HAL
open science

# A DEVS-based pivotal modeling formalism and its verification and validation framework

Kehinde G Samuel, Nourou-Dine M Bouare, Oumar Maïga, Mamadou Kaba Traoré

## To cite this version:

Kehinde G Samuel, Nourou-Dine M Bouare, Oumar Maïga, Mamadou Kaba Traoré. A DEVS-based pivotal modeling formalism and its verification and validation framework. SIMULATION: Transactions of The Society for Modeling and Simulation International, inPress, 10.1177/0037549720958056 . hal-02968082

**HAL Id: hal-02968082**

**https://hal.science/hal-02968082v1**

Submitted on 22 Oct 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# A DEVS-based pivotal modeling formalism and its verification and validation framework

Kehinde G. Samuel [(1)]
keliake@aust.edu.ng
Nourou-Dine M. Bouare[(2)]
nm.bouare@usttb.edu.ml
Oumar Maïga[(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

System verification is an ever-lasting system engineering challenge. The increasing complexity in system simulation requires some level of expertise in handling the idioms of logic and discrete mathematics to correctly drive a full verification process. It is recognized that visual modeling can help to fill the knowledge gap between system experts and analysis experts. However, such an approach has been used in one hand to specify the behavior of complex systems, and on the other hand to specify complex requirement properties, but not simultaneously. This paper proposes a framework that is unique in supporting a full system verification process based on the graphical modeling of both the system of interest and the requirements to be checked. Patterns are defined to transform the resulting models to formal specifications that a model checker can manipulate. A real-time crossing system is used to illustrate the proposed framework.

**Keywords:** High Level Language for Systems Specification (HiLLS), Discrete Event System Specification (DEVS), formal verification, temporal logic, model transformation, UPPAAL.

## 1. Introduction

The continuous growth of systems of various kinds through human activities and the increasing complexities of the activities themselves lead to the continuous development of formal techniques to ensure safe and secure systems. The evolution of a system's functional requirements is often inevitably accompanied by a corresponding evolution of its complexities in structure and behavior, hence more efforts are required to monitor this evolution process to ensure that the resulting system is reliable and safe. This may involve a combination of scientific techniques such as simulation, prototyping and formal analysis to carefully investigate the model before the implementation of the emerging system. Significant efforts and advances have been made towards making these techniques accessible individually. However, not much success has been recorded in unifying the various formalisms, thus, different models of different aspects of a system have to be treated. This is characterized by communication gaps between the different experts and consequently reduced efficiency of the entire process.

Simulation models are usually represented by different domain specific modeling languages (DSML) most of which have no precise semantic definition. The semantics of simulation models are left to model simulators and translators; these are defined by general-purpose programming languages, which is unacceptable for formal analysis [1]. Although the syntax of some recent DSML is formally described with metamodel, generally, the lack of formal model transformations contributes to the challenges of formal

analysis at the model level. It is, therefore, a challenge to describe simulation models and its requirement specification formally to improve formal verification and analysis of models.

The High-Level Language for System Specification (HiLLS) has been proposed to fill such a gap [2]. HiLLS is a scalable visual modeling language that serves as a one-stop reference point through a harmonious combination of modeling paradigms from system theory and software engineering to integrate the different aspects of a system in one coherent whole. As such, it is a pivotal visual language that allows model simulation, enactment, and formal analysis. In order to ensure these combined features, concepts have been borrowed from three formalisms that are universal, each in one of these three analysis domains, and seamlessly integrated through formalism weaving techniques, namely DEVS, UML and first-order logic. DEVS, which is recognized as a universal simulation modeling formalism [3], provides the semantic domain for HiLLS-specified models simulation. Similarly, a UML-specified pattern provides the architecture for HiLLS-specified models enactment [4], while first-order logic is used as the semantic domain for formal verification of HiLLS-specified models [5].

However, while the objectives of having a highly communicable graphical concrete syntax and multiple semantic domain mappings for simulation, enactment, and accessibility to formal analysis, have been achieved, there is a major concern in making such an approach effectively and easily usable. As illustrated by Figure 1, such a concern stands for each of the analysis methods (i.e., simulation, enactment, and model checking) in that a complete framework is needed to allow users to drive a whole process from visual modeling of a system to its analysis. Indeed, while a DEVS model can be automatically derived from a HiLLS specification, the full simulation process of such a model entails the specification of additional aspects (such as the experimental frame, simulation initialization, parameter tuning, etc.). Similarly, the full enactment process of the HiLLS-derived model entails the definition of additional aspects (such as real-time concerns, human-in-the-loop interface, etc.). And so does the full formal verification process of the HiLLS-derived model. The latter is the focus of this paper, and we aim at achieving a supporting framework for such a process.

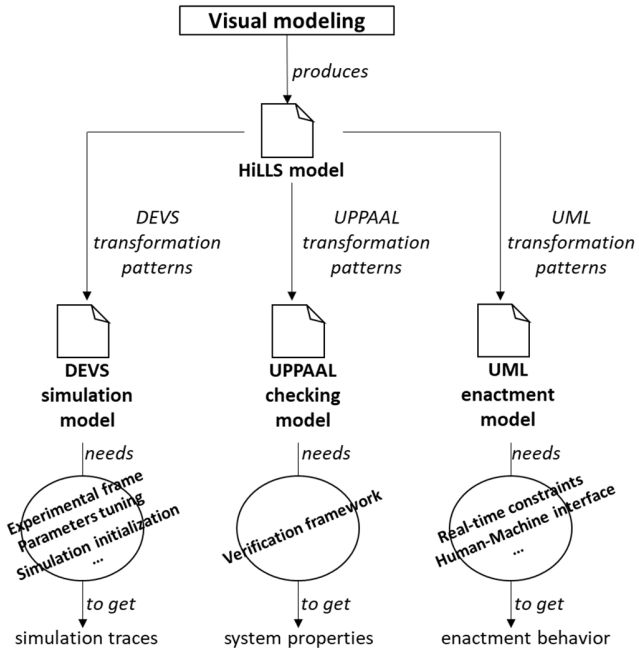



*Figure 1.* From pivotal visual modeling to multiple analyses of complex systems

The paper is organized as follows. Section 2 discusses related works. Section 3 presents the HiLLS formalism and its editor. Section 4 introduces patterns that will serve as the building blocks for defining HiLLS semantics in a way that it is amenable to formal checking. Section 5 extends HiLLS to the specification of system requirements and provides semantics for such an extension. Section 6 shows how both the HiLLS-based system modeling and HiLLS-based requirements specification fall within a full formal verification framework. Section 7 illustrates the application of the framework with a well-known study case. Section 8 concludes the paper, by summarizing the work done and by giving perspectives for future work.

## 2. Related Works

Related works have addressed the formal analysis of DEVS models. These proposals range from formal model-checking of sub-classes of DEVS, the transformation of DEVS into formal methods for verification purposes, generation of traces from DEVS models for testing, or introducing clock constraints to DEVS to conform to formal methods. We present some notable works in this area:

- Hong and Kim [6] proposed a method of verification of DEVS models in the DEVSim++ environment. The approach was to specify the model in DEVS and use temporal logic to specify the properties and time constraints of the system. The authors use a projection technique to reduce the state space. The lifetimes of the states are not taken into account, but the temporal logic allows expressing constraints on sequences of states. The technique used by the authors is very similar to the technique of model-checking using Buchi automata [7].
- Zeigler et al. [8] used a subclass of DEVS models having finite sets of states, inputs, and outputs, named FD-DEVS [9] to map the system representation onto a non-deterministic automaton that is subject to model checking using the SPIN/PROMELA model checker.
- Several related works align on the principle combining DEVS with Timed Automata, and the use of the UPPAAL model checker [10-13].

Our work differs from all of the above in that we use a pivotal visual notation at both sides: one that has equivalent DEVS representation for system behavior and the other that has equivalent temporal logic representation for system requirements. Moreover, by building a full framework based on these visual notations, we provide a systematic verification approach, which most of the related works don't, as they require to add a suitable verification component for checking specific properties of interest.

## 3. HILLS-Based System Modeling

HiLLS can be seen as a visual language for DEVS (see in annex a brief recall of DEVS), with specific features for formal analysis and direct prototyping [2], [4], [14]. This is the point of view adopted here, although any of the two other formalisms and their underlying paradigms could be used as the entry point.

### 3.1. HiLLS Syntax

A template of how HiLLS represents a DEVS model is shown in Figure 2. A HiLLS-specified system is represented by an HSystem, which is denoted by a box similar to the UML class with an additional horizontal compartment and two vertical compartments. The left (respectively right) hand side vertical compartment has input (respectively output) ports attached to it. The concept of a port is defined as in DEVS. All declarations in HiLLS (whether ports or any other variables or functions) are done in first-order logic. The top horizontal compartment contains the name of the model and the declaration of its parameters. The immediate compartment below contains the declaration of state variables. The third compartment from

the top contains the definitions of operations that use and manipulate all variables, including parameters and ports. Therefore, while a message received on some given input port causes a change of the internal state of the model, a call to some given modifier operation causes a change of the value of some given parameter. The bottom compartment contains the system's behavior described by the configuration transition diagram (i.e., the HiLLS automaton), an automaton in which nodes are configurations and which edges are configuration-to-configuration transitions that can occur in the system. A configuration is defined by the assignment of specific values or constraints to state variables.

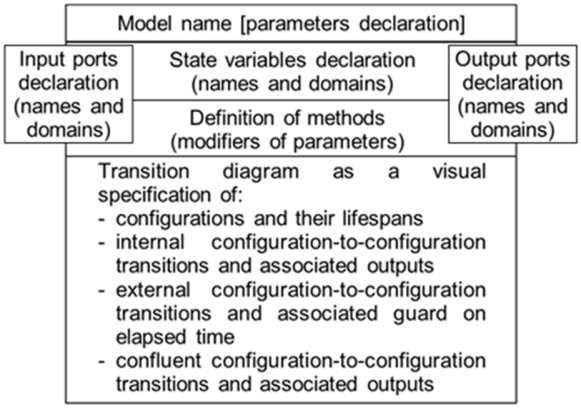



*Figure 2.* Template for HiLLS representation of a DEVS model

One can notice that the assignment of a specific value to each of the state variables gives a state in the sense of DEVS (i.e., a particular configuration), while the assignment of constraints (rather than specific values) to some or all of the state variables gives a configuration that corresponds in DEVS to a family of states (instead of a single one). As such, a configuration depicts a set of properties that several states share. Configurations are a way to cluster a DEVS state set (whether finite or not) into a finite partition [5]. Figure 3 shows how the syntactic elements are visually captured by graphical elements.

A configuration has three visual representations: finite, passive and transient configurations. A finite configuration (Figure 3.a) is a 4-compartments box, which respectively contains the label of the configuration, the logic specification of its properties (such as the assignment of values and constraints to state variables), its sojourn time (which corresponds in DEVS to the value of the time advance function at states falling within this configuration), and the description of activities to be carried out when the system is in this configuration (which has no equivalence in DEVS but serves for the purpose of enactment). An infinite configuration (Figure 3.b) is a configuration in which sojourn time is $+\infty$; therefore, its visual representation is reduced to a 3-compartments box, the compartment related to sojourn time being replaced by a double line at the right-hand edge of the box. A transient configuration (Figure 3.c) is the one in which sojourn time is 0; therefore, its visual representation is reduced to a 3-compartments circle. A black circle (Figure 3.g) allows to make reference to the initial configuration of the model.

The three kinds of configuration transitions are denoted by different labeled arrows with the operations accompanying the transitions (for the update of state variables, when needed) as part of the labels. For internal transition (Figure 3.d), the value sent on the output port is also part of the label. For external transition (Figure 3.e), the label includes the triggering condition on the receipt of a value on a port and the time elapsed by the model in its current configuration. The label for confluent transition (Figure 3.f) is similar, except that there is no condition on the elapsed time (since it is known to be the sojourn time in this case). Decision nodes (Figure 3.h) can be used to define various possible routes during a transition, depending on conditions to be met by state variables.

An HClass (Figure 3.j) denotes a software component that does not represent the model of a dynamic system, as opposed to an HSystem (Figure 3.i). As such, it is a simple resource manipulated by model components and corresponds to a "regular" class in UML with attributes and methods specified by logical predicates. Both HSystem and HClass can be parameterized. The HSystem and HClass components can be linked by relationships such as aggregation, composition, generalization, and reference, with cardinalities attached to, as described by the UML metamodel (indeed HClass and HSystem are specializations of the UML Classifier mother class).

Consequently, an HSystem can be composed of other HSystems, and such a description corresponds to a DEVS coupled model. Interestingly, DEVS atomic and coupled models are visually described in HiLLS the same way.

Moreover, while a traditional coupled model will have in HiLLS a single configuration specifying the coupling information between its sub-components, a dynamic structure coupled model will have a configuration transition diagram with more configurations, each of them specifying a given architecture of the coupled model, and the transitions specifying the rules for the dynamic change of structure.

