

Live Modeling in the Context of State Machine Models and Code Generation

Mojtaba Bagherzadeh · Karim Jahed ·
Benoit Combemale · Juergen Dingel

Received: date / Accepted: date

Abstract Live modeling has been recognized as an important technique to edit behavioral models while being executed and helps in better understanding the impact of a design choice. In the context of Model-driven Development (MDD) models can be executed by interpretation or by the translation of models into existing programming languages, often by code generation. This work is concerned with the support of live modeling in the context of state machine models when they are executed by code generation. To this end, we propose an approach that is completely independent of any live programming support offered by the target language. This independence is achieved with the help of a model transformation which equips the model with support for features which are required for live modeling. A subsequent code generation then produces a self-reflective program that allows changes to the model elements at runtime (through synchronization of design and runtime models).

We have applied the approach in the context of UML-RT and created a prototype (Live-UMLRT) that provides a full set of services for live modeling of UML-RT state machines such as re-execution, adding/removing states and transitions, and adding/removing action code. We have evaluated the prototype on several use-cases. The evaluation shows that (1) generation of a self-reflective and model instrumentation can be carried out with reasonable performance, and (2) our approach can apply model changes to the running execution faster than the standard approach that depends on the live programming support of the target language.

1 Introduction

Model-Driven Development (MDD) is a model-centric software development approach in which models serve as the primary software development artifacts, rather than source code [19]. Thanks to existing MDD tools (e.g., [52], [46]) many facilities are available to simplify software development using models specifically in the domain of Real-time Embedded Systems (RTE). One of the main facilities is the execution of models, which is supported either by interpretation or by the translation into existing programming languages, often by code generation (translational execution) [18].

Live programming [15, 57] aims to free developers from the “edit-compile-run” cycle, and allows them to change programs at runtime and get immediate feedback on the change. Often, a form of live programming is supported by several existing programming languages and Integrated Development Environments (IDEs) (e.g., [35, 58, 59, 65, 74]), and its benefits and utility are discussed in several studies (e.g., [16, 53, 56]). Inspired by this line of work, some efforts [85, 88, 90] have recently been made towards live modeling, i.e., the application of the changes to the models while they are being executed. However, they only have focused on the support of live modeling in the context of model interpretation (when models are executed by interpretation), and no work supports live modeling when the models are executed by code generation into general programming languages.

As suggested by [90], one possible way of implementing live modeling is by leveraging specific features provided by the target programming language of the generated code. For example, in C/C++, shared libraries could be used to dynamically load updated version of specific parts of the code. While the simplicity of this approach is appealing, it still suffers from several issues:

- (1) **Edit latency.** Changing a running execution to reflect an updated model typically involves several steps: (a) incremental code generation, (b) building the generated code, and (c) injecting the new artifact into the running program by, for instance, hot-swapping, inter-positioning code, or dynamic linking [2, 32]. Unfortunately, these steps are pretty time consuming and they often take much more than the 500ms threshold that, if exceeded, can greatly degrade the users’ experience [55, 56]. Obviously, larger models exacerbate this problem.
- (2) **Dependency on the target language of the generated code.** Live modeling support is limited by the capabilities of the target language of the generated code. Different programming languages provide different levels of support for live programming. For instance, many programming languages only support *fix-and-continue* [15] which allows only a limited set of code changes excluding, for example, state transfer [42]. A change to a model element may require runtime updates that are not supported by the target language, and the lack of support for state transfer typically requires restarting the execution of the model for the effects of the change to become visible. Not to mention that live programming support needs to be re-implemented for every programming language that the code generator supports.
- (3) **Inconsistent runtime state.** As discussed in [85], some changes such as

removing the active state of a state machine can invalidate its runtime state. Mechanisms to recover from or even prevent these inconsistencies require a deep understanding of the code generation process and may be challenging to implement. In addition, since modeling tools often support code generation in several languages, the differences in capabilities provided by those languages with respect to live programming will make it difficult to provide support for live modeling that is uniform, consistent, and user-friendly.

This work is concerned with the support of live modeling in the context of state machine models when they are executed by code generation. To overcome the above-mentioned problems, we propose an approach that is completely independent of any live programming support offered by the target language. This independence is achieved with the help of model transformation [51] and code generation techniques. The approach consists of two phases: (1) *Generation of a self-reflective program* which is realized through: (a) automatic instrumentation of models using model transformation techniques to allow for controlling the execution and generation of the execution traces, (b) generation of reflective code that allows not only introspection of the program execution at runtime, but also changes to the model elements (through a synchronization of design and runtime models), and (c) creation of a live modeling plugin that hooks into the execution of the model, and uses the self-reflective features of the generated code to provide live modeling services. (2) *Live modeling* which is directly provided via the interaction with the self-reflective program. This decreases the edit latency significantly since there is no need for code generation, compile & build, and hot-swapping for each edit.

We have applied the approach in the context of UML-RT (a language for the modeling of real-time systems) [76], and created a prototype (*Live-UMLRT*) that supports the live modeling of UML-RT models. To maximize the impact of our work, our implementation is publicly available [4], and only uses *open source* tools such as the Papyrus-RT MDD tool for modeling, the Papyrus-RT code generation extension to generate self-reflective code, and the Epsilon [52] tools for model transformation. *Live-UMLRT* provides a full set of services for live modeling of UML-RT state machines, such as a execution replay mechanism that prevents inconsistent states, adding/removing states and transitions, and adding/removing action code. We have evaluated *Live-UMLRT* on several use-cases. The experimental evaluation shows that (1) generation of a self-reflective program and model instrumentation can be carried out with reasonable performance, and (2) our approach can apply model changes to the running execution much faster than the standard approach that depends on the live programming support of the target language (i.e., minimize edit latency).

The rest of this paper is organized as follows. In Section 2, we provide necessary terminologies and formal notations, a running example, as well as the scope of the work. Section 3 describes our approach for live modeling, followed by Section 4 that explains how the approach can be applied to UML-RT and presents *Live-UMLRT*, a tool that supports live modeling of UML-RT models and embodies our approach. We discuss our evaluation and its results

in Section 5. We review related work in Section 6, and conclude the paper with a discussion, summary, and directions for future research.

2 Background

2.1 UML-RT

To be able to illustrate our approach, we use the UML profile for Real-Time systems (UML-RT). UML-RT [69, 76] is a language specifically designed for Real-Time Embedded (RTE) systems with soft real-time constraints. Over the past two decades, it has been used successfully in industry to develop several large-scale industrial projects (e.g., [25]), and has a long, successful track record of application and tool support, via, e.g., IBM RSA-RTE [46], RTist [41], Eclipse eTrice [28], and Papyrus-RT [27]. In the following we present a concise formalization of UML-RT which is required to understand our approach and its application. A detailed discussion of UML-RT can be found in [69, 76].

Definition 1 (Read function (Projection))

Let tp be a tuple $\langle r_1 \dots r_n \rangle$ where $r_1 \dots r_n$ refer to the names of the tuple entries. We use $tp.r_i$ to read the value of entry r_i . E.g., to read the value of entry *name* of tuple $person\langle name, family \rangle$ we can use $person.name$.

In UML-RT, a system is designed as a set of interacting capsules. A capsule is similar to an active class in object-oriented programming. Being active implies that each capsule may have autonomous behaviour. Capsules own a set of *ports* that are typed with *protocols*. A protocol defines the different incoming and outgoing *messages* that a capsule can receive or send through its ports. A port is the only interface for the communication between the capsules, which guarantees high encapsulation. Ports of two capsules can be connected through *connectors* only if they are typed with the same protocol. A port can be conjugated which means that the direction of messages is reversed. Furthermore, capsules can have *attributes*, *operations*, and *parts* (a.k.a. sub-capsules) [75, 76].

Capsules' behaviour is modeled using *UMLRT State Machines (USM)*. An USM consists of several *states* connected with *transitions*. States can be of three kinds: basic states, composite states (containing sub-states), and pseudo-states (e.g., initial pseudo-state, choice-point). A basic or composite state can have *entry* and *exit* actions that are executed when the state is entered or left, respectively. A *transition* connects a *source* state to a *target* state. It may contain a *triggering event*, a *guard*, and an *action*. A transition is taken when the triggering event is received and the guard evaluates to true. When it is taken, the action of the transition is executed. Entry, exit, and transition actions are expressed using an action language.

Definition 2 (Modeling Structure of an RTE System)

We define a protocol as a set of pairs (m, d) , where $m \in M_U$ (i.e., a universal set

of messages) is a message, and $d \in \{input, output\}$ specifies whether a message is consumed (*input* message) or produced (*output* message). A message can have a payload, which is a set of values conveyed by the message. We define a capsule as a tuple $\langle P, V, B \rangle$, where $P \subseteq P_U$ (i.e., a universal set of ports) is a set of ports, V is a set of variables, and β refers to the specification of the capsule's behavior. A port is defined as a pair $(t, conjugated)$, where $t \in \mathcal{I}$ refers to the type of the port, and $conjugated \in \{true, false\}$ specifies whether or not the port is conjugated (the direction of messages of conjugated ports is reversed). Finally, we define the structure of an RTE system as a tuple $\langle \mathcal{C}, \mathcal{I}, con, in \rangle$, where $\mathcal{C}$ is a set of capsules, $\mathcal{I}$ is a set of protocols, con is a connectivity relationship $\subseteq P_U \times P_U$, and in is an acyclic containment relationship $\subseteq \mathcal{C} \times \mathcal{C}$. Whenever two ports p_1, p_2 are connected by con (i.e., $(p_1, p_2) \in con$) then both have the same type (i.e., $p_1.t = p_2.t$) and exactly one of them must be conjugated. This condition ensures that connected ports are 'compatible'.

Definition 3 (Action Language)

Action languages support primitive operations such as accessing/updating variables, arithmetic/boolean expressions, control flow constructs, and sending/receiving messages. MDD tools provide action languages either by adapting a subset of well-known programming languages or by creating a specific action language. E.g., Papyrus-RT uses a subset of C++ as the action language, the UML *Alf* action language [63] is designed for UML, and *YAKINDU* [47] provides its own action language. In this work, we assume the existence of an action language with the standard capabilities, but not define a particular syntax for it.

Definition 4 (UML-RT State Machine (USM))

We specify the behavior of a capsule c using a UML-RT state machine (*USM*) that is defined as a tuple $\langle \mathcal{S}, \mathcal{T}, in \rangle$. $\mathcal{S} = \mathcal{S}_b \cup \mathcal{S}_c \cup \mathcal{S}_p$ is a set of states, $\mathcal{T}$ is a set of transitions, and $in \subseteq \mathcal{S}_c \times (\mathcal{S} \cup \mathcal{T})$ denotes an acyclic containment relationship. States can be basic ($\mathcal{S}_b$), composite ($\mathcal{S}_c$), or pseudo-states ($\mathcal{S}_p$). Basic states are primitive states that the execution stays in until an outgoing transition is triggered. Composite states encapsulate a sub-state machine. Pseudo-states are transient control-flow states. There are six kinds of pseudo-states, including *initial*, *choice-point*, *history*, *junction-point*, *entry-point*, and *exit-point*, (i.e., $\mathcal{S}_p = \mathcal{S}_{in} \cup \mathcal{S}_{ch} \cup \mathcal{S}_h \cup \mathcal{S}_j \cup \mathcal{S}_{en} \cup \mathcal{S}_{ex}$). Composite and basic states can have entry and exit actions that are expressed using an action language.

Definition 5 (Transition)

Let $inp(c)$ refer to the messages that can be received by capsule c . A transition t is a 5-tuple $(src, guard, trig, act, des)$, where $src, des \in \mathcal{S}$ refer to non-empty source and destination of the transition respectively, $guard$ is a logical expression coded using the action language, $trig \subseteq inp(c)$ is a set of messages that trigger the transition, and act is the transition's action also expressed using the action language.

