Software and Systems Modeling (2021) 20:1825–1855
https://doi.org/10.1007/s10270-021-00923-9

SPECIAL SECTION PAPER

Unified verification and monitoring of executable UML specifications

A transformation-free approach

Valentin Besnard[1] · Ciprian Teodorov[2] · Frédéric Jouault[1] · Matthias Brun[1] · Philippe Dhaussy[2]

Received: 10 May 2020 / Revised: 16 August 2021 / Accepted: 24 August 2021 / Published online: 21 November 2021
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2021

Abstract
The increasing complexity of embedded systems renders software verification more complex, requiring monitoring and formal techniques, like model-checking. However, to use such techniques, system engineers usually need formal expertise to express the software requirements in a formal language. To facilitate the use of model-checking tools by system engineers, our approach uses a UML model interpreter through which the software requirements can directly be expressed in UML as well. Formal requirements are encoded as UML state machines with the transition guards written in a specific observation language, which expresses predicates on the execution of the system model. Each such executable UML specification can model either a Büchi automaton or an observer automaton, and is synchronously composed with the system, to follow its execution during model-checking. Formal verification can continue at runtime for all deterministic observer automata used during offline verification by deploying them on real embedded systems. Our approach has been evaluated on multiple case studies and is illustrated, in this paper, through the user interface model of a cruise-control system. The automata-based verification results are in line with the verification of the equivalent LTL properties. The runtime overhead during monitoring is proportional to the number of monitors.

Keywords Model-checking · Monitoring · Model interpretation · Embedded systems · Observation Language · Synchronous Composition

CR Subject Classification Model-driven software engineering · Software verification · Model checking · Interpreters · Embedded software

Communicated by Tao Yue, Man Zhang, and Silvia Abrahao.

✉ Valentin Besnard
valentin.besnard@eseo.fr

Ciprian Teodorov
ciprian.teodorov@ensta-bretagne.fr

Frédéric Jouault
frederic.jouault@eseo.fr

Matthias Brun
matthias.brun@eseo.fr

Philippe Dhaussy
philippe.dhaussy@ensta-bretagne.fr

[1] ERIS, ESEO-TECH, Angers, France

[2] Lab-STICC UMR CNRS 6285 ENSTA Bretagne, Brest, France

1 Introduction

In the context of embedded cyber-physical systems, the software design becomes increasingly more complex. This exposes software programs to several potential failures (e.g., design faults, bugs, security flaws) that are more and more intricate to detect, understand, and fix. With model-driven engineering, software systems can be designed using models and verified with formal verification techniques (e.g., model-checking) during early design phases. These techniques give promising results (e.g., verification of Mars exploration rovers and of medical device transmission protocols with the Spin model-checker [31]), but require abstractions of the system environment, which might miss some real execution cases and do not consider failures due to deficient hardware components. For these reasons, embedded systems

increasingly rely on monitoring as a way to continue formal verification during runtime execution.

To perform all these verification activities, system requirements must be expressed in a formal language (e.g., linear temporal logic (LTL)). For offline verification, these properties are typically transformed into Büchi automata and then composed with the system model to verify its correctness by model-checking. However, the transformation of properties from temporal logic into Büchi automata is usually hidden to users, notably because the resulting automata are expressed in a specific formalism (e.g., the Hanoi Omega-Automata (HOA)[1] format), which can be hard to master for engineers without formal background. For online verification, one technique [5,27] aims at synthesizing monitors from formal safety properties to take advantage of the complementarity between model verification and runtime monitoring. However, a transformation is usually required to specialize the synthesized monitor on embedded targets, thus creating a semantic gap between the monitor model and the monitor code. Not only the executable code corresponding to the monitors has to be generated, but instrumentation code is also needed to expose system objects and link monitors with them. Moreover, an equivalence relation has to be built and proved to ensure that the generated code conforms to the LTL properties (or the equivalent monitors) used during model verification.

Through these observations, we notice that at least two issues remain. First, monitors designed or synthesized for model verification cannot be reused directly to monitor system execution. They require a transformation for code instrumentation. Second, languages used for formal properties specification and system modeling are usually different. This makes the use of formal verification techniques a complex task for system engineers who do not have a formal background. Specifically, expressing properties in a formal language as well as understanding verification results may be difficult for them without the help of formal methods experts.

This study presents the first UML verification tool that uses UML for both property specification and system modeling, and that bridges the gap between offline verification and runtime monitoring through a unified transformation-free approach. This work extends our embedded model interpreter (EMI) introduced in [6–8]. EMI is a tool that can be used to execute, simulate, and verify embedded systems, specified as UML [49] models, with the same implementation of the UML semantics for all of these activities. Prior work on this model interpreter showed how to perform simulation, trace generation, and LTL model-checking activities, but did not bridge the aforementioned gaps. The present work addresses this shortcoming by focusing on a unified way to verify and monitor formal properties. For this purpose, we introduce the concept of *Property UML State Machines*, called PUSMs in the rest of this paper, to express system requirements in UML. In our previous conference paper [10], we show how PUSMs can encode the system requirements as **deterministic observer automata** to model-check, and monitor safety properties. This journal paper is an extension that adds the UML support to encode and model-check more complex properties with (i) **Büchi automata** and (ii) **non-deterministic observer automata** expressed as PUSMs. These PUSMs are interpreted with the same UML semantics as the one used to execute the system model. Each PUSM is then synchronously composed with the system execution so that it can observe it and detect the failures as soon as they occur. The resulting execution component can be used either for offline verification by steering execution with a model-checking algorithm, or for runtime monitoring by deploying it on an embedded target. As a result, this tool aims at validating system design specifications, i.e., high-level representation of the intended final software system. By continuing verification online, our tool enables to validate the system behavior coupled with the actual embedded target. This makes sense as the system is usually tightly linked to the execution platform characteristics.

The main contributions of this work are the following: (1) The introduction of an observation language, which by exposing the system objects, enables the link between the UML system and the properties to verify on it; (2) the extension of executable UML models with this observation language to obtain the PUSMs used for expressing and verifying temporal properties; (3) a transformation from LTL to PUSM, which offers the possibility of using PUSMs as a backend for LTL model-checking; (4) the formalization of the concepts needed to link the language semantics, the properties semantics, and verification tools; (5) the use of deterministic UML observer automata, deployed unchanged on real embedded targets, for runtime monitoring; (6) the use of the observation language for expressing conditional breakpoints during preliminary debug phases. Furthermore, to the best of our knowledge, our approach is the first to support multiverse debugging [58] on a practical language.

To validate our approach, several experiments have been carried on different case studies including a level-crossing system [8], a soccer player robot [9], a cardiac defibrillator[2] from [23], and a landing gear system from [13]. One of them, a UML model of a cruise-control user interface is used as illustration in this paper. For all these case studies, the system requirements can be modeled as PUSMs, executed using the EMI UML semantics, and synchronously composed with the system execution. The PUSM-based

[1] HOA format: http://adl.github.io/hoaf/.

[2] Implementation of a cardiac defibrillator: https://github.com/Pyponou/defibrillator.

verification results obtained with OBP2[3] [8,14], an explicit-state on-the-fly model-checker, are identical to the results of model-checking equivalent LTL properties. While Büchi and non-deterministic observer automata can only be used for offline verification, all deterministic observer automata have also been deployed on embedded targets (e.g., STM32 discovery boards) to monitor the system execution.

The remainder of this paper is structured as follows. Section 2 describes the cruise-control interface used as a running example. An overview of the approach is given in Sect. 3. Then, Sect. 4 describes how to observe and control the model execution on EMI, while Sect. 5 explains the process used to express formal properties as PUSMs. The synchronous composition of these automata with the system model is presented in Sect. 6, while in Sect. 7, we detail the process used to perform offline verification and runtime monitoring. In Sect. 8, we present the results of applying the approach to our example and we discuss some points of our work in Sect. 9. Section 10 reviews the state of the art, and we conclude this paper in Sect. 11 emphasizing some future research directions. Some appendices are also available. "Appendix A" provides a more detailed description of the motivating example. "Appendix B" describes more deeply our action and observation languages, while "Appendix C" gives the complete list of atomic propositions used in this paper.

2 Motivating example

To illustrate our approach, we consider the user interface of a cruise-control system (CCS). The CCS automatically controls the speed of a vehicle by adjusting the throttle position to maintain a steady speed as set by the driver. This motivating example has been designed for the purpose of this paper and is partially based on [19,41] as well as past experiences of some of the authors on similar systems.

To better understand how work a CCS (in which a CCI is integrated), Fig. 1 presents a component diagram showing its interactions with the driver and the physical vehicle. For this example, we focus our design and verification efforts on the user interface, which we call cruise-control interface (CCI), because this subsystem contains most of the control logic of the CCS. Hence, all components external to the CCI are considered as the environment of this subsystem. In "Appendix A", we describe the considerations that have helped us to have a better understanding of this environment and make a relevant abstraction of it for the verification step.

The behavior of the system, designed in UML, can be summarized as follows. The driver interacts with the *CruiseControlInterface* (CCI) through different *Buttons* (i.e., start, stop, inc, dec, set, pause, resume) and three pedals (i.e., *ClutchPedal*, *BrakePedal*, *Throttle pedal*). These interactions result in the sending of events to the CCI. Inside the CCI subsystem, these events are received by the controller either directly or through a pedal manager that makes a preprocessing of all pedals events. Using these events, the controller delegates the computation of the cruise-speed to a cruise-speed manager and the activation/deactivation of the *ControlLoop* to another object called actuation. The *ControlLoop* is considered here as a black box that executes a control algorithm for computing the command to apply on the *PhysicalVehicle* engine. Finally, the *PhysicalEnvironment* may apply some forces on the vehicle (e.g., road profile, air friction) that disrupt its actual speed value.

To use the CCI and more globally the *CruiseControlSystem* (CCS), in which it is integrated, the driver has first to start the CCI. At this time, the CCS is turned on but still disengaged (i.e., the CCS does not act on the engine). To engage the CCS, the driver need to set the cruise-speed at the vehicle speed to activate the *ControlLoop* acting on the engine. The driver can then trigger the different behaviors of the system including the change of the cruise speed. Please note that a more complete description of this design model is given in "Appendix A".

This paper is focused on the verification and the monitoring of formal properties. For this purpose we have selected six system requirements of the CCI, which along with the CCI model will serve as a basis for better understanding our contribution. These requirements (the last three are taken from [19,41]) have been picked out for their representative and illustrative potential in the context of this paper. The chosen requirements are the following:

- (R1) When a "stop" event has been received, the CCS will finally be disengaged.
- (R2) When a "set" event has been received and if the system is disengaged, the CCS will finally be engaged.
- (R3) When a "pedalReleased" event has been received from the throttle pedal and if the control loop can be resumed, the "pedalReleased" event is not consumed until a "resume" event is sent to the controller.
- (R4) After the detection of an event that turns the control loop off and until a contrary event is sent, the CCI should not try to send new setpoints.
- (R5) The cruise speed should not be below 40 km/h or above 180 km/h.
- (R6) When the system is engaged, the cruise speed should be defined.

Figure 2 gives two examples of specialized UML state-machines, which we call PUSMs capturing two of the requirements: one representing a non-deterministic Büchi automaton encoding R2 in Fig. 2a and one representing a deterministic observer automaton encoding R4 in Fig. 2b.

[3] OBP2: https://www.obpcdl.org.

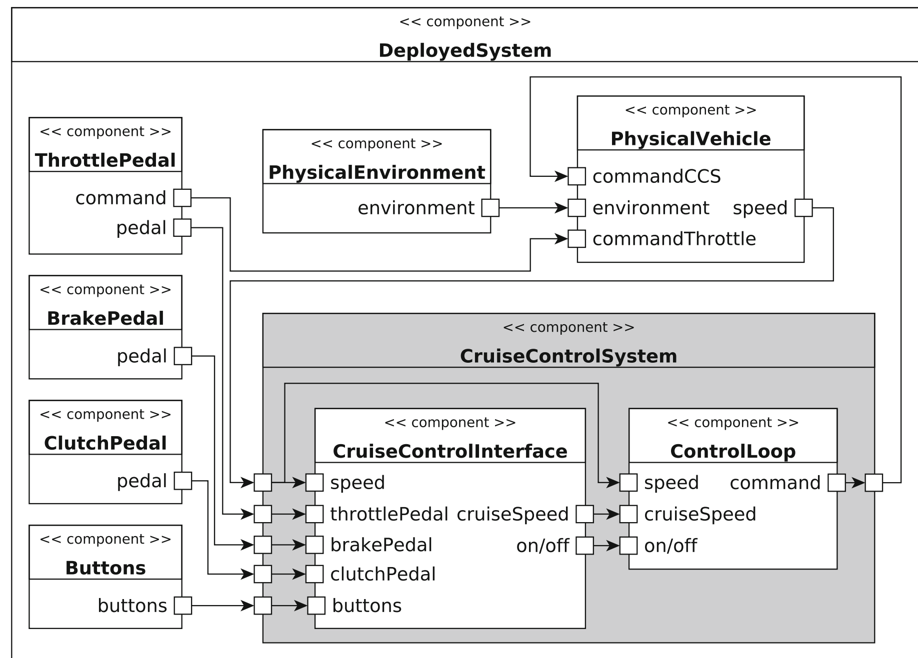
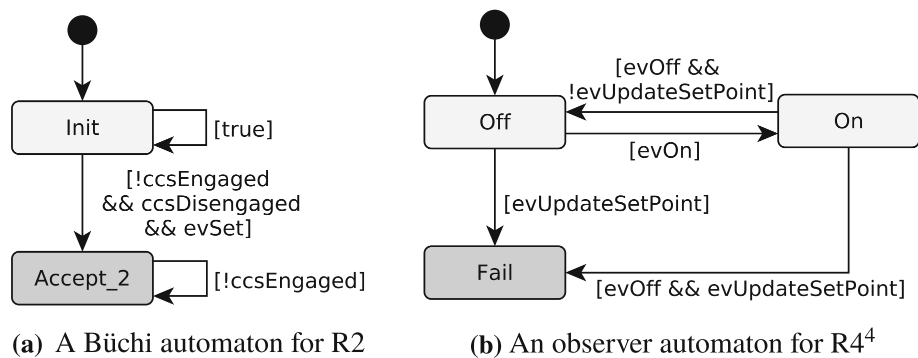


Fig. 1 Component diagram of a cruise control system



(a) A Büchi automaton for R2 **(b)** An observer automaton for R4[4]

Fig. 2 State machines of PUSMs for the CCI

The former can be used only for offline verification, while the latter can also be used for online verification by being deployed on the actual embedded platform. Despite their syntactic resemblance, these two automata are semantically very different. On the one hand, as shown in [3], the non-deterministic Büchi automata (encode ω-regular languages, which can be seen as a generalization of the regular languages to include infinite words) are commonly used in model-checking to check temporal safety and liveness properties through a reduction to the language inclusion problem. On the other hand, the more restricted observer automata can be defined as special processes that monitor changes in the *state* of a model (e.g., attribute values, contents of event pools, current state of state machines) and the *events* occurring in it (e.g., signal events) [45]. Composed synchronously with the monitored system, the observers can be used to verify safety properties. By further constraining them to be deterministic, the same observers (used during the verification process) can be deployed as runtime monitors.

The purpose of this paper is to convince the reader that, by **avoiding the use of model transformations**, it is possible to unify the verification and the runtime monitoring of UML models while minimizing the gap between the specification (UML-based PUSM in our case) and the design language (a bare-metal executable subset of UML).[4]

3 Approach overview

To better understand the scope of this work, this section gives an overview of our approach. It describes the integration of a verification and monitoring infrastructure with EMI, our UML model interpreter [8]. A main contribution of this work is to show that properties can be encoded as a UML model and then used for both offline verification and runtime monitoring without the cost of proven model transformations.

