a.I$ should be fulfilled by all sub-ASTDs of a.

The execution of an ASTD is defined by a labeled transition system using a Plotkin-like operational semantics. The set of states is denoted by State. Each type of ASTD comes with its own type of states, but each state type has a property $E : \mathtt{Var} \rightarrow \mathtt{Term}$ which represents the values of attributes declared in the ASTD. Some states may be final and enable subsequent ASTDs to start. Final states of an ASTD a are determined by the Boolean function *final* of type ASTD × State → Boolean. Function *init* of type ASTD × ($\mathtt{Var} \rightarrow \mathtt{Term}$) → State returns the initial state of an ASTD. In the sequel, we use some mathematical operators from the B notation; their definition is given in Table 3.

2.1 Automaton

The ASTD type Automaton is built on a set of states related by transitions. It has the following structure: *ASTD Automaton* $\hat{=} \langle\, \mathsf{aut}, \Sigma, S, \nu, \delta, SF, DF, n_0 \,\rangle$ where $\Sigma \subseteq$ Event is the alphabet and $S \subseteq$ Name is the set of state names. $\nu \in S \rightarrow$ ASTD maps each state

ASTD	**Attributes**	**Initialisation**	**Invariant**	**Guard**
A	x_A	$x_A := 0$	$x_A \geq 0$	
B	x_B	$x_B := x_A + 1$	$x_B > 0$	
C	x_C	$x_C := 0$	$x_B \geq x_A$	
D	x_D	$x_D := x_C + 1$	$x_D \geq 0$	
E	x_E	$x_E := x_A$	$x_E \geq 0 \wedge x_E < x_B \wedge x_E \leq x_A$	$x_B > x_A + 4$
F	x_F	$x_F := 0$	$x_F \geq 0$	
$S0$			$x_D > x_C \wedge x_C \geq 0$	
$S1$			$x_C \geq x_D \wedge x_A > 0$	
$S2$			$x_F = 0 \vee x_A > x_E$	
$S3$			$x_F > 0 \wedge x_A > x_E$	

Table 1. Properties of ASTDs in Fig. 1

Event	**Guard**	**Action**
e1		$x_C := x_C + x_D;\ x_B := x_B + x_C;\ x_A := x_A + 1$
e2		$x_A := x_A + x_B;\ x_F := x_F + 1$
e3	$x_A < 10000$	$x_A := x_A - x_E$

Table 2. Transitions of ASTDs in Fig. 1

Description	**Expression**	**Definition**
domain antirestriction	$S \lhd\!\!\!- r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge x \notin S\}$
range antirestriction	$r \rhd\!\!\!- S$	$\{x \mapsto y \mid x \mapsto y \in r \wedge y \notin S\}$
override	$r_1 \Leftarrow\!\!\!+ r_2$	$(\mathsf{dom}(r_2) \lhd\!\!\!- r_1) \cup r_2$

Table 3. Definitions of B operators

to its sub-ASTD, which can be elementary (noted Elem) or complex (*i.e.*, of any ASTD type). 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 type of the transition. In this paper, we consider simple transitions of the form $\langle n_1, n_2 \rangle$, where n_1 and n_2 are respectively the source and target states of the transition. Symbol *final?* is a Boolean: when $\mathit{final?} = \mathsf{true}$, the source of the transition is graphically decorated with a bullet (*i.e.*, $\bullet$); it indicates that the transition can be fired only if n_1 is final. This is useful only when n_1 is not an elementary state (ie, it is a complex ASTD). $SF \subseteq S$ is the set of shallow final states, while $DF \subseteq S$ denotes the set of deep final states, with $DF \cap SF = \emptyset$ and $DF \subseteq \mathsf{dom}(\nu \rhd\!\!\!- \{\mathsf{Elem}\})$. A deep final state is final iff its sub-ASTD is final; a shallow final is final, irrespective of the state of its sub-ASTD. $n_0 \in S$ is the name of the initial state. In this paper, for the sake of simplicity, we denote by A_{tr} the sequential composition of the actions executed during a transition, that are, the actions executed when exiting the source state, on the transition and when entering the target state. The type of an Automaton state is $\langle \mathsf{aut}_\circ, n, E, s \rangle$ where $\mathsf{aut}_\circ$is a constructor of the Automaton state. $n \in \mathsf{S}$ denotes the name of the current state of the automaton. E contains the values of the Automaton attributes. $s \in \mathsf{State}$ is state of the sub-ASTD of n, when n is a complex state; $s = \mathsf{Elem}$ when n is elementary.

Automaton F defines the attribute x_F initialised by $(x_F := 0)$. The transition labelled with the event e2 permits to move from $S2$ to $S3$. When it is triggered, the action $(x_A := x_A + x_B; x_F := x_F + 1)$ is executed.

To define the semantics of an Automaton a, the functions *init* and *final* are defined as follows:

$$\begin{aligned} init(a, G) &\mathrel{\widehat{=}} (\mathsf{aut}_\circ, a.n_0, a.E_{init}(\!|G|\!), init(a.\nu(n_0), G \lessdot\!\!+ a.E_{init})) \\ final(a, (\mathsf{aut}_\circ, n, E, s)) &\mathrel{\widehat{=}} n \in a.SF \vee (n \in a.DF \wedge final(a.\nu(n), s)) \end{aligned}$$

where G and E_{init} denote respectively the environment (*i.e.*, current values of attributes defined in the enclosing ASTDs of a) and the initial values of the attributes of a, which may refer to variables declared in enclosing ASTDs, and thus they are replaced with their current values defined in G using the substitution operator $(\!|\ |\!)$ (eg, $(x_F := x_A + 1)(\!|x_A := 0|\!)) \equiv x_F := 1$). Note that the sub-ASTD of n_0 is initialised by recursively calling *init* on the ASTD of n_0.