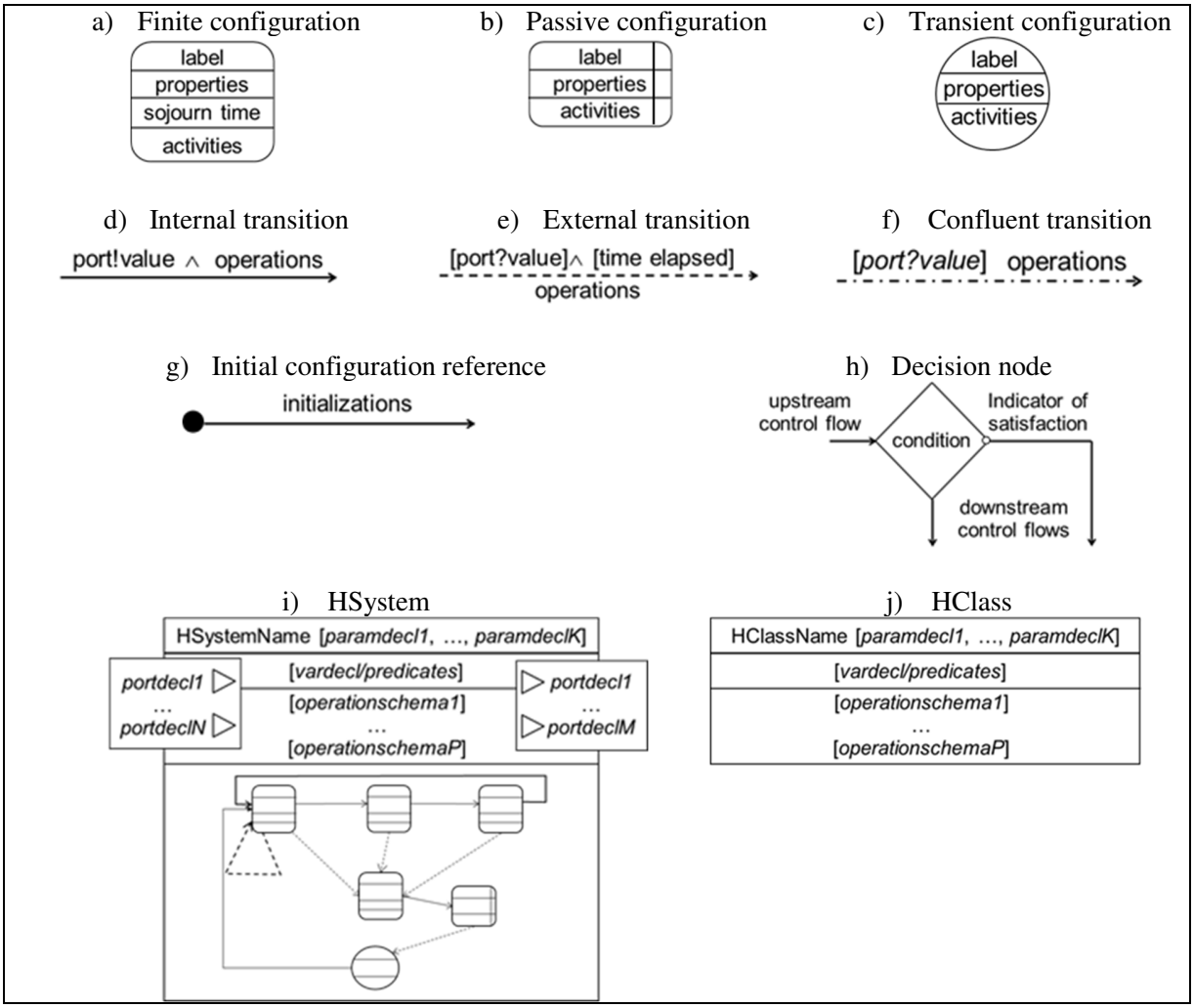



*Figure 3.* HiLLS concrete syntax

### 3.2. HiLLS Editor

An editor has been developed for HiLLS modeling, using the Graphical Modeling Framework (GMF) of the Eclipse IDE. As HiLLS model specification requires both graphical and textual representations, Xtext was used to capture the textual aspects of the model (such as labels, properties, and activities) and EuGENia, a plugin that overlays GMF, was used to process the geometric shapes that compose the model. The HiLLS Editor allows the drag and drop modeling of complex systems, as shown in Figure 4. The editor displays a workspace, with a central area where the model is drawn. To the left of the workspace is the Model view, where appears the hierarchy tree of the model, which provides easy navigation throughout the model components. On the right side of the workspace, are three panels for modeling. The upper palette contains all the basic elements of the HiLLS concrete syntax (Configuration, Declaration, Activity, HSystem, etc.). The middle panel displays all connectors (Transition, Aggregation, Reference, and Composition). The bottom panel displays the temporal logic items for requirement specification.

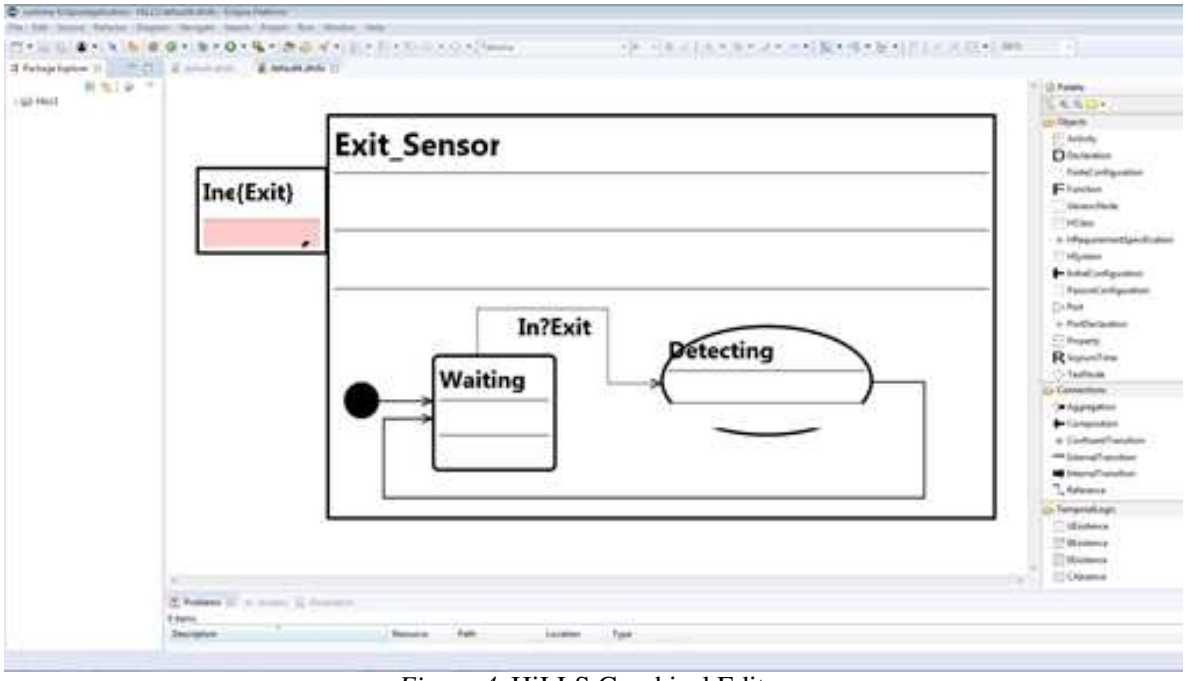



*Figure 4.* HiLLS Graphical Editor

### 3.3. HiLLS-DEVS Relation

Details of the correspondence between a DEVS model and a HiLLS automaton are given in [5]. The idea of HiLLS automaton is that a DEVS model (whether with finite or infinite state set) can be represented graphically with a finite HiLLS automaton for the purpose of formal analysis, without any loss of behavioral property, while still being adequate for simulation and enactment. Multiple states of an atomic DEVS model are mapped onto a single configuration in the corresponding HiLLS automaton. A Coupled DEVS model is mapped onto a composed HiLLS automaton, which uses its configuration(s) to specify the coupling information between the sub-components of the corresponding DEVS model. The focus of this paper not being the HiLLS-DEVS relation, we avoid here the provision of unnecessary details and refer interested readers to [5] for more on that aspect. We rather focus on providing a full formal verification framework to HiLLS. However, Figure 5 shows a DEVS model (a patient) and its HiLLS counterpart.

$DEVS_{Patient} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, with

$X = \{(in, x) / x \in Viruses\}$

$Y = \{(out, x) / x \in Viruses\} \cup \{(status, y) / y \in \{S, E, I, D\}$

$S = \{1, 2, 3, 4, 5\} \times \Re_0^{+\infty}$

$ta : S \rightarrow \Re_0^{+\infty}$

$ta(s, \sigma) = \sigma \quad \forall s \in \{1, 2, 3, 4, 5\}$

$\delta_{int} : S \rightarrow S$

$\delta_{int}(2, \sigma) = (3, t_{INC}) \quad \forall \sigma \in \Re_0^{+\infty}$

$Prob(\delta_{int}(4,\sigma)=(5,+\infty))=\sigma_{FAT}$ and $Prob(\delta_{int}(4,\sigma)=(1,+\infty))=1-\sigma_{FAT}$, $\forall \sigma \in [0, t_{INF}]$

$Prob(\delta_{int}(3,\sigma)=(3,t_{INC}))=\sigma_{INF}$ and $Prob(\delta_{int}(3,\sigma)=(1,+\infty))=1-\sigma_{INF}$, $\forall \sigma \in [0, t_{INC}]$

$\lambda : S \rightarrow Y$

$\lambda(2, \sigma) = (status, E), \forall \sigma \in \Re_0^{+\infty}$

$Prob(\lambda(3,\sigma)=\{(status,I),(out,vir)\})=\sigma_{INF}$ and $Prob(\lambda(3,\sigma)=(status,S))=1-\sigma_{INF}$, $\forall \sigma \in [0,t_{INC}]$

$Prob(\lambda(4,\sigma)=(status,D)) = \sigma_{FAT}$ and $Prob(\lambda(4,\sigma)=(status,S)) = 1-\sigma_{FAT}$, $\forall \sigma \in [0, t_{INF}]$

$\delta_{ext} : Q \times X \rightarrow S$, with $Q = \{((s,\sigma), e) / (s,\sigma) \in S, 0 \leq e < \sigma\}$

$\delta_{ext}((1, \sigma), e, (in, vir)) = (2, 0) \quad \forall \sigma \in \Re_0^{+\infty} \quad \forall e \in [0, \sigma)$

$\delta_{ext}((3, \sigma), e, (in, vir)) = (3, \sigma-e) \quad \forall \sigma \in [0, t_{INC}] \; \forall e \in [0, \sigma)$

$\delta_{ext}((4, \sigma), e, (in, vir)) = (4, \sigma-e) \quad \forall \sigma \in [0, t_{INF}] \; \forall e \in [0, \sigma)$

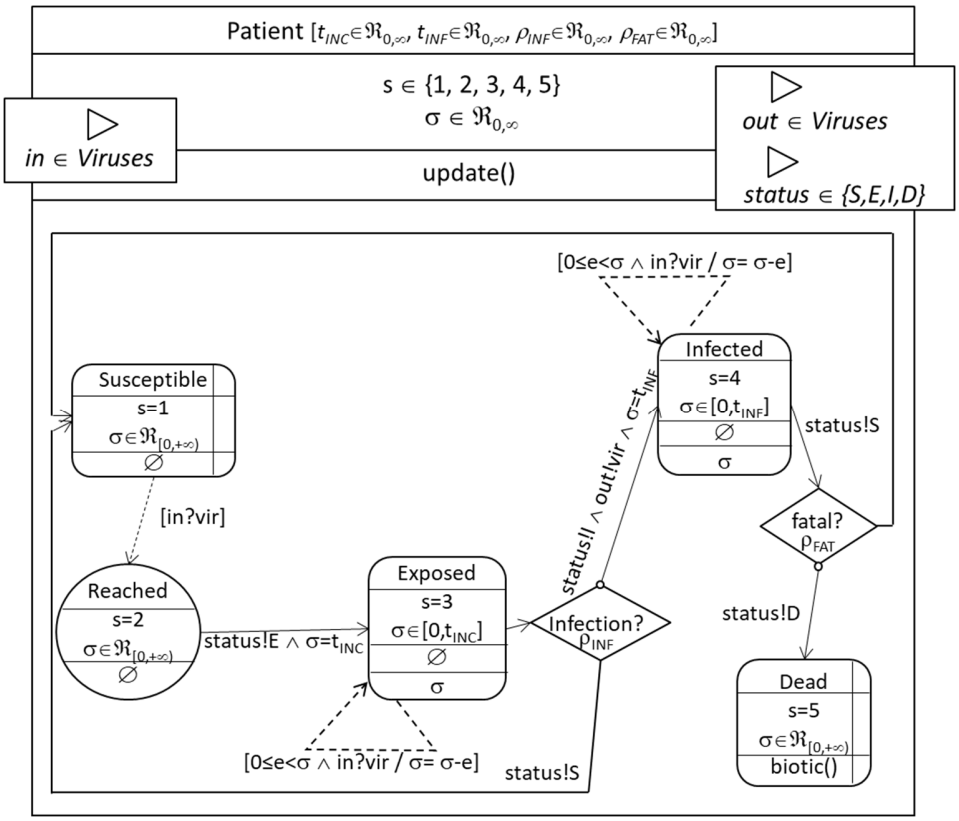



*Figure 5.* DEVS model example and its HiLLS counterpart

A larger illustration of the concepts introduced in this section is given through the application presented in Section 7. As the application is meant to demonstrate the entire methodology proposed in the paper, it is

presented after all theoretical aspects of the methodology are introduced. However, at this stage, in order to get an illustration of HiLLS modeling on the application, the reader can get the application presentation in Section 7, and the expression of the corresponding system models in Section 7.1 (which refers to Annex D for complements). Going to Section 7's introduction and Section 7.1, and then coming back to Section 4 is a way to match Section 3 with the application and possibly improve the readability of this section, while keeping the general structure of the paper from theory and methodology to application.

## 4. HiLLS-to-UPPAAL Patterns

Timed automata are among the most widely used models for the verification of real-time systems. To semantically map HiLLS to one of the formal method tools available (namely UPPAAL), we define patterns that provide building blocks to building the HiLLS formal method-based semantics. The subsequent sub-sections present these patterns.

Similar to matching Section 3 with the application presented in Section 7, reader can switch between sub-sections of Section 4 at one hand, and Section 7.1 and Annex E at the other hand, to better match the concepts introduced in Section 4 with the application.

### 4.1. Semantic pattern for configuration

The pattern shown in Figure 6 expresses the formal semantics of a HiLLS model in a given configuration, using a Timed Automaton (TA) composed of four types of locations, namely $C_{curse}$, $C_{end}$, $C_{interrupt}$ and $C_{dilemma}$, and a unique clock.

In this pattern, the lifetime of a configuration C starts at $C_{curse}$, where the property of the configuration is (e < ta) AND (mail =∅), with mail representing the input bag of the system. To overcome the fact that model checking tools often restraint on time representation as limited to integer values only, we represent a real-time value, such as e (elapsed time) and ta (time advance), as a pair of integer values ($t_{int}$, $t_{dec}$), where $t_{int}$ is the integral part of the time value, and $t_{dec}$ is its decimal part. That way, t = t' is translated into (($t_{int}$-$t'_{int}$=0) AND ($t_{dec}$-$t'_{dec}$=0)), while t < t' translates to (($t_{int}$-$t'_{int}$<0) OR (($t_{int}$-$t'_{int}$=0) AND ($t_{dec}$-$t'_{dec}$<0))), with AND (respectively OR) being the conjunction (respectively disjunction) logical operator. Pairs of integer values ($t_{int}$, $t_{dec}$) are implemented in UPPAAL as 2D arrays, where t[0] implements $t_{int}$, and t[1] implements $t_{dec}$. Such a representation of time is one of the possible forms of superdense time as introduced in [15] and brought to discrete event simulation in [16] within a very formal framework.

