MODEL-LEVEL DEBUGGING IN THE CONTEXT OF THE
MODEL-DRIVEN DEVELOPMENT

by

MOJTABA BAGHERZADEH

A thesis submitted to the School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
September 2019

Copyright © Mojtaba Bagherzadeh, 2019
Abstract

Model-driven Development (MDD) is a software development approach that advocates using models rather than source code. MDD can decrease complexity by raising the level of abstraction, and increase productivity by automation, e.g., code generation.

Developers spend around 50% of their time on debugging and fixing bugs, and the cost of debugging, testing, and verification account for almost 50 – 75% of the software development budget. Existing MDD tools do not provide proper support for the debugging of models. Thus, developers often debug the generated source code in order to debug their models, which contradicts MDD principles, as many of the benefits of the abstraction are lost. This thesis introduces a novel approach to platform-independent model-level debugging, with the help of model transformation that equips the model with support for debugging services. The approach does not rely on program debuggers, and any changes to, e.g., the code generator or the hardware platform, leave the debugger unaffected. Based on the approach, a model-level debugger \textit{MDebugger} is created that allows the debugging of UML-RT models.

The typically iterative and incremental nature of software development implies that many development artifacts are incomplete until later stages of development. None of the model-level debuggers, including \textit{MDebugger}, allows the debugging of incomplete (i.e., partial) models, because the model execution techniques (interpretation and code generation) cannot execute partial models. This often prevents useful analyses that would allow the detection of design flaws and bugs early in the development process, when they are easiest to fix. We propose a conceptual framework to execute partial models. Based on the framework, an execution engine of partial models (\textit{PMExec}) is developed that allows the debugging and execution of partial UML-RT models at any stage of the software development process, assuming that the structural models are defined and proper input is provided.

Finally, to keep pace with program debugging services, and optimize the \textit{PMExec} and \textit{MDebugger} we introduce a new approach for live modeling that allows the updating of model elements during the execution.
Related Publications

Earlier versions of the work in this thesis are published or under review as listed below:


- Mojtaba Bagherzadeh, Karim Jahed, Benoit Combemale, and Juergen Dingel. Live modeling in the context of model-driven development and code generation. Software & Systems Modeling. (Under review) (Chapter 7)

Also, the following publications were produced during my Ph.D. study. However, they are not directly related to the thesis.


I would like to express my sincere gratitude to my advisor Professor Juergen Dinggel, for his support throughout my Ph.D. studies. During my studies, there were times when I was feeling hopeless, but his patience, encouragement, and trust were essential to me in dealing with it all.

Additionally, I would like to thank my thesis committee: Professors Ahmed E. Hassan and Mohammad Zulkernine, for their insightful comments and guidance. I also extend thanks to the members of my examination committee: Professors Lionel Briand, Yuan Tian, and Ying Zou, for taking the time to critique my work.

I thank my fellow labmates, especially Nafiseh Kahani, Nicolas Hili, Karim Jahed, and Majid Babaie, for the stimulating discussions, for the sleepless nights we spent working together before deadlines. My thanks also go to Professors James R. Cordy, Cor-Paul Bezemer, Francis Bordeleau, Benoit Combemale, Ernesto Posse, Bran Selic, Simon Redding, and Charles Rivet who have provided me insightful feedback and helped me vitally during my Ph.D.

Above all, I would like to thank my wife, for her love, constant encouragement and support, and for all the sacrifices she made throughout my Ph.D. But most of all, thank you for being my best friend. I owe you everything.

To my son, Soren, thank you for coming into our lives. Your existence has changed my perspective on life. You have made me stronger, better, and more fulfilled than I could have ever imagined.

I would like to thank my parents for supporting me spiritually throughout my life. Thank you for all of the life lessons that taught me how to become who I am now.

Finally, this thesis would not have been possible without the financial support of the Natural Sciences and Engineering Research Council (NSERC), for which I am incredibly grateful.
Statement of Originality

I, Mojtaba Bagherzadeh, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.
# Table of Contents

Abstract ......................................................... i

Related Publications ........................................... ii

Acknowledgments ................................................ iv

Statement of Originality ....................................... v

Table of Contents ............................................... vi

List of Tables .................................................. ix

List of Figures .................................................. x

1 Introduction .................................................. 1
   1.1 Problem Statement ......................................... 3
   1.2 Thesis Overview ............................................ 4
   1.3 Thesis Contribution ......................................... 8

2 Background ................................................... 9
   2.1 Real-Time Embedded Systems ............................... 9
   2.2 Model-Driven Development ................................. 10
   2.3 Model Transformation ...................................... 11
   2.4 Debugging .................................................. 12
      2.4.1 The Probe Effect ..................................... 14
      2.4.2 The Observability Problem ......................... 15
      2.4.3 Debugging Overhead ................................ 15
      2.4.4 Debugging of Timed Behaviours .................... 16
   2.5 UML for Real-Time .......................................... 17
      2.5.1 Modelling Structure of a System in UML-RT ....... 17
      2.5.2 A Running Example .................................. 19
      2.5.3 Modelling Behaviour of a System in UML-RT ....... 20
### 7 Live Modeling in the Context of the Code Generation  
**7.1** Introduction ........................................ 124  
**7.2** Background ........................................... 126  
  7.2.1 A Running Example  ................................... 126  
  7.2.2 Live Modelling by Leveraging Live Programming (Initial Experiment) ......................... 127  
**7.3** Approach ................................................ 130  
  7.3.1 An Overview ........................................... 130  
  7.3.2 Live Modeling of UML-RT Models ...................... 135  
**7.4** Tool Support ........................................... 141  
  7.4.1 Implementation ......................................... 141  
  7.4.2 Live-UMLRT Features .................................. 142  
**7.5** Evaluation ............................................. 145  
  7.5.1 Evaluation Metrics .................................... 145  
  7.5.2 Experiments ............................................ 147  
  7.5.3 Results ................................................ 149  
**7.6** Summary ............................................... 151  

### 8 Conclusion and Future Work  
**8.1** Future Work .......................................... 156  

**Bibliography**  

List of Tables

2.1 Detailed actions of the CTR USM ............................................ 22
2.2 Helper functions .............................................................. 23
5.1 Model complexity of use cases and instrumentation time ........ 74
6.1 Model complexity of use cases, worst case transformation and analysis time ..................................................... 115
6.2 Required time for loading and selection of execution rules .... 122
7.1 Model complexity of use cases, median of code generation and instrumenting time ................................................. 146
7.2 Performance of edit operations using our approach and live program 150
List of Figures

1.1 An overview of the thesis ........................................... 5
2.1 The structure of TrafficLight ........................................ 19
2.2 The USM of TrafficLight controller (CTR) ..................... 21
2.3 Execution rules of an USM ........................................ 26
3.1 Model-Level debugging workflow using program debuggers ... 34
4.1 The structure of FailOver system ................................. 45
4.2 The root of Server capsule's USM .............................. 46
4.3 The structure of DigitalWatch system ......................... 47
4.4 The structure of ParcelRouter system ......................... 49
5.1 USM of the TrafficLight controller that annotated with the transitions type ........................................... 53
5.2 Model-Level debugging workflow .............................. 54
5.3 Model instrumentation overview (an N2N transition) ........ 57
5.4 Taxonomy of events (adapted from [87]) ...................... 60
5.5 Alternative transformation rules .............................. 61
5.6 Execution trace .................................................. 63
5.7 MDebugger implementation overview ........................ 69
Model-driven Development is a model-centric software development approach in which models serve as the primary software development artifacts, rather than source code [42]. By leveraging abstraction and automation, MDD can simplify communication and design activities, increase compatibility between systems, and boost development efficiency. Thanks to existing MDD tools (e.g., [109], [16]) 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 (sometimes also referred to as simulation) or by the translation of models into existing programming languages, often by code generation (translational execution) [61].

The cost of debugging, testing, and verification account for almost $50 - 75\%$ of the software development budget [145]. Typically, developers spend a significant part of their time debugging applications and fixing bugs [141, 140] supported...
by sophisticated program debugging tools. However, modeling tools currently do not provide proper support for debugging and understanding the run-time behaviour at model-level [100, 99, 161]. Instead, developers must use a program debugger on the generated source code to debug their models, which contradicts model-driven development (MDD) principles and goals, as many of the benefits of the abstraction are lost. Additionally, understanding the generated source code can be challenging and error-prone for developers who use only models for development and are not versed in the target language, i.e., the language of the generated code is in [161].

Inadequate debugging support is an important obstacle to a broader adoption of MDD [127] and a certain amount of research has already been conducted to address the challenges and requirements for model-level debugging (e.g., [100, 99, 71, 92, 86, 84, 69, 176, 190]). The most frequent proposals are, to (1) realize model-level debugging through a model interpreter, or to (2) maintain traceability information between model and code and leverage an existing program debugger. Interpreter-based proposals require the implementation of an interpreter for the modeling language, the action code language, and the run-time services; this is not only time-intensive but also creates a potentially harmful discrepancy between the environment that the model is debugged in and the environment that the code generated from the model will execute in; this discrepancy might create spurious bugs or mask real bugs that depend on, e.g., the choice of target language or hardware platform; also, we would not expect a high degree of debugger portability from one MDD tool to the next. Program debugger-based approaches
are very dependent on the target language, the operating system, and the hardware platform, meaning a change in any of these is likely to necessitate substantial changes to the debugger. Also, the use of a program debugger for model debugging exposes the ‘semantic gap’ between the modeling language and the target language. Typically, this gap is substantial, which means that some modeling concepts cannot easily be mapped to corresponding programming language concepts, and vice versa, and that the translation cannot be ‘hidden’ [185]. E.g., according to the UML-RT execution semantics, a capsule (i.e., component) instance is assigned to its own logical thread at model-level (i.e., it has its own independent flow of execution) and multiple logical threads can be assigned to a single physical thread in the target language. However, since the notion of logical threads is typically not supported by the program debugger, debugging capsule instances using program debuggers becomes unnecessarily complicated.

1.1 Problem Statement

Overall, given its significance, debugging appears to be an insufficiently researched topic, not just in the MDD context, but also for more traditional software development using general-purpose or domain-specific languages. Enhancing model-level debugging techniques can save time and cost of the debugging models, and contribute to the more widespread use of MDD. Also, it can stimulate and enable follow-up work to facilitate additional quality assurance activities such as model-based testing, simulation, and analysis.
Thesis Statement: The problem of model-level debugging can be addressed using model transformation techniques at model-level. This allows the support of model-level debugging in a way that is compatible with the semantics of the modeling languages and independent of programming languages used for the generated source code and program debuggers.

Indeed, in this thesis, we address three essential problems of model-level debugging, with the help of model-transformation techniques. First, we propose an approach for realizing model-level debugging that is platform-independent. To this end, we use model transformation to instrument a model to be debugged with information that allows it to support debugging activities. Second, we propose a conceptual framework for the execution and debugging of partial models, which uses model transformation to fix problematic elements of the model, to allow their execution. Third, we address live modeling (i.e., updating models’ elements at run-time) 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 that equips the model with support for, e.g., debugging and state transfer, both of which are required for live modeling.

1.2 Thesis Overview

In the following, we present a brief overview of the thesis, as shown in Figure 1.1. We first present introductory chapters, as follows.
1.2. THESIS OVERVIEW

Figure 1.1: An overview of the thesis

**Chapter 2: Background and Definitions**

In this chapter, we provide the reader with background information and define key terms that we will use throughout the thesis.

**Chapter 3: Related Work**

To position this thesis with respect to prior research, we present a survey of research on model-level debugging, partial models, and live modeling.

**Chapter 4: Case Studies**

In this chapter, we describe the case studies that are used to evaluate our proposed solutions, throughout the thesis.
1.2. THESIS OVERVIEW

Next, we shift our focus to the main body of the thesis. We provide novel solutions to three important problems, which are presented in three separate chapters. Each chapter consists of the following sections. (1) **Background:** which provides complementary background information specific to the chapter (if required) and a description of a running example. (2) **Approach:** which motivates and articulates the proposed solutions. (3) **Tool Support:** which briefly discusses the developed prototypes and their features from the users’ points of view. For each prototype, an open source repository and a demonstration video are provided. (4) **Evaluation:** which discusses the evaluation methods, conducted experiments, and results. At the end of each chapter, a summary of the chapter is presented. In the following, we present an overview of the main chapters.

**Chapter 5: Model-level Platform-Independent Debugging**

Providing proper support for debugging models at model-level is one of the main barriers to a broader adoption of Model Driven Development (MDD). In this chapter, we introduce a new platform-independent approach to implementing model-level debuggers. We describe how to realize support for model-level debugging entirely in terms of the modeling language, and show how to implement this support in terms of a model-to-model transformation. We also describe an implementation of the approach in the context of Papyrus-RT [8], an open source MDD tool based on the modeling language UML-RT [158].

**Chapter 6: Debugging and Execution of Partial Models**

The typically iterative and incremental nature of software development implies that many development artifacts are incomplete until later stages
of development. Unfortunately, this partiality often prevents useful analyses that would allow the detection of design flaws and bugs early in the development process, when they are easiest to fix [179]. Existing work on dealing with partial models at design time allows for the specification, analysis, verification, and transformation of partial models (e.g., [74]). However, to the best of our knowledge, no work has addressed the execution (through the interpretation of models or code generation) of partial models. In this chapter, we present an approach and supporting tool that addresses this problem in the context of MDD. Our approach allows the execution and debugging of partial models, thus facilitating early detection and correction of flaws.

Chapter 7: Live Modelling in the Context of the Code Generation

Live programming [58, 134] 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. Inspired by this line of work, some efforts [181, 177, 183] have recently been made towards live modeling, i.e., the application of the changes to the models while they are being executed. However, they have only focused on model interpretation, and no work supports live modeling when the models are executed by code generation into general programming languages (GPL). In this chapter, we propose a conceptual framework and supporting tool for supporting live modeling in the context of model execution 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.

Finally, in Chapter 8, we draw conclusions and discuss future research directions.

1.3 Thesis Contribution

This thesis advances the state of the art in the context of model debugging and execution, theoretically and practically, by proposing novel and efficient solutions along with open source tooling to model-level debugging, execution/debugging of partial models, and live modeling. The most important contributions of the thesis are as follows.

- Proposing a novel approach for model-level platform-independent debugging.

- Proposing a conceptual framework for execution of partial models and implementation and formalization of the framework in the context of UML-RT.

- An efficient solution for live modeling that can also enable dynamic adaptation and help with optimizing of our proposed solutions for the partial model execution and model-level debugging.

- Three open source tools, as proof of concepts for the proposed solutions, that can be integrated with existing tools and extended for future research.

- Experimental analysis of the proposed solutions and providing supporting evidence that shows the problem of model-level debugging can be supported using model transformations.
The objective of this chapter is to discuss the terms and concepts that are used in this work. First, we introduce the Real-Time Embedded (RTE) systems, Model-Driven Development (MDD), Model Transformation (MT), and debugging. Then, we describe UML for Real-Time (UML-RT) using a running example, and present a formal and concise specification of UML-RT.

2.1 Real-Time Embedded Systems

A Real-Time Embedded (RTE) system is a computer system whose correctness depends not only on the logical results of the computations, but also on the time within which results are produced [110]. These systems are present in quite diverse application domains, including, but not limited to, automotive electronics, avionics, railways, telecommunication, the health sector, security, consumer
electronics, fabrication equipment, smart buildings, logistics, robotics, and military applications. RTE systems often require a higher reliability and safety standard than general-purpose computing, and often their failure is unacceptable. In some cases, deviations from expected behaviour can cause catastrophic failure. Often, these systems are classified, according to their failure consequences, into *hard* and *soft* real-time systems. A hard real-time system is one in which any timing fault is intolerable, and can cause catastrophic failure. Fault occurrence in a soft RTE system only downgrades the system performance, however does not lead to catastrophic failure [119, 115, 128, 12]. Our work uses the UML-RT language, which is concerned with soft RTE systems.

Due to the complexity of RTE systems, and the essential safety requirements, the software development methods used for these systems must provide effective means to develop software and prove its correctness. MDD has recently been enjoying increasing popularity, and is seen as a key technology to deal with the complexity and demands of future systems [28]. With MDD becoming more prevalent, a lot of research and development has been directed toward introducing new tools, standards, and languages in this area. UML, SDL, SysML, IBM Rational Rhapsody, Papyrus-RT, AF3 [15, 8, 27, 11, 30, 31, 158, 11] are examples of these tools and standards. Our work uses Papyrus-RT as a modeling tool.

### 2.2 Model-Driven Development

MDD which is used interchangeably with Model-driven Engineering (MDE) typically refers to software development approaches in which models are used to define, develop, and communicate a solution rather than source code or other
artifacts. Models usually are abstract and close to the problem domain. Thus, they can simplify the understanding of a complex problem and its potential solutions. Often, the defined models are systematically transformed into concrete implementations using model transformation. Models of a system can be used at both design-time and run-time, to provide an abstract, precise, and unambiguous representation of the system. Moreover, the defined models may be used for different activities, other than implementation, such as performance analysis, load, safety, liveness, reliability, and automation of testing and documentation [42, 160, 79]. Meta-models are used to define the appropriate and necessary structures, and properties to which a model must conform. Metametamodels, such as MOF [21], Ecore [7], and KM [17] provide infrastructure to define metamodels. Usually, metametamodels are self-defined, i.e., defined in terms of themselves [17, 104].

In general, the MDD approaches can be classified into UML-like and non-UML categories. The UML-like category uses UML (the Unified Modeling Language) as a base for the modeling, while the non-UML category is based on modeling techniques other than UML, such as formal specification languages (e.g., Alloy) [79, 35, 101]. Our work is concerned with UML-like approaches, which are defined using Ecore metamodeling, and support generation of executable implementations from models.

2.3 Model Transformation

A model transformation is a program used to transform a model from one representation to another. The input to the transformation is called an input model,
which conforms to a source metamodel, and its output is an output model, which conforms to a target metamodel (in model-to-model (M2M) transformations), or grammar (in model-to-text (M2T) transformations). A model transformation definition written in a model transformation language defines how one or more input model(s) are transformed into one or more output model(s). If the language of the transformation definition is rule-based, the transformation definition will consist of a set of transformation rules. The transformation engine/tool uses the model transformation definition to produce output model(s) from input model(s) [107, 105].

Based on the representations of the input and output models of the transformation, model transformation tools can be classified into three main categories: model-to-model (M2M), model-to-text (M2T) and text-to-model (T2M). The output of a M2M transformation is an instance of a target metamodel, whereas M2T approaches typically use target grammars to describe the structure of their textual output [107, 105].

2.4 Debugging

Debugging is the process of understanding, locating, and fixing software errors [93] to eliminate deviations between the expected and existing behaviours. The complexity and cost of the debugging vary based on the time of the debugging, programming approaches, and the type of software to be debugged. Typically, developers debug software during different stages of the software lifecycle, i.e., development, testing, and deployment, and the debugging complexity increases with each step [93, 116]. The programming approaches used to develop software, such
as sequential, concurrent, and event-driven, affect debugging complexity. For example, the debugging of sequential systems is easier than debugging concurrent or event-driven programs [135], as parallel programs are often not deterministic [135, 117]. Moreover, the complexity and constraints imposed by the type of the system complicate the debugging process. For example, the debugging of RTE systems with limited resources (e.g., memory and processing unit) is more complicated than the debugging of a system with sufficient resources, or debugging a distributed system is harder than a centralized system. RTE systems are usually concurrent, event-driven, and distributed with resource constraints, which make their debugging a very challenging task. Our work is mainly concerned with concurrency and event-driven aspects of RTE systems.

Existing debugging techniques to understand and analyze the dynamic behaviour of software fall mainly into three classes [116, 135, 117]: (1) Tracing techniques that use logging statements to generate logs. The logs are collected and analyzed to understated systems’ behaviours. These techniques range in complexity from simply printing the logs to high-performance, scalable, and configurable tracing frameworks, such as LTTng [70]. (2) Live debugging (controlled execution) that allows us to interrupt the execution of a program at specific points, control execution flow, and examine program state (e.g., view variables and program stack). Program debuggers, such as the GNU debugger (GDB) [5] and the Java Debugger [32] provide a full set of features for controlled execution of software written in C, C++, and Java. (3) Static analysis techniques applied to detect anomalies in program behaviour. These techniques are distinct from formal methods
because they only attempt to show that the program cannot enter certain predefined erroneous states without performing a formal proof. The techniques do not require execution of programs for debugging, which makes them interesting for a certain type of problems, such as synchronization errors (e.g., deadlock detection), and data usage errors (e.g., reading an uninitialized variable). Our work is mainly focused on tracing and live debugging techniques.

Also, fault localization of programs/models aims to automatically find pieces of code or elements of models that account for faulty behaviour, e.g., delta debugging [189], algorithmic debugging [59, 167], and statistical debugging [120], model-level statistical debugging of Simulink models [122, 121]. These techniques can be built on top of the debugging techniques presented in this thesis.

2.4.1 The Probe Effect

Werner Heisenberg introduced the uncertainty principle, showing that in quantum mechanics, measurement or observation can affect the behaviour of the observed object. In software engineering, this principle is called probe effect [83, 118, 98, 135] and refers to the different behaviours of a concurrent program that result when embedding a small delay in the execution flow. The delay may change the behaviour of the program in either positive or negative form: (1) In the positive form, a bug in the program may go away, and an erroneous concurrent program works correctly after embedding the delay. This type of bug is referred to as a “Heisenbug”. (2) In the negative form, a new bug may be introduced, and a functioning program stops working after introducing the delay. Both debugging techniques (i.e., tracing and live debugging) introduce delays in program execution,
which can lead to probe effects.