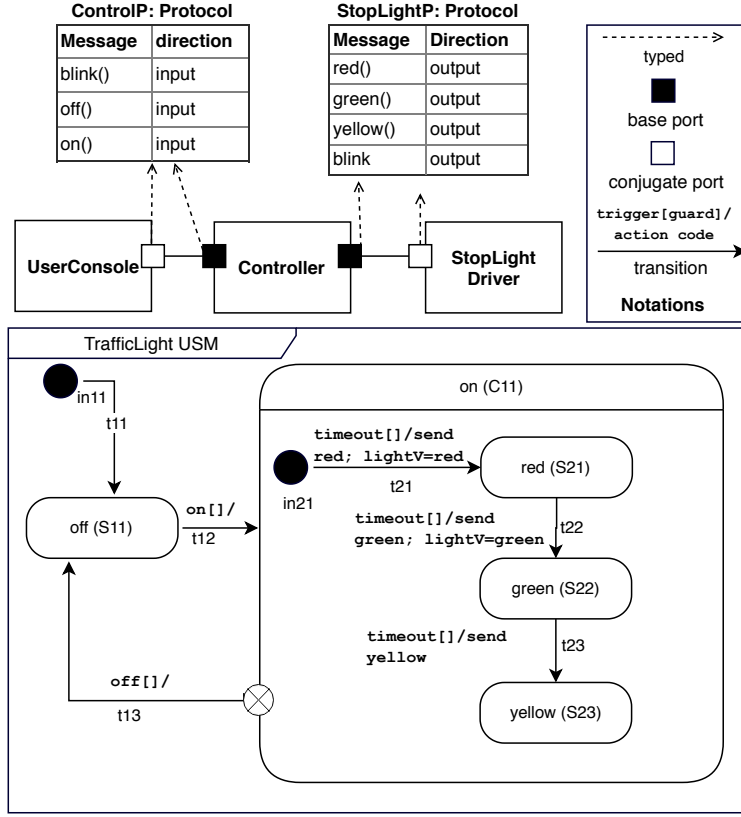


Fig. 1: Model of a traffic light in UML-RT

Definition 6 (Assumptions concerning well-formedness and execution semantics of *USMs*)

Following [27, 46, 69], we make the following assumptions concerning the well-formedness and execution of *USMs*:

- Only transitions that start from a choice-point can have a guard, and no transition that starts from a pseudo-state can have a trigger.
- There are no orthogonal regions (i.e., a USM cannot be in two basic states at the same time), and there is no support for the UML concepts *fork*, *join*, *shallow history*, and *final states*.
- States do not have idle (*do*) actions.
- There is no notation for history. Instead, any transition to a composite state is assumed to end in an implicit history state inside the composite state (i.e., when returning to a previously activated composite state, the execution of the sub-state machine will resume from its last active state).

- Triggers of transitions starting from the same basic or composite state must be disjoint, i.e., $\forall t1, t2 \in \mathcal{T} : t1.src = t2.src \wedge t1.src \notin \mathcal{S}_p \implies t1.trig \cap t2.trig = \emptyset$.

Note that these rules are also enforced by existing UML-RT tools and none of them has been defined specifically for this study. Still, our approach can be extended to support AND-states and other concepts not offered in UML-RT.

2.2 Running Example

We use the control system of a simple traffic light as a running example. The top-level structure of the system is shown in Figure 1 which consists of three capsules: *UserConsole* (*UC*), *Controller* (*CTR*), and *StopLightDriver* (*SLD*). *CTR* is connected to *UC* and *SLD* using two base ports, which are typed by protocols *ControlP* and *StopLightP* accordingly. The *UC* capsule collects user input, which it passes on to the *CTR* capsule, the capsule controlling the light. *CTR* sends the control actions as messages to the *SLD* which transfers these messages through a hardware port to the traffic light. The behaviour of *CTR* is shown in the USM *TrafficLight* in the Fig. 1 which is intentionally left incomplete to demonstrate live modeling features. Note that variable *lightV* is used to save the current color of the light.

2.3 Live Modeling

Model execution is a key enabler for the performance of validation and verification (V&V) activities of complex system behaviors. For instance interactive debugging, testing, model checking, and trace analysis are some of the V&V techniques that have proven useful to analyze the behavior of complex systems based on executable models.

However, with the way model execution infrastructures and V&V tools for executable models are currently built, the development process for executable models is very rigid, requiring the modeler to re-start an execution or analysis whenever the need for changes on the model arises. In particular, in the current practice, domain engineers are forced to follow a fixed process for exploring changes on their executable models: First update the model, then run the execution that consists of several steps depending on the execution mechanism (interpretation or code generation), examine and analyze the observed behavior, stop the execution and return to updating the model. This edit-execute-edit cycle significantly limits the agility of the development process of executable models and thus also developer productivity.

Live modeling can free modellers from the edit-execute-edit cycle by providing continuous, real-time, and interactive feedback to modelers about the impact of their changes on the execution of a model. As a result, it becomes much easier for modellers to explore the design space of the system they are

developing. This not only eases the development of complex software-intensive systems, but also facilitates learning modeling languages and has the potential to improve the quality and efficiency of the development process [85, 90].

Similar to live programming, there is no single approach to realizing live modeling as the applicable techniques depend on the type of modeling language and its execution mechanism. While the main goal of the live modeling always is to shorten the edit-execute-edit cycle and provide the developer with more immediate feedback on the changes, the way this goal is achieved can differ greatly. Below we collect some of the main criteria that can be used to distinguish and classify live modeling. The criteria capture user-level capability differences, but also implementation-level technical differences. Concepts and examples from live programming are used for illustration due to the higher degree of maturity of live programming.

1. **Responsiveness.** This criterion concerns how quickly and readily feedback is provided to the user in response to changes in the model. With the highest degree of responsiveness (live, instantaneous feedback) the effect of a change is reflected immediately and without the need for user intervention. Domains that offer tools with a very high degree of responsiveness include live music production, visual signal processing, or graphical user interfaces. Examples of languages used include Viva [81], Max Language [72], and Vivide [80]. The supporting language infrastructures maintain a mapping between program elements and their views and provide live feedback through rapid, continuous data flow between the elements and their views. Supporting such a mapping and the data flow is not possible in many domains and languages, including many modern general purpose programming languages including imperative languages such as Java. Apart from live feedback, two other techniques can be used to get timely, yet slightly less responsive, feedback: (a) *Read-eval-print loop (REPL)*: A REPL processes user inputs (e.g., a single expression or statement), evaluate or execute it, and display the result. A *REPL* is intended for quick feedback, and its scope is often only limited to simple expressions or statements. (b) *Fix-and-continue*: This technique, supported by many modern IDEs, allows code to be changed while the program is executing. However, the execution is updated upon user request. Often, this update requires pausing the program execution before the update and resuming the execution after the update.
2. **State transfer.** This criterion concerns how the relevant aspects of the current execution state of the running program (data segment) are migrated after the execution (code segment) has been updated. There are two main techniques for state transfer: (a) *Real time*: This technique keeps the current state of the program intact and allows the changes to only affect the next execution steps (e.g., [35]). This mechanism is often used in environments that support *fix-and-continue*. In some domains in which certain code blocks in the program are executed continuously (e.g., visual programming) or the past executions are not important (e.g., music perfor-

mance), this method can work well. However, it has several problems and does not deal well with, e.g., changes that are applied to instructions that have already been executed [56]. (b) *Re-execution*: This technique recreates the execution state by replaying the execution traces while changes are being taken into account (e.g., [23]). This technique is an expensive approach and introduces extra complexities and can delay immediate feedback.

3. **Execution mechanism.** Models can be executed by interpretation or code generation. With interpretation, live modeling features and mechanisms can be freely selected and implemented depending on the modeling language. However, with code generation, the support for live modeling is limited based on the services provided by the target language, i.e., the programming language that the generated code is in. Also, in the context of code generation, extra challenges are imposed due to the need for bi-directional mapping of the concepts of the modeling language to concepts of the target language.
4. **Supported operations and their scope.** This criterion targets the edit operations supported (e.g., add/remove/update) and their scope. For instance, most of the IDEs for imperative programming languages only allow updates of method bodies. The addition of new methods or changes to the signature of a method are not allowed. Often, the reasons of these limitations are technical, implementation-level challenges, or the desire to improve performance or simplify the implementation.
5. **Integration with debuggers.** How a program executes is often not apparent from its output. This is particularly true for complex systems. Live programming only updates certain views of the program execution, which may or may not support one of the most important development activities sufficiently well: debugging. Thus, even in the presence of live modeling, there likely still is a need to use debugging services such as tracing, replaying and stepping through executions to be able to locate and correct bugs. This is particularly important when the execution does not produce appropriate output. As a result, many live programming (e.g., [59]) approaches integrate live programming services with debugging facilities to provide effective debugging support.

2.4 Scope of our work

In terms of these criteria, the focus of our work is to develop support for live modeling in the context of state machine models that are executed using code generation, more specifically live modeling using UML-RT models when they are executed by generation of *C++* code. We aim to propose a solution that supports fix-and-continue, real time state transfer, and integration with debugging services to provide REPL capability. Integrating real time state transfer with debugging support is challenging. Suppose, e.g., that the action of transition *t23* is changed when the execution of the traffic light is in state *yellow*. In that situation, there is no way for users to ever see the effect of the

Element	Add	Remove	Update
<i>Capsule</i>	No	No	No
<i>Port</i>	No	No	No
<i>Protocol</i>	No	No	No
<i>Variable</i>	Yes	No	No
<i>State</i>	Yes	Yes	Yes
<i>Transition</i>	Yes	Yes	Yes
<i>Action</i>	Yes	Yes	No
<i>Trigger</i>	Yes	Yes	Yes
<i>Guard</i>	Yes	Yes	No

Table 1: Supported Edit Operations

change. To deal with this challenge, we also provide a limited type of execution replay which will be discussed in Section 4.

In theory, every element of the model can be edited (update/add/delete) during a live modeling session. However, as shown in Table 1, our work only focuses on changing state machines. Changes to, e.g., the structure (by, e.g., adding or removing a port or capsule) except for adding variables are not supported. Removing or editing a variable without restarting the execution requires a complex memory manipulation of the running model that can lead to subtle errors. In addition, we only support basic addition and removal operations involving statements to send messages and arithmetic and logical expressions. Note that there is no fundamental limitation to extending the support for addition and removal and can be achieved in the context of our prototype. Currently supported actions are sufficient to show the applicability of our approach.

Some of the possible live modeling operations on the USM *TrafficLight* include **OP1**: add actions to transition *t23* which lacks an update of variable *lightV*, **OP2**: add a transition along with actions and a trigger from state *yellow* to state *red*, and **OP3**: remove state *yellow* when it is the active execution state.

2.5 Live modeling by leveraging live programming (initial experiment)

Figure 2 shows the process for realizing live modeling using services offered by the target language of the code generation. We have implemented the process in the context of Papyrus-RT and generated C++ code to provide a fair and realistic assessment of the existing approach. The bold arrows show the steps required for the application of changes to a model while it is running. We assume that changes are applied to model *M1* which result in model *M2*. The model-level changes (i.e., $Diff(M1, M2)$) are translated into code using incremental code generation, then resulting code is compiled and shared libraries are built using code-based build tools (i.e., *gcc* and *Make*). Then the running program, generated from model *M1*, is updated to refer to the shared libraries without restarting the execution which is achieved by generating the code that

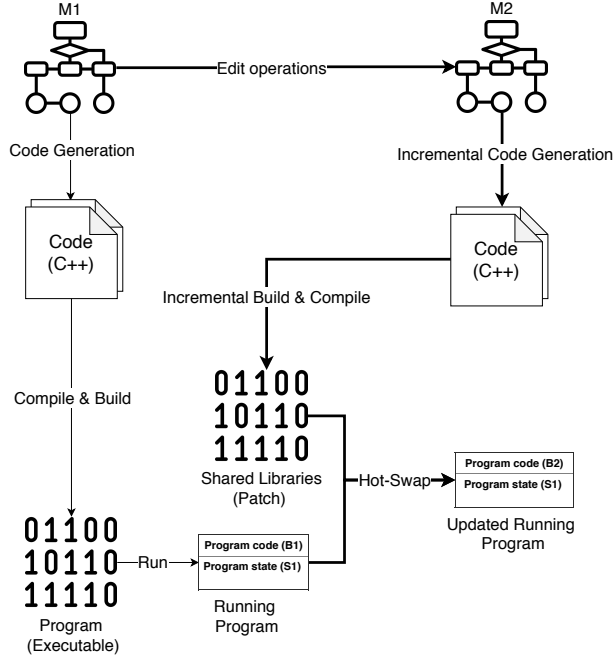


Fig. 2: Existing approach for live modeling by leveraging live programming services offered in the context of the target language

can detect changes to shared libraries and load/unload them at runtime. Note that this process is almost the same for *compiled languages* (e.g., C, C++, Java), while for *scripting languages* a much simpler process, without the requirement to compile and build, can be used. Usually, in the context of RTE systems, the target language of the generated code is a compiled language.