Inference rule aut_1 defines the semantics of an automaton a for a transition between two states n_1 and n_2:

$$\mathsf{aut}_1\ \frac{a.\delta((n_1, n_2), \sigma', g, A_{tr}, final?) \qquad \Psi \qquad \Omega_{loc}}{(\mathsf{aut}_\circ, n_1, E, s_1) \xrightarrow{\sigma, E_e, E'_e}_a (\mathsf{aut}_\circ, n_2, E', init(a.\nu(n_2), E'))}$$

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'. The sub-ASTD of n_2, denoted by $a.\nu(n_2)$, is initialised. The premise provides that such a transition is possible if there is a matching transition, which is represented by $\delta((n_1, n_2), \sigma', g, A_{tr}, final?)$. σ' is the event labelling the transition, and it may contain variables. The value of these variables is given by the environment E_e, which contains the values of variables in ASTDs enclosing the automaton (*i.e.*, the super-ASTDs of a) and attributes of a, given in by E. This match on the transition is provided by the premise Ψ defined as follows.

$$\Psi \mathrel{\widehat{=}} \big(\ (final? \Rightarrow final(a.\nu(n_1), s)) \wedge g \wedge \sigma' = \sigma\ \big)(\!|E_g|\!)$$

Ψ can be understood as follows. If the transition is final (*i.e.*, $final? = \mathsf{true}$), then the current state s must be final with respect to the ASTD of n_1. 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. Environment E_g, defined as $E_e \lessdot\!\!+ E$, denotes the list of variables of a and its super-ASTDs. The premise Ω_{loc} determines how the new values of the attributes in the environment are computed when the transition occurs; its definition is omitted for the sake of concision.

Rule aut_2 handles transitions occurring within a complex automaton state n.

$$\mathsf{aut}_2\ \frac{s \xrightarrow{\sigma, E_g, E''_g}_{a.\nu(n)} s' \qquad \Theta}{(\mathsf{aut}_\circ, n, E, s) \xrightarrow{\sigma, E_e, E'_e}_a (\mathsf{aut}_\circ, n, E', s')}$$

The transition starts from a sub-state s and moves to the sub-state s' of the state n. Actions are executed bottom-up. E''_g denotes the values computed by the ASTD of the state n. Premise Θ determines how E''_g is computed, and it is reused in all subsequent rules where a sub-ASTD transition is involved; it is omitted here for the sake of concision and simplicity.

2.2 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. The Kleene closure ASTD has the following structure:

$$\mathsf{Kleene\ closure} \mathrel{\widehat{=}} \langle \bigstar, b \rangle$$

where $b \in \mathsf{ASTD}$ is the body of the closure. A Kleene closure is in a final state when it has not started or when its sub-ASTD b is in a final state. The type of a Kleene closure state is $\langle \bigstar_\circ, E, started?, s \rangle$ where $s \in \mathsf{State}$, $started?$ is a Boolean indicating whether the first iteration has been started. It is essentially used to determine if the closure can immediately exit (*i.e.*, if it is in a final state) without any iteration. For a Kleene closure ASTD a, the initial and final states are defined as follows.

$$\begin{aligned} init(a, G) &\mathrel{\widehat{=}} (\bigstar_\circ, a.E_{init}(\!\![G]\!\!), \mathsf{false}, \bot) \\ final(a, (\bigstar_\circ, E, started?, s)) &\mathrel{\widehat{=}} \neg started? \vee final(a.b, s) \end{aligned}$$

where $\bot$ denotes an undefined state. The semantics of a Kleene closure ASTD is defined by two inference rules: $\bigstar_1$ allows for starting a new iteration (including the first one); $\bigstar_2$ allows for execution on the sub-ASTD.

$$\bigstar_1 \ \frac{final(a, (\bigstar_\circ, E, started?, s)) \qquad init(a.b, E_e) \xrightarrow{\sigma, E_g, E_g''}_{a.b} s' \qquad \Theta}{(\bigstar_\circ, E, started?, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\bigstar_\circ, E', \mathsf{true}, s')}$$

$$\bigstar_2 \ \frac{s \xrightarrow{\sigma, E_g, E_g''}_{a.b} s' \qquad \Theta}{(\bigstar_\circ, E, \mathsf{true}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\bigstar_\circ, E', \mathsf{true}, s')}$$

In Figure 1, C is a Kleene closure ASTD whose initial state is $(\bigstar_\circ, (\!\![x_A = 0, x_B = 1, x_C = 1]\!\!), \mathsf{false}, \bot)$; its sub-state is undefined (denoted by $\bot$). But when the first possible event is received (*i.e.*, e1), the sub-ASTD D is initialised ($x_D := x_C + 1$), the transition e1 is triggered from *S0* and the action ($x_C = x_C + x_D; x_B = x_B + x_C; x_A = x_A + 1$) of the transition is executed. The current state is now *S1* and the values of attributes are $(\!\![x_A = 1, x_B = 4, x_C = 3, x_D = 2]\!\!)$. As *S1* is a final state of D (*i.e.*, the sub-ASTD of C), D is final, and so is C, and a new iteration of D can be started again by receiving e1. D is reinitialised to start a new iteration, so x_D is reinitialised prior to this new transition, but the values of x_A, x_B, x_C are unaffected by the initialisation of D.

