Software and Systems Modeling
(Author's Accepted Manuscript version)

A Model-Based Architecture for Interactive Run-Time Monitoring

Nicolas Hili · Mojtaba Bagherzadeh · Karim Jahed · Juergen Dingel

Received: 13 June 2019 / Revised: 17 November 2019 / Accepted: 3 January 2020

Abstract We present a model-based architecture for monitoring executions of models of real-time and embedded systems. This architecture is highly configurable and allows for the combination of various run-time monitoring tools, not only for observing the system execution, but also for interacting with it. Using a variety of case studies, we illustrate the use of the architecture for connecting the code generated from a model with a range of external tools for different purposes, including execution animation and run-time verification. However, the external tool can not only consume information from the execution, but also generate input for it and thus influence and steer it.

1 Introduction

Run-time monitoring is a fundamental technique for debugging and analysing modern real-time systems as the collection of run-time information during execution can be used to ensure the correctness of the system and to detect flaws that could not be easily detected at design-time [19, 8]. For instance, run-time tracing and monitoring is central to quality assurance, because it can reveal bugs, performance problems, and security breaches, and validate service-level agreements and optimizations. However, it can also drive animations that provide suitable views to, e.g., system operators, trainees, or potential customers. When the execution can not only be observed by an external component, but also receive input from it, additional possibilities open up: for instance, the external component can host a simulation that is updated via monitored events and provides input to the execution about, e.g., expected sensor or

Nicolas Hili · Mojtaba Bagherzadeh · Karim Jahed · Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
{hili, mojtaba, jahed, dingel}@cs.queensu.ca

user inputs allowing the system to prepare. But, the external component can also be given explicit control over the structure and behaviour of the system by allowing it to issue, e.g., reconfiguration or reset commands and thus causes the monitored system to adapt to changing environmental conditions observed in the trace. The ability to connect the system effectively with different external tools thus also appears to be very useful for supporting many of the capabilities that, e.g., future 'smart' cyber-physical systems are envisioned to have including self-adaptation, self-optimization, and even self-correction [7].

Research question: The question addressed in this paper is: *How can this wide range of monitoring and interaction scenarios and their associated tools be supported in the context of the model-driven engineering of real-time embedded (RTE) systems?*

Contribution: To answer this question, we present a model-based architecture that supports the combined use of several different run-time monitoring tools, not only for observing the system execution, but also for providing it with input and even steering it. The proposed architecture is highly configurable, and supports configuration (i.e., the connection of external monitoring tools) at design-time (static) and at run-time (dynamic). Since the validation of real-time constraints is a core concern, the architecture strives to minimize the impact of the monitoring and interaction on system performance.

We present a prototype implementation based on the modelling language UML for Real-Time (UML-RT) and the open source MDE tool Papyrus-RT [38]. The efficacy of the prototype is demonstrated using a performance experiment and three different case studies involving a parcel router and a rover, i.e., a small vehicle equipped with a Raspberry Pi board, motors and sensors.

An Early Vision: In the vision paper [13], we suggest the use of customizable code generation to support a wide range of activities aimed at determining or improving the quality of the models, or supporting communication with different stakeholders including other developers, students, trainees, and current or potential customers.

The heart of the approach lies in a customizable context configuration that describes the information exchanged between the running system and any external tools. More precisely, the configuration specifies to what extent aspects of the system execution are observable to external components, what information external components can send to the system, and how the system will communicate with each external component. The glue code necessary to realize the described interaction is generated automatically. The approach aims to be flexible and applicable in a wide range of situations, while hiding some of the implementation details of the communication and minimizing manual work. The envisioned approach is illustrated using a model of a failover system which, at runtime, is connected with the Papyrus-RT tool to provide a simple execution animation by displaying the current state of the system and with the Linux Trace Toolkit Next Generation (LTTng) tracing tool to detect violations of timing constraints.

In the present paper, we reformulate and refine a part of this vision, present a completely new prototype implementation with model-level and IDE support, describe two novel case studies, and summarize the results of an experimental study to determine the costs associated with the use of the prototype.

Paper Structure: The paper is structured as follows: Section 2 provides background; Sections 3 and 4 present the architecture, its implementation, and different ways of using and adapting it; Section 5 evaluates the suitability of the architecture for real-time systems by describing an experiment for measuring the performance overhead; Section 6 details some applications in which the architecture has been used; Section 7 discusses the current status of the implementation and future work; Section 8 presents related work; Section 9 concludes.

2 UML for Real-Time (UML-RT)

UML-RT [41] is a UML profile for the development of RTE systems. It features a light syntax and has a small, cohesive set of concepts. Figs. 1 and 2 show a simplified version of the UML-RT metamodel. A core concept are *capsules*, which are similar to actors [1]. A capsule is a composite structure which can contain typed *attributes*, typed *ports*, and *parts* (or instances) of other capsules. It may have a user-defined behaviour modelled using a *state machine*. A state machine (cf. Fig. 2) models how the capsule can interact with other capsules by means of *message exchange*. State machines in UML-RT models are inherited from Unified Modeling Language (UML) with some constraints, e.g., no parallel region and no *do* activity, implicit history states, etc. States's entry/exit action code and transitions' effects are usually expressed in C++ directly, or using some action languages [18].

A *protocol* (cf. Fig. 1) formally defines all the messages a capsule can receive (incoming messages) or send (outgoing messages) through its ports. Two capsules can be connected if they both own a port sharing the same protocol, and if one of the two ports is *conjugated*. A conjugated port simply inverts incoming and outgoing messages in the definition of the protocol.

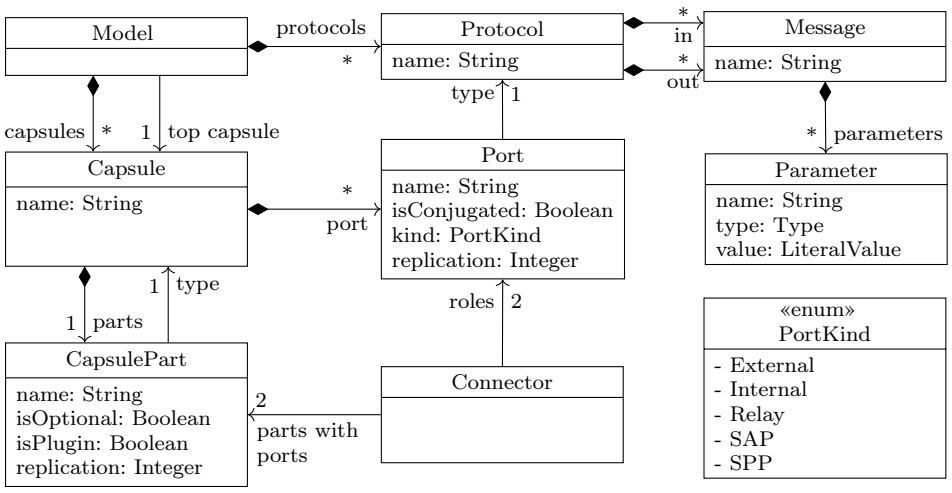


Fig. 1 Simplified illustration of the UML-RT metamodel (structure). For readability purpose, capsule attributes and the composition relation between a capsule and a connector are not shown.

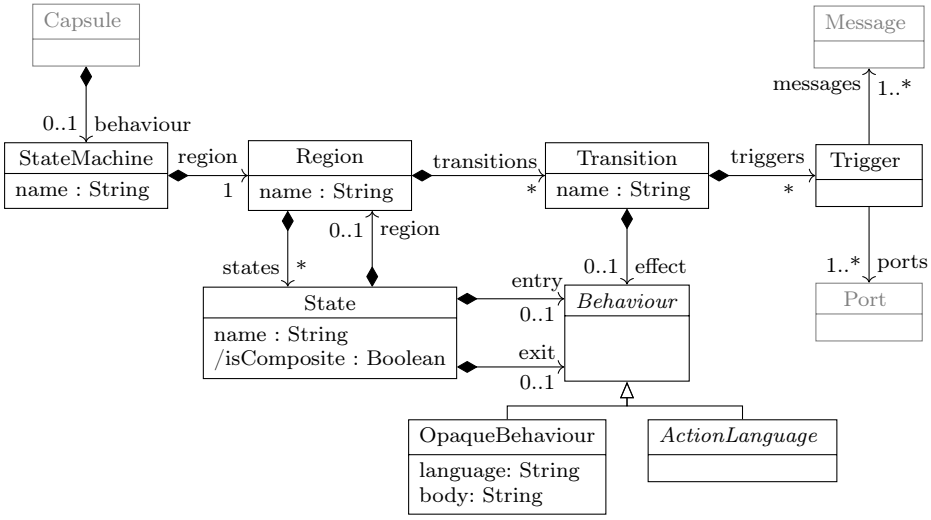


Fig. 2 Simplified illustration of the UML-RT metamodel (behaviour). Semi-transparent concepts are shown in Fig. 1. For readability purpose, relations between states and transitions (inherited from UML) are not shown.

UML-RT supports different kinds of ports (See *PortKind* in Fig. 1). *External* (resp. *internal*) *ports* allow a capsule to receive messages from and send messages to external (resp. internal) capsules, while *relay ports* merely forward messages from an external capsule to an internal one. Therefore, relay ports are important to ensure encapsulation in UML-RT. Finally, UML-RT also defines *Service Access Point (SAP)* and *Service Provision Point (SPP)* ports [42]. A SPP port can be virtually connected to SAP ports if they share the same protocol and the same name. This is particularly useful when defining application and platform models. Indeed, a platform can be seen as a provider of services while the application consumes those services. SAP and SPP ports allow for a dynamic binding when a capsule is instantiated.

The two next sections present the architecture for interactive run-time monitoring and its implementation. While our implementation is UML-RT-specific, our contribution can be applicable to other component-based and state-based modelling languages. In particular, any modelling languages complying with the Precise Semantics Of UML Composite Structures (PSCS) [35] and the Precise Semantics of UML State Machines (PSSM) [34] standards could benefit from our architecture. Therefore, the two next sections detail our contribution using the UML-RT terminology. General applicability to other modelling languages is discussed in Section 4.3.

3 Interactive Run-Time Monitoring

To be maximally useful, support for connecting external tools with the execution of the code generated from the models should satisfy the following requirements:

A range of different external tools should be supported including tools for data collection, animation, simulation, analysis, adaptation, and control. These tools may or may not be collocated with the running system. For instance, resource constraints

may require the use of remote communication via, e.g., TCP/IP. On the other hand, the detection of the violation of timing constraints may require very fast communication via, e.g., shared memory. Also, at different times different tools may be interested in different information ranging from information about the state of the system (e.g., currently active capsules, states, and timers), the execution (e.g., transitions taken, messages and timeouts received, capsule instances or connectors created), to the use of resources (e.g., number of active capsules and timers, size of message queues or data structures). Finally, external tools should also be able to provide different kinds of inputs to the system including sensor or user inputs, simulation or analysis results, or execution or reconfiguration commands.

Fig. 3 shows the realization of the vision [13]. From a user-defined model, Model-to-Model (M2M) transformation rules are used to translate a user-defined model M into an instrumented model M' from which code can be generated using standard code generators. M2M transformation rules are used to add an observability layer to the instrumented model. Once the code is generated, its execution can be observed by any external monitoring tools thanks to the observability layer. A key difference with [13] is that the approach presented in this paper relies on M2M transformations rather than on code generation extensions to add observability capabilities to a user-defined model. As stated in [3], this makes our approach completely independent of any implementation or code generator. The generation of the observability layer is fully automatic and transparent to the developer.

