Calur: an Action Language for UML-RT

Nicolas Hili
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: hili@cs.queensu.ca

Ernesto Posse
Zeligsoft
Gatineau, Quebec, Canada.
Email: eposse@zeligsoft.com

Juergen Dingel
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: dingel@cs.queensu.ca

***Abstract*—UML for Real-Time (UML-RT) is a profile of UML specifically designed for real-time embedded (RTE) systems. It has a long, successful track record of application and tool support via, e.g., IBM Rational RoseRT, IBM RSA-RTE, and now Papyrus-RT. Papyrus-RT is an Eclipse-based, open-source modelling and development environment for UML-RT systems. It allows the generation of complete, executable code from models and advances the state-of-art via support for model representation with mixed graphical/textual notations and an extensible code generator.**

Together with commercial UML-RT tools, Papyrus-RT currently uses C/C++ as the action language to support the definition of behaviour. However, the use of a powerful, general-purpose language such as C/++ can also easily break the abstraction that UML-RT wants to offer developers (e.g., developers have to be familiar with some of the intricate details of the C/C++ syntax and semantics) and greatly complicates almost any kind of analysis. To address this issue, action languages have been proposed for, e.g., UML. However, no suitable action language for UML-RT exists yet. This paper introduces Calur, a proposed action language for UML-RT, intended to be integrated within Papyrus-RT. We describe the syntax and semantics of Calur, and a preliminary implementation.

Keywords: Real-time Embedded Systems; UML-RT; UML; Action Language; MDE; State Machines.

I. Motivation

Model-Driven Engineering (MDE) is often touted as an approach that can tackle the challenges of developing complex, yet robust RTE systems. Modelling languages such as UML-RT [1] aim at raising the level of abstraction to, e.g., reduce the effort required to deal with lower level concerns such as data serialization, low-level concurrency management, platform-specific aspects, etc.

To promote the development of industrial-strength open source tools for the development of RTE systems, the Eclipse PolarSys Working Group was recently created [2]. The group currently consists of 23 members from industry and academia and supports 24 projects that focus on different aspects of embedded systems development including modelling and tracing. Eclipse Papyrus for Real-Time (Papyrus-RT) [3], [4] is a project within PolarSys; currently in version 0.9, version 1.0 is scheduled to be released in July 2017. Papyrus-RT is based on the Papyrus/Eclipse platform and was designed to be extensible, allowing users to add, with relative ease, their own customizations or extensions. Its target audience are industrial developers who want to build custom solutions, researchers who want to prototype and evaluate novel techniques, and educators who want to teach students the strengths and weaknesses of modelling and MDE.

One of the existing pitfalls of Papyrus-RT is its lack of a specific action language to be used in order to model the behaviour of systems. Currently, C/C++ is used for modelling transition effects and state entry/exit action code. The use of a general-purpose language such as C++ as action language inside models has several disadvantages. Firstly, it creates a tight dependency between the model and the target language in which the code is generated. Therefore, it prevents users from generating code in multiple languages, reducing the maintainability of models developed with the modelling tool. Secondly, it requires users to be familiar with some intricate details of the C++ syntax and semantics. On the one hand, this makes it hard for the tool to offer good support for the development of correct and reliable models (e.g., using a self transition triggered with a periodic timer instead of a loop statement for periodically executing a task). On the other hand, using C/C++ as an action language enables users to bypass the abstraction and encapsulation mechanisms offered by the modelling language. For example, a user can choose to directly access a method of a specific object via a reference to its pointer, breaking the encapsulation UML-RT is built around. Thirdly, as a programming language is not specific to the syntax and semantics of the modelling language, it makes it difficult to offer users a user-friendly action language editor with features such as syntax highlighting, auto-completion, auto-formatting, etc.. Finally, the complexity of C/C++ makes many analyses very difficult to perform.

Several attempts have been made to equip UML state machines with a dedicated action language that is fully aligned with UML. Examples include fUML [5], ALF [6], and Precise Semantics of UML State Machines (PSSM) [7]. However, such attempts are focusing on UML only and are therefore not specific to UML-RT. As UML-RT is a subset of UML with its own run-time services and semantics, a specific action language is required to make the best use of it.

This paper introduces CALUR, a proposed action language for UML-RT, and its integration in Papyrus-RT. The remainder of this paper is structured as follows: Section II introduces the core modelling concepts of UML-RT; CALUR is introduced in Section III and an implementation is sketched in Section IV; New opportunities and challenges are discussed in Section V; Section VI presents related work; Section VII concludes.

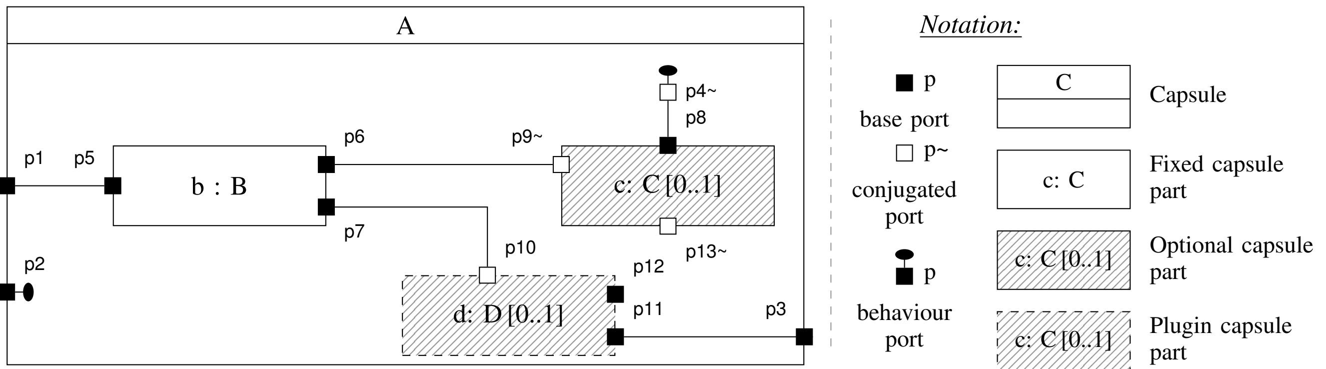

Fig. 1: A UML-RT Structure (Capsule) Diagram

II. UML FOR REAL-TIME (UML-RT)

In this section we describe informally the main concepts of UML-RT, in particular we describe the notions of capsules and structure diagrams in II-A and state machines in II-C. While UML-RT covers other types of UML diagrams, we focus only on these two, as they are the most important for UML-RT modelling. For more information on UML-RT we refer the reader to [1], [8], [9]. The official account of UML can be found in [10], [11].