[4] In comparison with [10], this observer has been refined to be independent of the vehicle engine implementation.

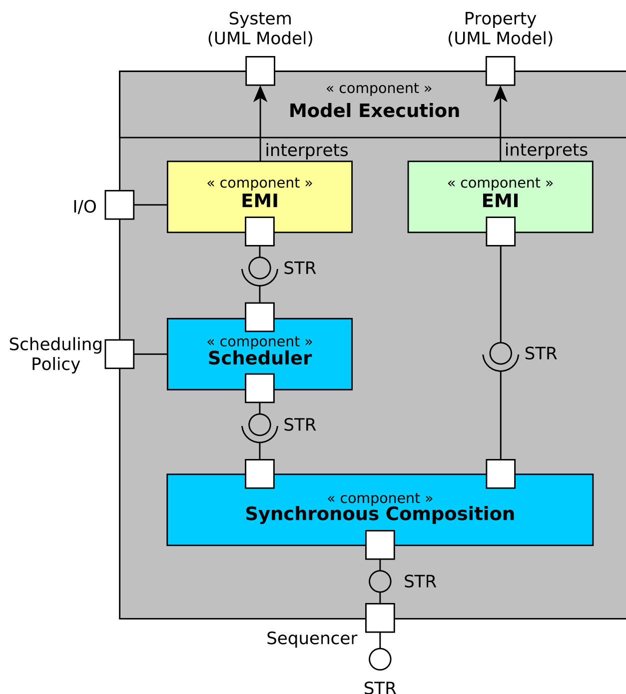


Fig. 3 Model execution component overview

An overview of the *Model Execution* component used for simulation, offline verification, and runtime monitoring is illustrated in Fig. 3. This component takes as input both the UML models of the *System* and of the *Property*, which contains a PUSM. The system model is designed in UML from its specification, while the property model is specified from the system requirements. Formal properties can be specified directly in the design language, here as PUSMs, or firstly in a temporal logic formalism (e.g., LTL) before being automatically transformed into UML. PUSMs can access the state of the system model using the observation language of our UML model interpreter, presented in Sect. 4.2.

Once the *System* and the *Property* have been designed in UML, each model is interpreted by an EMI instance. Thus, our UML interpreter is instantiated twice, such that the same UML semantics implementation is used for both parts. Then, a *Synchronous Composition* operator produces the synchronous product of the property model execution with the system model execution. This operator, external to the UML semantics, is central to our contribution. This composition is made such that each time a transition is fired in the system model, a transition is also fired on the PUSM of the property model. For this purpose, a set of Boolean predicates, called atomic propositions, is evaluated on the system model by the system interpreter. The resulting valuations give a Kripke view [40] on the system execution to the property model, which can be used to decide which transition has to be composed with the system transition. Using this mechanism, the property model execution is closely following the system execution without the need of any model transformation.

This synchronous composition operator is also able to consider the system execution scheduling because the system semantics (that result of the composition of active objects in the system model) is usually not deterministic. The *Scheduler* is responsible for selecting which transition of the system will be fired on the next step. This component can be configured with a *Scheduling Policy* that specifies how the choice is made.

Finally, a *Sequencer* is used to control the execution of the product automaton, resulting from the *Synchronous composition*, through our Semantic Transition Relation (STR) interface. The *Sequencer* can be of different natures depending on which activity the *Model Execution* component is used for. (i) To perform *interactive simulation*, it may be the user through a graphical user interface. (ii) For *offline verification*, a *Model-checking Algorithm* adapted to the appropriate (Büchi or observer automata) formalism is used to control *Model Execution* as shown on Fig. 4a. In both cases (simulation and model-checking), the system model is closed with an *Abstract Environment* model used to interact with the system. This abstraction is designed as another UML model connected to the I/O port. (iii) For *runtime monitoring*, the sequencer is the algorithm in charge of running the main *Execution Loop* of the execution platform. In this case, the system model is connected to its *Real Environment* through actual I/O of the embedded execution platform. An additional component called *Assertion Acceptance* checks if a failure has been detected by PUSMs and outputs the *Monitoring Status*.

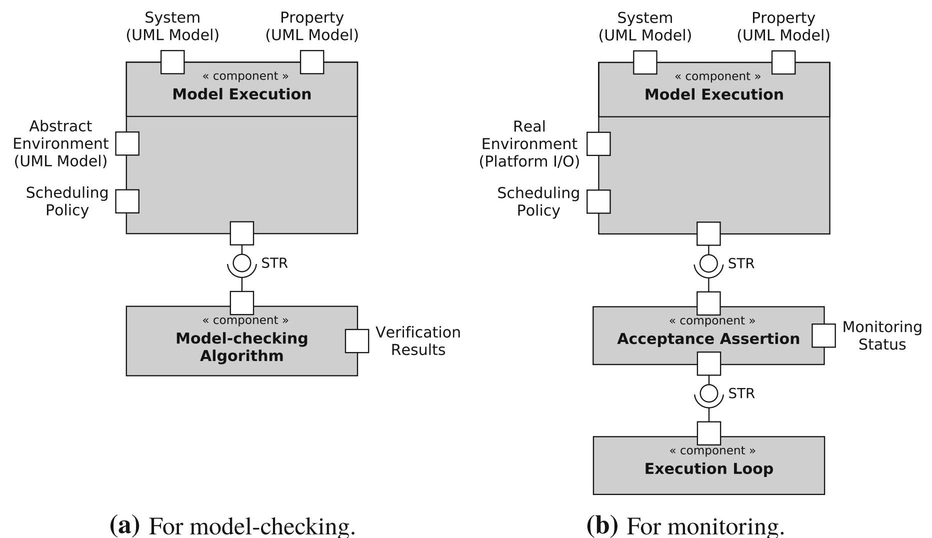
4 Controlling and observing model execution

A key point in the application of the synchronous composition lies in facilities offered by EMI to control and observe model execution. This section introduces the communication interface of EMI, as well as both the action language used in this work for system modeling, and the observation language required, in property models, to speak about the system execution.

4.1 Steering execution of UML models

To control the model execution, our approach uses a Semantic Transition Relation (STR) interface, which is presented here with a formal syntax strongly inspired by Lean[5] [18]. This formal syntax removes ambiguities due to the notation and provides a machine-checkable version of our mechanisms

[5] Lean Prover: https://leanprover.github.io/.



(a) For model-checking. (b) For monitoring.

Fig. 4 Model-checking and monitoring architecture overview

as pseudo-code (see our GitHub repository[6] for the fully Lean-compliant version). The Semantic Transition Relation abstracts over the system and property semantics, enabling a language independent formalization of our approach. In practice the real semantics is bound to this abstraction through a thin adaptation layer. Given the following types:

- `C`: the type of configurations. A configuration is a memory dump of all runtime data handled by an execution engine (i.e., its execution state) at a given time.
- `A`: the type of actions. An action is an abstract representation of fireable transitions or execution steps.

An execution engine can be controlled using the STR interface which provides the three following functions:

```
structure STR (C, A : Type) :=
    (initial : set C)
    (actions : C → set A)
    (execute : C → A → set C)
```

The **`initial`** function returns the possible initial configurations (`set C`) of the model. The **`actions`** function is used to get all actions (`set A`) that are enabled in a given source configuration (`C`). Note that this function exposes the scheduling non-determinism (if each action returned is seen as belonging to a ready process). The **`execute`** function executes one action (`A`) in a source configuration (`C`) and returns the possible target configurations (`set C`) that can be reached. While typical execution functions are deterministic, the STR abstraction allows for non-deterministic execution, which is sometimes necessary for high-level specification languages. For instance, some non-determinism appears when an attribute takes a value in an interval (e.g., for an integer x, $x = [0; 2]$ leads to three different configurations: $x = 1$, $x = 2$, and $x = 3$). Note, however, that this is not the case in the context of this paper since the UML run-to-completion step semantics is deterministic. While UML has language non-determinism because multiple transitions can be fireable at the same time, it does not have execution non-determinism. Indeed, with the same execution context, the execution of a given UML run-to-completion step always leads to the same target configuration.

Our model interpreter also has its own communication interface, and the goal is now to implement the STR interface using this communication interface. For this purpose, it is important to know that EMI keeps in memory both the current `model` being executed as well as a `dynamic` memory area where the current configuration is stored as a byte array. As a state-monad, all functions of the EMI communication interface can also access this state or modify it by side-effect. A simple definition of the `EMI` type and of the `EMIState` monad is given here. In practice, our C implementation of the `EMI` structure is more complex, but the definition presented here offers a suitable abstraction for the purpose of this paper. In particular, `UMLModel`, `EMIDynamicMemory`, and `EMITransition` are complex data structures that are informally described and only considered as abstract data types in this paper.

```
def UMLModel := the UML model to
   interpret
def EMIDynamicMemory := the current
   configuration of EMI
def EMITransition := internal
   representation of an EMI action

structure EMI : Type :=
    (model : UMLModel)
```

[6] GitHub repository: https://github.com/ValentinBesnard/emi-verifying-and-monitoring-uml-models.

```
    (dynamic : EMIDynamicMemory)

def EMIState := state EMI
```

Our model interpreter has different functions in its communication interface to: reset the model execution, get and set the current configuration, collect the set of fireable transitions, and fire a transition. They are based on `EMIState`, the internal state of the interpreter, and `EMITransition`, the internal representation of UML run-to-completion steps. The full implementation of all these functions cannot be presented here because it depends on the UML semantics, which is quite complex. However, signatures of these functions can be defined in the following way:

```
def reset : EMIState unit
def get_configuration : EMIState
   EMIDynamicMemory
def set_configuration (c :
   EMIDynamicMemory) : EMIState unit
def get_fireable_transitions : EMIState
    (set EMITransition)
def fire (t : EMITransition) : EMIState
    unit
```

Given these definitions, it is now possible to express the `STR` interface in terms of the `EMI` interface. For this purpose, we define three adapter functions, which rely on the `EMIState` monad, to present the appropriate interface conversion.

```
def emi_initial :
  EMIState EMIDynamicMemory :=
  do reset, c ← get_configuration, return c

def emi_actions (c : EMIDynamicMemory) :
  EMIState (set EMITransition) :=
  do set_configuration c,
  a ← get_fireable_transitions, return a

def emi_execute (c : EMIDynamicMemory) (t :
   EMITransition) :
  EMIState EMIDynamicMemory :=
  do set_configuration c, fire t,
  x ← get_configuration, return x

def EMI_to_STR
  (emi : EMI)
: @STR EMIDynamicMemory EMITransition := (
initial ← { prod.fst (emi_initial.run emi) },
actions ← λ c, prod.fst ((emi_actions c).run emi
   ),
execute ← λ c a, { prod.fst ((emi_execute c a).
   run emi) })
```

As a result, `EMI_to_STR` gives the possibility to steer the execution of a UML model running on EMI with the STR interface. In the general case, both **`initial`** and **`execute`** functions of STR can return several possible configurations (e.g., with generic specification languages). However, in our case, `EMI_to_STR` highlights the fact that only one configuration is returned (the initial one or the one obtained after executing an action).

4.2 Action language and observation language for UML models

This section presents the action language used for system modeling, and the observation language used in PUSMs to speak about model execution. The foundational subset of UML, called fUML [48], already has a standardized action language called Alf [46]. Even if Alf can be used with state machines, thanks to the Precise Semantics of UML State Machines (PSSM) standard [47], its roots are deeply linked with UML activities. Unlike Alf, the goal of our action language is to define a minimal set of concepts that can be used with UML state machines without the need to support UML activities. Moreover, the Alf reference implementation is in Java, whereas we use an embedded model interpreter implemented in C. This makes the reuse of the Alf reference implementation difficult. For all these reasons, we choose to define our own action language to better fit our needs. Additionally, this action language offers the possibility of being easily extended with additional operators to define the observation language needed for PUSMs.

Figure 5 gives an overview of both the action language and the observation language used in this work. These languages may read and/or write runtime data used during model execution. We distinguish two kinds of runtime data: (i) data linked to explicit model attributes (e.g., instance variables) and (ii) data internally defined in the execution engine to implement the language semantics. Using this classification, language operators can be divided into four groups: expression language, effect language, expression language extension, and effect language extension.

4.2.1 Action language

The action language is the language used to detail the fine-grained behavior of the system in UML models. This language is only composed of the expression language and of the effect language used respectively to read and write explicit model attributes. It can be used in UML models to specify guards and effects of state machines transitions. For this purpose, it offers read/write access to model attributes. In practice, the action language should not give access to internal runtime data for two reasons: (i) to prevent interfering

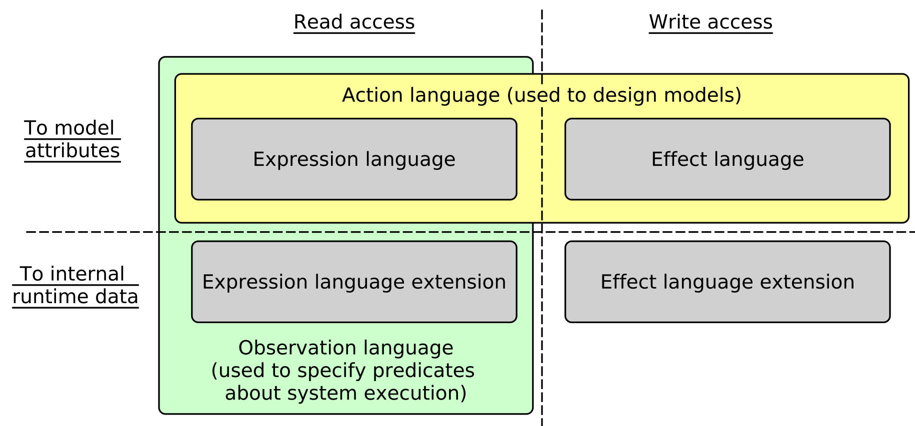


Fig. 5 Overview of the action language and the observation language

with the execution engine and (ii) to keep the UML model independent of the execution platform.

4.2.2 Observation language

The observation language is composed of the expression language and its extension such that it can have read access to both model attributes and internal runtime data to express properties. In the same way as the expression language extension, it may be possible to define an effect language extension to write internal runtime data. This is outside the scope of this paper, but this possibility, that goes beyond observation, may be useful for some analysis activities like debugging.

The observation language is used to express (i) guards of PUSMs and (ii) atomic propositions of LTL properties. These atomic propositions are composed with temporal logic operators to form LTL properties. Given that formal properties are expressed about dynamic execution paths of model execution, they are verified for a given UML model executed on a given execution engine. This language provides some introspection capabilities required to specify properties that cannot be expressed in relation to observable facts in the environment of the verified system. In addition to providing access to runtime data, this expression language extension also provides more relaxed navigation rules and facilities for model verification. As the observation language is based on UML concepts, it can help engineers to specify or understand the atomic propositions used in the formal properties.

As mentioned before, both the action language and the observation language defined in this work are based on the programming language C extended with a set of operators defined as C macros. Contrary to conventional programming languages, the particularity of these languages is that they manipulate external data (i.e., some data declared and defined in UML models). To get references on these external data, the parsing phase is entirely separated from the symbol resolution phase. Each identifier of the parsing phase is associated with the appropriate symbol using lookup tables. Each symbol can be an attribute or a method of an object, or a global symbol of the model. In practice, we rely on facilities offered by the C compiler (especially the C preprocessor) to perform all these lookup operations statically (without a runtime cost).

All operators provided by the action language and the observation language in addition to C language constructs are detailed in "Appendix B". Among them, the `GET` operator is used to navigate the model (e.g., to get a model attribute). `ROOT_instMain` gives access to the instance of the `Main` composite structure from which all system objects can be accessed. `IS_IN_STATE` checks if the current state of an active object is a given state of its state machine.