2.3 Sequence

The Sequence ASTD allows for the sequential composition of two ASTDs. When the first ASTD reaches a final state, it enables the execution of the second ASTD. In that case, it is the reception of the next event that determines which ASTD is executed: if both the first and the second can execute it, then a non-deterministic choice is made between the two. When the second ASTD starts it execution, the first ASTD becomes disabled. The Sequence ASTD enables decomposing problems into a set of tasks that have to be executed in sequence. The Sequence ASTD has the following structure:

$$\mathsf{Sequence} \mathrel{\hat{=}} \langle \rightarrowtriangle, \mathit{fst}, \mathit{snd}\rangle$$

where *fst* and *snd* are ASTDs denoting respectively the first and second sub-ASTD of the Sequence. A Sequence state is of type $\langle \rightarrowtriangle_\circ, E, [\mathsf{fst} \mid \mathsf{snd}], s\rangle$, where $\rightarrowtriangle_\circ$ is a constructor of the Sequence state, $[\mathsf{fst} \mid \mathsf{snd}]$ is a choice between two markers that respectively indicate whether the Sequence is in the first sub-ASTD or the second sub-ASTD and $s \in \mathsf{State}$. Since s does not indicate which ASTD is currently executed, the marker $[\mathsf{fst} \mid \mathsf{snd}]$ is used for that purpose. Functions *init* and *final* of a sequence ASTD are defined as follows.

$$\begin{aligned}
\mathit{init}(a, G) &\mathrel{\hat{=}} (\rightarrowtriangle_\circ, a.E_{init}(\![G]\!), \mathsf{fst}, \mathit{init}(a.\mathit{fst}, G \mathbin{\lhd\!\!+} a.E_{init}))\\
\mathit{final}(a, (\rightarrowtriangle_\circ, E, \mathsf{fst}, s)) &\mathrel{\hat{=}} \mathit{final}(a.\mathit{fst}, s) \wedge \mathit{final}(a.\mathit{snd}, \mathit{init}(a.\mathit{snd}, E))\\
\mathit{final}(a, (\rightarrowtriangle_\circ, E, \mathsf{snd}, s)) &\mathrel{\hat{=}} \mathit{final}(a.\mathit{snd}, s)
\end{aligned}$$

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, or (ii) it is executing the second sub-ASTD which is in a final state.

In Figure 1, the ASTD B is a Sequence ASTD that allows the sequential execution of ASTDs C and E. B starts by executing C. As the initial state of E is not final, B is final only when the final state of E is reached.

Three semantic rules are necessary to define the execution of the Sequence. Rule $\rightarrowtriangle_1$ deals with transitions on the sub-ASTD *fst* only. Rule $\rightarrowtriangle_2$ deals with transitions from *fst* to *snd*, when *fst* is in a final state. Rule $\rightarrowtriangle_3$ deals with transitions on the sub-ASTD *snd*. Note that the arrow connecting ASTD C and E is not labeled with an event pattern, because event patterns only occur on automaton transitions; when the execution goes from C to E, it is an event of E that is executed, in this case an event of automaton F.

$$\rightarrowtriangle_1\ \frac{s \xrightarrow{\sigma, E_g, E_g''}_{a.\mathit{fst}} s' \qquad \Theta}{(\rightarrowtriangle_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\rightarrowtriangle_\circ, E', \mathsf{fst}, s')}$$

$$\rightarrowtriangle_2\ \frac{\mathit{final}(a.\mathit{fst}, s) \qquad \mathit{init}(a.\mathit{snd}, E_e) \xrightarrow{\sigma, E_g, E_g''}_{a.\mathit{snd}} s' \qquad \Theta}{(\rightarrowtriangle_\circ, E, \mathsf{fst}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\rightarrowtriangle_\circ, E', \mathsf{snd}, s')}$$

$$\rightarrowtriangle_3\ \frac{s \xrightarrow{\sigma, E_g, E_g''}_{a.\mathit{snd}} s' \qquad \Theta}{(\rightarrowtriangle_\circ, E, \mathsf{snd}, s) \xrightarrow{\sigma, E_e, E_e'}_{a} (\rightarrowtriangle_\circ, E', \mathsf{snd}, s')}$$

In Figure 1, in the sequence ASTD B, the ASTD E can be executed only when the ASTD C reaches its final state, that is, it is not started at all or it is in the state *S1*. The first event executed in E is e2 because this is its only event that starts from its initial state.

2.4 Guard

A Guard ASTD defines a conditional execution of its sub-ASTD using a predicate. To be enabled, the first event executed must satisfy the Guard predicate. Once the guard has been satisfied by the first event, the sub-ASTD of the guard executes the subsequent events without further constraints from its enclosing guard ASTD. The guard predicate can only refer to attributes declared in its enclosing ASTDs. The Guard ASTD has the following structure:

$$\mathsf{Guard} \mathrel{\widehat{=}} \langle \Rightarrow, g, b \rangle$$

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 $\mathit{started}?$ states whether the first transition has been done, $s \in \mathsf{State}$. The initial and final states of a Guard ASTD a are defined as follows.

$$\begin{aligned} \mathit{init}(a, G) &\mathrel{\widehat{=}} (\Rightarrow_\circ, a.E_{\mathit{init}}(\![G]\!), \mathsf{false}, \bot) \\ \mathit{final}(a, (\Rightarrow_\circ, E_{\mathit{init}}, \mathsf{false}, s)) &\mathrel{\widehat{=}} \mathit{final}(a, s) \\ \mathit{final}(a, (\Rightarrow_\circ, E, \mathsf{true}, s)) &\mathrel{\widehat{=}} \mathit{final}(a, s) \end{aligned}$$

The semantic of the Guard ASTD is defined by two inference rules: $\Rightarrow_1$ deals with the first transition and the satisfaction of the guard predicate; $\Rightarrow_2$ deals with subsequent transitions.

$$\Rightarrow_1 \frac{g(\![E_e]\!) \qquad \mathit{init}(a.b, E_e) \xrightarrow{\sigma, E_g, E_g''}_{a.b} s' \qquad \Theta}{(\Rightarrow_\circ, E_{\mathit{init}}, \mathsf{false}, \mathit{init}(a.b, E_e)) \xrightarrow{\sigma, E_e, E_e'} (\Rightarrow_\circ, E', \mathsf{true}, s')}$$

$$\Rightarrow_2 \frac{s \xrightarrow{\sigma, E_g, E_g''}_{a.b} s' \qquad \Theta}{(\Rightarrow_\circ, E, \mathsf{true}, s) \xrightarrow{\sigma, E_e, E_e'} (\Rightarrow_\circ, E', \mathsf{true}, s')}$$

Let us use the ASTD of Fig. 1 to explain when ASTDs are initialised in a sequence ASTD. Suppose that the system is in the state $\mathit{S1}$ and that the event e2 is received. Since $\mathit{S1}$ is a final state of D and C, the rule $\twoheadrightarrow_2$ allows for the execution of the transition e2 from the initial state of F. To trigger e2, the rule $\Rightarrow_2$ requires that the guard ($x_B > x_A$+4) of E must be satisfied with the current values of x_A and x_B. Finally, if the transition e2 had a guard, it should also be satisfied with the current values of its enclosing ASTDs A, B, E and F. The variables in E and F are initialised only when the transition e2 is evaluated. If the guards are satisfied, e2 is executed and the state moves to $\mathit{S3}$; if not, the system stays in $\mathit{S1}$, and then E and F will be initialised again when a new occurrence of e2 is received. In other words, E and F are initialised for good only when the first transition of F can be executed.