When the system enters the $C_{curse}$ location, e is set to 0.0 (hence e[0] = 0, and e[1] = 0), and ta is determined by update(ta). $C_{curse}$ means that the system is in the curse of C, and no input has been received yet by the system, nor the lifetime of C has elapsed yet. The TA takes a transition from $C_{curse}$ to $C_{end}$ when (e = ta) AND (mail = ∅), but if before that condition is met, a message ?x is received at elapsed time e, the TA transits to $C_{interrupt}(x)$. There are as many locations $C_{interrupt}(x)$ as the number of possible values of x that the system can receive in the C configuration. From each $C_{interrupt}(x)$, the TA will do as many external transitions as the number of possible values for e. From a theoretical point of view, there is a potential of combinatorial explosion for the number of locations of type $C_{interrupt}(x)$, as well as for the number of external transitions that can be taken from each location $C_{interrupt}(x)$. However, in practice, models have very limited number of different cases for an external transition. In any case, this aspect is intrinsically a limitation of our approach.

at the $C_{end}$ location, an internal transition takes place and an output !y is sent, unless a message is received exactly at that moment, leading then to a transition to $C_{dilemma}(X)$ where a confluent transition takes place and an output !y is sent. Internal, external and confluent transitions taken from any location of the C

configuration lead, each to the location corresponding to the curse of another configuration. Then the same pattern applies to that new configuration, and so on.

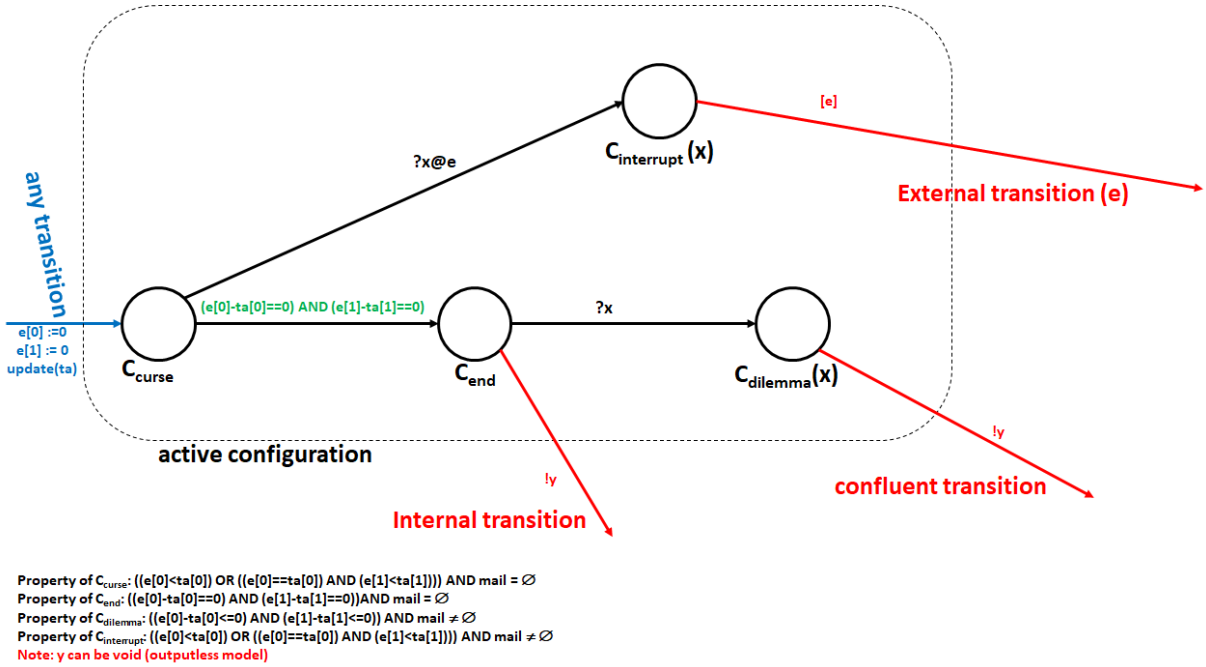
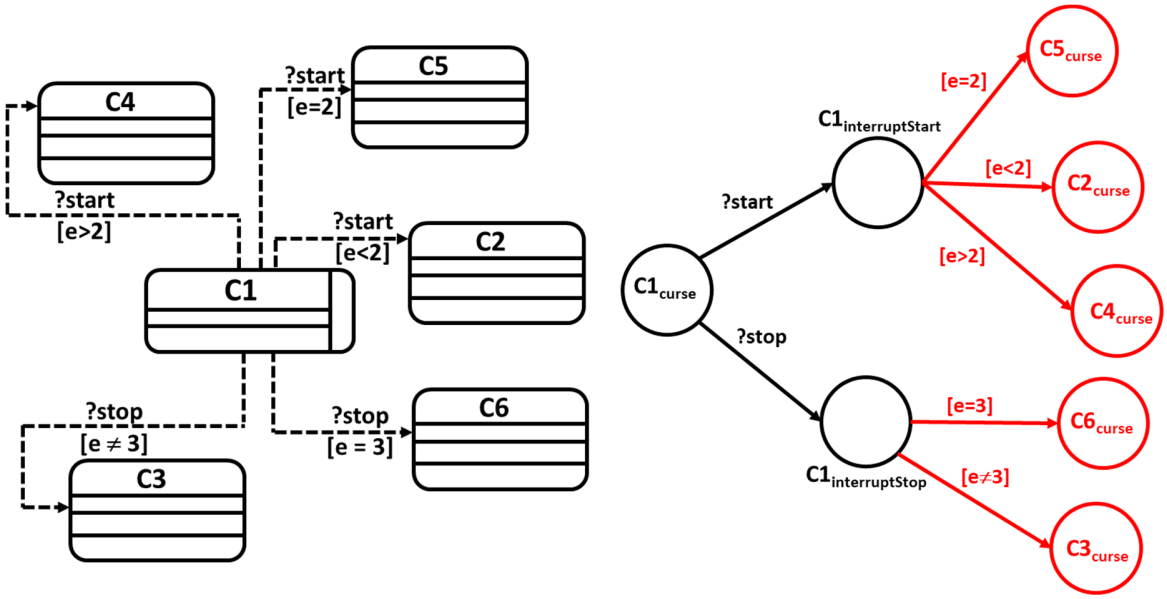



*Figure 6.* Semantics pattern for HiLLS active configuration

Figure 7 illustrates how the semantics of a HiLLS model is given by a TA, using the pattern. Figure 7a shows the HiLLS automaton, and Figure 7b shows how its semantics is built in TA (but only the translation for the C1 configuration is shown).




(a) (b)

*Figure 7.* Illustration of (partial) semantics mapping from HiLLS to UPPAAL

From $C1_{curse}$ of Figure 7b, two types of external transitions are possible based on the type of message received. If ?start is received then a transition to $C1_{interruptStart}$ (for $C1_{interrupt}$(Start)) will happen and if ?stop is received, then a transition to $C1_{interruptStop}$ (for $C1_{interrupt}$(Stop)) will happen. The external transition from each of the $C1_{interrupt}$(x) can go to different other configurations' curses, depending on the elapsed time, as stated by the HiLLS model.

### 4.2. Pattern variants

From the general configuration pattern previously presented, we derive the variants presented by Figure 8.

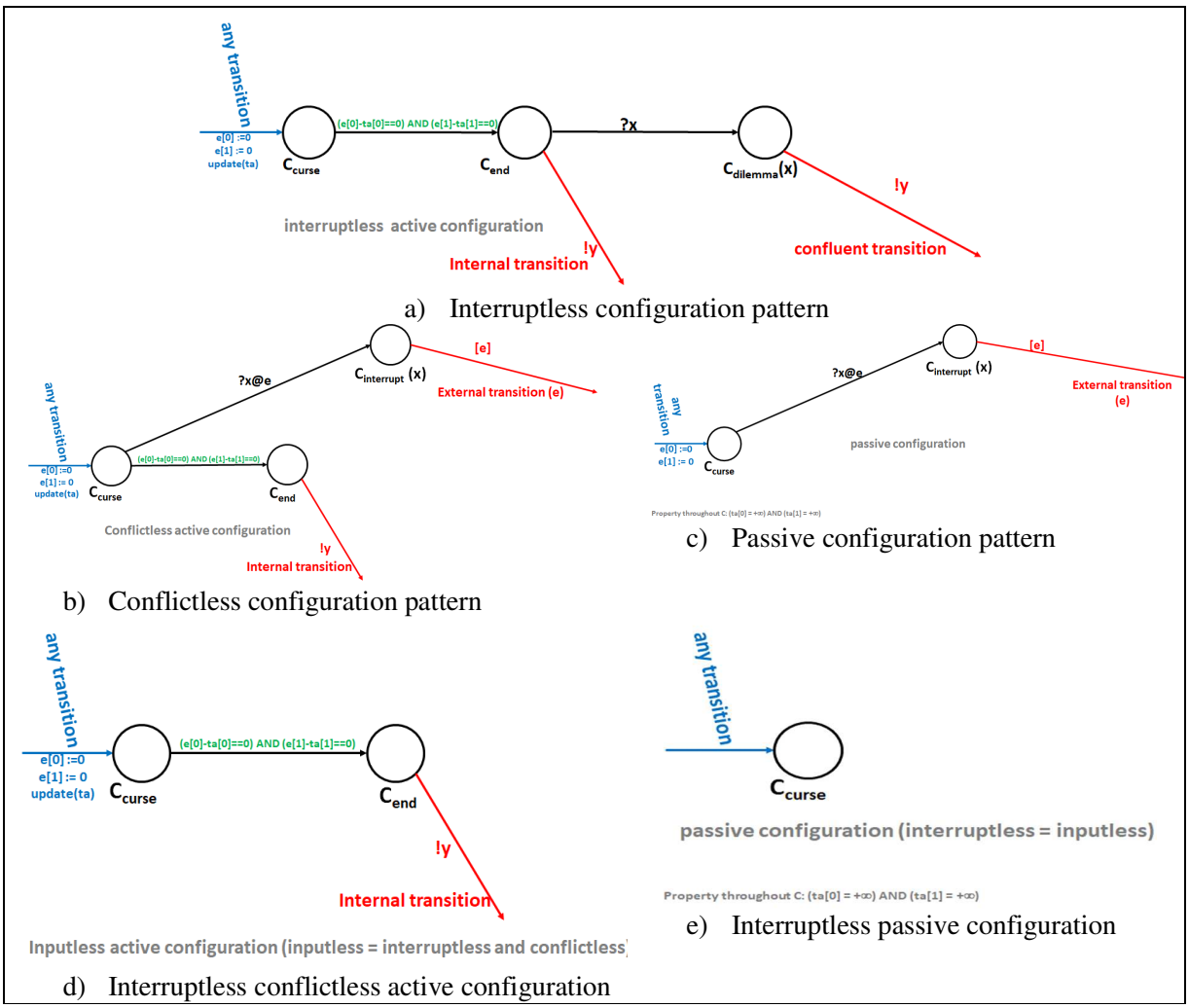



*Figure 8.* Variants of the semantics pattern for HiLLS configurations

These variants, together with the general pattern, form the building blocks for a complete translation of any HiLLS model into its UPPAAL TA counterpart:

- Figure 8a presents the *interruptless configuration pattern*, where only internal and confluent transitions are possible, as no input is received by the corresponding HiLLS model in the corresponding configuration (therefore no external transition happens).

- Figure 8b presents the *conflictless configuration pattern*, where only internal and external transition are possible (no confluent transition).
- Figure 8c presents the *interruptless and conflictless configuration pattern*, where an only internal transition happens.
- Figure 8d presents the *passive configuration pattern*, where only external transitions are possible.
- Figure 8e presents the *interruptless passive configuration*, no transition (whether external, internal or confluent) is possible.

### 4.3. Patterns for Hierarchical Composition

None of the patterns defined in the previous section takes care of the case of HiLLS composed automata. In order to handle this case, we proceed as follows:

(1) A feedback loop location transition is used 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) Messages are systematically labeled in UPPAAL with the name of the sender/receiver components. That way, when a message is sent by a HiLLS component, its encapsulating HiLLS model identifies the origin of the message, and with the composition information, transforms it to be a message tagged with the name of the recipient.

The composition steps described above is illustrated in Figure 9. The interruptless passive configuration pattern corresponding to a composed HiLLS model translates into a single location in UPPAAL, in conformance with the pattern of Figure 8e. Assuming 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 of the composed HiLLS has a feedback loop such that if a message is sent by the sender component, it is intercepted by this location, which in turn will emit a message with the same content but tagged with the identity of the receiver component (we know that the composition information is defined in the configuration's property).

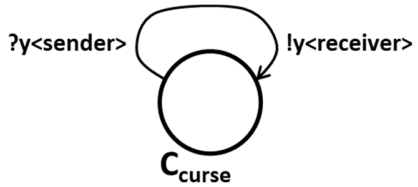



*Figure 9.* Semantic pattern of message transmission between two components by the composed model

Another way to address the case of HiLLS composed automata is to apply the "closure under composition" property of DEVS, which establishes that any coupled DEVS model has a corresponding atomic model. That way, any coupled model can be translated into an equivalent atomic model, and doing this would avoid the need to use the pattern introduced here. However, such an approach can be error-prone if not automated, as well as time-consuming for very complex systems hierarchies. In a given situation, the choice of either using the pattern introduced here or flattening the HiLLS composed automata before translating it into UPPAAL depends on how practical would be the application of the closure under composition property.

### 4.4. HiLLS to UPPAAL Transformation with Atlas Transformation Language (ATL)

We automate the HiLLS-to-UPPAAL transformation by implementing the general configuration pattern and its variants as ATL (Atlas Transformation Language) rules. ATL [17] is a model transformation language specified as both a metamodel and a textual concrete syntax. ATL follows the model transformation process that takes a source model in a specific form as inputs and outputs another form of the target model according to a set of predefined rules. ATL snippets of HiLLS-to-UPPAAL rules are detailed in Annex B.