A. Capsule Diagrams

UML-RT allows for modelling a system's structure through *structure diagrams*, also called *capsule diagrams*. Capsule diagrams are essentially UML Composite Structure diagrams, where the classifiers in the structure are so-called *capsules*. Fig. 1 shows a typical UML-RT capsule diagram.

A *capsule*, as its name suggests, is a highly encapsulated active entity, which may have some behaviour specified via a state machine (see II-C). It is in fact, a UML active class. Capsules may execute concurrently with other capsules and communicate with them only by sending and receiving signals through *ports* (p_1, p_2, ..., p_{13} in Fig. 1). Ports in different capsules are linked by *connectors*. A connector links only two ports[1]. Each port has a type specified with a *protocol*, which identifies signals that can be sent or received via the port. Communication may be asynchronous or synchronous. Capsules can have internal structure consisting of *capsule parts*. A capsule part can be thought of as a "slot" or placeholder for other capsule instances. A part is a UML Property whose type is a capsule. External ports of these parts are connected (wired) statically or can be connected at run-time. Connected ports must implement the same protocol and be "compatible", i.e., the *output (send) signals* of one port must be the *input (receive) signals* of the other port and vice versa. In this case, one of the ports is said to be the *base* port and the other the *conjugate* port, e.g., p_6 and p_9 in Fig. 1. A port marked with $\sim$ implements the conjugated version of a protocol, with the input and output signals inverted.

[1]A port may be replicated so that it consists of many port instances, with each instance connecting to one and only other port instance

The set of ports of a capsule defines its *interface*. There are five kinds of ports: *external ports*, *relay ports*, *internal ports*, Service Access Points (SAPs) and Service Provision Points (SPPs). External ports are ports linked to external capsules, and used directly by the capsule's state machine (if it has one) to either send or receive messages (e.g., port p_2 in Fig. 1). Relay ports are ports directly connected to some sub-capsule (thus relay messages between some external capsule and some sub-capsule, e.g., ports p_1 and p_3 in Fig. 1). Internal ports are used to communicate between the capsule's state machine and some sub-capsule (e.g., port p_4 in Fig. 1).

Some ports such as p_{12} and p_{13} may be declared as *unwired*, but they may become connected or *wired* at run-time by explicit actions on the part of the capsules that own these ports. This is achieved when one of the ports is registered at runtime by its capsule as a *service provision point* or SPP for short, under a unique service name, and the other port is registered by its capsule as a *service access point* or SAP for short, under the same unique service name. When both ports are registered (which may be done asynchronously), a new connector links them. It is also possible to deregister ports and reregistering them, thus allowing a dynamic reconfiguration of the connections among capsules. SPPs and SAPs are typically used when the modelled architecture consists of service layers to access services in the underlying layer or platform.

There are three kinds of capsule part: *fixed*, *optional* or *plug-in*. A fixed part is one where its instance(s) is(are) created (resp. destroyed) when its containing capsule is created (resp. destroyed) and is permanently attached to its containing capsule. An optional part is one where its instance(s) may be *incarnated* (i.e., created) or destroyed at a different time, but is(are) still owned by the containing capsule. Plug-in parts are "placeholders" for capsules which can be "imported" and "deported" dynamically, but are not owned by the containing capsule, so they can be shared between different capsules. In Fig. 1, b is a fixed part of type B, c is an optional part of type C, indicated by its pattern, and d is a plug-in part of type D, indicated by its pattern and dashed border.

B. Runtime semantics

Conceptually a capsule instance is an executable entity running concurrently with other capsule instances, so it can be viewed as a *logical thread*. Nevertheless, each capsule instance can be assigned to some *physical thread*, i.e. a thread in the underlying platform. That is, several capsule instances can share the same real system thread or be on their own.

More precisely, the run-time system consists of one or more *controllers*, each of which runs in its own physical (OS) thread. A controller is like an interpreter for capsules, which manages their lifecycle and executes their behaviour. It contains an *event pool* for all events whose intended receiver is a capsule instance allocated to the controller[2]. The controller divides the execution of capsule by processing each event in the pool at a time. This enforces *run-to-completion* semantics, this is, when the controller processes an event, this event is passed to its target state machine and it is processed fully, one at a time, executing an entire transition chain from the state machine's current (stable) state, executing all actions in the chain and possibly passing through several pseudo states until it reaches another (stable) state and before processing the next available event in the pool. This ensures that a capsule's state machine cannot be interrupted while processing an event, which avoids preemptive interruptions and all the concurrency issues that go with it.

C. State Machine Diagrams

The behaviour of a capsule is specified using UML-RT state machines [1], [8] which are similar to UML state machines [10], [11] but with some key differences. Fig. 2 shows a typical UML-RT state machine diagram.

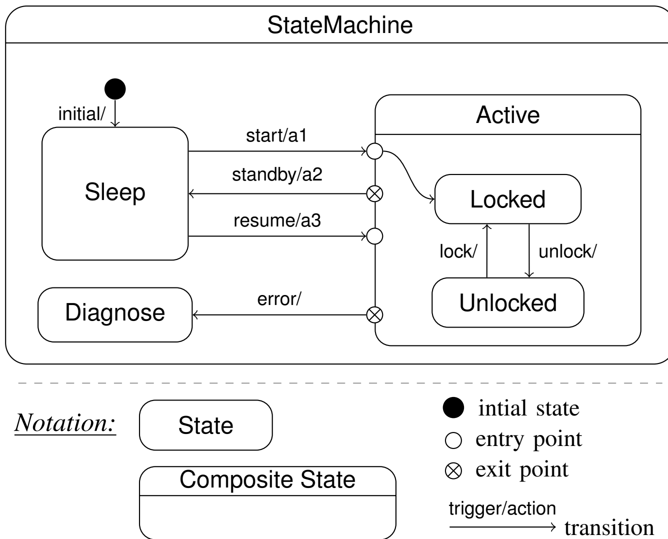

Fig. 2: A UML-RT State Machine Diagram

[2]The event pool is shared between all capsule instances allocated to the same controller. This distinguishes UML-RT from other common concurrency models where the queue is per process/thread or per port.

