Formal verification with HiLLS-specified models: A further step in multi-analysis modeling of complex systems

Kehinde G Samuel, Oumar Maiga, Mamadou Kaba Traoré

Formal Verification with HiLLS-Specified Models: A Further Step in Multi-Analysis Modeling of Complex Systems

Kehinde G. Samuel [(1)] keliake@aust.edu.ng

Oumar MAIGA [(2)] oumary.maiga@usttb.edu.ml

Mamadou K. Traoré [(3)] mamadou-kaba.traore@u-bordeaux.fr

(1) African University of Science & Technology, Abuja, Nigeria
(2) Université des Sciences des Techniques et des Technologies, Bamako, Mali
(3) University of Bordeaux, IMS CNRS UMR 5218, France

ABSTRACT

The multi-analysis modeling of a complex system is the act of building a family of models which allows to cover a large spectrum of analysis methods (such as simulation, formal methods, enactment…) that can be performed to derive various properties of this system. The High-Level Language for Systems Specification (HiLLS) has recently been introduced as a graphical language for discrete event simulation, with potential for other types of analysis, like enactment for rapid system prototyping. HiLLS defines an automata language that also opens the way to formal verification. This paper provides the building blocks for such a feature. That way, a unique model can be used not only to perform both simulation and enactment experiments but also to allow the logical analysis of properties without running any experiment. Therefore, it saves from the effort of building three different analysis-specific models and the need to align them semantically.

Keywords: Discrete Event System Specification (DEVS), High Level Language for Systems Specification (HiLLS), Multi-analysis modeling, Discrete Event Simulation, Model Checking.

1. INTRODUCTION

The multi-analysis modeling of a complex system is the act of building a family of models which allows to cover a large spectrum of analysis methods that can be performed to derive various properties (structural, temporal, functional…) of this system. The core underlying principle is that the efficient design and development of a complex system requires an iterative process of modeling, performance evaluation, logical analysis for requirement matching, and enactment for run-time testing [1]. Therefore, models are built to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout the entire life cycle [2]. Depending on the questions to be answered about the system under study, models employed in the iteration loops to mine the desired knowledge of the system are often built for a given analysis methodology. Analysis methodologies promote reasoning about systems from somewhat divergent viewpoints. A viewpoint can be defined, in the general context of software and systems engineering, as a description of appropriate machinery consisting of domain, languages, specifications, and methods to capture and process one or more related engineering or technical concerns about a system and the information associated with such concerns [3-4]. However often, analysis methodologies are necessarily required in combinations for sound system designs.

Computer simulation (also known as M&S, for Modeling & Simulation) is widely acknowledged as one of the major methodologies for complex systems analysis [5]. The Discrete Event System Specification (DEVS) is recognized as a universal M&S formalism [6-8] which provides a computational basis for a general dynamic systems theory. The High-Level Language for Systems Specification (HiLLS) has recently been introduced as a graphical concrete syntax for DEVS [9-11] with potential for other types of analysis, like enactment for rapid system prototyping [12]. HiLLS defines an automata language that also opens the

way to formal verification. This paper establishes a framework for such a feature. That way, a unique formalism can be used not only to perform both simulation and enactment experiments but also to allow the logical analysis of properties without running any experiment. Therefore, it saves from the effort of building multiple analysis-specific models and the need to align them semantically.

Section 2 discusses related works. Since HiLLS is a DEVS-based graphical language, an overview of related works to formal verification with DEVS-based simulation models is given, as well as the difference between them and the work proposed here. Section 3 presents the HiLLS automata, the diagram built by the modeler to graphically describe the system under study in place of the set-theoretic structure-oriented DEVS specification. Section 4 introduces our formal verification framework, where we propose a generic approach to the formal analysis of system properties, and we adopt a pragmatic approach by using an existing software tool that can perform such an analysis. Section 5 illustrates the application of our approach. We conclude the paper with Section 6, where we summarize the work done and draw perspectives for future work.

2. RELATED WORKS

It is widely acknowledged that combining formal methods and simulation can provide a powerful mean to the formal analysis of dynamic systems properties [13-14]. In that context, a pioneering work has proposed linking Timed Automata (TA) and DEVS [15], to extend the design methodology for control systems, using DEVS to specify the low-level behavior of the control being designed, and TA to specify the high-level properties (i.e., the requirements) that the control should meet. The authors used the concept of bi-simulation between DEVS models and TA in order to verify that the control meets the specification of the concept. Using this approach, it is possible to augment the design quality, increasing at the same time the understanding of a system by counting on validation and verification techniques consisting of not only simulation but as well model checking. However, the bi-simulation relations are not generally established, but usually built ad-hoc between specific DEVS and TA models. Such a lack of genericity remains a key challenge. Moreover, the automation of the formal verification process is restricted by the unbounded state space of the models obtained that way.

Most of the following related works align on the principle of a combined DEVS/TA methodology, and the use of the UPPAAL model checker. However, they don't have exactly the same limitations even if they share some common issues.

- Dacharry and Giambiasi [16] introduced Time Constrained DEVS (TC-DEVS), a class of DEVS that expands the DEVS atomic model definition with the introduction of multiple clocks incremented independently of other clocks. While classic DEVS atomic models can be seen as having only one clock that keeps track of elapsed time in a state and is reset on each transition, TC-DEVS models add clock constraints to function as guards on external and internal transitions. However, TC-DEVS also allows state invariants to be expressed as clock differences, which UPPAAL does not allow. Therefore, the transformation from TC-DEVS to UPPAAL TA is solely theoretic and no details is given to explain how state invariants with clock differences are supported by the UPPAAL model.
- Furfaro and Nigro [17] used mathematical transformations called system morphisms, to syntactically map a system expressed in Real-Time DEVS (RT-DEVS), a DEVS extension that is suitable to model real-time systems [18], into its TA counterpart so as to ensure consistency between the results of verification of emergent foreseen behaviors using computational analytical techniques and the discovery of emergent unforeseen behaviors through dynamic simulations. This work has been extended [19] to implementation aspects, where the authors transformed an RT-DEVS-modelled train-gate-controller system into multiple UPPAAL-implemented TA

components that synchronize weakly with each other. An algorithm was introduced to build a timed reachability tree to be used for safety analysis.

- Saadawi and Wainer [20] introduced a new extension to the DEVS formalism, called Rational Time-Advance DEVS (RTA- DEVS). The authors introduced some methodological procedures to systematically create TA models that are behaviorally equivalent to the RTA-DEVS models. The transformation allows them to use the UPPAAL model checking tool to verify RTA-DEVS models. They compared the expressiveness of RTA-DEVS with TA and concluded that the general TA with diagonal constraints is more concise than RTA-DEVS.
- Madlener et al. [21] introduced the Reconfigurable Discrete Event Specified System (RecDEVS), a formalism suitable to model reconfigurable embedded systems. The authors proposed an algorithm to automatically transform a RecDEVS model into an equivalent representation for UPPAAL. Certain limitations of the proposed transformation, which are due to UPPAAL constraints are bypassed by making strong assumptions that the authors considered as always holding in most of the real-world practical applications.

A more recent direction to combining DEVS and formal verification is proposed in [22]. This approach diverges from the ones previously presented in that it uses a pivotal approach instead of a linear transformation from a sub-set of DEVS to TA. The pivotal formalism used is FD-DEVS [23], a subclass of DEVS models having finite sets of states, inputs, and outputs. At one hand, the pivotal FD-DEVS model is mapped onto a non-deterministic automaton that is subject to model checking using SPIN/PROMELA. At the other hand, the FD-DEVS model is elaborated into a full-fledged DEVS models that can be simulated to obtain the behavior of interest and possibly to discover unexpected or emergent behaviors via simulation.

All the related works mentioned above differ from our work in that they do not tick at least one of the following options, while our framework ticks them all:

(a) *Modularity*. The usual DEVS-TA transformation approaches mainly focus on atomic DEVS models and does not exploit the hierarchical nature of DEVS modeling. Instead, they implicitly rely on the closure under composition property of DEVS, which establishes that any coupled DEVS model has a corresponding atomic model. Such a property guarantees the theoretical applicability of the transformation approach, not the operational one.

(b) *Generality*. A severe restriction due to the UPPAAL model checker is the lack of support to the use of double values for time (including no representation of positive infinity). Related works barely mention this limitation and often present study cases that use only integer values for time, which are also limited to finite values. Such a common issue sets a restriction on the category of models that can be considered, limiting them to a sub-set of DEVS.

(c) *Extensibility*. A common limitation in related works is that they do a syntactic transformation from DEVS (or a sub-set of DEVS) to TA (or another verification formalism). The inherent limitation to such an approach is the fact that DEVS has a higher expressive power than the verification formalism, therefore a metamodel-based transformation is not suitable. Rather, the semantic mapping of DEVS into the verification formalism has the potential to provide a full-fledged interpretation of DEVS from the viewpoint of the verification formalism. That way, the DEVS model is taken as the system to be verified, which implies that not only the syntax of the model has to be considered, but also the operational semantics behind. Consequently, DEVS and any of its extensions whose semantics is given in DEVS (like dynamic structure-related extensions), can be interpreted without restriction in the verification formalism.

From a more general perspective, this work differs from works related to combining multiple analysis methods in that those related works usually realize pair wise integrations of their corresponding supported methodologies (under usual circumstances, transformation rules are defined such that one model expressed in the source formalism is given as input and a model specified in the target formalism is obtained as output). Examples of such combined use of simulation and formal analysis include [24-27]. Similarly, several

proposals have been made to bridge the gap between enactment and formal methods [28-31], while some efforts to achieve pair wise integration of simulation and enactment are reported in [32-35]. Contrariwise, the framework adopted here has the potential to integrate more than only two analysis methodologies (including but not limited to simulation, enactment, and formal verification) by providing multiple semantics to the same pivotal concrete syntax, one in each of the target methodologies.

3. FROM DEVS TO HILLS AUTOMATA

The DEVS formalism proposes two mathematical structures to capture the structural and behavioral aspects of a dynamic system: the atomic model structure and the coupled model structure. The latter can be turned into its equivalent atomic model, and such a property of DEVS modeling is called closure under coupling. When moved from abstract sets to encoded sets, DEVS models can be written as finite state automata, which we call HiLLS automata. This section shows how atomic, as well as coupled DEVS models, can be expressed as HiLLS automata.