2.4.2 The Observability Problem

The observability problem in distributed systems is the difficulty of understanding the behaviour of the observed program. Even when gathering detailed traces of program execution, we may not be able to detect and understand the correct behaviour of the system for some reason, such as: (1) In a system with many observers, due to the propagation delays associated with the messages, observers may receive events in a different order. (2) Due to synchronization issues due to differences between processor clocks, observers may see incorrect orderings of events. (3) Some of the events in the system are unrelated. However, the observers can relate them based on their execution order. Logical clocks [78, 150] are one of the existing solutions to deal with the observability problem.

2.4.3 Debugging Overhead

In addition to the probe effect and observability problems, debugging techniques impose overheads, such as increasing the program size and downgrading the performance. The former can be a result of program instrumentation on the source code level or binary code level. Usually, the size overhead is a function of program size and the granularity level of the instrumentation. Increasing the size of a program can increase the memory and disk space required for saving, and the time needed for loading the program. The increased execution time of a program is due to the time required for the execution of logging statements. Similar to size overhead, the granularity level of the instrumentation has an impact
on the performance overhead. Despite the size overhead, the performance overhead is intolerable specifically in time-sensitive systems. To address this, several studies have tried to provide solutions for efficient tracing and minimization of the performance overhead. The proposed solutions are classified into software and hardware categories. The former solutions use effective data structures, algorithms, and communication protocols to provide efficient tracing, e.g., [70, 88]. The hardware solutions provide facilities, at the hardware level, to efficiently observe the execution using hardware counters such as the Linux perf utility [18].

2.4.4 Debugging of Timed Behaviours

In addition to the issues discussed in Sections 2.4.3, 2.4.2, and 2.4.1 there is one specific issue related to RTE systems: dealing with time. Developers use timers to manage the timing requirements of the RTE systems. The timers can be provided by operating systems and run-time libraries, or be implemented directly using hardware interrupts. In all cases, the time progresses based on the wall-clock time, and there is no direct way to stop and resume the timers while stopping and resuming the execution of the RTE systems, during debugging.

Despite this limitation, there are still many cases where the existing debugging techniques can assist in the debugging of RTE systems. For example, the system execution can be stopped in places where there is no active timer.
2.5  UML for Real-Time

In this thesis, we use the UML for Real-Time (UML-RT) language to implement and evaluate our proposed solutions. UML-RT \([162, 149]\) is a language specifically designed for 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, and has a long, successful track record of application and tool support, via, e.g., IBM RSA-RTE \([16]\), RTist \([13]\), Eclipse eTrice \([6]\), and Papyrus-RT \([8]\). UML-RT is designed as a UML profile, with simplified notations and concise syntax and semantics. It only provides two kinds of diagrams: capsule and state machine diagrams. In the following, we provide an informal and formal description of UML-RT. Our formalization is simplified and focused on aspects that matter most to the thesis, and interested readers can refer to \([149, 162]\) for more in-depth information regarding UML-RT.

2.5.1  Modelling Structure of a System in UML-RT

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 internal and external \textit{ports} that are typed with \textit{protocols}. A protocol defines the different incoming and outgoing \textit{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 \textit{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) [162, 158].

**Definition 1.** (Read function (Projection)) Let \( tp \) be a tuple of attributes \( \langle a_1, \ldots, a_n \rangle \) where \( a_1 \ldots a_n \) refer to attributes names. We use \( tp.a_i \) or \( a_i(tp) \) to read the value of attribute \( a_i \). E.g., to read the value of attribute *name* of tuple \( \text{person} = \langle \text{name}, \text{family} \rangle \) we can use \( \text{person.name} \) or \( \text{name(person)} \).

**Definition 2.** (Formal definition of UML-RT structural model) Let us define a protocol as a set of pairs \( (m, d) \), where \( m \in M \) (i.e., a universal set of messages) is a message, and \( d \in \{ \text{input}, \text{output} \} \) 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 component as a tuple \( \langle P, V, \beta \rangle \), where \( P \subseteq P \) (i.e., a universal set of ports) is a set of ports, \( V \) is a set of variables, and \( \beta \) refers to the specification of the component’s behaviour. A port is defined as a pair \( (t, \text{conjugated}) \), where \( t \in I \) refers to the type of the port, and \( \text{conjugated} \in \{ \text{true}, \text{false} \} \) 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 \( \langle C, I, \text{con}, \text{in} \rangle \), where \( C \) is a set of components, \( I \) is a set of protocols, \( \text{con} \) is a connectivity relationship \( \subseteq P \times P \), and \( \text{in} \) is an acyclic containment relationship \( \subseteq C \times C \). Whenever two ports \( p_1, p_2 \) are connected by \( \text{con} \) (i.e., \( (p_1, p_2) \in \text{con} \)) 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’.
2.5. UML FOR REAL-TIME

2.5.2 A Running Example

We use the control system of a simple traffic (TrafficLight) light as a running example throughout the thesis. The top-level structure of the system is shown in Figure 1, which consists of three capsules: UserConsole (UC), Controller (CTR), and StopLightDriver (SLD). The CTR is connected to UC and SLD using two ports UCPort and SLDPort, which are typed by protocols ControlP and StopLightP accordingly. ControlP has two messages (on() and off()) and StopLightP has five messages (red(), green(), yellow(), on(), off()). The UC component collects user input, which it passes on to the CTR component, the component controlling the light. Using the corresponding messages, the CTR component sends the control actions to the SLD component, which transfers the messages through a hardware port to the traffic light.

Figure 2.1: The structure of TrafficLight
2.5. UML FOR REAL-TIME

2.5.3 Modelling Behaviour of a System in UML-RT

Capsule behaviour is modeled using *UML-RT State Machines (USM)*. A 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 is 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 transitions actions are expressed using an action language. Action languages support primitive operations such as accessing/updating variables, arithmetic/boolean expressions, control flow constructs, and sending 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 [1] is designed for UML, and *YAKINDU* [38] provides its own action language. In this work, we assume the existence of an action language with the standard capabilities, but do not define a particular syntax for it.

**Definition 3.** *(Formal definition of USM)* An *USM* is defined as a tuple \((S, T, \text{in})\). \(S = S_b \cup S_c \cup S_p\) is a set of states, \(T\) is a set of transitions, and \(\text{in} \subseteq S_c \times (S \cup T)\) denotes an acyclic containment relationship. States can be basic \((S_b)\), composite \((S_c)\), or pseudo-states \((S_p)\). There are six kinds of pseudo-states, including *initial*, *choice-point*, *history*, *junction-point*, *entry-point*, and *exit-point*, (i.e., \(S_p = S_{in} \cup S_{ch} \cup S_h \cup S_j \cup S_en \cup S_ex\)).
2.5. UML FOR REAL-TIME

Let $inp(c)$ refer to the messages that can be received by the owner component $c$ of the USM. A transition $t$ is a 5-tuple $(src, guard, trig, act, des)$, where $src, des \in S$ 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 expressed using the action language.

An example: the behaviour of capsule $CTR$ of TrafficLight is shown in Figure 2.2 that has seven states and seven transitions. $in_{11}$ is an initial state, state $on$ is a composite state, states $s_{11}, s_{21}, s_{23}, s_{23}$ are basic states and $en_{1}$ is an entry-state. Transitions are annotated with $trigger[guard]/action$. Table 2.1 shows the details.
2.5. UML FOR REAL-TIME

Table 2.1: Detailed actions of the CTR USM

<table>
<thead>
<tr>
<th>Action</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>act₀</td>
<td>red_time=3000; green_time=3000; yellow_time=1000;</td>
<td>Set the initial values of the variables.</td>
</tr>
<tr>
<td>act₁</td>
<td>SLDPort.on().send(); timer1.informIn(UMLRTTimespec(red_time,0));</td>
<td>Turns on the traffic-light and starts timer timer1 for 3000ms.</td>
</tr>
<tr>
<td>act₂₁</td>
<td>SLDPort.red().send();</td>
<td>Turns on the red light.</td>
</tr>
<tr>
<td>act₂₂</td>
<td>SLDPort.green().send(); timer1.informIn(UMLRTTimespec(green_time,0));</td>
<td>Starts timer timer1 for 3000ms and turns on the green light.</td>
</tr>
<tr>
<td>act₂₃</td>
<td>SLDPort.yellow().send(); timer1.informIn(UMLRTTimespec(yellow_time,0));</td>
<td>Starts timer timer1 for 1000ms and turns on the yellow light.</td>
</tr>
<tr>
<td>act₂₄</td>
<td>SLDPort.red().send(); timer1.informIn(UMLRTTimespec(red_time,0));</td>
<td>Starts timer timer1 for 3000ms and turns on the red light.</td>
</tr>
<tr>
<td>act₂</td>
<td>timer1.cancel(UMLRTTimespec(red_time,0)); SLDPort.off().send();</td>
<td>Cancels the timer timer1 and turns off the traffic light.</td>
</tr>
</tbody>
</table>

of the actions using the UML-RT action language and a brief description of them.

Definition 4. (Helper functions) Table 2.2 lists the helper functions (along with samples in the context of the running example, if possible) that will be used in the rest of the thesis. Note that we treat the root of an USM as a composite state, which can be accessed using the root(USM) function.

2.5.4 Execution Semantics of UML-RT

The semantics of UML-RT state machine is similar to that of UML state machine, with some restrictions, including: (1) there is no AND-state (orthogonal regions), (2) the UML concepts fork, join, shallow history, and final states are prohibited in UML-RT, (3) transitions cannot cross state boundaries, and (4) states do not have idle (do) actions [149].

The execution of UML-RT is managed by a Run-Time System (RTS) library, which defines one or more controllers to monitor the concurrent execution of
### Table 2.2: Helper functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inp(c)</code></td>
<td>returns possible input messages of component <code>c</code>. E.g., <code>inp(CTR)= {on,off,red,green,yellow}</code>.</td>
</tr>
<tr>
<td><code>in_t(s)</code></td>
<td>returns incoming transitions to state <code>s</code>. E.g., <code>in_t(s11)= {t11, t13}</code>.</td>
</tr>
<tr>
<td><code>out_t(s)</code></td>
<td>returns outgoing transitions from state <code>s</code>. E.g., <code>out_t(s11)= {t12}</code>.</td>
</tr>
<tr>
<td><code>handled(s)</code></td>
<td>returns triggers of outgoing transitions of <code>s</code> and <code>parents(s)</code>. E.g., <code>handled(s21)= {off, timeout}</code>.</td>
</tr>
<tr>
<td><code>root(sm)</code></td>
<td>returns root of the HSM <code>sm</code>.</td>
</tr>
<tr>
<td><code>child(s)</code></td>
<td>returns states inside state <code>s</code>. E.g., <code>child(on)= {s21, s22, s23}</code>.</td>
</tr>
<tr>
<td><code>parent(s)</code></td>
<td>returns the first-level container state of state <code>s</code>. E.g., <code>parent(s21)= {on}</code>.</td>
</tr>
<tr>
<td><code>parents(s)</code></td>
<td>returns all container states of state <code>s</code>. E.g., <code>parents(s21)= {on, root(TUSM)}</code>.</td>
</tr>
<tr>
<td><code>dead(s)</code></td>
<td>returns <code>true</code> if state <code>s</code> and its parents do not handle any message (i.e., <code>handled(s)=∅</code>). E.g., <code>dead(s22)=false</code>.</td>
</tr>
<tr>
<td><code>u_h(s, h)</code></td>
<td>if <code>parent(s) ≠ ∅</code> and <code>s ∈ S_b</code>, updates the last visited state of <code>parent(s)</code> to <code>s</code> (i.e., entry of <code>parent(s)</code> in <code>h</code>) and returns the updated <code>h</code>.</td>
</tr>
<tr>
<td><code>head(q)</code></td>
<td>reads, removes, and returns the first element in queue <code>q</code>.</td>
</tr>
<tr>
<td><code>next_s(s_c, H)</code></td>
<td>(1) returns the last visited state inside state <code>s_c</code> from history <code>H</code>, (2) if (1) is unsuccessful (i.e., the composite state is active for the first time), returns the default state (initial state) inside <code>s_c</code>, and (3) if (1) and (2) are unsuccessful, returns <code>∅</code>. E.g., if <code>H = ∅</code> then <code>next_s(s11, H) = ∅</code>,</td>
</tr>
<tr>
<td><code>next_t(s, µ)</code></td>
<td>checks state <code>s</code> and its ancestors in bottom-up order, and returns the first (i.e., most deeply nested) outgoing transition, which can be triggered by message <code>µ</code>. It returns <code>∅</code> if no transition can be triggered. E.g., <code>next_t(s21, on)=∅, next_t(s21, off)=t13</code>.</td>
</tr>
<tr>
<td><code>up_s(s, t)</code></td>
<td>returns <code>s</code> and a subset of its parents in bottom-up order from state <code>s</code> to the state that <code>t</code> originated from. E.g., <code>up_s(s21, t13)= s21, c11</code>.</td>
</tr>
<tr>
<td><code>eval(E, g)</code></td>
<td>evaluates guard <code>g</code> based on the values in map <code>E</code> and returns the result.</td>
</tr>
<tr>
<td><code>exec(E, a_1 . . . a_n)</code></td>
<td>executes a sequence of actions <code>a_1 . . . a_n</code> based on the values in map <code>E</code> and returns the updated <code>E</code>.</td>
</tr>
</tbody>
</table>

( `s` is a state, `sm` is an HSM, `t` is a transition, `q` is a queue, `E` is mapping from variables to their values, `s_c` is a composite state, `a_1 . . . a_n` is a sequence of actions.)

Each capsule. A controller is assigned to a physical thread, and controls the execution of a set of capsules. An important characteristic related to the execution
semantics of UML-RT is run-to-completion, which guarantees that an incoming
message will be fully processed before the processing of the next message starts.

**An example:** when the execution of *USM CTR* (ref. Figure 2.2) starts, the transi-
tion \( t_{11} \), (i.e., initial transition) is taken. The action of \( t_{11} \) (i.e., \( act_0 \) in Table 2.1) and
the entry action of state \( off \) are executed in this order. State \( off \) remains active un-
til a message matching the trigger of transition \( t_{12} \) (i.e., a message \( on \)) is received.
Upon reception of a message \( on \), \( t_{12} \) is taken and a sequence of actions, consisting
of the exit action of state \( off \), the transition action of \( t_{12} \) and \( t_{21} \), and the entry ac-
tion of state \( red \) are executed. Run-to-completion guarantees that the execution
of the entire sequence of action is uninterrupted, until the execution reaches the
next basic state.

**Definition 5. (Formal definition of the execution semantics of USM)** We use a La-
beled Transitional System (LTS) which is a tuple \( \langle \Gamma, \mathcal{A}, \gamma_0, Q, \rightarrow \rangle \) to define the exe-
cution semantics of an *USM*. \( \Gamma \) is a set of configurations, \( \mathcal{A} \) is a set of actions (i.e.,
entry, exit, and transition actions defined in *USM*), \( Q \) is a first-in, first-out (FIFO)
queue that stores received messages, \( \rightarrow \) is a transition relation (to avoid confusion
with the syntax of *USM*, we use the term ‘execution step’ instead of ‘transition’ in
the rest of the paper), and \( \gamma_0 \in \Gamma \) is the initial configuration. A configuration \( \gamma \in \Gamma \)
is defined as a tuple \( \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \) where \( \sigma \in S \) refers to the current state of the con-
figuration, \( \mathcal{E} \) refers to a mapping from the component variables to values, and \( \mathcal{H} \)
is a mapping from composite states to their last visited sub-states (if any).

An execution step is defined as a tuple \( \langle \gamma, a_1 \ldots a_n, \gamma' \rangle \) that moves the execution
from configuration \( \gamma = \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \) (source configuration) to configuration
\( \gamma' = \langle \sigma', \mathcal{E}', \mathcal{H}' \rangle \) (target configuration), while executing a possibly empty sequence
of actions $a_1 \ldots a_n$ with $a_i \in \mathcal{A}$ for all $1 \leq i \leq n$ that may result in updating $\mathcal{H}$ and $\mathcal{E}$, and producing outputs. We use the following notation to show an execution step.

$$\langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{\mathcal{E} \leftarrow \text{exec}(\mathcal{E}, a_1 \ldots a_n)} \langle \sigma', \mathcal{E}', \mathcal{H}' \rangle$$

Let us define a stuck configuration as a configuration that no execution step can start from, i.e., the execution cannot progress anymore when it reaches a stuck configuration. We use notation $\gamma \not\rightarrow$ to show that configuration $\gamma$ is a stuck configuration.

The initial configuration $\gamma_0$ is set to $\langle \text{initial}, \mathcal{E}_0, \emptyset \rangle$ at the beginning of the execution, where $\text{initial}$ refers to the initial state inside the root of the USM (i.e., $\text{initial} = S_{\text{in}} \cap \text{child}(\text{root}(USM)))$ and $\mathcal{E}_0$ refers to default values of the variables. Note that if the initial state is not defined, the execution cannot start (missing initial state).

Let us assume that $\gamma = \langle \sigma, \mathcal{E}, \mathcal{H} \rangle$ refers to the current configuration. The rules in Figure 2.3 define the operational semantics [146] of USMs. The presentation of the rules makes use of definitions from Table 2.2. The rules are based on existing implementation of UML-RT (i.e., Papyrus-RT [8], RSARTE [16], eTrice [6], and Rtist [13]) and inspired by the execution semantics of UML-RT, presented in [149, 186]. Details of the rules are as follows.

**Rule-1, 2:** These rules are applicable to configurations whose current state is one of the pseudo-states, except for history and choice-point. According to Rule-1, an execution step is taken if there is an outgoing transition from the current state
\[ \sigma \in S_p \setminus (S_h \cup S_{ch}), t = \text{out}_t(\sigma) \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{E' \leftarrow \text{exec}(\mathcal{E}, \text{act}(t), \text{entry}(t, \text{des})))} \langle t, \text{des}, \mathcal{E}', u_{\text{h}}(t, \text{des}, \mathcal{H}) \rangle \]

\[ \sigma \in S_p \setminus (S_h \cup S_{ch}), \text{out}_t(\sigma) = \emptyset \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{} \]

\[ \sigma \in S_b, \text{dead}(\sigma) \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{} \]

\[ Q \neq \emptyset, \sigma \in S_b, \neg \text{dead}(\sigma), t = \text{next}_t(\sigma, \text{head}(Q)) \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{E' \leftarrow \text{exec}(\mathcal{E}, \text{exit}(t, \text{src}), \text{exit}(\text{up}_s(y, \sigma, t)), \text{act}(t), \text{entry}(t, \text{des})))} \langle t, \text{des}, \mathcal{E}', u_{\text{h}}(t, \text{des}, \mathcal{H}) \rangle \]

\[ Q \neq \emptyset, \sigma \in S_b, \neg \text{dead}(\sigma), \text{next}_t(\sigma, \text{head}(Q)) = \emptyset \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{} \]

\[ \sigma \in S_c, s = \text{next}_s(\sigma, \mathcal{H}) \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{E' \leftarrow \text{exec}(\mathcal{E}, \text{entry}(s)))} \langle s, \mathcal{E}', u_{\text{h}}(s, \mathcal{H}) \rangle \]

\[ \sigma \in S_c, \text{next}_s(\sigma, \mathcal{H}) = \emptyset \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{} \]

\[ \sigma \in S_{ch}, t = \text{out}_t(\sigma) \land \text{eval}(\mathcal{E}, t, \text{guard}) \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{E' \leftarrow \text{exec}(\mathcal{E}, \text{act}(t), \text{entry}(t, \text{des})))} \langle t, \text{des}, \mathcal{E}, u_{\text{h}}(t, \text{des}, \mathcal{H}) \rangle \]

\[ \sigma \in S_{ch}, \exists t \in \text{out}_t(\sigma) \mid \text{eval}(\mathcal{E}, t, \text{guard}) \]

\[ \langle \sigma, \mathcal{E}, \mathcal{H} \rangle \xrightarrow{} \]

Figure 2.3: Execution rules of an USM

that executes the related actions and moves the execution to a new configuration. Conversely (Rule-2), if there is no outgoing transition, the execution stops there, and the current configuration is considered stuck (issue ‘broken chain’ in Chapter 6).

**Rule-3, 4, 5:** These rules are applicable to configurations whose current state is a
basic state. If the current state is a dead state (Rule-3), the execution stops there, and the current configuration is considered stuck (issue dead state). Otherwise, if a message exists in the queue, one of the following rules is applied based on the result of the function \( \text{next}\_t(\sigma, \text{head}(Q)) \) (Ref. Table 2.2). (Rule-4) a transition can be triggered, which results in an execution step that executes the related actions and moves the execution to a new configuration, as shown in the bottom of the rule. Conversely (Rule-5), if a transition cannot be triggered (i.e, the incoming message is an unexpected message), an execution step cannot be taken. We consider the configuration to be stuck. However, it is also possible to configure the RTS to throw away the unexpected messages. As a result, the execution can recover and continue. We argue that in the domain of RTE systems, in which most of the applications are safety-critical, it is not safe to throw away any message.