To facilitate the specification of the interaction between the code and external monitoring tools, we have extended the hierarchy of monitorable events (cf. Fig. 4) we proposed in [13] in order to make it less dependent to a specific modelling language. Each event is represented by an *event source* and an *event type*. Fig. 4 shows the list of events, structured into two levels of hierarchies: the first level (*signal*, *method*, ...) shows the event sources while the second level (*signal send*, *method call*, ...) shows

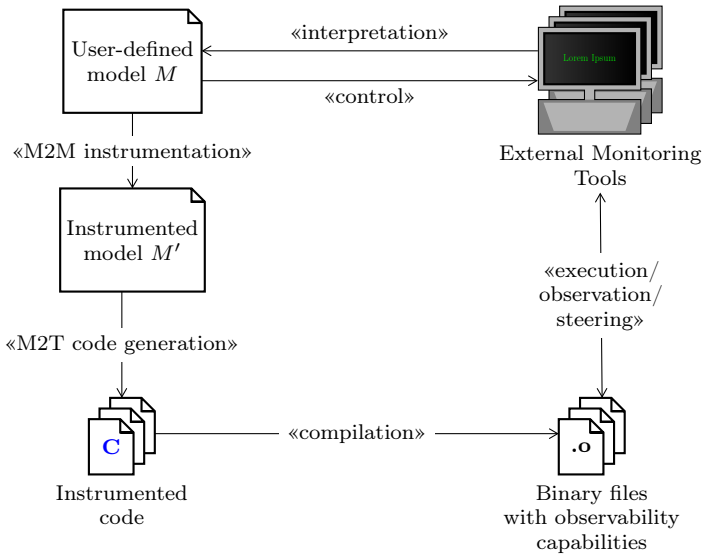


Fig. 3 Observation process.

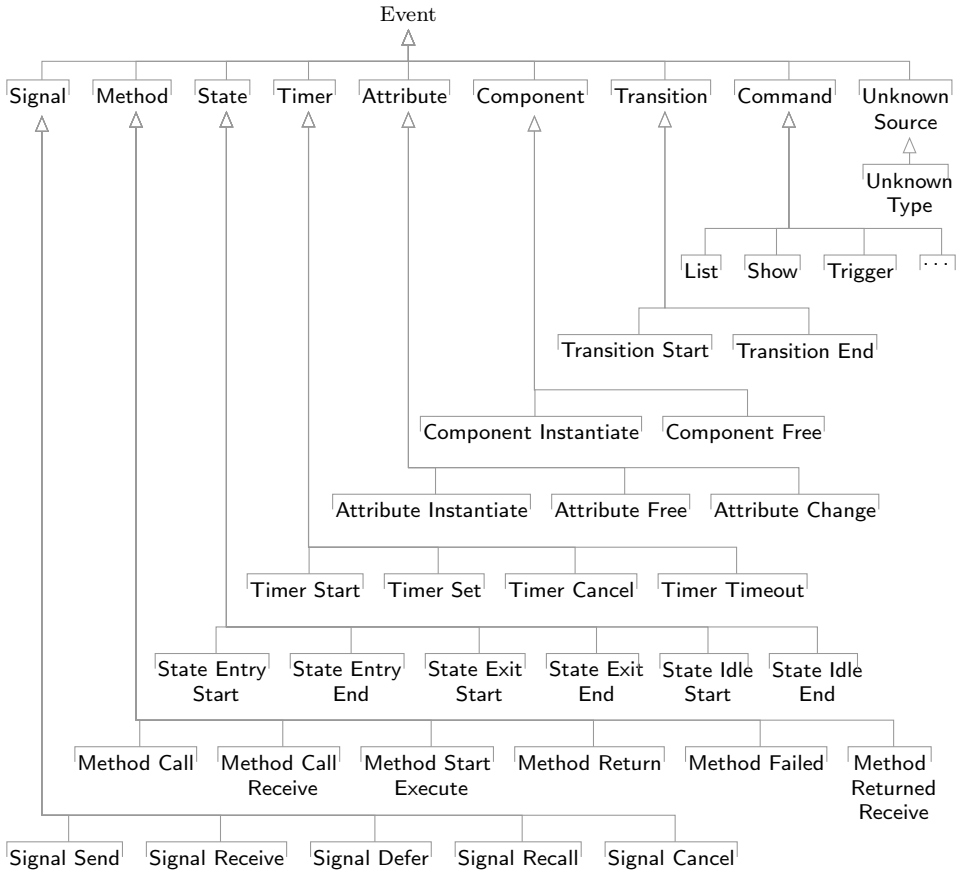


Fig. 4 Taxonomy of events (adapted from [16]). The first level shows event sources. Subsequent levels show event types.

the event types. Two specific event sources (*Unknown Source* and *Command*) can be used to extend our frameworks to support specific activities. A discussion about how they have been used in the context of model-based debugging [3] is given in Section 7. Finally, we have defined three command event types that allow external monitors to functionally steer the execution. This is detailed in Section 3.4.

In comparison with our previous taxonomy [13], the taxonomy presented in this paper is more in line with the one defined in [16] so that the scope is increased to any component- and state-based programming languages, including not only UML-RT, but also any other UML-based modelling languages, Moore and Mealy state machines, and so on. The *capsule events* from the taxonomy we proposed in [13] have been replaced with the more generic *component events*[1]. In addition, missing events from the taxonomy of [16], like the *timer cancel* event and *attribute events* have been added.

[1] The taxonomy of [16] call them *object events*.

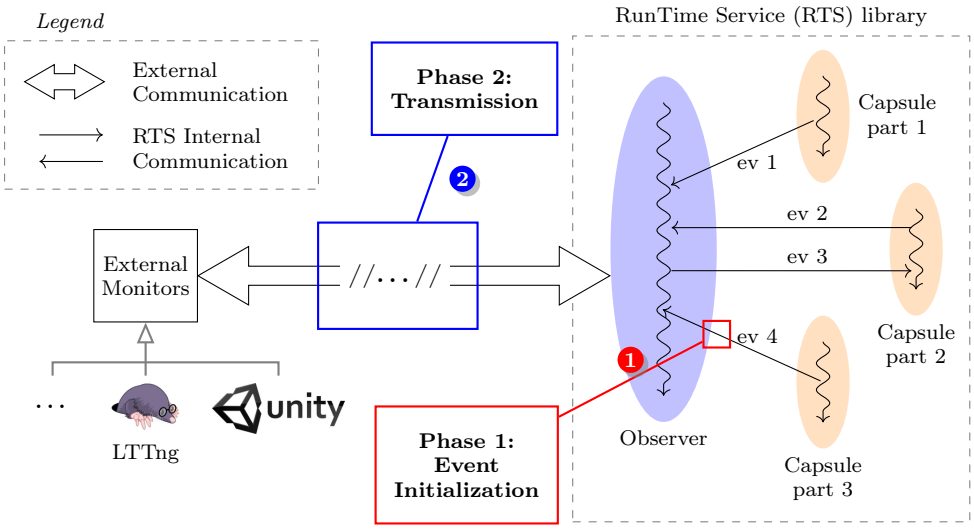


Fig. 5 Interactive run-time monitoring architecture.

3.1 Architecture Overview

To support a high degree of dynamism and address the requirements with an architecture that is user-friendly and meshes well with model-driven development, we use an internal capsule, called *Observer*, to connect the system with external monitors. The resulting indirection prevents the system from having to be altered when new monitoring capsules are added. For all intents and purposes, the observer execution flow keeps track of the different monitoring tools, their preferred way of communication, as well as output format. The architecture supports multiple monitors and does not require a regeneration, recompilation, or restart of the system.

Fig. 5 shows an overview of the architecture. On the right side, the observer execution flow is clearly isolated from the system execution flow. Internal mechanisms (e.g., threads, shared memory) ensure a fast communication with low overhead and the schedulability of the queued events. Consequently, even if the interaction between the Observer and the external capsule introduces some latency (caused by e.g., the choice of a remote communication protocol), this latency does not necessarily affect the system execution.

Once the Observer receives an event from a capsule part, it forwards it to a *consumer* through an external communication. Different communication protocols are supported by our implementation (cf. Section 3.3), in order to support internal as well as external third party monitoring tools. An external monitoring tool acts as the consumer. For example, LTTng, or Unity can consume events coming from the Observer.

The use of the architecture involves two phases. During the first phase (1), when an event occurs in the system, an Event object is initialized. This object contains relevant information such as a timestamp or currently CPU load or memory usage. This event is then forwarded to the Observer. During the second phase (2), the event object is serialized by the Observer capsule and transmitted to a consumer which previously

registered itself to the Observer[2]. The consumer can then deserialize and gather the relevant information embedded in the event.

For the sake of unification, the Observer is modelled as a capsule in UML-RT. Consequently, we assume that its implementation is generated like any other capsule of the model using existing code generators. Therefore, the internal communication between *capsule parts*, i.e., instances of specific capsules, and the observer is ensured by the Run-Time Service (RTS) library provided by the language which guarantees the delivery of events produced by the different capsules and preserves their ordering when consumed by the Observer. In addition, the infrastructure supports dynamicity, as it clearly distinguishes events coming from different capsule parts. Therefore, if a capsule is instantiated several times at run-time, then each instance can be uniquely identified.

Finally, the infrastructure illustrated in Fig. 5 supports bi-directional communication protocols between the system and external monitors. Bi-directional communication allows the external tool not only to observe the system execution, but also interact with it and steer it. Details of the implementation are given in Section 4.

3.2 Observer Capsule

The Observer is modelled explicitly as a UML-RT capsule and therefore has its behaviour defined using a state machine. Fig. 6 describes its state machine. It consists of one single state and four transitions. During the initial transition (transition configure), the communication mode is set. Currently, three modes are supported: *Shared memory*, *TCP*, and *Message Queuing Telemetry Transport (MQTT)*. TCP and MQTT can be used to connect external monitors to the Observer. They support remote connections, which is particularly useful for distributed or embedded systems where the monitoring tools cannot always be colocated with the system. However, TCP communication is rather slow and therefore not suitable to adequately monitor real-time constraints. To cover that, the Observer also supports the *shared memory* protocol, which is faster and reduces the memory and CPU overload. However, this mode does not support remote connections.

The three modes support bi-directional communications, therefore an external component can not only observe the execution of the system for debugging purposes or communication, but it can also interact with it (cf. Section 3.4).

[2] The registration phase is not illustrated in Fig. 5.

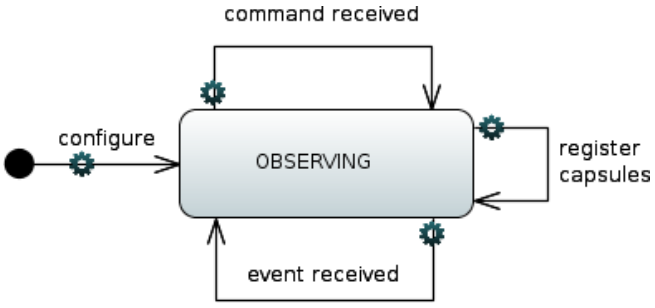


Fig. 6 Observer behaviour. It consists of one state and four transitions.

In the OBSERVING state, the Observer has three tasks: keeping track of newly created capsule parts (register capsules transition), collecting new events generated by the system and forwarding them to external monitors (event received transition), and listening to commands coming from external monitors to steer the system execution (command received transition).

3.3 Observing the System Execution

On the model side, modelling the Observer as a UML-RT capsule confers several benefits. As explained earlier, the Observer capsule can be automatically bound to any capsule to monitor through its SPP port, given that the capsule to bind to owns the corresponding SAP port.

