HAL open science

Verifying and Monitoring UML Models with Observer Automata: A Transformation-Free Approach

Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Philippe Dhaussy

To cite this version:

Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Philippe Dhaussy. Verifying and Monitoring UML Models with Observer Automata: A Transformation-Free Approach. 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, 2019, Munich, Germany. pp.161-171, 10.1109/MODELS.2019.000-5 . hal-02433749

HAL Id: hal-02433749

https://hal.science/hal-02433749v1

Submitted on 21 Apr 2020

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

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

Distributed under a Creative Commons Attribution 4.0 International License

Verifying and Monitoring UML Models with Observer Automata

A Transformation-free Approach

Valentin Besnard
ERIS, ESEO-TECH
Angers, France
valentin.besnard@eseo.fr

Ciprian Teodorov
Lab-STICC UMR CNRS 6285
ENSTA Bretagne, Brest, France
ciprian.teodorov@ensta-bretagne.fr

Frédéric Jouault
ERIS, ESEO-TECH
Angers, France
frederic.jouault@eseo.fr

Matthias Brun
ERIS, ESEO-TECH
Angers, France
matthias.brun@eseo.fr

Philippe Dhaussy
Lab-STICC UMR CNRS 6285
ENSTA Bretagne, Brest, France
philippe.dhaussy@ensta-bretagne.fr

***Abstract*—The increasing complexity of embedded systems renders verification of software programs more complex and may require applying monitoring and formal techniques, like model-checking. However, to use such techniques, system engineers usually need formal experts to express software requirements in a formal language. To facilitate the use of model-checking tools by system engineers, our approach consists of using a UML model interpreter with which the software requirements can directly be expressed as observer automata in UML as well. These observer automata are synchronously composed with the system, and can be used unchanged both for model verification and runtime monitoring. Our approach has been evaluated on the user interface model of a cruise control system. The observer verification results are in line with the verification of equivalent LTL properties. The runtime overhead of the monitoring infrastructure is 6.5%, with only 1.2% memory overhead.**

***Index Terms*—Observer Automata, Monitoring, Model Interpretation, Embedded Systems**

I. Introduction

In the context of embedded cyber-physical systems, the design of software components becomes increasingly more complex. Such complexity exposes software programs to several potential software failures (e.g., design faults, bugs, security flaws) that are more intricate to detect, understand, and fix. With Model-Driven Engineering (MDE), software systems can be designed using models and verified with formal verification techniques during early design phases. However, even if these techniques seem to give promising results, they may not be sufficient to detect all bugs and design faults. Offline verification is not always applicable due to state-space explosion problem. Moreover, it requires 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, an increasing number of embedded systems rely on runtime monitoring, as the last resort, for detecting runtime failures and triggering safe system recovery procedures.

To perform these activities, system requirements must be expressed in a formal language (e.g., LTL). Based on the formal safety properties, one technique [HR02], [BLS11] aims at synthesizing monitors to take advantage of the complementarity between model verification and runtime monitoring. However, two model transformations are usually required. The first one is used to transform LTL safety properties into monitors at model-level, which can be used for model verification. A second transformation is also required to specialize these monitors on the embedded target for runtime monitoring. 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 the system execution but require a model tranformation as well as code instrumentation to achieve it. Second, the use of formal verification techniques remains a complex task for system engineers that do not have a formal background. In particular, expressing properties in a formal language may be difficult for them without the help of formal methods experts.

To address these issues, our approach aims at using synchronous observer automata for verifying and monitoring embedded system software. This work extends our Embedded Model Interpreter (EMI) presented in [BBD+17], [BBJ+18a], [BBJ+18b]. This tool can be used to execute, simulate, and verify embedded systems, specified as UML [OMG17] models, with a unique implementation of the language semantics. Prior work on this model interpreter shows how

to perform simulation, trace generation, and LTL model-checking activities, but misses the importance of runtime monitoring. This paper addresses this shortcoming by focusing on verifying and monitoring safety properties encoded into the design language (here UML) as synchronous observer automata. The main contributions of this paper are *(1)* the use of UML observer automata, rather than LTL, for expressing and verifying system requirements on UML models and *(2)* the deployment of the same observer automata on real embedded targets for runtime monitoring. To reach this goal, observer automata are synchronously composed with the system during model execution. For model verification, the OBP2 model-checker [TLRDD16], [TDLR17] (https://plug-obp.github.io/) is connected to the interpreter to check that properties encoded by observer automata are satisfied. For runtime monitoring, these observer automata monitor the current execution trace of the actual system.

To validate our approach, a UML model of a cruise control user interface has been designed and analyzed. The safety system requirements have been modeled as observer automata with UML state machines and verified with the OBP2 model-checker. Results obtained are identical to results of model-checking equivalent LTL properties. The same observer automata have been deployed on a STM32 discovery board to monitor execution of the system. In addition to the cost of monitors, resource overhead induced by monitoring is only of 6.5% for execution time and 1.2% for memory footprint. This seems acceptable for embedded systems execution depending on the context of each system.

The remainder of this paper is structured as follows. Section II describes the cruise control interface used as example. An overview of the approach is given in Section III. Then, Section IV explains how safety properties can be expressed as observer automata while in Section V, we detail the process used for model verification and runtime monitoring. In Section VI, we presents the results of applying the approach to our example. Section VII reviews the state of the art and we finally conclude this paper in Section VIII.

II. Motivating Example

To illustrate our approach on an actual embedded system, we consider the user interface of a Cruise Control System (CCS) as example. The CCS automatically controls the speed of a vehicle by adjusting the throttle position to maintain a steady speed as set by the driver. For this example, we focus our design and verification efforts on the user interface, that we call Cruise Control Interface (CCI), because this subsystem contains most of the control logic of the CCS. This motivating example has been designed for the purpose of this paper and is partially based on related works [DLRT14], [LDD14] as well as past experiences of some of the authors on this kind of systems.