**Rule-6, 7:** These rules are applicable to configurations whose current state is a composite state (implicit history state). If function \( \text{next}\_s(\sigma, \mathcal{H}) \) (Ref. Table 2.2) returns a state, then an execution step is taken that applies the entry code of the related composite state and moves the execution to a new configuration, as shown in the bottom of the rule (Rule-6). Conversely, if the selection is unsuccessful, the execution cannot move, and the configuration is a stuck configuration (Rule-7).

This can happen for two reasons: (1) the current state has no child (issue childless composite state), (2) the current state has no initial state ('missing initial state').

**Rule-8, 9:** These rules are applicable to configurations whose current state is a choice-point. Guards of the outgoing transitions from the current state are evaluated, and the first transition whose guard evaluates to true is selected. This results
in an execution step that executes the related action code and moves the execution to a new configuration, as shown in the bottom of the rule (Rule-8). Conversely, if none of the outgoing transitions’ guards holds (issue ‘non-exhaustive guards’), the execution cannot move, and the configuration is a stuck configuration.

Note that Rules 2, 3, 5, 7, and 9, which are related to stuck configurations, can be merged into one rule. However, we refer to different rules for the sake of clarity.
In this chapter, we survey the related research for the thesis. We organize the work into three sections: model-level debugging, partial models, and live modeling. At the end of each section, we discuss how the related work motivates our proposed solutions.

3.1 Model-Level Debugging

In the context of MDD, models can be executed either by interpretation (sometimes also referred to as simulation) or by the translation of models into existing programming languages, often by code generation (translational execution) [61]. The translational execution allows the execution of the model on the target platform.
3.1. MODEL-LEVEL DEBUGGING

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.

3.1.1 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 [29], AF3 [2], xtUML [11], and YAKINDU [38].

Mierlo et. al [137] address debugging of PythonPDEVS [184], 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 [82]. To support the debugging, they model the modal part of the simulator using a statechart, and include the debugging logic in the state-chart. This provides the debugging of models through the debugging of the simulator. In his thesis [180], 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) [148]. Our work on platform-independent model-level debugging (ref. Chapter 5) is close to this work. However, our work supports debugging when models are executed by code generation, and we instrument user-defined models.

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., [55, 65]). Corley et al. [64, 63] explore this approach to the debugging of model transformations in AToMPM [175]. Their implementation records each change at the end of a transformation step, and provides support to step back to the previous states. Tendeloo et al. [182] address omniscient debugging for a PDES models. The deterministic formalism of PDES and their implementation based on interpretation can minimize the required traces by removing arbitrary intermediate states, as they can be computed again using interpretation.

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. [57, 56] 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
3.1. MODEL-LEVEL DEBUGGING

semantics defined as an execution metamodel and transformation. They specify details of the execution based on the execution metamodel using GMOC studio [54].

Overall, while the simulation is necessary and useful for finding bugs at early design-time, based on functional requirements, it is insufficient for situations where bugs can be caused by either the sensitiveness of the system to timing constraints [92, 10, 157, 86] or the configuration of the target platform. Such bugs can be found using complex representations of the system, where timing constraints and the environment are modeled or by the execution of the system on the target platform. However, this requires more sophisticated environments and mechanisms.

3.1.2 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. [96] 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 [92, 176, 100, 99, 84]. Iyengar et al. [100, 99] 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. [69] propose a configurable tracing tool based on LT-Tng. 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 [84] 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 [69], or by being stored in mapping files [100, 99].

**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 [27], IBM RSARTE [16], and Timing Architects [33] support live debugging capabilities. Further, some studies try to address model-level debugging using traditional approaches [71, 168, 147, 41, 190, 86]. Martin et al. [168, 147, 41] develop an integrated debugging plug-in for equation-based models created by Modelica [22]. They use GDB [5] to debug the generated C code, and then map the debugging results to the equation-based model element. Dotan et al. [71] develop a model-level debugger for IBM RSARTE. Similarly, Graf et al. [86] 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 [19] with model-to-code mapping data, and
create a model-level debugger for Stateflow [29] which support debugging models when the code is generated in C and C++.

Figure 3.1 summarizes the implementation of the existing approaches to model-level debugging at the target platform. They usually rely on code instrumentation to generate the meta-data required to keep a bi-directional mapping between code artifacts and model elements (states, transitions, etc.) [71, 187]. Binary code is then generated, which contains debugging symbols used by existing program debuggers to debug the application. An additional component at model-level interfaces with the program debugger to display the debugging information directly on the model, and to trigger debugging commands from the modeling tool. While
this solution works in practice, it suffers from several limitations, as follows.

- **Semantic mapping:** in addition to the bi-directional mapping between code artifacts and model elements (syntax mapping), the semantic gap between model and code elements makes translating debugging information to the model-level difficult. E.g., in the context of UML-RT, the following concrete example can be given: According to the UML-RT execution semantics, a capsule (i.e., component) instance executes in its logical thread, i.e., even if the capsule code is not assigned to a physical thread in the target language, the controller in the UML-RT run-time system will create the illusion to the user that the instance is executing concurrently; however, since the notion of logical threads is not supported by the program debugger, debugging capsule instances using program debuggers becomes unnecessarily complicated.

- **Integration overhead:** this limitation is caused by the need to integrate the model debugger with different program debuggers that are specific to a target language. For instance, at least three different integrations with program debuggers are required to support three different target languages (e.g., C++, Java). The definition of different mappings is furthermore impacted by the number of supported architectures (e.g., Intel x86, ARM). The resulting dependencies make the task of maintaining the comprehensive debugging capabilities at model-level complex and time-intensive for tool vendors.

- **Lack of reusability:** existing work is based on instrumenting the code instead of the modeling language, which introduces a dependency on the MDE
tool. For instance, UML-RT is supported by IBM RSA-RTE and Papyrus-RT. However, the need for instrumenting the code generator for debugging purposes would prevent a model-based debugger used in one tool from being easily reused in another tool. Consequently, the proposed solutions for model-level debugging are difficult to port from one MDE tool to another one.

- **User Experience:** finally, after all, efforts, the resulting model-based debuggers do not provide a proper debugging experience to the users, due to the variation between program debuggers.

Due to the issues mentioned above, we propose a platform-independent solution to model-level debugging that can overcome the existing limitations. The details of this approach are discussed in Chapter 5.

### 3.2 Execution and Debugging of Partial Models

In this section, we discuss the most relevant existing studies dealing with partial models and programs. Existing work can be divided into three categories: (1) work on partial models, which tries to address specification, analysis, and transformation of partial models, (2) work on partial programs that deals with the parsing, analysis, and completion of partial programs in the context of different programming languages, and (3) work on auto-completion of models.
3.2. EXECUTION AND DEBUGGING OF PARTIAL MODELS

3.2.1 Partial Models

In the context of MDD, the partial models are mainly used to deal with uncertainties of type known unknown. Existing research proposes mechanisms to define partial models using relaxed meta-models [166], model annotation [74], UML profiles [191], and graphical notations [77]. They leverage the partial models for analysis [74, 163], requirement management and analysis [153], testing [166, 192], and bi-directional transformation [72]. Also, some research addresses the refinement [154, 152], transformation [75] and completion [164] of partial models. E.g., the work in [74, 73] presents a rich formalism for partial models, which marks model elements with four special annotations (may, set, variable, and open) with well defined semantics. The authors show how the partial models can be concretized into possible design candidates. Sen et al. [166] present a semi-automated tool that supports the specification and completion of partial models, which are used for the testing of model transformation. They show that the testing of model transformation using partial models is as effective as using human-made models.

The necessity of partial model execution and debugging has been discussed in several studies [160, 129, 81]. However, to the best of our knowledge, there is no work addressing this problem specifically. This motivates us to propose a framework for the execution of partial models (ref. Chapter 6). Despite the work mentioned above, our work does not require specification of partial elements explicitly by users, since it detects all of them automatically by static analysis. Automatic detection of partial elements allows users to execute the models with minimum effort. Note that the partiality that our approach detects only concerns the execution, and may not be suitable for managing uncertainties of requirements or
3.2. EXECUTION AND DEBUGGING OF PARTIAL MODELS

design models.

3.2.2 Partial Programs

An extensive body of work exists for dealing with and leveraging partial programs. The most important of them can be classified as follows.

(1) Parsing of partial programs Typically existing compilers can handle only complete programs. As a partial program is a subset of a complete program, many of its variables’ types and library calls are unknown. Thus, the parsing partial program requires extra effort, mainly for the inference of missing types, and resolving unknown function calls. E.g., Zhong et al. [193] propose an approach that resolves unknown types and function calls for partial Java programs by analyzing the existing complete program versions. Melo et al. [136] present a technique to support the compilation of incomplete C code.

Koppler [111] presents a systematic approach to implementing fuzzy parsers, which extract high-level structures out of incomplete or syntactically incorrect programs. Moonen [138] proposes a solution in the form of island grammars that partitions code into islands (recognizable constructs of interest) and water (remaining parts). Dagenais et al. [67] propose a framework that uses heuristics to recover the declared type of expressions and resolve ambiguities in partial Java programs. Note that since the models are saved in the form of an abstract syntax tree (AST), the need for this type of research is unnecessary in the context of MDD.
(2) **Partial program analysis/verification to deal with poor scalability and missing components** E.g., modular model checking, introduced in [89], verifies properties of system modules, under some assumptions about the environment. Colby et al. [62] present an approach for automatically closing an open concurrent reactive system (i.e., a system some of whose components are missing) by generating an environment that can provide any input at any time to the system. The result is a self-executable system, which can exhibit all the possible reactive behaviours of the original system and therefore can be used for the state space exploration that is required for verification and analysis purposes.

(3) **Program synthesis techniques based on partial programs (synthesis by sketching).** Instead of synthesizing a program from scratch, work in this category uses a partial program (i.e., a program with holes) along with a specification, test harness, or reference implementation, and tries to fill the holes using synthesis techniques. E.g., Solar-Lezama et al. [170] introduce the concept of programming with sketches and present *Stream Bit* as a new programming approach based on sketching. Existing sketching techniques (e.g., [169]) translate the partial program into a propositional satisfiability problem, and leverage counter-example-guided inductive synthesis to generate a program using existing SAT solvers. Hua et al. [97] introduce *EdSketch* that performs execution-driven sketching for synthesizing Java programs using a backtracking depth-first search.
3.2. EXECUTION AND DEBUGGING OF PARTIAL MODELS

3.2.3 Auto-completion of Models

Model completion helps users during modeling activities through the suggestion or application of possible operations on the defined models. While the completion is only limited to the next few steps in modeling activities, and does not create a complete model, their approaches can be inspiring for our research. Mazanaek et al. [130, 131] study auto-completions for model and diagram editors, based on graph grammars. Their method calculates all possible completions based on the abstract syntax of modeling languages. Sen et al. [164] present a methodology for completing partial models. First, they use constraint logic programming to specify the required constraints on model properties, and then create a transformation from a partial model, its meta-model, and additional constraints to a constraint logic program. They use back-tracking as a means to resolve the resulting constraints and obtain possible value assignments for the undefined properties in the partial model. In another work [165], the authors perform transformation into an Alloy constraint model and use an SAT solver to generate possible completed models. Kuschke et al. [114, 113] propose an approach that analyzes editing operations and suggests the next operations to users based on a predefined but extensible catalog of common modeling activities for structural UML models. Janota et al. [103] present an interactive approach to assist users by providing hints over valid editing operations.

Further, several papers address fixing inconsistencies in models automatically [188, 178, 90, 171, 126, 151, 68]. For instance, Xiong et al. [188] propose a new language called Beanbag, similar to OCL. A Beanbag program includes two parts: (1)
define a relation over data, and (2) define a fixing procedure to maintain the relationship over data by automatically updating models.

3.3 Live Modeling

Live modeling is not well addressed in the context of MDD, and only some efforts have been made towards live modeling. Tendeloo et al. [183] propose a multi-paradigm approach to support the 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. Ulyana et al. [177] 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. Meanwhile, our approach is in the context of model execution by code generation, prevents changes that cause inconsistency, and allows users to do state transfer by replaying the execution. Rozen et al. [181] 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.

As suggested by [183], a possible solution for supporting live modeling in the context of translational execution is implementing the live modeling features on top of the target language of the source code being generated. This approach appears straightforward, but has several problems, more importantly:
3.3. LIVE MODELING

- **Edit latency.** The typical sequence of steps for reflecting changes to a model in its running execution consists of: (a) incremental code generation, (b) compile & built, and (c) applying the update to the execution (often by hot-swapping via replacing code, inter-positioning code, or dynamic linking [40, 80]). Even for a small model, these three steps together take more than half a second. Any delay of more than 500ms in this context is considered harmful to user experience, and can decrease developers’ productivity [132, 133]. Increasing model size can increase the delay and exacerbate the problem.

- **Dependency on the target language of the generated code.** Different programming languages provide different levels of support for live programming. So, the support for live modeling may be limited by the capabilities of the target language. E.g., many programming languages only support *fix-and-continue* [58] which allows only a limited set of code changes, excluding, e.g., state transfer [95]. 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 the restarting of the execution of the model for the effects of the change to become visible. Also, modeling tools often support code generation in several languages. In this case, differences in the target languages with respect to their support for live programming will make it difficult to provide support for live modeling that is uniform, consistent, and user-friendly.

- **Inconsistent runtime state.** As discussed in [177], 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, and may be challenging to implement.

To overcome the above-mentioned problems, we propose a conceptual framework that is completely independent of any live programming support offered by the target language, as discussed in Chapter 7.
In this chapter, we briefly discuss several case studies that we use throughout the thesis to evaluate our proposed solutions. Note that the case studies are developed by members (including me) of our laboratory [23] using Papyrus-RT [8] and their complexity ranges from simple to large models. Papyrus-RT [8] is built on top of Papyrus, provides modeling facilities based on UML-RT, and supports code generation in C++ from the UML-RT models. Also, it provides a run-time service library. Papyrus is an open source modeling tooling framework for UML, which offers advanced facilities for creating customized modeling tools. It has been customized for different UML profiles and purposes (e.g., SysML, MARTE, and information modeling) [24] and has recently attracted considerable attention in the modeling communities [106].
4.1 FailOver System

The FailOver system [51, 108] is an implementation of the fail-over mechanism. It consists of a set of servers processing client requests. To achieve high availability, the system supports two replication modes, passive and active [91]. In passive replication, one server component works as the master, handling all the client requests while backup servers are largely idle, except for handshake operations. Whenever a malfunction occurs, a backup server is ranked up as the new master. In active replication, client requests are load-balanced between several servers. In addition to processing client requests, each server has to update its status to inform other servers of its availability. Therefore, each server can be notified whenever a malfunction causes the failure of one of its peers.

The structure of FailOver is shown in Fig. 4.1 using UML-RT. It consists of one
4.1. FAILOVER SYSTEM

Figure 4.2: The root of Server capsule's USM

$ENV$, two servers, and five clients capsules. The $ENV$ capsule keeps track of the configuration information, such as the replication mode (active or passive replication) and the list of master servers. Also, it monitors possible failures of the servers and provides to any other capsule the current running state of the system.

Fig. 4.2 shows the root of the server capsule’s USM. A state Failure corresponds to the failure of the master server and is reached after a failure occurs. When the server recovers from a failure, it may restart as either a master (i.e., transitions to state RunAsMaster) or a backup server (i.e., steps to state Backup), depending on the configuration received from the $ENV$. The master server is required to update its state by sending two kinds of messages: IAmAlive (sent to the backup servers), and IAmMaster (sent to the environment capsule). If it fails in sending these messages in a specific period, its execution is considered to have failed, and
a new server must be ranked up. The master also processes and replies to client requests. In contrast, the backup server is mostly idle, waiting to be ranked up whenever the master server fails.

4.2 DigitalWatch System

The DigitalWatch is an implementation of a classical digital watch, which is described in [94]. In addition to showing the current time, the DigitalWatch provides stopwatch and alarm functionality. I designed this system as an assignment while teaching RTE system design at the Royal Military College of Canada [9]. Despite seeming simple, a proper solution for the DigitalWatch uses most of the UML-RT and RTE systems design concepts (e.g., concurrency, timing constraints, and time precision).
Figure 4.3 shows the structure of the DigitalWatch which consists of seven capsules. Capsule watchCore provides the core services and controls the watch. Capsule time tracks the progress of time. Capsule stopWatch and alarms provide the stopwatch and alarm functionalities, accordingly. Capsule display interfaces with the watch display, and shows the required text on the display. Capsule button-_reader reads user inputs and transfers them to the watchCore. dispGateKeeper works as the gatekeeper and protects the relevant capsules from a race for using the display.

4.3 Rover System

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

The design of Rover consists of three layers (see [39] for more details). The bottom layer is composed of UML artifacts that define the services for accessing the GPIOs. Read/write access mode can be set individually for each pin. The intermediate model layer consists of capsules and protocols to define how the application layer may access the different sensors and actuators. For example, capsule EngineController allows the application to cause the rover to move forward and backward and turn left and right. Finally, the upper layer is the application model, which defines what the rover is to do (e.g., drive to a certain location, or collect data while avoiding obstacles).
4.4. PARCEL ROUTER

Figure 4.4: The structure of ParcelRouter system

4.4 Parcel Router

The Parcel Router [174, 125] is an automatic system where tagged parcels are routed through successive chutes and switchers to a corresponding bin. The system is time-sensitive, and parcels can jam due to the variation of time spent by a parcel to transit through the different chutes.

The capsule structure of the Parcel Router is shown in Figure 4.4. It consists of a Gen capsule that generates parcels, and three stages responsible for conveying parcels to one of four destination bins. Each stage is further decomposed into chutes, switchers, and sensors (not shown in Fig. 4.4). We created two different versions of the same system. The complete version checks for parcel jams and prevents a parcel from being transferred from one chute to another one until it is empty. The simplified version ignores jams.
4.5 Simple Models

We also use simple academic models, including a Counter system that counts the elapsed seconds and a Car Door Central Lock system that simulates the opening and closing of car doors in a centralized way.
5.1 Introduction

Ideally, debuggers for MDD tools would allow users to ‘stay at the model-level’ and would not require them to refer to the generated source code or figure out how the code generator works. As discussed in Section 3.1, existing approaches to model-level debugging do not satisfy this requirement and are unnecessarily complex and platform-specific due to their dependency on program debuggers. In this chapter, we propose a novel approach for realizing model-level debugging that overcomes the limitations of existing by being completely platform-independent, i.e., it does not depend on any architecture-specific program debugger. Instead, our approach relies on model transformation to instrument the model to be debugged with information that allows it to support debugging activities. Since the
debugging support is realized on the model level, debuggable code can be generated from the model with the standard code generator and without the need for additional, code-level instrumentation or a program debugger.

We have implemented our approach in a model-based debugger called MDebugger [20, 45]. To maximize the impact of our work, our implementation only uses open source tools including the Papyrus-RT for modelling and code generation, and the Epsilon tools for model instrumentation. Just like Papyrus-RT, MDebugger is applicable to models expressed in UML for Real-Time (UML-RT) [159]. However, our approach is transferable to other modeling languages and tools.

The remainder of this chapter is organized as follows. The next section discusses the background materials specific to this chapter; Section 5.3 presents our approach; Section 5.4 describes our implementation and discussed MDebugger feature from the users point of view; Section 5.5 evaluates our approach using performance and code size metrics; Section 5.6 concludes.

5.2 Background

In this chapter, we classify the transitions of an USM based on whether or not their source or target states are pseudo-states. The rationale for this classification comes from the execution semantics of the USM that treats them differently (ref. Section 2.5.4). We distinguish the following four groups of transitions:

1. P2P: transitions in this group connect two pseudo-states. Assuming that there is not any guard associated with the transition, a P2P transition is taken as soon as its source state becomes active.

2. P2N: transitions in this group connect a pseudo-state to a non-pseudo state.
Similar to $P2P$ transitions, a $P2N$ transition is taken as soon as the source state becomes active.

3. $N2N$: transitions in this group connect two non-pseudo states. In contrast to $P2P$ and $P2N$ transitions, an $N2N$ transition is only taken when an event matching its trigger is raised by the RTS or another capsule and its guard holds. An $N2N$ transition executes, in this order, the exit code of its source state, the transition's effect code, and entry code of the target state.

4. $N2P$: transitions in this group connect a non-pseudo state with a pseudo-state. Similar to $N2N$ transitions, an $N2P$ transition is only taken when it has been triggered and its guard holds. An $N2P$ transition executes the exit code of its source state and then the transition's effect code.
**Example:** Figure 5.1 shows an annotated version of *TrafficLight* USM.

Also, we refer to a group transition (e.g., $t_{13}$ in Figure 5.1) as a one whose source is a composite state and a self transition as one whose source and target states are the same.

### 5.3 Approach

As stated before, existing approaches suffer from several limitations many of which result from the dependency on program debuggers specific to a particular language or architecture. For instance, in addition to having to generate the code for different configurations, existing model-level debuggers have to provide wrappers for each program debugger. Our approach relies on a model instrumenta-
tion process which is completely platform-independent, as shown in Figure 5.2. Differences with existing approaches (cf. Figure 3.1) are highlighted in the figure. The first difference is that our approach uses Model-to-Model (M2M) transformation techniques for creating an instrumented version of the user-defined model supporting debugging activities. This support is added via model transformation rules that are defined for each construct of the modelling language. This step is essential to our approach. It allows, without having to instrument the code, the generation of applications providing debugging services by themselves, i.e., without having to rely on a program debugger. The fine control provided by the model transformation techniques allows us to implement at model-level not only the communication between the debuggable system and the model debugger, but also advanced capabilities such as component introspection and attribute value change. Finally, as the generated code is debuggable, the compilation step does not require the addition of language- or architecture-specific debugging symbols into the binary files, reducing their size.