Mathematically speaking, the *areDynamicallyBound* relation between the SPP port p_{obs} of the Observer capsule c_{obs} and a SAP port p of the capsule c to monitor can be formalized as follows:

$$\begin{aligned} &(p,\, p_{obs}) \in areDynamicallyBound \text{ if and only if:} \\ &p.kind = SAP \wedge p_{obs}.kind = SPP \wedge p.name = p_{obs}.name \end{aligned} \tag{1}$$

That is, the two ports p and p_{obs} are dynamically bound if and only if p is a SAP port, p_{obs} is a SPP port, and both p and p_{obs} share the same name. Using Eq. 1 we now formalize observability:

$$\begin{aligned} &p \in isObservableSapPort \text{ if and only if:} \\ &\exists p_{obs} \in c_{obs} \,|\, (p, p_{obs}) \in areDynamicallyBound \end{aligned} \tag{2}$$

That is, a port p is 'observable', i.e., it is the counterpart of the SPP port of the Observer capsule c_{obs}, if and only if there exists a port p_{obs} owned by the Observer capsule such that both ports p and p_{obs} are dynamically bound.

Finally, with Eqs. 1 and 2, we can formalize the *isMonitoredBy* relation between a capsule c and the Observer capsule c_{obs} as follows:

$$\begin{aligned} (c,\, c_{obs}) \in isMonitoredBy \text{ if and only if:} \\ \exists p \in c.ports \,| \\ p \in isObservableSapPort \end{aligned} \tag{3}$$

That is, the capsule c is monitored by the Observer capsule c_{obs} if and only if it owns a port p that is observable, i.e., dynamically bound to the SPP port of the Observer capsule.

Fig. 7 illustrates how the Observer capsule is instantiated and how it monitors specific capsule instances. The figure contains three capsule instances c1:C1, c2:C2, and c_{obs}:Observer. The Observer capsule c_{obs} owns the conjugated SPP port called p_{obs}, to which external capsules can be automatically bound, if they own the corresponding SAP port. Thus, in this example, only instances of C1 are monitored by the Observer, instances of C2 are not. This small example shows how simple it is to monitor specific capsules and how small the modifications necessary on the model are. It is therefore

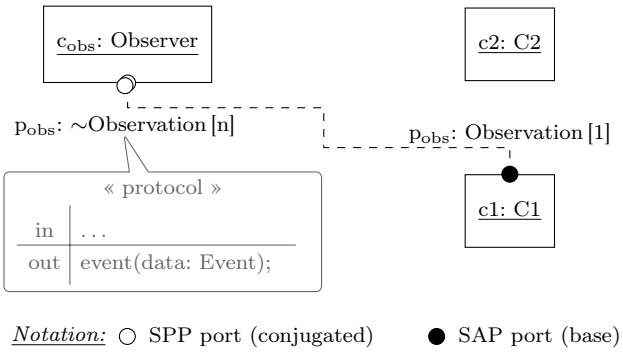


Fig. 7 Observer instantiation.

possible, given a specific use case, to monitor the whole system, or to select the relevant capsules to monitor while ignoring the others. This may reduce the amount of trace events produced by the system considerably.

Through its SPP port, the Observer capsule can receive the event message defined within the Observation protocol. It owns a data parameter which is initialized in the sending capsule and serialized by the Observer. In addition, incoming messages can be defined in the Observation protocol. This mechanism is used for steering the system execution. More details are given in the following section.

To facilitate the initialization of events by a capsule part, we implemented an Event utility class, illustrated in Fig. 8. It contains several attributes such as an ID to uniquely identify an event, the source of the event, the capsule instance which generated the event, and additional monitoring information, e.g., the timestamp from when the event was initialized, or the CPU tick.

This class can be instantiated to create specific events. It supports the creation of objects for every event that can occur in a UML-RT model. Each event is identified by an *event type* and an *event source*. Both values have to be consistent. For instance, the TIMER_RESET event type is only meaningful for the TIMER event source.

The Event utility class provides a certain number of constructors to initialize an event object, and to deserialize event objects that were previously serialized. In addition, it also defines a function to serialize events. Serialization and deserialization can be customized so a specific monitoring tool may require the Observer to serialize the object with the proper information it needs. This allows for, e.g., reducing the size of traces and the memory load by only embedding the required information in the traces.

3.4 Steering the System Execution

The architecture can be adapted to allow an external tool to influence and steer the execution of the system. This requires a small modification in the way the architecture is used (cf. Fig. 5). More precisely, the two phases (1, 2) are inverted. First, the external tool initializes an event object, which is then serialized and transmitted to the Observer capsule. The Observer then parses it and passes it on using a routing table of all capsule instances (registered during the registration phase), so messages can be sent

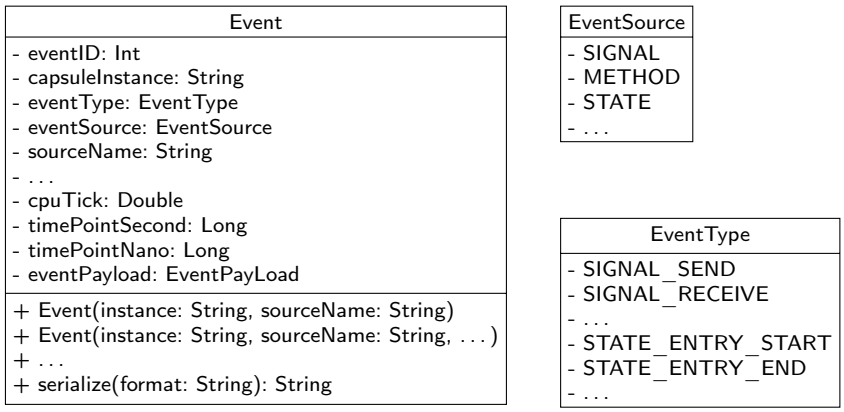


Fig. 8 Event initialization.

to specific instances. These messages may trigger specific transitions in the receiving capsule instance, hence affecting the behaviour of the system.

Steering the system execution is also supported by the relation that binds a capsule to the observer through the SPP/SAP ports. Concretely, for the Observer capsule to trigger a specific transition in a capsule state machine, the transition must contain a trigger element (see Fig. 2) properly configured to be activated when a message coming from the Observer capsule is received on the SAP port p_obs. That message must be defined as part of the Observation protocol list of incoming messages[3].

Mathematically speaking, the *isTriggerableBy* relation between the capsule c, a transition t of the capsule state machine, and the Observer capsule c_{obs} can be formalized as follows:

$$\begin{aligned} (t,\, c,\, c_{obs}) \in \mathit{isTriggerableBy} \text{ if and only if:} \\ (c,\, c_{obs}) \in \mathit{isMonitoredBy} \wedge \exists \mathit{trig}_{obs} \in t.\mathit{triggers} \mid \\ \exists p \in \mathit{trig}_{obs}.\mathit{ports},\, m \in \mathit{trig}_{obs}.\mathit{messages},\, \mid \\ m \in \mathit{Observation.in} \wedge\ p \in \mathit{isObservableSapPort} \end{aligned} \tag{4}$$

Let t be the transition to be steered. t is triggerable by the observer if and only if (i) it is owned by the state machine of a capsule c monitored by the Observer capsule c_{obs}, and (ii) it contains a trigger $trig_{obs}$ defined as the tuple (p, m) where p is an observable SAP port (i.e., dynamically bound to the SPP port of the Observer capsule c_{obs}) and m is a message part of the incoming list of messages of the Observation protocol. In the following, we distinguish two cases where steering can be useful and we discuss the impact on the architecture.

3.4.1 Functional Steering

Functional steering is a specific form of the above formalization. It is limited to steering only the already available functional parts specified in the model. It can be used for

[3] Since the Observer's SPP port is conjugated, the incoming message becomes an outgoing message for the Observer capsule.

testing unexpected behaviour. For example, a desirable scenario consists in manually triggering specific transitions of the system to visualize the impact on the system behaviour and possibly detect flaws in the design of the system.

It also has benefits in unit testing. The Observer capsule can help *mock* the behaviour of other capsules that may have not been modelled yet. To enable this scenario, the capsules to mock do not have to be defined, only the list of messages (as part of the definition of a UML-RT protocol) they are supposed to send and receive in order to interact with other capsules of the system. An example of the use of this approach for steering a Rover is given in Section 6.2.

The *isFunctionallyTriggerableBy* relation between a capsule c, a transition t of the state machine of that capsule, and the Observer capsule c_{obs} can be formalized as follows:

$$\begin{aligned}(t,\, c,\, c_{obs}) \in \mathit{isFunctionallyTriggerableBy} \text{ if and only if:}\\ (t,\, c,\, c_{obs}) \in \mathit{isTriggerableBy} \wedge\\ \exists \mathit{trig}, \mathit{trig}_{obs} \in t.\mathit{triggers} \mid\\ \mathit{trig} \neq \mathit{trig}_{obs} \;\wedge\; \mathit{sameSigs}(\mathit{trig}.\mathit{messages}, \mathit{trig}_{obs}.\mathit{messages})\end{aligned} \tag{5}$$

That is, t is functionally triggerable by the Observer if and only if t is triggerable by the Observer and that trigger $\mathit{trigger}_{obs}$ has the same signature as the message that *trig* consists of (i.e., the message name, as well as the number, names, types, and directions of the parameters are the same).

We note that functional triggerability is a special case of triggerability, i.e., we have:

$$\mathit{isFunctionallyTriggerableBy} \subset \mathit{isTriggerableBy} \tag{6}$$

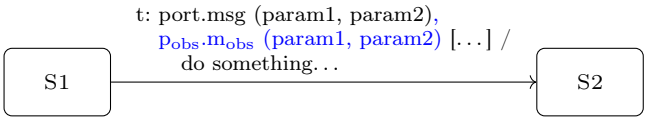


Fig. 9 Duplicating triggers.

Fig. 9 shows the modification to the state machine so the Observer capsule can trigger the transition t between two states S1 and S2. Let us assume that t is normally triggered when receiving the message *msg* which has two parameters $\mathit{param}1$ and $\mathit{param}2$. A second trigger is defined and added to the list of triggers of t. That second trigger defines the message m_{obs} which comes from the p_{obs} port and has the same signature as *msg*. Consequently, the reception of either the *msg* or the m_{obs} message triggers the transition. Of course, defining the duplicate message m_{obs} requires the Observer capsule to be able to send that particular message. Thus, m_{obs} has to be defined as an incoming message in the Observation protocol.

Note that UML constraints are not part of the trigger signature; thus, any constraint associated with the transition in the form of a guard has to evaluate to 'true' for the transition to be taken, no matter if the transition is triggered by the Observer or not.

3.4.2 Non-Functional Steering

Seamlessly adding capabilities to the model through model instrumentation has proved to be useful not only for observing, but also for debugging. We have tried a similar approach in the context of live model-based debugging in [3]. M2M transformation rules [23] have been applied to translate a user-defined model into an instrumented model including debugging capabilities. The approach has been designed to be fully platform independent, without having to rely on any program debugger (e.g., the GNU Project Debugger (GDB)) so as to prevent developers from having to maintain mappings between the model environment and multiple code-level debugger implementations. A Command Line Interface (CLI) has been designed to debug the model and an extension of the Eclipse Graphical User Interface (GUI) debugger has been implemented to support debugging in a graphical way [4]. This application used the extension of the event taxonomy (cf. Fig. 4) in order to add specific commands for debugging. Additional events include setting and removing breakpoints, stepping through the execution, changing values of capsule attributes, etc. A detailed presentation of our implementation is given in [4].

Supporting such scenario through the Observer library can be done thanks to the bi-directional protocols the Observer relies on for monitoring and steering capsules. It requires two additional changes to the architecture. First, the taxonomy needs to be extended to support the events and commands required for the specific interaction between the Observer, the external monitoring tools, and the capsules of the system. Second, the default M2M transformation rules specified for the Observer to monitor and to steer the execution need to be able to deal with the new events and commands on both sides of the Observer.