Figure 1 presents a component diagram of a CCS showing its interactions with the driver and the physical vehicle. This component diagram has been designed with the assumption

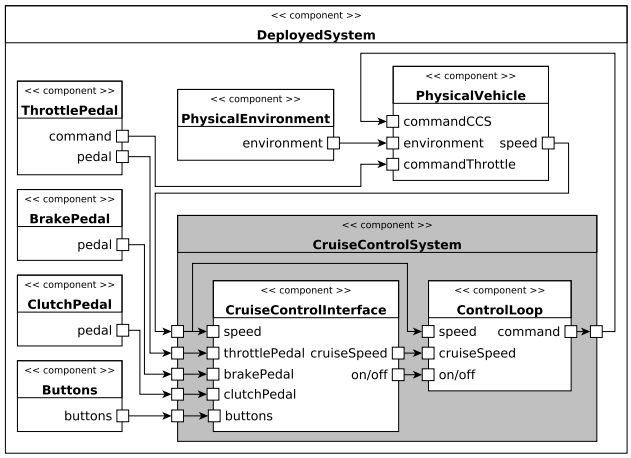


Figure 1: Component diagram of a cruise control system.

that the CCS operates independently of any other systems (e.g., Electronic Stability Program (ESP)).

As illustrated on this diagram, the CCS interacts with the physical process to control, named here *PhysicalVehicle*. This component takes as inputs both the command from the *CruiseControlSystem* and the command from the *ThrottlePedal* to control automatically or manually the vehicle engine. The last input comes from the *PhysicalEnvironment* that may apply some forces on the vehicle (e.g., road profile, air friction) and disrupt the vehicle driving. The *PhysicalVehicle* is also equipped with a sensor that measures the current speed. The data captured is given as input to the *CruiseControlSystem*.

The *CruiseControlSystem* is composed of two components: the *CruiseControlInterface* and the *ControlLoop*. The former is responsible for managing all inputs received by the CCS: all-or-nothing data from the three pedals of the vehicle to know if each pedal is pressed or released, and data from buttons on which the user can press to control the CCS. The *CruiseControlInterface* aims at driving the *ControlLoop* of the CCS in charge of adjusting the speed of the vehicle to the cruise speed calculated by the CCI. The *ControlLoop* is considered here as a black box that executes a control algorithm for computing the command to apply on the *PhysicalVehicle* engine.

In this paper, we focus on the verification of the CCI subsystem. Hence, all components external to the CCI are considered as the environment of this subsystem. Figure 1 has helped us to have a better understanding of this environment and make a relevant abstraction of it for the verification step.

To apply our approach to this motivating example, we have designed a UML model of a CCI. The composite structure diagram of this model is shown in Figure 2. The *Main* class is the root composite class of the model. An instance specification named *instMain* is associated to this class and used by the model interpreter to instantiate it and all its containing parts. 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.

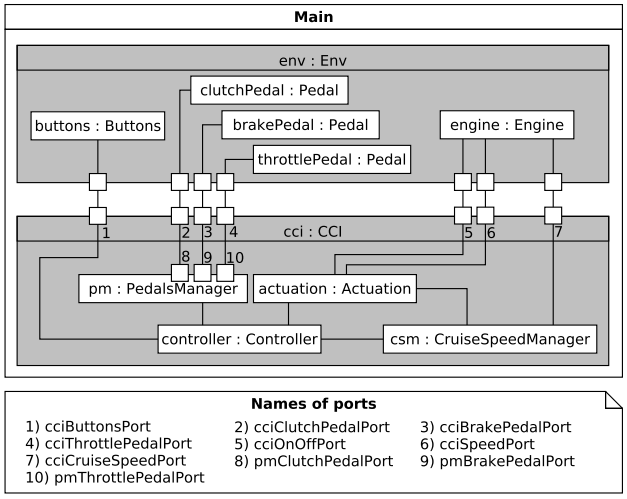
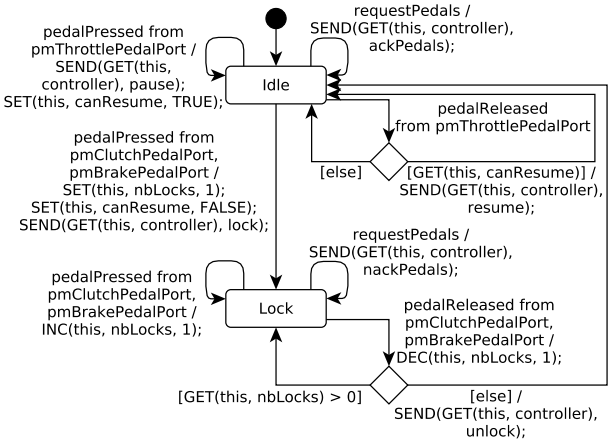


Figure 2: Composite structure diagram of the CCI model.