Compared to existing approaches, ours is more general, less dependent of the specifics of code generators or deployment configurations. As model instrumentation is done at model-level, the resulting debugger is easier to port from one tool to another. In addition, there is no gap between debug information provided at code- and model-level as the debugger does not rely on any program debuggers.

The remainder of this section describes the application of our approach to UML-RT. It also discusses to what extent our approach guarantees the preservation of behaviour.
5.3. APPROACH

5.3.1 Model Instrumentation

Model-level debuggers should provide a wide range of operations, such as attribute view and change and crash analysis. These operations can be built upon three composite operations including: (1) *stop and resume* operations for controlling the execution of the system via breakpoints, pause, resume, step in, and so on; (2) *view and change attribute* operations to inspect and modify the system status; (3) *detailed tracing* operations providing a foundation for crash analysis, event chain analysis, punctuality analysis, utilization analysis, and so on.

To formalise our approach, we first observe that instrumentation can be strictly bounded to *transition chains* as computation only occurs in three places: transition actions, state entry actions, and state exit actions. Indeed, the semantics of UML-RT state machines excludes any *do* activities. Therefore, we define a *transition chain* of a transition $t_1$ between two states $s_1$ and $s_2$ as the sequence of actions that is executed when the transition is fired. Since pseudo-states do not contain exit or entry actions, the actions in the transition chain of a transition thus depend on which one of the four groups defined at the end of the previous section the transition is in. Let $t_{P2P}$, $t_{P2N}$, $t_{N2P}$, and $t_{N2N}$ denote transitions with source state $s_1$ and target state $s_2$ in each of these four groups. Then, their transition chains are given by the following formulas in which $exit(up_s(\sigma, t_{N2x}))$ refers to a sequence of exit actions for $\sigma$ (i.e., current execution state) and a subset of its parents in bottom-up order from $\sigma$ to the state that $t_{N2x}$ (i.e., $t_{N2N}$ and $t_{N2P}$) originated from.
5.3. APPROACH

![Diagram showing Original and Refined Transition Chains](image)

Figure 5.3: Model instrumentation overview (an N2N transition)

(ref. Section 2.5.4 and Table 2.2 for details).

\[
\text{chain}(t_{P2P}) = \langle \text{action}(t_{P2P}) \rangle
\]

\[
\text{chain}(t_{P2N}) = \langle \text{action}(t_{P2N}), \text{entry}(s_2) \rangle
\]

\[
\text{chain}(t_{N2P}) = \langle \text{exit}(up_s(\sigma, t_{N2P})), \text{action}(t_{N2P}) \rangle
\]

\[
\text{chain}(t_{N2N}) = \langle \text{exit}(up_s(\sigma, t_{N2N})), \text{action}(t_{N2N}), \text{entry}(s_2) \rangle
\]

The execution of a capsule in UML-RT can be entirely represented by the composition of all transition chains that lead the system from one state to another. Consequently, by focusing on transition chains only, our approach supports the instrumentation of the entire model for debugging purposes.

The following shows the formalisation of different transformation rules we use. We formulate them at the model-level and then employ model query and transformation techniques to refine the user-defined model. Figure 5.3 shows how a N2N transition \(t_1\) between states \(s_1\) and \(s_2\) is instrumented in order to provide debugging support for the transition chain. The left side of the figure shows
the original transition $t_1$. The right side shows the refined transition after instrumentation in which the new elements are colored blue. The refined model introduces a choice point and a composite state a capsule may enter whenever a breakpoint is reached. We will now describe how the refined model allows the three different kinds of debugging operations mentioned above.

**Suspend and Resume Operations** Figure 5.3 shows how a capsule execution can be suspended and resumed. The presence of the choice point allows the application to check whether a breakpoint on the transition is set or not. If it is not set, the execution continues normally by executing the transition chain. Otherwise, it enters the composite debugging state and the transition chain is executed step-by-step. To do so, a mapping is created between the different actions composing the initial transition chain and the local transitions in the composite debugging state. It provides support for common stop and resume operations such as resume and step over. Stepping over the transitions $exit_{s_1}$, $action_{t_1}$, and $entry_{s_2}$ allows for executing the different actions (exit action of $s_1$, transition action of $t_1$, entry action of $s_2$) of the transition chain separately.

Initially, the execution goes into the debugging state $D_{s_1}$, where it is suspended before the exit action code of the original state $s_1$ is executed. In this state, the execution is pending, waiting for commands from the model debugger to resume its execution. When the command is received, the transition $exit_{s_1}$ is taken, causing the original exit action code to be run, and suspending the execution in state $D_{s_2}$. Another command results in taking the transition $action_{t_1}$ associated with the execution of the transition action of $t_1$ and in suspending the execution in $D_{s_3}$. A third command causes the execution of the entry action of the original
state $s2$ associated with the transition $entry_s2$. A last command is required to leave the composite state and resume the regular execution.

**Attribute View and Change Operations** In addition to providing support for controlling the execution of a capsule, the transformation rule also adds one self-transition $dbg_tl$. It provides a support for inspecting and changing the system status at run-time. Program debuggers such as GDB directly access the program stack and heap to provide these operations, which results in a tight coupling with the architecture. In contrast, our implementation relies on self-reflection techniques [124]. Therefore, when the execution is suspended, the model debugger can directly inspect and change the attributes of the suspended capsule. Note that certain languages, such as C++, provide limited, if any, support for self-reflection (in contrast with, e.g., Java [34]). However, this limit is overcome by our approach as the support for self-reflection is directly provided by the instrumented model.

**Detailed Tracing Operations** Having access to the history of the system execution during the debugging process is important for localizing bugs efficiently. For example, analysing execution traces can help to find the root cause of an unexpected behaviour. Execution traces can also provide information for other analysis activities which are crucial for RTE systems (e.g., Worst Case Execution Time analysis, Run-time verification). To support execution traces, we add trace-points to capture all possible events generated by the system. Similar to the work on schedulability analysis in [87, 36], we define an event class and event type so that the captured execution traces contain the necessary information. Figure 5.4 shows a taxonomy of generated events that is adapted from [87].
Figure 5.4: Taxonomy of events (adapted from [87]).

Coverage of all Transitions  To cover the transformation of all transition types (i.e., N2N, N2P, P2N, P2P) discussed in Section 5.2, we define a transformation rule per each transition type. The transformation rule illustrated in Figure 5.3 is applied to N2N transitions; since the chain of N2N transitions contains all possible action code four debugging substates and related transitions are required. Figure 5.5 shows the transformation rules for the other three transition types. In other words, the rules in Figs. 5.5a, 5.5b, and Figures 5.5c are applied to P2N, N2P, and P2P transitions respectively.
(a) Second rule: P2N

(b) Third rule: N2P

(c) Fourth rule: P2P

Figure 5.5: Alternative transformation rules
5.3. APPROACH

5.3.2 Behaviour Preservation

Behaviour preservation is an important concern to address when developing debuggers. While debuggers always impact the performance on a debugged application, the added overhead resulting from their use should be minimized as much as possible in order to prevent a significant change in the system behaviour. In this part, we discuss in which sense our approach to debugging is behaviour preserving.

Figure 5.6 shows how an execution trace is altered when using our approach. Top and bottom parts respectively show the execution trace of the transition $t_1$ before and after model instrumentation. Normal and debugging paths are respectively shown with solid (→) and dashed lines (→). In both cases, the transition chain is started when the event triggering the transition $t_1$ is fired. In the instrumented model, the capsule checks ($is\text{Debug}$) if the debugging mode is set. In that case, the execution follows the debugging path. As explained before, states $D_s1$ to $D_s4$ are states in which the capsule waits for debugging commands for resuming its execution or changing its state. Stepping over these states is done when receiving the $step$ command from the model debugger. Each step results in the execution of a part of the transition chain of $t_1$. After executing the entire transition chain, the execution is resumed. When the debugging mode is not active ($is\text{Debug}$ returns false), we can observe that the instrumented transition chain executes the same sequence of actions as the original. The $is\text{Debug}$ function is only used to check whether the system is in or should enter the debugging mode (e.g., after reaching a debugging breakpoint). It is read-only, very small (1 line of code) and executes very quickly. Unless the system is sensitive to very slight delays, the
execution of the isDebug function will not change the behaviour of the debugged system.

In debugging mode (isDebug returns true), the execution can be delayed or modified with the help of inspection and step commands. However, the order in which the steps in the transition chain are executed is preserved. Moreover, unless the user alters the system state (by, e.g., changing attribute values), the execution of the transformed transition chain will terminate in the exact same state as the original transition chain. Stepping operations from the model debugger do
not alter the system state, and are only used for advancing through the original transition chain.

In the context of concurrent executions, delays introduced by the debugger can have serious side-effects in the system behaviour and its preservation cannot be guaranteed. Indeed, while the execution of a debugged component is interrupted after, e.g., a breakpoint is reached, other components are still running and may send messages to it. It may result in a loss of messages or timeout errors, and thus makes the debugged and suspended component unable to process incoming messages.

While there is no universal solution in existing approaches, this issue is partially addressed in our approach. By relying on the `defer` and `recall` mechanism of UML-RT, our approach prevents messages from being lost. To do so, we modeled a transition `defer_t1` (cf. Figure 5.3) which captures all incoming messages sent by other capsules to recall them after the debugged capsule returns to its normal execution. This mechanism preserves the order of the received messages. Therefore, the transition `rec_t1` helps ensure that the behavior of the debugged capsule is preserved. However, it does not guarantee that the behavior is preserved for time-sensitive systems. Note that the debugging of timed distributed systems suffers from this problem in general.

## 5.4 Tool Support

This section describes our implementation of model-level debugging of UML-RT based on the proposed approach. We use Papyrus-RT as the primary tool for modelling UML-RT models and Epsilon [109] to implement the transformation rules
5.4. TOOL SUPPORT

Listing 5.1: Main loop of the transformations

```java
addGateWay();
refineStructure();
for (SM in allStateMachines){
    refineForSRO(SM);
    for (s in allStates){
        s.addTrace(traceType);
        s.guardCodes();
    }
}
```

required for instrumenting the models. This section is divided into four parts. The first part details the implementation of the transformation rules. The second part details the implementation of a model-level debugger called *MDebugger*\(^1\). The third part discusses the features of the *MDebugger* for debugging of UML-RT models. The last part mentions the limitations of the current implementation.

5.4.1 Model Instrumentation using Epsilon

Transformation rules used for instrumenting the models are implemented using the Epsilon Object Language (EOL) which supports a set of instructions to create, query, and modify models expressed in languages described with the Eclipse Modeling Framework (EMF) [107]. Besides, as an imperative modeling language, it provides support for programming constructs and the Object Constraint Language.

Listing 6.2 is the main loop of the transformations rules written in EOL. The *addGateway* function is responsible for enabling the model to interface with the debugger. It adds a UML-RT port to each capsule. These ports are typed with a specific protocol used for debugging purposes. The *refineStructure* function adds

---

\(^1\)MDebugger repository: [https://github.com/moji1/MDebugger](https://github.com/moji1/MDebugger)
required attributes and methods to each capsule to support debugging. An example of an added attribute is a map used for maintaining breakpoint information, required by the isDebug method during debugging. Examples of methods added are the isDebug method and a set of methods for supporting attribute view and change operations. The isDebug method returns a boolean value indicating whether a debugging session is opened or needs to be opened according to the current state and the next transition about to be taken. The generation of self-reflection methods was inspired by work on physics engine development [25]. These methods provide support for viewing and changing attribute values. To do so, the refineStructure function iterates over all attributes of each capsule and generates the corresponding helper functions, such as getters and setters. The refineForSRO function applies the four transformation rules explained in Section 5.3. For each transition chain, a helper function is used in order to determine which instrumentation rule needs to be applied, based on the kinds of source and target states. Based on the result, the refineForSRO calls the proper transformation rule. The addTrace function adds support for detailed tracing operations to each state. Finally, the guardCodes function adds a guard to every entry and exit code to prevent them from being executed when the capsule is being debugged.

5.4.2 Model-Based Debugging

After applying the instrumentation, code is generated from the instrumented model and an executable and debuggable binary is built from the code. Fig. 5.7 gives an overview of our current implementation. The debuggable binary consists of two
parts: the instrumented binary itself, and a debugging agent responsible for managing debugging sessions and for interfacing with the model debugger. The debugging agent interfaces on one side with the instrumented binary through messages exchanged using the Papyrus-RT RTS library, and on the other side with the model debugger through two queues: an event queue for sending detailed traces and a command queue for receiving debugging commands for, e.g., suspending the execution or viewing an attribute. Shared memory was favored as it provides efficient and fast asynchronous communication so that the model debugger can interface with the system without disrupting its execution.

Debugging Agent

The debugging agent plays a central role to provide effective debugging services. It was designed as a customized version of a more generic run-time monitoring architecture currently being developed [69, 139]. The debugging agent communicates with the system based on the execution semantics of UML-RT. This brings several benefits. E.g., it cannot make the system unschedulable or cause unpredictability or synchronization issues.

Initially, the debugging agent creates the event and command queues for communicating with the model debugger, then waits for every instrumented capsule instance to register. The registration phase is essential so the debugging agent can maintain a list of every capsule instance. It allows the model debugger to identify and send commands to a specific capsule instance to debug. After the registration phase, the debugging agent performs two tasks: 1) it receives and forwards trace
events from the registered capsule instances to the event queue, and 2) it periodically polls the command queue and processes any received commands. When a received command is valid, it is transmitted by the debugging agent to the target capsule instance.

**MDebugger**

MDebugger is a model-level debugger which provides debugging services by interacting with the debuggable binary. It consists of several components including a core component, a command interface, an event manager, and a communication layer. The core component implements the main logic of the debugger. Initially, it queries the list of registered capsule instances from the debugging agent. For every capsule instance, it keeps track of certain data including: 1) last events generated by the capsule, 2) the active state or transition, 3) the list of attributes, their type and value, 4) the execution mode (e.g., stepping or suspended), 5) the list of breakpoints set, and 6) the debugging command history related to the capsule instance.

In addition to the core component, the communication layer provides services for reading from and writing to the shared memory, and for handling TCP connections. The command interface is responsible for receiving and parsing the debugging commands from the command line interface or from external applications connected via TCP.
5.4. TOOL SUPPORT

Figure 5.7: MDebugger implementation overview

(a) MDebugger GUI
(b) Excerpts of MDebugger CLI

Figure 5.8: An overview of MDebugger features

5.4.3 MDebugger Features

Similar to program debuggers (e.g., GDB [5]), MDebugger supports debugging features such as setting breakpoints, listing and modifying variables. However, in contrast with program debuggers which usually work on the code-level and allow for debugging the execution at thread-level, MDebugger works on the model-level and allows for debugging the execution at the capsule-level, i.e., it allows for

2A demonstration video: https://youtu.be/L0JDn8czwQ
viewing, controlling, or modifying the execution of specific capsule instances. As shown in Figure 5.8 MDebugger provides a command line and a graphical user interface with the help of Eclipse Debugger Plug-in. In the following, we discuss features we have implemented to debug a system developed in UML-RT using our approach along with the limitations of alternative approaches. An Overview of the features is given in Figure 5.8. Graphical symbols (e.g., 1, 2) are provided to show to which parts of Figure 5.8a or Figure 5.8b the description refers to.

**View running capsules** Usually, viewing the current state of a program and its components at runtime is the preliminary step to start debugging. MDebugger provides the list command (1) that shows the list of running capsule instances, their current execution state that corresponds to the last executed action code, and their execution mode. Possible modes are suspended (i.e., the execution is suspended and waits for a debug command), run (i.e., the execution will run without any interruption), and continued (i.e., the execution will stop at the next breakpoint). Also, the current execution state is highlighted with red in the related state machines as shown in 5 and 6.

**Control execution commands.** The commands breakpoint, run, next, and continue allow for controlling the execution of capsule instances. Using the breakpoint (b) command, breakpoints can be set before or after action code. The run command executes the instance without interruption, disregarding of any breakpoints set. The next (n) command steps the execution of the capsule instance. The continue command executes until a breakpoint is reached. Intuitively, if no breakpoint is set, the two commands continue (c) and run (r) behave the same. These commands work at the capsule-level, e.g., a specific capsule instance can
be stopped while other capsules are running. Execution control in the GUI is managed in the main toolbar and breakpoints are set by selecting the related elements in the state machines. Also, a list of breakpoints is shown in the breakpoint view.

Due to the semantic gap, it is almost impossible to provide this feature at capsule level using program debuggers unless each capsule is assigned to a separate physical thread (which may change system behaviour and thus lead to other issues). However, using MDebugger each capsule execution flow can be controlled separately. Since this feature is essential for interactive debugging but cannot be supported easily with a program debugger, it illustrates why the use of a program debugger to develop a model-level debugger is problematic.

**View and change variables.** The *modify* \((m)\) and *watch* \((w)\) commands are used to view and change the values of the variables (attributes) of a capsule. *watch* shows the variable values based on the last execution trace even when the capsules are executed in *run* mode which is useful for runtime analysis activities. *modify* only works when the execution of a capsule is suspended. shows a typical scenario involving these commands. shows how these commands are supported in the GUI.

**Backtracing:** The *backtrace* \((bt)\) command can be used to ‘rewind’ the execution of a capsule by showing the list of the \(n\) last execution steps. shows sample output of the *backtrace* command where the traces are ordered based on their occurrence time. Each trace contains detailed information that allows the user to understand where and when specific attribute values are modified, which message caused a transition to be taken. In the GUI, MDebugger shows the last 5
5.5. EVALUATION

execution traces as shown in \( \dagger \).

**Integration with other tools** While MDebugger provides the foundational features for model-level debugging, many other activities and value-added services can be developed by using MDebugger services. Other applications can use the MDebugger services in three ways: (1) Execution traces can be saved in a textual format using the `save` command and analyzed in offline mode. (2) The core component also provides a `connect` command for connecting to external applications using *TCP* and publishing all execution traces in real-time. (3) Finally, external applications can interact with the debuggable program and use its services directly.

### 5.4.4 Exceptions and Limitations

Currently, instrumentation only applies to explicitly defined transitions. Since transitions from the history state to possible target states are implicitly implemented, the current approach does not provide support for suspending the execution before the entry action code of a state when it is reached from the history state. The same limitation exists for exit code on sub-states during a group transition. Also, our current implementation only supports value change of primitive attributes.

### 5.5 Evaluation

This section details the experiment we conducted in order to assess the applicability of our approach. In the following, we describe evaluation metrics, conducted experiment experiments and results.
5.5. EVALUATION

5.5.1 Evaluation Metrics

Metric 1 (Performance of the model instrumentation.) We use model transformations to create instrumented models that are required for the model-level debugging. The applied model transformations are the core of our approach, and their performance is a crucial metric for the practicality of our approach. Thus, this metric measures the time required for the transformations of models.

Metric 2 (Performance overhead.) Performance overhead is one of the main factors impacting the applicability of debugging tools. This factor is even more critical for UML-RT, which is used for the development of RTE systems where time sensitiveness and behaviour preservation are the main concerns during debugging. Thus, we measure and evaluate the performance overhead of our approach. Also, to position our work, we compare the performance overhead of our approach with the existing methods that use program debuggers.

Metric 3 (Size Overhead.) As discussed, the instrumentation adds certain elements to make them debuggable. These new elements increase the complexity of the models in terms of the number of components, states, transitions, and the generated code that result in a larger generated program size (binary). This metric first measures how the complexity of instrumented models changes in comparison with the original models. Also, existing model-level debugging techniques based on the program debugger require debugging symbols and mapping data that result in larger program size. Thus, we also measure how the application size is increased by existing work and compare it with our approach.
Table 5.1: Model complexity of use cases and instrumentation time

<table>
<thead>
<tr>
<th>Model</th>
<th>Model Complexity</th>
<th>Instrumented</th>
<th>Instr. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>C</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>Car Door Central Lock</td>
<td>4</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>Simplified Parcel Router</td>
<td>8</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>Parcel Router</td>
<td>8</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td>Rover</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>FailOver System</td>
<td>9</td>
<td>28</td>
<td>43</td>
</tr>
</tbody>
</table>

C: Capsule, S: State, T: Transition
Refer to Chapter 4 for a detailed description of use cases.

5.5.2 Experiments

In the following, we discuss the experiments used to calculate the metrics. To perform the experiment, we used a computer equipped with a 2.7 GHz CPU and 8GB of memory. All hardware and software configurations and workload of the system during the entire experiment were identical.

Measuring the model instrumentation time (EXP-1): To measure the instrumentation time, for each use case listed in Table 5.1, we ran the transformation 20 times and calculated the average transformation time. It would have been interesting to compare the time required for instrumenting models in our approach with the time required for instrumenting code in existing model-based approaches. Unfortunately, no data is available for establishing a comparison. Also, in order to assess the scalability of the instrumentation, we applied the model instrumentation process twice: first on the FailOver system model and then again on the instrumented version of the model, i.e., the result of the first instrumentation. While there is no need to apply the instrumentation on an already instrumented model, this experiment allowed us to check that the instrumentation time
does not skyrocket when the model size grows exponentially. Note that, although intensive efforts are being made to make Papyrus-RT an industrial-grade MDE tool, still no industrial model is available to assess the scalability of our approach.