The atomic DEVS model is a structure $\langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta \rangle$ where:

- X, Y, and S are respectively the input set, output set, and state set (at any time, the system modeled is in one of the states of S)
- $ta : S \rightarrow \Re_0^{+\infty}$ is the time advance function (i.e., it gives the lifespan of each state), with $\Re_0^{+\infty}$ designating the set of non-negative real numbers, including $+\infty$
- $\delta_{int} : S \rightarrow S$ is the internal transition function (i.e., it is triggered only when the elapsed time in the system's current state s_{curr} has reached $ta(s_{curr})$ without the system being disturbed by any receipt of input event)
- $\lambda : S \rightarrow Y$ is the output function (i.e., it computes the output of the system, each time an internal transition is occurring)
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function (i.e., it is triggered only when the system receives an input event, while the elapsed time in the system's current state s_{curr} has not reached $ta(s_{curr})$), and $Q = \{(s,e) / s \in S, 0 \leq e < ta(s)\}$ is called the total state
- $\delta_{conf} : S \times X \rightarrow S$ is the confluent transition function (i.e., it is triggered only when the system receives an input event at exactly the time that the elapsed time in the system's current state s_{curr} has reached $ta(s_{curr})$)

An automata-like representation of such a structure (which we call here DEVS automata) is shown in Figure 1, where DEVS states are nodes of the automaton, and DEVS state transitions are edges linking the corresponding nodes of the automaton.

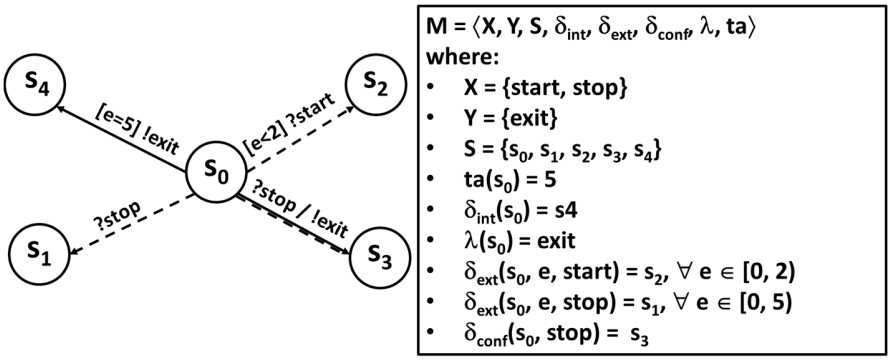


Figure 1. DEVS automaton and its corresponding atomic DEVS specification

Different types of edge are considered. The ones that correspond to internal transitions in DEVS are represented by solid arrows, external transitions are represented by dotted-line arrows, and confluent transitions are represented by a composition of solid and dotted-line arrows. Moreover, the receipt of message in the DEVS model is represented by a signal that triggers the corresponding dotted-line arrow in the automaton. A specification such as ?x means an input event x has been received. Similarly, an output in the DEVS model is represented by an action done during the corresponding solid arrow in the automaton. A specification such as !y means that an output event y has been sent. A transition can be guarded by a condition (mostly a constraint over the elapsed time) enclosed in square brackets.

Definition 1: A DEVS automaton is a tuple DEVSA = (X, Y, S, s_0, T_{int}, T_{ext}, T_{conf}, W), where

- S is the set of states,
- $s_0 \in S$ is the initial state,
- $T_{int} \subseteq S \times Y \times S$ is the internal transition relation,
- $T_{ext} \subseteq S \times X \times \mathbb{R} \times S$ is the external transition relation,
- $T_{conf} \subseteq S \times X \times S$ is the confluent transition relation,
- $W \subseteq S \times \mathbb{R}$ is the temporal weight relation.

Definition 2: given DEVSA = (X, Y, S, s_0, T_{int}, T_{ext}, T_{conf}, W), we define the following:

- $\forall s \in S, \Gamma(s) = \{s' \in S / \exists y \in Y, (s, y, s') \in T_{int}\}$
- $\forall s \in S, \Gamma^{-1}(s) = \{s' \in S / \exists y \in Y, (s', y, s) \in T_{int}\}$
- $\forall s \in S, \forall n \in \mathbb{N}, n > 1, \Gamma^{-n}(s) = \{s' \in S / \exists y \in Y, \exists s'' \in \Gamma^{-n+1}(s), (s', y, s'') \in T_{int}\}$
- $\forall s \in S, \Lambda^{-1}(s) = \Gamma^{-1}(s)$
- $\forall s \in S, \forall n \in \mathbb{N}, n > 1, \Lambda^{-n}(s) = \Gamma^{-n}(s) \cup \Lambda^{-n+1}(s)$

Lemma 1: given DEVSA = (X, Y, S, s_0, T_{int}, T_{ext}, T_{conf}, W), $\forall s \in S$, $card(\Gamma(s)) \leq 1$
Proof: $\Gamma(s)$ is the set of states immediately reachable from state s by an internal transition. In a properly defined DEVS automaton, such a set is either empty (if $ta(s) = +\infty$) or limited to one element (i.e., $\delta_{int}(s)$).

$\Gamma^{-1}(s)$ is the set of states from where s is immediately reachable by an internal transition. $\Gamma^{-n}(s)$ is the set of states from where s is reachable after n consecutive internal transitions. $\Lambda^{-n}(s)$ is the set of states from where s is reachable after 1 or n consecutive internal transitions. That way, $\Lambda^{-\infty}(s)$ is the set of all states from where s is reachable after a finite or infinite sequence of internal transitions. We will use this set later in the paper, which we call the graph of s ancestors.

3.1. Encoding states

One major issue with such a graph is that it is manageable only when the DEVS model has a finite state set. Otherwise, the corresponding automaton will not be graphically representable. The idea with the HiLLS automata is that a DEVS model (whether with finite or infinite state set) can be represented by a graphically representable automaton, which we can treat as a finite automaton specifically useful for the purpose of formal analysis, without any loss of behavioral property/information and while still being adequate for simulation and enactment.

Let's first establish that a DEVS state set S (whether finite or infinite) can always be captured by a finite set of variables. This is obvious since we can always define a single variable which domain is S. However, in many cases, S will be characterized by more than one variable. Let's assume we have defined a finite set of variables, say V = {v1, v2, v3, …, vn} then $S \subseteq \prod_1^n dom(vi)$. Without any loss of generality, we can assume we've defined V such that $S = \prod_1^n dom(vi)$. Therefore we can define a bijective function enc that

characterizes each state of S by a vector in $\prod_1^n vi$. We call dec the inverse function of enc (enc and dec are respectively for encoding and decoding). We have:

$\forall s \in S, \forall \alpha i \in dom(vi)$ pour $i \in \{1, \ldots, n\}$ $\quad enc(s) = (\alpha 1, \ldots, \alpha n) \Leftrightarrow dec(\alpha 1, \ldots, \alpha n) = s$

A new automaton is obtained by replacing in the previous one each $s \in S$ such that $enc(s) = (\alpha 1, \ldots, \alpha n)$ by a node C (called configuration), which is defined by the predicate: prop(C): $(v1, \ldots, vn) = (\alpha 1, \ldots, \alpha n)$. By extension, we write C = enc(s). Moreover, any temporal guard [e = k] defined on a solid arrow of the DEVS graph is removed and k defines the lifetime of the corresponding source configuration. We also make explicit any computation (i.e., activity) performed when the automaton is in a given configuration, given that these computations are done during the lifetime of the corresponding configuration. Figure 2 shows the encoded automaton that corresponds to the DEVS automaton of Figure 1.

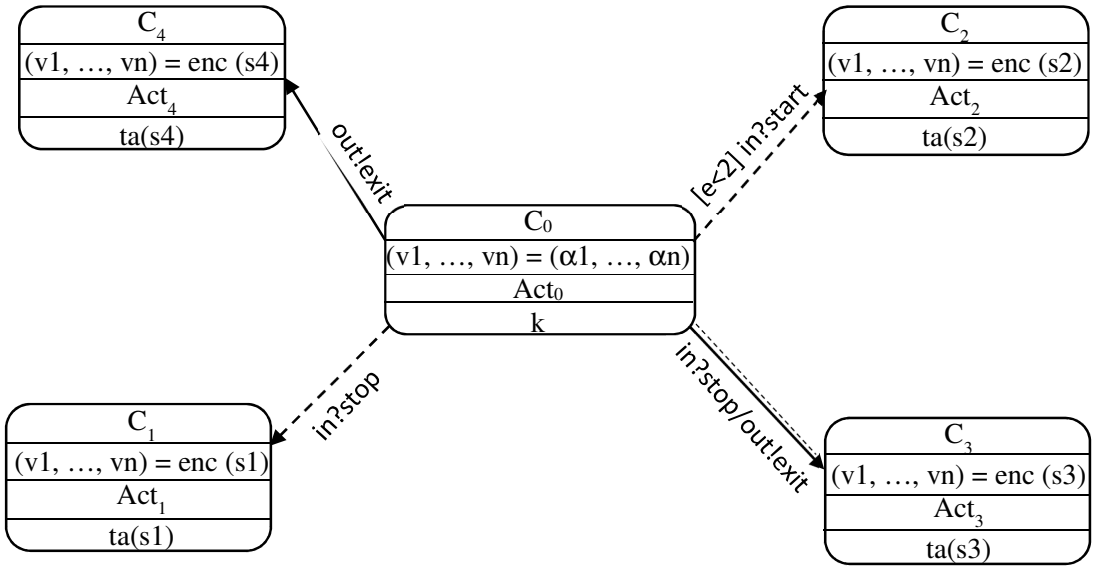


Figure 2. Encoded automaton of the previous DEVS automaton

At any time, a system represented by an encoded automaton is in a given configuration, that is satisfying some property (and possibly doing some activity), and for a given lifetime. This captures both the notion of state graph (when focus is on properties) and event graph (when focus is on activities). State graph serves for simulation (activities are hardly necessary in that case, therefore they are ignored or not mentioned in the automaton), while event graph serves for enactment (the indication of lifetime may not be necessary in that case, as the real lifetime of the configuration would be the execution time of the activity). Configuration transitions occur due to the elapse of the current configuration's lifetime (internal transition) or the receipt of an input (external transition). As for state set, the input and output sets are also encoded by variables (called ports), which are indicated as prefixes of input and output events (U?x means the event x has been received on the input port U, and V!y means the event y has been sent on the output port V). Moreover, we allow computations to be possible also during transitions, and this is particularly useful in the case of infinite-state set.