3 Proof obligations for invariant satisfaction

In this section, we describe a systematic approach for generating the proof obligations that ensure the satisfaction of the invariants of an ASTD for its reachable states defined by the transition system. Proof obligations are generated according to the structure of the ASTD. Hereafter, we introduce the definitions of some concepts that we use in the sequel of the paper.

3.1 Definitions

We introduce the following definitions that are used as hypotheses when proving an invariant.

Definition 1. *The full invariant of an* ASTD *a is defined as follows:*

$$Inv_{full}(a) = \begin{cases} a.I & \textit{if type(a)=Elem} \\ a.I \wedge \left(\bigvee_{s \in a.S} Inv_{full}(a.\nu(s))\right) & \textit{if type(a)=Automaton} \\ a.I \wedge (Inv_{full}(fst) \vee Inv_{full}(snd)) & \textit{if } a \mathrel{\widehat{=}} (\twoheadrightarrow, fst, snd) \\ a.I \wedge Inv_{full}(b) & \textit{if } a \in \{(\bigstar, b), (\Rightarrow, g, b)\} \end{cases}$$

$Inv_{full}(a)$ denotes the conjunction of $a.I$ and the invariants of its sub-ASTDs. When a contains several sub-ASTDs, we take the disjunction of their invariants, because the sub-state of a is in one of them.

Definition 2. *The invariant of the final states of an* ASTD *a is defined as follows:*

$$Inv_F(a) = \begin{cases} a.I & \textit{if type(a)=Elem} \\ a.I \wedge \begin{pmatrix} \bigvee_{n \in a.SF} Inv_{full}(a.\nu(n)) \\ \vee \\ \bigvee_{n \in a.DF} Inv_F(a.\nu(n)) \end{pmatrix} & \textit{if type(a)=Automaton} \\ a.I \wedge Inv_F(snd) & \textit{if } a \mathrel{\widehat{=}} (\twoheadrightarrow, fst, snd) \\ a.I \wedge Inv_F(b) & \textit{if } a \mathrel{\widehat{=}} (\Rightarrow, g, b) \\ a.I & \textit{if } a \mathrel{\widehat{=}} (\bigstar, b) \end{cases}$$

$Inv_F(a)$ is the conjunction of $a.I$ and the disjunction of the invariants of its final states. A final state of an automaton can be deep or shallow. For a shallow final state n of a, we take the full invariant of n, because according to the definition of *final*, n is final irrespective of its current sub-state, so its sub-state can be in any of its sub-ASTDs. For a deep final state n, we know that it is final when its sub-ASTD is final, so we recursively call Inv_F on n to get only the invariants of its final states. For a sequence, it suffices to take into account $Inv_F(snd)$, because when a sequence is in its first ASTD, the initial state of the second ASTD must also be final, and it becomes a special of the second case. In addition, for a Kleene clsoure, the state is final while not started. Therefore, in that particular case, $Inv_F(a) = a.I$.

3.2 Proof obligation generation

To ensure that an ASTD is correct, we have to establish that the invariant of each reachable state is fulfilled. To generate the PO related to the correctness of an ASTD, we distinguish two cases:

– Initialisation: the state is determined by the *init* function at the initialisation of an ASTD;
– Transition: the state is reached through a transition.

We define in the following sections two functions to generate proof obligations, one for the initialisation, and another for the transitions. These functions recursively traverse an ASTD to generate POs for all of its invariants declared in its sub-ASTDs.

3.3 Proof obligations for initialisations

Following the semantics of ASTDs, initialisations are done from the main ASTD down to its sub-ASTDs. Therefore we introduce a recursive function $PO_i(a, J, H, Act)$ to generate proof obligations for initialisations where:

- a stands for the ASTD whose POs for initialisation are generated.
- J stands for the conjunction of all the invariants of enclosing ASTDs of a; it provides information on the values of the variables occurring in the initialisation expression Act. It will be used as an hypothesis when proving an invariant of a to provide properties of variables which are not initialised by Act.
- H contains the hypotheses obtained from enclosing ASTDs that are needed for the initialisation of the subsequent steps of a sequential execution; it is used in the Kleene closure and the sequence ASTDs, because the values of the enclosing variables are determined by the final states of the last executed ASTD.
- Act stands for all the actions that are executed before executing the initialisation of ASTD a, through the *init* functions or the transitions that lead to a complex state of an Automaton.

To generate the PO of the initialisation of the main (*i.e.*, root) ASTD a of a specification and those of all its sub-ASTDs, the following call to PO_i is used: $PO_i(a, \mathsf{true}, \mathsf{true}, \mathsf{skip})$. Hereafter, we give the definition of PO_i according to each type of ASTD. We illustrate them with the example of Fig. 1. We argue on the correction of these POs with respect to the semantics of the ASTDs.