3.5 Using the Architecture

Tab. 1 summarizes the way the observed capsule, the observer, and the event initialization are used to achieve monitoring or steering. For observation, a SAP port needs to be added to each capsule to monitor. When steering the system execution, each transition that should be triggerable by the observer should have a duplicated trigger used by the Observer capsule. This requires changes in both the capsule state machine and in the observer protocol. Finally, the way of using the Event utility class is also adapted when steering the execution. In that context, an event object no longer describes a real event that occurred in the system execution, but instead a trigger created by an external monitoring tool to alter the system execution. The only difference is in the use of the *capsuleInstance* which no longer identifies the capsule instance from which the event originates, but instead, the recipient for which the event is prepared.

4 Implementation

The instrumentation of the models has been fully automated through M2M transformation rules. This allows for instrumenting every state and transition of the model automatically. For a finer-grained control of the instrumentation process, we have im-

Table 1 How to use the architecture for monitoring and steering.

	Monitoring	Steering
Observed Capsule	Owns the SAP port.	Triggers of externally triggerable transition duplicated.
Observer	Nothing to do.	Messages sent to capsule added to observation protocol.
Event Initialization	*capsuleInstance* is the capsule part producing the event.	*capsuleInstance* is the capsule part to which event is addressed.

plemented a small user interface extending the open source Eclipse Papyrus for Real-Time (Papyrus-RT) modelling environment[4].

This section first describes the implementation of the M2M transformation rules, then it describes our implementation within Papyrus-RT.

4.1 Model-to-Model Transformation

Algorithm 1 Top-level transformation rule for instrumenting a UML-RT model.

```
1: inputs
2:    M,                                  → the user-defined UML-RT model
3:    unobserved_capsules                 → a set of non-observable capsules
4: output
5:    M'                                  → the instrumented model
6: procedure InstrumentModel
7:    M' ← M                              → deep copy
8:    loadObserver()                      → load the Observer package
9:    instrumentObserver()                → instrument the Observer capsule
10:   M'.top.parts.push(createObserverPart()) → create and add an Observer capsule part
11:   for all capsule ∈ M'.capsules do
12:      if capsule ∈ unobserved_capsules then
13:         continue
14:      else
15:         capsule.ports.push(createSapPort()) → instantiate a SAP port
16:         refineBehaviour(capsule)      → add events to the capsule
17:      end if
18:   end for
19: end procedure
```

The M2M transformation rules have been implemented using the Epsilon Object Language (EOL) from the Epsilon toolset. Algorithm 1 gives an overview in pseudo-code of the transformation rules we have implemented. It takes as input a user-defined UML-RT model M and an optional set of unobserved capsules, and returns an instrumented model M'. Instrumentation is done on a copy of the input model M (line 7) so that the instrumentation is invisible for the user and that the input model is not altered by the transformation process. Additional configuration variables are not shown

[4] Our implementation is open source and available at `https://github.com/nicolas-hili/Observer`

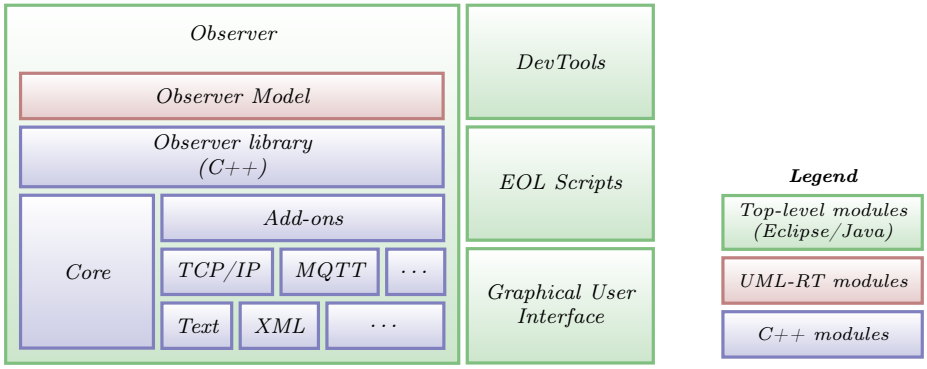


Fig. 10 Overview of the main modules composing our implementation.

in Algorithm 1. They include the selection of the serialization format and the communication method that set up the Observer capsule for a specific usage.

The first step of the algorithm is to load the Observer capsule, to create a capsule part, and to add it into the top capsule of the model M' (lines 8–10). During this step, the Observer capsule is itself instrumented so that: (i) it is correctly set up with the selected serialization format and communication method, and (ii) it can trigger any transition of the state machines of the observed capsules (functional steering).

The second step of the algorithm is to instrument every capsule that has not been tagged as *unobserved*. This is the core of the instrumentation process, so that events can be generated according to the taxonomy presented in Fig. 4. It involves: (i) adding a SAP port to each capsule (line 15) so that it is automatically bound to the SPP port of the Observer, and (ii) refining the behaviour of the capsule (line 16). During this last step, the state machine of the capsule is refined so that events are generated every time a state is entered, a transition is triggered, a signal is deferred, etc.

4.2 Papyrus-RT Implementation

We have implemented our framework within the open source Eclipse Papyrus for Real-Time (Papyrus-RT) environment. It has been developed from the ground up to be modular and easily extensible, whether one wants to use it for instrumenting UML-RT models with the default taxonomy of events, or to extend it to add activity-specific events into the taxonomy or even to adapt the Observer library to support other UML-based languages.

Fig. 10 shows an overview of the different modules that were developed. It contains four main modules: *Observer* contains the Observer model and code, *DevTools* contains development specific tools to facilitate the development of the Observer capsule, *EOL Scripts* containing the EOL transformation rules, and *Graphical User Interface* containing the Eclipse plug-ins used to extend the Papyrus-RT environment.

The Observer module is composed of two parts: (i) a UML-RT model that contains the definition of the Observer capsule, its state machine, and the Observation protocol; and (ii) an underlying C++ library the model relies on. C++ being the target language

into which code is generated from the model, this facilitates the link at compile-time between the C++ library and the code generated from the instrumented model.

The Observer has been designed to be as modular as possible. Models and code have been kept separate so that the underlying C++ library can be extended, tested, and even used in standalone mode for C++ code projects. Furthermore, *add-ons* (being output formats or communication protocols) have been separated from the *core* library so that a user can bundle the Observer with his/her add-ons of choice or create new ones.

Finally, we have also developed a GUI in order to facilitate the use of our architecture. It supports the instrumentation process and allows the user to configure the Observer and the add-ons (s)he selected. It also offers a graphical way to filter the capsules to observe[5].

4.3 General Applicability to Other Modelling Languages

The Observer architecture and its implementation have been designed to be more generally applicable to other modelling languages, not only to UML-RT. This includes other UML-based modelling languages that extend the more general StructuredClassifier concept from UML, such as SysML. SysML blocks share common features with UML-RT capsules. They both have attributes, ports, and parts. SysML parts are, like capsule parts, hierarchically defined, and parts can be connected through connectors.

Table 2 Mapping between UML-RT and UML terminologies. The left side shows the classifiers while the right side shows the associations.

UML-RT	UML
Capsule	Class[a]
Capsule part	Property
Port	Port[b]
Connector	Connector[c]
Protocol	Interface

Classifiers

UML-RT	UML
Capsule::parts	StructuredClassifier::/part[d]
Capsule::ports	EncapsulatedClassifier::/ownedPort[e]
Port::type	Port::/required & Port::/provided[f]
ConnectorEnd::partWithPort[g]	
ConnectorEnd::Role[h]	

Associations

[a] A UML class extends EncapsulatedClassifier. Therefore, it can own ports and parts.
[b] SAP/SPP port kinds in UML-RT do not exist in UML.
[c] Connectors have two connector ends in both languages. This was not shown in Fig. 1 for readability purpose. See [36, p. 225] for more detail.
[d] Property derived from StructuredClassifier::ownedAttribute.
[e] Property derived from StructuredClassifier::ownedAttribute.
[f] UML-RT protocols provide a higher-level construct than the low-level UML concepts of provided / required interfaces.
[g] Connector::partWithPort designates parts in both languages.
[h] Connector::Role designates a port in both languages.

Tab. 2 illustrates the mapping between some UML-RT concepts shown in Fig. 1 and the corresponding UML concepts. From this mapping we can see how we can adapt the different equations in Section 3 to make them adaptable to other modelling languages

[5] A detailed documentation of our implementation is available at: `https://github.com/nicolas-hili/Observer`

complying with UML structured classifiers. For example, Eq. 1 should be reformulated, in a UML-RT term-free way, as follows:

$$(p,\, p_{obs}, con) \in areStaticallyBound \text{ if and only if:} \\ \exists (end, end_{obs}) \in con.end \mid \\ end.role = p \wedge end_{obs}.role = p_{obs} \tag{7}$$

That is, given two ports p and p_{obs} and a connector *con*, the connector *con* statically binds the two ports if and only if each connector end has each of the two ports as role. Note that since the notion of SAP and SPP are specific to UML-RT, ports are statically bound by a connector that appears explicitly in Eq. 7.

We attempted to provide a taxonomy of events (see Section 3) free of UML-RT terminology so that it can be reused without any change for other modelling languages. At the behavioural level, the semantics of steering has a specific meaning depending on the chosen formalism. In our work, we focused on state-based formalisms in which steering influences the behaviour of the system by artificially injecting messages that cause specific transitions to be triggered. That semantics may differ for other behavioural notations. For example, in activity diagrams, behaviour is caused by the arrival of 'tokens' representing either control flow (e.g., the previous action has finished) or data flow (i.e., some required data has arrived). So, the Observer would have to produce these tokens and put them in the input pins of the activity or action to be executed. This example illustrates how we could adapt the Observer capsule for other notations by replacing the key terminology and concepts.

Our implementation makes the assumption that C++ code is ultimately generated from the model so that the underlying C++ library the Observer relies on can be interfaced with the generated code. Modelling languages involving other target languages (e.g., C, Java) require porting the C++ library to these languages.

Table 3 Non-exhaustive list of other discrete system-based modelling languages and tools to which the Observer library could be adapted.

Language	Tool	Platform	Target language
Finite State Machines (FSMs)	Yakindu Statechart Tools[a]	Eclipse	Java, C, C++, ...
xtUML	Bridgepoint	Eclipse	Java, C, C++, ...
UML-RT	RSARTE	Eclipse/Proprietary	C++
UML/SysML	MagicDraw[b]	Proprietary	Java, C++, C#, ...
UML	IBM Rhapsody	Proprietary	Java, C, C++[c]

[a] Yakindu Statechart Tools Professional Edition is required to support deep C/C++ integration, required to integrate the Observer library.
[b] MagiDraw professional C++ edition is required to support C++ code generation.
[c] Generate only C++ code frame.

Finally, Tab. 3 lists some modelling tools and languages to which the Observer architecture could be adapted. Yakindu Statechart tools offers a dedicated environment for modelling Harel's statecharts and Finite State Machines (FSMs). It is based on the Eclipse platform. It does not offer a way to define FSMs within individual components like UML-RT does, but orthogonal regions could simulate multiple components

interacting with each other. xtUML Bridgepoint offers a modelling environment with a language similar to UML-RT. It includes five main concepts: component, interface, class, state, and action. It is also based on Eclipse. Rational Software Architect RealTime Edition (RSARTE) is an alternative commercial version to the open-source Papyrus-RT environment to model systems using the UML-RT language. Other UML and SysML commercial tools could be used, such as MagicDraw and IBM Rhapsody.

Most of the tools already support C++ code generation so that the Observer library could be interfaced with the code generated from the model. In the case of Yakindu, the support for custom C/C++ types (to interface with the Observer C++ library) is available only into the professional edition. Reusing the EOL transformation rules can be done on any Eclipse-based tool supporting the installation of the Epsilon toolset.

5 Experiment

This section details the experiment we conducted in order to assess the applicability of our approach. The first part details the different case studies we set up. The second part presents the experimental protocol and hypotheses we formulated. The third part details results showing the benefits of our approach in terms of performance, scalability, and usability.