When the DEVS automaton has a finite state set, the encoded automaton is just an annotation of the DEVS automaton, in the sense that it provides more details without modifying the semantics of interest (i.e., the trajectories generated by simulation). When the DEVS automaton has an infinite number of nodes, we fold it to obtain a finite number of nodes, while ensuring that there is no loss of behavioral properties. In other words, we show that any DEVS automaton with infinite state set has a corresponding HiLLS automaton with finite configuration set.

Notice that the HiLLS concrete syntax has some customized configuration representations, which we do not present extensively here. For example, a configuration with ta = 0 (also called a transient configuration) is represented as a circle, while a configuration with ta=+∞ (also called passive configuration) is represented by a rounded rectangle with a vertical double line at its right-hand side (like the Queue configuration of Figure 3). A configuration with 0 < ta < +∞ (also called active configuration) is represented by the regular rounded rectangle displayed so far (as the ones of Figure 2). The interested reader can refer to [9-11], [36] for more on the HiLLS concrete syntax.

3.2. Folding multiple states into a single configuration

The notion of folding multiple states into a single configuration can be understood through the example of a first-in-first-out queuing system with infinite queue length. The DEVS automaton of such a system is shown in Figure 3(a), while Figure 3(b) shows its corresponding HiLLS representation with all states except "empty queue" state folded into a single configuration.

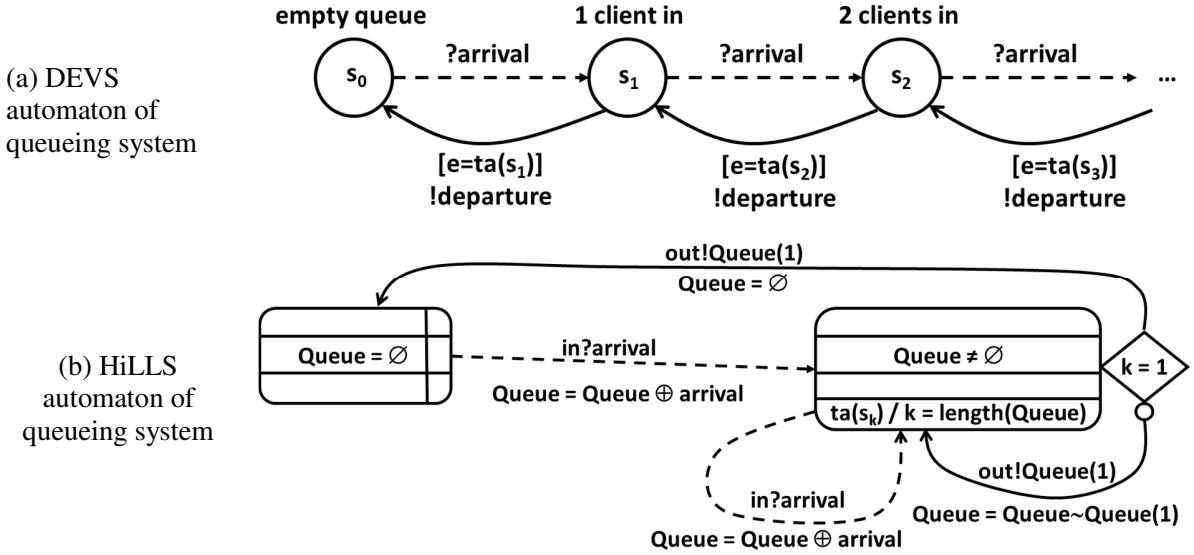


Figure 3. Example of states folded into a configuration

While the DEVS automaton has an infinite set space, the corresponding HiLLS automaton has a finite configuration space. The DEVS empty queue state is mapped onto the HiLLS left-hand side configuration, while all the other DEVS states are encoded and folded into the HiLLS right-hand side configuration. When multiple states are folded, the code of the resulting configuration (e.g., Queue ≠ ∅) is the disjunction of the codes of the states (e.g., Queue = a given set for each state). Therefore, each time the HiLLS automaton must enter a configuration, some updates are done (e.g., Queue = Queue ⊕ arrival, where ⊕ is the operation that appends an element to Queue, or Queue = Queue ~ Queue(1), where ~ is the operation that removes an element from Queue, and Queue(1) is the first element of Queue) to select among this disjunction of codes the one that must be the current value. Similarly, the life time of a configuration (e.g., $ta(s_k)$ where k is the length of Queue) is a selection among the lifetime of its corresponding folded states (e.g., $ta(s_k)$ for each state s_k). The diamond shape in the HiLLS automaton represents a test with positive (i.e., the result of the test is true) and negative (indicated by the inhibiting circle at one edge of the diamond) branches.

Definition 3: Given DEVSA = (X, Y, S, s_0, T_{int}, T_{ext}, T_{conf}, W), a corresponding (and not unique) HiLLS automaton is a tuple (X, Y, V, enc, C, cT_{int}, cT_{ext}, cT_{conf}, cW), where

- V is a vector of variables, with dom(V) = S
- enc: S $\rightarrow$V is the state encoding function
- C = {c_1, c_2} is the configuration set, with c_1 = {enc(s) / s $\in$ S, ta(s) = +∞}, and c_2 = {enc(s') / s' $\in$ S, $\exists$ s $\in$ S, ta(s) = +∞, s' $\in$ $\Lambda^{-\infty}$(s)}, and the initial configuration is c_k / enc(s_0) $\in$ c_k
- cT_{int} = {(c_2, y, c_1)} is the set containing the only internal configuration transition, with y $\in$ {y_j / $\exists$ (s_i, y_j, s_k) $\in$ T_{int}, enc(s_i)$\in$ c_2, enc(s_k)$\in$ c_1}
- cT_{ext} ={(c_a, x, e, c_b) / a, b $\in$ {1, 2}, $\exists$ (s_j, x, e, s_k) $\in$ T_{ext}, enc(s_j)$\in$ c_a, enc(s_k)$\in$ c_b} is the external configuration transition relation,
- cT_{conf} ={(c_a, x, c_b) / a, b $\in$ {1, 2}, $\exists$ (s_j, x, s_k) $\in$ T_{conf}, enc(s_j)$\in$ c_a, enc(s_k)$\in$ c_b} is the confluent configuration transition relation,
- cW = {(c_2, t), (c_1, +∞)} is the lifetime relation, with t $\in$ {τ $\in$ $\mathbb{R}$ / $\exists$ (s, τ) $\in$ W}

Figure 4 shows the different steps that allow to build, from a DEVS automaton, the corresponding HiLLS automaton as formalized by Definition 3. Figure 4(a): consider a DEVS automaton, where only the internal transition relations are represented. State with infinite time advance (called passive states) are double circled. Three situations can occur: (i) a graph of finite or infinite states belonging to the graph of ancestors of a passive state; (ii) a passive state which graph of ancestor is empty; and (iii) a graph of states with a circuit in the internal transition relations. Figure 4(b): for each passive state, all states in the graph of ancestors are folded into a configuration. This leads to Figure 4(c), where the number of passive states can be finite or infinite, but each of them has a finite number of ancestors. Figure 4(d): all passive states are folded into one configuration, leading to Figure 4(e), where there is only one passive configuration (c_1 in Definition 3). Figure 4(f): all ancestors of the passive configuration are folded into a configuration (c_2 in Definition 3).

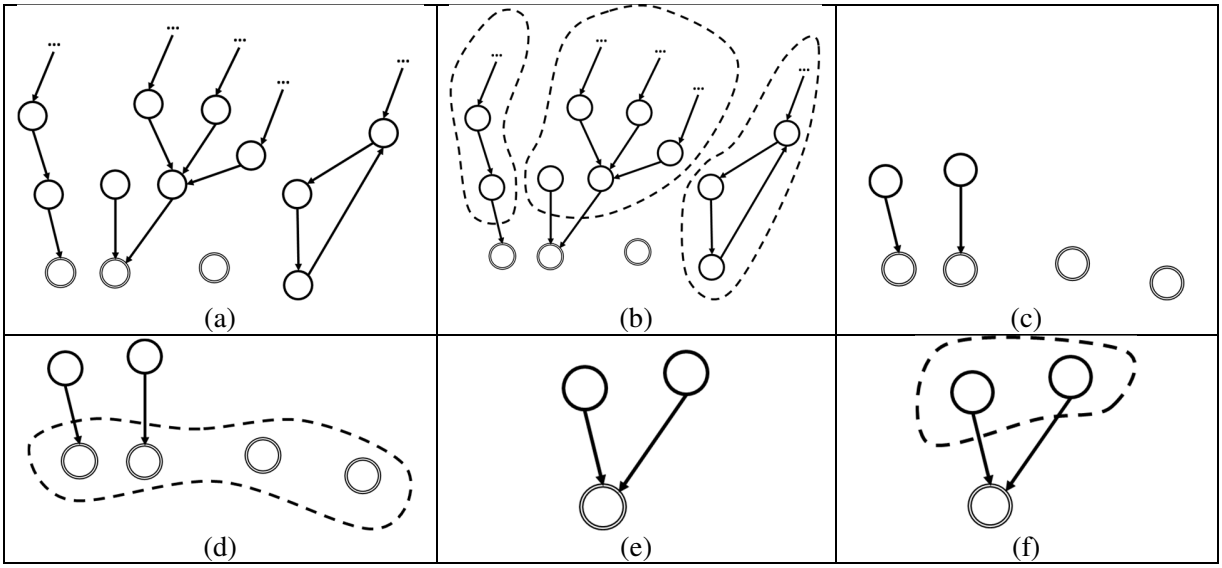


Figure 4. Reduction of an infinite DEVS automaton to a finite HiLLS automaton

3.3. Composed HiLLS automata

HiLLS allows for the composition of HiLLS automata in a way such that the composed model is also expressed as a HiLLS automaton. Moreover, while a traditional composed model will have in HiLLS a

single configuration specifying the coupling information between its sub-components, a dynamic structure composed model will have a configuration transition diagram with more configurations, each of them specifying a given architecture of the composed model, and the transitions specifying the rules for the dynamic change of structure. Figure 5 shows two composed HiLLS models, one describing a static structure composed model (left-hand side), and the other a dynamic structure (right-hand side). The left-hand side model (Permanent Control System) defines a permanent coupling between its two components named *mnt* (a health monitor) and *ctz* (a patient/citizen). This is captured by the single passive configuration appearing in its transition diagram, which establishes that whenever an output is sent on the status port of the patient, then this is treated as in input for the monitor. The dynamic structure model (Steady Control System) defines a system where the monitor component interacts with the patient components only 8 time units over 24 (e.g., let time unit be hour to reflect the fact that the patient is under controls for only some time in a day).