As an example, all atomic propositions of the six properties of the CCI motivating example have been expressed with this observation language (see "Appendix C" for all atomic proposition definitions). For the sake of simplicity, these propositions have been defined as predicates with labels. The `ccsEngaged` predicate that "checks if the current state of the *actuation* state machine is *Engaged*" is shown here for illustration purposes:

```
ccsEngaged = "IS_IN_STATE(GET(GET(ROOT_instMain, cci),
              actuation), STATE_Actuation_Engaged)"
```

5 Expressing properties as UML models

This section presents how formal properties can be expressed in UML by mapping Büchi automata and observer automata to UML state machines. The specificities of each formalism regarding UML modeling are discussed. Moreover, a tool used to convert automatically LTL properties into PUSMs is also introduced.

5.1 Modeling properties in UML

In this work, formal properties are expressed directly in UML using the observation language and the same UML subset as the one used for system modeling. In the design phase,

each property is encoded as a PUSM. More concretely, a PUSM is an instance of an active UML class whose behavior is described with a UML state machine. All PUSMs are instantiated as parts of a composite class called *Prop* and its instance specification *instProp*. The *Prop* class is used as a root composite structure for PUSMs instantiation.

PUSMs are intentionally designed to represent either Büchi or observer automata. According to the literature [3], we can give a common definition for both kinds of automata. An automaton is defined by a tuple $A = (\Sigma, Q, \delta, q_0, F)$ where:

- Σ is a finite set of labels called the alphabet of A,
- Q is a finite set of states of A,
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation of A,
- $q_0 \in Q$ is the initial state of A,
- $F \subseteq Q$ is a finite set of acceptance states of A.

The main difference between Büchi automata and observer automata concerns their acceptance condition. For Büchi automata (operating on infinite traces), a word (or a trace) is accepted if an acceptance state is reached *infinitely often*, while for observer automata (operating on finite traces), a word is accepted if an acceptance state is reached *once*.

Given this formal background, we can define a mapping from these automata to PUSMs. In PUSMs, labels of the alphabet are Boolean predicates expressed with the observation language. These predicates, also called atomic propositions, are used to define guard constraints of PUSMs transitions. These guards are used to specify how the corresponding state machine goes from one state to another and potentially reaches acceptance states. Each state (respectively, transition) of the Büchi or observer automaton is modeled as a UML state (respectively UML transition) of the PUSM. The initial state is preserved and easily identified in UML by an incoming transition coming from the initial pseudostate of the state machine.

The goal of these automata is to detect failures in the system behavior. For this purpose, UML state machines used for PUSMs need to define acceptance states. The acceptance states are specified with global state invariants in our approach. For instance, the state invariant to verify the property encoded by the observer automaton of Fig. 2b is:

“`IS_IN_STATE(GET(ROOT_instProp, observer4), STATE_Observer4_Fail)`”.

The use of state invariants generalizes more language-specific approaches such as the use of UML profiles with stereotypes for the acceptance states.

As an additional constraint, PUSMs used to model these automata do not interact with objects of the system model, (e.g., send or receive events) but only observe changes in the state of this model.

5.2 Modeling Büchi automata in UML

A first formalism that can be encoded into PUSMs is the Büchi automata formalism. Such automata are typically used in model-checking to encode both safety and liveness properties.

In terms of expressivity, Büchi automata can encode any ω-regular language, i.e., any regular language with infinite execution traces. These automata are even more expressive than LTL. With our solution, it remains possible to manually write Büchi automata as UML state machines to exploit the full expressivity of Büchi automata. In this way, some properties that cannot be encoded in LTL can still be model-checked.

In general, Büchi automata have non-determinism that enables to explore different possible paths at the same time. Moreover, these automata do not require to be complete, i.e., some execution paths can be cut intentionally such that they will not be explored if, provably, no failure can happen on those paths.

To perform the verification task, Büchi automata encodes the negation of the property to verify. All bad behaviors that should not be observed in the system are expressed in this automaton. Using a synchronous composition with the system, a failure is detected if an acceptance state is reached infinitely often [22]. In other words, a failure is detected if a bad behavior specified by the Büchi automaton is found in the system.

As an example, the first requirement (R1) of the CCI motivating example has been expressed into a PUSM to model a Büchi automaton. The state machine of this PUSM is described in Fig. 6 where the shaded state *Accept_1* denotes an acceptance state. This automaton starts in the *Init* state and can stay in it indefinitely using the self-transition with the guard “true”. This enables to explore the whole model state-space. Therefore, during this state-space exploration, if the cruise control has not been disengaged and a “stop” event has been received, the guard “!ccsDisengaged `&&` evStop” expressed using our observation language becomes true (cf. “Appendix C” to see the meaning of ccsDisengaged and evStop). In this case, the *Accept_1* state is reached. If the “!ccsDisengaged” atom remains true indefinitely, the

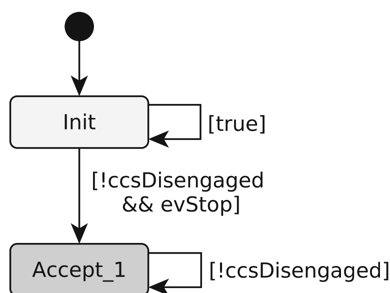


Fig. 6 State machine of PUSM (Büchi1) for R1 of the CCI

self-transition on this state can be fired continually and an acceptance loop is detected. Hence, the property is violated and a counterexample has been found.

Moreover, only one PUSM is synchronously composed with the system at a time. Theoretically, only one property can be verified at a time. However, a trick can be used to make the verification of N properties, named P_1 to P_N, simultaneously. All these properties can be composed together in one property (P_{all}) using the "and" operator (&& in LTL): $P_{all} = P_1$ && P_2 && ... && P_N. Then, a Büchi automaton can be produced for the P_{all} property and verified with a model-checker. Nevertheless, this technique has one drawback. If a counterexample is found, the engineer cannot know, without further analysis, which property has been violated.

5.3 Modeling observer automata in UML

The second kind of formalism which is considered in this work is the observer automata formalism, which is commonly used to monitor the system execution.

For runtime monitoring, the verification capabilities are restricted to the analysis of the current run of the system, as opposed to model-checking that analyzes all runs. This partial observation of the running system limits the expressivity of the verified properties to *monitorable properties* [5], which includes all safety properties. Nevertheless, this constraint can also be seen as a benefit for offline verification because the use of observer automata reduces the model-checking problem to a reachability problem. The model-checker has only to check if the acceptance states, also called "fail" states, of the observer automata are reached in at least one configuration of the whole state-space.

For runtime monitoring, PUSMs used to model observers must satisfy two additional constraints. The *determinism constraint* ensures that observer automata do not introduce (language) non-determinism in the running system but just follow model execution. To ensure determinism, we require that the guards of outgoing transitions of an observer state are exclusive. The *completeness constraint* ensures that observer automata do not block system execution when composed synchronously with the system. This constraint is automatically enforced by the synchronous composition operator, which completes the observer automata with implicit loop transitions (stuttering steps). Therefore, only deterministic and complete observers (i.e., with exactly one fireable transition at a time) must be deployed on the actual system for runtime monitoring. Due to the determinism constraint, multiple observer automata can be composed synchronously with the system. Therefore, any number of monitors can be used simultaneously on the actual execution platform to check the system behavior. Even in this case, it is quite easy to know which property has been violated simply by identifying which observer automaton reached a fail state.

As an example, the last requirement of the CCI motivating example (R6) has been expressed as a deterministic observer automaton, while requirements R5 and R6 have been combined together in a non-deterministic observer automaton. The state machines of these observer automata are described in Fig. 7. Each of them defines at least a "fail" state, shaded in the figure, that has to be reached in case of failure. All transition guards are expressed with our observation language (cf. "Appendix C"). Both observers are not complete yet but this will be inferred automatically during the synchronous composition.

Observer6 starts in its *Running* state and stays in it until the guard "ccsEngaged && unknownCS" expressed with our observation language becomes true. In this case, the *Fail* state is reached and the property is violated.

ObserverND switches between its *Engaged* state and its *Disengaged* state according to the system execution. Two kinds of failures can be detected. If the cruise control is disengaged and the cruise speed is not in its working interval and not unknown (i.e., not equal to -1), the *Fail_1* state is reached. If the cruise control is engaged and the cruise speed is not in its working interval, the *Fail_2* state is reached. *ObserverND* is non-deterministic because *Engaged* and *Disengaged* states have two outgoing transitions that may be fireable at the same time (i.e., both guards can be true at the same time). If this happens, two execution paths (one for each transition) are thus required to explore and verify the system behavior.

5.4 Conversion of LTL properties into PUSMs

We strongly believe that the formal verification of a UML model should be based on the UML executable semantics captured via an interpreter and not via transformations and intermediate languages. However, from the specification point of view, we do not want to overly constrain the designer in terms of the input language, providing the ltl4uml as an alternative to manual PUSMs modeling.

While our approach gives the possibility to write Büchi automata in UML by hand, the generation of such automata in UML (or in another formalism) can be easily automated

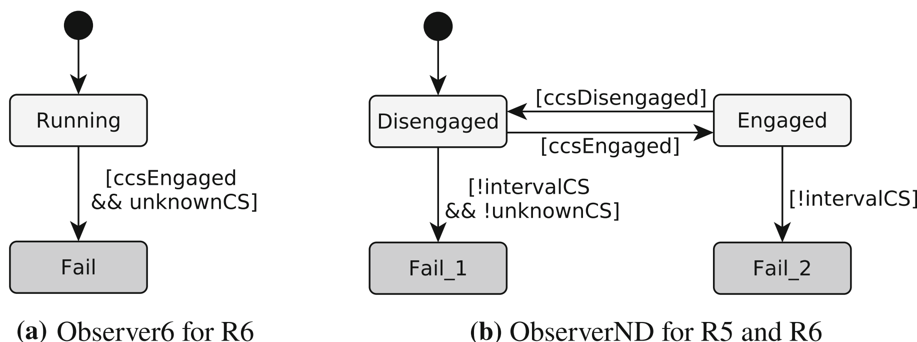


Fig. 7 State machines of PUSMs (representing observer automata) for the CCI

(a) Observer6 for R6

(b) ObserverND for R5 and R6

when a formal description of the property is given. For this purpose, this work presents a tool called ltl4uml that can automatically convert LTL properties into tUML [34,35], a textual formalism for UML. This tool adds a UML front-end to the powerful ltl3ba [2] library that performs efficient conversion of LTL properties into Büchi automata. ltl3ba has been initially designed by Paul Gastin and Denis Oddoux under the name ltl2ba in [25] before being improved by Tomáš Babiak et al. in [2]. Our ltl4uml tool takes as input the LTL property to verify. It captures all atomic propositions expressed into our observation language and adds a negation to the property. The resulting LTL property is sent to ltl3ba that produces the Büchi automaton encoding all undesired behaviors. A code generator made with Xtend[7] is then used to generate the tUML code corresponding to the PUSM representing the produced Büchi automaton.

For instance, the requirement R1 of the motivating example can be expressed in LTL as: `"[] (|evStop| -> (<>|ccsDisengaged|))"` where `|evStop|` and `|ccsDisengaged|` denote atomic propositions. Given this LTL property, ltl4uml generates the PUSM shown in Listing 1. This PUSM encodes the state machine presented in Fig. 6.

6 Synchronous composition

Given a property model and a model of the system, an essential concept of the verification and monitoring process is the synchronous composition of these models. This section discusses our synchronization operator.

6.1 Theoretical description

Informally, the principle of synchronous composition is quite simple: each time a transition of the system model is fired, the PUSM, representing the property model, also makes a step to follow the system execution. In fact, at each step, a synchronous transition composed of one transition of the system and one transition per PUSM is fired. This way, a failure in the system execution is detected as soon as it occurs. This offers a fail-fast detection mechanism.

```
class |PropBüchi1| behavesAs SM {
  stateMachine SM { region R {
    Initial -> Init;
    Init -> Init :
      [constraint "true" is
      opaqueExpression =
        'true' in C;] /;
    Init -> Accept_1 :
      [constraint "!ccsDisengaged &&
    evStop" is
      opaqueExpression =
        '(!(IS_IN_STATE(GET(GET(
    ROOT_instMain, cci),
        actuation),
    STATE_Actuation_Disengaged))
        && (EP_CONTAINS(GET(GET(
    ROOT_instMain, cci),
        controller), SIGNAL_stop)))' in C;] /;
    Accept_1 -> Accept_1 :
      [constraint "!ccsDisengaged" is
      opaqueExpression =
        '(!(IS_IN_STATE(GET(GET(
    ROOT_instMain, cci),
        actuation),
    STATE_Actuation_Disengaged)))' in C;] /;
    initial pseudoState Initial;
  }}
}
```

Listing 1 Generated tUML code for property 1 of the CCI motivating example.

We will now give a formal description of our synchronous composition operator to show how it works more concretely. The synchronous operator takes as inputs two execution engines: one EMI instance running the system model and another EMI instance running the property model containing a PUSM (cf. Fig. 3). For this definition, we assume that the system automaton is the `lhs` term while the PUSM is the

[7] Xtend: https://www.eclipse.org/xtend/.

rhs term. In Sect. 4, we show that the EMI communication interface can be easily converted into an STR interface. The synchronous composition operator uses this STR interface to steer each model interpreter. One evaluation function is also required for each execution engine.

To define this operator, L is the type of labels (or atomic propositions). The synchronous_composition of the system model (lhs) with the property model (rhs) gives an STR defined as:

```
def sychronous_composition (C₁ C₂ A₁ A₂ L₁ :
    Type)
  (lhs : @STR C₁ A₁)
  (eval₁ : L₁ → C₁ → A₁ → C₁ → bool)
  (rhs : @STR C₂ A₂)
  (eval₂ : C₂ → A₂ → L₁)
: @STR (C₁ × C₂) (A₁ × A₂) := ⟨
initial ← { (c₁, c₂) |
    ∀ c₁ ∈ lhs.initial c₂ ∈ rhs.initial },
actions ← λ (c₁, c₂), { (a₁, a₂) |
    ∀ a₁ ∈ lhs.actions c₁
     a₂ ∈ rhs.actions c₂
     t₁ ∈ lhs.execute c₁ a₁,
        eval₁ (eval₂ c₂ a₂) c₁ a₁ t₁ },
execute ← λ (c₁, c₂) (a₁, a₂),{ (t₁, t₂) |
    ∀ t₁ ∈ lhs.execute c₁ a₁
     t₂ ∈ rhs.execute c₂ a₂ }⟩
```

The initial configuration of the composition (**initial**) is the concatenation of both the initial system configuration and the initial PUSM configuration. To build synchronous transitions (**actions**), the execution engine of the PUSM executes its **eval**$_2$ function on all available actions and returns a list of atomic propositions (or atoms) needed for actions evaluation. These atoms are predicates that are evaluated with **eval**$_1$ on each executed step of the system (the tuple source configuration c_1, action a_1, target configuration t_1). Hence, the step a_1 has to be executed to get the target configuration of the system. Using the atoms valuations, transition guards of the PUSM can now be computed to know if the corresponding transitions are fireable. If a guard evaluates to true, it means that this transition of the PUSM (a_2) can be synchronized with the one of the system automaton (a_1). As a result, we get a synchronous transition corresponding to the tuple (a_1, a_2). The last part concerns the execution of a synchronous transition (**execute**). From a given configuration of the synchronous product, a synchronous transition is fired to obtain its target configuration. This means that the system transition (a_1) is fired on the system automaton and that the PUSM transition (a_2) is fired on the PUSM. The concatenation of both target configurations (t_1, t_2) results in the target configuration of the synchronous composition.