5.1 Case Studies

To illustrate the use of the architecture, we set up a set of case studies involving different kinds of monitoring tools. The models have different complexities that range from a few model elements to hundreds. The left side of Tab. 4 details the complexity of the models. Simple academic models include a *Counter* system that counts the elapsed seconds and a *Car Door Central Lock* system that simulates the opening and closing of car doors in a centralized way. More complex models are detailed below.

Rover System: We built a prototype of a Rover system, based on a Raspberry PI 3 board [39, 2]. The Rover features three wheels controlled by two servomotors and several sensors (such as a detection sensor which calculates the distance to any potential obstacle). Sensors and servomotors are connected to a Raspberry PI 3 through its General-Purpose Input Output Pins (GPIOs).

The Rover model has been developed in UML-RT with Papyrus-RT (see [2] for more details). The model is executable, and code can be generated from it to be further deployed on the Raspberry PI 3. It consists of three layers. The bottom layer is composed of UML artifacts that define the services for accessing the GPIOs. Read/write access mode can be set individually for each pin. The intermediate model layer consists of capsules and protocols to define how the application layer may access the different sensors and actuators. For example, we modelled an EngineController capsule to allow the application to cause the rover to move forwards and backwards and turn left and right. Finally, the upper layer is the application model which defines what the rover is to do (e.g., drive to a certain location or collect data while avoiding obstacles).

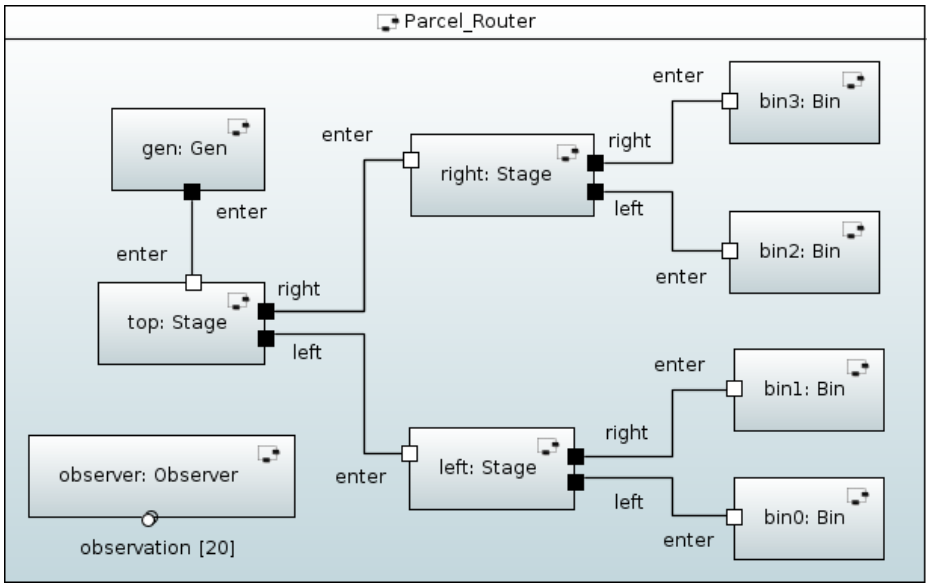


Fig. 11 Capsule diagram of parcel router model with observer.

Parcel Router: The *Parcel Router* [43, 28] is an automatic system where tagged parcels are routed through successive chutes and switchers to a corresponding bin. The system is time-sensitive and parcels can jam due to the variation of time spent by a parcel to transit through the different chutes.

As the Rover model, the Parcel Router model was created using UML-RT with Papyrus-RT and code can be generated and executed in order to test the system. The capsule structure of the Parcel Router is shown in Fig. 11. It consist of a Gen capsule that generates parcels and three stages responsible for conveying parcels to one of four destination bins. Each stage is further decomposed into chutes, switchers, and sensors (not shown in Fig. 11).

FailOver: The *FailOver* system [5, 22] is an implementation of the fail-over mechanism. It involves a set of servers processing client requests. To meet high availability, the system supports two replication modes, *passive* and *active* [17]. In passive replication, one server component works as the master, handling all the client requests while backup servers are largely idle, except for handshake operations. Whenever a malfunction occurs, resulting in a failure of the master server, a backup server is ranked up as the new master. In active replication, client requests are load-balanced between several servers. In addition to processing client requests, each server has to update its status to inform other servers of its availability. Therefore, each server can be notified whenever a malfunction causes the failure of one of its peers.

5.2 Experiment Protocol

In order to assess the applicability of our approach, we formulated two hypotheses w.r.t. the expected gains compared to the cost of the instrumentation. The two hypotheses are formulated below:

Hypothesis 1 (Code Size) *While our approach, through the instrumentation process, increases both the complexity of the models and the generated code size, we hypothesize that the increase is reasonable w.r.t services the Observer provides.*

Hypothesis 2 (Performance) *We hypothesize that our approach relying on model instrumentation causes reasonable performance overhead, negligible enough so it can be applicable to RTE systems.*

To validate the second hypothesis, we observe that the instrumentation can be strictly bounded to *transition chains* as computation only occurs in three places: *transition effects*, *state entry actions*, and *state exit actions*. Indeed, the semantics of UML-RT state machines excludes any *do* activities. Therefore, we define a *transition chain* of a transition t_1 between two states s_1 and s_2 as the sequence of actions that is executed when the transition is fired. Consequently, we argue that the system-wide impact on performance is a function of and can be reduced to the impact on individual transition chains. More precisely, the goal of the experiment is to determine to what extent the execution of individual state machines *transition chains* is slowed down by the instrumented code causing the sending of the observation events to the observer.

5.3 Experiment Results

Beside assessing the coverage of our implementation, we evaluated our approach based on different quality metrics, such as the size overhead, instrumentation time, and performance overhead. To perform the experiment, we used a computer equipped with a 2.7 GHz CPU and 8GB of memory. All hardware and software configurations and workload of the system during the entire experiment were identical. For the experiments, we focused on events related to the capsule's state machines. This represents six types of events of the taxonomy: four types related to the execution of states (*state entry start*, *state entry end*, *state exit start*, and *state exit end*) and two types related to the execution of transitions (*transition start* and *transition end*).

Table 4 Model complexity.

Model	**Model Complexity**									
	Original					Instrumented				
	C	S	T	Tr	LoCs	C	S	T	Tr	LOCs
Counter	1	2	2	1	361	3	4	7	2	2768
Car Door Central Lock	4	8	18	9	2118	6	10	23	18	5153
Parcel Router	8	14	25	16	6237	10	16	30	32	9940
Rover	6	16	21	18	3830	8	18	26	36	7442
FailOver System	9	28	43	25	6202	11	30	48	50	10365

C: Capsules and Classes, *S*: States, *T*: Transitions, *Tr*: Triggers, *LoCs*: Lines Of Code

5.3.1 Size Overhead

This metric shows the impact of the model instrumentation process on the model complexity and size of the generated code. As part of the first hypothesis, the size of the generated code is directly proportional to the increased complexity of the instrumented model. More specifically, the increased complexity is due to the generation of events (and their transmission to the Observer capsule) related to the six types of events we are monitoring, i.e., for each transition and state of the system.

To evaluate the size overhead metric, we generated for each use case illustrated in Tab. 4 the code of both the original and the instrumented models. Then, we compared the complexity of both models and the generated code from them.

Tab. 4 shows that the increase in terms of model elements is constant and predictable. The number of capsules, states, and transitions are only impacted by the addition of the Observer to the model. The number of triggers in the instrumented model is twice higher than in the original model, since all triggers are duplicated to allow the Observer to steer the execution. This increase could be ignored if only observation capabilities have to be supported.

The increases of the numbers of both model elements and Lines of Code (LOCs) in the models' action code impacts the generated code size which grows by 1.5 to 7.5 times depending on the size of the case study. As shown in Fig. 12, as the models increase in terms of complexity, the code size ratio between the original models and the instrumented ones decreases. This is due to the fact that part of the increase is due to the addition of the Observer capsule and the related libraries which is constant for all models and proportionally smaller for large models than for smaller ones.

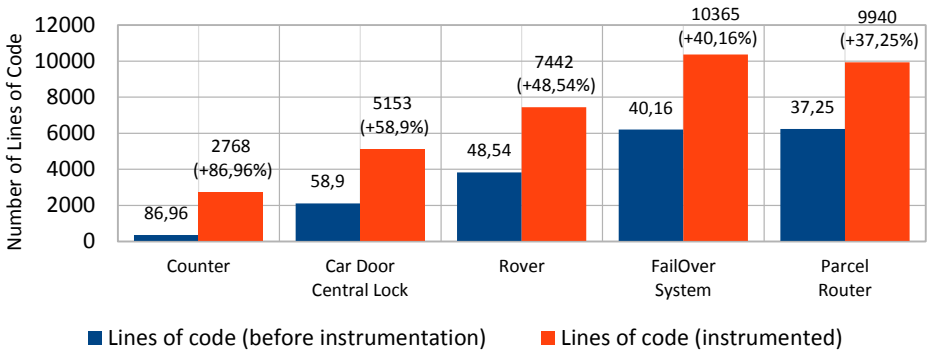


Fig. 12 Code size ratio.

5.3.2 Instrumentation Time

To verify that our approach is scalable, we also measured the time required by the tool to create the instrumented version of the model, from which code is generated. Tab. 5 enumerates the average time required by the instrumentation process to instrument the different models. For each case study, we performed 20 measures in order to reduce the margins of error. Tab. 5 shows that the time required by the instrumentation process

varies between 429 and 893 milliseconds which is in the order of magnitude of a second. It would have been interesting to compare the time required for instrumenting models in our approach with the time required for instrumenting code in existing model-based approaches. Unfortunately, no data is available for establishing such comparison.

In order to assess the scalability of our approach, we applied the model instrumentation process on a generated model with the complexity of 2363 states and 3725 transitions and 11 capsules. While there is no need to apply the instrumentation on a generated model, this experiment allowed us to check that the instrumentation time does not skyrocket when the model size grows exponentially. Note that, although intensive efforts are being made to make Papyrus-RT an industrial-grade Model-Driven Engineering (MDE) tool, it is still in incubation phase, hence no industrial model is available to assess the scalability of our approach.

The instrumentation time for the large model is 14 seconds on average which shows our approach is scalable enough for industrial models.

Table 5 Instrumentation time.

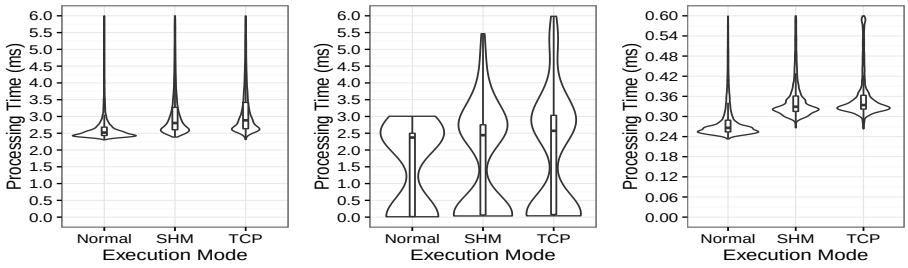
Model	Instrumentation time (in ms)
Counter	429
Car Door Central Lock	755
Parcel Router	788
Rover	581
FailOver System	893

5.3.3 Performance Overhead

Performance overhead is one of the main factors impacting the applicability of observability tools. This factor is even more important in RTE systems where time sensitiveness is the main concern during observation.

To evaluate this metric, we set up a benchmark for evaluating the performance over the FailOver system. To do so, we configured the system in the active replication mode with five clients and two servers, and we set up a simple scenario where each client checks the available servers, sends a message, and waits for a reply. Upon receiving the reply, the client processes the response and sends a new message after a specific amount of time elapses. The two servers process client requests and send replies. In addition, every five seconds, each of the two servers has to report its state, so both servers are notified whenever a failure affects their sibling.