## 5. HILLS-Based Requirement Specification

In the context of a HiLLS-based systems engineering, the sequence of configurations visited during execution describes the behavior of a system. Hence, a temporal logic can be used for the specification of, and reasoning with, the behavior of an ideal system. This behavior of the ideal system can serve as the metamodel that specifies the required behavioral properties of the real system. Therefore, with the help of verification techniques such as model checking [18], we can verify whether or not a given model of the real system satisfies the required properties.

Good candidates for the description of temporal property requirements exist, such as Linear Temporal Logic – LTL [19], or Computation Tree Logic (CTL) also known as branching temporal logic [20]. However, it is common knowledge that dealing with such formalism is usually non-trivial. It takes some level of expertise in handling the idioms of logic and discrete mathematics to correctly read and/or write complex requirement properties. Lack of this expertise has been widely acknowledged by formal methods researchers as one of the main inhibitors to the wide adoption of formal verification tools.

In an effort to proffer a solution to this problem, Dwyer et al. [21] hypothesized that the experience base of experts in specification formalisms could be captured in parameterized patterns in formalism-independent formats to allow for systematic mapping to equivalent representations in some known specification formalism. They argued that this could be an easy way to transfer the experiences of experts in the domain to emerging practitioners and potential users.

Dwyer et al. were inspired by the successes that had been recorded with the use of design patterns to provide guidance on the best ways to language features to solve recurring problems by documenting tested solutions to such problems in patterns that can be easily reused to solve similar problems. With this, they envisioned the success of a pattern-based approach to the formal specification of properties of finite-state systems for verification. The output of their research was the recognition of some commonly occurring requirement property patterns from a collection of over five hundred property specifications they collected about thirty-five sources comprising academia and industry. They then proposed parameterized templates for the recognized property patterns in five property specification formalisms, including LTL, CTL, and LCTL, which some other researchers later reproduced in additional formal methods' languages [22-23].

We propose to use variants of the elements of HiLLS for expressing graphically the templates suggested by [21]. We believe that uniformity of notations in both system and requirement models, due to the use of a pivotal language, will aid the user's specification and understanding of required temporal properties for complex systems. Similar benefits have motivated Meyers et al. [24] and Klein & Giese [25] to propose a framework to support the use of domain-specific notations for specifying properties in Domain-Specific Languages (DSLs).

To express the temporal properties, we propose basic notations (Figure 10) as the building blocks to specify temporal properties based on Dwyer's property patterns:

- The universal existence configuration notation (Figure 10.a) is a generic representation for any configuration that matches the set of information given by the notation (name, or/and predicates). In addition, this pattern specifies that all configurations visited during the lifetime of a system within a given scope (to be defined) must match the set of information given by the pattern.
- The eventual existence configuration notation (Figure 10.b) is a similar generic representation, with the difference that it only requires at least one of the configurations visited within a given scope (to be defined) must match the set of information given by the pattern.
- The absence configuration notation (Figure 10.c) is also defined similarly, with the difference that it requires that any of the configurations visited within a given scope (to be defined) must match the set of information given by the pattern.
- The bounded configuration notation (Figure 10.d) is defined the same generic way, with the difference that it requires the configurations visited within a given scope (to be defined) must match the set of information given by the pattern only a bounded number of times. The lower (respectively upper) bound is indicated in the lower (respectively upper) compartment of the circle at the right-hand side of the generic configuration. The default value (i.e., when not indicated) for the lower (respectively upper) bound is 1 (respectively $+\infty$).
- The implication notation (Figure 10.e) relates two generic configuration notations in that the matching of the first one implies the matching of the second.
- The immediate implication notation (Figure 10.f) is similar, with the difference that the implied configuration must be matched at the next transition of the system.
- The concurrency notation (Figure 10.f) corresponds to the logical AND between two generic configurations.

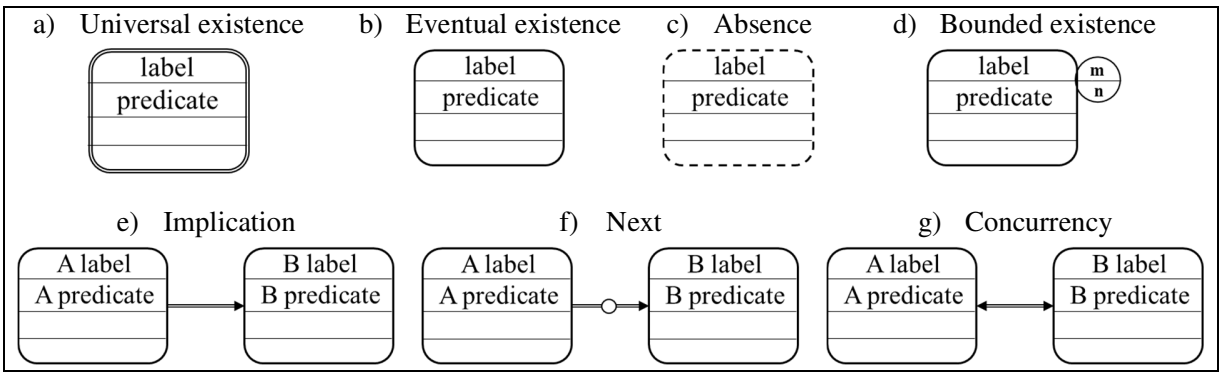



*Figure 10.* Building blocks for property patterns representation in HiLLS

### 5.1. Property Scope Notations

Table 1 presents the graphical notations introduced in HiLLS to support the property scopes defined in [21]. Since a temporal property specification is an abstract assertion on a segment of the execution of a system; we denote the entire execution by the elements between the initial configuration (solid ball) and final configuration (bull's eye) symbols. Each of the scopes describes the segment of the entire execution within which the specified property pattern (represented by dotted lines) must hold. Thus, to use any of the scope templates, we replace the dotted lines with the property pattern to be checked.

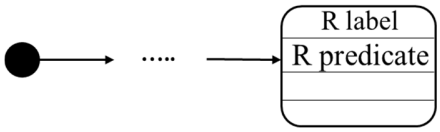
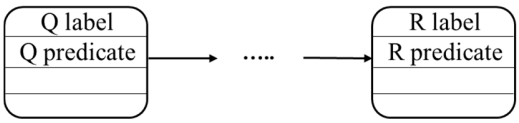
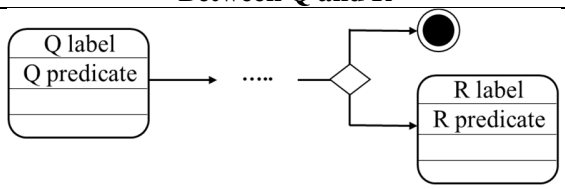
The scopes are:

- "Globally" scope, as the name implies, specifies that a property should hold throughout the execution of a system.

- "Before R" (respectively "After R") scope specifies that a given property must hold before (respectively after) the occurrence of a specified property R.
- "Between Q and R" implies that a given property must hold after the occurrence of Q and before R where it is certain that R will eventually occur.
- "After Q until R" has a similar implication with "Between Q and R" except that, in the former, it is not certain whether R will occur or not.

Note that the transitions between the generic configurations are abstract transitions without specific operations, triggers or output events. Hence, they do not specifically indicate any of the three kinds of configuration transition.

*Table 1.* Property scope notations in HiLLS

| Property scope notations | Descriptions |
|---|---|
| **Globally** | The property pattern (to replace the dotted lines) must be satisfied in every configuration throughout the execution between the initial and final configuration. |
|   **Before R** | The property pattern (to replace the dotted lines) must be satisfied *before* a transition into a configuration matching R. |
| **After R** | The property pattern (to replace the dotted lines) must be satisfied *after* a transition into a configuration matching R. |
|   **Between Q and R** | The property pattern (to replace the dotted lines) must be satisfied *after* a transition into a configuration matching Q and *before* a transition into a configuration matching R. |
|   **After Q until R** | The property pattern (to replace the dotted lines) must be satisfied *after* a transition into a configuration matching Q, and continue to hold until a configuration matching R occurs. If R does not occur, then the scope of the specified pattern continues until the end of execution. |

### 5.2. Property Pattern Notations

Dwyer et al. have classified property patterns into two categories: occurrence and order, to describe properties on the occurrences or non-occurrence of properties, and relative order of properties respectively within the segment of execution defined by the associated scopes.

Occurrence patterns include:

- Absence (which specifies properties that must never occur within the specified scope);
- Universality (which specifies properties that must continuously occur within the specified scope);
- Existence (which specifies properties that must occur at least once within the specified scope); and

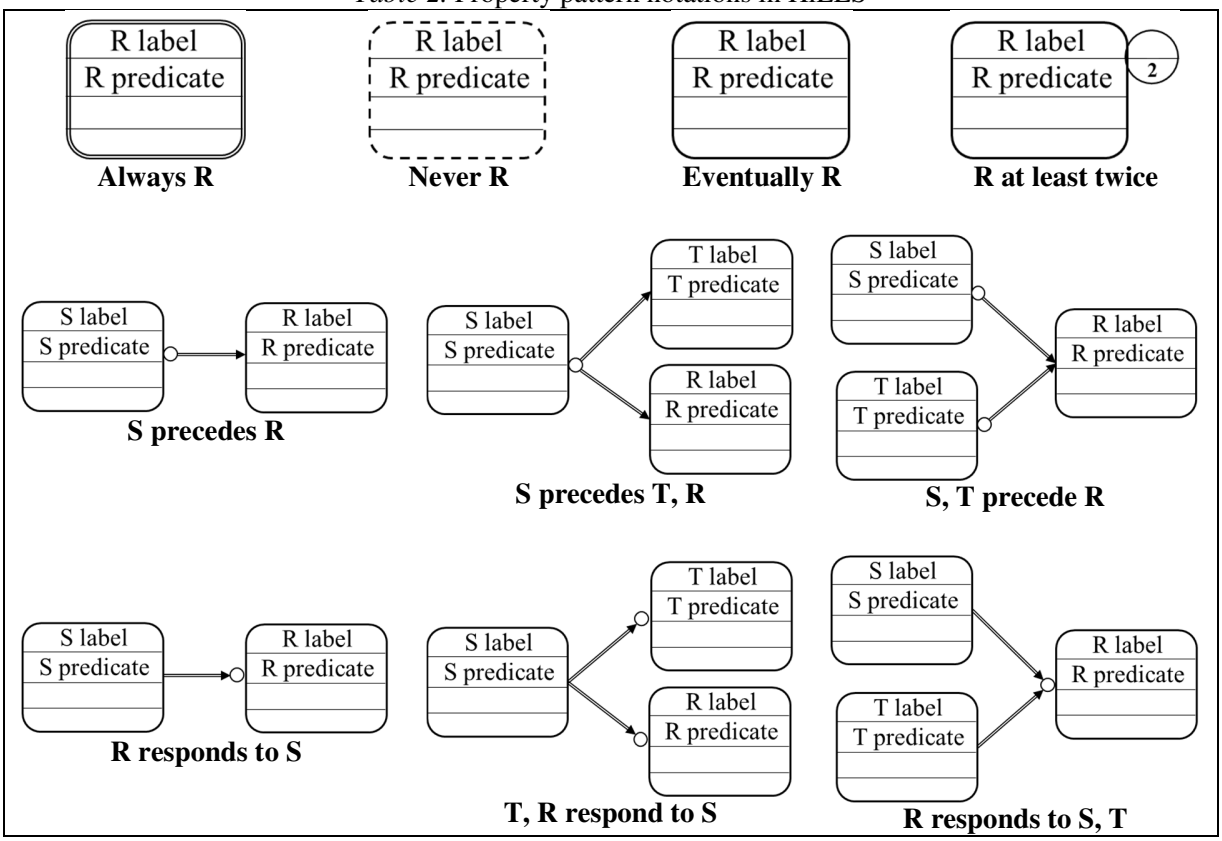
- Bounded existence (which specifies the maximum possible number of occurrences of certain properties within the specified scope).

Order patterns include:

- Precedence (which specifies a cause and effect relationship between two properties such that the occurrence of one must always have been preceded by the occurrence of the other within the specified scope);
- Response (which specifies a stimulus and response relationship between two properties such that the occurrence of one must always eventually be followed by the occurrence of the other within the specified scope);
- Chain precedence (which specifies a variant of the precedence pattern with m-cause to n-effect where m, n ∈N); and
- Chain response (which specifies a variant of the response pattern with m-stimulus to n-response where m, n ∈N).

Table 2 presents graphical notations introduced in HiLLS to support these patterns. The basic notations support directly the Absence, Universality, Existence and Bounded existence patterns. They are combined with additional symbols to support the remaining property patterns. Notice that the circle used in the precedence patterns can be seen as a mnemonic indication of the required property in an implication relationship (e.g., in S precedes R, S is required anytime R occurs, while in R responds to S, R is required anytime S occurs).

*Table 2.* Property pattern notations in HiLLS



### 5.3. Patterns and Scopes for Composed Models

In order to extend these notations to composed HiLLS models, we allow the requirements specification of such models to make reference to the generic configurations of their components by tagging them with the name of the corresponding components, as indicated by Figure 11. The relationships defined in Table 2 still hold for the elements of Figure 11. Consequently, requirements can be specified for HiLLS composed models, either by using generic configuration notations derived from its configuration transition diagram or by using the ones derived from the configuration transition diagrams of its component models.

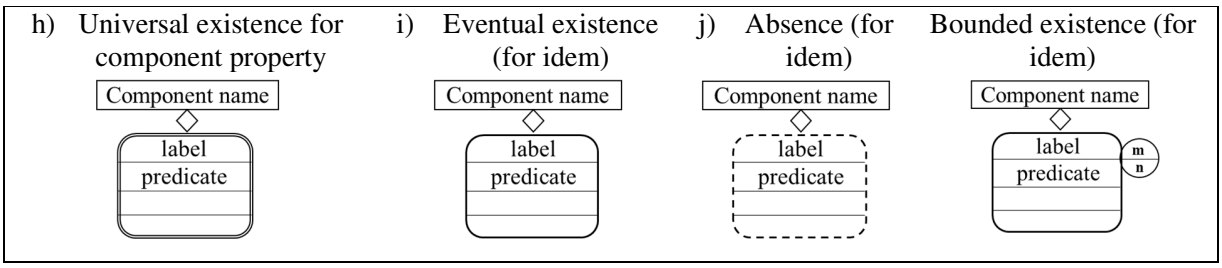



*Figure 11.* Generic configuration specification of a HiLLS composed model

### 5.4. Mapping to TCTL

The requirements specifications can be mapped onto TCTL (and many other temporal formalisms), such that once the user has a HiLLS-specified requirement model, the corresponding logic-based queries can be generated. There exist tools to automatically check such queries against the system model. We use the UPPAAL tool for that purpose [26-27]. Annex C gives for each of the property patterns, the corresponding TCTL/CTL specifications in the context of the scope patterns.