6.2 Optimization for runtime monitoring

For runtime monitoring, we can optimize this synchronous composition operator since only one execution path is covered. Before building synchronous transitions, the scheduler is called to choose the system transition to fire at the next execution step. In this case, the synchronous composition only has to compute one synchronous transition, which is more efficient than doing it for all fireable transitions of the system automaton. These fireable transitions are always computed in the current configuration of the system interpreter and the next transition to fire is always fired from this configuration. The target configuration of the fired transition is then considered as the current configuration for the next execution step. When the selected system transition has been fired, we can determine which transition of the PUSM can be synchronized with it and then fire this transition on the PUSM. Not only does this make more sense for runtime monitoring because the system transition is only fired once but it also helps to improve execution performance.

6.3 Adding implicit transitions

Using this generic synchronous composition operator, different formalisms can be supported. Each one has, however, its own specificities such that the synchronous composition has to be slightly adapted for each formalism. One common adjustment is to add implicit transitions either on the system automaton or on the PUSM such that the resulting automaton cannot result in deadlock. This mechanism is used to complete one automaton such that its execution will never block. For this purpose, the add_implicit_transitions operator can be used before the application of the synchronous composition operator to complete one automaton. Its principle is quite simple. If some actions from the input STR are available, these actions are used for model execution. Otherwise, if a deadlock is detected (i.e., no action is available), an implicit action is added such that a new self execution step can be taken from the current configuration.

```
def add_implicit_transitions
  (str : @STR C A)
  [∀ c, decidable (str.actions c = ∅)]
: @STR C (completed A) := ⟨
initial ← str.initial,
actions ← λ c, if str.actions c = ∅ then
    (singleton completed.deadlock)
  else
    { oa | ∀ a ∈ str.actions c, oa = completed.some
    a },
execute ← λ c oa, match oa with
  | completed.deadlock := singleton c
  | completed.some a := str.execute c a ⟩
```

Given this common basis, we will now describe the specificities of each formalism.

6.4 Synchronous composition with a Büchi automaton

In case of a Büchi automaton as PUSM, several specific cases need to be handled to apply the synchronous composition.

If no transition of the PUSM can be synchronized with a system transition, the execution trace provided by this system transition is cut. This means that this execution trace is of no interest for the property verification (e.g., the model-checker is sure that no violation can occur on this trace).

Another special case is linked to the Büchi automata formalism, which focuses only on infinite execution traces. If the system execution results in a deadlock, the execution trace is finite and no synchronous transition can be built. The execution of the PUSM is blocked and some failures may be missed. To overcome this limitation, an implicit self-transition is added to the system automaton, using `add_implicit_transitions`, such that some synchronous transitions can be computed and the execution of the property model can always proceed.

6.5 Synchronous composition with an observer automaton

A particularity of the synchronous composition with an observer automaton as PUSM concerns the computation of synchronous transitions. An observer automaton fires an explicit transition if one outgoing transition of its current state is fireable. Otherwise, the observer automaton will fire an implicit self-transition, created in accordance with `add_implicit_transitions`, to ensure the completeness requirement. As a result, observer automata will never block the system execution. Using this setup, it means that PUSMs representing observer automata in Figs. 2b and 7 are complete because implicit self-transitions are inferred automatically.

7 Verification and monitoring architecture

Based on PUSMs, this section describes the verification process used (i) to verify formal properties with a model-checker or (ii) to monitor the system execution running on an actual embedded target.

7.1 Model-checking architecture

With the *Model Execution* component, it becomes possible to connect a model-checker to our UML model interpreter for verifying properties encoded by PUSMs. An important

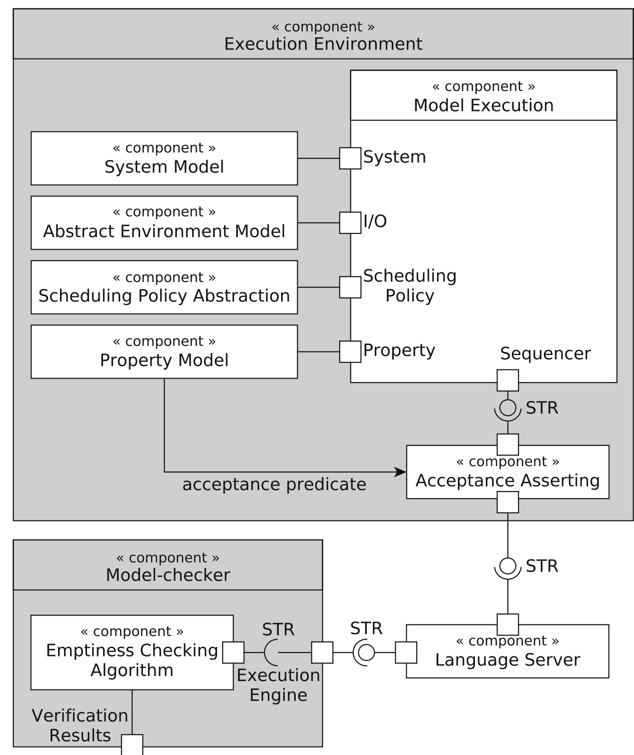


Fig. 8 Architecture for model-checking with PUSMs

prerequisite for applying model-checking techniques is to close the system model with a proper abstraction of its operational environment. For this reason, we strive to understand as much as possible the context in which the CCI operates for modeling a relevant abstraction of it (cf. "Appendix A.1").

The software architecture used for model verification with PUSMs is shown in Fig. 8. For offline verification, the *Model Execution* component is connected to the *System Model*, the *Abstract Environment Model*, and the *Property Model* that have all been designed in UML. This component is also connected to an abstraction of the scheduling policy (*Scheduling Policy Abstraction*) to consider a superset of all possible cases. A suitable, very general, abstraction is to return all fireable transitions of the system to explore all the model state-space.

Moreover, the model-checker is connected to the *Model Execution* component through a *Language Server* that provides language specific facilities such that verification tools remain as modular as possible. A model-checking algorithm, called here *Emptiness Checking Algorithm* in a generic way, is used as the main sequencer of the verification process. It explores all the model state-space by communicating with the *Execution Environment* through the view exposed by the *Synchronous Composition* with the STR interface.

Furthermore, the execution of the *Emptiness Checking Algorithm* requires to store one additional bit in each explored configuration. The value of this bit is computed at each step

by the *Acceptance Asserting* component that takes as input the *acceptance predicate* given by the property model. The *acceptance predicate* is a Boolean expression that enables to determine if the PUSM is in one of its acceptance states. The resulting Boolean value is added to the current configuration before being sent to the model-checker. During the state-space exploration, the model-checker checks if the acceptance condition (cf. Sect. 5.1) has been fulfilled. As soon as this condition is satisfied, the model-checker stops the verification and returns the counterexample found as a trace. Otherwise, it will explore the entire model state-space to ensure that the property encoded by the PUSM is verified. This approach offers the advantage to express verification results directly in terms of design concepts. This also avoids the use of model transformations (from code back to model) to obtain the same result, approach sometimes used in other works [16,44,45].

Given this generic verification architecture, different formalisms can be supported. The only difference is the model-checking algorithm used as *Emptiness Checking Algorithm*. For Büchi automata, we use the *acceptance cycle detection algorithm* defined by Gaiser and Schwoon in [24]. For observer automata, the verification problem is reduced to a reachability problem (cf. Sect. 5). Therefore, only a *reachability algorithm* can be used as the main sequencer of the verification process.

7.2 Runtime monitoring architecture

Once the model verification has been performed, the UML model can be deployed on the actual embedded target. To continue verification of monitorable properties at runtime, it is possible to embed all PUSMs representing deterministic observer automata. Contrary to model-checking that verifies the software program offline in an abstract environment, monitoring enables the verification of a running system online in its real (or simulated) environment.

The software architecture used for monitoring is shown in Fig. 9. The *Model Execution* component used during model verification is reused for monitoring with the same *System Model* and *Property Model*. However, this time, *Model Execution* is linked to actual *I/O* of the embedded board and the *Actual Scheduling Policy* of the system is used rather than an abstraction of it. At each step, the scheduler will select one and only one transition to fire among the set of fireable transitions of the system. For monitoring, the choice of the next transition to fire is made before applying the synchronous composition to eliminate the risk of scheduler-interference on the system monitoring and keep efficient monitoring performance. To steer the *Model Execution* component, the main *Execution Loop* on the deployment platform is used as a sequencer. For each step, three main operations are performed. First, it computes the next synchronous transition

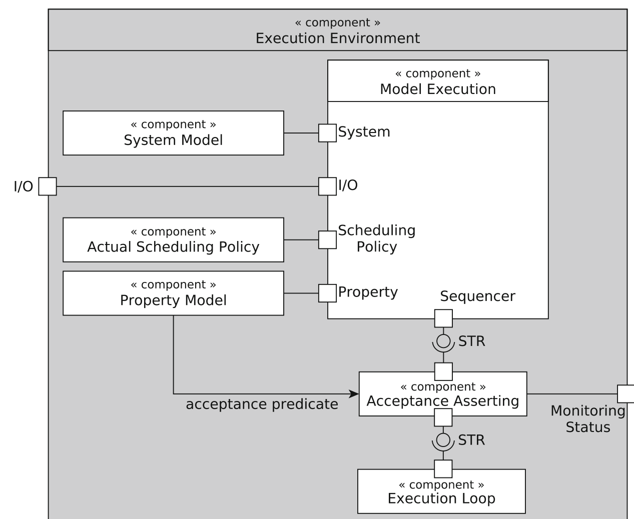


Fig. 9 Runtime monitoring architecture with PUSMs

to fire. Then, it fires this transition. Finally, it delegates the verification of formal properties to the *Acceptance Asserting* component. This last component checks if observer automata have reached one of their acceptance states and updates the *Monitoring Status*.

One main advantage of our approach is that the same deterministic observer automata used during the verification phase can be deployed on the target and reused for runtime monitoring without effort (i.e., without transformation, code generation, or model binding). Despite the possibility of offline verification, it still remains useful to monitor system execution for several reasons. First, if the abstraction of the environment used during model-checking is not complete or badly defined, it is possible that not all real cases have been covered and that a bug has been missed. Second, due to the state-space explosion problem, it is not always possible to model-check safety properties. With our approach, such properties can always be monitored at runtime without the need of costly model transformations. Another benefit is that monitoring can detect violation of safety properties caused by deficient hardware components, which is not possible with model-checking. When a failure is detected, PUSMs can notify the problem to the user (e.g., by printing an error message) or activate the appropriate fail-safe controllers (e.g., error recovery, runtime-safety enforcement). Finally, the traces of observer automata can be used in post-mortem analysis to understand why the system has failed.

In terms of limitations, the use of observer automata in monitoring, like most of monitoring activities, has a resource overhead both in memory footprint and execution performance. A trade-off between verification quality and execution performance must be found for each context. Another drawback is that monitoring can only detect the presence of errors. Monitoring, unlike exhaustive veri-

fication techniques, observes execution steps taken by the system under the actual environment. Therefore, its efficiency depends on the failure coverage provided by monitors embedded with the system.

8 Experiments and results

The UML language is used as the de facto standard in industry to design software systems. In the context of our work, we especially focus on critical embedded systems that require to apply analysis activities (i.e., simulation, debugging) during early design phases as well as formal verification activities such as model-checking and runtime monitoring. Our approach is especially valuable for this kind of system because V&V efforts are huge and applied all along the development cycle to check critical behaviors of these embedded systems. For these reasons, multiple experiments have been applied on UML models in different fields of this target domain: automotive [10], railway [8], aeronautics [13], robotics [9], healthcare [23]. This section sums up all analysis activities that have been applied on these models using EMI. These experiments aim at evaluating our approach for checking the validity of formal requirements expressed for each system.

During these experiments, system requirements have been expressed as PUSMs in the same way as requirements of the CCI motivating example (introduced in Sect. 2). The verification of these properties as PUSMs has been made using model verification with the OBP2 model-checker and results have been compared with verification results of identical properties expressed in LTL. Deterministic observer automata have also been deployed on an actual embedded target (an STM32 discovery board) to perform runtime monitoring and measure the induced overhead.

8.1 Simulation and debugging

Before applying formal verification, an essential step has been to design the UML models for each system. To help in this task, OBP2 provides a graphical interface, shown in Fig. 10, enabling simulation and multiverse debugging [58]. To perform these activities, OBP2 is connected to EMI running either on a desktop computer or on an embedded target.

The state-based simulator enables to explore some execution traces. It also provides a back-in-time functionality such that it is possible to change the current configuration of the model interpreter and continue the simulation from this point.

Based on the same setup, we have also applied multiverse debugging on our UML models. This analysis activity has been introduced in [58] as the possibility to define breakpoints that can halt the execution in different execution paths,

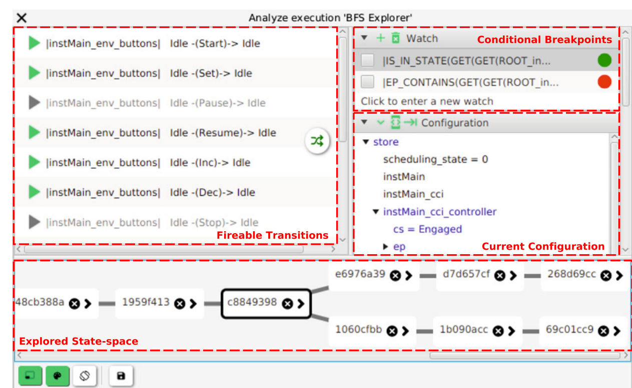


Fig. 10 OBP2 graphical interface for simulation and multiverse debugging

also called universes. This functionality has been implemented in OBP2 with a reachability algorithm. Indeed, from a given configuration, it is possible to explore the state-space until a predicate becomes true (or false). In the graphical interface of OBP2, we add the possibility to define some conditional breakpoints as predicates in the observation language of EMI (cf. Sect. 4.2). As a result, the reuse of the observation language simplifies this task for engineers because these predicates are directly expressed in terms of design concepts. OBP2 evaluates these predicates in each new configuration explored and shows the result to the user using a green or a red indicator if the predicate evaluates, respectively, to true or false.

Results: Both state-based simulation and multiverse debugging are very helpful to finely tune the system model

8.2 Model-checking the system behavior

For offline verification, system requirements of each modeled system have been designed as PUSMs. Liveness properties (corresponding for instance to R1, R2 and R3 in the motivating example) have been encoded into Büchi automata, while safety properties (corresponding to R4, R5 and R6 in the motivating example) have been expressed as deterministic observer automata. Following the setup in Fig. 8, these PUSMs have been loaded in the *Execution Environment* with OBP2 as model-checking component.

This process has been performed with PUSMs of the motivating example and we obtain the following results. For liveness properties, each Büchi automaton has been verified separately and no property violation has been detected by OBP2 for any of them. Using the trick mentioned in Sect. 5.2, the Büchi automaton resulting from the conjunction of these three properties has also been generated and successfully verified (cf. "Appendix A.2"). For safety properties, both properties 4 and 5 are verified, while property 6 is violated.

To check the validity of our approach, we have compared these results with model-checking results of identical properties expressed in LTL. For this purpose, all system requirements have been specified in LTL (cf. "Appendix A.2"). These LTL properties link the atomic propositions, which are directly evaluated on the UML model interpreter, with different LTL operators: not (`!`), or (`or`), and (`&&`), globally (`[]`), eventually (`<>`), until (`U`), weak until (`W`), and implies (`->`). Atomic propositions involved in these LTL properties are the same as the ones used for transition guards of PUSMs and thus defined using the observation language.

As a result, expressions of these properties in LTL are more complex to write because it requires the knowledge of LTL operators and especially temporal modalities (e.g., globally, weak until). For instance, the safety property 4 of the motivating example, which is not a state-invariant, is not trivial to express in LTL while the corresponding observer automaton (Fig. 2b) is quite simple to design. Model-checking of these LTL properties with OBP2 reports the same results to those obtained with PUSMs. For instance, for the CCI example, all properties were verified except the property 6 which is violated.