We ran our system under normal mode (to show the real performance of the system) and observation mode using our approach, using both the TCP and the shared memory protocols (we chose not to include MQTT as it imposed a performance overhead that is closely similar to that of TCP). We focused on three specific transition chains: *RequestReply*, during which a client sends a message to both servers and waits for a reply; *ProcessResponse*, during which the client receives a response from one of the servers and processes it; *SendKeepAlive* during which a server reports its status to its peers. Finally, based on the system logs, we collected the computation time for replying to a server request (i.e., RequestReply transition), processing message response by the clients (i.e., ProcessingResponse transition), and reporting the availability of each server to its peers (i.e., SendKeepAlive transition). The three transitions chains we



(a) RequestReply (b) SendKeepAlive (c) ProcessResponse

Fig. 13 Processing time of transitions and overall processing time of 10,000 messages.

focus on are simple transition chains (they only involve two states and one transition). Therefore, the overhead to measure is due to the generation of six events in the following order: *state exit start* → *state exit end* → *transition start* → *transition end* → *state entry start* → *state entry start*.

Fig. 13 shows violin plots of the computation times measured for the three transition chains. The box plots within the violin plots show the median of data. Computation times are recorded until 10,000 messages are processed. The wideness bars show the density of computation time in the specific range. As shown by the figure, the system performance is slightly impeded by the use of the Observer capsule for both the shared memory and the TCP protocols. The majority of the *ProcessResponse* messages are processed within 0.3ms to 0.4ms, with a median time of 0.33ms when using shared memory, and a median time of 0.34ms when using TCP. The computation time in both modes is close to the processing time when the system is run under normal mode (i.e., without the observability layer) with a median time of 0.27ms. The overhead for the *ProcessResponse* transition is within the range of microseconds and therefore negligible. The overhead is similar for the *RequestReply* and *SendKeepAlive* messages. While the median time for *RequestReply* is 2.9ms and 2.9ms using shared memory and TCP respectively, the median under normal mode is 2.5ms. For the *SendKeepAlive* transitions, the median using shared memory and TCP are 2.4ms and 2.5ms respectively and is 2.3ms under normal mode.

In summary, we can observe that for each transition chain, the local performance overhead is always below 0.5ms for the transition chain to be completed (implying the generation and transmission of six events). Since the semantics of UML-RT is *run-to-completion* and that computation only occurs on transition chains, we can argue that the overhead of our approach is small and appears quite acceptable for many RTE systems.

5.4 Threats to Validity

The hypotheses we formulated have been validated on different models we created for the purpose of the experiment. The experiments showed that the instrumentation process increases the size of the generated code in a reasonable way and that most of the increase is due to the addition of the Observer capsule which is constant across the

model. Besides, our experiment showed that the instrumentation time is kept below one second for all models we tested and that the parallel execution of the Observer capsule does not lead to performance bottleneck. In the following, we detail three elements we identified that can threaten the validity of our experiment:
1) We evaluated or approach on relatively simple models we created. Therefore, we cannot assess the applicability of our approach to industrial-size systems. This is due to the fact that, even though there is evidence the applications of UML-RT in industry (specially in the telecommunications area), no industrial models are publicly available that we could reuse for our experiments.
2) The model used in the experiment is not industrial-size. However, we think it is sufficient to determine the impact of our approach on the time it takes to execute individual transitions. The experiment shows that this impact is a slowdown of less than half a millisecond on average. Note that this slowdown is not affected by the size of the model in terms of, e.g., number of capsules. In other words, the slowdown would not change significantly if a larger model (with, e.g., a larger number of clients or servers) was used.
3) We do not have a comparative baseline for our experiments, either from our previous prototype [13] or any other work in the literature. Finding such baseline in the literature to compare the performance overhead within the same experimental setup is very difficult. Our previous implementation supporting the early vision [13] was far from being mature and therefore, no code and performance analyses were performed.

6 Applications

In the vision paper [13], we suggested the connection of external components to the automatic system to support a wide range of activities with two goals in mind: improving the quality of the produced models and supporting communication. This section discusses some applications in which the Observer architecture has been used to support these two goals.

6.1 Timing Constraint Validation with LTTng

In our previous work [13], we proposed an approach for monitoring the execution of real-time systems modelled in UML-RT. This approach allowed us to annotate UML-RT behavioural models (i.e., state machines) of a system to automatically generate trace points information in the code when specific events occur (e.g., state change, transition triggered). By executing the code and running LTTng [14] – an open-source tracing framework that supports the monitoring of user-defined applications – on the physical platform, traces are produced and collected by Papyrus-RT. Traces are then analyzed with respect to specific timing constraints expressed in the model (e.g., *"the transition from state S1 to state S2 should be completed in less than 200ms"*) in order to detect violations.

This approach, however, had a significant limitation: In UML-RT, the context of a state machine is the capsule, so the generated traces belong to a capsule, instead of a capsule instance. Therefore, it was impossible to distinguish trace events produced by different capsule instances. The Observer architecture presented in this paper allows us

to overcome this issue because events contain information about which capsule instance they originated in.

Fig. 14 illustrates how we connected LTTng with the Observer capsule. Communication between the capsules and the Observer is ensured by the message passing implementation of the Papyrus-RT RTS library. To connect the Observer capsule with LTTng, we implemented a small wrapper. Communication between the wrapper and the Observer is ensured by shared memory, for improved performance.

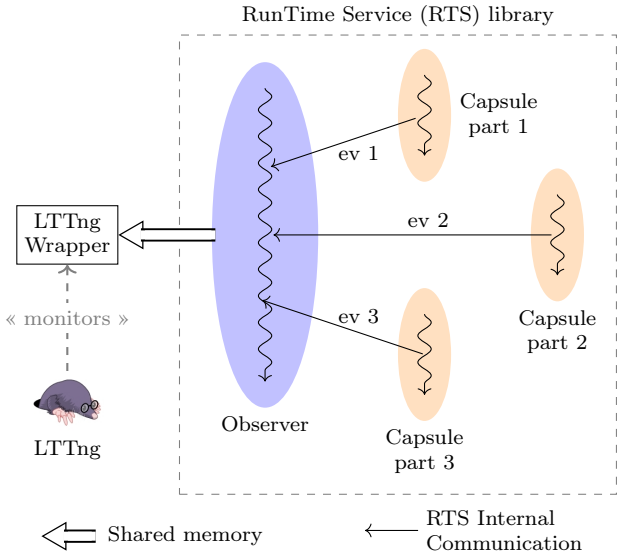


Fig. 14 Connecting LTTng with the observer capsule.

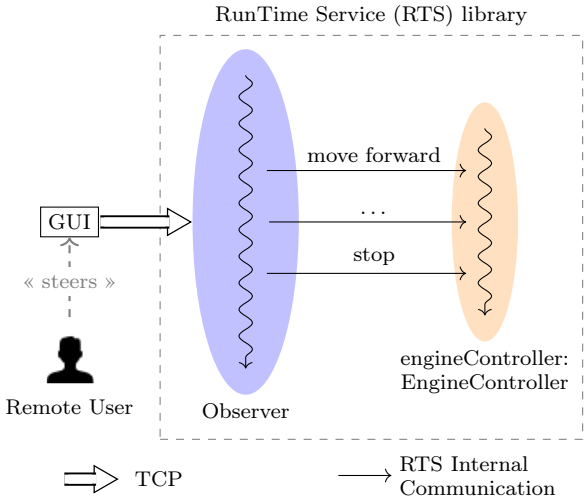


Fig. 15 Steering the execution with the observer capsule.

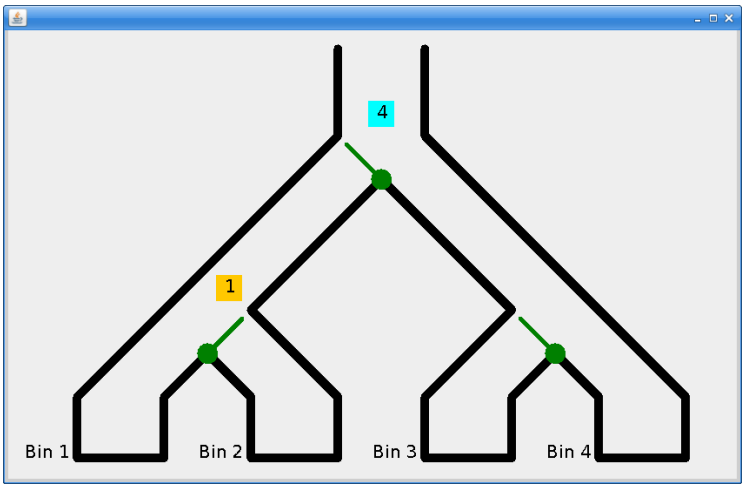


Fig. 16 Parcel router animation.

6.2 Functional Steering of a Rover System

We used the Observer architecture to control the Rover. In this case study, we removed the application layer containing the pre-built automatic scenario to drive the Rover. Instead, we injected custom events to alter the system execution. We focused on functional steering (see Section 3.4.1) of the EngineController, a capsule responsible for driving the Rover. TCP communication was used in that context and a small GUI was developed to steer the Rover execution.

As explained in Section 3, functional steering of the model execution requires some changes in the model of the system, which include duplicating the five different incoming messages of the EngineControler state machine (move forward, move backwards, turn left, turn right, and stop). Those changes are automatically performed by the M2M transformation rules. The code is then generated and deployed on the embedded platform.

6.3 Run-Time Animation

We applied the Observer capsule in order to monitor the execution of the Parcel Router model. The communication was bi-directional, which allowed us to observe the execution while steering it at the same time. To do so, we created a GUI in order to animate the execution of the system (cf. Fig. 16). This animation view was used in concert with a simple TCP client GUI we built in order to inject commands into the system and to collect raw traces[6]. Specific pieces of information were collected in order to update the animation and to interact with the GUI. This application shows how our infrastructure can support the combination of various monitors, and its ability to observe and steer the system execution at the same time.

We evaluated the benefits of such animations in the context of a graduate course on MDE. Students were given two partial models (a simplified version of the Parcel

[6] A screencast is available at `https://youtu.be/EbMIgEX9O58`

Router and a simple model of the Rock/Paper/Scissors game) written in UML-RT and were asked to complete them. Students could use the Observer and animation views such as illustrated in Fig. 16 to help them understand, debug, and complete the models. At the end of the assignments, students were asked to complete an anonymous questionnaire to help us evaluate the benefits of using the Observer architecture and the usefulness of the animation views. 62.5% of students found the animation views extremely helpful to understand the system (31.25% helpful, 6.25% neutral). 37.5% (resp. 37.5%) of students found the animation views extremely helpful (resp. helpful) to find bugs in the partial models (12.5% neutral and one student found it globally not helpful). 18.5% (resp. 31.25%) of students found the animation views extremely helpful (resp. helpful) for testing new features they have implemented (the rest ranged from neutral to not helpful). For this last result, it is notable to see that students tended to find the animation views more useful when the model was complex. Finally, 75% of the students found the Observer service easy to use (12.5% relatively easy, 12.5% neutral) and 37.5% found the service transparent to the user (37.5% relatively transparent, 12.5% neutral, and 12.5% relatively not transparent).

6.4 Selective Data Collection for IoT Applications

The Internet of Things (IoT) is arguably one of the biggest leaps in the evolution of the Internet since the introduction of the World Wide Web. The enormous amount of data generated by thousands of sensors embedded in our lives coupled with powerful data analytics and machine learning techniques promises a far-reaching impact that touches nearly every domain. The IoT is therefore very data-centric.