In addition to the complexity of the incremental code generation which often is not supported properly by existing MDD tools, we review and illustrate (in the context of the running example) the most significant challenges in the above solution:

- **Edit Latency.** The process for applying a change on a running model (i.e., incremental code generation and build, and hot swapping) is time-consuming, and our experiment shows that, on average, the process takes more than half a second for each change. This can greatly degrade user experience [55, 56] and prevents immediate feedback.
- **State transfer.** No state transfer mechanism is supported by default in the context of C++.
- **Debugging support.** In some application domains in which program execution produces continuous and observable outputs, the live update can be enough for live programming/modeling. However, in the context of live modeling, where the model execution may perform computation without

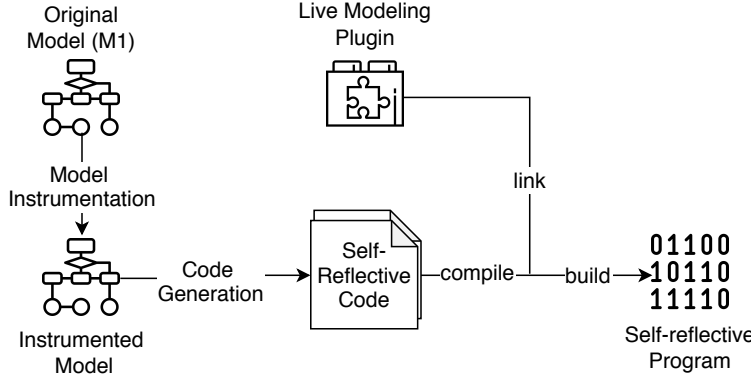


Fig. 3: Generation of a self-reflective program

observable outputs, the application of changes without the ability to observe their effect does not appear to be useful. Thus, similar to popular IDEs (e.g., Java, Eclipse, Visual C#/C++) that mix live programming with debugging, we need to provide a meaningful, effective integration with debugging to provide live feedback. This is not supported by default and requires extra efforts. Consider, e.g., changes to the way variable *lightV* is updated (*OP1*, and *OP2*). Without debugging support it is not possible to verify whether the not directly observable effects of these changes are as expected.

3 Approach

As discussed in Section 2.5, the use of services offered in the context of live programming to implement live modeling imposes several challenges and restrictions. To overcome these, we propose an approach for live modeling independent of live programming services in the context of state machine models when they are executed by code generation. The approach consists of two phases: *Generation of a self-reflective program* and *live modeling using the self-reflective program*. As shown in Fig. 3, first, code generation and model transformation techniques are used to automatically create a program (a self-reflective program) that embeds all required services for live modeling and debugging along with an interface for using them. Second, live modeling services are directly provided via interaction with the self-reflective program as shown in Fig. 4. This decreases the edit latency significantly since there is no need for code generation, compile & build, and hot-swapping for each edit. Also, model debugging services provide an infrastructure for REPLs. In the following, we provide an overview of each phase.

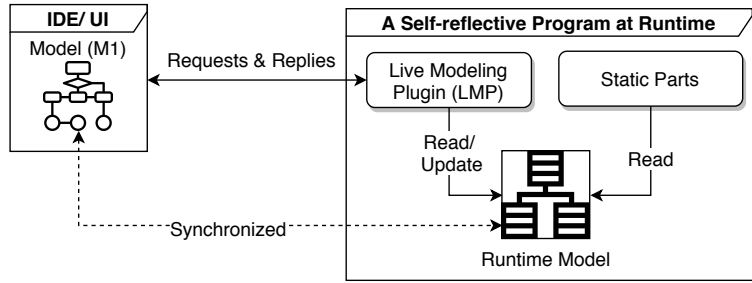


Fig. 4: Live modeling using self-reflective program

3.1 Generation of a Self-reflective Program

Fig. 3 shows the three steps for the generation of a self-reflective program capable of supporting live modeling:

Model instrumentation. In this step the model is instrumented automatically using model-to-model transformation to provide debugging services mainly for (a) variable view/changes that are required for the read-eval-print loop (REPL), (b) producing execution traces to support execution replay, and (c) adding a mechanism for pausing the execution of the model when users want to apply model changes to the execution or use the REPL. As discussed, our approach supports fix-and-continue in which the execution is updated upon user requests by pausing the program execution and resuming the execution after the update.

Code generation. The main purpose of this step is to generate a self-reflective program that allows the live update of the model elements during the execution. Similar to programming languages, the execution semantics of models is defined based on their Abstract Syntax Tree (AST). Thus, to allow model updates during the execution: (a) the AST of the model (i.e., model elements which are to be modifiable) should be explicitly embedded/defined in the generated code, (b) whenever it is required, the execution should progress using the compiled version of the embedded AST rather than the statically generated code, and (c) the embedded AST should be modifiable during execution. The complexity of this step is language-dependent. The conciseness of the abstract syntax and semantics of UML-RT simplified the implementation of our prototype.

Compile, link, and build. Existing code-based tools are used to compile and build the generated code and create the self-reflective program. Also, a live modeling plugin that provides an interface for using live modeling and *REPL* services is linked to the generated program in this step. Implementing the interface as a separate plugin allows the separation of concerns. The plugin imposes minimum overhead since it only needs to be loaded when the live modeling service is used.

3.2 Live modeling using the self-reflective program

Figure 4 shows an overview of how live modeling is supported via interaction with the self-reflective program. During live modeling, a change in the design model is translated to a live modeling command and sent to the live modeling plugin (*LMP*) which is loaded as part of the self-reflective program. The *LMP* processes the request and updates the corresponding elements in the runtime model so that the design and runtime models are kept synchronized. The static parts in the self-reflective program correspond to the generated code of the model that mainly controls the execution based on the runtime model and provides support for the self-reflection services. By separating the runtime model that captures the editable part of the model from the static part that is compiled, the self-reflective program can benefit from the best of both worlds: better performance for the static part due to compilation on the one hand, and increased flexibility and support for runtime change via interpretation on the other hand.

Also, *LMP* provides an interface for the REPL, which can be used to evaluate or execute simple expressions or statements to, e.g., inspect or modify the values of variables.

3.3 Generality of our approach

The approach proposed above can be applied in the context of different modeling languages to realize live modeling and debugging if the language has the following properties:

- Deterministic execution. Without deterministic semantics, some of the live modeling services such as execution replay cannot be realized properly.
- Essential constructs. The language should support required constructs to allow the implementation of self-reflection, interaction with external applications (e.g., message passing), and control flow (e.g., choice-points) to add debugging services into models by instrumentation (e.g., pause and resuming the execution).

Since the execution of models is a language-dependent concept, there is no way to provide a generic implementation of the framework using existing techniques and tools. However, still, the communication, self-reflection, and tracing can be implemented similarly. Also, our model transformation implementation can be ported to other state machine languages (e.g., UML state machines). Additionally, our live modeling plugin can be adapted to other modeling languages for which code generation to C/C++ is available.

4 Live Modeling of UML-RT Models

In this section, we discuss the application of our approach to live modeling in the context of UML-RT models and demonstrate a tool that illustrates it.

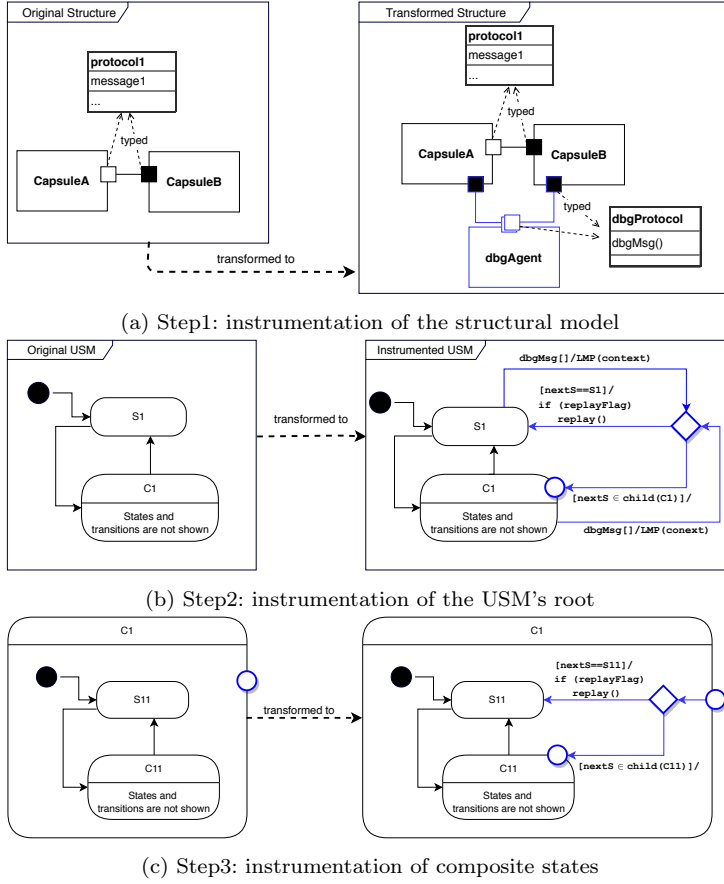


Fig. 5: Demonstration of UML-RT model instrumentation by examples. The added elements are colored blue ($child(C1)$ returns states inside $C1$).

4.1 Instrumentation of UML-RT models

We adapt the model debugging services, including variable view and message injection proposed in MDebugger [5, 7] to implement the REPL. Also, we apply model instrumentation in four steps (the first three steps are shown in Figure 5) to equip the model with the capabilities required for supporting live modeling. In the following, we discuss the rationale behind each step and the capabilities it provides.

1. *Instrumentation of the structural model.* As shown in Figure 5a, the first step of the instrumentation adds capsule $dbgAgent$ and protocol $dbgProtocol$ to the structural model, both of which have been adapted from MDebugger. Then, a port is added to each capsule that is typed with protocol $dbgProtocol$ and connected to the $dbgAgent$ capsule. Capsule $dbgAgent$ acts as a

gateway between external applications (e.g., a live modeling environment) and the capsules. It accepts external messages through *TCP* or shared memory and passes them to the relevant capsules [7]. This step allows the injecting of the *dbgMsg* message by communicating with capsule *dbgAgent* to pause the normal execution (execution of the user-defined part of the USM) of the capsule and start a live modeling session in which the model execution can be updated and the REPL service can be used.

2. *Instrumentation of the root of the USM.* Steps 2 and 3 instrument the USM such that (1) its normal execution is paused and a live modeling session is started upon injection of the *dbgMsg* message, (2) the normal execution is continued in an appropriate execution state when the live modeling session is finished.

Figure 5b shows how a the root of an USM is instrumented in Step 2. First, a choice point state (*ch*) is added to the root. Second, transitions are added from all basic and composite states in the root to *ch* whose triggers are set to *dbgMsg*. Thus, the added transitions are taken upon reception of *dbgMsg*. The actions of the transitions are set to *LMP(context)* which is a function that starts the live modeling session and is a part of the live modeling plugin. The function returns the next execution state (*nextS*) and thus specifies where the execution needs to be steered from the *ch*. Also, argument *context* of *LMP* is used to pass the relevant information to the plugin, more importantly, the current active state and a callback address that is used by the plugin to call self-reflection functions of the capsule such as add or remove state.

Third, entry-points are added to all composite states inside the root. Then, transitions are added from *ch* to all basic states and these entry-points. The guard of the transitions that end at a basic state is set to a boolean expression that checks if the basic state is the same as *nextS*. The guard of transitions that end at entry-points is set to check that *nextS* is one of the states directly contained in the composite state being entered. Also, the action of the transitions to the basic states calls function *replay* if *replayFlag* is set. Function *replay* is discussed in detail in Sec. 4.3.

By default, the execution of the capsule returns to the last active state (i.e., *nextS* is set to the most recently active state) before the live update (i.e., *fix and continue*), but the user can select other basic states for *nextS*. In that case, the live modeling plugin replays the execution based on the most recent execution traces of the basic state until the execution reaches the last state before the update or a state which has not been visited before (i.e., no execution trace is available for it).

Note that the handling of messages in UML-RT (as in many other state machine languages) is subject to the run-to-completion semantics, i.e., the processing of an incoming message will not start until the processing of the previous message has completed. Since that *dbgMsg* is a regular UML-RT message, it also is subject to run-to-completion. This means that the activation of a live modeling session is consistent with the UML-RT semantics. It also means that all messages sent by other capsules during the execu-

tion of a live modeling session are queued. Thus, the normal execution of the capsule is paused correctly during the live modeling session, and no message is lost.