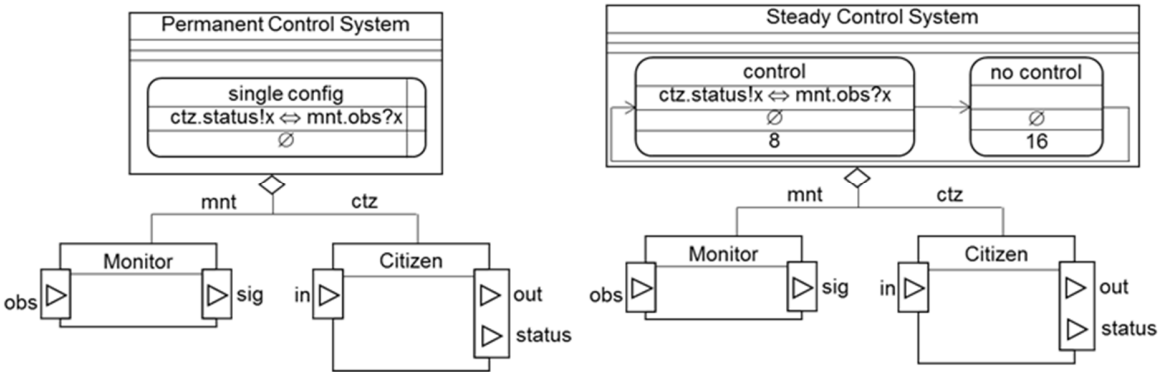


Figure 5. Static structure versus dynamic structure HiLLS models [36]

4. HILLS-BASED VERIFICATION FRAMEWORK

HiLLS can semantically be mapped onto automata-based formal methods. In this paper, we target TA, similarly to most of the related works, mainly because of the available operational UPPAAL tool. However, the approach adopted here can be used with other formal methods. Instead of performing a syntactic transformation from HiLLS to TA (by mapping the metamodels of both, one onto the other), we define in TA the logical semantics of HiLLS that corresponds to its operational semantics (which we know from DEVS). Therefore, we identify a set of patterns in the HiLLS model's dynamics, which we give the semantics in UPPAAL TA. As such the set of patterns covers all possible cases of configuration and configuration transition. Therefore, one can always get the full semantics of any HiLLS-specified model by applying for each of its configurations and transitions the appropriate semantics mapping pattern.

4.1. Active configuration

When a HiLLS model is in a given configuration C, it means one of the following:

(1) The system has entered the configuration, but no input has been received and the lifetime has not elapsed yet. We call such a situation C_{curse} for any given configuration C, and it is characterized by the predicate $P(C_{curse})$: $(e < ta)$ AND $(mail = \varnothing)$, where mail is the bag of input events received by the model at a given time.

(2) The system has reached the end of C's lifetime without any receipt of input. We call such a situation C_{end} and it is characterized by the predicate $P(C_{end})$: $(e = ta)$ AND $(mail = \varnothing)$.

(3) The system has not reached the end of C's lifetime and has received input (one or more, all stored in the mail). We call such a situation $C_{interrupt}$ and it is characterized by the predicate $P(C_{interrupt})$: (e < ta) AND (mail ≠ ∅).
(4) The system has reached the end of C's lifetime and has received input. We call such a situation $C_{dilemma}$ and it is characterized by the predicate $P(C_{dilemma})$: (e = ta) AND (mail ≠ ∅)

The exploratory nature of model checking often set a constraint on time representation as limited to integer values only. To overcome that, we suggest the simulated time be represented by a couple of values (t_{int}, t_{dec}), where t_{int} is the integral part of time value, and t_{dec} is its decimal part. That way, a condition such as e = ta translates into (e_{int}-ta_{int} = 0) AND (e_{int}-ta_{int} = 0), while a condition such as e < ta translates into (e_{int}-ta_{int} < 0) OR ((e_{int}-ta_{int} = 0) AND (e_{int}-ta_{int} < 0)). For UPPAAL implementation purpose, we represent a time couple value par a 2D array, i.e., t_{int} (respectively t_{dec}) is implemented as t[0] (respectively t[1]).

Figure 6 shows the logical semantics of the HiLLS model being in an active configuration C. It has four locations, each of which corresponds one of the cases previously described. The automaton operates with a single clock, which corresponds to the elapsed time e, and which is represented by its integral and its decimal parts. Any transition of the system to configuration C comes to the C_{curse} location. Then, the elapsed time is set to 0.0 (therefore e[0] = 0, and e[1] = 0), and the lifetime of the location is determined by update(ta).

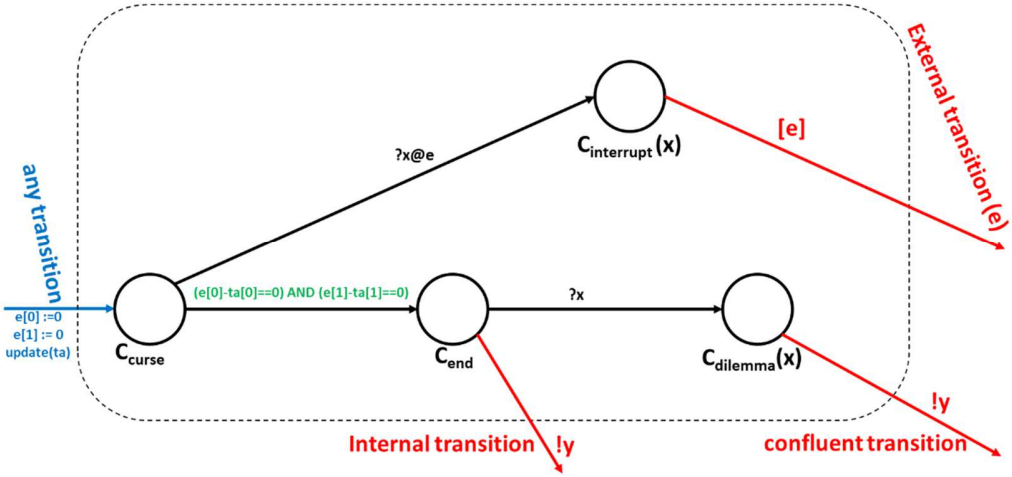


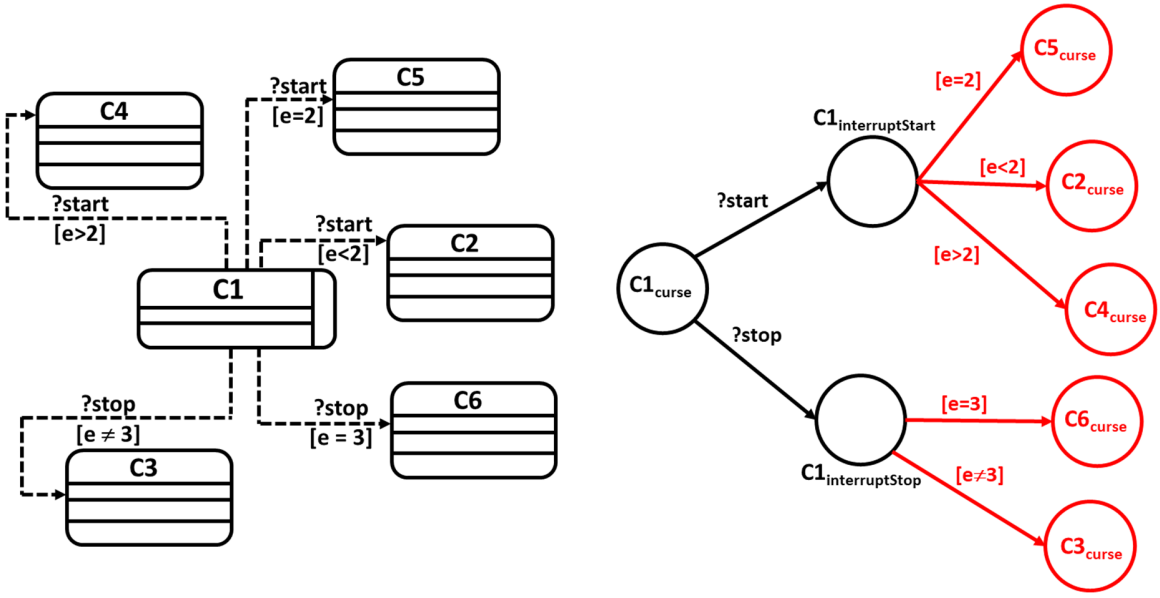
Figure 6. Semantics pattern for HiLLS active configuration

When at C_{curse}, a transition to C_{end} occurs when ((e[0]-ta[0] == 0) AND (e[1]-ta[1] ==0)). But, if an input ?x@e is received (meaning message x is received at the time elapsed e) before such a condition is met, then a transition occurs to $C_{interrupt}(x)$. Since the $C_{interrupt}(x)$ location depends on x, there are as much target locations as possible values for x. In other words, if the C configuration in the HiLLS model has n external transitions triggered by n different messages, then the pattern will generate n different $C_{interrupt}$ locations, each labelled with one of the messages. Figure 7 illustrates such a situation. Figure 7(a) shows a HiLLS configuration (C1) with multiple external transitions. There are two different values for the inputs received: start and stop. Therefore, in Figure (b), $C1_{interruptStart}$ and $C1_{interruptStop}$ are created.

When at C_{end}, if an input ?x is received, then a transition occurs to $C_{dilemma}(x)$. Similarly to the case of $C_{interrupt}(x)$, there are as much $C_{dilemma}(x)$ locations as possible values for x. On contrary (i.e., no input has been received), a transition occurs to G_{curse}, for each G that is a configuration in the HiLLS model such that there is an internal transition from C to G. An ouput y is then sent (notice that y can be void).

When at $C_{interrupt}(x)$, a transition occurs to G_{curse}, for each G that is a configuration in the HiLLS model such that there is an external transition from C to G due to the receipt of x at elapsed time e. If several external transitions exist in the HiLLS model with the same x received but at different values of e, for each of them, the pattern will generate an external transition from location $C_{interrupt}(x)$ to G_{curse}, where G is the target configuration of the corresponding external transition in HiLLS. Figure 7 illustrates such a situation, with both $C1_{interruptStart}$ and $C1_{interruptStop}$. For each of these locations, there are several external transitions, each guarded by the corresponding condition on the elapsed time.