A UML-RT state machine has hierarchical states and guarded transitions, which are triggered by events received on ports. Each state declares its entry and exit actions and transitions have effects, so they can contain actions that are to be executed when the transition is fired. Just like in standard UML state machines, event handling in UML-RT state machines will follow a "run-to-completion" semantics: a state machine will handle one and only one event at a time, and any transition chain enabled will be fully followed and its actions fully executed before the next event is handled.

However, there also are some important syntactic and semantic differences between UML-RT state machines and UML state machines:

1) UML-RT state machines cannot contain "and-states" (orthogonal regions). All states are "or-states". So, during execution a given UML-RT state machine can be only in at most one simple state.
2) Transitions in UML-RT state machines are not allowed to cross state boundaries and they may have explicit *entry* and *exit* points, here collectively called *connection points* (in UML terminology, *entry* and *exit pseudo-states*). Hence, to represent a boundary-crossing transition, it must be broken up into segments, where each segment links connection points, either at the same level of nesting, or between a state and an immediate sub-state. During execution, connected segments form *a transition chain*, which is executed as one step.
3) In UML-RT entry points are by default connected to deep history pseudo-states. Suppose a composite state n is the target of a transition and that the associated entry point is *not* linked to a sub-state of n. If n has not been previously visited and there is an initial transition pointing to the *default state*, then the initial transition is followed and the default state entered. If, however, n has been visited previously, then the last sub-state visited in n is entered. If it has not been visited and there is no initial transition, no sub-state is entered and the state machine remain "at the border" of n. This policy is applied recursively. Hence, entering a state can be interpreted as "resuming computation where it previously left off". In standard UML state machines, on the other hand, it is possible not to connect entry points to deep history pseudo-states, but to "shallow" history pseudo states, or to the boundary of the state, in which case an initial state is always entered, if an entry point is not explicitly connected to a sub-state. Since all states have deep-history semantics, we avoid the common notation of depicting deep history pseudo-states explicitly, to avoid clutter in the diagrams.
4) Actions may be related to concepts specific to UML-RT such as capsule operations. In particular an action may send an event through a port, create or destroy an optional capsule, import or deport a plug-in capsule, connect or disconnect unbound ports, and perform normal operations on objects.

5) UML-RT supports timing requirements using a special timing protocol and internal ports which implement this protocol. A capsule, which contains a port that implements the timing protocol, can schedule an event by sending a signal through this port. Scheduling can be a part of the entry or exit behavior of a state or as an action on a transition. After a specified amount of time, the capsule will receive a timeout event from the port which it can process as any other signal.

D. Time

In UML-RT, time is assumed to progress according to an external timing service (usually provided by the underlying platform). The timing service adheres to a timing protocol with a distinguished *timeout signal* and a period or a deadline. The timing service is accessible by UML-RT models through a standard port with the corresponding timing protocol, so time signals can be treated as any other signal. Since the timing service is external, it can proceed in any way that maintains time consistency, *i.e.*, if two timers with timeout signals tmo_1 and tmo_2 are set up at the same time t_0 with timeouts $t_1 > t_0$ and $t_2 > t_0$ respectively, and such that $t_1 < t_2$, then the timing service must guarantee that signal tmo_1 will be triggered before tmo_2. Besides this requirement, the semantics of UML-RT does not make any assumptions about the rate of progress of these clocks. Furthermore, since UML-RT is not concerned with performance or scheduling, it makes no assumptions about the duration of specific actions. We assume that individual actions in the underlying action language take a negligible amount of time with respect to the minimum time unit of the time services used. Furthermore, other activities such as entering or exiting a state, or relaying a message on a relay port, also take a negligible amount of time. In the case of asynchronous communication between capsules, the amount of time between the sending of a message and its reception and consumption is undetermined. If a capsule is in a state listening to a normal port and a timeout port, and the environment sends the message before the timeout, the language does not guarantee that the message will be consumed before the timeout signal arrives.

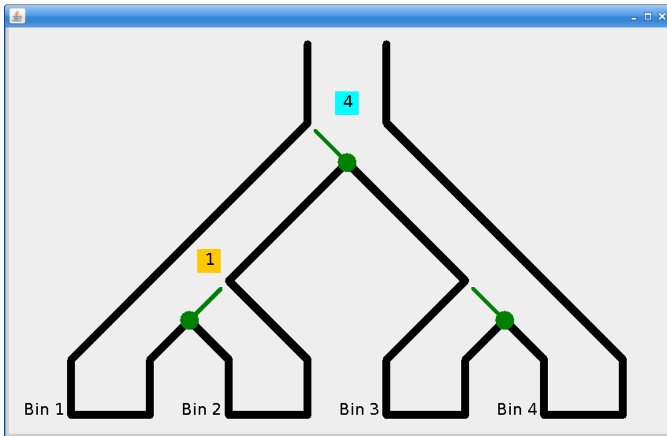

Fig. 3: Parcel Router System

E. Illustrative Example: a Parcel Router

Fig. 3 illustrates a *parcel router* system we modelled in UML-RT. A parcel router [12], [13] 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. According to the parcel tags, switchers drive the orientation of doors in order to route the parcels to the corresponding bins.

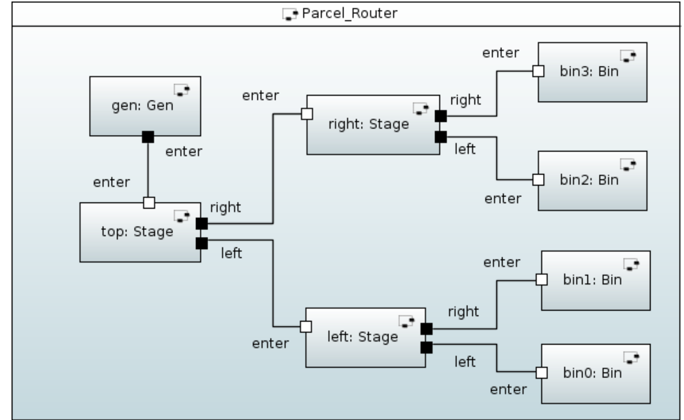

Fig. 4: Parcel Router Top Capsule Diagram

The UML-RT model consists of a single top capsule. The capsule structure of the Parcel Router is shown by the capsule diagram in Fig. 4. It consists of a *Gen* capsule that generates tagged parcels and three stages responsible for conveying parcels to one of four destination bins.