## 6. HILLS-Based Verification Framework

Systems properties can be expressed at different levels of abstraction. At higher levels, requirements resemble general principles and global expectations, for which there is no tool for automated verification. At the lower levels, requirements are detailed such that formal tools can be used, but this entails mathematical skills that are not commonly shared. We suggest a layered approach to bridge the gap between these extremes, as shown by Figure 12's right-hand side. In this organization, properties at a given level can be expressed in terms of properties at the immediate lower level. We consider three levels of abstraction:

(1) At the higher level are conceptual properties, such as fairness (also known as starvation-freedom), deadlock-freedom (also known as progress), termination, real-time correctness…
(2) The medium level is where the user needs to reduce a higher-level property to either a safety property, or a liveness property, or a combination of both properties. In a generic way, Safety specifies that "Something bad will never happen", while Liveness specifies that "Something good will eventually happen" [28]. For example, deadlock-freedom can be expressed either as repeated liveness [29] or safety [30]; termination can be expressed as liveness to some desired end [31-32], and fairness can be expressed either as repeated liveness or safety [18] – e.g., in the mutual exclusion property, having always at most one process in its critical section is a typical safety property, where the bad thing is that more than one process is in its critical section.
(3) The lowest level is where Safety and Liveness are expressed as Reachability properties. A reachability property states that some particular situations can be reached. If P is the something in Safety, then "Something bad will never happen" translates to "Never P". If Q is the something in Liveness, then "Something good will eventually happen" translates to "Eventually P".

Consequently, both are reachability problems. The user needs to place them within a given scope (e.g., the entire lifetime of the system, or a given lifetime window). The resulting requirement can be graphically captured using the notations we have introduced and automatically translated to logic queries that a formal tool can check.

Figure 12 depicts the full process of HiLLS-based visual modeling and the UPPAAL-based verification framework. On one side, HiLLS offers visual modeling means to capture the structure and behavior of the system. The patterns defined allow us to translate such a model into a UPPAAL TA. On the other side, the user can start with a high-level elicitation of requirements, then reduce it to an intermediate level of property analysis, and then a low level of reachability. Then the extended notations to HiLLS offer a visual means to capture the resulting requirements model. Rules defined in Annex translate such a model into TCTL statements, which can directly be expressed as UPPAAL queries. The UPPAAL tool allows to check the queries against the TA model.

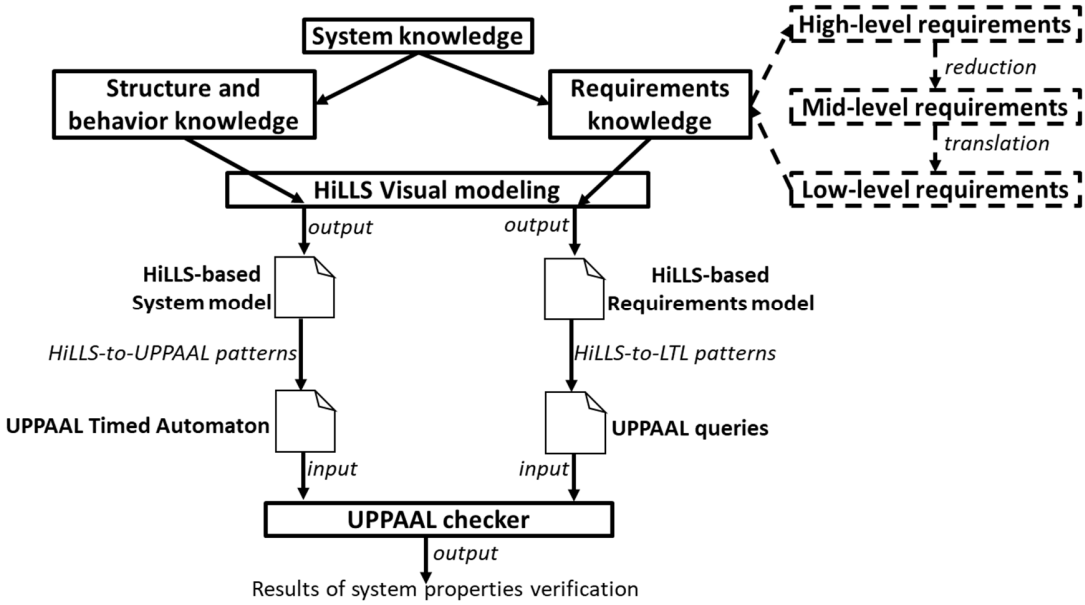



*Figure 12.* HiLLS-based formal verification framework

## 7. Application

To illustrate the application of our framework, let us capitalize on the well-known automated unmanned railway level crossing system described in Figure 13.

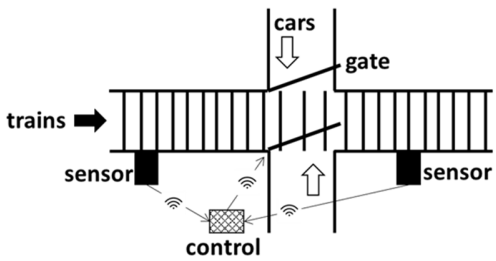



*Figure 13.* Automated unmanned railway level crossing system

The crossing is guarded by a gate, which is used to close the road when a train is crossing. Two sensors, one located upstream of the crossing, and the other downstream, are used to detect the arrival and exit of a train. The two sensors communicate the entrance and the exit of a train to the controller. The control system receives remote signals from the sensors, and remotely closes and opens the gate.

### 7.1. HiLLS-based System Modeling

Figure 14 shows the HiLLS model of the Crossing system as drawn in HGE. The entire system is a composition of 5 system components. These components are individually detailed in Annex D.

The entire system has a unique passive configuration labeled Network which depicts the relationships between components, and which translates the static nature of the composition. The predicate of the Network configuration provides the coupling information between the components, i.e., the Train's output port "Out" is connected to both the sensors' and the Controller's input ports "In", while the Controller's output port "Out" is connected to the Gate's input ports "In".

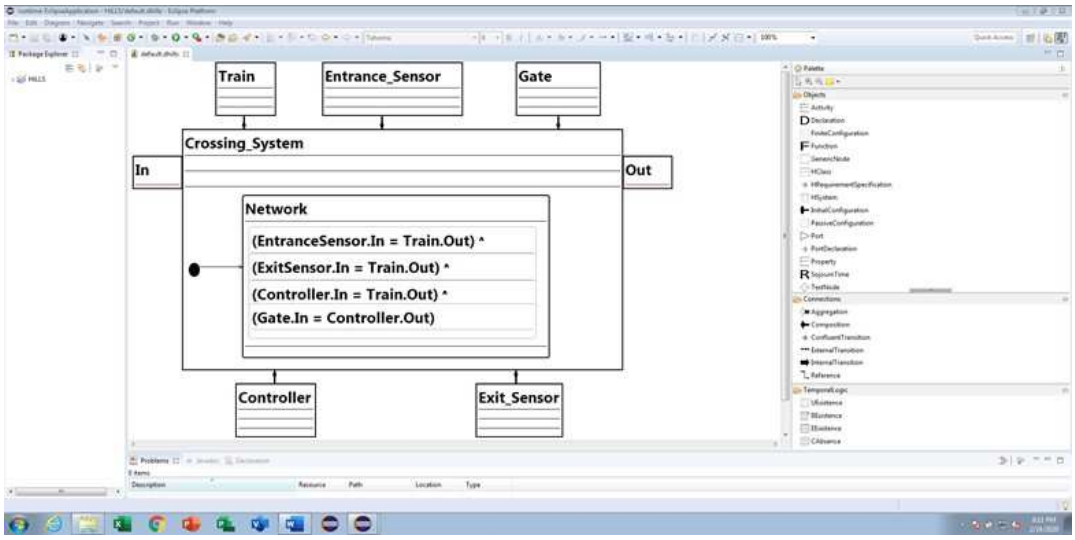



*Figure 14.* HiLLS model of the crossing system

Figure 15 presents the corresponding UPPAAL Timed Automata, using the semantic pattern defined in Figure 9. Semantic translations of all the HiLLS component models into corresponding UPPAAL automata using the configuration pattern variants previously defined are displayed in Annex E.

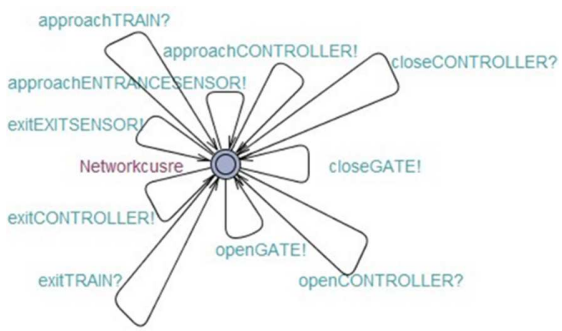



*Figure 15.* UPPAAL Timed Automata of the Crossing System

The composed system, while in its unique location $Network_{curse}$, receives signals from some components and translates them into signals to other components:

- If *approachTRAIN* is received from Train, then *approachENTRANCESENSOR* is sent to Entrance_Sensor and *approachCONTROLLER* is sent to Controller.
- If *openCONTROLLER* is received from Controller, then *openGATE* is sent to Gate.
- If *closeCONTROLLER* is received from Controller, then *closeGATE* is sent to Gate.
- If *exitTRAIN* is received from Train, then *exitEXITSENSOR* is sent to Exit_Sensor and *exitCONTROLLER* is sent to Controller.

### 7.2. HiLLS-based Requirements Specification

Our high-level requirements knowledge is that such a system has security concerns (among others), as regards to vehicles and pedestrians crossing the level by the road. We do not want users to be crushed by a high-speed train. We then reduce this security concern to "something bad will never happen" (mid-level Safety requirement), where something bad is "gate open while train crossing". The corresponding low-level requirements (i.e., scope and properties elucidated) are given by Figure 16.

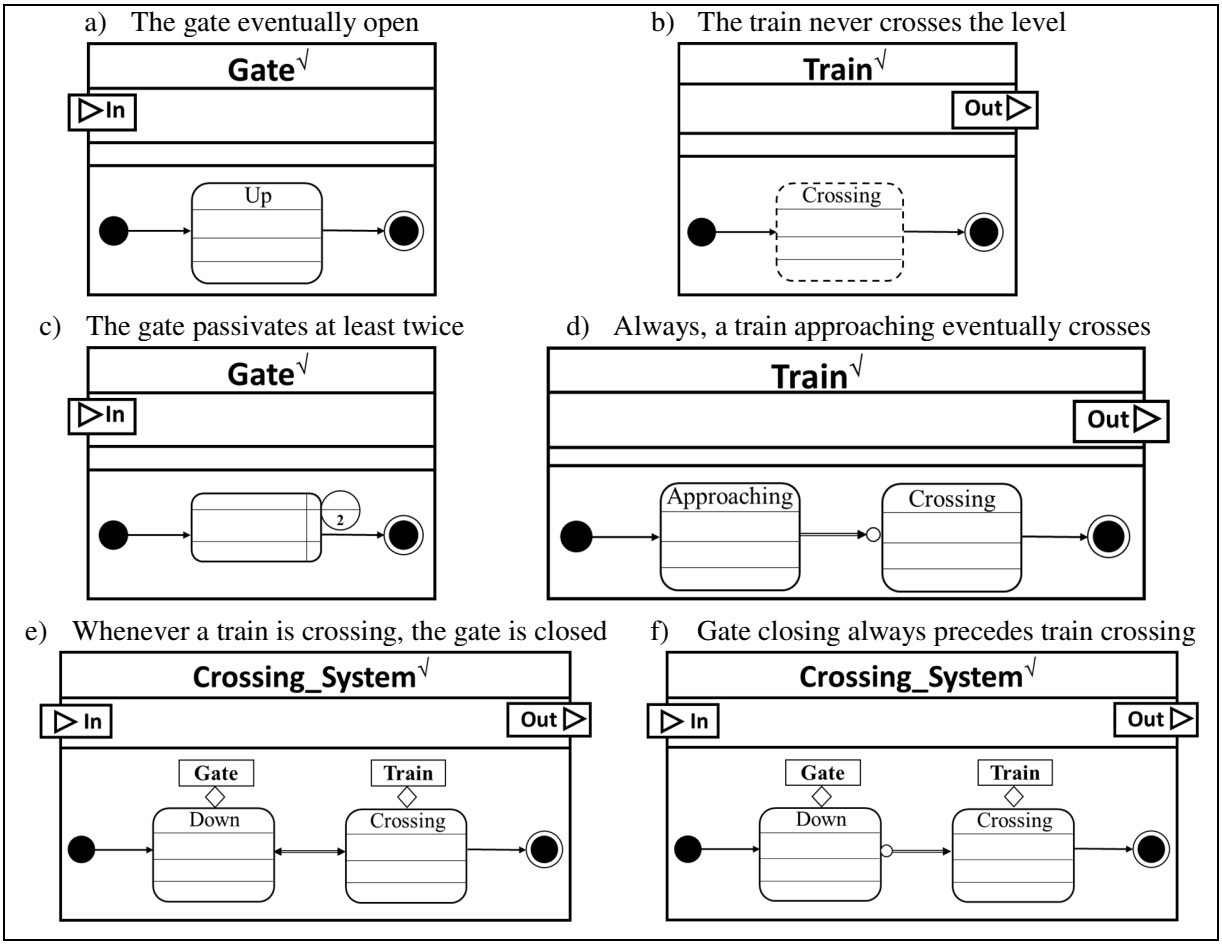



*Figure 16.* HiLLS-based requirements specifications for the crossing system

As shown by Figure 16, three cases can be considered:

(1) The requirement is expressed as a specific configuration of a model. While Figure 16.a requires that the gate must eventually open, Figure 16.b requires that the train must never cross the level.
(2) The requirement is expressed as a predicate that one or more configurations can satisfy. Figure 16.c requires that the gate must be in a passive configuration at least twice, while Figure 16.d requires that whenever a train approaches, it crosses eventually.
(3) The requirement is defined over multiple components of a composed model. Figure 16.e requires that always the gate must be closed when a train is crossing, while Figure 16.f requires that always the crossing of a train must be preceded by the closing of the gate.