**Measuring the performance overhead (EXP-2):** To measure the performance overhead, we set up a benchmark for evaluating the performance of the FailOver system under normal mode (to show the real performance of the system) and debugging mode using our approach. In all cases, we configured the system in the active replication mode with five clients and two servers, and we set up a simple scenario where each client checks the available servers, sends a message, and waits for a reply. Upon receiving the reply, the client processes the response and sends a new message after a specific time. The two servers process client requests and send replies. In addition, every five seconds, each server has to report its state to the second server, so other servers are notified whenever a failure affects a running server. We executed the system until 10,000 messages were sent by the clients and processed by the servers. Then, based on the system logs, we collected the computation time for replying to a server request (i.e., RequestReply transition), processing message response by the clients (i.e., ProcessingResponse transition), and notifying the availability of each server to its peers (i.e., SendKeepAlive transition). Besides, we also measured the overall execution time, from the first message sent to the last message received and processed to calculate the overall processing time of 10,000 messages.

**Measuring the size overhead (EXP-3):** To measure the size overhead, first, we ran the instrumentation on the use cases illustrated in Table 5.1 and recorded the
complexity of original models and their instrumented versions. Then, we generated for each use case, the code of both the original and the instrumented models. From the code of the original model, we created three binaries: one without debugging symbols (to get the real size of the system) and two with debugging symbols compiled with `-ggdb` and `-g3` flags that are required by GDB. Then, we created a third binary by compiling the code generated from the instrumented model. We compared the four binaries to compare the size overhead of GDB and our approach. This comparison allows us to compare our approach with general-purpose program debuggers, such as GDB. It does not take into consideration extra overhead caused by existing model-based approaches, e.g., to maintain a mapping between binary-, code-, and model-levels.

**Rationale for comparison with GDB.** When applicable, we compared MDebugger with GDB, one of the most widely used program debuggers. Despite being semantically different as they operate at different levels, the comparison is viable as existing model-based approaches rely on program debuggers, hence inherit the overhead induced by them, in addition to other possible overhead resulting from the mapping between code- and model-level. It may be argued that GDB provides more advanced features and debugging services at binary-level, making the comparison unfair. While this is true, it is worth mentioning that most of the services offered by GDB or other program debuggers are not beneficial at model-level, hence merely impede the performance of model-based debuggers.
5.5. EVALUATION

![Graphs showing processing time for different execution modes.](image)

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

Figure 5.9: Processing time of transitions of 10,000 messages

5.5.3 Results

**Instrumenting time:** Based on the result of EXP-1, the column *Instr. time* of Table 5.1 shows the average time required to transform each use case. It varies between 445 and 1,523 milliseconds, depending on the use case, which is within the range of a second. It is, therefore, safe to conclude that the performance of the instrumentation of our approach is reasonable. Note that instrumentation is required only once and the required time appears negligible w.r.t. the benefits offered by the instrumented model.

Also, the resulting model from instrumentation of the instrumented *Failover* model includes 2363 states and 3725 transitions and the instrumentation time is an average of 41 seconds which suggests that our approach is scalable enough for industrial-like models.

**Size overhead:** Table 5.1 shows the added model complexity of each model. Depending on the use case complexity, models grow proportionally to the number of transition chains to instrument. Therefore, the number of states is increased by 5 to 11 times and the number of transitions by 7 to 10 times. As for the structural
part of the models, one capsule is always added, which corresponds to the debugging agent capsule. The added model complexity impacts the generated code size, which is increased by 3 to 6 times, depending on the use case.

Both instrumented models and generated code are intermediate in our approach, and their size growth does not impact the user experience. However, they do impact the size of the binaries created from them. Figure 5.10 shows a comparison of the size of the binary files using our approach and GDB (compiled with -ggdb and -g3 flags). It shows that for each use case, the debuggable binary resulting from our approach is an average of 8 times as large as the original binary. We argue that this is reasonable and even comparable with program debuggers as the size overhead is within the range of values when compiling the code with debugging symbols (between 7.98 and 8.17 times as large as the original binary files).

Note that compared to traditional model-level debugging approaches, our approach does not add additional overhead other than the one caused by the model instrumentation process. Besides creating an overhead due to the integration of
debugging symbols into the binary files, existing methods cause an additional overhead to generate and maintain a mapping between the executed binaries and the models. As the description of existing approaches does not include the measurement of this overhead, we could not compare our approach with existing ones. We can argue that in addition to the overhead of debugging symbols, those methods need to use instrumentation or other techniques to manage the mapping between model level and code level. Generating and managing mapping data can have a significant size overhead so that their size overhead should be more significant than program debuggers and our approach.

**Performance overhead:** Figure 5.9 shows violin plots of computation times for the three transition chains. The box plots within the violin plots show the median of data. Computation times are recorded until 10,000 messages are processed. The wideness bars show the density of computation time in the specific range. As shown in Figure 5.9, for all three transitions, the system performance is impeded by the use of MDebugger. Using MDebugger, the majority of the \textit{ProcessResponse} messages are processed within 0.28 to 0.41ms, with an average time of 0.35ms and a median time of 0.33ms respectively, which is close to the processing time when the system is in normal mode (average and median times of 0.29 and 0.27ms respectively). The overhead for the \textit{ProcessResponse} transition is within the range of microseconds and therefore, negligible. The overhead is similar for \textit{RequestReply} and \textit{SendKeepAlive} messages. While the median and average of computation time for \textit{RequestReply} is 48.26ms and 48.45ms using our approach, respectively, the median and average in normal mode are 47.41 and 48.35ms. For the \textit{SendKeepAlive} transitions, the median, and average using our approach are 0.07ms and 0.08ms.
respectively and is 0.01ms for both in normal mode. In summary, we can argue that for each transition, the overhead of our approach is small and acceptable for many application domains.

Also, the total processing time of 10,000 messages is 507 seconds under normal mode and 519 seconds using our approach. It represents an added overhead of 2.31%. This overhead is quite low and is acceptable for many application domains. It might be worth mentioning that the added execution time for each transition appears to be constant, meaning that this result may vary depending on the complexity of the system to debug.

5.6 Summary

In this chapter, we proposed model-level debugging approach and formalised and implemented it in the context of UML-RT. Compared with existing approaches that rely on program debuggers to work, our approach relies on model instrumentation techniques. To do so, we formulated and applied the necessary instrumentation at the model-level, bringing debugging capabilities to the model itself. As a consequence, our approach does not require any additional instrumentation at code-level. It is therefore more generic and portable, meaning that the debugger can support a range of code generators, target languages, and HW platforms without change. Along with the approach, we implemented a model-level debugger called MDebugger. It supports most of the common debugging features, and can be used from the command line, or via a graphical debugging interface we developed in Eclipse. We also conducted an experiment to assess the applicability or
our approach. The experiment showed that the size overhead of the generated binary files is comparable to other approaches and the use of our implementation impacts the performance of the debugged system only slightly.
6.1 Introduction

Models of a system can be partial for several reasons, which can be classified in two groups, in the context of model execution. (1) **Un-intentional partiality:** due to the iterative and incremental nature of software development, typically a model of a system is not complete for the execution until certain stages of development are reached. Also, it is often helpful or even necessary to postpone certain design decisions and details until a later phase of development. Any missing specification (e.g., a trigger of a transition in a state machine) can lead to a situation in which the execution of the model cannot progress anymore or reach certain states. (2) **Intentional partiality:** typically, a software system has several components. For many reasons, such as efficiency, simplicity, and availability of
external components, developers may want to execute only part of the system (partially), and ignore the rest of it, even if it is specified (e.g., unit testing). Execution of partial models in this case can fail, due to the unresolved dependencies. A component rarely exists in isolation, and the execution of the component often cannot be carried out if its dependencies are not met.

Execution and debugging of partial models allows debugging of models at early stages of software development. Generally, early debugging of software systems allows early detection of defects, which can be fixed more easily and at lower cost [179, 52, 53]. In addition, debugging of partial models can facilitate unit testing, synthesis by sketching, and the partial analysis and debugging of models. As discussed in Section 3.2.2 there is a certain amount of work in dealing with partial models at design time, which allows for specification, analysis, verification, and transformation of partial models (e.g., [152, 74, 154, 76]). However, existing model execution approaches (interpretation of models or code generation) do not support the meaningful execution of partial models. By “meaningful” we mean that the system execution can progress and reach all relevant states, assuming proper inputs are provided.

A possible workaround to execute partial models is the simulation of missing components by techniques such as mocking [155]. These solutions are mainly designed for unit testing, and have several deficiencies when used for the debugging of models: (1) They are not fully automated, and developers still need to do extra work to create them. (2) Often, they are applicable only at the component-level and require a component to be simulated fully, while for debugging purposes,
developers may need to simulate only parts of a component. (3) More importantly, while, in the code-base development context, there are several mocking frameworks (e.g., Mockito, EasyMock, JMock, Opmock, etc.) that can be used to simulate components of a system [172], there is a lack of facilities, guidelines, and frameworks in the context of MDD to help to create mockers [173].

This work advances the state of the art in model-level debugging and utilizing partial models, by providing support for execution and debugging of partial models. We propose a conceptual framework for the execution of partial models, which consists of three steps: static analysis, automatic refinement, and input-driven execution. First, a static analysis that respects the execution semantics of models is applied, which detects problematic elements that prevent the execution from progressing or reaching certain states. Second, to make the partial models executable, model-to-model (M2M) transformations [105] are used to refine the models automatically by adding decision points where the elements are missing. Third, during the execution these decision points allow users to interactively (1) inspect and modify the system using debugging services, and (2) select one of the possible options to continue the execution. The interactive execution requires manual intervention, which can be tedious and time-consuming. To mitigate this problem, our approach includes a scripting language that captures user input as execution rules which can be applied automatically during execution, without stopping the execution and interaction with users.

We extend our previous work on model-level debugging (ref. Chapter 5) in the context of UML-RT (i.e., a language for modeling of soft real-time systems) [162],
and created an engine for the execution of partial model \((PMExec)\) that embodies the proposed framework. To maximize the impact of our work, our implementation is publicly available, and only uses open source tools, including the Papyrus-RT MDD tool, for modelling and code generation, and the Epsilon tools for model transformation. We evaluated \(PMDebugger\) based on several use-cases that show that the static analysis, refinement, and handling of user input performed with reasonable performance, and the overhead of approach, which is caused by increasing the complexity of models by the refinement, is manageable.

The rest of this chapter is organized as follows. In Section 6.2, we present a running example. Section 6.3 describes our proposed framework for the execution of partial models and its application to UML-RT. Section 6.4 demonstrates features of \((PMExec)\). We discuss our evaluation approach and results in Section 6.5. Finally, Section 6.6 summarizes the chapter.

### 6.2 A Running Example

We use a partial version of \(TrafficLight\) model as a running example. As shown in Figure 6.1, the behaviour (i.e., USM) of capsule \(UC\) is missing, the behaviour of capsule \(SLD\) is complete, and the behaviour of capsule \(CTR\) is incomplete which is shown in the bottom of the figure (no incoming and outgoing transition is defined to/from \(yellow\) and no trigger is defined for transition \(t_{13}\)).

Let us focus on two debug scenarios in the context of the example.
6.2. A RUNNING EXAMPLE

![TrafficLight HSM Diagram]

**Figure 6.1:** A partial version of TrafficLight model

**Scenario-1: (Execution of the example as is)** Existing MDD tools do not support the execution of the example out-of-box, due to two issues: (1) An appropriate stub for capsule UC needs to be created, which is a time-consuming task, (2) even if the stub is provided, the execution of CTR stops at state yellow, due to the missing specification, i.e., no outgoing transition is defined from state yellow. Currently, the only possible way to resolve the second issue is to postpone the debugging until the missing specifications are provided.

**Scenario-2: (Execution of the example by ignoring SLD, i.e., intentional partiality)** To support this scenario, in addition to replacing the UC with a stub, the existing behaviour of the SLD should be ignored and replaced by a proper stub. In this chapter, we will
Figure 6.2: A conceptual framework for executing partial models

present a systematic solution for automatically creating an executable version of
the partial models, which allows for their execution and supports both scenarios
above.

6.3 Approach

6.3.1 A Conceptual Framework

Figure 6.2 shows a conceptual framework for executing partial models, which con-
sists of three parts (Static Analysis, Automatic Refinement and Input-driven Exe-
cution). In the following, we discuss the configuration and the three parts.

Configuration. As we discussed in Section 5.1, models can be partial intention-
ally or unintentionally. In both cases, the completeness level of a model is often
decided by different stakeholders, based on constraints and goals. We do not have
any rule about when a model is complete. Instead, we allow users to specify the
completeness level of each component by providing a configuration. Supported
levels are $c\text{levels} = \{\text{complete, partial, absent/ignored}\}$. Complete components are
assumed to be complete, and are not required to be analyzed and refined. Partial components are assumed to be incomplete. Thus, their current specification (structure and behavior) is analyzed and refined. Absent/ignored components are assumed that have no behavior specification. However, their existence may be necessary for the execution of other (partial and complete) components, due to the dependency between them. Thus, the absent/ignored components are analyzed based on their structure (inputs and outputs) and possible dependencies of other components to them. Then they are given behaviour sufficient for simulation. The configuration allows the execution of the models for different scenarios. E.g., to execute the running example of the debugging scenarios that are discussed in Section 6.2, users can set the CTR to partial, SLD to complete, and UC to absent/ignored to execute the system for Scenario-1. For Scenario-2, CTR should be set to partial and the other capsules should be set to absent/ignored. Note that the appropriate choice of completeness levels is dependent on the debugging scenario and goal of the execution. E.g., it may be appropriate and useful to label a fully defined component as absent/ignored (e.g., Scenario-2).

Static Analysis. There are two groups of problems associated with executing partial models: lack of progress and lack of reachability. The former is related to situations where the execution cannot progress anymore from a certain point. The latter concerns the execution being unable to reach certain defined states. Static analysis is performed on the user-defined model. The analysis identifies (1) missing elements, (2) existing elements with problematic specifications, and (3) missing and un-handled inputs.

Automatic Refinement. During the refinement phase, depending on the results
of phase 1 and the configuration, the user-defined model is refined automatically using model-to-model transformation techniques. The goal of the refinement is to fix the problematic elements or modify them in such a way that users can provide more information about them during the execution. Depending on the modelling languages, certain language constructs can be used to enable models to interact with users during the execution. E.g., for USMs, we use choice-points with certain actions and guards.

The refinement should meet the following constraints. (1) Refined models must preserve the original behavior of the user-defined models. (2) The execution of the refined model must not get stuck, assuming the proper inputs are provided. (3) The execution of the refined model must be able to reach all defined states in a finite number of execution steps, assuming proper inputs are provided. In addition, it is desirable that the refined models be executable using the same method and tool used for the execution of the original models. Thus it is ensured that the refined models can be executed by out-of-box features provided by most MDD tools. Also, the implementation of the framework can be transferred to new tools easily.

**Input-driven Execution.** The refined model can be executed via interpretation or code generation. During the execution, the executed model provides an interface for reading user input either in interactive or batch mode. It is also crucial to provide debugging facilities. Thus, users can investigate the execution before providing inputs.

Generally, the above discussed framework can be applied in the context of different modeling languages, to allow execution of partial models. However, 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. In the rest of this section, we discuss the application of the framework for executing partial UML-RT models.

### 6.3.2 Execution of Partial UML-RT Models

#### Static Analysis

In the following, we discuss the details of the static analysis of UML-RT models, with respect to the problems of lack of progress and reachability.

**Lack of progress.** Based on the execution semantics of USMs (cf. Section 2.5.4), the execution of an USM can be stopped due to several issues, which can be divided into two groups, as follows.

- **Missing/problematic elements:** The execution of an USM cannot start, or moves to a stuck configuration, due to the following issues. *P1*: missing initial state, *P2*: childless composite states, *P3*: broken chain, *P4*: dead state, *P5*: unexpected messages, *P6*: non-exhaustive guards of choice-points. Except for *P6*, the elements with these issues can be queried from the structure of the USM, as follows.

\[
\begin{align*}
P1 & \leftarrow \{s_c \in (root(USM) \cup S_c) : S_{in} \cap child(s_c) = \emptyset\} \\
P2 & \leftarrow \{s_c \in (root(USM) \cup S_c) : child(s_c) = \emptyset\} \\
P3 & \leftarrow \{s \in S_p \setminus (S_h \cup S_{ch}) : out_t(s) = \emptyset\} \\
P4 & \leftarrow \{s \in S_b : handled(s) = \emptyset\} \\
P5 & \leftarrow \{s \in S_b : inp(c) \setminus handled(s) \neq \emptyset\}
\end{align*}
\]
6.3. APPROACH

Algorithm 1: Refinement of a Partial UML-RT Model

**Input**: A UML-RT model `sys` and a configuration `conf`
**Output**: A refined model

1. Add a debugging interface `dbg_int` and a debugging agent `dbg_agent` into `sys`
2. Add ports types with `timing` and `dbg_int` into `dbg_agent`
3. `forall` `c ∈ sys.C` do
   4. `switch` `c.conf` do
      5. **case** `partial` do
         6. Add port `p` of type `dbg_int` into component `c`
         7. Add a connection using port `p` with `dbg_agent`
         8. `c.β ← refineUSM(c.β, c)`
      9. **case** `absent/ignored` do
         10. Add port `p` of type `dbg_int` into component `c`
           11. Add a connection using port `p` with `dbg_agent`
           12. Delete all elements from USM of `c` (`c.β`)
           13. `c.β ← refineUSM(c.β, c)`
      14. **otherwise** do
         15. `dbg_agent.β ← refineUSM(dbg_agent.β, c)`

As for `P6`, we assume that all choice-points have this problem, (i.e., `P6 ← Sc`). This is an overestimation, and covers all possible situations of `P6`. Checking the exhaustiveness of guards of a choice-point during design time is a difficult problem, and requires expensive computation. Thus, fixing this problem at design-time can increase the refinement time significantly, and even make it unsolvable. Also, the applicability of the existing techniques on partial models is not supported by default, and requires extra work and research.

- **Missing inputs**: A prerequisite for taking an execution step from basic states is the reception of a new message (cf. Rule-4 of Fig. 2.3) that can enable an
outgoing transition from the current state. The execution can be stuck, if
the required messages for triggering possible transitions are not produced
by the connected components. This can happen for two reasons: (1) the con-
nected component lacks a behavior specification, i.e., components are con-
figured as absent/ignored (P7), (2) the behavior of a connected component
is partial (P8). Detecting P7 is trivial and can be queried from the interface
specification of components. Let I be a set of possible input messages of
all partial and complete components, and let O be a set of possible outputs
messages of absent/ignored components, then 
\[ P7 \leftarrow I \cap O. \]
Detecting P8 at design time suffers from a similar problem as P6. Thus, we
overestimate again and assume that all partial components have this prob-
lem, (i.e., \( P8 \leftarrow \) all partial components). As we will discuss later, we provide
debugging commands for sending messages to the other components from
partial components, during the execution. That way, users can fix this prob-
lem by manually injecting the related messages during the execution.

**Lack of Reachability.** We classify reasons for the reachability problem, as follows.

- **Lack of progress:** Anything causing the lack of progress problem also causes
  a lack of reachability. There is no way for the execution to reach any state
  after being stopped.

- **Missing/problematic elements:** The following issues cause a lack of reach-
  ability. \( P9 \): isolated states refer to states, except for initial and history states,
  which have no incoming transitions. There is no way for the execution to
  reach these states. \( P10 \): not-takeable transitions originate from a basic or
  composite state and have no trigger. \( P11 \): To allow reachability of all states
effectively, we define another condition that concerns steering the execution from configurations whose current state is a basic state to any configuration whose current state is any state except composite, choice-point, or initial states. Except for \textit{PII}, the elements with these issues can be queried from the structure of a component \( c \) with an \textit{USM} as follows.

\[
P9 \leftarrow \left\{ s \in S \setminus (S_{in} \cup S_h) : \text{in}_\text{trans}(s) = \emptyset \right\}
\]

\[
P10 \leftarrow \left\{ t \in T : \text{src}(t) \in S_b \cup S_c \land \text{trig}(t) = \emptyset \right\}
\]

As for \textit{PII}, this capability needs to be addressed in all basic stats, i.e., \( PII \leftarrow S_b \).

**Refinement of Partial UML-RT Models**

In the following, we discuss the details of the refinement, applied to fix the problematic elements (\textit{PI-PII}) extracted in the analysis phases.