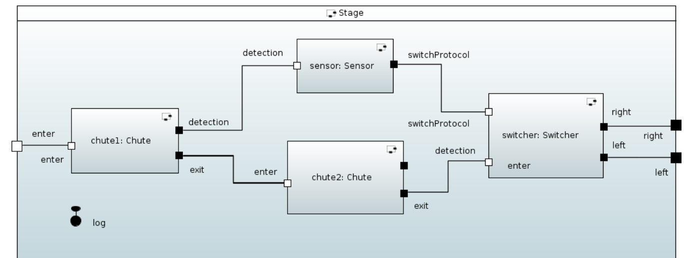

Fig. 5: Parcel Router Stage Capsule Diagram

Each stage is further decomposed into chutes, switchers, and sensors. The capsule diagram in Fig. 5 details the content of the *Stage* capsule. It contains two successive chutes that are chained with a switcher. When a parcel is conveyed by the first chute, a sensor reads the tag of the parcel and sends the value to the switcher. The latter, upon reception of the parcel, can route the parcel on its right or left side to the next stage or the four bins.

III. Core Action Language for UML-RT (CALUR)

As mentionned in section II, the behaviour of the parcel router system can be modelled using UML-RT state machines

where the actions embedded in each state and transition is written in C++. It results in the different drawbacks discussed in section I (limited analysis capabilities, misuse of C++, lack of editing support, etc.).

To address these issues, we have defined CALUR[3], an action language for UML-RT. CALUR stands for *Core Action Language for UML-RT*. It is intended to be used for writing action code in an independent way without being tight to a specific target language implementation (e.g., C++). Originally, CALUR was part of a proposal for defining a full textual notation of UML-RT models called TUMLRT [14]. To the contrary of TUMLRT, CALUR is not intended to be used as standalone but rather to specify in a textual way action code embedded in UML-RT models where the structure is still specified graphically by means of capsule and state machine diagrams, conciliating the best of both worlds.

Listing 1 is a partial Backus-Naur Form (BNF) describing the syntax of CALUR. It consists of a set of statements to, e.g., send signals, create timers, log messages, and so on.

The **send** keyword is used for sending a signal to a specific port defined by its name and an optional index. The signal corresponds to an outgoing signa defined in the protocol that is used for typing the port. Therefore, only legit signals that can be sent by this port can be used. Signals can also have arguments and a value for each argument of the signal (as defined in the protocol typing the port) must be provided. The **send** statement can behave differently following the case:

```
<statement> ::=
 send <signal−name> to <port−ref> (with <args>)? ;
 | incarnate <capsule−name> at <part−ref> <args>? ;
 | destroy <part−ref> ;
 | import <capsule−name> at <part−ref> <args>? ;
 | deport <part−ref> ;
 | register <port−ref> to <port−ref> ;
 | deregister <port−ref> ;
 | log <message> (with <args>)? ;
 | call <operation−name> ;
 | (timer <name> :) ? inform in <expr>
   <time−unit> (at <port−name>)? ;
 | (timer <name> :) ? inform every <expr>
   <time−unit> (at <port−name>)? ;
 | cancel timer <timer−name> ;
 | var <name> : <type−name> ;
 | <var−name> := <expr> ;
 | if <expr> then <statement> ( else <statement> )?
  <args> ::= ( <expr> (and <expr> )* )
  <port−ref> ::= <port−name> <index>?
  <part−ref> ::= <part−name> <index>?
  <index> ::= <expr>
  <time−unit> ::= seconds|milliseconds|microseconds
  <expr> ::= <literal>
 | <var−name>
 | <unary−op> <expr>
 | <expr> <binary−op> <expr>
 | <var−name> . <operation−name> <args>?
```

Listing 1: Calur Syntax Proposal [14]

[3]An early implementation of CALUR is available at: https://bitbucket.org/nicolas-hili/calur-experiment

1) The port is not replicated. Therefore, there can be only one recipient to which the signal is addressed. As a consequence, the communication is "unicast" and the *index* parameter of the *port-ref* is optional;
2) The port is replicated and the index parameter of the port is specified. Therefore, the recipient to which the signal is addressed is unique and identified by its index. The communication is again "unicast". Note that the replication of the outgoing port/capsule part must match the replication of the incoming port/capsule part;
3) The port is replicated and the index parameter is not specified. Therefore, the signal will be sent to all recipients connected to the specified port. The communication is "broadcast".

Note that the behaviour above holds whether the outgoing port is external, internal, relay, or SPP. If the port is SAP, then it is supposed to be connected to a unique SPP port at run-time, therefore the index parameter of the *port-ref* should be unspecified.

incarnate and **destroy** are statements used in the context of optional capsules that can be dynamically incarnated and destroyed, while **import** and **deport** are statements specific to the use of plugin capsules. <capsule−name> designates the type of capsules that has to be incarnated or imported, and <part−ref> the properties referring to the capsule part. Note that, as for ports, the *part-ref* rule in Listing 1 allows for specifying a specific index, in case the capsule part is replicated.

register and **deregister** can be used for registering a specific port to another port or de-registering it. They can be used to bind a SAP port to a SPP one, as well as normal ports whose registration is not set to automatic.

log is a specific statement for logging messages to the console. It includes a mandatory messages and optional arguments to display their values into the console.

call is a statement that can be used for invoking a specific operation defined in a capsule. As for other UML-RT elements embedding action code (states, transitions, ...), a UML-RT operation can be specified using C++ or CALUR.

inform in, **inform every**, and **cancel timer** are statements to deal with the use of timers in UML-RT. The first two allow the creation of timers that time out after a given amount of time (given in seconds, milliseconds, or microseconds) once or periodically. Both statements return a timer id (defined by the name attribute in Listing 1) that is used by the **cancel timer** statement for cancelling the timer when, e.g., a specific event occurs. The duration can be specified using a numerical value, or using another expression (such as a variable). As for the log service port, a simplified notation is provided here. If no timer service port is defined, our implementation raises an error. If one timer service port is defined, then our implementation implicitly assumes it is the one to use. However, to the contrary of log service ports, defining multiple timers is common in a UML-RT model as they may be simultaneously used for different purposes. For example, a first timer can be used to periodically execute an action while a second timer

can timeout if no new message is received during a certain amount of time (e.g., for detecting a software failure). As a consequence, when multiple timer ports are defined, the user must explicitly define which timer port is used.

Variables can be created using the **var** keyword. Primitive types (integer, boolean, string, ...) can be used, as well as more complex types, such as enumerations or structured types defined in the model. Values of variables can be set, and operations of variables structured with complex types can also be invoked.