When at $C_{dilemma}(x)$, a transition occurs to G_{curse}, for each G that is a configuration in the HiLLS model such that there is a confluent transition from C to G due to the receipt of x. In case of multiple confluent transitions in the HiLLS model due to different values of input, the same semantics mapping principle as the one previously described with $C_{interrupt}(x)$ applies.



(a) (b)

Figure 7. Example of semantics mapping from HiLLS active configuration to UPPAAL

4.2. Pattern variants

The pattern previously described is a general case for active configuration. However, not all configurations have all the characteristics of such a case. To ease the semantics mapping, we derive variants where some of the characteristics are not present (in the HiLLS model).

Figure 8 presents these variants, which are:

- interruptless active configuration, where there is no external transition from that configuration to another one (see Figure 8a);
- conflictless active configuration, where there is no confluent transition from that configuration to another one (see Figure 8b);
- interruptless and conflictless active configuration, where there is no external transition, nor confluent transition from that configuration to another one (see Figure 8c);
- passive configuration, where there can't be internal transition (nor confluent transition) from that configuration (as the lifetime is positive infinity) to another one (see Figure 8d); and

- interruptless passive configuration, where there is no external transition from that configuration (which already can't have internal/confluent transition departing from it) to another one (see Figure 8e).

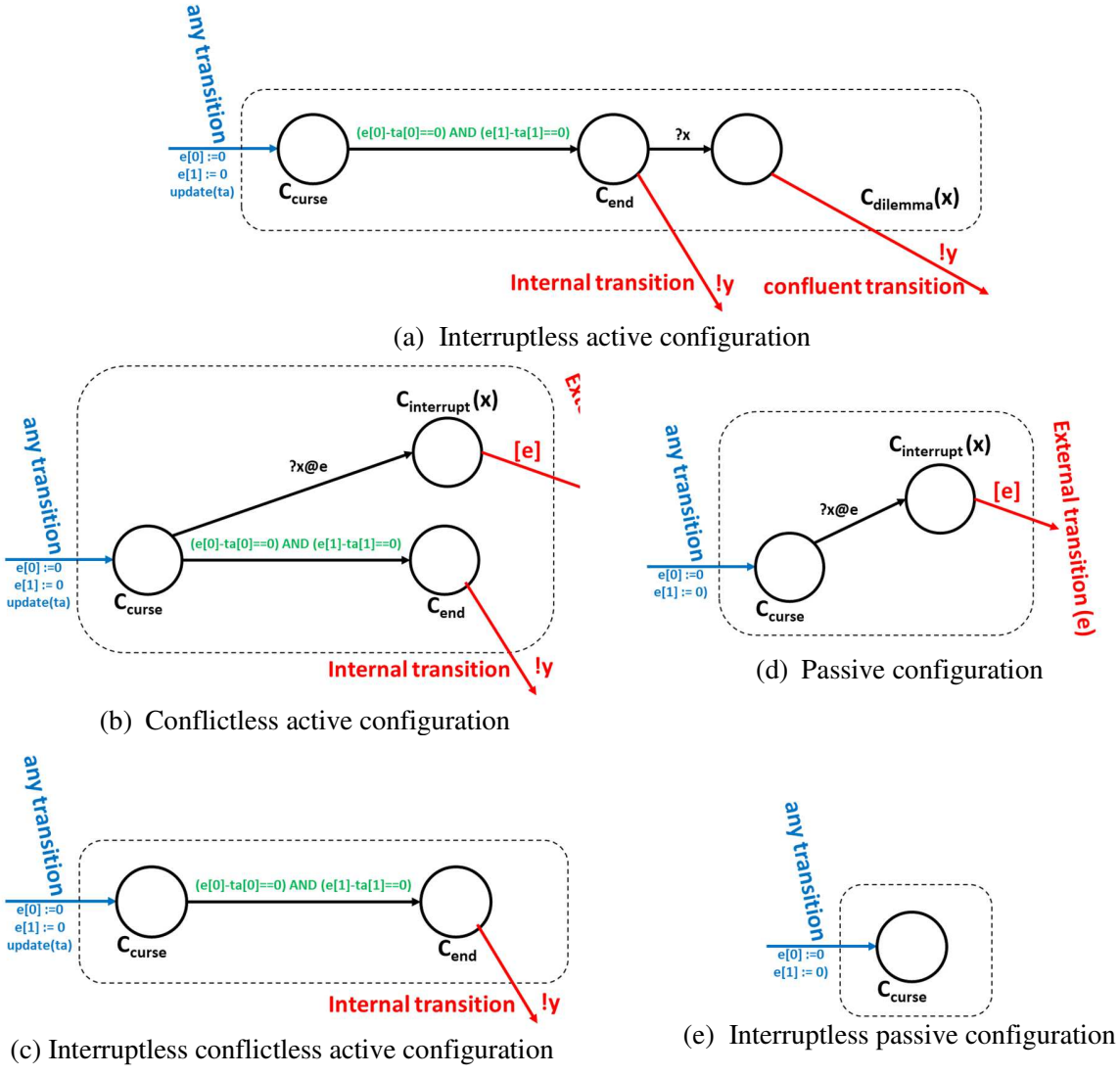


Figure 8. Variants of the semantics pattern for HiLLS configurations

4.2. Patterns for hierarchical composition

Altogether, the six previous patterns (i.e., the initial one and its five variants) completely define the semantics mapping rules of HiLLS into UPPAAL TA in the case the HiLLS model is not composed of other models. To address the hierarchical composition of HiLLS components, we need to translate the composition information carried by the configuration(s) of the composed model into semantical information into UPPAAL. We achieve that by introducing two mechanisms:

(1) We introduce a feedback loop location transition in UPPAAL to semantically translate the DEVS simulation protocol, according to which, when a HiLLS component sends a message, the composed

model is the one to first receive that message, before it forwards it to the appropriate recipients of the message.

(2) We systematically tag in UPPAAL all messages sent and received by each HiLLS component with the name of that component. That way, when a message is sent by a HiLLS component, its embedding HiLLS model identifies the origin of the message, and with the composition information, transform it to be a message tagged with the name of the recipient.

Figure 9 illustrates this principle with an interruptless passive configuration corresponding to a composed static structure HiLLS model (as the one shown at the left-hand side of Figure 5). This configuration translates into a single location in UPPAAL, in conformance with the pattern of Figure 8e. Assume this composed model has a component named *sender* (therefore, all messages sent from or received by any location that corresponds to a configuration of the *sender* component is tagged in UPPAAL with <*sender*>), and a component named *receiver* (therefore, all messages sent from or received by any location that corresponds to a configuration of the *receiver* component is tagged in UPPAAL with <*receiver*>). The location has a feedback loop such that if a message is sent by the sender, it is intercepted by the location, which in turn will emit a message with the same content but tagged with the identity of the appropriate receiver (as given by the composition information defined in the predicate of the corresponding HiLLS configuration). The principle is similar with dynamic structure models, as the feedback loop transition applies to each of the locations in UPPAAL.

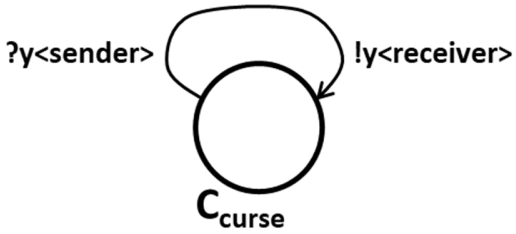


Figure 9. Semantics in UPPAAL of message transmission between two HiLLS components

5. APPLICATION

In this section, our approach is used to model and verify the automated unmanned railway level crossing system depicted by Figure 10. The crossing is guarded by a gate, which is used to close the road in case a train is passing. Two sensors, one located upstream of the crossing, and the other located at the exit of the crossing, are used to detect the arrival and exit of a train. The control system receives remote signals from the sensors, and remotely controls the closing and the opening of the gate.

We assume that:

- It takes 5.8 seconds for an approaching train to be detected by the entrance sensor;
- It takes 8.6 seconds for the train to travel between the upstream sensor and the crossing entry;
- It takes 5.2 seconds to pass the crossing;
- It takes 2.0 seconds to travel between the crossing exit and the downstream sensor;
- The gate takes 2.3 seconds to open or close.

The railway managers have two major concerns about this crossing system:

- Safety concerns, i.e., the gate must never be opened when the train is crossing.
- Suitability concerns, i.e., the gate should not be closed when there is no train crossing/approaching (to not stop car traffic when there is no security reason to do so).

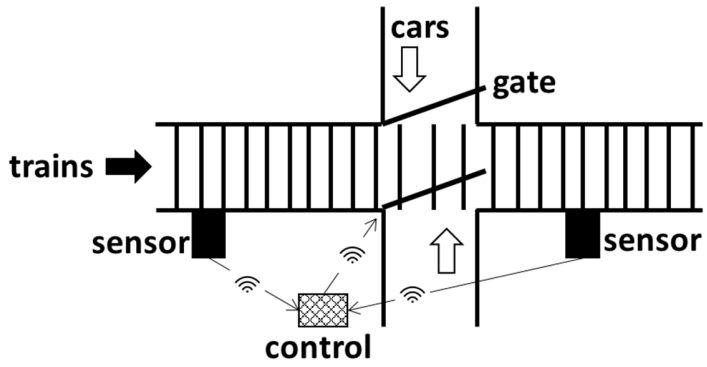


Figure 10. Automated unmanned railway level crossing system

5.1. HILLS-specified system model

The problem is modeled with six models specified in HiLLS, namely: Train, Gate, EntranceSensor, ExitSensor, Controller, and Crossing system. They are presented in Figure 11:

- The Train model (Figure 11a) has five configurations: *Approaching*, *Before_Crossing*, *Crossing*, *After_Crossing,* and *Moving-Away*. The initial configuration is *Approaching* and is finite. As such an internal transition takes place at the expiration of the sojourn time (5.8 seconds), and the transition takes the Train to *Before_Crossing* while outputting *Approach*. The Train stays at the *Before_Crossing* configuration for 8.6 seconds as this is the travel time from being detected to entering the crossing area. Then, the Train takes an internal transition to *Crossing* where it spends 5.2 seconds and then transits to *After_Crossing*. An internal transition from *After_Crossing* will output *Exit* after 2.0 seconds and transit the Train to *Moving_Away* where the Train spend reasonably longer time (ta is generated by a random function) before returning to its initial *Approaching* configuration. Such a loop in the Train's behavior translates the frequent passing of trains in real world, with the assumption that the inter-arrival times of trains to the crossing are distributed according to the random law indicated.
- The Gate model (Figure 11b) has four configurations: *Up*, *Lowering*, *Down* and *Raising*.The initial configuration of the Gate is a passive configuration *Up*. The Gate is open and it remains open infinitely until it receives an input signal *Close*. The signal will cause an external transition to *Lowering*, a finite configuration with sojourn time of 2.3 seconds. From *Lowering*, the Gate takes an internal transition to *Down*, where it stays until it receives an input signal *Open*. The *Open* signal transits the Gate to *Raising*. The Gate stays in this configuration for 2.3 seconds, the time required to open the Gate before transiting internally to *Up*.
- There are two sensor models, namely EntranceSensor and ExitSensor (Figure 11c and Figure 11d). Each of the sensors has two configurations, *Waiting* and *Detecting*. The two sensors function the same way, except that they output different signals: *Entering* from EntranceSensor and *Exiting* from ExitSensor. The initial configuration of the Sensor is *Waiting*. The Sensor keeps waiting in this passive configuration until it detects the signal *Approach*. It will then take an external transition to a transient configuration *Detecting* where it spends no time. Therefore, an internal transition immediately takes place from *Detecting* back to *Waiting* while a message *Entering* (respectively *Exiting*) is sent out for EntranceSensor (respectively ExitSensor).
- The Controller model (Figure 11e) has 3 configurations: *Inactive*, *Closing* and *Opening*. The initial configuration of the model is *Inactive*. The Controller remains in this passive configuration until it receives a signal. Once the signal is received, the Controller takes an external transition to *Closing* (if the signal received is *Approach*), or to *Opening* (if the signal received is *Exit*). No time is spent

at the transient configurations (*Closing* and *Opening* respectively), and an internal transition takes the Controller back to *Inactive* while outputting or sending *Open* and *Close* signal respectively.

- The Crossing system model (Figure 11f) represents the entire system, a composition of the components previously presented. For the sake of readability, we don't show components' ports, since they have been detailed before. The entire system has a unique passive configuration *Network* that depicts the relationships between components. This translates the static nature of the composition. The predicate of the *Network* configuration provides the coupling information between the components. For example, *EntranceSensor.In = Train.out* means the output port *out* of the Train component model is connected to the input port *In* of the Entrance_Sensor component model. The ∧ operator is the AND logical operator.

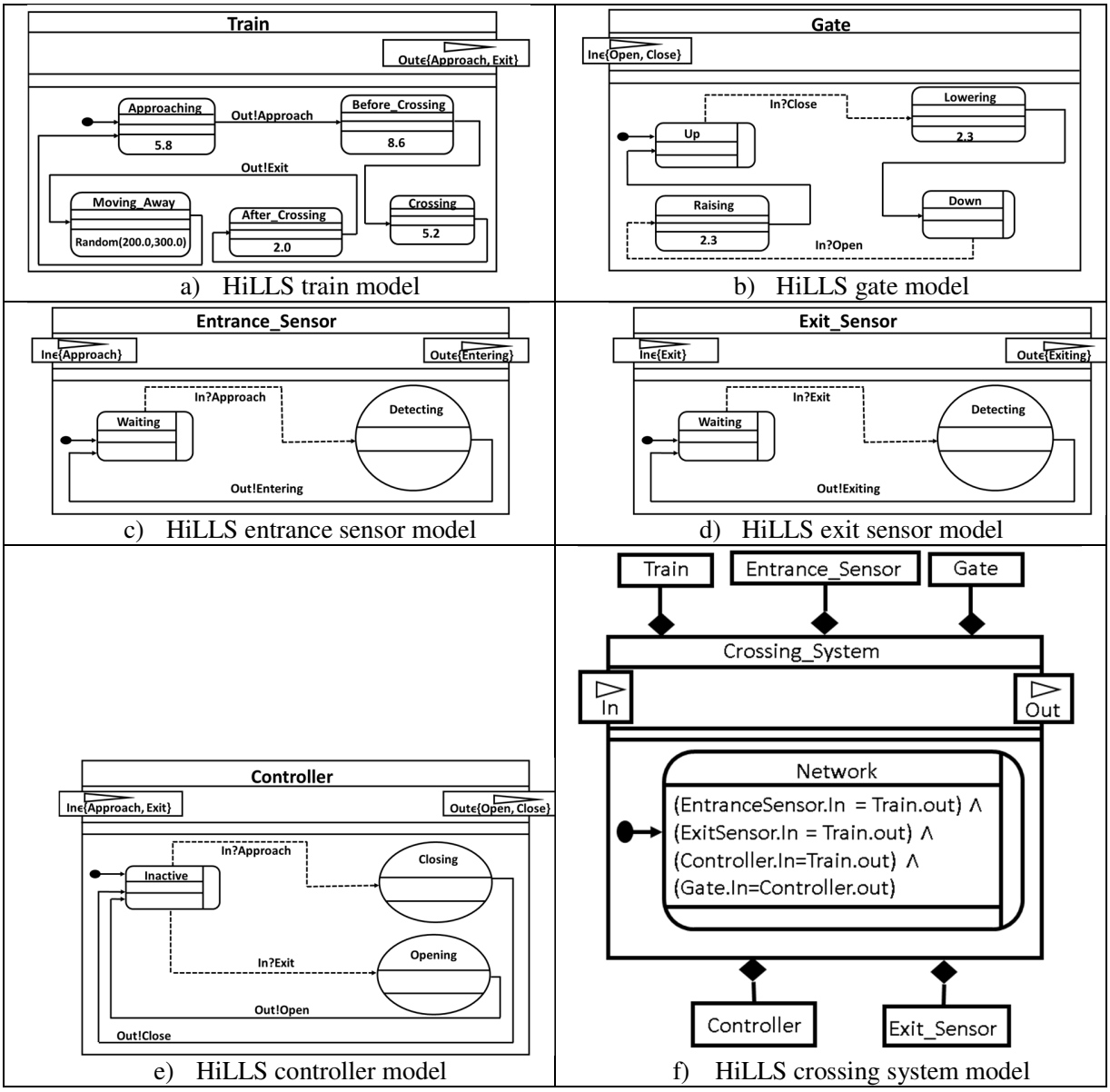
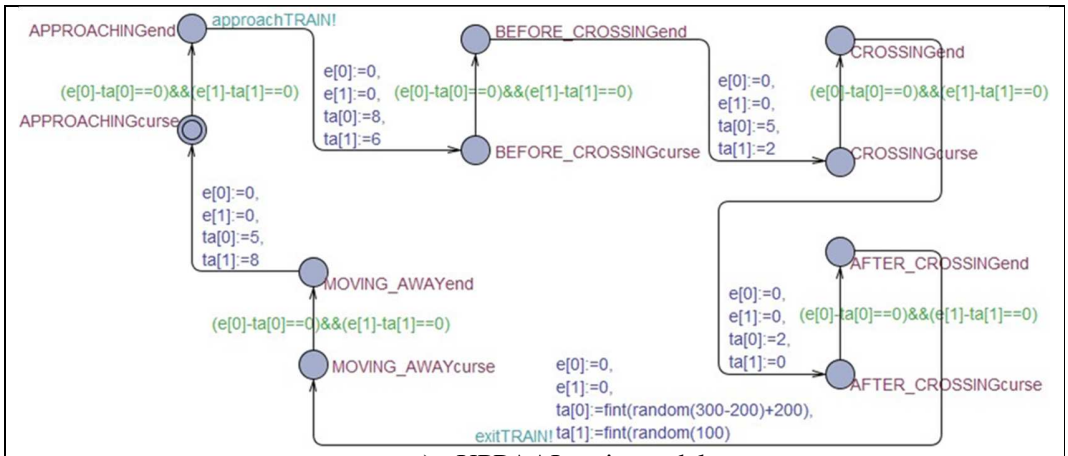


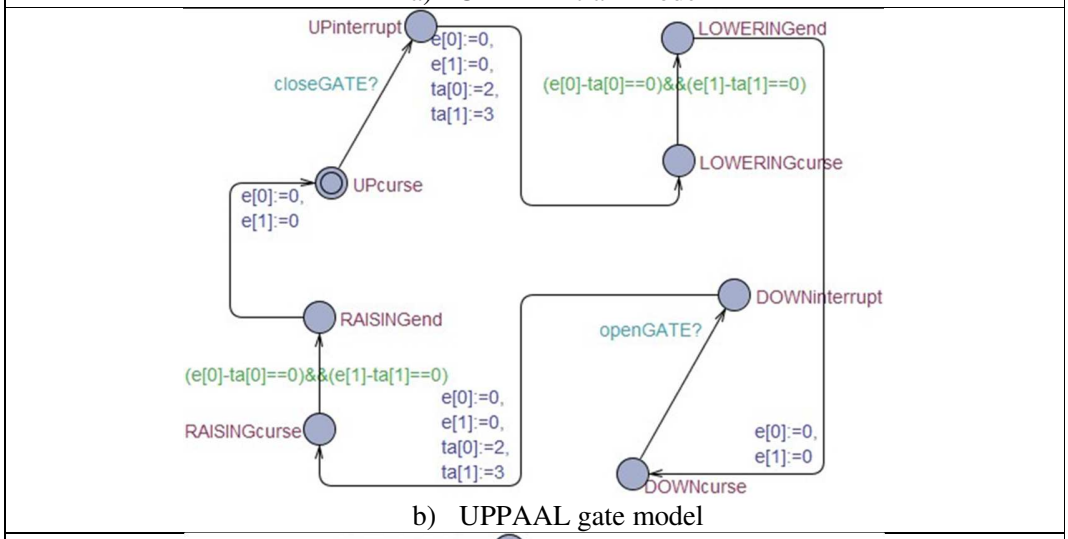
Figure 11. HiLLS models of the Crossing System

5.2. UPPAAL TA models

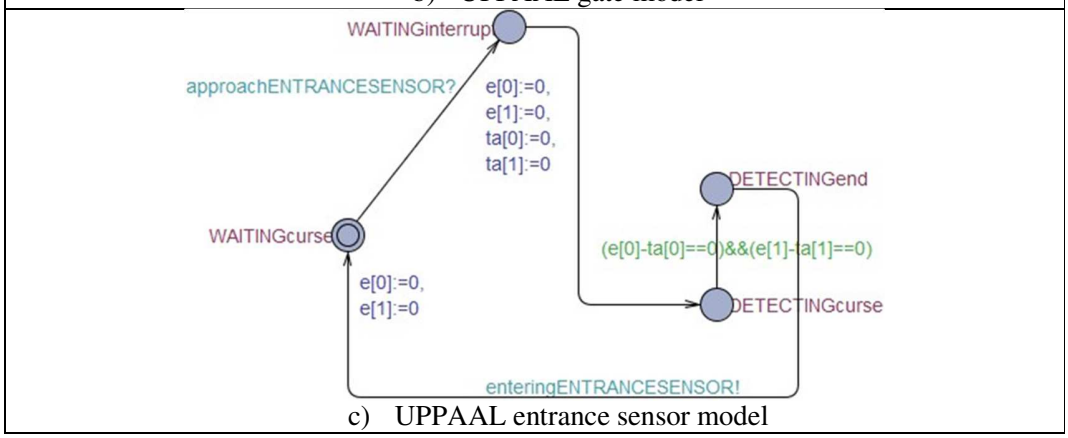
The UPPAAL model of the HiLLS crossing system is designed as a complete semantically equivalent model, using the semantics mapping patterns previously presented. The detailed UPPAAL model components are presented in Figure 11 (a-f). Each of the HiLLS component models translates semantically to a corresponding UPPAAL model. Each HiLLS configuration matches with one of the patterns presented.