Kleene closure initialisation

Let a be a Kleene closure ASTD on an ASTD b: $a \mathrel{\widehat{=}} (\bigstar, b)$. In that case, we have to prove that $a.J$ is verified and, for each iteration of the Kleene closure, the invariant of the initial state of b is verified too. This is expressed by the following proof obligation:

$$\begin{aligned} PO_i(a,J,H,Act) \mathrel{\widehat{=}}\ & \{H \Rightarrow [Act; Init(a)](a.I)\} \cup && (i)\\ & PO_i(b,(J \wedge a.I), H, (Act; Init(a))) \cup && (ii)\\ & PO_i(b,(J \wedge a.I),(J \wedge a.I \wedge Inv_F(b)), \mathsf{skip}) && (iii) \end{aligned}$$

(i) PO (i) aims at verifying that $a.I$ holds after executing the initialisation $Init(a)$ of a following the initialisation actions Act executed in enclosing ASTDs. H provides the properties of variables which are not affected by $(Act; Init(a))$. Invariants of enclosing ASTDs do not have to be proved again, because $Init(a)$ does not modify variables of enclosing ASTDs.

(ii) PO (ii) is related to the first iteration of the ASTD b. $a.I$ is added to the invariant of enclosing ASTDs of b. $Init(a)$ is added to the sequence of actions that have been executed.

(iii) PO (iii) corresponds to the second and next iterations of b. As the next iterations will happen from the final state of b, the value of the variables of the enclosing ASTDs, which are in the initialisation of b, are described by $J \wedge a.I$ and the final values of these variables at the end of b, given by $Inv_F(b)$; thus, these two formulas are conjoined and passed as the value of H for proving the invariant of b. skip is used as the set of previous actions executed (parameter Act), because H denotes what is known about the values of the variables of the enclosing ASTDs.

To illustrate these definitions, consider the following call to compute the POs for the initialisation of ASTD A of Fig. 1: $PO_i(A, \mathsf{true}, \mathsf{true}, \mathsf{skip})$. It generates the following PO:

$$\mathsf{true} \Rightarrow [Init(A)(A.I)$$

which is reduced to $\{(0 \geq 0)\}$ after applying the substitutions. In addition, it generates the following two recursive calls:

1. $PO_i(B, A.I, \mathsf{true}, Init(A))$
2. $PO_i(B, A.I, (A.I \wedge Inv_F(B)), \mathsf{skip})$;

where $Inv_F(B) = B.I \wedge E.I \wedge F.I \wedge S3.I$, because B has only one final state, $S3$.

Sequence initialisation

Let a be a Sequence ASTD: $a \mathrel{\widehat{=}} (\twoheadrightarrow, fst, snd)$. In that case, we have to prove that the invariant of the initial states of fst and snd are fulfilled when these states are reached. The generated POs are:

$$\begin{aligned} PO_i(a, J, H, Act) \mathrel{\widehat{=}} \; & \{H \Rightarrow [Act; Init(a)](a.I)\} \cup & (i) \\ & PO_i(fst, (J \wedge a.I), H, (Act; Init(a))) \cup & (ii) \\ & PO_i(snd, (J \wedge a.I), (J \wedge a.I \wedge Inv_F(fst)), \mathsf{skip}) & (iii) \end{aligned}$$

(i) This PO follows the same pattern as case (i) of a Kleene closure.

(ii) This PO is related to the initialisation of the ASTD fst which occurs at the start of the sequence ASTD. It follows the same pattern as case (ii) of a Kleene closure.

(iii) This PO corresponds to the initialisation of the ASTD snd. According to the rule $\twoheadrightarrow_2$ (see Section 2.3), the initialisation of snd can occur only when fst is in a final state. Therefore, the PO is generated by taking $(J \wedge a.I \wedge Inv_F(fst))$ as hypothesis and skip as previous action since no additional action is executed when moving from the ASTD fst into snd in a sequence ASTD.

To illustrate these definitions, consider the following call to compute the POs for the initialisation of ASTD B of Fig. 1: $PO_i(B, A.I, \mathsf{true}, Init(A))$. By applying the definitions, we obtain one generated PO and two recursive calls:

(i) $true \Rightarrow [Init(A); Init(B)] \; B.I)$: after substitution, we obtain the PO: $\{(0 + 1 > 0)\}$

(ii) $PO_i(C, (A.I \wedge B.I), \mathsf{true}, (Init(A); Init(B)))$;

(iii) $PO_i(E, (A.I \wedge B.I), (A.I \wedge B.I \wedge C.I), \mathsf{skip})$: where $C.I = Inv_F(C)$ because C is a Kleene closure.

Guard initialisation

Let $a \mathrel{\widehat{=}} (\Rightarrow, g, b)$ be a Guard ASTD on an ASTD b. The invariant of the initial state of b is verified by the following proof obligation:

$$\begin{aligned} PO_i(a, J, H, Act) \mathrel{\widehat{=}} \; & \{H \Rightarrow [Act; Init(a)](a.I)\} \cup & (i) \\ & PO_i(b, (J \wedge a.I), H, (Act; Init(a))) & (ii) \end{aligned}$$

(i) This PO follows the same pattern as in case (i) for a Kleene closure.
(ii) This PO is related to the initialisation of the sub-ASTD b. It follows the same pattern as case (ii) of a Kleene closure and a sequence.

Note that the guard predicate g is not used in the generated POs. According to the guard semantics given by rules $\Rightarrow_1$ and $\Rightarrow_2$, the guard only applies to the first transition of b. The initialisation of a guard ASTD is executed before g is evaluated in the first transition of the guard body. Therefore, no information can be obtained from g in a guard ASTD initialisation. To illustrate these definitions, consider the call (iii) from the previous section:

$$PO_i(E, (A.I \wedge B.I), (A.I \wedge B.I \wedge C.I), \mathsf{skip})$$

It generates the following PO and one recursive call:

(i) $(A.I \wedge B.I \wedge C.I) \Rightarrow [Init(E)]\ (E.I)$: After substitution, we obtain the PO:

$$(A.I \wedge B.I \wedge C.I) \Rightarrow (x_A \geq 0 \wedge x_A < x_B \wedge x_A \leq x_A)$$

(ii) $PO_i(F, (A.I \wedge B.I \wedge E.I), (A.I \wedge B.I \wedge C.I), Init(E))$;

Automaton initialisation

Let a be an automaton with $a.n_0$ as initial state. Automaton initialisation POs are generated as follows:

$$\begin{aligned} PO_i(a, J, H, Act) \mathrel{\widehat{=}} \; & \{H \Rightarrow [Act; Init(a)](a.I)\} \cup & (i) \\ & PO_i(a.\nu(n_0), (J \wedge a.I), H, (Act; Init(a))) & (ii) \end{aligned}$$

(i) This PO follows the same pattern as in case (i) for a Kleene closure.
(ii) This PO is related to the initialisation of the initial state of a. This initialisation occurs at the start of the $\mathsf{Automaton}$ ASTD according to the syntax. It follows the same pattern as case (ii) of a Kleene closure, sequence and guard.