CALUR also contains an if-then-else conditional statement. However, it does not include any loop statements (*for*, *while*), to encourage users to keep the action code simple and encode loops on the level of the state machine. At first glance, this design decision may appear odd to the reader. The rational behind the removal of loop statements is to discourage the user from creating "catch-all" states and to encourage them to use instead state machine and control nodes. The same rational would also hold for conditional statements and we do agree that using state machine choice points and guards is more consistent. However, conditionals in state entry and exit are still useful, for example when a specific operation returns a success code and it is frequent that real-world models contain conditionals. Without conditionals, some state machine diagrams can rapidly become unwieldy. However, conditionals should be parsimoniously used, and an excessive amount of conditionals in a model can be the sign of bad design. Research in design patterns and anti-patterns can help figure out when the use of conditionals is disproportionate compared to the amount of appropriate control nodes (guards, choice points, etc.) [15].

Also supported (but not shown in Listing 1) are standard arithmetic functions (e.g., *random*, *round*) and logical operators (*and*, *or*, etc.).

IV. IMPLEMENTATION

The following describes our implementation using Papyrus-RT. First, we give an overview of how we implemented CALUR into Papyrus-RT. Then, we detail the implementation of specific validation rules and their corresponding quick fixes.

A. Implementation Overview

CALUR has been implemented in Papyrus-RT using Xtext, an Eclipse-based framework for implementing textual Domain Specific Languages (DSLs) described as LL grammars. Xtext leverages the ANTLR parser and provides a full toolset including linker, typechecker, and compiler, as well as editing support in Eclipse.

Fig. 6 shows the integration of CALUR within Papyrus-RT. For this, we extended the *Code Snippet* view (1) that is initially used for adding action code when a graphical element (transition or state) is selected. For a state, the view contains two tabs "entry" and "exit" in order to add action code for the entry and exit action of the selected state. When a transition is selected, then the code snippet view contains two tabs, respectively for the transition effect and guard of the transition. Other elements can be used as well. For example, guards of choice points can also be defined in this view. The view and all of these tabs currently support C++, but do not provide any editing support (such as syntax highlighting, content-assist, or auto-formatting) or error checking.

To support CALUR with error checking and other "smart" editor features, we slightly modified this view (cf. Fig. 6). The main area is an Xtext textual editor supporting the CALUR grammar and providing the required features for e.g., content-assist and syntax highlighting. CALUR can be used for modelling the sequence of actions that are executed for each state entry and exit, and transition effect. It can also be used for modelling guards. Currently, the following features have been implemented:

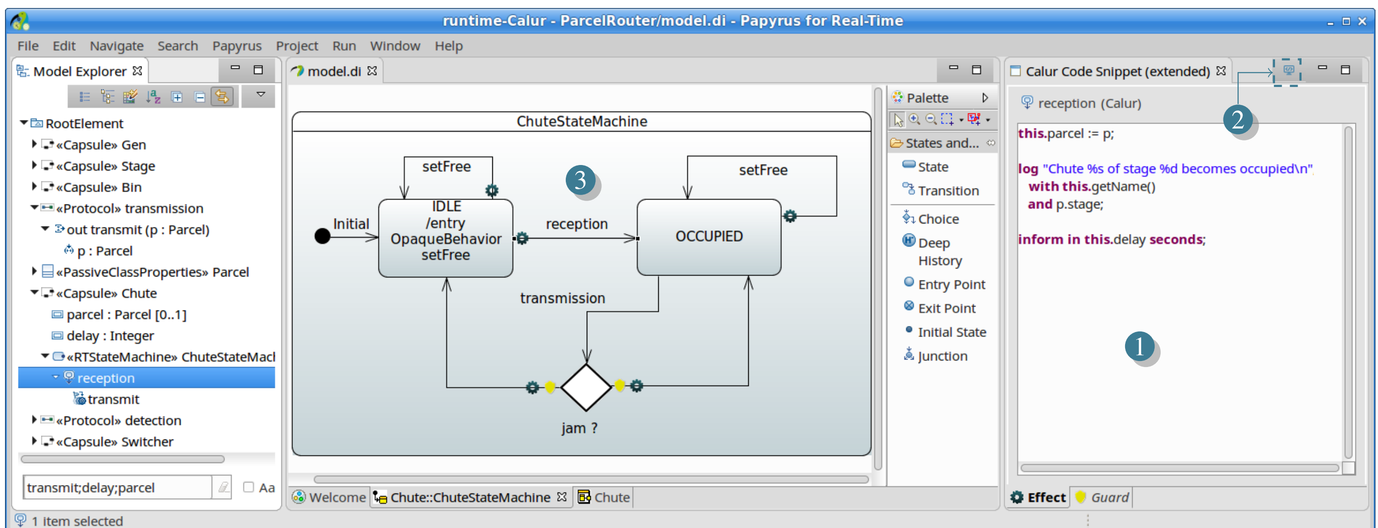

Fig. 6: CALUR Integration with Papyrus-RT

1) *Syntax Highlighting/Content Assist:* Using the extended view, the syntax of action code written in CALUR is highlighted and content assist is possible for assisting users in writing action code. Content assist allows for recognizing the names of UML-RT model elements in the view.
2) *Validation and Quick Fixes:* Validation is possible for helping users correct errors. Examples of errors the validation checks for include detecting misspelled references to UML-RT model elements, the use of timers while no timer port has been defined, and so on. In addition, some quick fixes have been implemented. For example, if the user uses the **log** statement while no *log* port has been instantiated, the editor automatically proposes to add the corresponding *log* port.
3) *Code Generation:* Xtend is used for automatically generating code from CALUR in any target language defined in Papyrus-RT (currently, only C++ is available). Anytime, it is possible to switch back to the C++ view by clicking on the button we have implemented 2. Therefore, it is possible not to use CALUR at all and revert back to C++, or even to modify the C++ code that is generated from a specific piece of action code in CALUR.

Fig. 6 on page 6 illustrates how CALUR can be used along with the Papyrus-RT graphical diagrams to model the parcel router. The main area 3 represents a state machine diagram to model the behaviour of a chute of the parcel router. It consists of two main states *IDLE* and *OCCUPIED*. Initially, a chute is idle, waiting for an upstream component to forward a parcel (identified by its number, current level, and current stage). In UML-RT, we modelled the transmission of a parcel from one chute to another as a signal sent by the upstream chute. The signal carries the parcel as a parameter p. When the signal is received, it triggers the *reception* transition, causing the chute that received the parcel to go from *IDLE* to *OCCUPIED*. Upon reception of the parcel, a timer is modelled to represent the duration the parcel stays in the chute before being transmitted to the next one. The timer causes the *transmission* transition to be fired, effectively transmitting the parcel to the next chute. Additional constructs are modelled to prevent jam.