At the same time, Model-Driven Development (MDD) has not been a foreign concept for the development of IoT applications. In fact, the scale and complexity of IoT systems and the new challenges they set forth has motivated the development of a remarkable number of MDD tools for IoT in recent years [31, 37, 12, 10].

We believe that the Observer pattern coupled with the MQTT protocol can greatly facilitate the passive data collection in IoT applications. The MQTT protocol is one of the most popular application-layer protocols in the IoT domain and is supported by virtually every major IoT platform and framework. MQTT is a lightweight publish-subscribe protocol that relies on a central message broker to manage subscriptions and handle message delivery. An MQTT 'subscriber' client subscribes to specific topics of interest through the broker. Likewise, through the broker, an MQTT 'publisher' client can publish arbitrary payloads to a given topic. The MQTT broker makes sure that the payload is delivered to every subscriber to that topic.

To illustrate the benefits of the Observer for data collection, consider a weather station system that collects information in each zone from various sensors such as thermometers, barometers, and hygrometers. The sensor readings are passed to a controller that stores, analyzes and acts upon the information if necessary. The controller's actions could be, for instance, to issue emergency alerts and weather advisories.

The Observer capsule, coupled with the MQTT channel, easily extends the capabilities and usefulness of the system beyond what is explicitly implemented by the controller. As discussed earlier, the Observer is capable of automatically collecting run time traces including state transitions and message parameters. In the context of the weather station example, this means that the Observer can collect readings from all the different sensors and make those readings available to external systems with very

Table 6 Summary of the use of the Observer architectures for the different applications.

Applications	Communication protocol	Communication direction	Event sources	Event meta-data
Timing constraint validation	Shared memory	Uni-directional	State and transitions	Timestamp, CPU load
Functional steering	TCP	Bi-directional	States, transitions, and commands	Object parameters
Run-time animation	TCP	Uni-directional	States, transitions, and timers	Object parameters
Selective data collection	MQTT	Uni-directional	States and transitions	Object parameters

little overhead. An IoT-enabled external irrigation system, for example, can make use of the temperature and humidity readings to reduce water consumption.

One caveat, however, is the huge amount of traces that can be generated even from trivial applications. Given that IoT devices are often resource-constrained with very little processing power, handling a vast amount of data to nitpick certain events of interest can be proved to be cumbersome. Fortunately, we can leverage MQTT's subscription management scheme to solve this problem. Each event generated by the Observer is published to a specific topic that indicates the origins of said event. For example, the temperature readings of 'Zone1' are always published to the topic 'WeatherStation/Zone1/Thermometer/Read'. Note that 'Read' is the state in the state machine of the 'Thermometer' capsule where temperature events are generated. The topic hierarchy also indicates that the 'Thermometer' capsule is contained within capsule 'Zone1', which in turn is contained within the 'WeatherStation' capsule.

In addition, MQTT's support for single-level and multi-level wildcards ('+' and '#' respectively) allows a client to subscribe to a group of topics at once. For instance, a client may receive all temperature readings in all zones by subscribing to the topic 'WeatherStation/+/Thermometer/Read'. Similarly, a client can receive all events in 'Zone1' by subscribing to the topic 'WeatherStation/Zone1/#'. It is worthy to note that subscription management is handled by the MQTT broker and, therefore, no overhead is imposed on the observed system. As a matter of fact, there is always exactly one TCP connection between the observed system and the MQTT broker regardless of the number of clients subscribed to the system's events. In contract, the raw TCP channel requires a dedicated connection between the system and each of external client.

6.5 Summary and Experience Gained

Tab. 6 summarizes how we used the Observer architecture for the different applications. It lists the protocol and direction of the communication between the Observer and the external monitoring tools, the types of the observed events, and the information collected from the traces. Shared memory was used when possible, i.e., when the external monitoring tool can be embedded alongside the code execution. For the second application, functionally steering the system execution through the Observer capsule shows the benefits of using the Observer for mocking functionalities provided by capsules that have not been implemented yet. Leveraging the use of the Observer coupled

to animation tools appeared to have helped students find defects in the partial models and complete them. Finally, the last application raised the question of collecting large amounts of data. It shows the benefits of delegating the management of the publish-subscribe mechanism to the outside of the system when the number of consumers is large or when the consumers do not require the same level of information.

7 Discussion and Future Work

At the moment, the architecture supports configuration at design-time and at run-time. At design-time, a user can choose the different capsules to monitor, in order to filter the amount of information that can be collected by the Observer. He or she can also select specific messages triggerable by the Observer capsule. At run-time, it is possible to set up the communication mode (TCP or shared memory) and to dynamically connect or disconnect external monitors to the system execution. However, for the moment, the configuration is done globally for all monitoring tools, and cannot be specific for each one. To go one step further, local configuration for specific monitoring tools would make the monitoring more flexible and versatile. For example, one monitoring tool may choose to ignore all message events produced by a specific capsule part or by all instances of a capsule, while another tool could choose to listen to a specific capsule instance, while ignoring all others. Such configuration could be done at run-time, but requires further instrumentation of the Observer architecture. This instrumentation would provide fine-grained configuration per capsule, but would also increase the time required for processing of the information collected from the system, which may affect execution negatively. In other words, configurations at design-time and at run-time make different tradeoffs between high performance (design-time) and support for a wide range of fined-grained configurations (run-time).

One area where the proposed architecture appears to be useful is the development, monitoring, and adaptation of cyber-physical systems [7]. Our architecture can support various interaction scenarios in which, e.g., monitoring is used to detect unsatisfactory system behaviour and steering triggers an appropriate response in terms of, e.g., a system reconfiguration.

The proposed architecture also opens up some interesting applications in Run-Time Verification (RV). RV relies on monitoring relevant events in order to verify properties specified in Timed Linear Temporal Language (TLTL) or similar languages [26]. The proposed architecture efficiently supports the generation of configurable events which could contain the data required (e.g., attribute values, active states, timing information) to evaluate TLTL specifications. Additionally, it is possible to extend the current work to express TLTL properties at the model-level and to generate the TLTL query-based monitors for the verification of these properties automatically, as suggested, for instance, in [30, 40].

Finally, steering the system by external applications raises important concerns about security. An appropriate realization should reject connections from malicious applications, which requires the implementation of a proper security mechanism that only provides access to authorized applications.

8 Related Work

A significant body of related work can be found, spanning a wide range of research fields including simulation, testing, run-time monitoring, debugging, and animation of UML models in the context of RTE system development and analysis. The TOPCASED project was one of the first to offer support for the simulation of UML state machine [11]. More recently, different model execution environments appeared, such as Moka [15] and Moliz [29]. They propose a customizable model execution engine based on fUML [33], and support execution animation. Our work differs in that it focuses on monitoring the execution of the *code generated from the model*, rather than on executing the model via an execution engine. This makes our work more suitable for the analysis of properties that can only be observed when executing the developed system. Thus, our work addresses different concerns and is complementary.

"Monitoring Oriented Programming (MOP)" is a framework focusing on instrumenting a system directly at code-level [30, 40]. They allow for monitoring systems by injecting *monitors* directly inside the code of the system. Monitors are synthesized from specifications the code should satisfy. The framework is platform- and language-independent. Several implementations exist in specific languages, such as *Java-MOP* for Java [9]. Java-MOP uses AspectJ for injecting monitors within the code of the system. The approach does not require to re-compile the code to inject new monitors. However, restarting the system is mandatory, for AspectJ to be able to alter the code execution at binary-level.

All the related work we are aware of provides a way to observe the system execution, but no way to steer it. In [25], the authors extended the monitoring and checking (MaC) framework [24] in order to provide feedback to the monitored system when a requirement specification is violated. However, as the previous work, it relies on code injection to observe and steer the system execution with specific monitors. In that context, the notion of steering is quite far from ours. It defines a way to repair the system in case of requirement violation. In the context of our work, the notion of steering is more general, and defines a way for external monitoring tools to seize control of the execution. In [32], the authors presented an approach based on non-intrusive event observation in order to steer the execution of a system specified as the composition of object slices. Events are employed as triggers of some transitions of the system and their observation (using *probes*) may lead to a state change in the system. This opens up interesting perspectives in terms of optimizations (only the relevant subset of events can be generated). Again, in our case, the notion of steering is more general.

The work described in [27] addresses the issue of combining multiple monitors for run-time verification. Combining monitors is motivated by the need for increasing the accuracy of the information collected by the monitors. In our case, the combination of external monitors is motivated by the need to cover several applications, and is not limited to verifying real-time or embedded (e.g., memory usage) constraints.

All the previous work is code-based, and monitoring is done with respect to formal specifications. In contrast, our work is in the context of MDE and allows the combination of multiple monitoring tools that can be dynamically connected at run-time. Examples of monitoring tools include tools for functional correctness and real-time property verification, animation tools, 3D engine, etc. For this, we leverage the different features of UML-RT in order to build our model-based architecture. Our implementation is based on Papyrus-RT, but the underlying ideas are generic enough to be

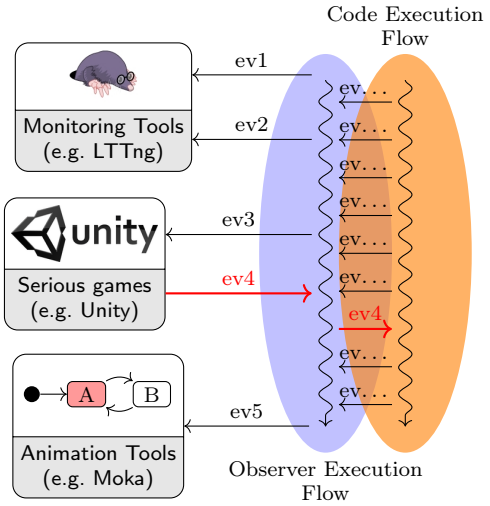


Fig. 17 Illustration of interaction between code, observer and external tools.

realized with other tools. For example, the RTS of RSA-RTE supports the concept of *external ports* [20] which could have been used to implement the Observer service.

The work described in [6] addresses the run-time monitoring and formal verification of UML models using observer automata. Observer automata are described as state machines synchronized with the UML model of the system to monitor. Monitoring is done with respect to safety properties. Each automaton describes one safety property to check. To the difference with our work, this work relies on UML interpreter. In comparison, our work relies on model instrumentation and code generation techniques to produce code that will be directly compiled and executed on the target platform. Our approach is more generic in the sense that the efficient, effective collection of a wide range of events for different purposes (e.g., not only run-time monitoring and verification, but also animation and steering) is the focus, rather than just the checking of safety properties.

Our architecture is highly configurable, both at design- and run-time. For observing the system execution, one can choose to observe the complete system, or just specific parts. For steering the model, some small changes have to be carried out on the model, in order to specify which transition can be triggered from the outside. Finally, the proposed architecture allows us to overcome specific challenges, such as the combination of multiple monitors.

While the present paper shares some of the goals of [13], it presents a different, much more mature and promising approach and implementation. More precisely, the work presented in this paper differs from that in [13] in the following ways:
1) The approach suggested in [13] advocates a *direct* connection of each external component with the system without any intermediary. However, this direct connection proved challenging to implement, especially in the context of dynamic connections which were difficult to support in a user-friendly, transparent way (e.g., without having to, e.g., regenerate code or restart the system after the addition of an external component).

Instead, this paper advocates a clear separation between the system and the code necessary for its observation (Fig. 17). This separation was formalized as an observer

Table 7 Comparison between the present and the past prototype.