The initial state $a.n_0$ could be an elementary state (*i.e.*type$(a.n_o) = \mathsf{Elem}$). Therefore we introduce a PO for a call on an elementary state n as follows:

$$PO_i(n, J, H, Act) = \{H \Rightarrow [Act](n.I)\}$$

This PO follows the same pattern as the first generated PO for a complex ASTD, except there is no additional action of initialisation. That is because an elementary state does not initialise variables.

In Fig. 1, F is an Automaton with an elementary state as initial state. With the call (ii) from the Guard initialisation:

$$PO_i(F, (A.I \wedge B.I \wedge E.I), (A.I \wedge B.I \wedge C.I), Init(E))$$

we obtain two generated POs:

(i) $(A.I \wedge B.I \wedge C.I) \Rightarrow [Init(E); Init(F)](F.I)$: After substitution, we obtain the PO:

$$(A.I \wedge B.I \wedge C.I) \Rightarrow (0 \geq 0)$$

(ii) $PO_i(S2, (A.I \wedge B.I \wedge E.I \wedge F.I), (A.I \wedge B.I \wedge C.I), (Init(E); Init(F)))$: we apply the definition of PO_i for an elementary state, generating the following formula:

$$\{\ (A.I \wedge B.I \wedge C.I) \Rightarrow [Init(E); Init(F)](S2.I)\ \}$$

After substitution, we obtain the PO:

$$\{\ (A.I \wedge B.I \wedge C.I) \Rightarrow (0 = 0 \vee x_A > x_A)\ \}$$

3.4 Proof obligations for local transitions

When a transition t is triggered, it makes the system move from a source state n_1 to a target state n_2. So, we have to verify that the invariant of n_2 and those of its enclosing ASTDs are fulfilled. To this aim, we take as hypotheses the invariant of n_1 and those of its enclosing ASTDs. To get the set of POs associated with transitions, we introduce the recursive function $PO_{tr}(a, J)$ where:

- a stands for the ASTD whose POs for transitions are generated.
- J stands for all the invariants from the enclosing ASTD of a. Besides $a.I$, a must verify J.

The POs associated with the transitions of the main (ie, root) ASTD a are generated by calling PO_{tr} as follows: $PO_{tr}(a, \mathsf{true})$. Hereafter, we give the definition of PO_{tr} according to each type of ASTD:

$PO_{tr}(a, J) =$

(i) If $a \mathrel{\widehat{=}} (\twoheadrightarrow, fst, snd)$: $\quad PO_{tr}(fst, J \wedge a.I) \cup PO_{tr}(snd, J \wedge a.I)$

(ii) If $a \in \{(\bigstar, b), (\Rightarrow, g, b)\}$: $\quad PO_{tr}(b, J \wedge a.I)$

(iii) If type(a)=Automaton:

$$\bigcup s \cdot (s \in a.S \wedge a.\nu(s) \neq \mathsf{Elem}) \mid (PO_{tr}(s, J \wedge a.I)) \qquad \text{(iii-1)}$$

$$\begin{aligned} &\cup \bigcup \tau \cdot (\tau \in a.\delta) \mid && \text{(iii-2)}\\ &\quad (\{J \wedge a.I \wedge H_{\tau.final?}(\tau.\eta.n_1) \wedge \tau.g \Rightarrow [\tau.A_{tr}](J \wedge a.I)\} \cup && \text{(iii-2.1)}\\ &\quad PO_i(\nu(\tau.\eta.n_2), J \wedge a.I, J \wedge a.I \wedge H_{\tau.final?}(\tau.\eta.n_1) \wedge \tau.g, \tau.A_{tr})) && \text{(iii-2.2)} \end{aligned}$$

where: $\quad H_{true} \mathrel{\widehat{=}} Inv_F \quad \& \quad H_{false} \mathrel{\widehat{=}} Inv_{full}$

(i), (ii) According to the ASTD syntax, transitions only occur in an Automaton. Thus, in a Sequence, Kleene closure or Guard ASTD, recursive calls on PO_{tr} are done on sub-ASTDs.

(iii-1) In an Automaton, there are states and transitions. For each states, if the state is a complex ASTD *i.e.*, is not elementary, then a recursive call on PO_{tr} must be done on this sub-ASTD to check all Automaton ASTD.

(iii-2) For each transition τ in an Automaton ASTD, POs have to be generated to verify that invariants of the target state are fulfilled after executing action $\tau.A_{tr}$.

(iii-2.1) This PO aims at verifying the preservation of invariants of enclosing ASTDs through the action of the transition. All invariants from the source state are gathered as hypotheses, as well as the guard $\tau.g$. A distinction is made on the *final?* property of the transition using term $H_{\tau.final?}$ because it determines if the previous state was final or not. For a final transition, we use Inv_F, providing more precise hypotheses for the proof, otherwise we use Inv_{full}.

(iii-2.2) This PO aims at verifying that the local invariant of the target state is fulfilled after the execution of the transition. A recursive call to PO_i is done because the target state could be a complex ASTD, so it is initialised with the transition as premises.