3. *Instrumentation of composite state.* Figure 5c shows how a composite state is instrumented in Step 3. First, a choice-point (*ch*) is added into the composite state. Then a transition from the entry point (added in Step 2) to *ch* is added. Then, for all basic states and entry points that have been added to the composite state, transitions are added from *ch* to these states and points. The guards and actions of the transitions are set as in Step 2. This instrumentation allows the execution of the capsule to be steered to any of the states inside the composite state when the live modeling session is complete. This capability is essential to steer the execution into a desirable state when the active execution state has been removed during the live modeling session.
4. *Instrumentation of basic states.* In the last step, the exit actions of basic states are instrumented to ensure the generation of appropriate trace information before the state is exited. An execution trace is defined as a tuple $\langle state, lastM, vars \rangle$ where $state \in \mathcal{S}_b$ refers to the state being exited, *lastM* refers to the message that caused the state to be exited, and *vars* is a map recording the values of variables right before the state was exited. This information allows replaying the execution from a state based on the last time the state was visited. Also, the exit actions of states are guarded to ensure that no exit action is executed when a transition from a basic state is taken due to the reception of *dbgMsg* (i.e., when live modeling plugin is activated).

Note that the instrumentation of the model is applied before the code generation. The instrumentation also needs to be updated as the model is changed during live modeling to keep the runtime model consistent, and allows the handling of *dbgMsg*, which is required for starting *LMP*. Thus, the instrumentation is updated accordingly, based on the type of each change (ref. Sec. 4.3.3). The update of the instrumentation at runtime is achieved by updating the runtime model rather than the design model.

Figure 6 shows the result of the instrumentation of the USM *TrafficLight*. As shown, the execution in any of the basic states is paused and the live modeling is started by calling *LMP(context)* as action of *t14* or *t16* upon receiving the *dbgMsg*. Also, the execution can move to any basic state (possibly through one or more choice-point states that have been added during instrumentation).

4.2 Generation of the Self-reflective Program

As discussed in Sec. 2, the syntax for behavioural specification in UML-RT is concise and consists mainly of states, transitions, variables, and actions. In order to embed these elements in the generated code while still allowing for their modification at runtime, a runtime model is used. In what follows, we

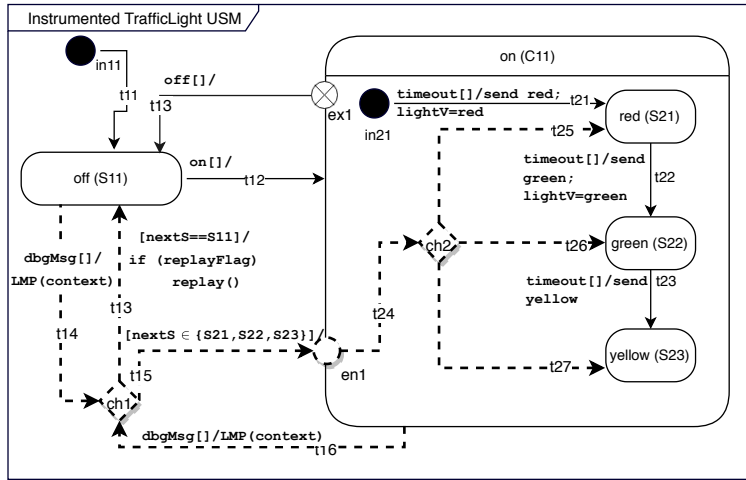


Fig. 6: Instrumented USM stoplight for state transfer and safe update (added elements are shown using dashed lines and to make the diagram more readable, only four of the outgoing and incoming transitions are labeled with a guard and trigger (t_{13} , t_{14} , t_{15} , t_{16})).

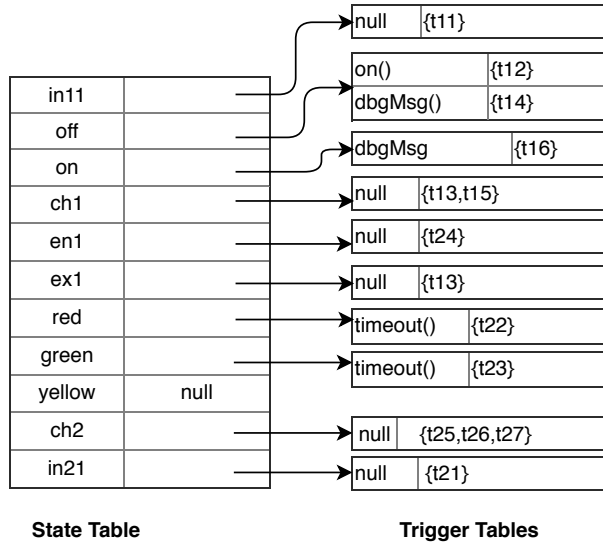


Fig. 7: State and trigger maps of the instrumented stoplight in Fig. 6

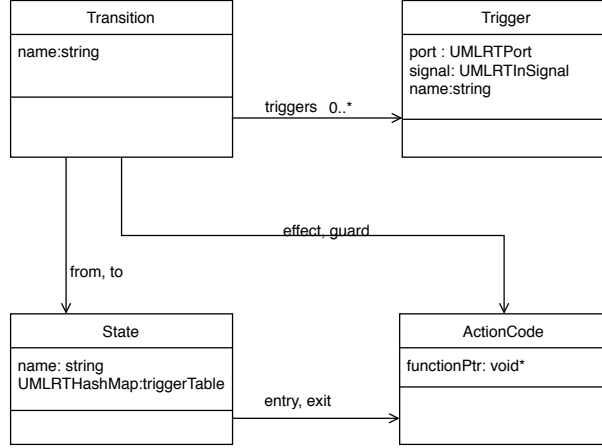


Fig. 8: Class diagram of extended classes to RTS

first describe the runtime model and then discuss how the code for the runtime model of UML-RT model is generated.

State and trigger tables. Let us define a state table of an USM as a map from its states to references to trigger tables where a trigger table is a map from a trigger to a non-empty set of transitions. A *null* reference is used when a state has no outgoing transitions. A trigger table records the outgoing transitions from a state along with their trigger. A *null* trigger is used when a transition has no trigger. Figure 7 shows how the instrumented stoplight model in Figure 6 is translated to a state and a trigger map. At runtime, a state s is represented as references to the entry and exit actions (e.g., a function pointer in C++), the parent of s , and the children of s (if any), as shown on the left of Figure 7. Similarly, for each transition, references to its source and target states, guard, and action are kept.

Runtime model. A runtime model of a system is a set of tuples $\langle c \in \mathcal{C}, sm, N_{vars}, N_{acts}, E_t \rangle$ where c refers to a capsule, sm refers to the state table of the capsule's USM, TM refers to the set of trigger tables that are referenced from sm , N_{vars} is a map of newly added variables, N_{acts} refers to the newly added or modified actions of the capsule's USM, and E_t is a map from basic states to the most recent execution trace generated before exiting the state. The generated program initializes the runtime model for the system at the beginning of the execution which is then used to support live update and feedback. Note that at the beginning of the execution the actions and guards of transitions refer to functions of the compiled code.

Generation of runtime model. We extend the code generation of Papyrus-RT [68] to generate the self-reflective program. The code generation of Papyrus-RT relies on a service library that provides basic functionalities such as communication, and base classes for capsules and protocols. Our extension for the generation of the self-reflective program is two-fold: First, we extend the

Listing 1: An excerpt of the code generated code for creation of runtime model of the running example

```

{
    addState( new UMLRTCapsule::UMLRTState( "off" ) );
    addState( new UMLRTCapsule::UMLRTState( "on::red" )); // add state red
        inside a composite state on
    //... omitted

    addTransition( getState( "off" ), new UMLRTCapsule::UMLRTTrigger(
        "controlPort", "turnOn" ), new UMLRTCapsule::UMLRTTransition(
        "off__on", getState( "off" ), ... );

    //... omitted
}

```

service library with a set of C++ classes and add functions (C++ methods) into the base class of the capsule to provide support for self-reflection and interaction with the live modeling plugin. E.g., function *addState* is added that supports addition of a state into the capsule's USM. Second, we change the code generation related to the capsule to generate methods that initialize the runtime model through the creation of appropriate states and transitions which are also added to the state and trigger tables.

Figure 8 shows the high-level class diagram of the C++ classes that are extended in the service library. As shown, for each design concept (e.g., state, and transition), a corresponding class is defined that captures not only the runtime information but also the properties that are required to capture their design properties (e.g., state names). Also, the class *ActionCode* is used to capture the actions and guards with the help of a function pointer. It is important to note that the function pointer refers to the compiled action code at the beginning of the execution, but when the action is updated at runtime during live modeling, it refers to a function proxy that is loaded as part of the live modeling plugin. The function proxy receives and interprets actions. By relying on this extension, we adapt the code generator to generate method calls for initialization of the runtime model. An excerpt of methods calls for the spotlight example is shown in Figure 1.

The explanation above provides a brief overview of the code generation process without dwelling into the technical detail. In addition, in this work, we advocate the idea of the generation of the self-reflective program. We intentionally do not constrain the code generation any further, because different solutions can be applied depending on the context of the modeling language and environment. Nevertheless, interested readers can find the source code of our implementation in [4].

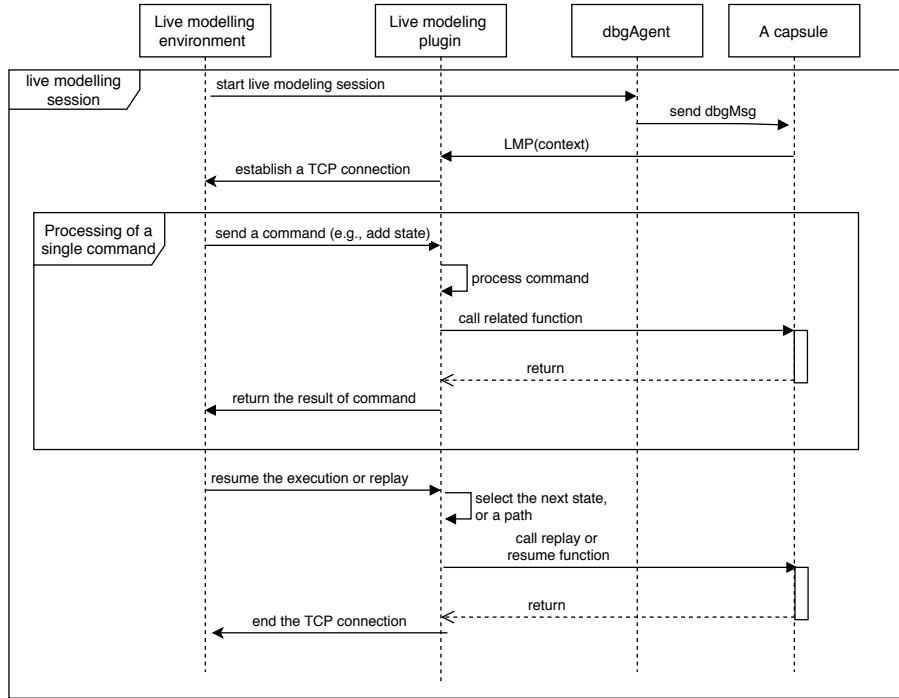


Fig. 9: The flow of live modelling via interaction with the self-reflective program

4.3 Supporting Live Modeling for UML-RT Using the Self-reflective Program

Figure 9 shows the flow of live modeling via interaction with the self-reflective program. The flow consists of the following three steps.

1. **Start a live modeling session.** An external application (e.g., live modeling environment) can update the execution of a capsule. To do that, first, a live modeling session needs to be started by sending a certain message to the *dbgAgent*. The message specifies the name of a capsule whose execution needs to be updated. *dbgAgent* processes the message and sends a *dbgMsg* to the capsule. As discussed in Sec. 4.1, reception of a *dbgMsg* triggers a transition in the capsule's USM that activates *LMP* by calling function *LMP(context)*. When the *LMP* is activated, it establishes a TCP connection with the external application, through which the live modeling services can be used by sending relevant messages.
2. **Updating the execution and using REPL.** When the live modeling session is established, the external application can use the live modeling services or REPL by sending relevant messages to *LMP* through the TCP connection. When *LMP* receives a message, it processes the message and

calls the relevant functions (the functions that are extended into the base class of the capsule) using the callback address that is passed to the LPM during its activation. Finally, *LMP* returns the result to the external application. Note that we assume the external application validates the user request (e.g., make sure the syntax of added action is valid, or the action does not make the model inconsistent) before sending the request to *LMP*.