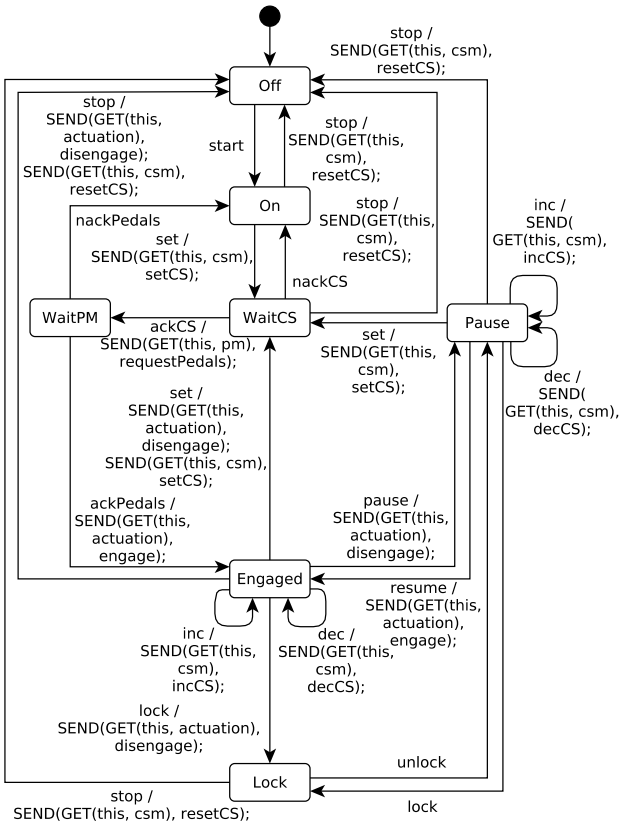
Environment abstraction. The *env* part contains a *buttons* object that models the different buttons (i.e., start, stop, inc, dec, set, pause, resume) that can be manipulated, as well as the three pedals (i.e., *clutchPedal*, *brakePedal*, *throttlePedal*) that can be pressed or released by the driver. According to Figure 1, both the *PhysicalVehicle* and the *ControlLoop* are also parts of the environment. In our UML model, they have been abstracted into 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. For this purpose, the *controller* receives events from *buttons* and from the pedals manager (*pm*), which ensures a first processing of pedals events. Based on these events, the *controller* determines the status of the CCS and delegates generation of output events to both *actuation* and cruise speed manager (*csm*) objects. The *actuation* 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 (*csm*) 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. The behavior of all these objects is defined by state machines that are presented in Figure 3 for further details. On 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.

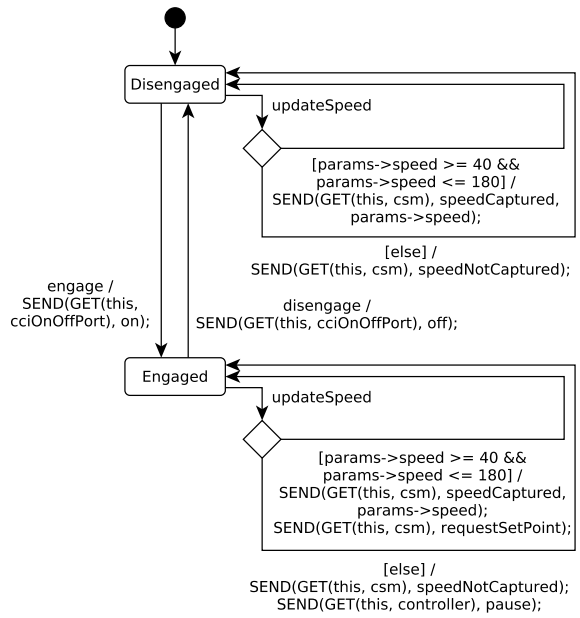
Using this CCI model, the goal of this paper is to demonstrate that we can verify formal properties using observer automata. For this purpose, we have selected three system requirements to check the validity of our UML model:



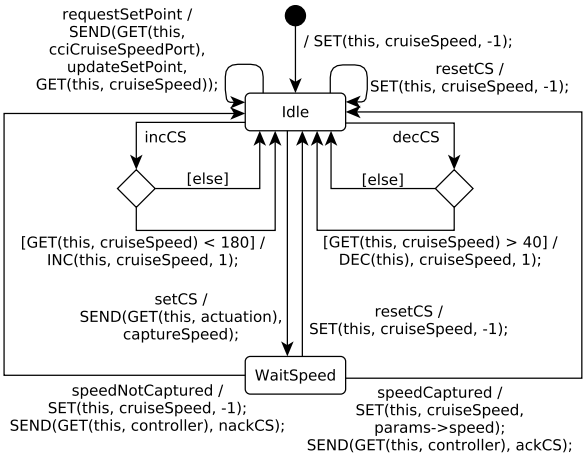
(a) State machine of the *PedalsManager* class.



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



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



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

Figure 3: State machines of the CCI model.

1) 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.
2) The cruise speed should not be below 40 km/h or above 180 km/h.
3) When the system is engaged, the cruise speed should be defined.

III. Contribution Overview

To better understand the scope of this work, this section overviews our approach describing the integration of the observer-based verification and monitoring infrastructure with an existing UML model interpreter [BBJ+18b]. One main contribution of this work is to show that observer automata can be used for model verification and runtime monitoring without the cost of model transformations.

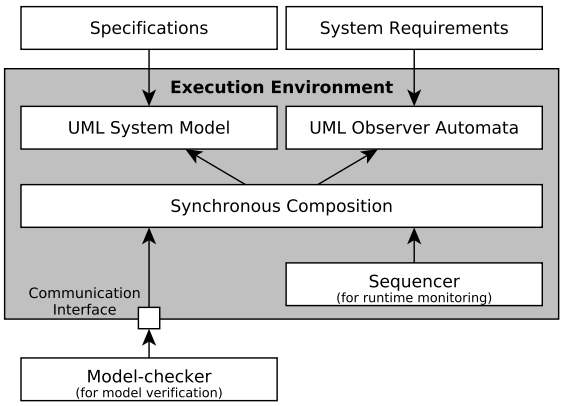


Figure 4: Approach Overview.

According to [OGO06], observer automata can be defined as special processes that monitor the changes in the *state* of a model (e.g., attributes values, contents of event pools, current state of state machines) and the *events* occurring in it (e.g., signal events). These automata can be used to express safety properties by specifying "fail" states to denote states that must be reached when the encoded property is violated. The expressivity of such automata is limited to safety properties but this limitation simplifies the use of model-checking techniques by reducing the verification problem to a reachability problem.