To understand why property 6 of the CCI is violated, the counterexample returned by OBP2 has been analyzed (see [10]). A design error due to a bad event interleaving has been identified and fixed. With this fix, the corrected CCI model has a state-space containing 17,134,122 configurations linked by 29,088,210 transitions.[8]

We have also made a comparison of performances between LTL-based and PUSM-based model-checking both in terms of time and memory taken by the model-checker to make the verification. In average for this CCI example, PUSM verification is more efficient than LTL verification of 6.7% in memory and 7.4% in time. These improvements can be explained by the fact that the C implementation of the synchronous composition used with PUSMs is apparently more efficient than the Java implementation used with LTL properties.

Results: PUSMs verification gives equivalent results to LTL verification without adding overhead.

8.3 Model-checking fail-safe mechanisms

During these experiments, we have also explored the possibility to use fail-safe mechanisms. For the CCI, if a failure is detected, an appropriate action can be to turn off the system such that the driver regains the control of the vehicle. For this purpose, we add the sending of a "stop" event to the controller on transitions incoming into "fail" states of observer automata. In this case, observer automata are not only observing the system execution but they are also reacting when a failure occurs. The good behavior of this mechanism can be verified using the following LTL property:

```
"[] (|observerInFailState| ->
    [] (|evStopToController| -> (<> |ccsDisengaged|)))"
```

This property has been successfully verified with OBP2 for observer 6 that reaches its "fail" state on the initial version of the model.

Results: Our observation language is sufficiently expressive to express properties about the monitored system and our observer automata can be used for creating fail-safe mechanisms.

8.4 Monitoring

Once verified, the UML model of each designed system has been deployed with our embedded model interpreter on STM32 discovery boards. For runtime execution, these models interact either with the real environment through inputs and outputs of the board, or with a simulated environment when real sensors and actuators could not be used. In the latter case, for simplicity, the simulated environment matches with the environment abstraction used for verification. The objects of the environment are thus managed by the scheduling policy and the sequencer of the *Execution Environment*. These objects send events to the system under development (as the real environment would have done) according to their abstract behavior. PUSMs representing observer automata have also been deployed with the UML model of the system to perform runtime monitoring following the setup in Fig. 9. No failure has been detected on the model-checked version of the UML models. These results are consistent with results obtained through model-checking. Moreover, for the motivating example, the deployment of the initial version of the CCI model has shown that the observer automaton for property 6 would have succeeded to detect the failure if it had occurred.

In terms of performance, runtime monitoring induces resource overheads compared to the execution of the same UML model without observer automata. In addition to the costs of monitors, monitoring increases the execution time of 6.5% due to the use of the synchronous composition. The cost of monitors depends on the size of the system model and on the number of observer guard evaluations required at each step. An estimation of the overhead (in %) induced by N monitors in terms of execution time is given by the following equation:

$$\texttt{overhead} \approx 6.5 + \frac{100}{\texttt{nb_ao}} \sum_{i=1}^{N} \frac{\texttt{nb_outgoings}_i}{\texttt{nb_states}_i}$$

[8] The difference that can be observed with our paper [10] is due to the fact that we have improved the abstraction of the environment model.

where nb_ao is the number of active objects in the system, nb_states_i is the number of states (excluding pseudostates and "fail" states) of monitor i, and $nb_outgoings_i$ is the sum of outgoing transitions of considered states. For instance, the use of one observer automaton with a system model containing 10 active objects will add an overhead of 10%, while this overhead would only be 1% if 100 active objects were used. For each observer, this cost is then weighted by $\frac{nb_outgoings}{nb_states}$ i.e., the average number of guards evaluated at each step for this observer automaton. For the CCI model, this equation gives an estimated overhead of 50.2%, while in practice we obtained 50.8%. In terms of memory footprint, the measured overhead is 8.2% including approximately 1.2% for the synchronous composition and 7% for the three monitors. These measures have been made by comparing the time taken by the *Execution Environment* to fire 1,000,000 transitions and the size of binary executables with and without observer automata. From our perspective, these resource overheads are acceptable for execution on embedded systems. However, in general, the overhead metrics should be corroborated with the specific constraints and criticality level of each system. From the overhead equation, it follows that this approach is scalable for runtime monitoring, because the relative cost of integrating one observer automaton decreases as the size of the system model increases.

Results: Monitoring results are consistent with model-checking results and the proposed solution for monitoring is scalable.

9 Discussions

In this section, we want to discuss some points in relation to our approach and its application on EMI.

9.1 Trade-off between language expressivity and verifiability

Our work tends to establish a balance between two often contradictory intentions: (1) the design of models that can be verified and analyzed by automated tools and (2) the design of industrial software applications by engineers that are not formal experts. While the former requires well-defined concepts (most of the time defined in a formal way) to apply V&V activities, the latter is usually looking for highly expressive languages using semantically complex concepts. In this paper, we aim at providing an approach for both objectives by allowing a uniform handling of V&V activities on embedded system models designed by engineers. Admittedly, our model interpreter only implements a subset of UML that can be represented by class, composite structures and state machines diagrams. Nevertheless, the implementation of the UML semantics captures by our tool can be extended to take into account additional UML concepts. Among them, let us take the example of hierarchical states and compound transitions that may be interesting to add in EMI such that engineers may benefit of these extensions to design UML models and encode formal properties as PUSMs. Some other tools like AnimUML [33] or USMMC [42] used for executing and analyzing UML models have already implemented these concepts with no major difficulty. Adding such concepts to the UML subset supported by EMI is only a technical limitation (not a scientific one). However, a trade-off has to be found between language expressivity and code certifiability, which is really important for embedded systems. Indeed, increasing the supported UML subset also increases code complexity and renders code certifiability more difficult. In particular, some highly expressive concepts may have complex semantics that could easily make V&V of the models impractical.

9.2 Mapping UML concepts on property formalisms

Regarding our automatic conversion from LTL to UML with the ltl4uml tool, adding hierarchical states and compound transitions is not needed because the Büchi automata formalism does not have such concepts. Therefore, the UML subset supported by EMI is currently sufficient to map Büchi automata concepts on those of UML. Nevertheless, adding hierarchical states and compound transitions may be useful to extend our work with other automata formalisms that have such concepts.

9.3 Richness of the execution semantics

In general, the more the execution engine supports a rich execution semantics, the more the engineers can design models easily and in a compact way. However, the complexity of the supported execution semantics is implementation-specific and quite independent of the substantial part of our approach that relies on the STR interface to connect analysis tools to the execution engine. The analysis tools have no idea if the supported language semantics implementation supports hierarchical states, compound transitions, or any other concepts. The way the semantics is implemented is hidden to analysis tools through the STR interface. Indeed, the execution semantics is only exposed in terms of configuration and observable execution steps. Extending the UML subset supported by our model interpreter is thus independent of our verification architecture due to the use of the STR interface.

9.4 Action and observation languages

Another interesting point concerns our action and observation languages used for expressing guards and effects on transitions of UML state machines. The alignment of

these languages with the underlying C system has been made possible by using, respectively, opaque expressions and opaque behaviors for guards and effects of transitions. These UML concepts enable engineers to define some expressions in a different language than UML (the C language in our case). Loading these expressions in the memory of our model interpreter, also using C language for its implementation, makes these expressions executable. The C macros used for implementing operators of these languages are thus directly operating on runtime data at implementation-level. Our action and observation languages can thus be easily extended by defining additional C macros that access different runtime data of the execution engine. These mechanisms based on opaque expressions and opaque behaviors also offer the possibility to replace our languages by different ones on condition that they respect the UML run-to-completion step semantics.

9.5 Limitations

The fact that guards and effects of state machine transitions are specified using C brings benefits (e.g., well-known language, certifiability) in the context of our approach but this can also be a limitation. In fact, it is difficult to ensure that these opaque expressions and actions respect the atomicity of the UML language and its run-to-completion step semantics (e.g., how to ensure that there is no infinite loop in C expressions?). These consistency checks still have to be made by the model designer itself. Another limitation is that our UML models have to be well designed such that the model state-space would be bounded. For instance, for the CCI model, the integer variable used to represent speed has been defined into the [0; 200] range and not into the full integer range (i.e., $[-2^{32}; 2^{32} - 1]$) to avoid state-space explosion. Moreover, the speed cannot be more abstract for verification purpose because it has to stay sufficiently concrete for being deployed on an embedded target. Nevertheless, our approach can enforce some analysis hypothesis (e.g., the hypothesis on reactive systems [20]) in a modular way and without modifying the design model. This is performed by applying some filters on the actions returned by the STR interface but this is out of the scope of this paper.

Regarding more global limitations, our approach currently targets embedded systems running on a single core and without temporal constraints. To overcome these limitations, it would be interesting to see how our approach can be transposed in (i) a distributed context where the system is deployed on multiple cores, and in (ii) a real-time context where properties about real-time constraints have to be verified.

10 Related work

The work presented in this paper proposes to use PUSMs for specifying, verifying and monitoring formal properties on UML models. The particularity of our approach is the use of a semantically homogeneous framework, based on UML statecharts, which relieves the need for model transformations. Multiple other works use Büchi or observer automata to specify and verify system requirements. This section is divided in two parts: the first one focuses on the use of both Büchi and observer automata for offline verification while the second one is dedicated to monitoring.

10.1 Offline verification with Büchi automata or observer automata

Büchi automata are typically used in model-checking [51] to exhaustively check if a property holds on a model state-space [3]. LTL properties can be easily transformed into Büchi automata using automatic translation tools like ltl2ba [25], ltl3ba [2], or SPOT [22]. The Büchi formalism is then used in model-checkers (e.g., OBP2 [56,57], SPOT [22]) to synchronously compose the property automaton with the system automaton and verify the validity of the property. In comparison with the use of Büchi automata in model-checkers, our approach enables to express Büchi automata in the design language. The understanding of these automata by engineers is simplified, and more expressive properties can be encoded than with LTL.

Observer automata are used to ensure that a system model or an implementation satisfies its requirements. One typical approach applies model transformation techniques for converting UML observer automata to the automaton formalism used by the verification tool [43]. A similar work [44,45] uses a UML profile to express timing constraints of embedded real-time systems as UML observer automata. A mapping of these observer automata to extended timed automata is made with the IF language. Another technique [38,39] used by Hugo/RT aims at transforming interaction diagrams into observer automata for checking that UML state machines interact according to the scenarios described as UML collaboration diagrams. In our approach, neither model transformation nor mapping towards an intermediate language is required because, with the UML model interpreter,

verification activities are directly applied to the design model. More tools about verification of models using the UML formalism can be found in [17].

The synchronous language Lustre [26] can be used to describe reactive systems and express safety properties using synchronous observer automata. In [11], Airbus uses such synchronous observer automata to specify safety properties and perform their verification with the SCADE model-checker. The main advantage of this technique is the use of a synchronous language that renders the synchronous composition straightforward. The technique presented in our paper can be seen as a transposition of these research efforts, from the synchronous-language community, to the world of model-based executable specification with UML. In brief, our approach enables the use of observer-based verification and monitoring in the context of UML with the same simplicity as synchronous languages. Moreover, our approach does not require code generation for deployment as it is usually the case in the context of synchronous languages.

10.2 Monitoring

Regarding monitoring, multiple works have defined efficient software architectures or algorithms to perform runtime verification of monitorable properties.

A first approach is to define monitoring algorithms for dedicated logics. The work in [28] defines two techniques to analyze Java programs by checking if a trace of events satisfies LTL properties. The first one is based on a rewriting-based framework that allows defining new logics for monitoring execution traces. The second one [27] aims at synthesizing monitors for safety properties by generating efficient code from LTL formulas. This work has been extended in [54] to synthesize monitors for an extension of past time LTL, called ptCaRet. This logic offers the ability to express safety properties about procedural programs that cannot be specified using LTL. This automatic synthesis of monitors is also used in [5] to focus on runtime verification of LTL properties by analyzing finite prefixes of infinite traces. Other works define new temporal logics (e.g., ALTL [53] or EAGLE [4]), based on LTL, that are better suited for runtime monitoring. These logics enable to define more efficient monitoring algorithms that reduce either space and/or time complexity. While these approaches focus more on the monitoring synthesis process, we noticed that none of these related works mention the possibility to perform runtime verification by directly deploying monitors on actual embedded systems, while this opportunity is offered by our tool. However, these works provide interesting mechanisms, to synthesize monitors from formal properties, that may be used to extend our ltl4uml tool.

For the analysis of Java programs, different tools can be used to perform runtime verification. The Monitoring and Checking (MaC) architecture [37] defines a modular and flexible architecture with a clear separation between high-level requirements, independent of the implementation, and low-level behaviors on which they rely. Inspired by MaC, Java PathExplorer [29] facilitates the instrumentation of Java bytecode to check high-level requirements with an observer but also low-level errors like deadlocks or data-races. The Monitoring-Oriented Programming (MOP) framework [15] enables to express formal specifications using annotations in the design program. Based on these annotations, monitors are automatically synthesized and inserted at appropriate locations in the program. The Java-MOP prototype relies on AspectJ aspects to weave monitors into the executable code. In the same spirit, Temporal Rover [21] can check temporal logic assertions written as annotations in Java, C, C++, Verilog or VHDL programs. For program instrumentation, AspectJ aspects are also used in Clara [12] and MARQ [52] to weave monitoring code into Java programs. Clara uses partial evaluation to statically optimize monitoring aspects such that only what fails to be proved safe at compile-time is monitored. This tool reduces the monitoring overhead and renders the remaining monitors more efficient. MARQ also focuses on monitoring performance. For this purpose, it implements different optimizations to check efficiently properties expressed as Quantified Event Automata (QEA). In comparison with these works that instrument the executable code using aspects, our approach brings a solution to monitor the system execution by operating only at model-level.

Furthermore, monitoring can be achieved by tracing model execution. In [32], a debugger uses an embedded monitor to produce back annotated traces and build sequence and timing UML diagrams in real-time for visualizing the model behavior. In the same way, the project in [16] aims at monitoring extra-functional properties using annotations in the UML design model and back-propagation of analysis results to this model. In the same way as our work, both approaches give analysis results in terms of the design concepts. However, none of them is able to make runtime verification of embedded systems requirements.

Regarding critical systems, the literature provides multiple solutions for monitoring these systems. The work in [30] proposes a model-based architecture to monitor the execution of real-time and embedded systems. This framework allows connecting various monitoring tools to observe the system execution (e.g., for runtime verification) but also to interact with it (e.g., for debugging purposes). A prototype has been designed for UML-RT in the Papyrus-RT tool. The instrumentation of UML-RT models has been automated using model transformations. For hard real-time programs, monitors can be designed in a dataflow language called Copilot [50]. From the monitor specifications, the Copilot compiler generates embedded monitors in C as well as its own scheduler for the real-time operating system. In [36], a software architecture has been designed to monitor safety-critical

embedded systems that rely on black-box components. In this architecture, a communication bus is used to communicate with the different components of the system and check high-level properties. More inline with the philosophy of our work, a model-based framework for testing and monitoring hybrid embedded systems has been designed in [55] to narrow the semantic gap between designs and implementations. At the modeling level, the system is composed of a monitoring automaton and a testing automaton to check safety properties when executing the testing scenarios. A code generator can then convert the system model and both the testing and the monitoring automata into code to check the same properties at the implementation level. In comparison with our work, all these approaches rely on model transformations or code generation to monitor system execution. Our approach avoids such techniques to ensure that the observer automata used for monitoring are exactly the same as the ones used during model verification.

Finally, we are not aware of other approaches enabling both observer-based model-checking and runtime monitoring of executable UML specifications without the use of costly model transformations.