### 7.3. HiLLS-based System Verification and Validation

Figure 17 shows how the requirement "Whenever a train is crossing, the gate is closed" translates to UPPAAL query, and the failure of the checking. This result reveals that the system is not safe since there exist cases where the train is crossing and the gate is open, which violates the safety property.

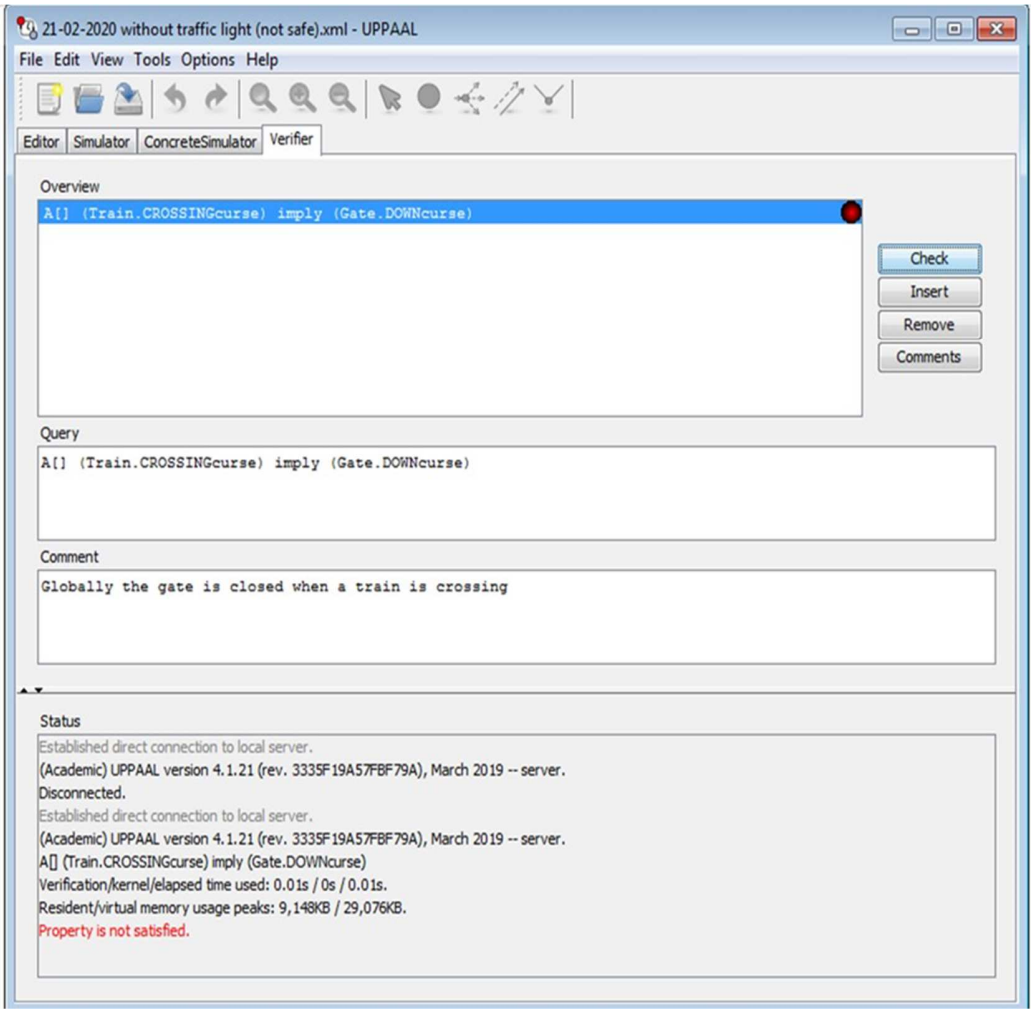



*Figure 17.* "Whenever a train is crossing, the gate is closed": Not satisfied

A known solution to this crossing system is to protect the crossing level by a traffic light that receives information to turn red or green from the controller. The controller also controls the opening and the closing of the gate. This solution has easily been modelled, verified and validated in the HiLLS framework.

## 8. Conclusion

This paper proposes a framework to support a full system verification process, using the High Level Language for Systems Specification (HiLLS) as a visual pivotal formalism. The process comprises the specification of both the discrete-event behavior of the system of interest and the temporal requirements to be checked against it. As HiLLS provides a graphical concrete syntax to the well-known DEVS formalism, the latter provides the semantic domain for the discrete-event simulation of HiLLS-specified models. To extend HiLLS capabilities to requirements specification, we adopted concepts from a pattern-based classification of Temporal Logic specifications of commonly occurring temporal requirements, for which we provided graphical notations using HiLLS syntactical elements. Well-defined transformation patterns allow to make HiLLS specifications amenable to UPPAAL checking, where at one side the system model is turned to a UPPAAL Timed Automaton and the requirements model is turned to UPPAAL queries.

We agree with the belief that the definition and use of high-level abstractions in writing formal specification is an important factor in making automated formal methods, specifically finite-state verification tools, more suitable. Therefore, providing graphical notations to describe systems and their requirements is a step towards bridging the gap between system experts and analysis experts. Moreover, having a highly communicable concrete syntax and multiple semantic domain mappings achieves the aim of providing a pivotal formalism for multiple analysis approaches, including the formal analysis of system properties without the need to run time-consuming experiments.

The proposed framework is unique in supporting a full system verification process based on the graphical modeling of both the system of interest and the requirements to be checked, using a drag and drop editor.

Much remains to be done for large-scale adoption, which we target as future work, including:

- The distribution of the HiLLS editor as an Eclipse plugin;
- The development of automated transformation engines for more temporal logic formalisms (other than TCTL), in order to allow the use of model checking tools other than UPPAAL; and
- The development of connectors from the HiLLS editor to existing DEVS tools (see http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm, last accessed on 28/02/2020).

## Reference


[1] Bryant, BR., Gray, J., Mernik, M., Clarke, PJ., France, RB., Karsal, G. Challenges and Directions in Formalizing the Semantics of Modeling Languages. *Computer Science and Information Systems* 2011; Vol.8, No.2: 225-253.

[2] Aliyu, HO., Maïga, O., and Traoré, MK. The High-Level Language for Systems Specification: A Model-Driven Approach to Systems Engineering. *International Journal of Modeling, Simulation, and Scientific Computing* 2016; Vol.7, No.1: 1641003.

[3] Zeigler, BP. *Theory of Modeling and Simulation*. 1st edition. New York: John Wiley & Sons. Inc., 1976.

[4] Aliyu, HO., Maïga, O., and Traoré, MK. A Framework for Discrete Event Systems Enactment. In: *Proceedings of the 29th European Simulation and Modeling Conference -ESM'15,* EUROSIS-ETI, Leicester, UK, 2015, pp.149–156.

[5] Samuel, KG., Maïga, O., and Traoré, MK. Formal Verification with HiLLS-Specified Models: A Further Step in Multi-Analysis Modeling of Complex Systems. *International Journal of Modeling, Simulation, and Scientific Computing* 2019; Vol.10, No.05: 1950032.

[6] Hong, KJ. and Kim, TG. DEVSpecL: DEVS Specification Language for Modeling, Simulation, and Analysis of Discrete Event Systems. *Information and Software Technology* 2006; Vol.48, No.4: 221–234.

[7] Büchi, JR. On a Decision Method in Restricted Second-Order Arithmetic. In: *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*, E. Nagel et al. (Eds.). Stanford University Press, Stanford, CA, 1962, pp.1–11.

[8] Zeigler, BP., Nutaro, JJ., and Seo C. Combining DEVS and Model-Checking: Concepts and Tools for Integrating Simulation and Analysis. *International Journal of Simulation and Process Modelling* 2017; Vol.12, No.1: 2–15.

[9] Zeigler, BP. and Sarjoughian, HS. *Guide to Modeling and Simulation of Systems of Systems*. Berlin, Germany: Springer Publication Company, 2012.

[10] Furfaro, A. and Nigro, L. A Development Methodology for Embedded Systems Based on RT-DEVS. *Innovations in Systems and Software Engineering* 2009; Vol.5, No.2: 117–127.

[11] Saadawi, H. and Wainer, G. Verification of Real-Time DEVS Models. In: *Proceedings of the 2009 Spring Simulation Multiconference*, Society for Computer Simulation International, San Diego, California, United State, 2009, pp.1–8.

[12] Madlener, F., Weingart, J. and Huss, SA. Verification of Dynamically Reconfigurable Embedded Systems by Model Transformation Rules. In: *Proceedings of IEEE/ACM/IFIP* 8th *International Conference on Hardware-Software Codesign and System Synthesis* (CODES+ISSS 2010) part of the Embedded Systems Week, Association for Computing Machinery, New York, NY, United States, 2010, pp.33–40. https://doi.org/10.1145/1878961.1878969

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

[14] Maler, O., Manna, Z., Pnueli, A. From timed to hybrid systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop* 1992. Springer-Verlag, London, 1992, pp.447–484.

[15] Nutaro, J. Toward a Theory of Superdense Time in Simulation Models. *ACM Transactions on Modeling and Computer Simulation* 2020; Vol.30, No.3: Article 16, 13 pages, DOI: https://doi.org/10.1145/3379489

[16] Jouault, F., Allilaire, F., Bezivin, J., Kurtev, I. ATL: a model transformation tool. *Science of Computer Programming,* 2008; Vol. 72, Issues 1–2: 31–39.

[17] Baier, C. and Katoen, JP. *Principles of Model Checking*. Cambridge, Massachusetts London, England: The MIT Press, 2008.

[18] Pnueli A. The Temporal Logic of Programs. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS'77),* IEEE Computer Society, 1730 Massachusetts Ave., NW Washington, DC, United States, 1977, pp.46–57. https://doi.org/10.1109/SFCS.1977.32

[19] Clarke, EM., Emerson, EA., and Sistla, AP. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems,* 1986; Vol.8, No.2: 244–263.

[20] Dwyer, MB., Avrunin, GS., and Corbett, JC. Patterns in Property Specifications for Finite-State Verification. In: *Proceedings of the 1999 International Conference on Software Engineering (ICSE-99),* IEEE, Los Angeles, 1999, pp.411–420.

[21] Ferro G. *AMC: ACTL Model Checker*. Reference Manual. Internal Report 1994, #B4-47.

[22] Kozen, D. Results on the Propositional μ-Calculus. *Theoretical Computer Science* 1983; Vol.27, No.3: 33–354.

[23] Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Wimmer, M., Vangheluwe, H. ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: *7th International Conference on Software Language Engineering (SLE),* Springer, Vasteras, Sweden, 2014, pp.1–20. *doi:10.1007/978-3-319-11245-9_1*

[24] Klein, F. and Giese, H. Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams. In: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering,* Springer Berlin Heidelberg, 2007, pp.185–199.

[25] Behrmann, David, Larsen, Pettersson, and Yi. Developing UPPAAL over 15 years', *Software Practice Experience* 2011; Vol.41, pp.133–142.

[26] Behrmann, G., David, A., Larsen, K., Hakansson, J., Petterson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: *Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST 2006),* IEEE Computer Society, Washington, DC, 2006, pp.125–126.

[27] Alpern, B. and Schneider, FB. Defining Liveness. *Information Processing Letters*, North-Holland 1985; Vol.21, No.4: 181–185.

[28] Callahan, J., Schneider, F., and Easterbrook, F. Automated Software Testing Using Model Checking. In: *Proceeding of the 1996 SPIN Workshop*, Citeseer, Rutgers University, New Brunswick, NJ, 1996, pp.118–127.

[29] Maandag, P. *Experiments in Unifying Model-Checking Approaches*. Master Thesis, Radboud University, Nijmegen, Netherlands, 2014.

[30] Yang, L. *Model Checking Concurrent and Real-Time Systems: The Pat Approach.* Ph.D. Thesis, National University of Singapore, Singapore, 2009.

[31] Akhtar, N. Requirements, Formal Verification and Model Transformations of an Agent-based System: A Case Study. *Computer Engineering and Intelligent Systems* 2014; Vol.5, No.3: 1–16.

**ANNEX A: DEVS formalism**

An atomic DEVS model is defined by the n-uple: $\langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta\rangle$, 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 possible states)
- $ta : S \rightarrow \Re_0^{+\infty}$ 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)
- $\lambda : S \rightarrow Y$ 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, 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 at exactly the time that the elapsed time in the system's current state $s_{curr}$ has reached $ta(s_{curr})$)

The operational semantics of an atomic DEVS model is informally described as follows: at the start, the systems are in an initial state and remain there until the time specified by ta is exhausted or until input event is received. In the former case, an internal transition function occurs then the system switches to another state after sending output event as defined by the output function $\lambda$. In the latter case if an input event is received before the specified time, then the external transition function is applied. When a collision occurs i.e., an external event is received concurrently with the elapsed time equal to the time specified by the time advance function, the confluent function is applied in such a way that the system sends output value and changes to a new state.