An overview of the observer automata verification architecture, applied for UML in our context, is illustrated in Figure 4. First, a model of the system (*UML System Model*) is designed in UML from its *Specifications*. To verify and monitor the model behavior, the *System Requirements* are encoded as *UML Observer Automata*, which are observer automata designed in UML with state machines. They can access the state of the system model using the action language of the UML model interpreter. In our case, this action language is used to specify guards and effects of state machine transitions, and provides C macros to access UML instances of the system model and their attributes. Then, a *Synchronous Composition* algorithm is applied to compose synchronously observer automata with the UML model of the system. This means that each time a transition is fired in the system model, a transition will also be fired on each observer automata. As a result, the observer automata are closely following model execution. This setup can be controlled by a model-checker connected to the communication interface of the interpreter to perfom model verification. For the verification procedure, the model-checker uses a reachability algorithm that checks if any of the observer automata has reached a "fail" state. For runtime monitoring, though, a *Sequencer* which runs the main execution loop is connected to the *Synchronous Composition* component.

This approach offers multiple benefits to perform model analysis. *(1)* No model transformation is required to transform

observer automata, used during model-checking, into runtime monitors. What is checked during model verification is exactly what is monitored at runtime. *(2)* This technique stays compatible with classical approaches that synthesize monitors from formal properties, but requires the generation of UML observer automata. *(3)* Our approach facilitates the use of verification tools by system engineers. Even if they are not formal experts, system engineers can easily specify UML observer automata, since they only rely on concepts of the design language. *(4)* Verification results returned by the model-checker are directly expressed in terms of UML concepts, which facilitate their understanding by engineers.

IV. Expressing Properties with UML Observer Automata

This section presents how the properties can be expressed in UML as observer automata and briefly discusses their expressivity.

A. UML Observer Automata

In this work, observer automata are expressed directly in UML using the same UML subset as the one used for the system model as well as an extension of the C action language to facilitate model navigation and verification. These observer automata defined in UML will be called UML observer automata (or simply observers) in the rest of this paper. In the design model, each UML observer automaton is an instance of an active UML class whose behavior is described with a UML state machine. All these UML observer automata are instantiated as parts of a composite class called *Obs* and its instance specification *instObs*. This class is similar to the *Main* class of the system and is used as a root composite structure for observer automata instantiation.

The state machines of these active classes are classical UML state machines but additional constraints must be satisfied to express properties in a formal way. First, UML observer automata are not supposed to interact with objects of the system model (e.g., send or receive events) but only observe changes in the state of this model. Second, UML observer automata must be deterministic and complete (i.e., exactly one transition must be fireable at each instant). The *determinism constraint* ensures that observer automata do not introduce non-determinism in the system but simply 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 ensured by the synchronous composition operator by completing observer automata with implicit loop transitions (stuttering steps).

The goal of these observer automata is to detect failures in the system behavior. For this purpose, UML state machines used for observer automata need to specify "fail" states that must be reached if a failure is detected. In our approach, the "fail" states are specified by the user as state invariants. Different other techniques can be adopted, like the use of a UML profile with a stereotype for "fail" states, but these improvements are beyond the scope of this study.

Another key point in the definition of UML observer automata is the definition of transition guards with the C action language. These guards are used to specify how the corresponding state machine goes from one state to another and potentially reaches "fail" states when the system behavior is in failure. For this purpose, UML observer automata require an extension of the action language to access UML objects of the system and their properties (e.g., current state of state machine, values of attributes). This action language extension has similar requirements to the one used in our previous work [BBJ+18b] for LTL verification. In this study we introduce some additional operators (C macros), which provides more relaxed rules for model navigation as well as facilities for model verification. The most important action language operators are: *(i)* `ROOT_instMain` and `ROOT_instObs` macros gives access to instances of root composite structures. All system objects and observer automata can be accessed from these two objects. *(ii)* Several C macros are available to introspect the content of event pools (i.e., the set of events received by an active object). For instance, the `EP_GET_FIRST` macro gets the first event of the event pool. *(iii)* The `IS_IN_STATE` macro checks if the current state of an active object is a given state of its state machine. *(iv)* Unlike the `GET` macro that returns a property of an object, `GET_ACTIVE_PEER` gives access to actual active objects (e.g., objects of the environment) connected to the other end of communication links. *(v)* The `OBSERVER_FAIL` macro is used to check if an observer automaton reaches one of its "fail" states in a given configuration (i.e., the execution state at a given time instant).

As an example, the three safety properties of the CCI motivating example have been expressed as UML observer automata. State machines of these observer automata are described in Figure 5. Each of them defines a "fail" state named *Fail* that will be reached in case of failure. For sake of simplicity, transition guards of these state machines have been defined using predicates with labels. All these predicates are expressed with the C action language of the UML model interpreter, and its extension for formal verification. Some of these predicates are:

- `evOn`: check if the first event in event pool of the object linked to *actuation* through *cciOnOffPort* is the *On* signal

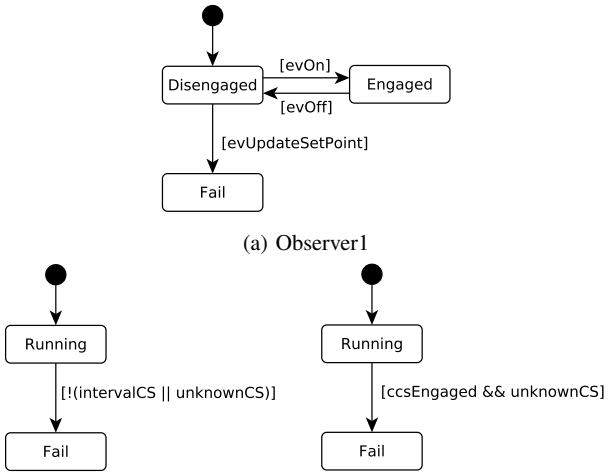
  ```
  evOn = "EP_GET_FIRST(GET_ACTIVE_PEER(
  GET(GET(ROOT_instMain, cci), actuation),
  cciOnOffPort)) == SIGNAL_On"
  ```

- `ccsEngaged`: check if the current state of the *actuation* state machine is *Engaged*

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

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

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



(a) Observer1

(b) Observer2

(c) Observer3

Figure 5: State machines of observer automata for the CCI.