3. **Resuming the execution of the capsule.** During the live modeling session, the normal execution (execution of user-defined parts) of a capsule is stopped and the *LMP* is activated. Thus, after finishing the update, the normal execution of the capsule needs to be resumed. This can be done either by using the replay function or simply resuming the execution. By resuming the execution, the *LMP* tries to select the next execution state (which, typically, would be the most recently active execution state). In case if the most recently active execution state has been removed during the live modeling session, *LMP* communicates with the external application to select the next execution state. The external application can select the next state via interaction with the user (our prototype supports this) or apply heuristics and other techniques to select the next execution state (e.g., [85]). We argue that the selection of the next state via communication with the user is simpler and better than other possible solutions since, during the development, it is reasonable to assume that the user is familiar with the model and can select a suitable state. Note that the user may have removed all states during the live modeling session. In that case, the execution cannot be resumed, and the user needs to define new states to resume the execution. After resuming the execution, the live modeling session is terminated. The details of the replay function will be discussed later in this section.

As discussed in Sec. 4.2, functions are added into the base class of the capsule to enable self-reflection. The added functions support the full range of live modeling services, including replaying execution, adding/removing/updating states, transitions, actions, and triggers. Variables can be added, inspected, and updated, but not removed. The implementation of most functions is straight-forward as they are implemented by adding, modifying, or removing entries in the runtime model. In the following, we first discuss how execution replay and the modification of actions are implemented. Then, we explain the other services by example. Finally, we discuss how OP1-3 (see 2.4) can be addressed in the context of running example.

4.3.1 Execution replay (re-execution)

We support real-time state transfer (ref. 2.4), which causes issues when the changes in model include an instruction that has already been executed, because users cannot see the effect of their changes. Suppose, e.g., the action of transition *t23* is changed when the execution of the traffic light is in state

Algorithm 1: Replay execution

```

Input : Source state (fromState), Destination state (toState)
1 Let requiredMsg and replayedState be an empty sequence
2 replay(fromState, toState, requiredMsg, replayedState)
3 Function replay(State: fromState, toState, Sequence: requiredMsg, replayedState )
4   if (fromState=toState) then
5     trace  $\leftarrow$  lastT(replayedState[0]) // get the most recent execution trace of
        replayedState[0]
6     reset(trace.vars) // reset variable values to values extracted from trace
        (trace.vars)
7     Set the current execution state to replayedState[0]
8     defer all messages in the capsule queue
9     inject(requiredMsg); // inject all messages required to replay using
        debugging command
10    return // replay complete
11  else
12    Let trace be the most recent execution trace of fromState
13    if (trace=null or replayedState.contains(fromState)) then
14      // error, there is no path to toState
15    else
16      requiredMsg.append(trace.lastM)
17      replayedState.append(fromState)
18      nextS  $\leftarrow$  lookupStateMap(fromState, trace.lastM) // find the next
        state using the state maps
        replay(nextS, toState, requiredMsg, replayedState);

```

yellow. In that situation, there is no way for users to ever see the effect of the change. To deal with this challenge, we provide a limited type of execution replay that allows execution the USM based only on the most recently processed message at each state.

Algorithm 1 shows how the execution replay is implemented. The algorithm accepts states *fromState* (source state) and *toState* (destination state) as input. It first creates two empty sequences *requiredMsg* and *replayedState* and calls the function *replay*. Function *replay* is a recursive function that tries to find an execution path (a sequence of transitions) from *fromState* to *toState* such that the sequence *requiredMsg* causes that path to be taken (i.e., *requiredMsg* contains exactly the messages needed to trigger the transitions on the path). The code of function *replay* is shown in Algorithm 1. Its execution branches based on whether or not the source *fromState* and *toState* are equal:

- (*fromState* is equal to *toState*). A path has been found. In this case, the variable values are reset using the most recent execution trace of the source state, and all required messages are injected to replay the execution to the destination state. Before injecting the required messages, all existing messages in the capsule queue are deferred to ensure the replaying will not be interrupted. The users can recall these messages after the replaying is complete by adding actions or a debugging command. The message injection is supported by calling a function that has been added into the capsule base

class (see Section 4.2). The function adds the relevant UML-RT message into the queue of the capsule.

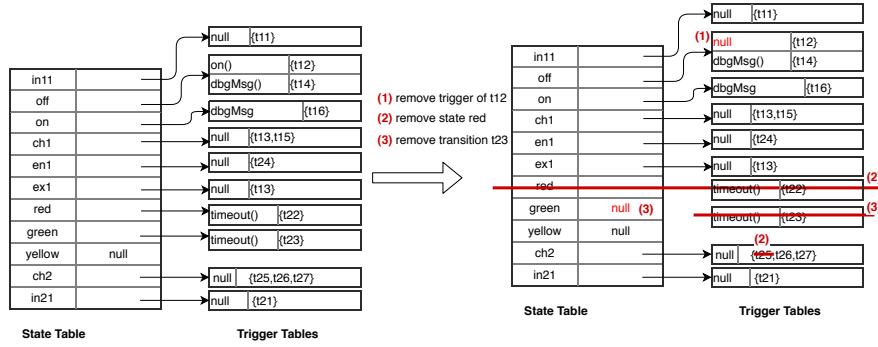
- (*fromState* is not equal to *toState*). A path has not been found yet. In this case, the function raises an error if: (1) *fromState* has not previously been visited by checking its most recent execution trace, since the reexecution of a path cannot be performed without a trace for each state in the path, or (2) there is a loop in the path. This means a path from the source state to the destination state cannot be found based on the most recent execution traces. If no error is raised, the function adds *fromState* and its last processed message (*lastM*) to *replayedState* and *requiredMsg* respectively and calls function *replay* recursively by using *nextS* as the next state in the path.

Note that the offered replay mechanism is mainly useful during debugging, specifically step-wise debugging, where a bug in a specific part of the model needs to be localized and fixed. In this situation, the replay function allows users to rewind (reset) the execution to a specific state and then see the effect of any changes to the part without having to restart the application. But since the replay function only replays the most recent traces, the rewind capability is limited in the number of steps it supports, especially when the state machine contains cycles. Also, the replay mechanism only focuses on replaying the execution of one capsule (target capsule) by injecting the required messages on behalf of the other capsule. During the replay, the target capsule can send/receive messages to/from other capsules that may not be expected. Thus, the capsules may not react as expected. However, this does not cause an issue for the re-execution of the target capsule since we inject all required messages. To deal with this issue, it is possible to extend the replay function to rewind the execution of the relevant capsules as well as the target capsule to allow the execution of the whole system to be replayed without the injection of messages. We left this for future work.

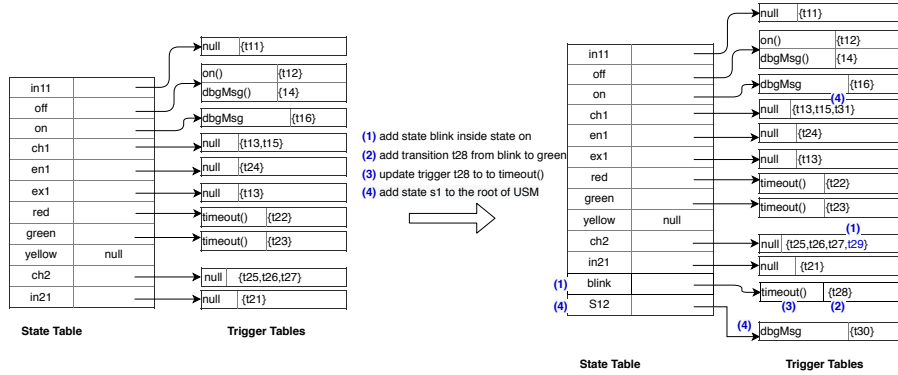
4.3.2 Editing actions

As discussed in Section 2, actions can be provided in the form of the effect of a transition, or as the entry or exit code of a state. Papyrus-RT allows actions to be written in C++. During code generation, a C++ function is generated for each action with a body that contains the action as given in the model (i.e., no translation is performed). Thus, the editing of actions at runtime would require using the live modeling support provided by C++. However, as discussed in Section 2 this support is limited. To overcome this issue, our method to support the editing of actions is two-fold, as discussed in the following.

- *Code generation*. As discussed, each transition and state in the runtime model has function pointers for actions. During the code generation, the function pointers are initialized to refer to the functions that are generated by the default code generation of Papyrus-RT. Thus, at the beginning of



(a) Remove state, transition, and trigger (actions and corresponding removed elements annotated using the same number.)



(b) Add state, transition, and trigger (actions and corresponding added elements annotated using the same number.)

Fig. 10: Examples of live modeling services that are provided by updating the runtime model.

the execution, all the actions are executed as binary code (generated by C++ compiler), and no performance overhead is introduced.

- *Edit at runtime.* To update an action at runtime, we update the relevant function pointer in the runtime model to refer to a proxy function that interprets the updated action. This means that as soon as an action is updated, it is no longer executed as binary code until the end of the execution. Note that an action is only interpreted when it is edited, and each of the actions is treated separately (editing of one action does not affect other actions). Also, in our current implementation, the interpretation support is limited to arithmetic expression, sending messages, and logical expressions. We left the extension of the interpreter as future work.

4.3.3 Other live modeling services

Figure 10 shows some examples of how adding/removing/updating states, transitions, and triggers is supported by updating the runtime model, in the context of the stoplight example. Note that by adding/removing states, the instrumentation is updated. By adding state *blink* inside state *on*, a transition from *blink* to *ch2* is added. Similarly, by adding state *S12*, transitions are added from/to *S12* to/from *ch1*. As discussed in 4.1, the added transitions allow the start of a live modeling session upon injection of the *dbgMsg* and the steering of the execution to an appropriate state, when the live modeling session is finished.

4.4 Live Update Operations on the Running Example

We review how the five operations in Section 2.1 can be performed based on the provided services:

- **OP1. (add actions to transition t23)** The action of transition *t23* can be modified. If the execution is past transition *t23*, the execution can be replayed from state *green* to state *yellow*. In state *green*, *REPL* commands can be used to inspect the effect of the modified action.
- **OP2. (add a transition along with actions and a trigger from state yellow to state red)** OP2 can be addressed by using add transition and add action services. Note that the trigger can be specified when adding transition.
- **OP3. (remove state yellow when it is the active execution state.)** The user can remove state by activation of the live modeling session, when the execution is in state *yellow*. During the resuming of the execution, the user is asked to select the next execution state, which can be any of the other basic states. Then the execution can be steered to the selected state possibility through choice-point states that are added during instrumentation.

4.5 Tool Support (Live-UMLRT)

This section presents *Live-UMLRT* [8], a tool that supports live modeling of UML-RT models and embodies our approach.

4.5.1 Implementation

To develop *Live-UMLRT*, we used Papyrus-RT as the primary tool to model RTE systems, the Papyrus-RT code generation extension to generate self-reflective code, and the Epsilon Object Language (EOL) [52] to implement the transformation rules required for refining the models into instrumented models. The *LMP* is implemented using C++, ANTLR [64], and the Boost

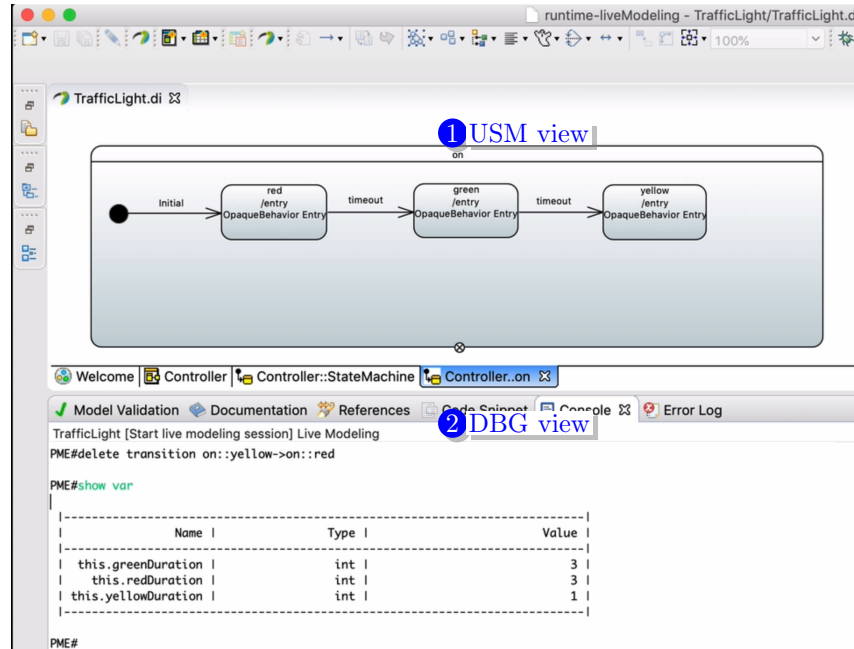


Fig. 11: User interface of Live-UMLRT.