11 Summary and future work

The approach presented in this paper aims at modeling formal properties with PUSMs in order to facilitate their design by engineers, and to unify model verification and runtime monitoring of UML models. Execution of these models relies on EMI, a UML model interpreter based on a single implementation of the UML semantics.

With this technique, formal properties can be expressed as Büchi automata or observer automata in UML. These automata are modeled as UML state machines and rely on an observation language to access system objects and their attributes as well as internal runtime data of the execution engine. Each automaton formalism provides different benefits. Büchi automata can express any temporal logic property, and are even strictly more expressive than LTL. Observer automata can only encode monitorable properties but they can also be used unchanged for runtime monitoring (if they are deterministic). Each PUSM can be directly executed by an instance of our UML model interpreter, and is synchronously composed with the system execution. The synchronous composition operator is the main UML extension on which our approach relies in order to synchronize PUSMs with the system execution. For offline verification, this setup can be used to check exhaustively, with a model-checker, that properties encoded by PUSMs are not violated. It does not require any model transformation because verification is directly applied on the design model interpreted by EMI. For runtime monitoring, all deterministic observer automata used during offline verification can be deployed on embedded targets without the need of costly proven model transformations or code instrumentation. As a result, what is monitored at runtime is exactly what is checked during model verification.

This approach uses the same design language for both system modeling and property specification, thus facilitating the use of formal verification techniques by system engineers. In practice, not only this facilitates the expression of formal properties but also the analysis of verification results, which are directly captured within the UML formalism.

The approach was evaluated on a UML model of a cruise control interface through different V&V activities. We applied multiverse debugging by reusing the observation language to specify conditional breakpoints. We have also performed model-checking and runtime monitoring with PUSMs. The results show that system requirements can be easily expressed as a UML property model. For offline verification, the PUSM-based verification results are equivalent to LTL-based model-checking, while being slightly more efficient in terms of performance. For online verification, the deployment of observer automata on an STM32 embedded board induces a slight overhead, both in memory footprint and execution performance. However, this overhead does not impede scalability because the relative cost of one observer automaton decreases as the size of the system increases.

Possible extensions of this work also include the integration of other model-based specification formalisms such as Property Sequence Chart (PSC) [1] that relies on an extension of UML 2.0 interaction diagrams. We are interested in showing the applicability of our approach outside the UML world by transposing our verification and monitoring infrastructure to other design languages and other design contexts (e.g., real-time, distributed). In terms of performance, further work also includes making a performance-wise comparison of the OBP2 back-end with other model-checkers and verification tools.

Acknowledgements This work has been partially funded by Davidson Consulting. The authors especially thank David Olivier for his advice and industrial feedback. This project has been supported by the French Directorate General of Armaments (DGA), the European Regional Development Fund (ERDF) of the EU, the Brittany Region, the Departmental Council of Finistère and Brest Mètropole as part of the Cyber-SSI project within the framework of the Brittany 2015-2020 State-Region Contract (CPER).

A Additional information about the motivating example

This appendix gives more detailed information on the motivating example used in this paper. "Appendix A.1" describes the context used to model the system and its environment as realistically as possible and the system behavior that has

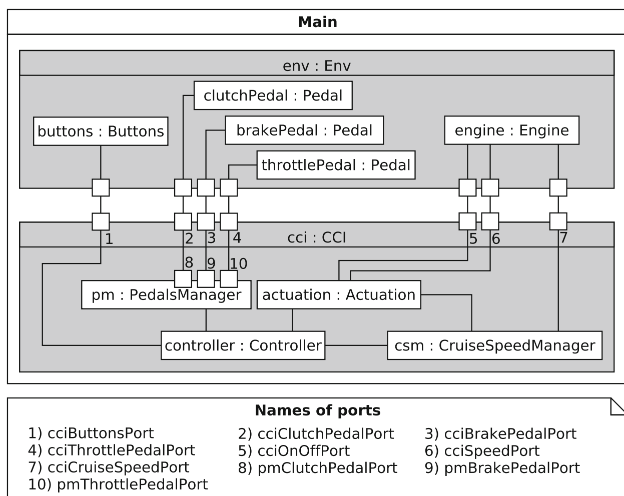


Fig. 11 Composite structure diagram of the CCI model

been captured in the design model. "Appendix A.2" presents the formal properties that have been expressed on this model using PUSMs or the LTL formalism.

A.1 Description of the motivating example

To apply our approach to the motivating example, we have designed a UML model of a CCI. The composite structure diagram of this model is shown in Fig. 11. The *Main* class is the root composite class of the model. It contains the *cci* part, which is the system under study, and the *env* part that models its environment. Both parts communicate by exchanging signals through ports.

Environment abstraction The *env* part contains a *buttons* object that models the different buttons (i.e., start, stop, inc, dec, set, pause, resume) that can be manipulated, as well as the three pedals (i.e., *clutchPedal*, *brakePedal*, *throttlePedal*) that can be pressed or released by the driver. According to Fig. 1, both the *PhysicalVehicle* and the *ControlLoop* are also parts of the environment. In our UML model, they have been abstracted as the *engine* object. In a real vehicle, the CCS will try to adjust the speed of the vehicle to the cruise speed given by the CCI, but, due to physical constraints (e.g., road profile, air friction), it is not always possible for the CCS to maintain the vehicle at the user-set speed. To take that into account, the *engine* does not make any correlation between the cruise speed given as input and the current speed it returns. As a result, the speed can go non-deterministically from 0 to 100 km/h in one step. This abstraction enables to consider a superset of all possible cases for the verification activity.

System under test The *cci* part describes the system that we want to verify. This system aims at sending new setpoints (i.e., the current value of the cruise speed) to the *engine* according to user actions and the current speed of the vehicle. The behavior of active objects contained in the *cci* is defined by state machines presented in Fig. 12. The fine-grained behavior of these state machines is described with our action language described in Sect. 4.2 of this paper. The *controller* (Fig. 12a) receives events from *buttons* and from *pm* (the pedals manager in Fig. 12b), which is connected to the three pedals (*clutchPedal*, *breakPedal*, and *ThrottlePedal*) through ports. Based on these events, the *controller* determines the status of the CCS and delegates generation of output events to both *actuation* and *csm* (cruise speed manager) objects. The *actuation* (Fig. 12c) sends *On* and *Off* signals to, respectively, activate the control loop when the CCS is engaged (i.e., the CCS is turned on and acts on the engine), and deactivate the control loop when the CCS is turned off or disengaged (i.e., the CCS is turned on but does not act on the engine). The cruise speed manager (Fig. 12d) computes the value of the cruise speed according to *buttons* events filtered by the *controller*, and sends new setpoints each time the *actuation* requests it. On all these state machines, some additional self-transitions (i.e., transitions that start and end in the same state) may be needed to explicitly ignore some events according to the event dispatching strategy chosen by the model interpreter.

A.2 Formal properties

For the CCI, the six formal properties have been expressed as PUSMs from the system requirements expressed in Sect. 2. In addition to the previously introduced PUSMs (cf. Fig. 6 for R1, Fig. 2 for R2 and R4, and Fig. 7 for R6), state machines of PUSMs representing a Büchi automaton for R3 and a deterministic observer automaton for R5 are shown in Fig. 13.

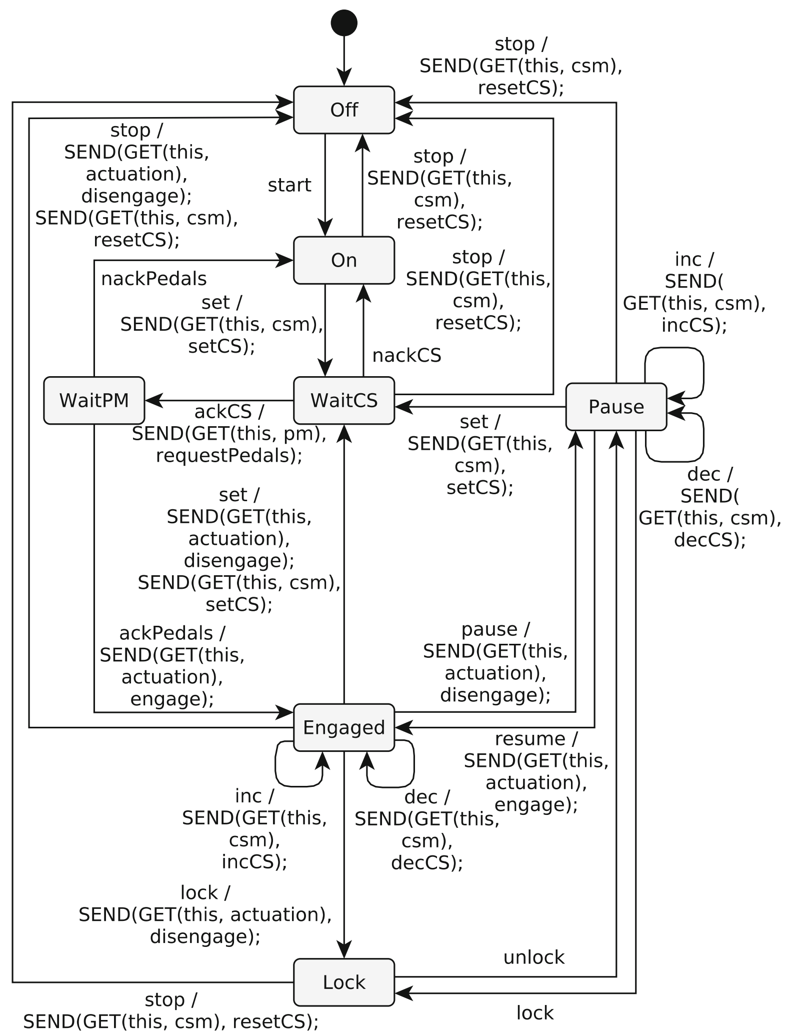
To verify the three requirements R1, R2, and R3 at the same time, the mechanism in Sect. 5.2 has also been used to generate one Büchi automaton that checks all three requirements. The state machine of this PUSM is shown in Fig. 14.

For our experiments, all system requirements have also been specified in LTL. The resulting LTL properties are:

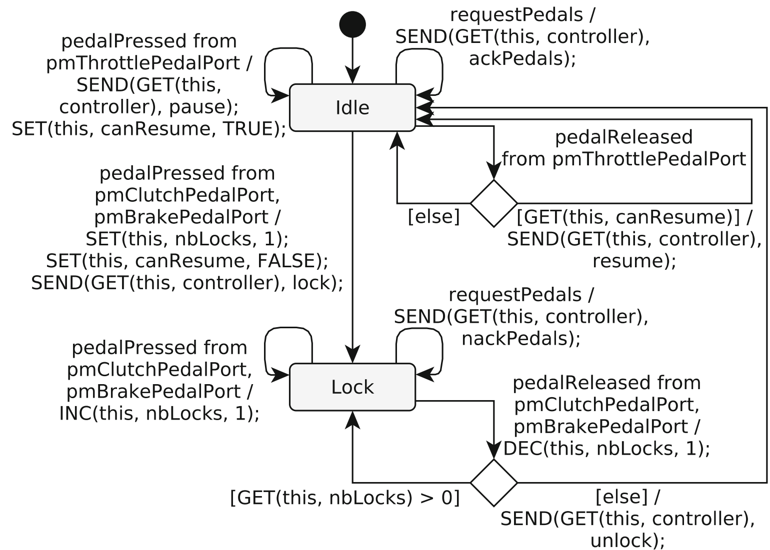
```
1. P1 = "[] (|evStop| -> (<> |ccsDisengaged|))"
2. P2 = "[] ((|ccsDisengaged| && |evSet|) ->
        (<> |ccsEngaged|))"

3. P3 = "[] ((|canResume| && |evThrottleReleased|) ->
   (|evThrottleReleased| U |evResume|))"

4. P4 = "(!|evUpdateSetPoint| W |evOn|) &&
   ([] (|evOff| -> (!|evUpdateSetPoint| W |evOn|)))"
5. P5 = "[] (|intervalCS| or |unknownCS|)"
6. P6 = "[] (|ccsEngaged| -> !|unknownCS|)"
```

Atomic propositions involved in these LTL properties are defined such as `|atom|` where `atom` is one of the labeled predicates (cf. "Appendix C") defined using our observation language.

Fig. 12 State machines of the CCI model

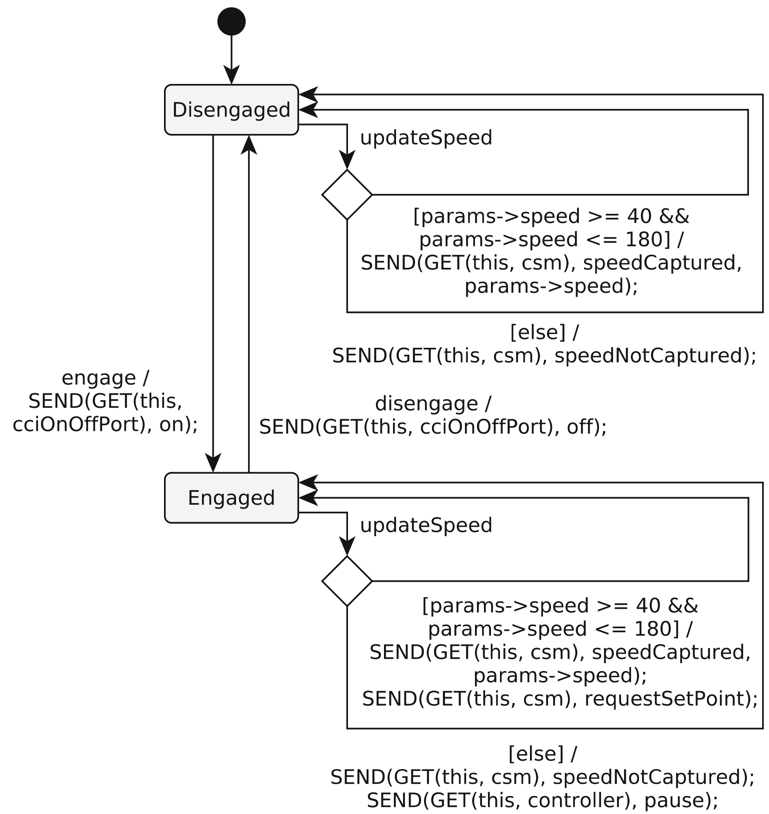


(a) State machine of the *Controller* class.

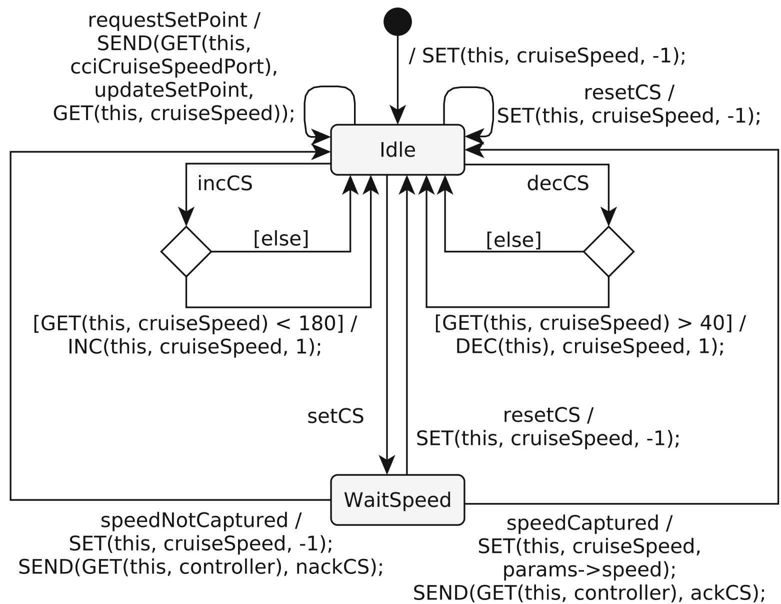


(b) State machine of the *PedalsManager* class.