A coupled DEVS model is a structure: $\langle X_{self}, Y_{self}, \{M_d\}_{d\in D}, \{I_d\}_{d\in D}, \{Z_{i,j}\}_{i\in D\cup\{self\},j\in Ii}\rangle$, where

- $X_{self}$ and $Y_{self}$ are defined the same way X and Y are for atomic models (self being here a reference to the coupled model, while component models are referred to using indices such as i, j or d)
- D is the set of component references (thus, not including self)
- $M_d$ is the component model referenced by d, an atomic or a coupled model, with $X_d$ and $Y_d$ as respectively its input and output set
- $I_d$ is the influence set of component model d, i.e., all other models sending input to d
- $Z_{self,d\in Iself} : X_{self} \rightarrow X_d$ are the external input transfer functions, which determine how inputs received by self are translated into inputs to component models influenced by self
- $Z_{d/self\in Id,self} : Y_d \rightarrow Y_{self}$ are the external output transfer functions, which determine how outputs sent by component models influencing self are translated into outputs of self
- $Z_{i\in D,j\in D-\{i\}} : Y_i \rightarrow X_j$ are the internal transfer functions, which determine how outputs sent by component models are translated into inputs to component models they influence

**ANNEX B: ATL snippets of HiLLS to UPPAAL transformation**

```
rule HSystemPorts2UPPAALCha {
        from
                hillsPort : HiLLS!Port
        to
                uppaalcha: UPPAAL!Channels (
                        Inchannel <- hillsPort.portName.input,
                        Inchannel <- hillsPort.portType.portName.toString(),
                        Ochannel <- hillsPort.portName.output,
                        Ochannel <- hillsPort.portType.concat(uppaalcha)
)

rule HiLLSDec2UPPAALDec{
        from
                hillsDecl : HiLLS!Variable
        to --HiLLS Declaration -> UPPAAL Delaration
                uppaalDec :UPPAAL!Declaration(
                name <- hillsDecl.variableName,
                type <- hillsDecl.variableType
                )
}

rule HiLLSConfig2UPPAALLocation{
        from
         hillsConfig : HiLLS!Configuration
         to --HiLLS Configuration -> UPPAAL Location
         uppaalLoc : HiLLS!Configuration (
         stateID <- hillsConfig.label,
         property <- hillsConfig.properties,
         timeAdvance <- hillsConfig.sojournTime
         )
}

rule HSystemPorts2UPPAALCha {
        from
                hillsPort : HiLLS!Port
        to
                uppaalcha: UPPAAL!Channels (
                        Inchannel <- hillsPort.portName.input,
                        Inchannel <- hillsPort.portType.portName.toString(),
                        Ochannel <- hillsPort.portName.output,
                        Ochannel <- hillsPort.portType.concat(uppaalcha)
                        )
}

rule InitialConf2InitialLoc{
```

```
        from
                hillsIni : HiLLS!InitialConfiguration
        to
                uppaalIni : UPPAAL!InitialLocation (
                        name <- hillsIni.startingCong.label
                )
}

rule Configuration2Location {
        from
                hillsConf : HiLLS!Configuration
        to
                uppaalLoc : UPPAAL!Location (
                        name <- hillsConf.label,
                        invariant <- hillsConf.sojournTime,
                        guard <- hillsConf.properties,
                        update <- hillsConf.activities
                )
}
```

**ANNEX C: TCTL/CTL templates for occurrence property patterns**

Variables p, q, and r, are user-defined properties. The ◊, □, and ○ operators are respectively the eventually, always and next operators. The w operator is the weak until operator which may be related to the strong until operator (Y) using any of the following equivalences:

$p\mathrm{w}q = (\Box p) \vee (p\mathrm{Y}q)$

$p\mathrm{w}q = \Diamond(\neg p) \Rightarrow (p\mathrm{Y}q)$

$p\mathrm{w}q = p\mathrm{Y}(q \vee \Box p)$

In addition to the logical and temporal operators used in LTL, CTL supports the use of the *existential path quantifier* ∃ (resp. *universal path quantifier* ∀) for the specification of properties that must be satisfied by *some* (resp. *all*) computations starting in a state of interest. For example, ∀◊p requires that ◊p holds in all paths of executions starting from the state of interest, while ∃◊p requires that ◊p holds in at least one path of executions starting from the state of interest.

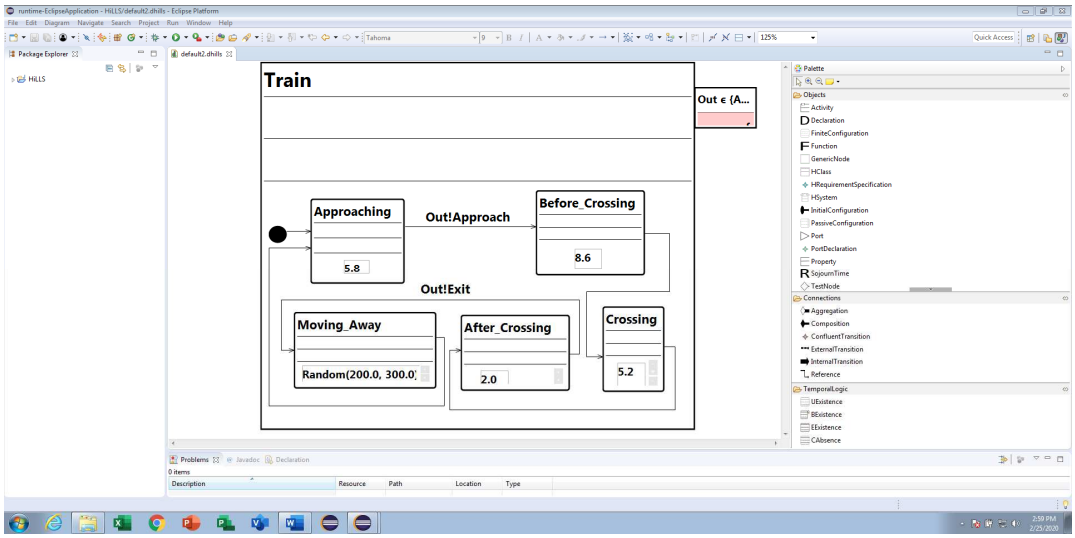
Readers may refer to [18] for more details on basic temporal operators, as well as a good introduction to LTL and CTL.

| **Absence** (*p* is false) | |
|---|---|
| Globally | ∀□(¬p) |
| Before *r* | ∀[(¬p∨ ∀□(¬r))w r] |
| After *q* | ∀□( q ⇒ ∀□ (¬p)) |
| Between *q* and *r* | ∀□(q∧¬r⇒ ∀ [(¬p∨ ∀□(¬r)) w r]) |
| After *q* until *r* | ∀□(q∧¬r⇒ ∀ [¬pw r]) |
| **Existence** (*p* becomes true) | |
| Globally | ∀◊p |
| Before *r* | ∀[¬rw ( p∧¬r)] |
| After *q* | ∀[¬qw ( q∧ ∀□( p))] |
| Between *q* and *r* | ∀□(q∧¬r⇒ ∀[¬rw ( p∧¬r)]) |
| After *q* until *r* | ∀□( q∧¬r⇒ ∀[¬r□ ( p∧¬r)]) |
| **Bounded Existence** (*p* occurs at most n times) | |
| Globally | ¬∃◊(¬p∧ ∃○( p∧ ∃◊(¬p∧ ∃○( p∧ ∃◊(¬p∧ ∃○( p))))) |
| Before *r* | ¬∃[¬r□ (¬p∧¬r∧ ∃○( p∧□ [¬r□ (¬p∧¬r∧ ∃○( p∧ ∃[¬r□ (¬p∧¬r∧ ∃○( p∧¬r))]))]))] |
| After *q* | ¬∃[¬q□ ( q∧ ∃◊(¬p∧ ∃○( p∧ ∃◊(¬ p∧ ∃○( p∧ ∃◊(¬p∧ ∃○( p))))))] |
| Between *q* and *r* | ∀□( q⇒¬∃[¬r□ (¬p∧¬r∧□○( p∧ ∃[¬r□ (¬p∧¬r∧ ∃○( p∧ ∃[¬r□ (¬ p∧¬r∧ ∃○( p∧¬r∧ ∃◊(r)))]))]))]) |
| After *q* until *r* | ∀□( q⇒¬∃[¬r□ (¬p∧¬r∧ ∃○( p□∃[¬r□ (¬p∧¬r∧ ∃○( p∧ ∃[¬r□ (¬ p∧¬r∧ ∃○( p∧¬r))]))]))] |
| **Universality** (*p* is true) | |
| Globally | ∀□( p) |
| Before *r* | ∀[( p∨ ∀□(¬r)) wr] |
| After *q* | ∀□( q⇒ ∀□( p)) |
| Between *q* and *r* | ∀□( q∧¬r⇒ ∀[( p∨ ∀□(¬r)) wr]) |
| After *q* until *r* | ∀□( q∧¬r⇒ ∀[ pwr]) |
| **Precedence** (*s* precedes *p*) | |
| Globally | ∀[¬pws] |
| Before *r* | ∀[(¬p∨ ∀□(¬r)) w (s∨r)] |
| After *q* | ∀[¬qw (q∧ ∀[¬pws])] |

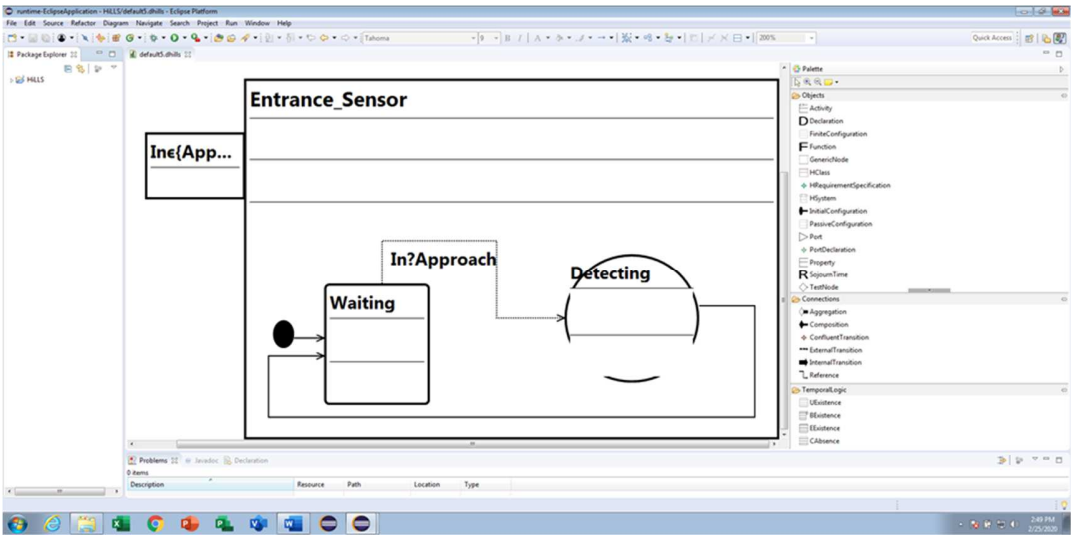
| | |
|---|---|
| Between *q* and *r* | $\forall\Box(q\wedge\neg r\Rightarrow \forall[(\neg p\vee \forall\Box(\neg r))\ w\ (s\vee r)])$ |
| After *q* until *r* | $\forall\Box(q\wedge\neg r\Rightarrow \forall[\neg p w\ (s\vee r)])$ |
| **Response** (*s* responds to *p*) | |
| Globally | $\forall\Box(p\Rightarrow \forall\Diamond(s))$ |
| Before *r* | $\forall[((p\Rightarrow \forall[\neg r\Box(s\wedge \neg r)])\vee \forall\Box(\neg r))wr]$ |
| After *q* | $\forall[\neg q w\ (q\wedge \forall\Box(p\Rightarrow \forall\Diamond(s))]$ |
| Between *q* and *r* | $\forall\Box(q\wedge \neg r\Rightarrow \forall[((p\Rightarrow \forall[\neg r\Box(s\wedge \neg r)])\vee \forall\Box(\neg r))\ wr])$ |
| After *q* until *r* | $\forall\Box(q\wedge \neg r\Rightarrow \forall[(p\Rightarrow \forall[\neg r\Box(s\wedge \neg r)])wr])$ |
| **Precedence chain** (*p* precedes *s*, *t*) | |
| Globally | $\neg\exists[\neg p U\ (s\wedge\neg p\wedge \exists\bigcirc(\exists\Diamond(t)))]$ |
| Before *r* | $\neg\exists[(\neg p\wedge \neg r)\ U\ (s\wedge \neg p\wedge \neg r\wedge \exists\bigcirc(\exists[\neg r U\ (t\wedge\neg r)]))]$ |
| After *q* | $\neg\exists[\neg q U\ (q\wedge \exists[\neg p U\ (s\wedge \neg p\wedge \exists\bigcirc(\exists\Diamond(t)))])]$ |
| Between *q* and *r* | $\forall\Box(q\Rightarrow \neg\exists[(\neg p\wedge \neg r)\ U\ (s\wedge \neg p\wedge \neg r\wedge \exists\bigcirc(\exists[\neg r U\ (t\wedge \neg r\wedge \exists\Diamond(r))]))])$ |
| After *q* until *r* | $\forall\Box(q\Rightarrow \neg\exists[(\neg p\wedge \neg r)\ U(s\wedge \neg p\wedge\neg r\wedge \exists\bigcirc(\exists[\neg r U\ (t\wedge \neg r)]))])$ |
| **Precedence chain** (*s*, *t* precedes *p*) | |
| Globally | $\neg\exists[\neg s U p]\wedge \neg\exists[\neg p U\ (s\wedge \neg p\wedge \exists\bigcirc(\exists[\neg t U\ (p\wedge \neg t)]))]$ |
| Before *r* | $\neg\exists[(\neg s\wedge\neg r)\ U\ (p\wedge \neg r)]\wedge\neg\exists[(\neg p\wedge \neg r)\ U\ (s\wedge \neg p\wedge \neg r\wedge \exists\bigcirc(\exists[(\neg t\wedge \neg r)\ U\ (p\wedge\neg t\wedge \neg r)]))]$ |
| After *q* | $\neg\exists[\neg q U(q\wedge \exists[\neg s U p]\wedge \exists[\neg p U(s\wedge \neg p\wedge \exists\bigcirc(\exists[\neg t U\ (p\wedge \neg t)]))])]$ |
| Between *q* and *r* | $\forall\Box(q\Rightarrow \neg\exists[(\neg s\wedge \neg r)U(p\wedge\neg r\wedge \exists\Diamond(r))]\wedge\neg\exists[(\neg p\wedge\neg r)U(s\wedge\neg p\wedge \neg r\wedge \exists\bigcirc(\exists[(\neg t\wedge\neg r)U(p\wedge\neg t\wedge \neg r\wedge \exists\Diamond(r))]))])$ |
| After *q* until *r* | $\forall\Box(q\Rightarrow \neg\exists[(\neg s\wedge \neg r)U(p\wedge \neg r)]\wedge \neg\exists[(\neg p\wedge \neg r)\ U\ (s\wedge \neg p\wedge \neg r\wedge \exists\bigcirc(\exists[(\neg t\wedge\neg r)\ U\ (p\wedge\neg t\wedge \neg r)]))])$ |
| **Response chain** (*s*, *t* respond to *p*) | |
| Globally | $\forall\Box(p\Rightarrow \forall\Diamond(s\wedge \forall\bigcirc(\forall\Diamond(t))))$ |
| Before *r* | $\neg\exists[\neg r U\ (p\wedge \neg r\wedge (\exists[\neg s U\ r]\vee\exists[\neg r U\ (s\wedge \neg r\wedge \exists\bigcirc(\exists[\neg t U r]))]))]$ |
| After *q* | $\neg\exists[\neg q U\ (q\wedge \exists\Diamond(p\wedge(\exists\Box(\neg s)\vee \exists\Diamond(s\wedge \exists\bigcirc(\exists\Box(\neg t))))))]$ |
| Between *q* and *r* | $\forall\Box(q\Rightarrow \neg\exists[\neg r U\ (p\wedge \neg r\wedge (\exists[\neg s U r]\vee \exists[\neg r U\ (s\wedge\neg r\wedge \exists\bigcirc(\exists[\neg t U r]))]))])$ |
| After *q* until *r* | $\forall\Box(q\Rightarrow \neg\exists[\neg r U\ (p\wedge\neg r\wedge(\exists[\neg s U r]\vee \exists\Box(\neg s\wedge\neg r)\vee \exists[\neg r U\ (s\wedge\neg r\wedge \exists\bigcirc(\exists[\neg t U r]\vee \exists\Box(\neg t\wedge \neg r)))]))])$ |
| **Response chain** (*p* responds to *s*, *t*) | |
| Globally | $\neg\exists\Diamond(s\wedge \exists\bigcirc(\exists\Diamond(t\wedge \exists\Box(\neg p))))$ |
| Before *r* | $\neg\exists[\neg r U\ (s\wedge\neg r\wedge \exists\bigcirc(\exists[\neg r U(t\wedge \neg r\wedge \exists[\neg p U r])]))]$ |
| After *q* | $\neg\exists[\neg q U(q\wedge \exists\Diamond(s\wedge \exists\bigcirc(\exists\Diamond(t\wedge\exists\Box(\neg p)))))]$ |
| Between *q* and *r* | $\forall\Box(q\Rightarrow \neg\exists[\neg r U\ (s\wedge\neg r\wedge \exists\bigcirc(\exists[\neg r U\ (t\wedge\neg r\wedge \exists[\neg p U r])]))])$ |
| After *q* until *r* | $\forall\Box(q\Rightarrow \neg\exists[\neg r U\ (s\wedge \neg r\wedge\exists\bigcirc(\exists[\neg r U\ (t\wedge\neg r\wedge (\exists[\neg p U r]\vee\exists\Box(\neg p\wedge\neg r)))]))])$ |