The right side of Fig. 6 shows the CALUR action code used for modelling the *reception* transition of the *Chute* state machine. The first line indicates that the message parameter p (which corresponds to the parcel that is transmitted) is stored into the *parcel* attribute of the capsule. Lines 3–5 log a message. Finally, the last line initializes a timer that will timeout after a certain delay. Note that *delay*, *parcel*, and p are visible in the model explorer in Fig. 6. Both *delay* and *parcel* are attributes owned by the *Chute* capsule. p is a parameter of the *transmit* signal that triggers the *reception* transition.

B. Name Resolution

Our implementation ensures the consistency and integrity of the actions that are written in CALUR. Consistency checking relies on the Xtext internal mechanism. Each model element in UML-RT is uniquely identified with a qualified name. Our implementation provides filtering rules (also called *scoping* in Xtext) in order to limit the number of potential candidate references used in CALUR. The *context* in which the action code is specified allows us to reduce the risk of "name collision", hence allowing us to replace the qualified name of model elements with simple names. For example, in Fig. 6 on page 6, the context of the action code is the *reception* transition, hence by transitivity, the *Chute* capsule. Therefore, **this**.parcel automatically refers to the *parcel* property of the chute (and not any other *parcel* property that could exist in any other capsules in the model) and p automatically refers to the parcel parameter that is carried by the message that triggered the *reception* transition.

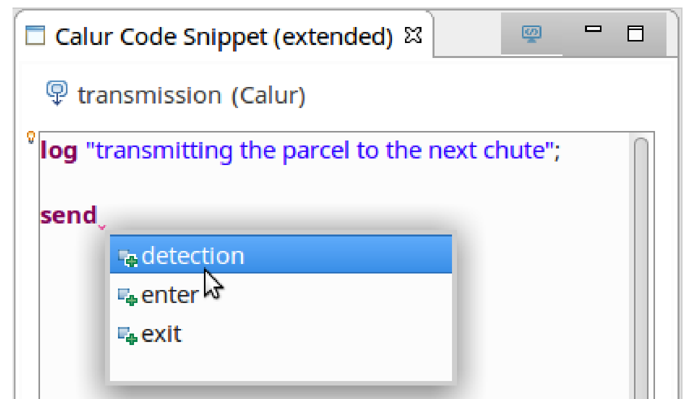

Fig. 7: Content-assist

The filtering of potential candidates is also used for providing the user with content-assist capabilities. Fig. 7 illustrates an example of use of the content-assist capabilities. When hitting ctrl + ␣, the list of potential candidates that matches the first letters (if any) entered by the user.

Note that filtering reduces the risk of name collision but does not entirely remove it. In Fig. 6, if we consider that the *parcel* signal parameter is named *parcel* rather than p, CALUR is capable of distinguishing both parcels in the line **this**.parcel = parcel because of the context when both reference instances are used. For the former, CALUR is expecting the name of an attribute of the capsule to be typed while for the latter, it is expecting the name of either a local variable (that was created with the **var** keyword) or a signal parameter that is defined in the protocol that types the incoming port of the capsule from which the signal that triggered the transition was received. The previous sentence is voluntary long to show the different relations between the constructs of UML-RT (transitions, ports, protocols, signal, etc.) that can be used to determine the context of the action code and to resolve names.

As a second example, consider this time a local variable named *parcel* that is created (using the **var** keyword) within the *reception* transition. In that case, CALUR would not be able to distinguish the local variable from the signal parameter when resolving the second occurrence of *parcel* in the line

this.parcel = parcel. Note that the same problem remains in any programming language as well.

C. Error Checking and Quickfixes

Our implementation also supports a certain number of errors that can be checked and quickfixes that can be proposed to the user when an error is found.

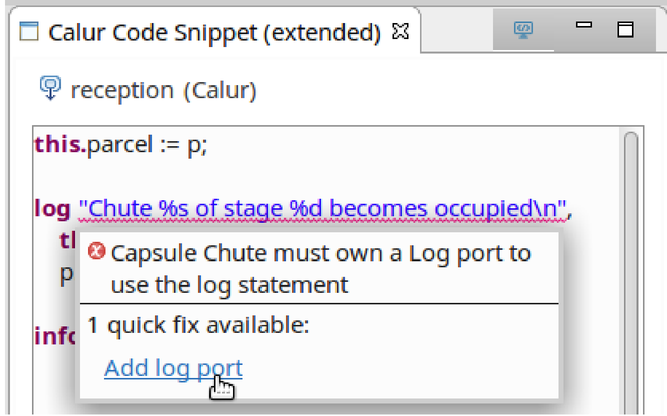

Fig. 8: Incorrect Use of the Log Statement

Fig. 8 shows an incorrect use of the **log** statement when the capsule does not contain a log port. In UML-RT, the logging service is provided via a service port. If no log port is defined, our implementation raises an error. A description of the error is given. A quickfix is also suggested to the user to automatically add a log port in the capsule.

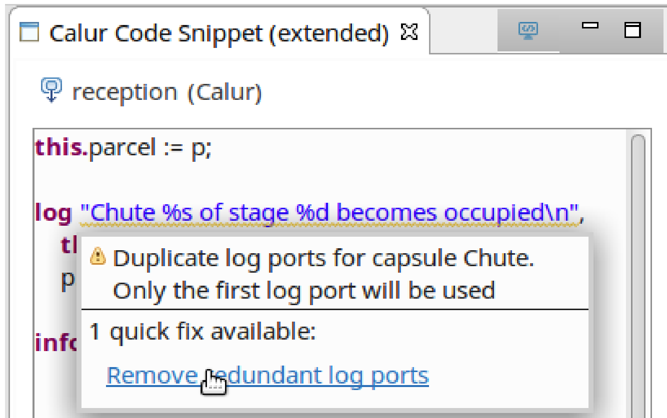

Fig. 9: Warning Message for Duplicated Log Ports

The example illustrated in Fig. 9 shows a second example of an incorrect use of the **log** statement when the capsule contains this time multiple log port. While it is authorized in UML-RT to have multiple log ports, using one or multiple ports does not affect the behaviour. However, using multiple ports burdens the reading of the model and can be the source of possible errors when a redundant port is deleted. The severity of the error is defined as "warning" and a quickfix is also suggested to automatically remove all redundant log ports.