B. Expressivity of Observer Automata

Runtime monitoring enables the detection of failures during the execution of the deployed system. As such, the verification capabilities are restricted to the analysis of the current run of the system, as opposed to model-checking that analyses all runs. This partial observation of the running system limits the expressivity of the verified properties to *monitorable properties* [BLS11].

In general, the class of monitorable properties can be captured by (deterministic) Finite State Machines (FSM). The complete and deterministic UML observer automata, used in this study, are a syntactic extension of simple FSM enabling, mainly, the use of variables to reduce the number of explicitly named states. However, their expressivity remains limited to the class of monitorable properties. Nevertheless, this constraint can also be seen as a benefit during verification, because the use of observer automata reduces the model-checking problem to a reachability-checking problem. Indeed, the model-checker has only to check if "fail" states of observer automata are reached in at least one configuration during the analysis. If none of the "fail" states is reached, the property expressed by the observer automaton is verified.

For offline verification, FSMs, and thus observer automata, are not sufficiently expressive to encode all linear time properties and in particular liveness properties, which express that something good will eventually happens [BLS11]. For instance, the CCI requirement "When a *stop* event has been received, the CCI will eventually be stopped" is a liveness property that cannot be captured with an observer automaton. In contrast to safety properties that are violated in a limited time (the violation witness is a prefix of the execution trace), liveness properties are violated in infinite time (the violation witness is an infinite trace that does not allow the required eventuality to happen). Dedicated automata formalisms (e.g., Büchi automata) as well as more complex model-checking algorithms (cycle detection) are required to detect violations of these properties [DLP04].

V. Verification and Monitoring with UML Observer Automata

Based on UML observer automata, this section describes the verification process used to verify safety properties with a model-checker and the UML model interpreter. This approach can be applied during the verification step but also for runtime monitoring by deploying the same UML observer automata on the actual target.

A. Synchronous Composition

The essential concept of the verification and monitoring process is the synchronous composition of UML observer automata with the system model. The software architecture used to achieve this synchronous composition is illustrated in Figure 6. On this component diagram, the design model is divided in two parts: one related to the system (*UML System Model*) interpreted by the *System Interpreter* and the other related to observer automata (*UML Observers Model*) interpreted by the *Observers Interpreter*. Thus, the UML interpreter is instantiated twice such that the same UML semantics is used for both parts. From the system model, a Kripke structure is built using the *Kripke Adaptor* component. The Kripke structure gives an abstraction of the global system automaton for a given set of atomic propositions (or atoms) provided by the *Property Proxy* component. In our approach, each transition guard of an observer automaton is considered as an atomic proposition for formal verification. This Kripke structure can then be used to build a finite automaton using

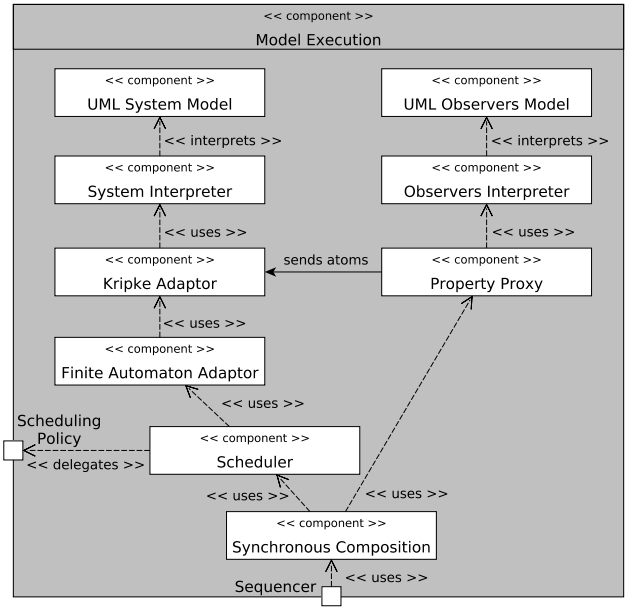


Figure 6: Component used for synchronous composition of observer automata with the system model.

the *Finite Automaton Adaptor*. It is important to note that both adaptors (*Kripke Adaptor* and *Finite Automaton Adaptor*) do not generate the whole state-space statically, but instead they rely on the semantics provided by the *System Interpreter* to create the required views dynamically.

Based on atom valuations, the *Synchronous Composition* component is able to compose synchronously observer automata with the finite automaton of the system. This operation is performed by dynamically building synchronous transitions. A synchronous transition is composed of one and only one transition of the system as well as one transition per observer automaton. This means that for each step of the system (i.e., for each transition fired), each observer automaton will also make a step. An observer automaton fires an explicit transition if one outgoing transitions of its current state is fireable. Otherwise, the observer automaton will fire an implicit self-transition to ensure the completeness requirement. The computation of a synchronous transition is made by firing in advance the system transition, evaluating atomic propositions in the target configuration, and determining for each observer automaton the transition that can be used for synchronization.

In Figure 6, a *Scheduler* is operating on the *Finite Automaton Adaptor* because the finite automaton that results from the composition of active objects of the system 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 all this *Model Execution* component. It will request to compute synchronous fireable transitions or to fire one of them at given time points. The sequencer may be the user through a user simulation interface or a software algorithm.

B. Verification of Safety Properties

With the *Model Execution* component, it becomes possible to connect a model-checker to the UML model interpreter for verifying safety properties encoded by UML observer automata. An important prerequisite for applying model-checking techniques on a model is to close the model with a proper abstraction of the system environment. This is the reason why for our example, we pay attention to understand as much as possible the context in which the CCI operates for modeling a relevant abstraction of it.