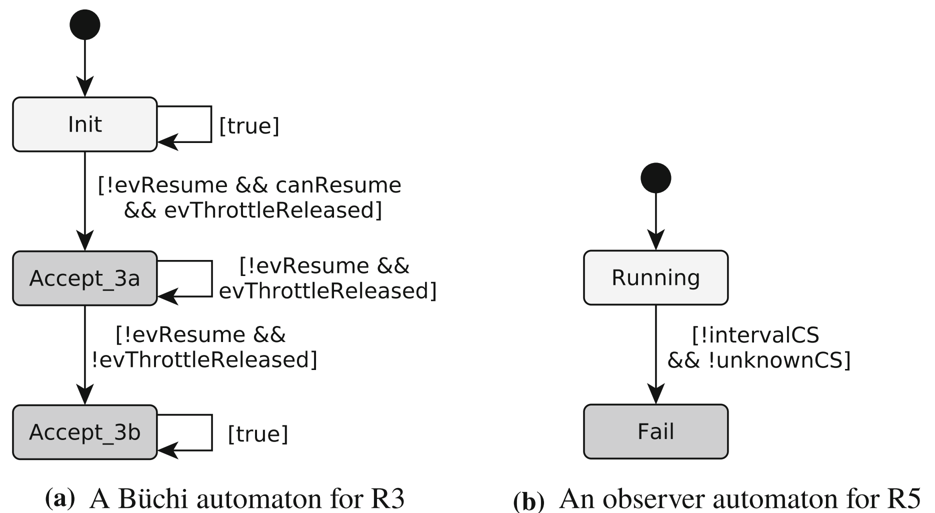
Fig. 12 continued



(c) State machine of the *Actuation* class.



(d) State machine of the *CruiseSpeedManager* class.



(a) A Büchi automaton for R3 **(b)** An observer automaton for R5

Fig. 13 State machines of PUSMs for the CCI

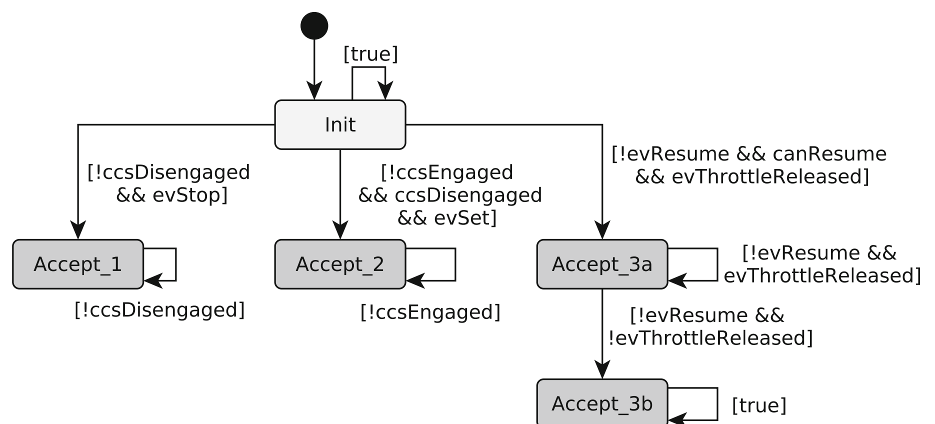


Fig. 14 Combination of all PUSMs in one that checks R1, R2, and R3

B Action and observation languages operators

This appendix presents operators of the action language used for system modeling and of the observation language used for property specification. These operators are brought together in four languages depending on whether they have reading or writing access to model attributes or internal runtime data of the model interpreter. These four languages are the expression language, the effect language, the expression language extension, and the effect language extension. The action language gathers operators of the expression language and of the effect language, while the observation language gathers operators of the expression language and its extension. All these languages rely on external data declared and defined in UML models. Figure 15 presents the metamodel of global references that can be used to refer to UML elements.

B.1 Expression language

The expression language is employed to model guards of state machine transitions. For this purpose, any C expression without side effect on model execution can be used. As shown in Fig. 16, four additional operators are added to C language constructs. *This* (with C macro `this`) is used to access the current execution context (typically the context of an object). *Get* is used to navigate the model (e.g., to get a model attribute). *At* is employed to access an element at a specific index (e.g., when a model attribute is a sequence). *Call* is used to call side-effect free methods. In the general case, the corresponding C macros can be obtained by putting the name of metaclasses in upper case and changing the definition style from camel case to snake case. For instance, `GET` is the macro associated with the *Get* metaclass.

B.2 Effect language

The effect language is used to perform actions as effects of state machine transitions. Any C statement can be used to perform this task. Figure 17 shows that several operators have also been defined : *Set* to assign a new value to a model attribute and *SetAt* to assign a new value at a specific index in a sequence. *Inc*, *IncAt*, *Dec*, and *DecAt* can be used to increment or decrement a model attribute. However, these

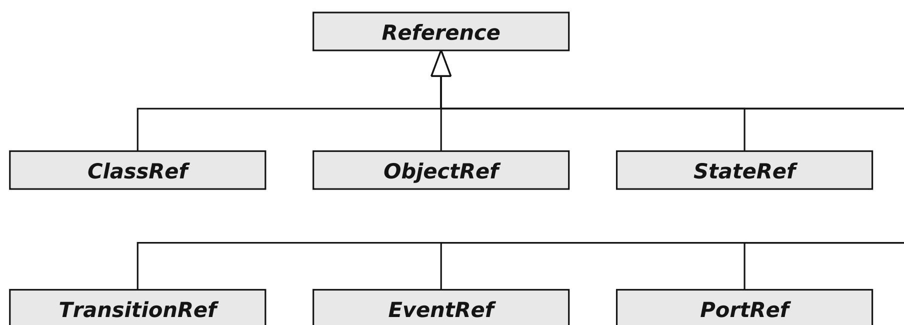


Fig. 15 Metamodel of references

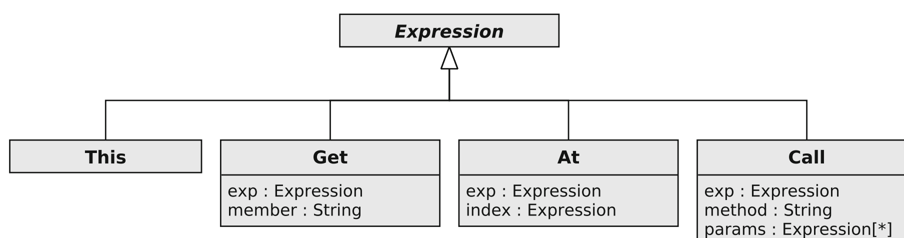


Fig. 16 Metamodel of additional operators of the expression language

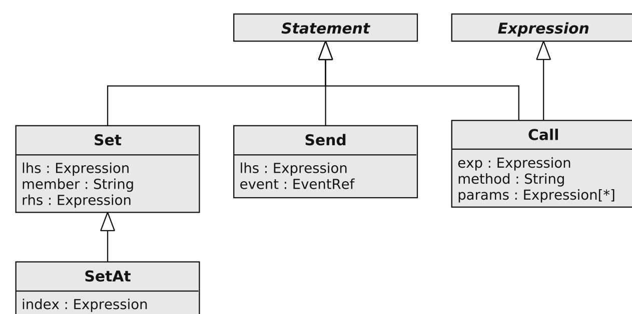


Fig. 17 Metamodel of additional operators of the effect language

four operators are not shown in Fig. 17 because they only provide syntactic sugar for both *Set* and *SetAt* operators. *Send* is used to send an event to another UML object and *Call* is used to call methods that have side-effects.

B.3 Expression language extension

Additional operators provided by the expression language extension are presented in Fig. 18. With this extension, it is possible to access the type of the current object, its state machine, its event pool (i.e., the set of events received by the object), and the current transition being fired. More specifically, it is possible to check if the current object is a given object (*IsObject*) or an instance of a given class (*IsTypeOf*). *IsInState* checks if the current state of an active object is a given state of its state machine. *TransitionExp* relates to the current transition being fired to know if its source (*TransitionHasSource*) or its target (*TransitionHasTarget*) is a given state, or if this transition is a given transition (*IsTransition*).

As shown in Fig. 19, several operators are also available to introspect the content of event pools. It is possible to know if the event pool is empty (*EpIsEmpty*) or full (*EpIsFull*), to get the number of events currently stored (*EpGetLength*), to get the first event (i.e., the oldest one) (*EpGetFirst*), to get the event at a given index (*EpGetAt*), and to get the last event (i.e., the newest one) (*EpGetLast*). Two other operators can also be used to check if the event pool of an object contains an occurrence of a specific event received on a given port (*EpContainsWithPort*), or on any port (*EpContains*).

Regarding more relaxed navigation rules, *ROOT_instMain* give access to the *Main* composite structure instance containing system objects. *ROOT_instProp* gives access to the *Prop* composite structure instance containing PUSMs. All system objects and PUSMs can be accessed from these two objects using eponymous C macros (i.e., `ROOT_instMain` and `ROOT_instProp`). *GetActivePeer* gives a direct access to active objects (e.g., objects of the environment) connected to the other end of communication links without the need to explicitly navigate the model through UML ports.

This extension also provides two operators to facilitate formal verification: *ObserverFail* to check if an observer automaton detects a failure in a given configuration, and *Deadlock* that becomes true when the system execution results in deadlock.

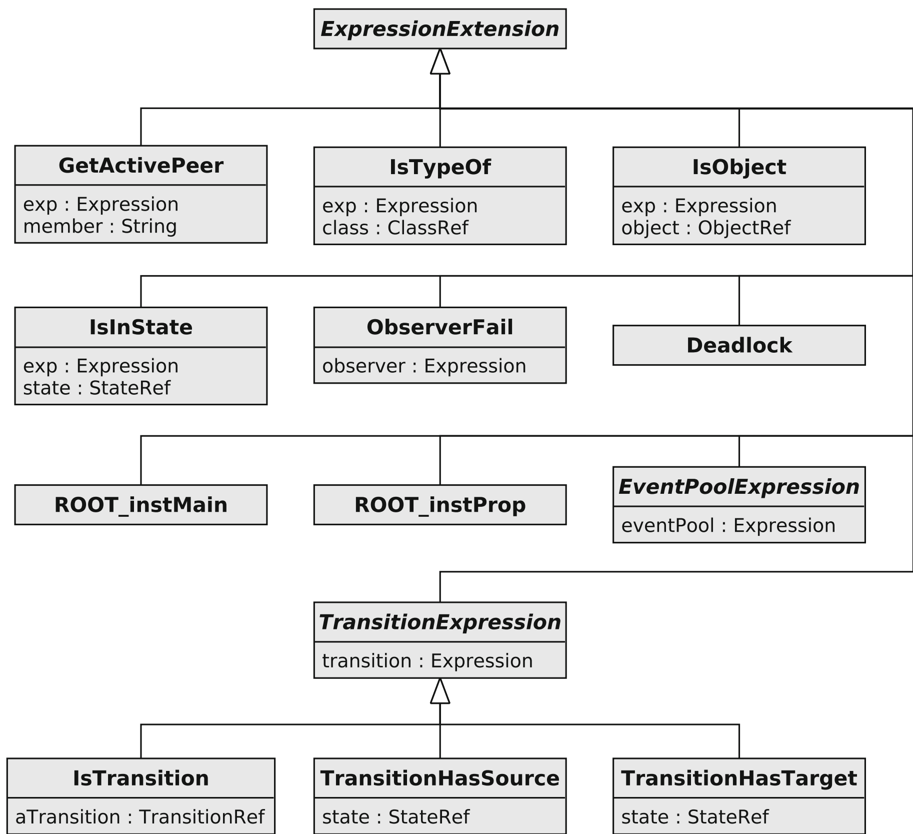


Fig. 18 Metamodel of additional operators of the expression language extension

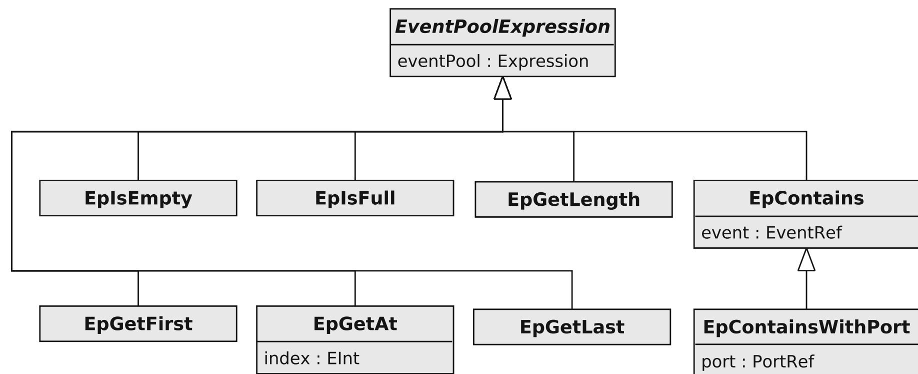


Fig. 19 Metamodel of event pool operators of the expression language extension

C List of atomic propositions

This section sums up all the atomic propositions that have been used in this paper for model verification. For each atomic proposition, we give its meaning in natural language (`[NL]`), its intuition in a pseudo-code (`[PC]`) with dotted notation, and the corresponding predicate expressed with our observation language (`[OL]`).

```
[NL] ccsDisengaged: check if the current state of the actuation
state machine is Disengaged

[PC] ccsDisengaged = "ROOT_instMain.cci.actuation.
IsInState(STATE_Actuation_Disengaged)"

[OL] ccsDisengaged = "IS_IN_STATE(GET(GET(ROOT_instMain, cci),
actuation), STATE_Actuation_Disengaged)"
```

[NL] ccsEngaged: check if the current state of the actuation state machine is Engaged

```
[PC] ccsEngaged = "ROOT_instMain.cci.actuation.
IsInState(STATE_Actuation_Engaged)"

[OL] ccsEngaged = "IS_IN_STATE(GET(GET(ROOT_instMain, cci),
actuation), STATE_Actuation_Engaged)"
```

[NL] evStop: check if the event pool of *csm* contains the *stop* signal

```
[PC] evStop = "ROOT_instMain.cci.csm.EpContains(SIGNAL_stop)"

[OL] evStop = "EP_CONTAINS(GET(GET(ROOT_instMain, cci),
csm), SIGNAL_stop)"
```

[NL] evSet: check if the event pool of *csm* contains the *set* signal

```
[PC] evSet = "ROOT_instMain.cci.csm.EpContains(SIGNAL_set)"

[OL] evSet = "EP_CONTAINS(GET(GET(ROOT_instMain, cci),
csm), SIGNAL_set)"
```

[NL] canResume: check if the *canResume* attribute of *pm* is TRUE

```
[PC] canResume = "ROOT_instMain.cci.pm.canResume == TRUE"

[OL] canResume = "GET(GET(GET(ROOT_instMain, cci), pm),
canResume) == TRUE"
```

[NL] evThrottleReleased: check if the event pool of *pm* contains a *pedalReleased* signal received on the *pmThrottlePedalPort* port

```
[PC] evThrottleReleased = "ROOT_instMain.cci.pm.
EpContainsWithPort(SIGNAL_pedalReleased,
PORT_PedalsManagerPedalPort_pmThrottlePedalPort)"

[OL] evThrottleReleased = "EP_CONTAINS_WITH_PORT(
GET(GET(ROOT_instMain, cci),pm), SIGNAL_pedalReleased,
PORT_PedalsManagerPedalPort_pmThrottlePedalPort)"
```

[NL] evResume: check if the event pool of *controller* contains the *resume* signal

```
[PC] evResume = "ROOT_instMain.cci.controller.
EpContains(SIGNAL_resume)"

[OL] evResume = "EP_CONTAINS(GET(GET(ROOT_instMain, cci),
controller), SIGNAL_resume)"
```