**Main Loop of the Refinement.** Algorithm 1 shows the main loop of the refinement of a UML-RT model. It takes a UML-RT model and a configuration as inputs. A configuration is a set of tuples \( \langle c \in C, clevel \in clevels \rangle \), where \( c \) is a component and \textit{clevel} specifies the level of completeness of the component. The algorithm first adds a new component to the model, called \textit{dbg_agent} with an empty \textit{USM}, creates a debugging interface, and adds debugging and timing ports into \textit{dbg_agent}. \textit{dbg_agent} is responsible for receiving from external applications and transferring \textit{dbg} messages to \textit{partial} and \textit{absent/ignored} components. After setting up \textit{dbg_agent}, the algorithm tries to apply certain refinements based on the configuration of the components, as follows.
Algorithm 2: Refinement of USM (refineUSM)

Input: An USM $sm$ and a component $c$
Output: A refined USM

// The following loop refines states in order of their nesting level, with the least
deeply nested state (root(USM)) refined first.
\[\text{forall } s_c \in \text{root}(sm) \cup (sm.S \in S_c) \text{ do} \]
  \[\text{dec}_p \leftarrow \text{add_state}(s_c, S_{ch}) \] // Add decision point
  \[\text{if } s_c \in P2 \text{ then} \] // Fix childless composite state (P2)
  \[\text{state}_{p,h} \leftarrow \text{add_state}(s_c, S_b) \]
  \[\text{if } s_c \in P1 \text{ then} \] // Fix missing initial state (P2)
  \[\text{add_state}(s_c, S_{in}) \]
  \[\text{forall } s_p \in (\text{child}(s_c) \cap P3 \setminus \text{dec}_p) \text{ do} \] // Fix broken chain (P3)
    \[\text{add_trans}(s_p, \text{dec}_p) \]
  \[\text{forall } s_{ch} \in \text{child}(s_c) \setminus \text{dec}_p \text{ do} \] // Fix non-exhaustive guards for choice-points (P6)
    \[t_1 \leftarrow \text{add_trans}(s_{ch}, \text{dec}_p) \]
    \[t_1.\text{guard} \leftarrow \lnot \bigvee (\text{guard(out}_\text{trans}(s_{ch})) \]
  \[\text{forall } s_b \in \text{child}(s_c) \cap P11 \text{ do} \] // Fix unexpected messages (P5), dead states (P4), and step 1 of fix for P11
    \[\text{if } s_b \in (P4 \cup P5) \text{ then} \]
      \[t_2 \leftarrow \text{add_trans}(s_b, \text{dec}_p) \]
      \[t_2.\text{trig} \leftarrow \text{inp}(c) \setminus \text{handled}(s_b) \]
    \[\text{forall } s \in (\text{child}(s_c) \setminus (S_{in} \cup \text{dec}_p)) \text{ do} \] // Fix isolated states (P9) and step 2 of fix for P11
      \[t_3 \leftarrow \text{add_trans}(\text{dec}_p, s) \]
  \[\text{forall } t \in P10 \text{ do} \] // Fix not-takeable transition (P10)
    \[t.\text{src} = \text{dec}_p \]
  \[\text{forall } s_{cc} \in \text{child}(s_c) \cap S_c \text{ do} \] // Step 3 of fix for P11
    \[\text{ex}_p \leftarrow \text{ex}_p \cup \text{add_state}(s_{cc}, S_{ex}) \]
    \[\text{en}_p \leftarrow \text{en}_p \cup \text{add_state}(s_{cc}, S_{en}) \]
    \[\text{to}_\text{child} \leftarrow \text{add_trans}(\text{dec}_p, \text{en}_p \cap \text{child}(s_{cc})) \]
    \[\text{from}_\text{child} \leftarrow \text{add_trans}(\text{ex}_p \cap \text{child}(s_{cc}), \text{dec}_p) \]
  \[\text{if } \text{parent}(s_c) \neq \emptyset \text{ then} \] // Last step of fix for P11
    \[\text{to}_\text{parent} \leftarrow \text{add_trans}(\text{dec}_p, \text{ex}_p \cap \text{child}(s_{cc})) \]
    \[\text{from}_\text{parent} \leftarrow \text{add_trans}(\text{en}_p \cap \text{child}(s_{cc}), \text{dec}_p) \]
6.3. APPROACH

1. For partial components, it adds a debugging port into the components and creates a connection between them and \texttt{dbg\_agent}. This allows them to receive the \texttt{dbg} message during the execution, which is essential for fixing elements in \texttt{P7-P8}. Then it calls the \texttt{refineUSM} function, which applies the behavioral refinement to fix the issues (line\# 4-7).

2. The behavior of absent/ignored components is removed, which results in an empty \texttt{USM}. Then, their empty \texttt{USM} is refined, which results in an \texttt{USM} that can receive and send all possible input and output messages of the component (line\# 8-12).

3. Finally, the \texttt{USM} of \texttt{dbg\_agent} is also refined as an absent/ignored component that results in an \texttt{USM} capable of processing debugging commands (line\# 14).

**Behavioral Refinement.** Algorithm 2 presents function \texttt{refineUSM}, which refines the \texttt{USM} of a component with respect to elements in \texttt{P1-P11} except for elements in \texttt{P7-P8}. Before discussing the details, let us define \texttt{add\_state} \((s_c \in S_c, ty \in S) \rightarrow S\) as a function that adds a state of type \(ty\) inside the \(s_c\) and \texttt{add\_trans} \((src, trg \in S) \rightarrow T\) as a function that adds a transition from state \(src\) to state \(trg\). The algorithm iterates over all composite states, and the root of the \texttt{USM}, and refines them in 9 steps, as follows.

1. It creates a choice-point state called \texttt{dec\_p}. \texttt{dec\_p} is used as a decision point during the execution (line\# 2). When a specification is missing, we refine the \texttt{USM} so that the execution is directed to \texttt{dec\_p}.

2. Fix elements in \texttt{P2} which have no child, by adding a new basic state inside the related state (line\# 3-4). Note that the added basic state has issues \texttt{P4}, \texttt{P5}, and
3. Fix elements in P1 which miss initial states, by adding a new initial state inside the related state (line# 5-6). Note that the added initial state has issue P3, and requires the corresponding fixes.

4. The elements in P3 (Broken chain) are fixed by adding a transition from the problematic states to \(dec_p\) (line# 7-8). This ensures that the execution will move to \(dec_p\) instead of stopping at the problematic states, and thus users can steer the execution to other states from \(dec_p\).

5. Fix elements in P6 (Non-exhaustive guards) by adding a transition from each choice-point to \(dec_p\) so that its guard is set to the negation of the disjunction
of the outgoing transitions’ guards (line# 9-10). This ensures that the execution moves to the \textit{dec\_p} if none of the guards for outgoing transition holds, instead of stopping there. Arguably, this solution is much cheaper than the design time analysis to detect and fix this issue.

6. During this step, a new transition is added from each basic state to \textit{dec\_p} and its trigger is set to all un-handled messages in the state (line# 12-14). This not only fixes the elements in \textit{P4} (dead state) by adding a transition from them to \textit{dec\_p}, but also (1) allows all un-handled messages to be handled as the new transition's trigger (\textit{P5}), and (2) allows the steering of the execution from any basic state to \textit{dec\_p} which is the first step of the fix for elements in \textit{P11}.

7. A transition is added from \textit{dec\_p} to all basic states and isolated states (line#15-16). This fixes issue \textit{P9} (isolated states), and also allows the steering of the execution to any basic state from \textit{dec\_p} which is the second part of the fix for \textit{P11}.

8. To fix non-takeable transitions (\textit{P9}), their source is changed to \textit{dec\_p} (line# 17-18). This allows them to be taken whenever the execution reaches \textit{dec\_p}. Since each state has a transition to \textit{dec\_p} which is added in step 6 with a trigger set to all un-handled messages, the non-takable transitions can be activated by any of the un-handled messages. Note that the \textit{dbg} message, which is added using Algorithm 4 is assumed to be an un-handled message.

9. At the end, an exit-point and entry-point states are added to each composite state to allow the execution to be steered from their sub-states to states in their parent state and vice-versa, which is the last part of the fix for elements in \textit{P10}. 
Two transitions to\_parent and from\_parent allow the execution to be steered from their sub-states to states in their parent state, and transitions to\_child and from\_child allow the execution to be steered from states in their parent state to their sub-states (line# 19-26).

Note that Algorithms 1 and 2 and do not show details for the actions which are added to USMs. E.g., (1) added actions to the USM of dbg\_agent for processing and injecting the message dbg, (2) guards of outgoing transitions from decision points which are set in a way that allows users to select one of them, (3) actions of incoming transitions to decision points that call a function to read user input. Interested readers can refer to the source code of the refinement [26].

**Refinement Result on the Running Example**

Figure 6.3 shows the result of Algorithm 2 on the partial CTR USM with partial configuration in which transitions and states are annotated with corresponding issues P1-P11. Let us review some examples of how the execution can be performed with missing specifications: (1) No transition from yellow to red was possible in the original model. This is fixed by adding transitions from yellow/red to dec\_p and vice-versa. Thus, any of the input messages or dbg messages can move the execution to state dec\_p from state yellow where users can select one of the outgoing transitions (e.g., the transition from dec\_p to state red). (2) the transitions t\_13 is un-takeable in the original model and its action cannot be executed. In the refined model a off or dbg message can move the execution to dec\_p in which the transition t\_13 is one of the possible transitions can be selected and its action be executed.
Execution of Refined Partial UML-RT Models

In this section, we briefly discuss our method for the execution and debugging of incomplete UML-RT models without dwelling on technical and implementation detail. Interested readers can refer to the project repository [26].

As discussed the partial models are refined by adding decision points where elements are missing/partial that allow users to execute the partial models and provide information about the missing/partial element during the execution. This requires a mechanism that (1) enables executed models to obtain user input either in interactive or batch mode, (2) provides debugging features to investigate and modify the execution of the model.

Figure 6.4 shows the execution flow of a refined USM in which a debugging probe is hooked into the execution of an USM by adding relevant actions in the initial transition of the USM. When an USM starts executing, two threads main and probe are started but only one of them is active at each time of the execution. Thread main executes the models as specified until it reaches a decision point where it sends the execution context to the probe and waits for the user input. The execution context contains relevant information concerning the execution of the USM such as current state of the execution and a list of possible options to continue the execution. Thread probe starts a debugging session (batch or interactive) that allows users to investigate the execution and provide input. At the end of the session, the user input is returned to thread main to continue the execution. The interaction between the threads is simply a function call from thread main to the probe that is implemented using an action in the transitions ending at the decision points. In the following, we review the features that are supported
Listing 1 High-level Grammar of the Debugging Services

```
grammar ExecRules;
script: execRule+;
execRule: 'rule' rID=ID 'where' '(' where ')' when?
'{' body=scriptStatement* '}'
when: 'when' '(' 'guard' ')';
where: State | Component;
statements: scriptStatement | interactiveStatement;
scriptStatement: simpleStatement | complexStatement;
interactiveStatement: (dbgCommands|umlrCmd) EOF;
random: 'random';
dbgCommands: viewCmd | selectCmd | simpleStatement | visited | controlCMD;
```

Listing 6.1 Main loop of the transformations

![Execution of a Refined HSM](image)

Figure 6.4: Execution flow of a refined **USM**

by the debugging probe.

(I) **Interactive execution.** If the execution mode is interactive, users are allowed to issue debugging commands listed in **interactiveStatement** of Listing 1, e.g., view
and modify variables. Most of debugging services are ported from MDebugger (ref Chapter 5). The new debugging commands to facilitate the execution of incomplete models are: (a) `viewCmd` that lists the possible options to continue the execution, (b) `selectCmd` which allows users to select one of the possible options, (c) `simpleStatement` which allows users to define new variables which can be accessed during the debugging session, access all attributes (i.e., USM’s variables and newly defined variables during the debugging session) and modify them using arithmetic expressions, and (d) `umlrtCmd` consists of `send` and `reply` commands which allow users to `send/reply` (inject) messages to other components. In addition, we support `random` parameter to generate random message payload (only integer) or options depending on the context of the debugging commands.

(2) **Batch execution.** Interactive execution stops and delays the execution which is not suitable for some situations especially for testing or debugging of time-sensitive systems. For that, a batch execution mode is supported that allows users to provide inputs using a script that consists of execution rules. As shown in `execRule` of Listing 1, an execution rule consist of a header and a body. The header consists of three parts: rule name, `where`, and `when`. The `where` statement is specified using component and state name and specifies on which elements the rule can be applied. Component/state names can be specified using ‘*’ to cover all components/states. The `when` specifies a guard of the rule. A rule is applied when the current execution context matches the `where` and the `when` is evaluated to true. It is worth mentioning that the execution rules specification method is inspired by the way the transformation rules are specified in rule-based model
transformation languages such as ETL [109]. The body of a rule is a sequence of debugging commands which are executed when the rule is applied.

Depending on the current execution context and defined rules, multiple rules may be applicable at each time, but only one rule can be applied at each time. The rule selection is performed according to the following conditions for decision points of an execution context with component $c$ and the current state $s$.

1. Guards of rules whose *where* (i.e., component and state name) is equal to $c$ and $s$ are evaluated based on the order of their appearance in the rules’ script file. The first rule whose guard holds is selected and applied.

2. If (1) is unsuccessful, guards of rules whose state name is exactly equal to $s$ and component name is equal to $\ast$ or empty are evaluated based on the order of their appearance in the rules’ script file. The first rule whose guard holds is selected and applied.

3. If (2) is unsuccessful, guards of rules whose component name is exactly equal to $c$ and state name is equal to $\ast$ or empty are evaluated based on the order of their appearance in the rules’ script file. The first rule whose guard holds is selected and applied.

4. If (3) is unsuccessful, guards of rules whose state and component name is $\ast$ are evaluated based on the order of their appearance in the rules’ script file. The first rule whose guard holds is selected and applied.

5. If none of the above holds, the execution stops and users are asked to provide input to continue the execution.
6.4 Tool Support

We have developed PMExec\textsuperscript{1} that embodies our approach and supports execution and debugging of partial UML-RT models. We used the Epsilon Object Language (EOL) \textsuperscript{109} to implement the transformation rules required for refining the models into executable models. EOL supports a set of instructions to create, query, and modify models. The part for the execution of the refined models (debugging probe) is implemented using C++, ANTLR \textsuperscript{142}, and the Boost C++ Library \textsuperscript{3}. Debugging features such as viewing and modifying variables are ported from MDebugger (ref. Chapter 5).

6.4.1 PMExec Features

In the following, we discuss the features of PMExec\textsuperscript{2} from the user point of view. When it is possible, the use of features is explained using the running example.

**Setup and run:** The PMExec is integrated into Papyrus-RT as an Eclipse plug-in and can be downloaded and installed from the PMExec Repository. After installation, it can be used to run partial UML-RT models simply by defining a run configuration (i.e., Eclipse run configuration) inside Papyrus-RT. The static analysis, transformation, code generation and build run automatically in the background without distracting end user. Upon successful execution, PMExec loads a UI as shown in Figure 6.5 as soon as the execution requires user input to continue the execution. The UI is split in two parts, a USM view and a DBG console. In the USM view the user can see the USM of the capsule where the current execution state is

\textsuperscript{1}https://moji1@bitbucket.org/moji1/partialmodels.git
\textsuperscript{2}A demonstration video: https://youtu.be/BRssselcMnc
highlighted. In the DBG console the user can interactively issue commands to investigate and fix the execution problems. The following commands are available: **View/select options** which lists the possible options to fix/continue the execution. E.g., the output of *view options* for the CTR when its execution is stuck in state *yellow* is shown in part 2 of Figure 6.5. Using the console, the execution can now be steered to any of the defined states inside the USM. Command *select* allows users to select one of the possible options, e.g., ‘*select state red*’ steers the execution to state *red*. 
6.4. TOOL SUPPORT

**Simple Expressions:** Similar to scripting languages (e.g., Python interactive console), *PMExec* allows the user to issue simple expressions (e.g., arithmetic expressions, define variables, and change/view the variables). This allows the user to investigate/modify the execution before deciding how to advance the execution. E.g., ‘x=5+1’ creates a new variable \( x \) and sets its value to 6. Defined variables can help the user record certain properties of the execution and define complex debugging and testing scenarios. Once defined, they can be used till the end of the execution.

**Communication commands** To allow the user to fix issues concerning the missing inputs (*P7*), three communication commands are provided: *inject*, *send*, and *reply*. Command *inject* sends a signal to a capsule to start a debugging session, command *send* sends messages on behalf of the capsule being debugged to the connected capsules, and command *reply* sends an incoming message back on the same channel it has been received. E.g., in the context of running example, no behavior is defined for the *UC*. Thus, the execution of the *CTR* will get stuck in state *off* and the overall execution of the system will be dead locked. The user can fix the problem by using the following communication command (1) ‘inject UC’ to start a debugging session with capsule *UC*. Note that the refinement fixes the behavior of capsule even with no defined behavior. (2) ‘send message on’ to send message *on* to the *CTR* where it will trigger a transition to turn on the *red* light.

**Random Execution:** Support for random testing is available through, e.g., the generation of a random message payload (only for integer types) or the injection of a randomly chosen input message.
Listing 2 An example script of the execution rules in the context of the TrafficLight

```plaintext
rule r1 where state yellow when receipt(timeout) {
    select state red
}
rule r2 where component * {
    reply random
    select state random
}
```

Listing 6.2 Main loop of the transformations

**Batch execution:** Interactive execution stops and delays the execution which is not suitable for some situations especially for testing or debugging of time-sensitive systems. PMExec supports a batch execution mode which allows users to provide inputs using a script of execution rules. The Listing 2 is an example of an execution script in the context of the running example. The rule r1 steers the execution to state red when a message timeout is received while in state yellow. The rule r2 replies to any received message using a random message and then moves the execution to a random state. The rules with header ‘*’ are only selected when no other rule matches in the current execution context. Note that having only one rule similar to r2 is enough for the random execution of any partial model using PMExec.

6.5 Evaluation

This section explains the validation of our approach which consists of two parts: formal validation and empirical evaluation. Formal validation is concerned with the properties of the refinement approach and shows formally how the applied refinement does not change the original specification of models but fixes the problems of lack of reachability and progress. Empirical evaluation is concerned with the applicability of the approach in practice. It applies our approach to execute
several partial UML-RT models in different scenarios and evaluates the performance and overhead.

6.5.1 Formal Validation

We use $\mathcal{R}^{USM}$ to refer to the result of applying Algorithm 2 on an $USM$ and call it the $RefinedUSM$ of the $USM$. In the following, first, we define the simulation relationship between $LTS$s, and then discuss the properties of $\mathcal{R}^{USM}$.

Behavioural Preservation

Definition 6. (Simulation Relation) Let $L_1 = \langle \Gamma_1, \mathcal{A}_1, \gamma_{10}, Q_1, \rightarrow_1 \rangle$ and $L_2 = \langle \Gamma_2, \mathcal{A}_2, \gamma_{20}, Q_2, \rightarrow_2 \rangle$ refer to LTSs of two USM, $USM_1$ and $USM_2$ respectively (detail of the LTS is discussed in Sec. 2.5.4). We write $L_1 \preceq L_2$ and say $L_2$ simulates $L_1$ if there is a binary relation $R \in \Gamma_1 \times \Gamma_2$ with the following two properties.

Start property: $\gamma_{10} \neq \emptyset \implies (\gamma_{10}, \gamma_{20}) \in R$. Note that, in the execution of USMs only one initial state is allowed.

Step property: Let $(\gamma_1, \in \Gamma_1, \gamma_2 \in \Gamma_2) \in R$.

For all $\gamma'_1 \in \Gamma_1$ and $a_i$, whenever $\gamma_1 \xrightarrow{a_1 \cdots a_n} \gamma'_1$ then there exists $\gamma'_2 \in \Gamma_2$ such that $\gamma_2 \xrightarrow{a_1 \cdots a_n} \gamma'_2 \land (\gamma'_1, \gamma'_2) \in R$.

The step property implies that when $(\gamma_1, \gamma_2) \in R$, any execution step started from $\gamma_1$ can be matched by an execution step started from $\gamma_2$ such that they both execute same actions and reach configurations that are in relation $R$.

Simulation implies trace containment, i.e., every sequence of actions that is
possible by the simulated LTS, is also possible by the simulating LTS [123] (i.e., the simulating LTS preserves the specification of the simulated LTS).

**Definition 7.** Let LTSs $L_o = \langle \Gamma_o, \mathcal{A}_o, \gamma_{o0}, Q_o, \rightarrow_o \rangle$ and $L_r = \langle \Gamma_r, \mathcal{A}_r, \gamma_{r0}, Q_r, \rightarrow_r \rangle$ represent the execution semantics of USM and $\mathbb{R}^{USM}$ respectively, $\mathbb{R}^{E}$ refers to a mapping from newly introduced variables during the refinement to their values, and $R \in \Gamma_o \times \Gamma_r$ is a binary relation defined as follows.

\[
R = \{ (\gamma_o \in \Gamma_o, \gamma_r \in \Gamma_r) | \gamma_o.\sigma = \gamma_r.\sigma \land \gamma_o.\mathcal{E} = \gamma_r.\mathcal{E} \setminus \mathbb{R}^{E} \land \gamma_o.\mathcal{H} = \gamma_r.\mathcal{H} \}
\]

**Proposition 1.** Assuming that $L_o$ and $L_r$ receive the same sequence of messages ($Q_r = Q_o$) and users do not issue any debugging commands during the execution of $\mathbb{R}^{USM}$, the relation $R$ (defined above) is a simulation relation, i.e., execution of an $\mathbb{R}^{USM}$ simulates the execution of USM ($L_o \leq L_r$).

**Lemma 6.5.1.** For message $\mu$ and basic state $s$, if function $\text{next}_t(s, \mu)$ (Ref. Table 2.2) returns transition $t$ in the context of USM, then it returns the same transition ($t$) in the context of $\mathbb{R}^{USM}$.