The software architecture used for model verification is exposed in Figure 7. The *Model Execution* component is connected to an abstraction of the scheduling policy (*Scheduling Policy Abstraction*) to consider a superset of all possible cases. A suitable abstraction, very general abstraction, is to return all fireable transitions of the system to explore all the model state-space. In this case, the scheduler does not make any choice and the synchronous composition is applied to a set of system transitions rather than only one. The verification is thus independent of the scheduling algorithm used. However, if the scheduling policy is known, it remains possible to use it for model verification.

The model-checker is connected to the *Model Execution* component through the *Language Server* that provides facilities for the verification task including atomic propositions compilation. Since the verification problem is reduced to a reachability problem (cf. Section IV), the *Reachability Algorithm* will be the main sequencer of the verification process. It explores all the model state-space using the *States Stream* component that manages states already explored and stored in the *State-space Storage*. To perform this operation, the model-checker also needs to interact with the *Execution Environment*. For this purpose, the *Proxy Runtime* exposes an interface that can be used for communicating with the *Execution Environment*, through the view exposed by the *Synchronous Composition* component. This interface enables the model-checker to get and set the current configuration of the execution environment (as described in [BBJ+18b]). Each time a new configuration is found, the *Observers Asserting* component checks if observer automata have reached one of their "fail" states before returning this configuration to the *States Stream*. As soon as an observer automaton reaches one of its "fail" states, 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 all safety properties encoded by UML observer automata are verified.

From a user point of view, the software architecture of the model-checker is configured as in Figure 7 for verifying simple LTL properties of the form:

```
"[] !|OBSERVER_FAIL(obs)|"
```

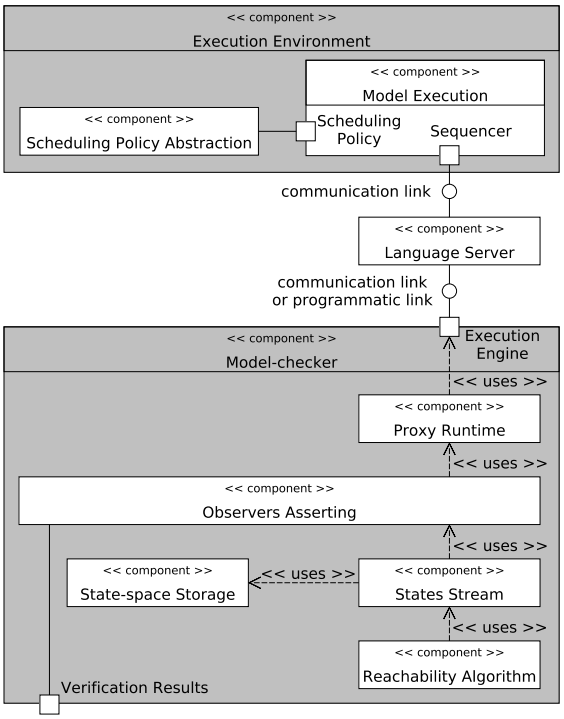


Figure 7: Model-checking of safety properties.

where `obs` is a UML observer automaton. This kind of property is called a state-invariant. They can be easily expressed in LTL and automatic generation of these invariants can be performed.

Despite the fact that synchronous composition is not standard UML, this approach offers multiple benefits for system engineers. Indeed, multiple observer automata can be composed synchronously with the system, which gives the possibility to check any number of safety properties simultaneously. Our approach also 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 [OGO06], [OGO04], [Cic14].

C. Runtime Monitoring

Once the model verification has been performed, the UML model can be deployed on the actual embedded target. To continue verification of safety properties at runtime, it is possible to embed UML 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 Figure 8. The same *Model Execution* component used during model verification is reused for monitoring. However this time, 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. In this case, the synchronous composition has only to compute one synchronous transition, which is more efficient than doing it for all fireable transitions. Since in monitoring only one execution path is covered, an optimization has been performed on the synchronous composition to keep efficient monitoring performance. Indeed, contrary to model verification, there is no need to change the current configuration of the model interpreter (i.e., its current memory state). Fireable transitions are always computed in the current configuration of the model interpreter and the next transition to fire is always fired from this configuration. The target configuration of the fired transition is considered as the current configuration for the next execution step. To drive the *Model Execution* component, the actual *Sequencer* of the system is linked to it. The *Sequencer* represents the main execution loop. For each step, three main operations are performed. First, it computes the next synchronous transition to fire. Then, it fires this transition. Finally, it delegates the verification of safety properties to the *Observers Asserting* component. This last component checks if UML observer automata reach one of their "fail" states and updates the *Monitoring Status*.

One main advantage of our approach is that the same observer automata used during the verification phase can be deployed on the target and reused for runtime monitoring

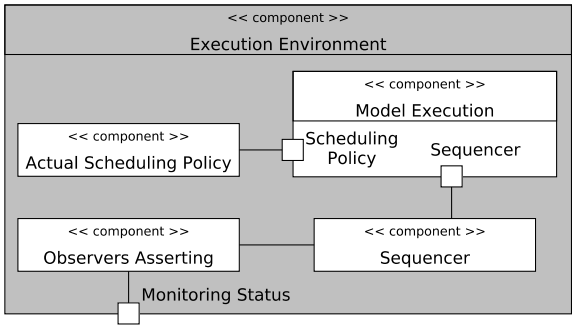


Figure 8: Runtime monitoring of safety properties.

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, observer automata can simply notify the problem to the user (e.g., by printing an error message) or it can activate the appropriate fail-safe controllers (e.g., for 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 verification 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.