`[NL] evOff`: check if the first event in event pool of the object linked to *actuation* through *cciOnOffPort* is the *off* signal

```
[PC] evOff = "ROOT_instMain.cci.actuation.
GetActivePeer(cciOnOffPort).EpGetFirst() == SIGNAL_Off"
```

```
[OL] evOff = "EP_GET_FIRST(GET_ACTIVE_PEER(GET(GET(ROOT_instMain,
cci), actuation), cciOnOffPort)) == SIGNAL_Off"
```

`[NL] evOn`: check if the first event in event pool of the object linked to *actuation* through *cciOnOffPort* is the *on* signal

```
[PC] evOn = "ROOT_instMain.cci.actuation.
GetActivePeer(cciOnOffPort).EpGetFirst() == SIGNAL_On"
```

```
[OL] evOn = "EP_GET_FIRST(GET_ACTIVE_PEER(GET(GET(ROOT_instMain,
cci), actuation), cciOnOffPort)) == SIGNAL_On"
```

`[NL] evUpdateSetPoint`: check if the first event in event pool of the object linked to *actuation* through *cciOnOffPort* is the *updateSetPoint* signal

```
[PC] evUpdateSetPoint = "ROOT_instMain.cci.actuation.
GetActivePeer(cciOnOffPort).EpGetFirst() == SIGNAL_updateSetPoint"
```

```
[OL] evUpdateSetPoint = "EP_GET_FIRST(
GET_ACTIVE_PEER(GET(GET(ROOT_instMain, cci),
actuation), cciOnOffPort)) == SIGNAL_updateSetPoint"
```

`[NL] intervalCS`: check if the *cruiseSpeed* is in the [40, 180] km/h working interval

```
[PC] intervalCS = ROOT_instMain.cci.csm.cruiseSpeed >= 40
&& ROOT_instMain.cci.csm.cruiseSpeed <= 180"
```

```
[OL] intervalCS =
"GET(GET(GET(ROOT_instMain, cci), csm), cruiseSpeed) >= 40
&& GET(GET(GET(ROOT_instMain, cci), csm), cruiseSpeed) <= 180"
```

`[NL] unknownCS`: check if the *cruiseSpeed* attribute of *csm* is equal to -1

```
[PC] unknownCS = "ROOT_instMain.cci.csm.cruiseSpeed == -1"
```

```
[OL] unknownCS = "GET(GET(GET(ROOT_instMain, cci),
csm), cruiseSpeed) == -1"
```

`[NL] observerInFailState`: check if the *observer6* reaches a “fail” state

```
[PC] observerInFailState = "observer6.ObserverFail()"
```

```
[OL] observerInFailState = "OBSERVER_FAIL(observer6)"
```

`[NL] evStopController`: check if the event pool of *controller* contains the *stop* signal

```
[PC] evStopController = "ROOT_instMain.cci.controller.
EpContains(SIGNAL_stop)"
```

```
[OL] evStopController = "EP_CONTAINS(GET(GET(ROOT_instMain, cci),
controller), SIGNAL_stop)"
```

References

1. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Autom. Softw. Eng. **14**(3), 293–340 (2007). https://doi.org/10.1007/s10515-007-0012-6
2. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi automata translation: fast and more deterministic. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 95–109. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/3-540-44585-4_6
3. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). MIT Press, Cambridge (2008). https://doi.org/10.5555/1373322
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: EAGLE Does Space Efficient LTL Monitoring. Pre-Print CSPP-25 (2003)
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. **20**(4), 14:1-14:64 (2011). https://doi.org/10.1145/2000799.2000800
6. Besnard, V., Brun, M., Dhaussy, P., Jouault, F., Olivier, D., Teodorov, C.: Towards one model interpreter for both design and deployment. In: 3rd International Workshop on Executable Modeling (EXE 2017). Austin, United States (2017)
7. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Embedded UML model execution to bridge the gap between design and runtime. In: Mazzara, M., Ober, I., Salaün, G. (eds.) Software Technologies: Applications and Foundations, pp. 519–528. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_38
8. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Unified LTL verification and embedded execution of UML models. In: ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18). Copenhagen, Denmark (2018). https://doi.org/10.1145/3239372.3239395
9. Besnard, V., Teodorov, C., Jouault, F., Brun, M., Dhaussy, P.: A model checkable UML soccer player. In: 3rd Workshop on Model-Driven Engineering Tools, pp. 211–220. Munich, Germany (2019)
10. Besnard, V., Teodorov, C., Jouault, F., Brun, M., Dhaussy, P.: Verifying and monitoring uml models with observer automata. In: ACM/IEEE 22th International Conference on Model Driven Engineering Languages and Systems (MODELS '19), pp. 161–171. Munich, Germany (2019). https://doi.org/10.1109/MODELS.2019.000-5
11. Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V.: Model checking flight control systems: the Airbus experience. In: 2009 31st International Conference on Software Engineering—Companion Volume, pp. 18–27 (2009). https://doi.org/10.1109/ICSE-COMPANION.2009.5070960
12. Bodden, E., Lam, P., Hendren, L.: Clara: a framework for partially evaluating finite-state runtime monitors ahead of time. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification, pp. 183–197. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_15
13. Boniol, F., Wiels, V.: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study, pp. 1–18. Springer, Cham (2014)
14. Brumbulli, M., Gaudin, E., Teodorov, C.: Automatic verification of BPMN models. In: 10th European Congress on Embedded Real Time Software and Systems (ERTS 2020). Toulouse, France (2020)
15. Chen, F., D'Amorim, M., Roşu, G.: A formal monitoring-based framework for software development and analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) Formal Methods and Software Engineering, pp. 357–372. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30482-1_31
16. Ciccozzi, F.: From models to code and back: a round-trip approach for model-driven engineering of embedded systems. Mälardalen University, Embedded Systems. Ph.D. thesis (2014)
17. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of UML models: a systematic review of research and practice. Softw. Syst. Model. (2018). https://doi.org/10.1007/s10270-018-0675-4
18. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction—CADE-25, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
19. Dhaussy, P., Le Roux, L., Teodorov, C.: Vérification formelle de propriétés : Application de l'outil OBP au cas d'étude CCS. Génie logiciel **109** (2014)
20. Diot, C., de Simone, R., Huitema, C.: Communication Protocols Development Using ESTEREL (1994)
21. Drusinsky, D.: The temporal rover and the ATG rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN Model Checking and Software Verification, pp. 323–330. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/10722468_19
22. Duret-Lutz, A., Poitrenaud, D.: SPOT: an extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS '04, pp. 76–83. IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/MASCOT.2004.1348184
23. Ferretti, J., Di Pietro, L., De Maria, C.: Open-source automated external defibrillator. HardwareX **2**, 61–70 (2017). https://doi.org/10.1016/j.ohx.2017.09.001
24. Gaiser, A., Schwoon, S.: Comparison of algorithms for checking emptiness on Büchi automata. In: Hlinený, P., Matyáš, V., Vojnar, T. (eds.) Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09), OpenAccess Series in Informatics (OASIcs), vol. 13, pp. 18–26. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2009). https://doi.org/10.4230/DROPS.MEMICS.2009.2349
25. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification, pp. 53–65. Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_6
26. Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous observers and the verification of reactive systems. In: Nivat, M., Rattray, C., Rus, T., Scollo, G. (eds.) Algebraic Methodology and Software Technology, vol. AMAST'93, pp. 83–96. Springer, London (1994). https://doi.org/10.1007/978-1-4471-3227-1_8
27. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.P., Stevens, P. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 342–356. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
28. Havelund, K., Roşu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Transf. **6**(2), 158–173 (2004). https://doi.org/10.1007/s10009-003-0117-6
29. Havelund, K., Roşu, G.: Monitoring Java programs with Java pathexplorer. Electronic Notes in Theoretical Computer Science **55**(2), 200–217 (2001). https://doi.org/10.1016/S1571-0661(04)00253-1. RV'2001, Runtime Verification (in connection with CAV '01)
30. Hili, N., Bagherzadeh, M., Jahed, K., Dingel, J.: A model-based architecture for interactive run-time monitoring. Softw. Syst. Model. (2020). https://doi.org/10.1007/s10270-020-00780-y
31. Holzmann, G.J., Joshi, R.: Model-driven software verification. In: Graf, S., Mounier, L. (eds.) Model Checking Software, pp. 76–91. Springer, Berlin, Heidelberg (2004)

32. Iyenghar, P., Pulvermueller, E., Westerkamp, C., Wuebbelmann, J., Uelschen, M.: Model-Based Debugging of Embedded Software Systems, pp. 107–132. Springer, New York (2017). https://doi.org/10.1007/978-1-4614-2266-2_5
33. Jouault, F., Besnard, V., Le Calvar, T., Teodorov, C., Brun, M., Delatour, J.: Designing, animating, and verifying partial UML models. In: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), MODELS '20. Virtual Event, Canada (2020). https://doi.org/10.1145/3365438.3410967
34. Jouault, F., Delatour, J.: Towards fixing sketchy UML models by leveraging textual notations: application to real-time embedded systems. In: Brucker, A.D., Dania, C., Georg, G., Gogolla, M. (eds.) OCL 2014, OCL and Textual Modeling: Applications and Case Studies, vol. 1285, pp. 73–82. Valencia, Spain (2014)
35. Jouault, F., Teodorov, C., Delatour, J., Le Roux, L., Dhaussy, P.: Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD. Génie Logiciel **109**, 21–27 (2014)
36. Kane, A.: Runtime Monitoring for Safety-Critical Embedded Systems (2015). https://doi.org/10.1184/R1/6721376.v1
37. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance approach for Java programs. Form. Methods Syst. Des. **24**(2), 129–155 (2004)
38. Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In: Damm, W., Olderog, E.R. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 395–414. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45739-9_23
39. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: Kühne, T. (ed.) Models in Software Engineering, pp. 42–51. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69489-2_6
40. Kripke, S.A.: Semantical analysis of modal logic i normal modal propositional calculi. Math. Logic Q. **9**(5–6), 67–96 (1963). https://doi.org/10.1002/malq.19630090502
41. Leroux, L., Delatour, J., Dhaussy, P.: Modélisation UML d'un régulateur de vitesse automobile. Génie Logiciel **109**, (2014)
42. Liu, S., Liu, Y., Sun, J., Zheng, M., Wadhwa, B., Dong, J.S.: USMMC: a self-contained model checker for UML state machines. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 623–626. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2491411.2494595
43. Mekki, A., Ghazel, M., Toguyeni, A.: Validating Time-constrained Systems Using UML Statecharts Patterns and Timed Automata Observers, vol. VECoS'09, pp. 112–124. BCS Learning & Development Ltd., Swindon, UK (2009)
44. Ober, I., Graf, S., Ober, I.: Validation of UML models via a mapping to communicating extended timed automata. In: Graf, S., Mounier, L. (eds.) Model Checking Software, pp. 127–145. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24732-6_9
45. Ober, I., Graf, S., Ober, I.: Validating timed UML models by simulation and verification. Int. J. Softw. Tools Technol. Transf. **8**(2), 128–145 (2006). https://doi.org/10.1007/s10009-005-0205-x
46. OMG: Action Language for Foundational UML (Alf) (2017). www.omg.org/spec/ALF/1.1/PDF
47. OMG: Precise Semantics of UML State Machines (2017). https://www.omg.org/spec/PSSM/1.0/Beta1/PDF
48. OMG: Semantics of a Foundational Subset for Executable UML Models (2017). https://www.omg.org/spec/FUML/1.3/PDF
49. OMG: Unified Modeling Language (2017). https://www.omg.org/spec/UML/2.5.1/PDF
50. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: a hard real-time runtime monitor. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification, pp. 345–359. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_26
51. Queille, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR, pp. 216–230. Springer, Berlin, Heidelberg (2008)
52. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 596–610. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
53. Roşu, G., Bensalem, S.: Allen linear (interval) temporal logic—translation to LTL and monitor synthesis. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, pp. 263–277. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11817963_25
54. Roşu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: this time with calls and returns. In: Leucker, M. (ed.) Runtime Verification, pp. 51–68. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89247-2_4
55. Tan, L., Kim, J., Sokolsky, O., Lee, I.: Model-based testing and monitoring for hybrid embedded systems. In: Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004, pp. 487–492 (2004)
56. Teodorov, C., Dhaussy, P., Le Roux, L.: Environment-driven reachability for timed systems. Int. J. Softw. Tools Technol. Transf. **19**(2), 229–245 (2017). https://doi.org/10.1007/s10009-015-0401-2
57. Teodorov, C., Le Roux, L., Drey, Z., Dhaussy, P.: Past-Free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis. Softw. Test. Verif. Reliab. **26**(7), 516–542 (2016). https://doi.org/10.1002/stvr.1611.Stvr.1611
58. Torres Lopez, C., Gurdeep Singh, R., Marr, S., Gonzalez Boix, E., Scholliers, C.: Multiverse Debugging: Non-deterministic Debugging for Non-deterministic Programs. ECOOP. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.27

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Valentin Besnard is a R&D engineer at Ateme, a software editor specialized in video compression and delivery. In 2020, he obtained his PhD degree from ENSTA Bretagne, Brest, France, and in collaboration with ESEO and Davidson. His research focuses on model execution and formal verification of UML models with an embedded model interpreter. Contact him at valentin.besnard.49@gmail.com

Ciprian Teodorov is an associate professor at ENSTA Bretagne, Brest, France. He is a member of the MOCS team of the Lab-STICC Research Laboratory CNRS UMR 6285. His research interests include formal verification and diagnosis of critical embedded-systems, model-driven engineering, and Domain-Specific Language (DSL) execution. In the past, Ciprian was software engineer at Dolphin Integration, contributing to the design of a high-performance mixed-signal circuit simulation infrastructure. For his work on model-driven physical-design for emerging nanoscale computing, in 2011, Ciprian received a PhD degree in Computer Science from the University of Western Brittany, France. He joined ENSTA Bretagne in 2013, and now leads the development of the OBP2 modular model-checking and xDSL diagnosis toolkit. Contact him at ciprian.teodorov@ensta-bretagne.fr, or visit https://www.ensta-bretagne.fr/teodorov.

Frédéric Jouault is a research associate at ESEO, France. He received his PhD from the University of Nantes before doing a postdoc at the University of Alabama at Birmingham. His research interests involve model engineering, transformation, synchronization, and execution, as well as their application to Domain-Specific Languages (DSLs) and model-based reverse engineering. Frédéric created ATL, a DSL for model-to-model transformation. He is now leading the development of ATL (language and toolkit) on Eclipse.org, and is in charge of the Eclipse modeling MMT project as well as a member of the modeling PMC. Contact him at frederic.jouault@eseo.fr.

Matthias Brun is an associate professor with the Department of Software and Systems, ESEO, Angers, France. His research interests include model-driven software engineering with a particular interest in model transformation and model interpretation for embedded systems. He received a PhD in computer science from the University of Nantes, Nantes, France, in 2010. Contact him at matthias.brun@eseo.fr.

Philippe Dhaussy is professor at CNRS Lab-STICC within ENSTA Bretagne. His expertise and his research interests include model-driven software engineering, formal validation for real time systems and embedded software design. He has an engineering degree in computer science from ISEN (French Institute of Electronics and Computer Science) in 1978 and received his PhD in 1994 at Telecom Bretagne (France) and his HDR in 2014. From 1980 to 1991, he had been software engineer and technical coordinator in consulting companies (Atlantide group), mainly in real-time system developments. He joined ENSTA-Bretagne in 1996, as professor. He has over 60 publications in the areas of software engineering and computer science. He has been co-supervisor for five PhD students, has been and is involved in several research projects as work package coordinator. Contact him at philippe.dhaussy@ensta-bretagne.fr.