TABLE I: Explicit Error Checking and Suggested Quickfixes

Statement	Description	Severity	Quickfix
log	No log port defined	Error	Add a log port
log	Multiple log ports defined	Warning	Remove redundant ports
inform in	Multiple timer ports while no explicit reference to a timer port name	Error	Explicit use of a timer port name
incarnate	Multiple frame ports defined	Warning	Remove redundant ports
import	Multiple frame ports defined	Warning	Remove redundant ports

Table I shows some explicit error checking we have implemented. Note that CALUR (through Xtext) implicitly ensures the consistency of the action code according to the name resolution and the filtering processes we discussed in Section IV-B and therefore, those errors are not represented in Table I. For example, CALUR is not only capable of detecting misspelled port names, but it can also detect that a **send** statement is incorrectly used whenever the user attempts to send a signal through a port while the signal is not defined as an outgoing signal of the protocol that types that port. Such errors are automatically detected by CALUR (without introducing an explicit rule) as the incorrectly used message will simply not be considered as a possible candidate by CALUR during the filtering process.

D. Code Generation

As mentionned earlier, our implementation automatically translates action code written in CALUR into the target programming language defined for the model. As such, CALUR can be seen as a pivot language. For the moment, only C++ is supported by Papyrus-RT. Listing 2 shows the CALUR action code of the *reception* transition of Fig. 6.

```
1 this.parcel := p;
2
3 log "Chute %s of stage %d becomes occupied\n" with
4   this.getName() and
5   p.stage;
6
7 inform in this.delay seconds;
```

Listing 2: CALUR Action Code

Listing 3 shows the resulting C++ code that is generated from it. As the parcel attribute is a complex type, each composed attribute is set individually, according to the values of *p*. Both log and timer statements in C++ are part of the run-time service of Papyrus-RT and have specific syntaxes, automatically generated from CALUR.

```
1 this->parcel = Parcel();
2 this->parcel.number = p.number;
3 this->parcel.level = p.level;
4 this->parcel.stage = p.stage;
5 log.show("Chute %s of stage %d becomes occupied\n",
       this->getName(), p.stage);
6 timer.informIn(UMLRTTimespec(this->delay, 0));
```

Listing 3: Resulting C++ Code Generated from CALUR

V. New Opportunities and Challenges

The definition of an action language that contains an interpretable semantics opens up interesting opportunities in terms of potential analyses we can drive. Model simulation, interpretation, and debugging are, among others, powerful and highly desired techniques, yet almost impossible to perform due to the complexity of C++. Leveraging the use of an interpretable action semantics for modelling the actions in states and transitions in UML-RT models makes powerful analyses possible.

To illustrate the new opportunities CALUR bring, we have implemented a first prototype of a UML-RT model interpreter using Moka [16]. Moka is a Papyrus project to interpret UML activity diagrams and state machines. It fully supports The Foundational UML (fUML) [5] and provides some extension mechanisms to implement new engines. That makes it suitable for supporting new action semantics such as CALUR. It also provides interesting features for model-level debugging and animation [17], [18]

We have developped a first prototype of an engine capable of supporting the action semantics of UML-RT in order to interpret UML-RT state machines augmented with the CALUR action language. As of today, our prototype only supports basic UML-RT constructs such as message passing. Therefore, it does not support the interpretation of complex UML-RT models. However, it shows evidence of the cruciality of dedicated action languages rather than plain C++ to support powerful analyses.

Our current prototype also reveals new challenges coming out when implementing a model interpretor for UML-RT models. The task is challenging as the model must be interpreted exactly the same way the code is executed on the target platform. Semantically, there is no difference between the UML-RT model and the code that is generated from it. Both are models that conform to a specific metamodel and that are interpreted by an interpretor. The generated code is compiled by GDB into an executable binary interpretable by a specific operating system. Similarly, the UML-RT model conforms to the UML metamodel and is interpreted by model interpretors such as Moka. Consequently, there should be no difference between the action semantics of the UML-RT model and the generated code. That conclusion can be hard to prove, as the action semantics of the execution is implicitly encoded in the way the code is generated and in the way the run-time service of the tool (e.g., Papyrus-RT) makes sense of it. If a gap exists between the semantics of execution, the interpretation of a UML-RT model can result in a different output than the real execution of the system, making the analysis worthless.

Existing approaches and tools can be used for defining a single semantics of execution of UML-RT for the different target models (C++ and UML-RT). The Gemoc Studio [19] is an Eclipse-based project for the complete definition of DSLs. In the Gemoc Studio, the syntax of a DSL is separated from its semantics, which allows the automatic generation of model interpretors using the Eclipse Debugger. To go one step further, one could envision to use the Gemoc Studio for defining the UML-RT syntax and semantics, and for automatically generating both the run-time service of Papyrus-RT and the model interpretor in Moka.

For the moment, validation and code generation are performed only after editing an action in the *Calur Code Snippet* view (2) (cf. Fig. 6). A second place where automatic validation and code generation has to be done is when a structural element of a capsule is changed. Consider for example the CALUR code in Listing 2. The last line implicitly assumes that only one timer is defined in the *Chute* capsule. Consider now that the user decides to add a second timer in the *Chute* capsule. The action code above is now invalid as the **inform in** statement has to explicitly define which of the two timers it has to be used. This decision also affects the subsequent code generation process. This example shows the complexity of the problem of ensuring the consistency of hybrid textual/graphical models.

VI. Related Work

A certain number of initiatives were done to equip UML with a formal semantics. Precise Semantics Of UML Composite Structures (PSCS) [20] and PSSM [7] are two proposition specifications from the Object Management Group (OMG). The two extend fUML [5] in order to add support for modelling and execution of composite structures and state machines. Both PSCS and PSSM are "compatible" with UML-RT and other variants of UML, such as xtUML [21]. Compatibility means that the execution semantics of PSCS/PSSM include most of the execution semantics of UML-RT among others as they account for issues like run-to-completion, dynamic incarnation, dynamic wiring, allocation of capsules to controllers, or timers. As such, CALUR can be seen as a concrete syntax that specializes the execution semantics of PSCS/PSSM to cope with other UML-RT constructs.