VI. Experiments and Results

Our approach has been applied to a UML model of a CCI, the user interface of a cruise control system, introduced in Section II. These experiments aim at evaluating our approach for checking the validity of the three system requirements expressed for the CCI. The verification of safety properties as UML observer automata will be made using model verification with the OBP2 model-checker and results will be compared with verification results of identical properties expressed in LTL. These observer automata will also be deployed on an actual embedded target to perform runtime monitoring and measure the induced overhead.

Model-checking. As shown in Figure 5, system requirements have been expressed as UML observer automata. Following the setup in Figure 7, these observer automata have been loaded in the *Execution Environment* with OBP2 as model-checking component. To check that they do not reach their "fail" states, the following LTL invariants where used:

1) `"[] !|OBSERVER_FAIL(obs1)|"`
2) `"[] !|OBSERVER_FAIL(obs2)|"`
3) `"[] !|OBSERVER_FAIL(obs3)|"`

The verification performed with a reachability algorithm shows that both properties 1 and 2 are verified while property 3 is violated. To check the validity of our approach, we have compared these results with model-checking results of identical safety properties expressed in LTL. For this purpose, all system requirements have been specified in LTL such as:

1) `"[] (|evOff| and !|evOn| -> (!|evUpdateSetPoint| W |evOn|))"`
2) `"[] (|intervalCS| or |unknownCS|)"`
3) `"[] (|ccsEngaged| -> !|unknownCS|)"`

These properties link the atomic propositions, which are directly evaluated on the UML model interpreter, with different LTL operators: not (`!`), or (`or`), and (`and`), globally (`[]`), weak until (`W`), and implies (`->`). Atomic propositions involved in these LTL properties are defined such as `|atom|` where `atom` is one of the labelled predicates defined for transition guards of UML observer automata.

As a result, expressions of these properties in LTL are more complex to achieve because it requires the knowledge of LTL operators and especially path operators (e.g., globally, weak until). In particular, the first safety property which is not a state-invariant is not trivial to express in the LTL formalism while the corresponding observer automaton (Figure 5a) is simple to design. Model-checking of these three LTL properties with OBP2 reports that properties 1 and 2 are verified while property 3 is violated. These LTL-based results are identical to those obtained with observer automata.

To understand why property 3 is violated, the counterexample returned by OBP2 has been analyzed. This counterexample shows that when the *controller* of the CCI goes from the *Engaged* state to the *Off* state, one *disengage* event is sent to the *actuation* and one *resetCS* event is sent to the cruise speed manager (*csm*). However, the scheduler can choose to process the *resetCS* event first and this will set the cruise speed at an undefined value (i.e., -1) while the *actuation* is still in the *Engaged* state. To fix this design error, we have added a state in the *Controller* state machine to send first the *disengage* event, expect the acknowledge of the *actuation* in this new state, and finally send the *resetCS* event. A new iteration of verification experiments show that all properties are now verified with observer automata and classical LTL model-checking. This fixed CCI model has a state-space containing 46,444,386 configurations linked by 82,734,350 transitions.

Monitoring. Once verified, the UML model of the CCI has been deployed with the embedded model interpreter on a STM32 discovery board. For experiment purpose and because we have not got a real CCI, the UML model has been deployed on the embedded target with a simulated environment (cf. Figure 2) and not with the physical sensors and actuators. For simplicity, the simulated environment matches with the environment abstraction used for verification. In this case, objects of the environment are also managed by the scheduling policy and the sequencer of the *Execution Environment*. These objects send events to the CCI according to their abstract behavior (not detailed in this paper for lack of space). The UML observer automata have been deployed with the UML model of the CCI to perform runtime monitoring following the setup in Figure 8. No failure has been detected on the corrected version of the UML model. These results are consistent with results obtained through model-checking. However, the deployment of the initial version of the model has shown that the third observer automaton would have succeeded to detect the failure if it had occurred.

In terms of performance, runtime monitoring induces resource overheads comparing 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 induced by N monitors in terms of execution time is given by the following equation:

$$overhead \approx 6.5 + \frac{1}{nb_ao}\sum_{i=1}^{N}\frac{nb_states_i}{nb_outgoings_i}$$

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 $\frac{nb_states}{nb_outgoings}$ 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 UML observer automaton decreases as the size of the system model increases.

VII. Related Work

Multiple other works use observer automata to specify and verify system requirements. One typical approach applies model transformation techniques for converting UML observer automata to the automaton formalism used by the verification tool [MGT09]. A similar work [OGO06], [OGO04] 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 [KMR02], [KW07] used by Hugo/RT aims at transforming interaction diagrams into observer automata for checking that UML state machines interact accordingly 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.

Regarding monitoring, the work in [HR04] define two techniques to analyze Java programs by checking if a trace of events satifisfied LTL properties. The first one is based on a rewriting-based framework that allows to define new logics for monitoring execution traces. The second one [HR02] aims at synthesizing monitors for safety properties by generating efficient code from LTL formulas. This last technique is also used in [BLS11] to focus on runtime verification of LTL properties by analyzing finite prefixes of infinite traces. Furthermore, monitoring can be achieved by tracing model execution. In [IPW+17], a debugger uses an embedded monitor to produce back annotated traces and build UML diagrams in real-time for visualizing the model behavior. In the same way, the project in [Cic14] aims at monitoring extra-functional properties using annotations in the UML design model and back-propagation of analysis results to this model. In comparison to our work, all these approaches rely on model transformations, and sometimes on code instrumentation, to monitor system execution. Our approach avoids such techniques to ensure that observer automata used for monitoring are exactly the same as the ones used during model verification.

The synchronous language Lustre [HLR94] can be used to describe reactive systems and express safety properties using synchronous observer automata. In [BVWW09], 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.

Finally, we noticed that none of the related works mention the possibility to perform monitoring by directly deploying UML observer automata on actual embedded systems while this opportunity is offered by our tool. We are also 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.