In Fig. 1, F is an Automaton ASTD. The call (obtained after going down the recursion from A)

$$PO_{tr}(F, (A.I \wedge B.I \wedge E.I))$$

We obtain four generated POs:

1. $(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S2.I) \Rightarrow [Act(\text{e2})](A.I \wedge B.I \wedge E.I \wedge F.I)$:
   After substitution, we obtain the PO:
   $(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S2.I) \Rightarrow$
   $(x_A + x_B \geq 0 \wedge x_B > 0 \wedge (x_E \geq 0 \wedge x_E < x_B \wedge x_E \leq x_A + x_B) \wedge x_F + 1 \geq 0)$
2. $PO_i(S3, (A.I \wedge B.I \wedge E.I \wedge F.I), (A.I \wedge B.I \wedge E.I \wedge F.I \wedge S2.I), Act\text{e2})$:
   We apply the formula for an elementary state:
   $(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S2.I) \Rightarrow [Act(\text{e2})](S3.I)$:
   After substitution, we obtain the PO:
   $(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S2.I) \Rightarrow (x_F + 1 > 0 \wedge x_A + x_B > x_E)$
3. $(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S3.I \wedge Guard(\text{e3})) \Rightarrow [Act(\text{e3})](A.I \wedge B.I \wedge E.I \wedge F.I)$:
   After substitution, we obtain the PO:
   $(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S3.I \wedge Guard(\text{e3}))$
   $\Rightarrow (x_A - x_E \geq 0 \wedge x_B > 0 \wedge (x_E < x_B \wedge x_E \geq 0 \wedge x_E \leq x_A - x_E) \wedge x_F \geq 0)$
4. $PO_i(S2, (A.I \wedge B.I \wedge E.I \wedge F.I), (A.I \wedge B.I \wedge E.I \wedge F.I \wedge S3.I \wedge Guard(\text{e3})), Act\text{e3})$:
   We apply the formula for an elementary state:
   $\{(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S3.I \wedge Guard(\text{e3})) \Rightarrow [Act(\text{e3})](S2.I)\}$:
   After substitution, we obtain the PO:
   $\{(A.I \wedge B.I \wedge E.I \wedge F.I \wedge S3.I \wedge Guard(\text{e3})) \Rightarrow (x_F = 0 \vee x_A - x_E > x_E)\}$

3.5 Proving proof obligations and strengthening invariants

In this section, we describe how to verify the generated proof obligations using Rodin and how to reinforce invariants using ProB when POs are unprovable. Using Rodin, proof obligations are represented as theorems of an EVENT-B contexts. Since variables of an ASTD are typed in their declaration (*i.e.*, $x : T$), we add this type in the invariant of the ASTD using $x \in T \wedge a.I$. Free variables are universally quantified in each PO when defining them as theorems.

The PO generated for the initialisation of the ASTD E is the following:

$$(A.I \wedge B.I \wedge C.I) \Rightarrow [Init(E)](E.I)$$

Adding types of variables, replacing invariants names by their definitions and adding quantifiers on free variables, we obtain the following formula which is added as a theorem in a RODIN context:

$$\begin{aligned}\forall x_A \cdot \forall x_B \cdot \forall x_C \cdot (\ & ((x_A \in \mathbb{Z} \wedge x_A \geq 0) \wedge (x_B \in \mathbb{Z} \wedge x_B > 0) \wedge \\ & (x_B > x_A \wedge x_C \in \mathbb{Z})) \\ & \Rightarrow \\ & (x_A \in \mathbb{Z} \wedge x_A \geq 0 \wedge x_A < x_B \wedge x_A \leq x_A)))\end{aligned}$$

This theorem is trivial to prove, since the goal consists of trivial properties or formulas available in the hypotheses.

The following PO, related to the initialisation of the ASTD D after more than one iteration of the closure ASTD C, is less trivial and requires some deduction rules which are automatically applied by the Rodin provers.

$$\begin{aligned}\forall x_A \cdot \forall x_B \cdot \forall x_C \cdot \forall x_D \cdot (\ & ((x_A \in \mathbb{Z} \wedge x_A \geq 0) \wedge (x_B \in \mathbb{Z} \wedge x_B > 0) \wedge \\ & (x_C \in \mathbb{Z} \wedge x_B > x_A) \wedge \\ & (x_D \in \mathbb{Z} \wedge x_D > 0) \wedge (x_C \geq x_D \wedge x_A > 0)) \\ & \Rightarrow \\ & (x_C + 1 \in \mathbb{Z} \wedge x_C + 1 > 0))\end{aligned}$$

As hypotheses, we have $x_C \in \mathbb{Z}$ and $x_C \geq x_D \wedge x_D > 0$ so the goal $(x_C + 1 \in \mathbb{Z} \wedge x_C + 1 > 0)$ is proven.

When a theorem is added to the context, RODIN will automatically try to prove it. If it fails, the user has to prove it using the interactive provers. When we fail to discharge a proof obligation, we use the model checker PROB [8] to find a possible counter-example for it. The counter-example gives the values of the different variables that violate one or several invariants. To fix this counter-example, two cases are distinguished:

1. The counter-example denotes a reachable state: it means that the invariant of a state is false and it should be corrected.
2. The counter-example is not a reachable state: in that case, the invariant that is violated is of the form P1 $\Rightarrow$ P2. This means that P1 is too weak; that is, P1 denotes an unreachable state. To fix that, we have to strengthen some state invariants to rule out this counter-example.

In Fig.1, the generated PO for the transition e3 is not provable. The counter-example found by PROB asserts that at the state $S3$, the values of the variables are as follows: $x_A = 2; x_B = 2; x_E = 1; x_F = 1$. Then, during the transition, the substitution $x_A := x_A - x_E$ is done, setting x_A to 1, and thus, $S2.I \equiv (x_F = 0 \vee x_A > x_E)$ is not satisfied. In fact, the state $S3$ is not reachable for $x_A \leq 2x_E$. That is because $S2 \equiv x_A \geq x_E \wedge x_B > x_E$ and the transition e2 does the substitution $x_A := x_A + x_B$ which leads to $x_A > 2x_E$ in state $S3$. Thus, we derive a new invariant for $S3$ that rules out the counter-example:

$$x_F > 0 \wedge x_A > 2x_E$$

It must be kept in mind that modifying an invariant modifies the generated proof obligations. The Rodin archive of the running example can be found in [3]. Proof obligations for the final version (corrected with the above new invariant) are automatically proved by Rodin.