**Proof.** (Lemma 6.5.1) According to Algorithm 2, the refinement applies the following changes to the basic states. (I) Add a transition from a basic state to $\text{dec}_p$. The trigger of this transition is set to $\text{in}(c) \setminus \text{handled}(s_b)$ in order to not affect the existing transitions. This ensures that if a transition of USM can be triggered by message $\mu$, it still can be triggered by the same message in $\mathbb{R}^{USM}$ and $\text{next}_t$ in both case returns the same transition. (II) The source of not-takeable transitions is changed to $\text{dec}_p$. $\text{next}_t$ never returns a not-takeable transition, thus this change does not affect function $\text{next}_t$. (III) A transition is added from $\text{dec}_p$ to isolated
6.5. EVALUATION

states. Function \( \text{next}_t \) never returns an incoming transition to a state as the result. Thus, this change does not affect \( \text{next}_t \) either. Based on (I), (II), and (III) the proof of this lemma is complete. □

**Lemma 6.5.2.** For basic state \( s \), if function \( \text{dead}(s) \) (Ref. Table 2.2) returns \( \text{false} \) in the context of \( \text{USM} \), then it also returns \( \text{false} \) for state \( s \) in the context of \( \mathbb{R}^{\text{USM}} \).

*Proof.* (Lemma 6.5.2) No state or takeable transition is removed by the refinement. Thus if state \( s \) or one of its ancestors (\( \text{parents}(s) \)) has a takeable transition (\( t \)) that prevents \( s \) from being dead, the same transition is also exists in \( \mathbb{R}^{\text{USM}} \). By that the proof of this lemma is complete. □

*Proof.* (Proposition 1) To prove that \( \mathcal{R} \) is a simulation relationship, first, we need to show that the start property holds which includes the following two cases.

- The initial state of \( \text{USM} \) is missing. This case in trivial, since without initial state, the execution of \( \text{USM} \) cannot start (c.f. Def. 2.5.4) and \( \gamma_0 = \emptyset \). Thus the start property holds for this case.

- \( \text{USM} \) contains the initial state (i.e., \( \gamma_0 \neq \emptyset \)). In this case we need show that \( (\gamma_0, \gamma_0) \in \mathcal{R} \) where \( \gamma_0 \) is the initial configuration of \( \mathbb{R}^{\text{USM}} \). (I) According to (line \# 5-6 of Algorithm 2) when the original \( \text{USM} \) contains the initial state \( \text{in}_0 \), the refinement keeps the same initial state in the refined \( \text{USM} \) (i.e., the initial states of \( \mathbb{R}^{\text{USM}} \) and \( \text{USM} \) are equal to \( \text{in}_0 \)). According to the execution semantics of \( \text{USM} \) (c.f. Def. 2.5.4), the execution of \( \text{USM} \) starts from the initial configuration where its current state is set to the initial state of the \( \text{USM} \). Thus, there is a initial configuration of \( \gamma_0 \in \Gamma_r \), where its current state is equal to \( \text{in}_0 \) i.e., \( (\gamma_0, \sigma) = (\gamma_0, \sigma) \). (II) \( \gamma_0, \mathcal{H} = \gamma_0, \mathcal{H} \) because the history
is set to empty for the initial configuration (c.f. Def. 2.5.4). (III) Similarly, \(\gamma_{o_0}.E = \gamma_{r_0}.E \setminus R^E\), because the initial values of variables are set to default values and the refinement does not remove any existing variable. Based on (I), (II), and (III) we can conclude that \((\gamma_{o_0}, \gamma_{r_0}) \in R\) and by that the start property of simulation holds for this case as well.

Second, we have to show that the step property holds for any \((\gamma_o \in \Gamma_o, \gamma_r \in \Gamma_r) \in R\) which includes two main cases according to the execution semantics of USM (c.f. Def. 2.5.4).

- \(\gamma_o\) is a stuck configuration, i.e., no execution step can originate from it. Thus, the step property holds for this case.

- \(\gamma_o\) is a not a stuck configuration. Based on the execution rules (c.f. Def. 2.5.4), this case includes 4 sub-cases: (1) the current state \((\gamma_o.\sigma)\) is a pseudo-state of kind initial, entry-point, exit-point, or junction-point (Rule-1), (2) the current state is a basic state (Rule-4), (3) the current state is a history state (Rule-6), and (4) the current state is a choice-point (Rule-8). Proof of all sub-cases is similar and here we only prove sub-case (2).

Let us assume that \(\gamma_o.\sigma \in S_b\), \((\gamma_o \in \Gamma_o, \gamma_r \in \Gamma_r) \in R\) and an execution step \(s_{t_o} = (\gamma_o \rightarrow_o \gamma'_o)\) is taken. First, we have to show that an execution step \(s_{t_r} = (\gamma_r \rightarrow_r \gamma'_r)\) can be started from \(\gamma_r\) that executes the same actions as \(s_{t_o}\).

To prove the existence of \(s_{t_r}\), we have to show that the following condition holds (c.f. Rule-4, Def. 2.5.4).

\[
Q_r \neq \emptyset \land \gamma_r.\sigma \in S_b \land \neg\text{dead}(\gamma_r.\sigma) \land \\
\exists t \in T \mid t = next_t(\gamma_r.\sigma, head(Q_r))
\]
6.5. EVALUATION

(I) Definition of \( R \) and assumption \( \gamma_o.\sigma \in S_b \) implies that \( \gamma_r.\sigma \in S_b \). (II) \( Q_o \neq \emptyset \) because the execution step \( st_o \) is not possible with an empty queue (c.f. Rule-4 2.5.4). \( Q_o = Q_r \) based on the assumption of the proposition. Thus \( Q_r \neq \emptyset \). (III) \( \lnot \text{dead}(\gamma_o.\sigma) \) holds, because without that execution step \( st_o \) is not possible. Thus, based on Lemma 6.5.2 \( \lnot \text{dead}(\gamma_r.\sigma) \) holds. (IV) First, \( \exists t \in T \mid t = \text{next}_t(\gamma_o.\sigma, \text{head}(Q_o)) \) holds, otherwise the execution step \( st_o \) is not possible. Second, \( Q_r = Q_o \) implies that \( \text{head}(Q_r) = \text{head}(Q_o) \). Third, \( \gamma_o.\sigma = \gamma_r.\sigma \). Based on above mentioned statements and Lemma 6.5.1 we can conclude the last part of above formula \( (\exists t \in T \mid t = \text{next}_t(\gamma_r.\sigma, \text{head}(Q_r))) \) holds. Based on I, II, III, and IV we conclude that execution step \( st \) exists.

Next, we have to show that \( st_r \) and \( st_o \) execute the same sequence of actions. According to Rule-4 (c.f. 2.5.4), definition of relation \( R \), and Lemma 6.5.1, both \( st_r \) and \( st_o \) execute the same actions, i.e, \( \text{exit}(t.src), \text{exit}(u_p_s(s, t)), \text{act}(t), \text{entry}(t.des) \), where \( s = \gamma_o.\sigma = \gamma_r.\sigma, \mu = \text{head}(Q_o) = \text{head}(Q_r) \) and \( t = \text{next}_t(s, \mu) \).

Finally we have to show that \( (\gamma'_o, \gamma'_r) \in R \). (I) We have already shown that \( \text{next}_t \) returns the same result for both current states \( \gamma_o.\sigma \) and \( \gamma_r.\sigma \). Thus \( \gamma'_o.\sigma = \gamma'_r.\sigma \) with \( t.des \) as active state where \( t \) is the result of \( \text{next}_t \) (Rule-4). (II) Similarly \( \gamma'_o.H = \gamma'_r.H \) which are set to \( u.h(t.des, H) \). \( H \) refers to history of \( \gamma_o \) and \( \gamma_r \) which are equal (Def. 7).

(III) Variables are only changed by the execution of actions. Since the same sequence of actions is executed by \( st_0 \) and \( st_r \), and \( \gamma_o.E = \gamma_r.E \setminus R.E \), we have \( \gamma'_o.E = \gamma'_r.E \setminus R.E \). Based on (I), (II), and (III) we can conclude that \( (\gamma_o \in \Gamma_o, \gamma'_r \in \Gamma_r) \in R \). Thus, the proof for sub-case (2) is complete.
Assuming that the other sub-cases can be proven similarly, we conclude that the relation $R \in \Gamma_o \times \Gamma_r$ is a simulation relation and by that execution of $R^{USM}$ simulates the execution of $USM$. □

While $R^{USM}$ preserves the behavior of the original $USM$, it also never gets stuck and provides useful features to steer the execution to relevant states and debug the execution of the $USM$, assuming the required inputs are provided. In the rest of this section, we discuss the important properties of $R^{USM}$.

**Reachability of the execution**

**Proposition 2.** (Reachability of States) Assume $L_r = \langle \Gamma_r, A_r, \gamma_r, Q_r, \rightarrow_r \rangle$ is the execution semantics of $R_{USM}$. Let $\gamma$ be the current configuration of $L_r$, where $\gamma.\sigma \in S_b$. By injecting a $dbg$ message, the execution can be steered by a finite number of execution steps to any configuration $\gamma'$ in which the current state is any state except initial, choice-points, and composite (implicit history) states, assuming that proper inputs are provided.

**Proof.** (Reachability of States) Let $\sigma$ be the current state of configuration $\gamma$, and $\sigma'$ be the current state of configuration $\gamma'$ to which we want to steer the execution. Proving that $\gamma'$ is reachable by taking a finite number of execution steps, includes three cases. (I) Both states $\sigma$ and $\sigma'$ have the same parent, i.e., $parent(\sigma) = parent(\sigma')$. In this case, based on the execution semantics of $USMs$, injecting a $dbg$ message starts an execution step that moves the execution to $dec\_p$ with the same parent (line# 12-14 of Algorithm 2). $dec\_p$ has outgoing transitions to all states that have same parent as $\gamma'$ (added by line# 16 of Algorithm 2). Thus the execution can be steered to $\gamma'$ by providing the required input. (II) $parent(\sigma) \in$
parents(σ') ∧ parent(σ) ≠ parent(σ'). In this case, after the execution reaches the first dec_p, it then can be moved using a series of to_child and from_parent transitions (line# 20-26 of Algorithm 2) until reaching γ' whose current state is σ'.

(III) parent(σ') ∈ parents(σ) ∧ parent(σ) ≠ parent(σ'). In this case, after the execution reaches the first dec_p, it then can be moved using a series of to_parent and from_child transitions (line# 20-26 of Algorithm 2) until reaching γ' whose current state is σ'. Based on (I), (II), and (III) the proof of Proposition 2 is complete.

Proposition 3. (Reachability of Transitions) Let γ be the current configuration of L_r, where γ.σ ∈ S_b. By injecting dbg and related messages, a sequence of execution steps can be taken to execute the action of any transition, except for transitions starting from choice-points or initial transitions.

Proof. (Reachability of transitions) Based on the execution semantics of USMs a prerequisite for the execution of action of transition \( t \) is to move the execution to a configuration γ whose current state is (1) the source state of transition \( t \), or (2) a basic state inside the composite state which is the source of transition \( t \) (when a transition \( t \) start from a composite state). According to Proposition 2, this can be done by injecting a dbg message and providing proper inputs. Thus we have to show that after reaching the source state of the transition \( t \) according to (1) or (2), an execution step can be taken to execute the action of the transition which includes five cases based on the source state of transition \( t \): (1) pseudo-state except for choice-points and initial states, (2) basic state, (3) composite state, (4) choice-points, (5) initial state.
(I) The proof for Case-1 is trivial. According to the Rule-1 (Ref. Sec. 2.5.4), the execution step is taken from these states if there is an outgoing transition originating from them. Thus, an execution step can be taken that executes the action of transition \( t \).

(II) The proof of Case-2 and Case-3 is similar to Case-1 assuming proper input messages are provided (trigger of transition \( t \)). As we discussed in Sec. 7.4.2, we provide message injection feature that simplifies this.

(III) Case-4 is not part of the proposition, because it is not possible to ensure that the transition \( t \) is executed due to its guard expression. Any buggy guard statement can prevent the execution of the transitions originating from a choice-point.

(IV) Case-5 is not part of the proposition. There is no way for the execution to revisit the transition starting from the initial state, except restarting the execution.

Based on the (I) and (I) proof of Lemma 3 is complete. □

Progress of the execution

**Proposition 4.** (Progress of the Execution) The execution of \( \mathfrak{R}^{U_{SM}} \) never reaches a stuck configuration assuming proper inputs are provided.

**Proof.** (Sketch) As we discussed in Sec. 15, the execution gets stuck due to two groups of issues: (1) Missing/ problematic specification. (2) Missing input messages. All of the elements in group (1) are fixed by the refinement. Also, \( \textit{dbg} \) and other relevant messages can be injected by users which prevents a component from getting stuck because of missing inputs. Here, we do not present a detailed proof, but it can be performed similar to the previous proofs. □
Table 6.1: Model complexity of use cases, worst case transformation and analysis time

<table>
<thead>
<tr>
<th>Model</th>
<th>Org. Model Size</th>
<th>Ana. Time (ms)</th>
<th>Trans. Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>Car Door Central Lock</td>
<td>5</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Digital Watch</td>
<td>9</td>
<td>47</td>
<td>57</td>
</tr>
<tr>
<td>Parcel Router</td>
<td>8</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td>Rover</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>FailOver</td>
<td>7</td>
<td>31</td>
<td>43</td>
</tr>
<tr>
<td>Debuggable FailOver</td>
<td>8</td>
<td>350</td>
<td>620</td>
</tr>
</tbody>
</table>

C: Component, S: State, T: Transition, Org.: Original
Refer to Chapter 4 for a detail description of use cases.

6.5.2 Empirical Evaluation

This section details experiments we conducted to assess the performance and overhead of our approach. In the following, we describe evaluation metrics, experiments, and results.

Evaluation Metrics

We formulated the following metrics to assess the practicality of our approach.

**Metric 1 (Performance of Analysis and Refinement).** We use model analysis and transformation to fix partial models for the execution. The analysis and refinement are the core of our approach, and their performance is a crucial metric for the practicality of our approach. Thus, this metric measures the time required for the analysis and transformation of models.

**Metric 2 (Overhead of refinement).** As discussed, the refinement adds certain elements to fix the execution of partial models. These new elements increase the
complexity of the models in terms of the number of components, states, and transitions. This metric first measures how the complexity of refined models changes in comparison with the original ones. The refined model is created temporarily before execution, and is only used for code generation. Thus, this metric also measures the code generation time for original and refined models, to investigate the side effects of the increase.

**Metric 3 (Performance of the debugging probe).** When executing the partial models, the execution of USM is passed to the debugging probe, to read and apply user input. In the interactive model, there is always a delay imposed by users in the loop, and the performance is not an important factor. However, in the batch mode, it is essential that the debugging probe efficiently selects and applies the execution rules. This metric measures the time required to load, select, and parse rules.

**Experiments**

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

**Measuring the performance of static analysis and model transformation (EXP-1).** To effectively measure the performance of analysis and transformation, first we used Epsilon [109] to create nine versions of each model (partial versions), listed in Table 6.1 by removing 10%-90% of their elements, randomly. This results in 60 models (including the original ones). In the rest of this section, we refer to these versions by merely mentioning the model name appended with the percentage of removed elements (e.g., Rover%10 refers to the model of the Rover...
system, 10% of whose elements have been removed randomly). Also, we use percentage without a model name to refer to all models with the same level of missing elements (e.g., 10% refers to versions of all use-cases, 10% of whose elements are missing).

Note that the Debuggable FailOver system in Table 6.1 is a debuggable version of the FailOver system, which is generated using MDebugger. 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.

Second, we ran the model analysis and refinements 20 times against the original and their partial versions, with a configuration in which all components are assumed to be partial. The rationale for the configuration is to measure the performance in the worst-case scenario. As discussed, the refinement and analysis of a partial component is much more expensive than the complete and absent/ignored components. No refinement is applied on complete components, and the behaviour of an absent/ignored component is replaced with a simple generic state machine. We recorded the time required for analysis and refinement, which is a reflection of their performance in the worst-case scenario. We also saved the partial and refined versions of the model that are used in EXP-2.

**Measuring the overhead of the refinement (EXP-2)** First, we measure the complexity of the models, and their refined version resulting from EXP-1 in terms of the number of components, states, and transitions. Second, we performed code generation on them 20 times, and recorded the execution time of the code generation. This experiment reveals how the model complexity is increased when
6.5. EVALUATION

Figure 6.6: Increased complexity of refined models (use cases and partial versions of them applying refinement, and what the effects of this increase are on the code generation.

**Measuring the performance of execution rule selection and application (EXP-2)** To measure the loading/selection time of the execution rules, we generated 10,000 rules with 100 Line of Code (LOC) in the context of the ABM system. We performed a test that loads the rules in four scenarios, in which 10, 100, 1000, 10,000 rules are used accordingly. We recorded the loading time in each scenario. Then, using a test program, we called the rule selection method for the random context based on the ABM system 1000 times, and measured the rule selection times.

To measure the time required to apply execution rules, we randomly generated four execution rule bodies, containing 1, 10, 100, 1000 lines in the context of the ABM system. We ran a test to measure the time required to parse the rule bodies, 20 times. We did not measure the execution time of the rules' body, since their execution time is dependent on their content, which is controlled by users. The
debugging probe executes rules’ body as they are.

**Setting and Reproducibility of Experiments**

We used a computer equipped with a 2.7 GHz Intel Core i5 and 8GB of memory, for all experiments, which is typical development PC. The experiments are automated using bash scripts. The scripts and models are publicly available at [26] 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 of partial models typically needs to be carried out daily.

**Results**

**Metric 1 (Performance of analysis and refinement).** Based on the result of EXP-1, the Analysis Time and Transformation Time columns of Table 6.1 show the median, maximum and minimum time required to analyze and transform the ten versions of each use-case. For the largest model (Debuggable failover), the medians of analysis and transformation are less than two and 14 seconds respectively. It is therefore safe to conclude that the performance of analysis and refinement is reasonable even when the configuration of all components is set to partial which is the worst-case configuration. Typically, the execution of partial models is focused on executing specific components, and the rest of the components are assumed to be complete or absent/ignored which is less expensive to analyze and refine.

**Metric 2 (Overhead of the refinement).** Based on the results of EXP-2, Fig. 6.6
6.5. EVALUATION

shows the percentage of added elements (states and transitions) to the original models and their partial versions, during the refinement (i.e., number of the added element divided by the number of elements before refinement times by 100). Not surprisingly, the number of added elements increases as the number of removed elements from models increases. Removing more elements introduces more problems for the execution, which requires more elements to be added, to fix them. The percentage of added states is between 20% (the median of the percentage of added states for original versions of models) and 216% (the median of the percentage of added states for model versions with 90% removed elements). The percentage of added transitions is between 67% (median of the percentage of added transitions for original versions of models) and 300% (median of the percentage of added transitions for model versions with 90% removed elements). Note that the percentage of added transition for the version with 90% is almost fixed (300%) because most of their elements are removed and the refinement always adds almost the exact same elements to refine them similar to absent/ignored components. The percentage of added transitions is higher than the percentage of added states, since many of the execution problems are fixed by adding transitions. Also, the number of components is only increased by one (i.e., the dbg_agent component). We argue that these overheads are reasonable with respect to the capabilities provided by the refined models, for the following reasons.

- In most of the cases, the refinement adds elements when there is a missing/problematic element and there is no other way to fix them using existing
tools and techniques. Our approach simply automates the fix for problematic elements. Otherwise, users have to fix them manually, which is time-consuming and tedious.

- The refined models are temporary models, which are only used for code generation. Thus, the overhead of added elements has no side effect except for the code generation. The result of the second part of EXP-2 shows that the code generation of the refined model is only 8% slower than the code generation for original models, which is calculated based on the median of time for code generation of refined models, divided by the time for code generation of original models.

- As discussed, the experiments are performed using the worst-case configuration, and their results reflect the maximum costs of our approach. Otherwise, using realistic configurations, which focus on the execution of certain components, can even decrease the complexity of the refined models w.r.t original models. E.g., the refinement of the Debuggable Failover system by setting the Client component as partial and the rest of the components as absent/ignored results in a refined model with 138 states and 326 transitions, which is almost 50% smaller than the original model.

**Metric 3 (Performance of the debugging probe).** Based on the result of EXP-3, Table 6.2 shows the required time for loading and selecting rules by the debugging probe. The selection time is the median time of rule selection for 1,000 times. The loading of rules occurs only once the execution of the system starts. During the loading, the script of execution rules is loaded and parsed. The parsing in this phase only parses the rules’ header, and saves their body as text. As shown in
6.5. EVALUATION

Table 6.2: Required time for loading and selection of execution rules

<table>
<thead>
<tr>
<th># of Rules</th>
<th>LOC of Rules</th>
<th>Loading time (ms)</th>
<th>Selection time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>3</td>
<td>0.006</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>13</td>
<td>0.006</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>96</td>
<td>0.006</td>
</tr>
<tr>
<td>10,000</td>
<td>100</td>
<td>950</td>
<td>0.010</td>
</tr>
</tbody>
</table>

Table 6.2 the debugging probe can load 10,000 rules in less than a second, which is acceptable, because it happens only once.

As discussed in Sec. 7.4.2, in batch mode execution, the debugging probe must select an applicable rule from the defined rules whenever the execution reaches the decision points. As shown in Table 6.2, the rule selection time is negligible (less than a millisecond), and it is, therefore, safe to conclude that rule selection performance is acceptable.

When an execution rule is selected, the debugging probe parses the rule’s body and executes it. Thanks to ANTLR [142], the parsing time of the rule’s body has reasonable performance. Rule bodies with 1-1000 LOC can be performed in less than a second (the median time of 550 milliseconds for 1000 LOC).