C++ Library [10]. *REPL* commands are implemented using debugging features that are ported from MDebugger [7].

4.5.2 Live-UMLRT Features

In the following, we discuss the features of *Live-UMLRT*¹ from the user point of view. When it is possible, the use of features is explained using the running example.

Setup and run: The Live-UMLRT is integrated into Papyrus-RT as an Eclipse plugin and can be downloaded and installed from the *Live-UMLRT* repository². After installation, it can be used to edit UML-RT models at runtime simply by defining a run configuration (i.e., Eclipse run configuration) inside Papyrus-RT. When a model is executed using the defined configuration, the model instrumentation, code generation, and build are executed automatically in the background without distracting end user. The UI of Live-UMLRT is shown in Figure 11 which is split in two parts, a *USM view* (①) and a *DBG console* (②). In the USM view, the user can view and edit the USM of the capsules. In the *DBG console* the user can issue *REPL* commands to investigate and change the model at runtime. Basic *REPL* commands (e.g.,

¹A demonstration video: <https://www.youtube.com/watch?v=6GrR-Y9je7Y>

²<https://moji1@bitbucket.org/moji1/live-umlrt.git>

view and change variables) are available. Next, we discuss the steps for live modeling along with several features of *Live-UMLRT*.

Starting live modeling session and applying changes: To start live modeling, a live modeling session must be started that activates *LPM*. Users can start a live modeling session by pressing the ‘b’ key in the *DBG* console. This will stop the execution (similar to a debugging breakpoint) and allow users to apply changes into the model execution. This scenario is similar to the way that popular IDEs such as Eclipse support live programming work.

Applying changes to the model execution: During the live modeling session the user has two ways to update the model execution: **(1) Changing the design model:** the user can use *USM view* to update the model and save. During the save *Live-UMLRT* serializes the last change as update commands and sends them to the *LMP*. As discussed, the *LMP* applies changes by updating the runtime model. **(2) Changing the runtime model:** the user can issue the supported commands via the *DBG* console that validates and forwards the commands to the *LMP*. With the current implementation, the changes from the runtime model are not propagated back to the design model. Thus, the changes affect the execution until the end of the live modeling session. Implementation of this part is left to future work.

Supported edit operations: Listing 2 shows the most important features supported by *Live-UMLRT*. In addition to the UI of *Live-UMLRT*, the features can be used via a TCP connection with the *LMP*. In the following, we briefly discuss these features. (1) The add/delete/update commands are used to add/delete/update states and transitions. (2) A variable can be defined simply by initializing it. (3) To define a new action code, a record command should be used after which the UML-RT action code interpreter is activated. It accepts and interprets the action code line by line. Upon successful interpretation, the code can be saved to the runtime model as well as the design model by a translation into C++ syntax. Since there is no mismatch between the supported commands and the C++ action code that is supported by Papyrus-RT, the translation is simple and addressed by the USM view. (4) Command *replay* allows the user to re-execute the previous execution steps to see the effect of the new changes. E.g., let us assume that the user completes the USM shown in part ❶ by adding a transition from state *yellow* to state *red* when the execution is stuck in state *yellow* and unable to handle the received messages. Without a replay mechanism, there is no way to see the effect of the new change and the execution will be stuck in state *yellow* forever. However, by issuing *replay* from *green* to *yellow*, *Live-UMLRT* steers the execution from state *green* to state *yellow* again and injects the last messages that received in states *green* and *yellow*. By that, the execution can advance by processing the injected messages, and the user can see the effect of the new changes.

Listing 2: Supported Edit Operations

```

Add state: add state <name>
Delete/Update state: delete/update state <name>
Add transition: add transition (<name>)? <from>-><to> (when <signal> on
    <port>)?
Delete transition: delete transition (<name>)? <from>-><to>
Update transition: update transition (<name>)? <from>-><to> (when <signal> on
    <port>)?
Add action: record (action code)* save
Delete action: delete action (state|transition) (entry|exit)?
Add/Update variable: <name>=<expressions>
Replay execution: replay from <state name> to <state name>

```

5 Experimental Evaluation

This section details experiments we conducted to assess the performance, benefits, and overhead of our approach. For that, we consider the following four research questions:

RQ1 (Performance of instrumentation and code generation). What is the performance of our approach to create self-reflective programs?

As discussed earlier, our approach for the creation of the self-reflective program consists of three steps: instrumentation of models, code generation, and compile/build. Creation of the self-reflective program is a core part of our approach. This research question examines the performance of the first two steps. Note that the compile/build is performed using code-based tools and their performance is out of our control.

RQ2 (Edit delay). How is the response time of edit operations in our approach compared with the approach relying on the services provided by the programming language of the generated code (live programming)?

As discussed earlier, our approach provides live modeling via interaction with the self-reflective program and does not require code generation, compile/build, or hot-swapping for each edit. This research question examines the efficiency of our approach and compares it with the approach relying on live programming.

RQ3 (Performance overhead). What is the performance of the self-reflective program compared with that of the program created using the default code generation?

As discussed earlier, our approach generates the code that explicitly embeds the AST of the model. For that, we change the code generation which may affect the performance of the created program. Thus, this research question examines the performance overhead of self-reflective program and compares it with that of the program created using the default code generation.

RQ4 (Memory overhead). What is the memory footprint of the self-reflective program compared with that of the program created using the default code generation?

Our approach instruments the models by adding new elements and creates a

Table 2: Model Complexity of Use-cases, Median of Code Generation and Instrumenting Time

Model	Model Complexity			Code Gen. (ms)		Inst. (ms)
	C	S	T	Def.	Ours	
Car Door Lock	5	11	15	1063	1056	1005
Digital Watch	9	47	57	885	920	2564
Parcel Router	8	14	25	1140	1098	1277
Rover	6	16	21	1109	1187	1662
FailOver	7	31	43	1197	1274	2003
Debuggable FailOver	8	350	620	43623	46692	6200

C: Component, *S*: State, *T*: Transition, *Def.*: Default
Inst.: Instrumentation Time, *Code Gen.*: Code Generation Time

runtime model during the execution. These two together introduce memory overhead. This research question examines the amount of memory overhead of our self-reflective program and compares it with that of the program generated by the default code generation.

5.1 Use cases

To perform experiments, several use cases are used. As shown in Table 2, models have different sizes ranging from simple models containing 11 states to models with 350 states. Simple models include the Car Door Central Lock system and the Digital Watch. The Car Door Central Lock system is a control system for locking and unlocking car doors. The Digital Watch is an implementation of a classical digital watch described in [40].

The Parcel Router [54, 78] is an automatic system where tagged parcels are routed through successive chutes and switches to a destination bin. The system is time-sensitive and jams can appear due to variations in the time required by a parcel to transit through the different chutes. The system checks for potential parcel jams, and prevents parcels from being transferred from one chute to another, until the next chute is empty. The simplified version ignores jams.

The Rover system model [1] allows an autonomous robot to move in different directions. It is equipped with three wheels, driven by two engines. It can move forward, move backward, and rotate. Additionally, it is equipped with several sensors, such as temperature and humidity sensors to collect data from the environment, and an ultrasonic detection sensor to detect and avoid obstacles.

The FailOver system [9, 50] 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 [37]. In passive replication, one server component works as the master, handling all

the client requests while backup servers are mainly 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.

The Debuggable FailOver system is a debuggable version of the FailOver system, which is generated using MDebugger [6]. The complexity of this model is high, and allows us to check that the refinement and analysis time do not skyrocket when the model size grows exponentially.

5.2 Experiments

In the following, we discuss the experiments used to calculate the metrics.

Measuring the performance of model instrumentation and code generation (EXP-1). To answer RQ1, we ran our code generation, the model instrumentation, and the default code generation (existing code generation of Papyrus-RT) 20 times against the use cases listed in Table 2 and in each case recorded the time required. We also saved the generated code and instrumented versions of the models that are used in *EXP-2*.

Measuring the performance of edit operations (EXP-2). To answer RQ2, first, we executed the code generated by our approach, i.e., the result of *EXP-1*, for the Debuggable Failover model and tried each of the supported edit operations (ref. Table 3) 20 times using a random element and recorded the required times for each operation on a single element. Then we tried each edit operation on ten elements distributed over three different components and recorded the time needed for each operation on these ten elements. Second, we repeated the same experiments on the code generated by an implementation of live modeling that relies on the services provided by the programming language of the generated code (live programming) (Section 2.5). Note that Debuggable Failover is the largest use case in our experiment and the results of the experiment can be generalized safely to the other use cases.

Measuring the performance and memory overhead of our approach (EXP-3). To answer RQ3 and RQ4, similar to *EXP-2* we executed the generated code of Failover in the context of our approach. During the execution we configured the system to process 10,000 client requests and recorded the CPU time and memory usage for processing the requests. Second, we repeated the same experiment using the code generated from the Failover model by the default code generation.

Also, since our approach interprets the actions as soon as they are modified during the live modeling, we executed action code with 100 lines of code in interpreted mode vs compiled mode and recorded the CPU time in each case to measure the overhead of the interpretation of actions.

5.2.1 Set up and Reproducibility of Experiments

We used a computer with OS MacOS 10.12 (Sierra) and OpenJDK 8 with 4GB maximum heap, equipped with a 2.2 GHz Intel Core i7 processor (4770HQ), and 8GB of memory, for all experiments, and this is a typical development PC. The experiments are automated using Bash scripts. The scripts and models are publicly available at [4] and can be used to repeat our experiments. Note that we intentionally used a standard computer comparable to those used by developers, rather than more powerful hardware, because the debugging and live modeling of models typically needs to be carried out daily.

5.3 Results and Discussions

In the following, we present the results of our experiments and discuss their impact.

5.3.1 RQ1 (Performance of the creation of the self-reflective program).

Based on the result of *EXP-1*, the *Code generation Time* and *Instrumentation Time* columns of Table 2 show the median of time required for instrumentation and code generation by our approach, and the default code generation. For the largest model (Debuggable Failover), the median time of the instrumentation and code generation are less than 47 and 7 seconds respectively. Code generation with the default code generator took 44 second which is slightly faster (3 seconds) than our approach. It is, therefore, safe to conclude that the performance of code generation and instrumentation time of our approach are reasonable. Note that code generation and instrumentation are required only once for program generation and the required time appears negligible w.r.t. the benefits provided by the self-reflective program.

5.3.2 RQ2 (Edit delay)

Based on the results of *EXP-2*, Table 3 shows the median of the time required for edit operations using our approach and the approach relying on live programming. For a single operation on a single element and ten elements in three components, on average our approach is 400 times and 92 times, respectively, faster than when live programming services are used. As discussed, the main reason for this difference is the need for regeneration and recompilation after each change.

We can conclude that our approach significantly improves the performance of edit operations (any change is applied in less than two milliseconds) which is considered quite acceptable in the context of live updates (e.g., according to [55, 56], users start noticing latency at 100ms and become distracted at 500ms).

Table 3: Performance of Edit Operation using Our Approach and Live Program

Operation	One Edit (ms)			10 Edits, 3 Comps. (ms)		
	Ours	Prog.	Ratio	Ours	Prog.	Ratio
Add State	1.3	608	467	10.6	1192	112
Rem./update State	1.5	608	405	11.5	1192	103
Add Trans.	1.5	608	405	12.9	1192	92
Rem./update Trans.	1.8	608	377	16.1	1192	74
Add Var	1.3	608	467	9.9	1192	120
Add Action	2.1	608	289	18.1	1192	66
Rem./update Action	1.5	608	405	12.3	1192	96
Average	1.6	608	405	13	1192	92

Comps.: Components, *Ours*: Our approach, *Prog.*: Live Programming

Trans.: Transition, *Ratio*.: Prog./Ours, *Rem.*: Remove

5.3.3 RQ3 (Performance overhead)

Based on the result of *EXP-3*, the code generated from the Failover model using our approach took 510ms of CPU time to process 10,000 requests. This is only 1% slower than the time required by the code generated with the default code generator (514ms). Thus it is safe to conclude that the change in the generated code to support live modeling causes negligible performance overhead w.r.t. to the provided services.