Other action semantics have been defined. RPL [22] is an action language used for building ObjecTime models (RPL is no longer supported in more recent tools such as IBM Rational RoseRT). RPL can be seen as a facade for SmallTalk and supports dynamically typed and interpreted action code. CALUR is in the same spirit of RPL (RPL used the concept of "actor classes" which is equivalent to the UML-RT capsules used in CALUR), although some syntactical differences exist. RPL follows the object-oriented notation where features are owned and exposed by objects (e.g., p.msg().send(. . .)). In CALUR we chose a *verb-first* style (e.g., "send message"). The first reason is that it can be read more easily. The second reason is more of a technical/pragmatic nature. Having a keyword first allows for an early disambiguation of rules parsed by the LL parser used by Xtext, simplifying the definition and maintenance of the grammar.

Object Action Language (OAL) [23] is another language designed by Mentor Graphics [24] and used in BridgePoint XtUML [21]. OAL is used to define the semantics of actions in structural and behavioural diagrams. It allows for defining the action code for modelling element such as states, operations,

or functions. The goals are similar to the ones achieved by CALUR. However, it lacks of dedicated structures for covering the semantics of UML-RT models (such as dynamic incarnation and wiring, plugin capsule imports, etc.).

The Action Language for Foundational UML (Alf) [6] provides a concrete textual syntax for fUML. Alf/fUML are intended to become the standard for defining executable UML models. It is today implemented in the last version of MagicDraw [25]. However, fUML does not provide any support for UML profile such as UML-RT. Besides, like OAL, Alf does not cover all the concepts of UML-RT. A decision for CALUR could have been to create a UML-RT Alf library. However, it appeared to us that these concepts are too essential to the language's semantics to be relegated to a library (even if they are implemented as functionality of a library) and deserve to be first-class constructs for analysis purpose.

Other tools integrate custom action languages. This is the case for example for Yakindu StateChart Tools [26]. Yakindu StateChart Tools provides a small textual language for state charts and supports code generation in Java, C, and C++. As such, it follows the same goal of CALUR of having a pivot language used to generate code in specific programming languages. Yakindu StateChart Tools also support integration of legacy C code (for the professional edition only). In comparison, CALUR also supports out of the box the integration of legacy code with the ability of switching between CALUR and any target language.

VII. CONCLUSION

In this paper, we have presented CALUR, a proposed action language for UML-RT. CALUR is a pivot language that can be used for writing action code for different UML-RT modelling elements (states, transitions, guards, choice points, etc.) independently of any target language implementation. We have sketched a prototype implementation that is intended to be integrated in Papyrus-RT. Our implementation supports CALUR with syntax highlighting, content-assist, error checking, and other "smart" features. It also supports code generators for C++ with the possibility of switching between C++ and CALUR in order to support legacy code integration.

We have also discussed the implementation of a prototype for model interpretation using Moka of UML-RT models augmented with CALUR. This shows evidence of the need of a platform-independent language semantics and opens up interesting directions in model analyses that could be more easily supported in Papyrus-RT.

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

REFERENCES

[1] B. Selic, "Using UML for modeling complex real-time systems," in *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, 1998, pp. 250–260.
[2] "PolarSys Working Group Homepage," https://www.polarsys.org/.
[3] "Papyrus for Real Time (Papyrus-RT)," https://www.eclipse.org/papyrus-rt, accessed: 2016-03-10.
[4] E. Posse, "PapyrusRT: Modelling and Code Generation," in *Workshop on Open Source for Model Driven Engineering (OSS4MDE'15)*, 2015.
[5] "Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models (fUML)," 2017, http://www.omg.org/spec/FUML/1.3.
[6] "Concrete Syntax For A UML Action Language: Action Language For Foundational UMLâĎć (ALFâĎć)," 2017, http://www.omg.org/spec/ALF/1.1/. In process.
[7] "Precise Semantics Of UML State Machines (PSSM) v1.0," 2017, http://www.omg.org/spec/PSSM/1.0/Beta1/. In process.
[8] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object Oriented Modeling*. Wiley & Sons, 1994.
[9] E. Posse and J. Dingel, "An Executable Formal Semantics for UML-RT," *Software & Systems Modeling*, vol. 15, no. 1, pp. 179–217, 2016.
[10] Object Management Group, "UML Superstructure Specification v2.4.1," http://www.omg.org/spec/UML/2.4.1/, Aug. 2011.
[11] ——, "UML Superstructure Specification v2.5," http://www.omg.org/spec/UML/2.5/, Sep. 2012.
[12] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, vol. 25, no. 7, pp. 438–440, 1982.
[13] J. Magee and J. Kramer, *State Models and Java Programs*. Wiley, 1999.
[14] Z. . Ltd., "Tumlrt: Textual Syntax for UML-RT Reference Guide," 2016, version 0.8.
[15] T. K. Das and J. Dingel, "Model Development Guidelines for UML-RT: Conventions, Patterns and Antipatterns," *Software and Systems Modeling*, 2016.
[16] "Papyrus: Moka Overview," 2016, http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.
[17] N. Das, S. Ganesan, L. Jweda, M. Bagherzadeh, N. Hili, and J. Dingel, "Supporting the Model-driven Development of Real-time Embedded Systems with Run-time Monitoring and Animation via Highly Customizable Code Generation," in *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS'16)*, ser. MODELS'16. Saint-Malo, France. October 2-7, 2016: ACM, 2016, pp. 36–43.
[18] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, Platform-independent Debugging in the Context of the Model-driven Development of Real-time Systems," in *11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*, ser. ESEC/FSE'17. Paderborn, Germany. September 04-08, 2017: ACM, 2017, pp. 419–430.
[19] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution Framework of the Gemoc Studio (Tool Demo)," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2016, pp. 84–89.
[20] "Precise Semantics Of UML State Machines (PSSM) v1.1," 2017, http://www.omg.org/spec/PSCS/1.1/PDF. In process.
[21] C. Starrett, "xtUML: Current and Next State of a Modeling Dialect," in *EXE@ MoDELS*, 2016, pp. 33–37.
[22] *RPL Language Guide*. ObjectTime Developer.
[23] BridgePoint, "Object Action Language Reference Manual v1.5," https://xtuml.org/wp-content/uploads/2012/09/Object_Action_Language_Reference_Manual2.pdf.
[24] Mentor, "Mentor Graphics Homepage," https://www.mentor.com/.
[25] No Magic, "MagicDraw Homepage," https://www.nomagic.com/products/magicdraw.
[26] Itemis, "Yakindu StateChart Tools Homepage," https://www.itemis.com/en/yakindu/state-machine.