VIII. Conclusion

Runtime monitoring provides interesting facilities to check system requirements on models of embedded systems. This activity offers a good complementarity to model verification for detecting additional failures, triggering safety techniques (e.g., system recovery), and assisting engineers in post-mortem analysis. The approach presented in this paper is based on the use of observer automata to unify both model verification and runtime monitoring of UML models. Execution of these models relies on a UML model interpreter with a single implementation of the operational semantics.

With this technique, safety properties can be expressed as UML observer automata to monitor model execution. These observer automata can directly be embedded in the UML model interpreter and are synchronously composed with the system model all along model execution. They can be used for model verification by checking exhaustively, with a model-checker, that properties encoded by these observer automata are not violated. The same UML observer automata can also be used for runtime monitoring without the need of costly model transformations or code instrumentation to deploy these monitors on embedded targets. As a result, what is monitored at runtime is exactly what is checked during model verification.

This approach uses the same language for both design model and property specification, thus facilitating the use of formal verification techniques by the system engineers. Furthermore, this approach facilitates the analysis of verification results because they are directly captured within the UML formalism.

The approach was evaluated on a UML model of a cruise control interface. The results show that safety properties can be easily expressed using our UML observer automata while the verification results are equivalent with LTL model-checking. The deployment of the UML observer automata on a STM32 embedded board induces an 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.

Currently, the guards in the UML observer automata can only be synchronized with state-based expressions, however their expressivity can be extended to state-event expressions as proposed in [HR04], [GM05]. Future work also includes the integration of other model-based specification formalism such as Property Sequence Chart (PSC) [AIP07] based on an extension of UML 2.0 interaction diagrams.

Acknowledgment

This work is partially funded by Davidson Consulting. The authors especially thank David Olivier for his advice and industrial feedback.

References

[AIP07] M. Autili, P. Inverardi, and P. Pelliccione. Graphical Scenarios for Specifying Temporal Properties: an Automated Approach. *Automated Software Engineering*, 14(3):293–340, September 2007. doi:10.1007/s10515-007-0012-6.

[BBD+17] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov. Towards one Model Interpreter for Both Design and Deployment. In *3rd International Workshop on Executable Modeling (EXE)*, Austin, United States, September 2017.

[BBJ+18a] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Embedded UML Model Execution to Bridge the Gap Between Design and Runtime. In *MDE@DeRun 2018 : First International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems*, Toulouse, France, June 2018.

[BBJ+18b] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. 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, October 2018. doi:10.1145/3239372.3239395.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64, September 2011. doi:10.1145/2000799.2000800.

[BVWW09] Thomas Bochot, Pierre Virelizier, Helene Waeselynck, and Virginie Wiels. Model Checking Flight Control Systems: The Airbus Experience. In *31st International Conference on Software Engineering - Companion Volume*, pages 18–27, May 2009. doi:10.1109/ICSE-COMPANION.2009.5070960.

[Cic14] Federico Ciccozzi. *From Models to Code and Back : A Round-trip Approach for Model-driven Engineering of Embedded Systems*. PhD thesis, Mälardalen University, Embedded Systems, 2014.

[DLP04] Alexandre Duret-Lutz and Denis Poitrenaud. 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, pages 76–83, Washington, DC, USA, 2004. IEEE Computer Society.

[DLRT14] Philippe Dhaussy, Luka Le Roux, and Ciprian Teodorov. Vérification formelle de propriétés : Application de l'outil OBP au cas d'étude CCS. *Génie logiciel*, 109, June 2014.

[GM05] Stefania Gnesi and Franco Mazzanti. A Model Checking Verification Environment for UML Statecharts. In *Proceedings of XLIII Congresso*, 2005.

[HLR94] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous Observers and the Verification of Reactive Systems. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, pages 83–96, London, 1994. Springer London. doi:10.1007/978-1-4471-3227-1_8.

[HR02] Klaus Havelund and Grigore Roşu. Synthesizing Monitors for Safety Properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-46002-0_24.

[HR04] Klaus Havelund and Grigore Roşu. Efficient Monitoring of Safety Properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, August 2004. doi:10.1007/s10009-003-0117-6.

[IPW+17] Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, Juergen Wuebbelmann, and Michael Uelschen. *Model-Based Debugging of Embedded Software Systems*, pages 107–132. Springer New York, New York, NY, 2017. doi:10.1007/978-1-4614-2266-2_5.

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking Timed UML State Machines and Collaborations. In Werner Damm and Ernst Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 395–414, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. doi:10.1007/3-540-45739-9_23.

[KW07] Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In Thomas Kühne, editor, *Models in Software Engineering*, pages 42–51, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69489-2_6.

[LDD14] Luka Leroux, Jérôme Delatour, and Philippe Dhaussy. Modélisation UML d'un régulateur de vitesse automobile. *Génie logiciel*, 109, June 2014.

[MGT09] Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating Time-constrained Systems Using UML Statecharts Patterns and Timed Automata Observers. In *Proceedings of the 3rd International Conference on Verification and Evaluation of Computer and Communication Systems*, VECoS'09, pages 112–124, Swindon, UK, 2009. BCS Learning & Development Ltd.

[OGO04] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML Models via a Mapping to Communicating Extended Timed Automata. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, pages 127–145, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-24732-6_9.

[OGO06] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *International Journal on Software Tools for Technology Transfer*, 8(2):128–145, Apr 2006. doi:10.1007/s10009-005-0205-x.

[OMG17] OMG. Unified Modeling Language, December 2017. https://www.omg.org/spec/UML/2.5.1/PDF.

[TDLR17] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, 19(2):229–245, Apr 2017. doi:10.1007/s10009-015-0401-2.

[TLRDD16] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. Past-Free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability*, 26(7):516–542, 2016. doi:10.1002/stvr.1611.