Also, our experiment of the execution of actions in interpreted and compiled mode shows that, not surprisingly, the interpretation of actions with 100 lines of code is 70% slower than the execution of their compiled versions. Note that an action is only interpreted when it is edited during the live modeling. Also, our interpreter is a prototype, not built with performance optimization in mind, whereas C++ compilers are highly sophisticated and optimized.

5.3.4 RQ3 (Memory overhead)

Based on the results of *EXP-3*, the peak memory usage of code generated from the Failover model using our approach is 2083 KB to process 10,000 requests. This is 25% more than the memory usage by the code generated with the default code generator (1664 KB). We can argue this memory overhead is acceptable for many applications.

5.4 Summary of the results

According to the results mentioned above (i.e., acceptable performance of code generation and instrumentation, significant improvement of the edit operations, negligible performance overhead, and slower execution of edited actions),

the offered edit operations and debugging services, safe update, and the replay mechanism, we conclude that our approach is practical for the live modeling in many application domains. However, since the memory overhead of our current implementation is 25%, and the execution of the edited actions code is slower, the use of our approach in the context of memory-constrained and time-sensitive systems would require extra work and optimization.

6 Related work

This section presents the most relevant work in three categories: model-level debugging, live modeling, and live programming.

6.1 Model-Level Debugging

As discussed in Sec. 2, live modeling services complements model debuggers, to support model edition at the time the running model is in pause. In general, models can be executed either by interpretation or by the translation of models into existing programming languages, often by code generation (translational execution) [18]. The translational execution allows the execution of the model on the target platform.

Based on the execution mechanism, we classify the related work for model-level debugging into two groups: debugging via model interpretation, and model-level debugging on a target platform (i.e., model-level debugging in the context of code generation). In the following, we discuss each of these categories in detail.

6.1.1 Model-level Debugging via Interpretation

Debugging via interpretation has been conducted by interpreting the models at design-time, where debugging features, such as setting breakpoints and stepping over the execution, are usually supported. Model interpretation is supported by several tools, e.g., Matlab StateFlow [84], AF3 [31], xtUML [91], and YAKINDU [47].

Van Mierlo et al [60] address debugging of PythonPDEVS [89], which is a modeling tool based on Parallel Discrete Event Simulator (PDES) . PDES is particularly concerned with the simulation of asynchronous systems, where events occur at irregular time intervals [33]. To support the debugging, they model the modal part of the simulator using a statechart, and include the debugging logic in the statechart. This provides the debugging of models through the debugging of the simulator. In his thesis [87], Van Mierlo extends and generalizes the work, and presents an architecture to help language engineers to create visual debugging environments for their language interpreter. The proposed architecture consists of three components: an instrumented simulator (i.e., interpreter) that provides debugging services, a debugging interface that

allows users to communicate with the instrumented simulator, and a model-specific user interface that visualizes the execution state of the simulation, compatible with the semantics of the language. The feasibility of the approach is evaluated by creating debuggers for several modeling languages, e.g., Causal Block Diagram (CBD) [70]. Also, Mustafiz et. al [62] uses a similar approach to ours for instrumenting Statecharts models with debugging support, by instrumenting the original model.

A particular case of debugging is omniscient debugging (back-in-time or reversible debuggers) proposed by several studies in the context of model interpretation. Omniscient debugging replays the execution of systems using the recorded traces, and provides step-by-step execution in both forward and backward modes, and variable view (e.g., [13, 21]). Corley et al. [20, 22] explore this approach to the debugging of model transformations in AToMPM [79]. Their implementation records each change at the end of a transformation step, and provides support to step back to the previous states.

Also, some studies try to automate the tracing and omniscient debugging of models when the execution is based on interpretation. E.g., Bousse et al. [11, 14] propose an approach to automatically generate a multidimensional (i.e., metamodels that provide many navigation paths to explore a trace) and domain-specific trace metamodel (trace structure), trace constructor, and trace analyzer facilities for a given modeling language. Their approach works based on the operational semantics defined as an execution metamodel and transformation. They specify details of the execution based on the execution metamodel using GEMOC studio [12].

6.1.2 Model-level Debugging on the Target Platform

Tracing and replay: in debugging by tracing, the model or the generated code is instrumented to generate useful execution traces. Then, the traces are collected and used for offline analysis and debugging. Hojaji et al. [44] surveys the existing work in the context of model execution tracing. Examples of existing work and MDD tools supporting trace analyses via code instrumentation include [34, 38, 43, 48, 49, 83]. Iyengar et al. [48, 49] introduce an optimized model-based debugging technique for RTE systems with limited memory. They use a monitor on the target platform to collect the generated traces and a debugger executed on a host with sufficient memory to analyze the traces offline, and to display results on the model elements. Das et al. [24] propose a configurable tracing tool based on LTTng. They rely on code instrumentation in order to produce useful tracepoints for LTTng. The tool supports timing constraint analysis via trace replay. It can be performed offline or live, using a remote connection to the target platform. Trace replay is directly represented via animation on the model elements.

In addition to the overhead of tracing, the connection between target platform and debugger is only one-way, and does not provide the required controls for rich debugging features, such as stepping over the execution, setting breakpoints, or changing attributes. To the best of our knowledge, only the proposed

work in [34] provides limited support for controlling the execution via signal injection. However, the proposed approach requires the maintenance of a mapping between the source code and the model elements, which can be addressed by instrumentation [24], or by being stored in mapping files [48, 49].

Live debugging on the target platforms: Live debugging on target platforms is the richest debugging service to debug the model execution. Despite its importance, only a few MDD tools, e.g., ProgramDev [71], IBM RSARTE [45], and Timing Architects [86] support live debugging capabilities. Further, some studies try to address model-level debugging using traditional approaches [3, 26, 36, 67, 77, 92]. Martin et al. [3, 67, 77] develop an integrated debugging plug-in for equation-based models created by Modelica [61]. They use GDB [73] to debug the generated C code, and then map the debugging results to the equation-based model element. Dotan et al. [26] develop a model-level debugger for IBM RSARTE. Similarly, Graf et al. [36] propose an extension for UML state machines that facilitates the construction of mappings from code to model-level and vice versa, and implement a debugger for model-level debugging of Stateflow. Similarly, Kebianyor et al., extend LLDB debugger [82] with model-to-code mapping data, and create a model-level debugger for Stateflow [84] which support debugging models when the code is generated in C and C++.

6.2 Live Modelling

Live modeling is not well addressed in the context of MDD and only some efforts have been made towards live modeling. Van Tendeloo et al. [90] propose a multi-paradigm approach to support live modeling of modeling languages generically. Their proposal for addressing live modeling with translational execution is relying on the service offered by the target language. The problems with this approach are discussed in Sec. 2. Our approach does not rely on the services offered by the target language. Ulyana et al. [85] propose a solution for state transfer in the context of model interpretation. They define the state transfer invariants and constraints using a language specifically developed for this purpose. They employ model finding techniques based on a Satisfiability Modulo Theories (SMT) solver to automatically find a new runtime model that satisfies the declared constraints. On the contrary, our approach is in the context of model execution by code generation and prevents changes that cause inconsistency and allows users to do state transfer by replaying the execution. Rozen et al. [88] propose an approach for live modeling in the context of textual domain-specific languages (DSLs) and interpretation which works by calculating differences between versions of the DSL program in terms of the metamodel of the language and applying the change at runtime.

6.3 Live Programming

We already have discussed existing challenges and approaches in live programming in Sec. 2. Also, some work has studied how presentation (graphical versus textual) and visualization impact the usability of live programming [29, 66]. Examples of graphical formalisms are VIVA [81] and Flogo [39] and textual formalisms are ElmScript [23] and Smalltalk [35]. Some usability work has focused on optimization of edit latency such as incremental compilation [56] and safe and efficient hot-swapping (e.g., [15, 17, 30]).

7 Conclusion and Future Work

In this work, we have proposed an approach for supporting live modeling in the context of the state machine models when they are executed by code generation. Our approach relies on model transformation and code generation rather than using any services or capabilities offered by the programming language of the code being generated. We have illustrated and validated our approach through the implementation of live modeling of UML-RT models in the Papyrus-RT tool. Our prototype supports live modeling of UML-RT together with safe update and execution replay for state transfer. Our empirical analysis shows that our implementation (1) reduces the edit latency significantly, (2) is applicable with reasonable performance, (3) introduces negligible performance overhead and, (2) has an acceptable memory overhead for many application domains.

Future and ongoing work to extend/improve our approach includes:

- Removing or mitigating the limitations of the current prototype. As discussed, our current implementation has three main limitations: (1) The execution replay only reexecutes the execution of the target capsule based on the most recent execution traces, (2) only interpretation of arithmetic and logical expressions, plus actions for sending messages, is supported, and (3) the interpretation of actions is relatively slow. While extending the interpretation to support additional actions only requires development efforts, the efficient interpretation of actions and advance execution replay can be addressed in future research work.
- Efficient tracing mechanism. The current approach provides the replay function only based on the most recent execution traces, because handling a large number of execution traces is computationally intensive, and can slow down the live modeling services. We have been working on efficient trace collection and generation mechanisms that can be used to extend the replay function.

Moreover, the extension of our approach to currently unsupported edit operations can be addressed as future work. Finally, our focus was live modeling, but the proposed approach and implementation (which are publicly available)

would also be of value for work on, e.g., model-based development of self-adaptive systems, models at runtime, and execution of partial state machine models.

References

1. Reza Ahmadi, Nicolas Hili, Leo Jweda, Nondini Das, Suchita Ganesan, and Juergen Dingel. Run-time monitoring of a rover: MDE research with open source software and low-cost hardware. In *EduSymp/OSS4MDE@MoDELS*, pages 37–44, 2016.
2. Jonathan Appavoo, Kevin Hui, Craig AN Soules, Robert W Wisniewski, Dilma M Da Silva, Orran Krieger, Marc A Auslander, DJ Edelsohn, Benjamin Gamsa, Gregory R Ganger, et al. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1):60–76, 2003.
3. Adeel Asghar, Adrian Pop, Martin Sjölund, and Peter Fritzson. Efficient Debugging of Large Algorithmic Modelica Applications. *IFAC Proceedings Volumes*, 45(2):1087–1090, 2012.
4. Mojtaba Bagherzadeh, Karim Jahed, Benoit Combemale, and Juergen Dingel. Live-UMLRT Repository. <https://bitbucket.org/moji1/live-umlrt.git>. Retrieved March 19, 2020.
5. Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. MDebugger repository. <https://github.com/moji1/MDebugger>, 2017. Retrieved June 5, 2017.
6. Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. In *11th Joint Meeting on Foundations of Software Engineering*, pages 419–430, 2017.
7. Mojtaba Bagherzadeh, Nicolas Hili, David Seekatz, and Juergen Dingel. MDebugger: A model-level debugger for uml-rt. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE ’18, pages 97–100, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5663-3. doi: 10.1145/3183440.3183473. URL <http://doi.acm.org/10.1145/3183440.3183473>.
8. Mojtaba Bagherzadeh, Karim Jahed, Benoit Combemale, and Juergen Dingel. Live-UMLRT: A tool for live modeling of uml-rt models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 743–747. IEEE, 2019.
9. Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, Aniruddha Gokhale, Christopher Gill, and Douglas C Schmidt. Adaptive Failover for Real-Time Middleware with Passive Replication. In *15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 118–127. IEEE, 2009.