	New prototype	**Past prototype [13]**
	Core	
Monitoring	Centralized into one single UML-RT capsule called *Observer*	Under the responsibility of each individual capsule of the system
Steering	Functional steering natively supported by the instrumentation process. Non-functional steering possible through minor change to the instrumentation process	None
Instrumentation process	Model-to-Model transformation	Code generator extension
Instrumentation genericity	Taxonomy-specific	Application-specific (depend on the code generator extension)
Filtering capabilities	Capsules and capsule parts	None
	Communication	
Protocol	TCP/IP (Server), Shared Memory, MQTT	TCP/IP (Slave)
Direction	Bi-directional	Uni-directional
Support for multiple connectors	Yes	No
	Output	
Output format	Text, XML, ...	Text
Trace format	Configurable (see Fig. 8 for the list of data embeddable in a trace)	Static (depend on the code generator extension)

service and implemented as an Observer component to make that service accessible by connecting a model component with the Observer at runtime. Our prototype implementation still uses UML-RT and the open source MDE tool Papyrus-RT [38], but we note that the architecture and its underlying ideas do not require these and could also be implemented in other MDE languages and tools that support dynamic connectors such as UML and IBM Rhapsody [21].

Based on the student experiment we performed, our infrastucture appears to be user-friendly, even for novice users. In a survey, over 87% of students in a graduate course on MDE found the Observer 'easy' or 'relatively easy' to use (see Section 6). Besides, we believe that our new implementation increases user-friendliness in comparison with the previous implementation: connections can be made at runtime and without the need for the manual inclusion of connectors; also, the observer service now becomes a simple extension of the RTS Library, the use of which users will already be familiar with.

The experiment described in Section 5 suggests that the communication overhead introduced by the observer can be kept relatively small and that the benefits of the tradeoff outweigh the performance costs for a significant class of applications. In the observer's default configuration, all external components receive all events generated by the model. To allow different external components to receive different events, we have

extended the observer to filter events based on 'subscription' specifications associated with an external component, giving the approach a certain similarity to the well-known 'publish/subscribe' architectural style [44]. Our implementation relies on the MQTT protocol, the open source Eclipse Mosquitto MQTT broker, and the open source Paho MQTT client. This has the benefit of deporting heavy computation due to the publish/subscribe mechanism outside of the model.
2) As a result of the changes sketched above, the prototype described in this paper is completely different from the one used in the vision paper (see Tab.7 for a non-exhaustive comparison list between the two prototypes). Moreover, it also is significantly more mature and offers more functionality, including support for remote and shared-memory communication, system steering via bi-directional communication links, and user-friendly use of the observer service through, e.g., an appropriate extension of the RTS library and the Papyrus-RT GUI, and the automatic creation of the connectors necessary for observation.
3) The preliminary Failover model in [13] is complemented by case studies involving a Parcel Router and a Rover to highlight the variety of external monitoring tools that can be connected to the running system and to showcase new applications, including run-time steering, run-time animation, and teaching.

9 Conclusion

In this paper, we have proposed a model-based architecture for supporting run-time monitoring activities in the context of real-time and embedded system development and analysis. The architecture is designed to be highly configurable. It supports the combination of various external monitors that can be dynamically connected to the monitored system. In addition to the dynamic configuration at run-time, the architecture also supports design-time configuration, in order to filter the amount of information produced by the system. Finally, we have presented how the architecture can be adapted, not only for observing the execution of a system, but also for steering it.

We have applied the architecture to several case studies including a Rover application and a Parcel Router system, and have illustrated its use for a wide range of applications such as real-time property verification and 3D real-time execution animation. We have described an experiment to measure the communication overhead of the architecture and the use of the architecture for teaching, both with encouraging results.

Acknowledgments

This work is supported by Ericsson Canada, EfficiOS, and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Agha G (1990) The Structure and Semantics of Actor Languages. In: Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, pp 1–59

2. Ahmadi R, Hili N, Jweda L, Das N, Ganesan S, Dingel J (2016) Run-time Monitoring of a Rover: MDE Research with Open Source Software and Low-cost Hardware. In: Workshop on Open Source for Model Driven Engineering (OSS4MDE'16)
3. Bagherzadeh M, Hili N, Dingel J (2017) Model-Level, Platform-Independent Debugging in the Context of the Model-Driven Development of Real-Time Systems. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 419–430
4. Bagherzadeh M, Hili N, Seekatz D, Dingel J (2018) MDebugger: a Model-Level Debugger for UML-RT. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ACM, pp 97–100
5. Balasubramanian J, Tambe S, Lu C, Gokhale A, Gill C, Schmidt DC (2009) Adaptive Failover for Real-Time Middleware with Passive Replication. In: 15th IEEE Symposium on Real-Time and Embedded Technology and Applications, IEEE, pp 118–127
6. Besnard V, Teodorov C, Jouault F, Brun M, Dhaussy P (2019) Verifying and Monitoring UML Modelswith Observer Automata. In: Model-Driven Engineering Languages and Systems (MODELS'19)
7. Bures T, Weyns D, Berger C, Biffl S, Daun M, Gabor T, Garlan D, Gerostathopoulos I, Julien C, Krikava F, Mordinyi R, Pronios N (2015) Software Engineering for Smart Cyber-Physical Systems – Towards a Research Agenda: Report on the First International Workshop on Software Engineering for Smart CPS. SIGSOFT Softw Eng Notes 40(6):28–32
8. Cassar I, Francalanza A, Aceto L, Ingólfsdóttir A (2017) A Survey of Runtime Monitoring Instrumentation Techniques. In: Francalanza A, Pace GJ (eds) Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, Torino, Italy, 19 September 2017, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol 254, pp 15–28, DOI 10.4204/EPTCS.254.2
9. Chen F, Roşu G (2005) Java-MOP: A Monitoring Oriented Programming Environment for Java, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 546–550. DOI 10.1007/978-3-540-31980-1_36
10. Ciccozzi F, Spalazzese R (2016) MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In: International Symposium on Intelligent and Distributed Computing, Springer, pp 67–76
11. Combemale B, Crégut X, Giacometti JP, Michel P, Pantel M (2008) Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In: 4th European Congress on Embedded Real Time Software (ERTS'08)
12. Costa B, Pires PF, Delicato FC, Li W, Zomaya AY (2016) Design and Analysis of IoT Applications: A Model-Driven Approach. In: Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C, IEEE, pp 392–399
13. Das N, Ganesan S, Jweda L, Bagherzadeh M, Hili N, Dingel J (2016) Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation. In: Model-Driven Engineering Languages and Systems (MODELS'16)
14. Desnoyers M, Dagenais M (2006) LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: OLS (Ottawa Linux Symposium), vol 2006,

pp 209–224
15. Eclipse Foundation (2016) Papyrus: Moka Overview, http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
16. Graf S, Ober I, Ober I (2006) A Real-Time Profile for UML. International Journal on Software Tools for Technology Transfer 8(2):113–127, DOI 10.1007/s10009-005-0213-x
17. Guerraoui R, Schiper A (1997) Software-Based Replication for Fault Tolerance. Computer 30(4):68–74
18. Hili N, Posse E, Dingel J (2018) Calur: an Action Language for UML-RT. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)
19. Hu B, Huang K, Chen G, Cheng L, Knoll A (2016) Evaluation and Improvements of Runtime Monitoring Methods for Real-Time Event Streams. ACM Transactions on Embedded Computing Systems (TECS) 15(3):56
20. IBM (2016) Rational Software Architect RealTime Edition (RSARTE): C++ RT Services Library, https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/b7da455c-5c51-4706-91c9-dcca9923c303/page/325220ca-b17d-4ea6-b382-4c704dbad0af/attachment/46c3438c-25d8-43ce-be22-730e46601365/media/RT%20Services%20Library.pdf
21. IBM (accessed July 19, 2016) IBM Rational Rhapsody. http://www-03.ibm.com/software/products/en/ratirhapfami
22. Kahani N, Hili N, Cordy JR, Dingel J (2017) Evaluation of UML-RT and Papyrus-RT for Modelling Self-Adaptive Systems. In: Proceedings of the 9th International Workshop on Modelling in Software Engineering, IEEE Press, pp 12–18
23. Kahani N, Bagherzadeh M, Cordy JR, Dingel J, Varró D (2019) Survey and Classification of Model Transformation Tools. Software & Systems Modeling 18(4):2361–2397, DOI 10.1007/s10270-018-0665-6, URL https://doi.org/10.1007/s10270-018-0665-6
24. Kim M, Viswanathan M, Ben-Abdallah H, Kannan S, Lee I, Sokolsky O (1999) Formally Specified Monitoring of Temporal Properties. In: Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on, IEEE, pp 114–122
25. Kim M, Lee I, Sammapun U, Shin J, Sokolsky O (2002) Monitoring, Checking, and Steering of Real-Time Systems. Electronic Notes in Theoretical Computer Science 70(4):95 – 111, DOI http://dx.doi.org/10.1016/S1571-0661(04)80579-6
26. Leucker M, Schallhart C (2009) A Brief Account of Runtime Verification. The Journal of Logic and Algebraic Programming 78(5):293–303
27. Levy J, Saidi H, Uribe TE (2002) Combining Monitors for Runtime System Verification. Electronic Notes in Theoretical Computer Science 70(4):112 – 127
28. Magee J, Kramer J (1999) State Models and Java Programs. Wiley
29. Mayerhofer T, Langer P (2012) Moliz: A Model Execution Framework for UML Models. In: Int. Master Class on Model-Driven Engineering: Modeling Wizards (MW'12)
30. Meredith PO, Jin D, Griffith D, Chen F, Roşu G (2011) An overview of the MOP runtime verification framework. International Journal on Software Techniques for Technology Transfer 14(3):249–289, DOI http://dx.doi.org/10.1007/s10009-011-0198-6
31. Nguyen XT, Tran HT, Baraki H, Geihs K (2015) FRASAD: A Framework for Model-Driven IoT Application Development. In: Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on, IEEE, pp 387–392

32. Ober I, Coulette B, Lakhrissi Y (2008) Behavioral Modelling and Composition of Object Slices Using Event Observation. In: International Conference on Model Driven Engineering Languages and Systems, Springer, pp 219–233
33. OMG (2016) Semantics of a Foundational Subset for Executable UML Models (fUML). Standard, Object Management Group, http://www.omg.org/spec/FUML/1.2.1
34. OMG (2017) Precise Semantics of UML State Machines (PSSM) v1.0. Standard PSSM/1.0/Beta1, Object Management Group, http://www.omg.org/spec/PSSM/1.0/Beta1/. In process
35. OMG (2017) Precise Semantics of UML State Machines (PSSM) v1.1. Standard PSCS/1.1, Object Management Group, http://www.omg.org/spec/PSCS/1.1/PDF. In process
36. OMG (2017) Unified Modeling Language v2.5.1. Standard UML/2.5.1/, Object Management Group, https://www.omg.org/spec/UML/2.5.1/PDF
37. Persson P, Angelsmark O (2015) Calvin–Merging Cloud and IoT. Procedia Computer Science 52:210–217
38. Posse E (2015) PapyrusRT: Modelling and Code Generation. In: Workshop on Open Source for Model Driven Engineering (OSS4MDE'15)
39. Raspberry Pi Foundation (2016) Raspberry PI 3 Model B Product Page - Raspberry PI. https://raspberrypi.org/products/raspberry-pi-3-model-b/, accessed: 2016-06-06
40. Roşu G, Chen F (2012) Semantics and Algorithms for Parametric Monitoring. Logical Methods in Computer Science 8(1):1–47, DOI http://dx.doi.org/10.2168/LMCS-8(1:9)2012, short version presented at TACAS 2009
41. Selic B (1998) Using UML for Modeling Complex Real-Time Systems. In: Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98), pp 250–260
42. Selic B (2008) Accounting for Platform Effects in the Design of Real-Time Software Using Model-Based Methods. IBM Systems Journal 47(2):309–320
43. Swartout W, Balzer R (1982) On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM 25(7):438–440
44. Taylor RN, Medvidovic N, Dashofy E (2009) Software Architecture: Foundations, Theory, and Practice. Wiley