**ANNEX D: HiLLS model components for the crossing system study case**

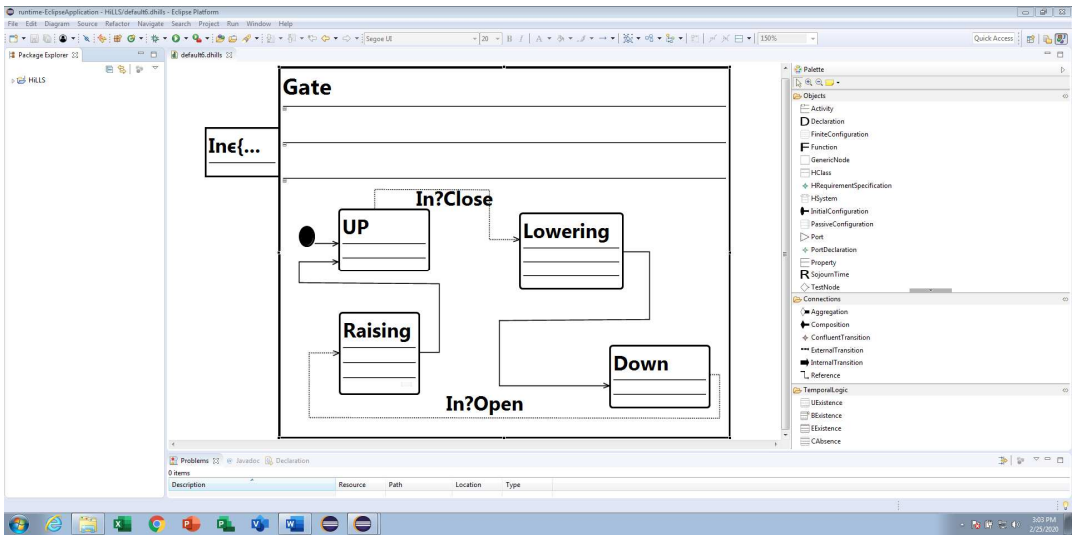
The Train has five configurations. The initial one is Approaching and is finite. As such an internal transition takes place after 5.8 seconds, and takes the Train to Before_Crossing while outputting the Approach signal. The Train stays there for 8.6 seconds as this is the travel time from being detected 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 transits the Train to Moving_Away where the Train spends reasonably longer time (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.



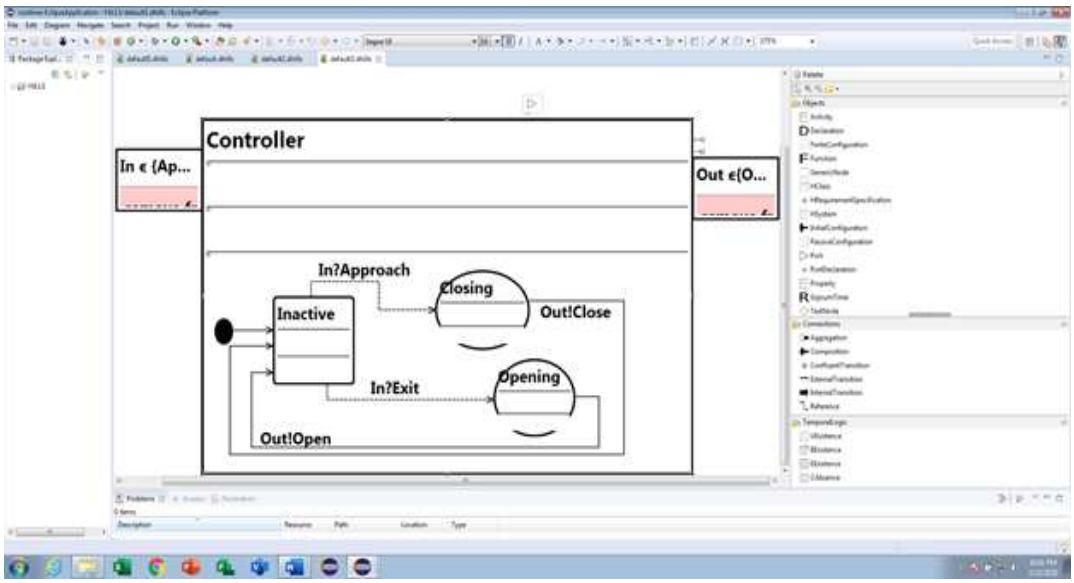

The sensor models (i.e., Entrance_Sensor and Exit_Sensor) function the same way, and have, each, two configurations: Waiting and Detecting. Initially, the Sensor is Waiting, and it keeps doing so until it detects the signal Approach. It will then take an external transition to a transient configuration Detecting where it spends no time. Therefore, an internal transition immediately takes place from Detecting back to Waiting.



The Gate is initially open (configuration Up) and it remains so until it receives an input signal Close, which will cause an external transition to Lowering, a finite configuration with sojourn time of 2.3 seconds. From there, 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.

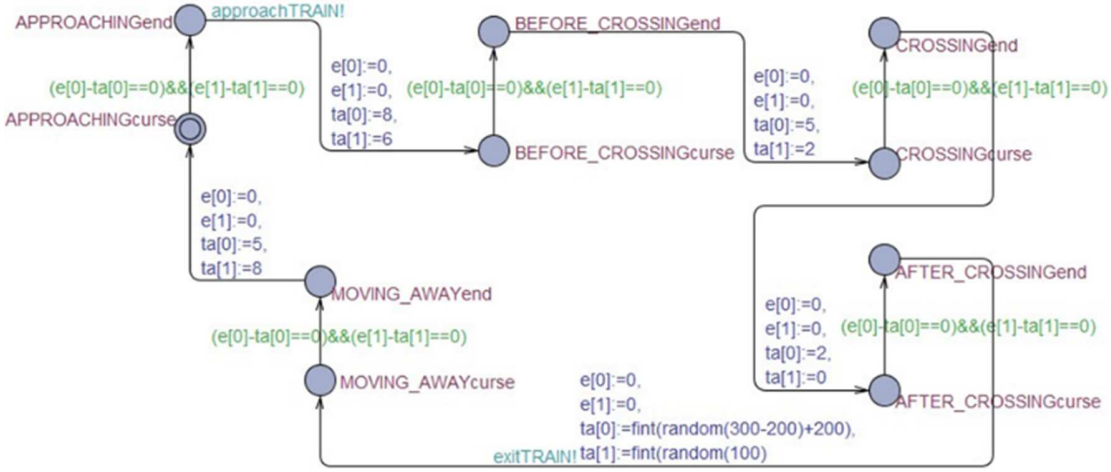



The Controller model has 3 configurations: Inactive (initial configuration), Closing and Opening. The Controller remains in a passive configuration until it receives a signal. Once the signal is received, the Controller takes an external transition to Closing (if the signal received is Approach), or 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.

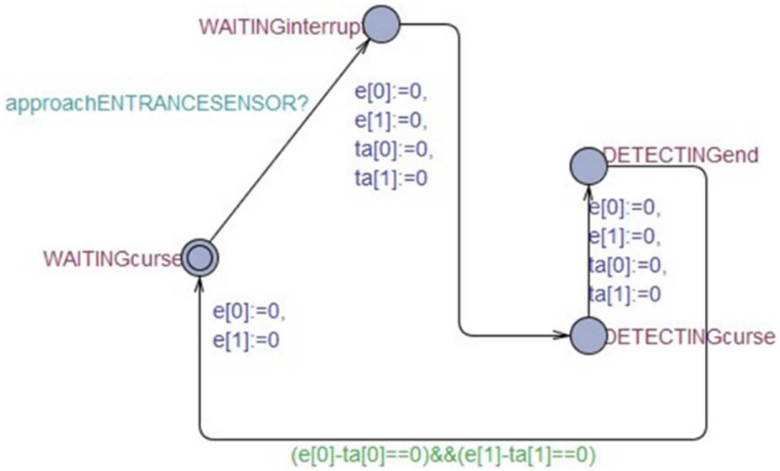


## ANNEX E: UPPAAL Timed Automata of the crossing system's components

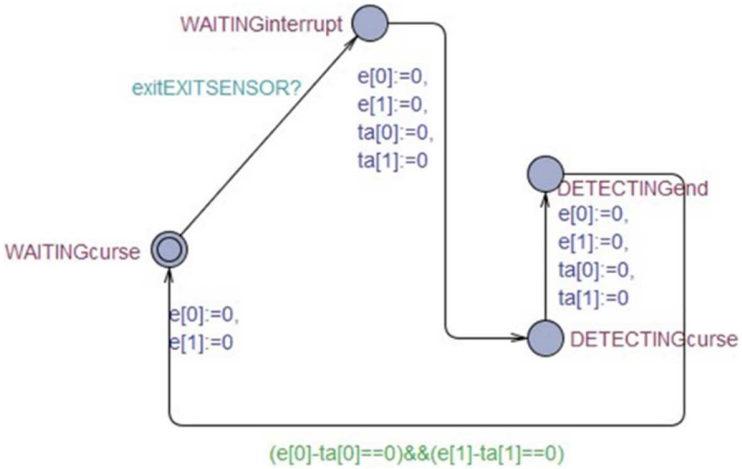
### Train TA



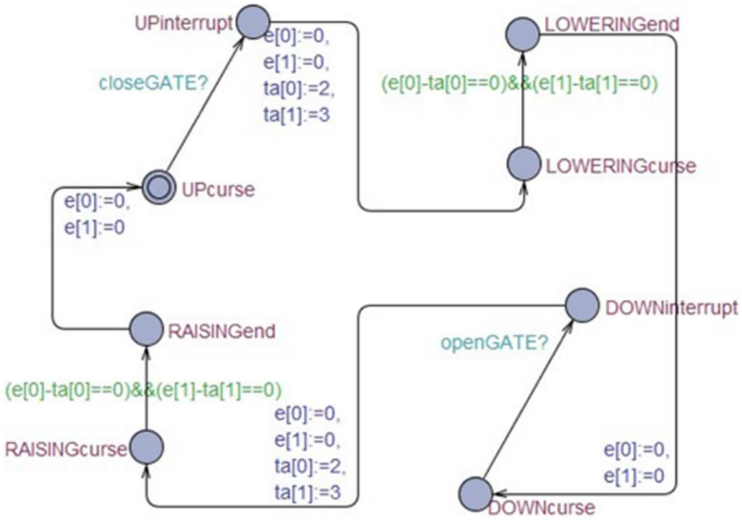

### Entrance sensor TA

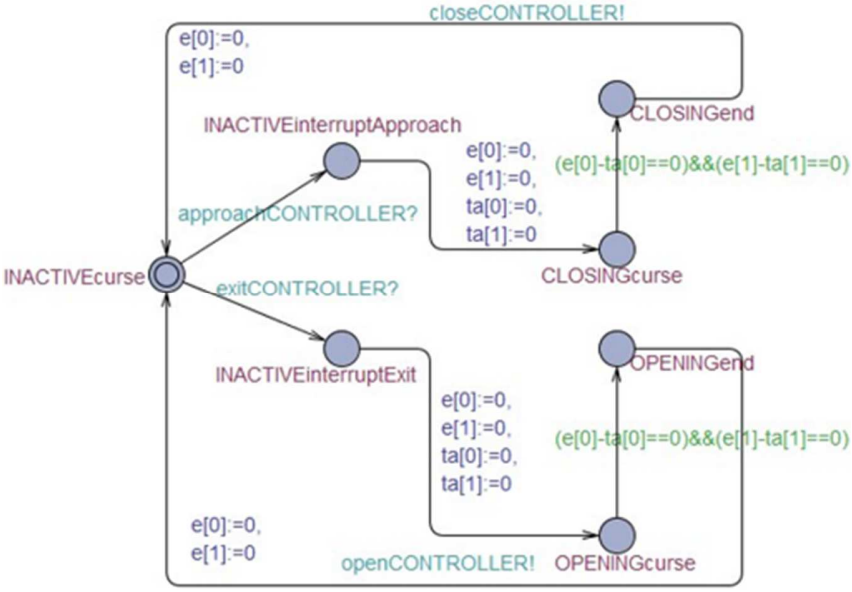



### Exit sensor TA



**Gate TA**




**Controller TA**