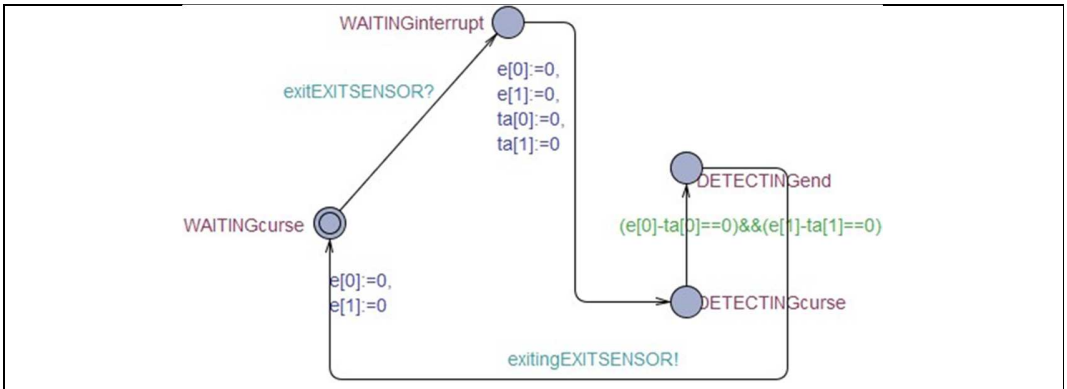
a) UPPAAL train model



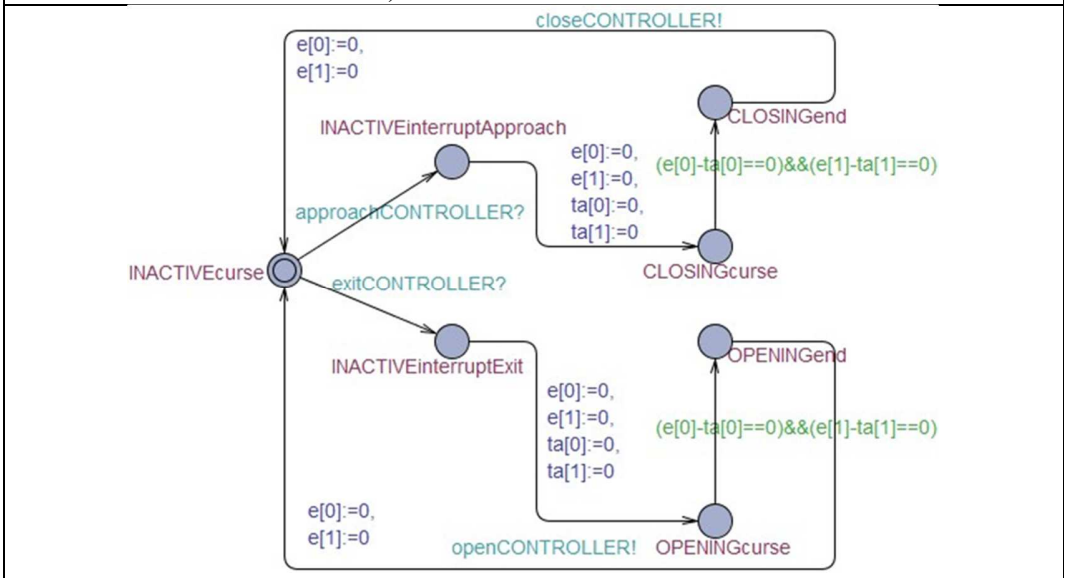
b) UPPAAL gate model



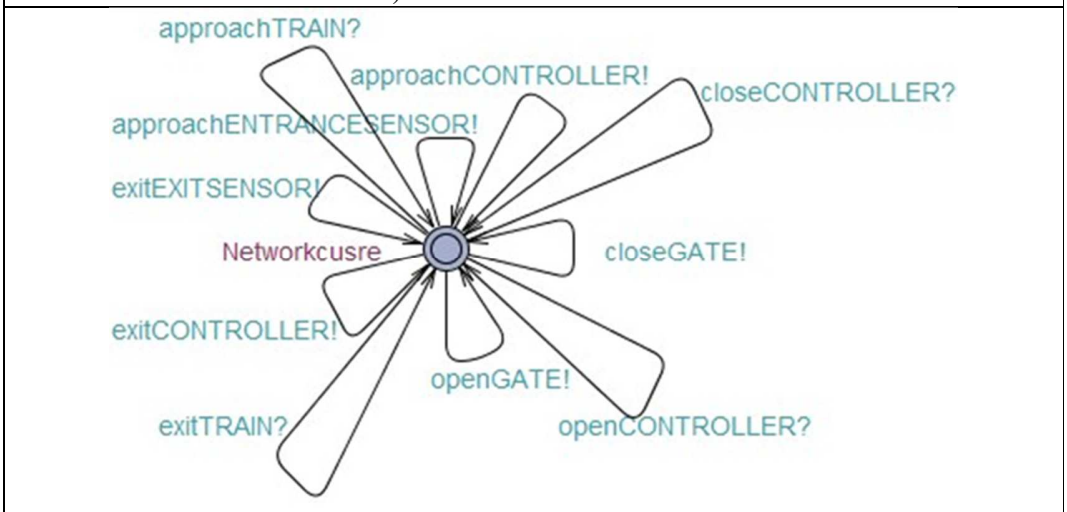
c) UPPAAL entrance sensor model



d) UPPAAL exit sensor model



e) UPPAAL controller model



f) UPPAAL crossing system model

Figure 12. UPPAAL models of the Crossing System

5.3. Results

To address the two major concerns of the system managers, we expressed requirements in TCTL, the query language of UPPAAL and we use the UPPAAL verifier to check them. We wrote the following four requirements (a screenshot of the model checker is given in Figure 13):

Property (1). E<>(Train.CROSSINGcurse) && (Gate.UPcurse)

Property (2). A[](Train.CROSSINGcurse) && (Gate.UPcurse)

Property (3). E<> ((not(Train.CROSSINGcurse))&&(not(Train.APPROACHINGcurse))) && (Gate.DOWNcurse)

Property (4). A[] ((not(Train.CROSSINGcurse))&&(not(Train.APPROACHINGcurse))) && (Gate.DOWNcurse)

Property (1) means: “eventually there exist a path when the train is crossing and the gate is open”. Property (2) means: “always the train is crossing, the gate is open”. Both properties relate to security concerns, the second one being a more serious security flaw. After running the UPPAAL verifier, Property (1) was satisfied, while Property (2) wasn’t.

Property (3) means: “eventually there exist a path when eventually there exist a path when the train is not crossing, neither is the train approaching but the gate is closed”. Property (4) means: “always the gate is closed when the train is not crossing and the train is not approaching”. Both properties relate to suitability concerns, the second one being a more serious suitability flaw. After running the UPPAAL verifier, Property (3) was satisfied, while Property (4) wasn’t.

As both concerns (safety and suitability) were breached, an alternative scenario has been considered, where the crossing level is protected by a traffic light. To do this, a sensor has been added, which emits a signal when the crossing level is closed. The control system then issues a command to the traffic light, which then turns to green. When the crossing level is open, the sensor signals it also to the controller, which then commands the light to turn red. With such an infrastructure, the security and suitability issues previously mentioned were solved. We do not give details of this alternative scenario, but we followed exactly the same principles introduced in our framework to assess the solution.

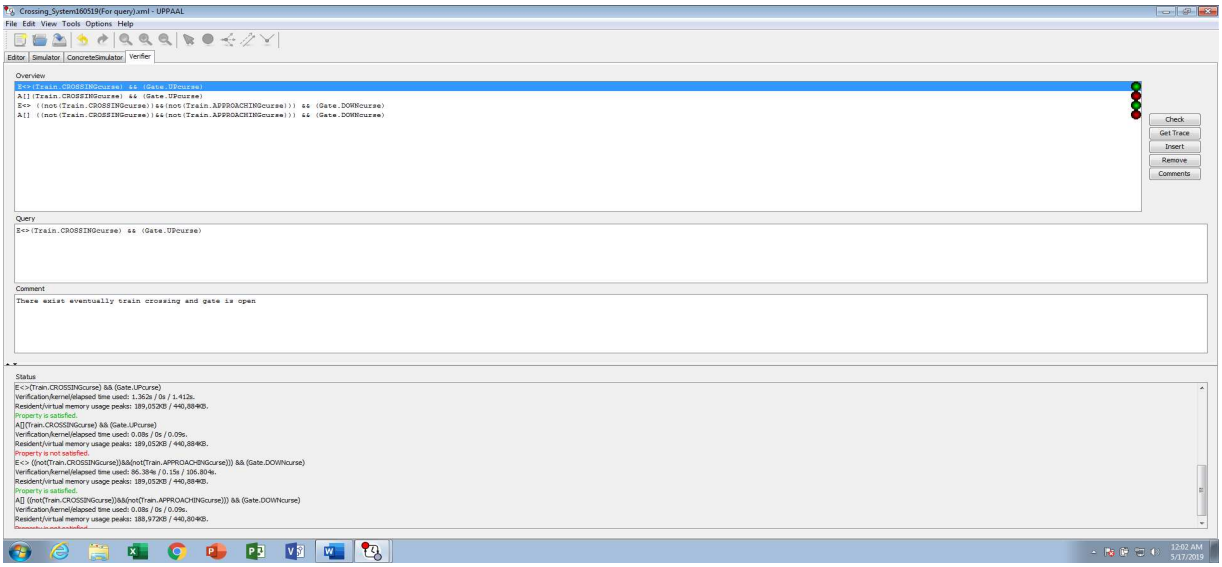


Figure 13. Screenshot of the UPPAAL model checker

6. CONCLUSION

In this paper, we have introduced a formal approach to the symbolic verification of models specified in the High-Level Language for Systems Specification (HiLLS) visual language. As HiLLS has been designed with operational semantics given for simulation (virtual time) and enactment (wall-clock time), we ensure with the new features developed that it also has a logical semantics that makes it amenable to model checking. Therefore, we achieve a true multi-analysis modeling framework in which, a single model of a complex system can cover the spectrum of analysis methods that are traditionally reachable only with a family of different models. A noticeable advantage of such a framework is that the semantical alignment required with multiple models is obtained as a built-in feature.