4 Conclusion

In this paper, we have presented a systematic formal approach to verify the satisfaction of local invariants of ASTDs diagrams. Roughly speaking, an ASTDs is a set of hierarchical states (simple or complex) related by process algebra operators and transitions. Local invariants can be associated to these states. We generate proof obligations to ensure that each reachable state satisfies its invariant. To this aim, our approach consists in recursively traversing the hierarchical states and analysing state initialization and transition actions to generate appropriate proof obligations. The generated proof obligations are defined as theorems in Event-B contexts and are discharged using the Rodin platform, and debugged using ProB by using it as a model checker of first-order formulas. To show the feasibility of our approach, we have applied it on several examples which are available in [3].

We are currently working on the implementation of a tool that automatically generates the proof obligations from the ASTD specification. We are also working on the proof obligations of the remaining ASTD operators (flow, choice, synchronization and their quantified versions). Shared variables within synchronized ASTDs represent a challenge for defining proof obligations, because potential interferences between different ASTDs must be taken into account. Future work also includes considering the timed extension of ASTDs as defined on basic ASTD operators in [2]. Finally, it would be important to formally prove the correctness of our proof obligations. We intend to use the approach proposed in [15], where the theory plugin of Rodin is used to build a meta-model of a specification language (Event-B). We could follow a similar approach and define the semantics of ASTDs in a Rodin Theory, and then show that our proof obligations are sufficient to show that the invariants are preserved over the traces of an ASTD. This is a quite challenging task, since the semantics of ASTDs is more complex than the one illustrated in [15].

References

1. de Azevedo Oliveira, D., Frappier, M.: Modelling an automotive software system with TASTD. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) Rigorous State-Based Methods (ABZ2023). LNCS, vol. 14010, pp. 124–141. Springer (2023)

2. de Azevedo Oliveira, D., Frappier, M.: TASTD: A real-time extension for ASTD. In: Glässer, U., Campos, J.C., Méry, D., Palanque, P.A. (eds.) Rigorous State-Based Methods (ABZ2023). LNCS, vol. 14010, pp. 142–159. Springer (2023)
3. Cartellier, Q.: `https://gitlab.com/QCartellier/icfem2023-poastd/-/tree/main/` (2023)
4. El Jabri, C., Frappier, M., Ecarot, T., Tardif, P.M.: Development of monitoring systems for anomaly detection using ASTD specifications. In: Aït-Ameur, Y., Crăciun, F. (eds.) TASE. LNCS, vol. 13299, pp. 274–289. Springer (2022)
5. Fayolle, T.: Combinaison de méthodes formelles pour la spécification de systèmes industriels. Theses, Université Paris-Est ; Université de Sherbrooke (Québec, Canada) (Jun 2017)
6. Frappier, M., Gervais, F., Laleau, R., Fraikin, B., St-Denis, R.: Extending Statecharts with process algebra operators. ISSE **4**(3), 285–292 (2008)
7. Khan, A.H., Rauf, I., Porres, I.: Consistency of UML class and Statechart diagrams with state invariants. In: MODELSWARD. pp. 14–24. SciTePress (2013)
8. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B Method. JSTTT **10**(2), 185–203 (2008)
9. Mammar, A., Frappier, M.: Modeling of a speed control system using Event-B. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods. LNCS, vol. 12071, pp. 367–381. Springer (2020)
10. Mammar, A., Frappier, M., Laleau, R.: An Event-B model of an automotive adaptive exterior light system. In: Raschke, A., Méry, D., Houdek, F. (eds.) Rigorous State-Based Methods. LNCS, vol. 12071, pp. 351–366. Springer (2020)
11. Milhau, J., Frappier, M., Gervais, F., Laleau, R.: Systematic translation rules from ASTD to Event-B. In: Méry, D., Merz, S. (eds.) IFM. LNCS, vol. 6396, pp. 245–259. Springer (2010)
12. Nganyewou Tidjon, L., Frappier, M., Leuschel, M., Mammar, A.: Extended algebraic state-transition diagrams. In: 2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 146–155. IEEE Computer Society (2018)
13. Porres, I., Rauf, I.: Generating class contracts from UML protocol statemachines. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation. MoDeVVa '09, ACM, New York, USA (2009)
14. Porres, I., Rauf, I.: From nondeterministic uml protocol statemachines to class contracts. In: 2010 Third International Conference on Software Testing, Verification and Validation. pp. 107–116 (2010)
15. Riviere, P., Singh, N.K., Ameur, Y.A., Dupont, G.: Formalising liveness properties in event-b with the reflexive EB4EB framework. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13903, pp. 312–331. Springer (2023). `https://doi.org/10.1007/978-3-031-33170-1\_19`, `https://doi.org/10.1007/978-3-031-33170-1_19`
16. Said, M.Y., Butler, M.J., Snook, C.F.: A method of refinement in UML-B. Softw. Syst. Model. **14**(4), 1557–1580 (2015)
17. Sekerinski, E.: Verifying Statecharts with state invariants. In: 13th Int. Conf. on Engineering of Complex Computer Systems (ICECCS). pp. 7–14. IEEE Computer Society (2008)
18. Tidjon, L.N., Frappier, M., Mammar, A.: Intrusion detection using ASTDs. In: Barolli, L., Amato, F., Moscato, F., Enokido, T., Takizawa, M. (eds.) AINA. Advances in Intelligent Systems and Computing, vol. 1151, pp. 1397–1411. Springer (2020)