10. David Abrahams Beman Dawes. Boost C++ Library. <https://www.boost.org/>, 2019. Retrieved March 19, 2019.
11. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. A generative approach to define rich domain-specific trace metamodels. In *European Conference on Modelling Foundations and Applications*, pages 45–61. Springer, 2015.
12. Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 84–89. ACM, 2016.
13. Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for executable DSLs. *Journal of Systems and Software*, 137:261–288, 2018.
14. Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software & Systems Modeling*, 18(1):385–421, Feb 2019. ISSN 1619-1374. doi: 10.1007/s10270-017-0598-5. URL <https://doi.org/10.1007/s10270-017-0598-5>.
15. Sebastian Burckhardt, Manuel Fähndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It’s alive! Continuous feedback in UI programming. In *ACM SIGPLAN Notices*, volume 48, pages 95–104. ACM, 2013.
16. Miguel Campusano, Johan Fabry, and Alexandre Bergel. Live programming in practice: A controlled experiment on state machines for robotic behaviors. *Information and Software Technology*, 108:99–114, 2019.
17. Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE ’07, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.65. URL <https://doi.org/10.1109/ICSE.2007.65>.
18. Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling*, Apr 2018. ISSN 1619-1374. doi: 10.1007/s10270-018-0675-4. URL <https://doi.org/10.1007/s10270-018-0675-4>.
19. Atkinson Colin and Kühne Thomas. Model-Driven Development: a Meta-modeling Foundation. *IEEE Software*, 20(5):36–41, 2003.
20. Jonathan Corley. *Exploring efficient and scalable omniscient debugging for MDE*. PhD thesis, 2016.
21. Jonathan Corley, Brian P Eddy, and Jeff Gray. Towards efficient and scalable omniscient debugging for model transformations. In *Proceedings of the 14th workshop on domain-specific modeling*, pages 13–18. ACM, 2014.
22. Jonathan Corley, Brian P. Eddy, and Jeff Gray. Towards efficient and scalable omniscient debugging for model transformations. In *Proceedings of the 14th Workshop on Domain-Specific Modeling*, pages 13–18, 2014.

23. Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.
24. Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 36–43. ACM, 2016.
25. L.A.J. Dohmen and L.J. Somers. Experiences and lessons learned using UML-RT to develop embedded printer software. In Markku Oivo and Seija Komi-Sirviö, editors, *Product Focused Software Process Improvement*, pages 475–484, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-36209-8.
26. Dolev Dotan and Andrei Kirshin. Debugging and Testing Behavioral UML Models. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, pages 838–839. ACM, 2007.
27. Eclipse Foundation. Eclipse Papyrus for Real Time (Papyrus-RT). <https://www.eclipse.org/papyrus-rt>, 2019. Retrieved March 19, 2019.
28. Eclipse Foundation. Eclipse eTrice - Real-Time Modeling Tools. <https://www.eclipse.org/etrice/>, 2019. Retrieved: 2019-02-10.
29. Jonathan Edwards. Subtext: Uncovering the simplicity of programming. *SIGPLAN Not.*, 40(10):505–518, October 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094851. URL <http://doi.acm.org/10.1145/1103845.1094851>.
30. R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. URL <http://dl.acm.org/citation.cfm?id=800253.807720>.
31. FORTISS GMBH. AF3. <http://af3.fortiss.org/>, 2016. Retrieved July 19, 2016.
32. Michael Franz. Dynamic linking of software components. *Computer*, 30(3):74–81, 1997.
33. Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.
34. Eran Gery, David Harel, and Eldad Palachi. Rhapsody: A Complete Life-Cycle Model-Based Development System. In *International Conference on Integrated Formal Methods*, pages 1–10. Springer, 2002.
35. Adele Goldberg. Smalltalk-80-the interactive programming environment. 1984.
36. Philipp Graf and Klaus D Muller-Glaser. Dynamic Mapping of Runtime Information Models for Debugging Embedded Software. In *Seventeenth IEEE International Workshop on Rapid System Prototyping, 2006*, pages 3–9. IEEE, 2006.

37. Rachid Guerraoui and André Schiper. Software-Based Replication for Fault Tolerance. *Computer*, 30(4):68–74, 1997.
38. Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten. Model-Level Debugging of Embedded Real-Time Systems. In *10th International Conference on Computer and Information Technology (CIT'10)*, pages 1887–1894. IEEE, 2010.
39. Christopher Michael Hancock. *Real-time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Cambridge, MA, USA, 2003. AAI0805688.
40. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
41. HCL. HCL RealTime Software Tooling (RTist). <https://www.hcltech.com/products-and-platforms/rtist>, 2019. Retrieved March 19, 2019.
42. Michael Hicks, Jonathan T Moore, and Scott Nettles. *Dynamic software updating*, volume 36. ACM, 2001.
43. Nicolas Hili, Mojtaba Bagherzadeh, Karim Jahed, and Juergen Dingel. A model-based architecture for interactive run-time monitoring. *Software and Systems Modeling*, pages 1–23, 2020.
44. Fazilat Hojaji, Tanja Mayerhofer, Bahman Zamani, Abdelwahab Hamoulhadj, and Erwan Bousse. Model execution tracing: a systematic mapping study. *Software & Systems Modeling*, Feb 2019.
45. IBM. IBM Rational Software Architect RealTime (RSARTE) edition, v9.5.0 product documentation. http://www.ibm.com/support/knowledgecenter/SS5JSH_9.5.0. Retrieved March 19, 2020.
46. IBM. IBM RSARTE, 2016. URL <https://www.ibm.com/developerworks/downloads/r/architect/index.html>. Retrieved July 19, 2016.
47. ITEMIS AG. Yakindu StateChart Tools. <https://www.itemis.com/en/yakindu/statechart-tools>, 2016. Retrieved July 19, 2016.
48. Padma Iyengar, Clemens Westerkamp, Juergen Wuebbelmann, and Elke Pulvermueller. A Model Based Approach for Debugging Embedded Systems in Real-Time. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, pages 69–78, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-904-6. doi: 10.1145/1879021.1879031. URL <http://doi.acm.org/10.1145/1879021.1879031>.
49. Padma Iyengar, Elke Pulvermueller, Clemens Westerkamp, Michael Uelschen, and Juergen Wuebbelmann. Model-Based Debugging of Embedded Software Systems. *Gesellschaft Informatik (GI)-Softwaretechnik (SWT)*, 2011.
50. Nafiseh Kahani, Nicolas Hili, James R Cordy, and Juergen Dingel. Evaluation of UML-RT and Papyrus-RT for modelling self-adaptive systems. In *Proceedings of the 9th International Workshop on Modelling in Software Engineering*, pages 12–18. IEEE Press, 2017.
51. Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, Mar 2018. ISSN 1619-

1374. doi: 10.1007/s10270-018-0665-6. URL <https://doi.org/10.1007/s10270-018-0665-6>.
52. Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The Epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.
53. Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: insights from the practice. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1090–1101. IEEE, 2018.
54. Jeff Magee and Jeff Kramer. *State Models and Java Programs*. Wiley, 1999.
55. Sean McDirmid. Living it up with a live programming language. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297073. URL <http://doi.acm.org/10.1145/1297027.1297073>.
56. Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62. ACM, 2013.
57. Sean McDirmid. The promise of live programming. In *Proceedings of the 2nd International Workshop on Live Programming, LIVE*, volume 16, 2016.
58. Microsoft. C++ Edit and Continue in Visual Studio 2015. <https://devblogs.microsoft.com/cppblog/c-edit-and-continue-in-visual-studio-2015/>, 2019. Retrieved March 19, 2019.
59. Microsoft. How to Use Edit and Continue (C#). <https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-edit-and-continue-csharp?view=vs-2019>, 2019. Retrieved March 19, 2019.
60. Simon Van Mierlo, Yentl Van Tendeloo, and Hans Vangheluwe. Debugging parallel DEVS. *SIMULATION*, 93(4):285–306, 2017.
61. Modelica Association. The Modelica Language Specification Version 3.2 revision 2. <https://www.modelica.org/documents/ModelicaSpec32Revision2.pdf>, 2013. Retrieved August 5, 2016.
62. Sadaf Mustafiz and Hans Vangheluwe. Explicit modelling of statechart simulation environments. In *Summer Simulation Multiconference*, page 445 – 452. Society for Computer Simulation International (SCS), 2013.
63. Object Management Group. About the Action Language for foundational UML Specification Version 1.1. <https://www.omg.org/spec/ALF/About-ALF/>, 2018. Retrieved August 5, 2016.
64. Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
65. Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29,

- 2007.
66. Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995. ISSN 0001-0782. doi: 10.1145/203241.203251. URL <http://doi.acm.org/10.1145/203241.203251>.
 67. Adrian Pop, Martin Sjölund, Adeel Asghar, Peter Fritzson, and Francesco Casella. Static and Dynamic Debugging of Modelica Models. In *Proceedings of the 9th International MODELICA Conference; Munich; Germany*, number 076, pages 443–454. Linköping University Electronic Press, 2012.
 68. Ernesto Posse. Papyrusrt: modelling and code generation. In *Workshop on Open Source for Model Driven Engineering (OSS4MDE'15)*, 2015.
 69. Ernesto Posse and Juergen Dingel. An Executable Formal Semantics for UML-RT. *Software & Systems Modeling*, 15(1):179–217, February 2016. ISSN 1619-1366. doi: 10.1007/s10270-014-0399-z. URL <http://dx.doi.org/10.1007/s10270-014-0399-z>.
 70. Ernesto Posse, Juan De Lara, Hans Vangheluwe, et al. Processing causal block diagrams with graphgrammars in atom3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*, pages 23–34, 2002.
 71. PRAGMADEV SARL. ProgmaDev. <http://www.pragmadev.com>, 2016. Retrieved July 19, 2016.
 72. Miller Puckette. Combining event and signal processing in the max graphical programming environment. *Computer music journal*, 15(3):68–77, 1991.
 73. Stan Shebs Richard Stallman, Roland Pesch. Debugging with GDB. <http://sourceware.org/gdb/current/onlinedocs/gdb.pdf.gz>, 2016. Retrieved August 5, 2016.
 74. Christopher Schuster and Cormac Flanagan. Live programming for event-based languages. In *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, volume 15, 2015.
 75. Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.
 76. Bran Selic, Garth Gullekson, and Paul T Ward. *Real-time object-oriented modeling*, volume 2. John Wiley & Sons New York, 1994.
 77. Martin Sjölund, Francesco Casella, Adrian Pop, Adeel Asghar, Peter Fritzson, Willi Braun, Lennart Ochel, and Bernhard Bachmann. Integrated Debugging of Equation-Based Models. In *Proceedings of the 10th International Modelica Conference; Lund; Sweden*, number 096, pages 195–204. Linköping University Electronic Press, 2014.
 78. William Swartout and Robert Balzer. On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM*, 25(7):438–440, 1982.
 79. Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A web-based modeling environment. In *Demos/Posters/Student Research MoDELS*, pages 21–25,

- 2013.
80. Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. Interleaving of modification and use in data-driven tool development. In *Proceedings of the 2014 ACM international symposium on new ideas, new paradigms, and reflections on programming & software*, pages 185–200, 2014.
81. Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127 – 139, 1990. ISSN 1045-926X. doi: [https://doi.org/10.1016/S1045-926X\(05\)80012-6](https://doi.org/10.1016/S1045-926X(05)80012-6). URL <http://www.sciencedirect.com/science/article/pii/S1045926X05800126>.
82. The LLDB Team. The LLDB debugger. <https://lldb.llvm.org/>. Retrieved March 19, 2020.
83. Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 1–8. IEEE, 2003.
84. The MathWorks, Inc. Stateflow - MATLAB & Simulink. <http://www.mathworks.com/products/stateflow/>, 2016. Retrieved July 19, 2016.
85. Ulyana Tikhonova, Jouke Stoel, Tijs Van Der Storm, and Thomas Degueule. Constraint-based run-time state migration for live modeling. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, pages 108–120. ACM, 2018.
86. Timing-Architects Embedded Systems GmbH. Timing Architects. <http://www.timing-architects.com/>, 2016. Retrieved July 19, 2016.
87. Simon Van Mierlo. *A multi-paradigm modelling approach for engineering model debugging environments*. PhD thesis, University of Antwerp, 2018.
88. Riemer van Rozen and Tijs van der Storm. Toward live domain-specific languages. *Software & Systems Modeling*, 18(1):195–212, Feb 2019.
89. Yentl Van Tendeloo and Hans Vangheluwe. PythonPDEVs: A distributed parallel DEVs simulator. In *Proceedings of the Symposium on Theory of Modeling and Simulation M&S Symposium*, pages 91–98, 2015. ISBN 978-1-5108-0105-9.
90. Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. A multi-paradigm modelling approach to live modelling. *Software & Systems Modeling*, 18(5):2821–2842, 2019.
91. xtUML.org. xtUML - eExecutable Translatable UML with BridgePoint. <https://xtuml.org>, 2016. Retrieved July 19, 2016.
92. Kebin Zeng, Yu Guo, and Christo K Angelov. Graphical model debugger framework for embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 87–92. European Design and Automation Association, 2010.