The verification approach developed here has the following properties:

- *Modularity:* with traditional formal verification methods, very often, a formal model to be used for property checking has to be elaborated from scratch each time a new system is considered, even if some of the components of this new system has previously been formally analyzed. This is due to the lack of composability of models with these methods. In our approach, composability is a built-in feature inherited from the hierarchical composability of DEVS. Therefore, component models can be built and formally verified independently, and later coupled to form larger systems that can also be formally verified the same way.
- *Generality:* a usual restriction in model checking is the lack of support to the use of double values for time and the lack of representation for positive infinity. Related works barely mention this limitation and often present study cases that use only integer values for time, which are also limited to finite values. We have provided a way to address both obstacles.
- *Extensibility:* rather than adopting a syntactic transformation from the model to be verified to the model to be checked (namely from DEVS model to TA model), we have adopted a semantics mapping approach. Consequently, DEVS and any of its extensions whose semantics is given in DEVS (like dynamic structure-related extensions) [37], can be interpreted without restriction in the verification formalism.
- *Expressive power*. From a more general perspective, this work differs from works related to combining multiple analysis methods in that those related works usually realize pair wise integrations of their corresponding supported methodologies. Contrariwise, the framework adopted here has the potential to integrate more than only two analysis methodologies (including but not limited to simulation, enactment, and formal verification) by providing multiple semantics to the same pivotal concrete syntax, one in each of the target methodologies.

This work is a step towards a more generic framework. In most cases, a systematic verification approach is not provided. Therefore, the user needs to build a suitable and ad-hoc verification component to be added for checking specific properties of interest. With the HiLLS automaton, we envision a full-blown methodology to combining simulation, enactment, and formal methods that prevent the modeler from such a repeated effort. It relies on a graph of property analyses, where generic verification concerns (such as safety, liveness, fairness, etc.) are identified and systematically translated in terms of HiLLS patterns. The approach presented in this paper will then be used to automate the verification process, once the HiLLS models of systems and requirements are specified.

ACKNOWLEDGEMENT

This work has been supported by the 2017 PAMI Travel Grant, and the 2019 AUST/AfDB Special Grant.

REFERENCE

[1] Hong, K. J., & Kim, T. G., 2006. DEVSpecL: DEVS specification language for modeling, simulation, and analysis of discrete event systems. Information and Software Technology, 48(4), 221-234.

[2] Estefan J.A., 2007. Survey of Model-Based Systems Engineering (MBSE) Methodologies. INCOSE MBSE Focus Group 25(8).

[3] Finkelstein A., Kramer J., Nuseibeh B., Finkelstein L., Goedicke M., 1992. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. International Journal of Software Engineering and Knowledge Engineering 2(1): 31-57.

[4] Kurpjuweit S., Winter R., 2007. Viewpoint-Based Meta Model Engineering. EMISA2007: 143-159.

[5] Gianni D., D'Ambrogio A., and Tolk A., 2014. Modeling and Simulation-Based Systems Engineering Handbook. CRC Press. ISBN 978-1-4665-7145-7.

[6] Zeigler B.P., 1976. Theory of Modeling and Simulation (1st ed.). Academic Press.

[7] Zeigler B.P., Praehofer H., Kim T.G., 2000. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press.

[8] Zeigler B.P., Muzy A., and Kofman E., 2018. Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations. Academic Press.

[9] Maïga. An integrated language for the specification, simulation, formal analysis and enactment of discrete event systems. Université Blaise Pascal - Clermont-Ferrand II, Ph.D. thesis 2015

[10] Aliyu, H.O. An Integrative Framework for Model-Driven Systems Engineering: Towards the Co-Evolution of Simulation, Formal Analysis and Enactment Methodologies for Discrete Event Systems. Ph.D. thesis in Computer Science, Université Blaise Pascal - Clermont-Ferrand II, December 2016

[11] Aliyu H.O., Maïga O., Traoré M.K., 2016. The High-Level Language for Systems Specification: A Model-Driven Approach to Systems Engineering. International Journal of Modeling, Simulation, and Scientific Computing 7(1), 1641003.

[12] Aliyu H.O., Maïga O., Traoré M.K., 2015. A Framework for Discrete Event Systems Enactment. Proceedings of 29th European Simulation and Modeling Conference. ISBN: 978-9077381-908. EUROSIS-ETI, pp. 149–156.

[13] M. K. Traoré, "Combining DEVS and Logic", pp. 307–317, 2005.

[14] Dr. Anastasia-Maria Leventi-Peetz, "Summary of the book: Formal Methods for Safe and Secure Computer Systems," Federal Office for Information Security, 2013.

[15] H. Dacharry and N. Giambiasi, 2005. Formal verification with timed automata and devs models: a case study. In Proceedings of ASSE'05. Argentine Society for Computer Science and Operational Research.

[16] H. P. Dacharry, and N. Giambiasi, 2007. A formal verification approach for DEVS, Proceedings of the 2007 Summer Computer Simulation Conference, July 16-19, 2007, San Diego, California

[17] A. Furfaro and L. Nigro, 2009. "A development methodology for embedded systems based on RT-DEVS," Innov. Syst. Softw. Eng., vol. 5, no. 2, pp. 117–127.

[18] Hong, J. S., and Kim, T. G., 1997. Real-time discrete event system specification formalism for seamless real-time software development, *Discrete Event Dynamic Systems: Theory and Applications* **7** (1997), 355-375.

[19] Cicirelli F., Furfaro A., Nigro L., Pupo F., 2010. "Temporal verification of RT-DEVS models with implementation aspects" Proceedings of the 2010 Spring Simulation Multiconference. Society for Computer Simulation International.

[20] H. Saadawi and G. Wainer, 2009. "Verification of real-time DEVS models", Proc. 2009 Spring Simul. Multiconference, p. 143.

[21] Felix Madlener, Julia Weingart, and Sorin A. Huss, 2010. "Verification of Dynamically Reconfigurable Embedded Systems by Model Transformation Rules", 4th IEEE/ACM International

Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS 2010), Oct. 2010.
[22] B. P. Zeigler, J. J. Nutaro, and C. Seo., 2017. “Combining DEVS and model-checking: Concepts and tools for integrating simulation and analysis”. International Journal of Simulation and Process Modelling, vol. 12, no. 1, pp. 2–15, 2017.
[23] Zeigler, B. P., and Sarjoughian, H. S., 2012. Guide to Modeling and Simulation of Systems of Systems. Springer.
[24] Kuhn D.R., Craigen D., Saaltink M., 2003. Practical Application of Formal Methods in Modeling and Simulation. Summer Computer Simulation Conference. Society for Computer Simulation International, pp. 726-731.
[25] Traoré M.K., 2006. Making DEVS Models Amenable to Formal Analysis. Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium. Huntsville, Alabama, USA, April 2-6. Society for Computer Simulation International.
[26] Trojet W., Berradia T., 2015. System Reliability using Simulation Models and Formal Methods. International Journal of Computer Applications 132(17):1-8.
[27] Yacoub A., Hamri M., Frydman C., 2014. A Method for Improving the Verification and Validation of Systems by the Combined Use of Simulation and Formal Methods. IEEE/ACM 18th International Symposium on Distributed Simulation and Real-Time Applications, pp. 155-162.
[28] Lano K., Clark D., Androutsopoulos K., 2004. UML to B: Formal Verification of Object-Oriented Models. Integrated Formal Methods. Springer Berlin Heidelberg, pp. 187-206.
[29] Lilius J., Paltor I.P., 1999. vUML: A tool for Verifying UML Models. 14th IEEE International Conference on Automated Software Engineering, pp. 255-258.
[30] Shah S.M., Anastasakis K., Bordbar B., 2009. From UML to Alloy and Back Again. Models in Software Engineering. Springer Berlin Heidelberg, pp. 158-171.
[31] Laleau R., Semmak F., Matoussi A., Petit D., Hammad A., Tatibouet B., 2010. A First Attempt to Combine SysML Requirements Diagrams and B. Innovations in Systems and Software Engineering 6(1-2):47-54.
[32] Kapos G-D., Dalakas V., Nikolaidou M., Anagnostopoulos D., 2014. An integrated framework for automated simulation of SysML models using DEVS. SIMULATION 90(6):717–744.
[33] Risco-Martín J.L., Jesús M., Mittal S., Zeigler B.P., 2009. eUDEVS: Executable UML with DEVS Theory of Modeling and Simulation. SIMULATION 85(11-12):750-777.
[34] Schamai W., Fritzson P., Paredis C., Pop A., 2009. Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior using Graphical Notations. 7th International Modelica Conference, Linköping University Electronic Press, pp. 612-621.
[35] Shaikh R., Vangheluwe H., 2011. Transforming UML2. 0 Class Diagrams and Statecharts to Atomic DEVS. Symposium on Theory of Modeling & Simulation: DEVS.
[36] Zeigler B.P., Traoré M.K., Zacharewicz G., 2018b. Value-based Learning Healthcare Systems: Integrative Modeling and Simulation. Institution of Engineering and Technology (The IET Book Series on e-Health Technologies). ISBN: 978-1-78561-326-5, 376p.
[37] Maiga, Oumar, Hazmat Olanrewaju Aliyu, and Mamadou Kaba Traore., 2015. “A New Approach to Modeling Dynamic Structure Systems.” Proceedings of the 2015 European Simulation Modeling Conference, 141–48. https://doi.org/10.1017/CBO9781107415324.