According to the results mentioned above (i.e., acceptable performance of analysis, refinement, and debugging probe and reasonable overhead of the refinement), we conclude that our approach is a practical approach for the execution and debugging of partial models.
In this chapter, we have proposed a conceptual framework for the execution and debugging of partial models, which consists of static analysis, automatic refinement, and input-driven execution. Using static analysis, we extract the problematic elements that prevent execution. The problematic elements are automatically fixed by adding decision points and related specification into the partial models. Finally, the refined models are executed with the help of user input, either interactively or by a script. We have created an engine for execution and debugging of partial UML-RT models (PMExec) based on the proposed framework. We have applied PMExec to the debugging of several use-cases, and have evaluated its performance for analysis, refinement, and handling of users input. Despite being a prototype, the performance of PMExec is acceptable, which shows that our approach is a viable approach for the debugging of partial models.
Live Modeling in the Context of the Code Generation

7.1 Introduction

Live programming [58, 134] 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., [156, 144, 85, 4, 14]), and its benefits and utility are discussed in several studies (e.g., [133, 112, 60]). Inspired by this line of work, some efforts [181, 177, 183] 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 model interpretation, and no work supports live modeling when the models are executed by code generation into general programming languages.
As suggested by [183], a possible solution for supporting live modeling in the context of translational execution is implementing the live modeling features on top of the target language of the source code being generated. As discussed in Section 3.3 using offered services in the context of the programming language of generated code imposes several limitations, e.g., edit latency. To overcome the limitations, we propose a conceptual framework that is completely independent of any live programming support offered by the target language. This independence is achieved with the help of model transformation and code generation techniques. The framework consists of two phases: (1) Generation of a Self-reflective Program which is realized through: (a) automatic instrumentation and refinement of models using model transformation techniques to allow for the saving and restoring of previous execution state, which is necessary to support re-execution and the transfer of program state, (b) generation of reflective target 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 debugger 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 using the self-reflective program 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. Also, model debugging services provide an infrastructure for live feedback and safe state transfer.

We have applied the framework in the context of UML-RT (a language for the modeling of soft real-time systems) [162], 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, and only uses open source tools such as the Papyrus-RT MDD tool for modeling, Xtend [37] for code generation, and the Epsilon [109] tools for model transformation. Live-UMLRT provides a full set of services for live modeling of UML-RT state machines, such as 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) code generation, refinement 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 chapter is organized as follows. In the next section, we describe an initial experiment of live modeling using the services offered in the context of the generated code, as well as a running example. Section 7.3 describes our conceptual framework for live modeling and explains how the approach can be applied to UML-RT. Section 7.4 describes the implementation and features of Live-UMLRT. We discuss our evaluation and its results in Section 7.5. Finally, we present a summary of this chapter in Section 7.5.

7.2 Background

7.2.1 A Running Example

As shown in Figure 7.1, we use a partial version of CTR USM of TrafficLight as a running example. Two examples of live modeling operations on the CTR USM
include **OP-1**: add action of transition \( t_{12} \) which is missing. **OP-2**: add a transition along with actions and a trigger from state *yellow* to state *red*.

### 7.2.2 Live Modelling by Leveraging Live Programming (Initial Experiment)

Figure 7.2 shows the process for realizing live modelling using services offered by the target language of the code being generated. 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 application of changes to a model while it is running. We assume that changes are applied to model \( M_1 \) which result in model \( M_2 \). The model-level
7.2. BACKGROUND

Figure 7.2: Live modelling by leveraging live programming services offered in the context of the target language

changes (i.e., \( \text{Diff}(M_1, M_2) \)) are translated into code using incremental code generation, then the 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 \( M_1 \), is updated to refer the shared libraries without restarting the execution which is achieved by generating the code that can detect changes to shared libraries and load/unload them. 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.

Let us review and illustrate (in the context of the running example) some of the significant problems with the above solution:

(1) **Edit Latency.** The process is time-consuming, and our experiment shows that, on average, the process takes more than half a second for each edit. According to \cite{132, 133}, users start noticing latency at 100ms and become distracted at 500ms.

(2) **State transfer.** There are two techniques for state transfer: (a) **Fix-and-continue:** This technique keeps the current state of the program intact and allows the changes to only affect the next execution steps (e.g., \cite{85}). 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 performance), this method can work well. However, it has several problems, as discussed in \cite{133}. E.g., if OP1 is applied when the execution of the CTR is in state **red**, there is no way for users to ever see the effect of OP1. Similar problems can occur for OP2 if the changes are applied when the execution is in state **yellow**. (b) **Re-execution:** This technique recreates the program by replaying the execution traces while changes are being taken into account (e.g., \cite{66}). This technique is not supported by default in the context of, e.g., C, C++ and Java. Re-execution is an expensive approach and introduces extra complexities and increases edit latency.

(3) **Supported operations.** Some approaches prohibit certain edit operations, such as adding new variables or new methods. Often these restrictions exist to
minimize the edit latency, prevent inconsistency in the current execution state, or simplify the implementation. In the context of UML-RT, we need to prevent users from performing some operations (e.g., removing the current state of the execution such as removing the state *green* when it is active). Automatic enforcement of these domain-specific restriction is not supported. Thus a separate method, which monitors the execution and validates the changes, needs to be implemented, which will impose additional latency and implementation overhead.

(4) **Live feedback and 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 observable outputs, the application of changes without their side effects 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.

### 7.3 Approach

#### 7.3.1 An Overview

As discussed in Sec. 7.2.2, 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 a conceptual framework for live modeling in the context of model execution by code generation which is independent of live programming services. An overview of the framework is shown in Fig. 7.3. It consists of two
phases: *Generation of a self-reflective program* and *Live modeling using the self-reflective program*. First, code generation and model transformation techniques are used to automatically create a program (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. 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 live feedback and safe state transfer. In the following, we discuss the details of each phase in generic terms.
Generation of a Self-reflective Program

Fig. 7.3a shows the three steps for 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 live feedback, (b) producing execution traces to support state transfer, and (c) adding a mechanism for the safe transition from the executing mode to the update mode and vice versa that prevents inconsistencies in the execution state during the live update (e.g., removing the current execution state).

**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, (c) the embedded AST should be modifiable during execution. The complexity of this step is language-dependent. Often, modeling languages have concise abstract syntax and semantics that simplify this step for them, e.g., implementing this step for UML-RT took less than a month for a Ph.D. student to complete.

**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 debugger
plugin that provides an interface for using live modeling and debugging services is linked to the 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/debugging service is required.

**Live modeling using the self-reflective program**

Figure 7.3b shows 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 debugging command and sent to the debugger plugin which is loaded as part of the self-reflective program. The debugger validates the request and updates the corresponding elements in the runtime model so that the design and runtime models are kept synchronized. Note that the runtime model encompasses all elements that can be modified during live modeling sessions. Hence, the approach offers an optimized balance where everything static and unchanging is compiled, while the rest (i.e., what can be modified at runtime) is interpreted through the runtime model. Also, debugging commands such as variable view/modify and setting break-points can be processed by the debugger plugin.

Generally, the framework above can be applied in the context of different modeling languages to realize live modeling and debugging. However, 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. In the rest of this section, we discuss the application of the framework for live modeling of UML-RT models.
Algorithm 3: Instrumentation of a UML-RT Model

**Input**: A UML-RT model sys

**Output**: An instrumented model

1. Add a debugging interface dbg_int[dbg_msg] into sys
2. **for all** \( c \in \text{sys.C} \) **do** // (components of sys)
   3. Add port \( \text{dbg}_p \) of type \( \text{dbg}_\text{int} \) into component \( c \)
   4. Instrument(\( \text{root}(c.\beta, \emptyset) \)) // \( c.\beta \) refers USM of \( c \)

**Function** Instrument(Composite state: \( sm \), Entry-point: \( e \))

6. **if** \( e \neq \emptyset \) **then** // (\( sm \) is a composite state)
   7. routing_s ← add_state(\( sm, S_{ch} \))
   8. add_trans(\( e, \text{routing_s} \))
   9. state_temp ← routing_s

10. **else** // (\( sm \) is the root of the USM)
   11. safe_s ← add_state(\( sm, S_{ch} \))
   12. state_temp ← safe_s

13. **for all** \( s_b \in \text{child}(sm) \cap (S_{b}) \) **do** // (all basic states)
   14. **if** \( e = \emptyset \) **then** // (root of the USM)
   15. \( t_1 \leftarrow \text{add_trans}(s_b, \text{state_temp}) \)
   16. \( t_1.\text{trig} \leftarrow \text{dbg}_\text{msg} \)
   17. \( t_2 \leftarrow \text{add_trans}(\text{state_temp}, s_b) \)
   18. \( t_2.\text{guard} \leftarrow \text{nextState} == s_b \)
   19. addExecTrace(\( s_b.\text{exit} \));

20. **for all** \( s_c \in \text{child}(sm) \cap S_c \) **do** // (all composite states)
   21. **if** \( e = \emptyset \) **then** // (root of the USM)
   22. \( t_1 \leftarrow \text{add_trans}(s_c, \text{state_temp}) \)
   23. \( t_1.\text{trig} \leftarrow \text{dbg}_\text{msg} \)
   24. \( e \leftarrow \text{add_state}(s_c, S_{en}) \)
   25. \( t_2 \leftarrow \text{add_trans}(\text{state_temp}, e) \)
   26. \( t_2.\text{guard} \leftarrow \text{nextState} == s_c \)
   27. Instrument(\( s_c, e \))
7.3. APPROACH

7.3.2 Live Modeling of UML-RT Models

Instrumentation of UML-RT models

Our approach adapts and extends the model instrumentation approach introduced in MDebugger (ref. Chapter 5). We use the variable view and change services from MDebugger and extend them with a) support for the generation of execution traces required for state transfer and b) a live update mechanism that respects the execution semantics of UML-RT model and prevents inconsistency. Algorithm 3 shows how a UML-RT model is instrumented for live modelling and uses the following helper functions: root(USM), which returns the root of the USM which is treated as a composite state, child(s ∈ Sc) which returns all states inside the composite state s, add_state(s ∈ Sc, ty ∈ S) which adds a state of type ty inside sC, and add_trans(src, trg ∈ S) which adds a transition from state src to state trg. The algorithm first adds a new interface dbg_int which contains the message dbg_msg which is used for communication between the debugger plugin and the model components (line# 1). For each component, we also add a port dbg_p to allow the debugger plugin to communicate with the component at runtime (line# 2). Then, the following instrumentation is carried out:

- Add choice-point safe_s (safe state) into the root state of the USM and choice-point routing_s (routing state) to all composite states of the USM (line# 6-12). Whenever the execution of the USM reaches the safe state, it activates the debugger plugin to process live update and debugging commands. This
guarantees that live updates can only happen in safe states and the execution of the USM is never interrupted during the transition between states which can lead to inconsistency and violate the run-to-completion semantics of UML-RT and UML (i.e., run-to-completion guarantees that an incoming message will be fully processed before the processing of the next message starts).

• Function \textit{addExecTrace} modifies the exit actions of basic states to ensure the generation of appropriate trace information before the state is exited. An execution trace is defined as a tuple \( \langle \text{state, lastM, vars} \rangle \), where \text{state} \in S_b \) refers to the state being exited, \text{lastM} \) refers to the message that caused the state to be exited, and \text{vars} \) is a map recording the values of variables before the state is 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 to a safe state is taken (line# 14).

• The loops inside the function \textit{Instrument} create transitions (1) from the basic and composite states inside the root of the USM to the safe state and vice versa (2) from routing states inside composite states to basic and composite states inside them. This allows the debugger plugin to (a) steer the execution of the USM from any basic state to the safe state in the root of the USM (main safe state) in which the live updates can be applied, (b) steer the execution from the main safe state to any basic state which allows the replaying of the
Figure 7.4 shows the result of the instrumentation on the USM TrafficLight. As shown, the execution can be steered to the safe state upon receiving the dbg_msg and vice-versa. Thus, to apply an update the debugger plugin can send the dbg_msg message to the component, the message will be processed based on the semantics of UML-RT without interrupting the ongoing execution which will move the execution to the safe state. In the safe state the update is applied and then the execution can move to any basic state (possibly through one or more routing states.
Figure 7.5: State and transitions maps of the instrumented CTR USM in Fig. 7.4

if the basic state is not inside the root of USM).

**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 describe this runtime model without dwelling on the technical details of the code generation. The interested reader may refer to the source code which has been made publicly available\(^1\).

Let us define a state map of an USM as a map from its states to references to trigger maps where a trigger map is a map from a trigger to a non-empty set of transitions. A trigger map of a state records the outgoing transitions from the state

\(^1\)https://moji1@bitbucket.org/moji1/live-umlrt.git
along with their trigger. A *null* trigger is used when a transition has no trigger. Figure 7.5 shows how the instrumented *TrafficLight* in Figure 7.4 is translated to a state map and trigger maps referred to from the state 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 7.5. Similarly, for each transition references to its source and target states, guard and action are kept.

A runtime model of a system is a set of tuples $\langle c \in C, \text{sm} \in TM, N_{vars}, N_{acts}, E_t \rangle$ where $c$ refers to a component, $\text{sm}$ refers to the state map of the component’s USM, TM refers to the set of trigger maps that are referenced from $\text{sm}$, $N_{vars}$ is a map of newly added variables, $N_{acts}$ refers to the newly added or modified actions of the component’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.

**Supporting Live Modeling for UML-RT Using the self-reflective Program**

The live update services are defined based on the runtime model which is populated at the beginning of the execution by the self-reflective program. Any change in the design model is translated to a debugging command and sent to the debugger plugin which, in turn, applies it to the runtime model. The full range of edit operations on UML-RT models including adding/removing/updating states, transitions, actions, triggers, and variables (except remove) are provided. Most of
the services are straightforward to provide simply by adding, modifying, or removing entries in the runtime model. Thus, in the following we only discuss how the execution can be replayed, and how an action can be added without the need to re-generate and re-compile the code.

Algorithm 4 shows how replay (i.e., Function \textit{replay}) and the addition of an action to a transition (i.e., Function \textit{addTAct}) are supported through changes to the runtime model and the use of debugging commands. The replay function first attempts to find a path from the source state \texttt{(fromState)} to the destination state \texttt{(toState)} together with all input messages necessary for that path to be taken. If a path is found, 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 component queue are deferred to ensure the replaying will not be interrupted. The users can recall these messages after the replaying is complete by adding action code or a debugging command. Note that actions added by \textit{addTAct} are saved as scripts and interpreted whenever needed. A special action proxy function loads and interprets these scripts with the help of the debugger. The UML-RT action code interpreter is implemented as part of the debugger plug-in. The interested reader can refer to the source code of the implementation for details.
**Algorithm 4:** Replay and Add Transition Action

1. **Function** `replay(State: fromState, toState, Sequence: requiredMsg, replayedState)
   2. if (fromState=toState) then
      3. trace ← lastT(replayedState[0])  // get the most recent execution trace of
         replayedState[0]
      4. reset(trace.vars)  // reset variable values to values extracted from trace
         (trace.vars)
      5. steerTo(replayedState[0]);  // steer execution, possibly using routing states
      6. defer all messages in the component queue
      7. inject(requiredMsg);  // inject all messages required to replay using debugging
         command
      8. return  // replay complete
   else
      9. trace ← lastT(fromState)
   10. if (trace=null or replayedState.contains(fromState)) then
       11. // error, there is no path to toState
       else
           12. requiredMsg.append(trace.lastM)
           13. replayState.append(fromState)
           14. nextS ← lookupStateMap(fromState,trace.lastM)  // find the next
                state using the state maps
           15. replay(nextS,toState,requiredMsg,replayedState);

16. **Function** `addTAct(Script act, Transition t)
   17. let proxy be a proxy function
   18. t.action ← proxy
   19. addAct(Nact, t, act);  // add action act to Nact

### 7.4 Tool Support

#### 7.4.1 Implementation

To develop *Live-UMLRT*, we used Papyrus-RT as the primary tool to model RTE systems, Papyrus-RT code generation extension to generate self-reflective code, and the Epsilon Object Language (EOL) [109] to implement the transformation rules required for refining the models into instrumented models. The debugger
plugin is implemented using C++, ANTLR [142], and the Boost C++ Library [3]. Debugging features for viewing and modifying variables are ported from MDebugger (ref. Chapter 5).

7.4.2 Live-UMLRT Features

In the following, we discuss the features of Live-UMLRT\(^2\) 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 repository\(^3\). After

\(^2\)A demonstration video: [https://www.youtube.com/watch?v=6GrR-Y9je7Y](https://www.youtube.com/watch?v=6GrR-Y9je7Y)

\(^3\) [https://moji1@bitbucket.org/moji1/live-umlrt.git](https://moji1@bitbucket.org/moji1/live-umlrt.git)
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, first the model instrumentation, code generation, and build are executed automatically in the background without distracting end user, and a connection with debugger plugin is established. Upon a successful connection with the debugger plugin, a UI is loaded as shown in Figure 7.6. The UI is split in two parts, a USM view and a DBG console. In the USM view, the user can view and edit the USM of the capsules. In the DBG console the user can interactively issue commands to investigate and change the model at runtime. Basic debugging commands (e.g., view and change variables) are available and have been ported from MDebugger. Next, we discuss the steps for live modeling along with several features of Live-UMLRT.

**Starting live update session and applying changes:** To start live modeling, a live update session must be started that moves the execution to a state in which the model execution can be changed consistently. A live update session can be started via two methods: 1) Interrupt the execution 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. 2) Live-UMLRT starts a live update session when the execution is stuck because of a missing specification in the model. The detection mechanism is ported from PMExec (ref. Chapter 6).

**Applying changes to the model execution:** During the live update session the user has two ways to update the model execution: (1) **Changing the design model:**
7.4. TOOL SUPPORT

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 *debugger plugin*. As discussed, the debugger plugin 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 debugger plugin. 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 update session. Implementation of this part is left to future work.

**Supported edit operations:** Listing 7.1 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 debugger plugin. 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. (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 1 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 yellow*,...
7.5. EVALUATION

Listing 7.1: 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 <state name>
```

Live-UMLRT steers the execution to state yellow again and injects the last messages that received in state yellow. By that, the execution can advance by processing the injected messages, and the user can see the effect of the new changes.

7.5 Evaluation

This section details experiments we conducted to assess the performance, benefits, and overhead of our approach. In the following, we describe evaluation metrics, experiments, and results.

7.5.1 Evaluation Metrics

We formulated the following metrics to assess the practicality of our approach.

**Metric 1 (Performance of creation of a self-reflective program).** As discussed earlier, our approach for the creation of the self-reflective program consists of the three steps: instrumentation of models, code generation, and compile/built. Creation of the self-reflective program is a core part of our approach. Thus we measure the performance of the instrumentation of the model and code generation steps to evaluate the applicability of our approach. Note that the compile/build
Table 7.1: Model complexity of use cases, median of code generation and instrumenting time

<table>
<thead>
<tr>
<th>Model</th>
<th>Model Complexity</th>
<th>Code Gen. (ms)</th>
<th>Inst. (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>Car Door Lock</td>
<td>5</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Digital Watch</td>
<td>9</td>
<td>47</td>
<td>57</td>
</tr>
<tr>
<td>Parcel Router</td>
<td>8</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td>Rover</td>
<td>6</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>FailOver</td>
<td>7</td>
<td>31</td>
<td>43</td>
</tr>
<tr>
<td>Debuggable</td>
<td>8</td>
<td>350</td>
<td>620</td>
</tr>
</tbody>
</table>

C: Component, S: State, T: Transition, Def.: Default
Inst.: Instrumentation Time, Code Gen.: Code Generation Time
Refer to Chapter 4 for a detail description of use cases.

is performed using code-based tools and their performance is out of our control.

**Metric 2 (Performance of edit operation (Edit delay))**. As discussed earlier, our approach provides live modeling via interaction with the self-reflective program and does not require code generation, compile/built, or hot-swapping for each edit. To show the efficiency of our approach, we measure the performance of edit operations using our approach and compare it with the approach relying on live programming.

**Metric 3 (Performance overhead of the generated code)**. 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, to show the performance overhead of our approach, we measure and compare the performance of the self-reflective program created using our approach with that of the program created using the default code generation.
Metric 4 (Memory overhead of the generated code). Our approach instruments the models by adding new elements and creates a runtime model during the execution. These two together introduce memory overhead. To understand the amount of memory overhead, we measure the memory usage of our self-reflective program and compare it with that of the program generated by the default code generation.

7.5.2 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 measure the performance of the code generation and transformation, 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 7.1 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.

Note that the Debuggable FailOver system is a debuggable version of the FailOver system, which is generated using MDebugger [44]. 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.

Measuring the performance of edit operations (EXP-2). 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 edit operations 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 live programming (Section 7.2.2). 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.

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 interpretation and compiled mode and recorded the CPU time in each case.

**Measuring the performance and memory overhead of our approach (EXP-3).** To measure the overhead of our approach, 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.

**Set up and Reproducibility of Experiments**

We used a computer equipped with a 2.7 GHz Intel Core i5 and 8GB of memory, for all experiments, which is typical development PC. Note that we intentionally used a standard computer comparable to those used by developers, rather than more powerful hardware, because the debugging and live modelling of models typically needs to be carried out daily.
7.5. EVALUATION

7.5.3 Results

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

**Metric 1 (Performance of code generation and model instrumentation).** Based on the result of EXP-1, the Code generation Time and Instrumentation Time columns of Table 7.1 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.

**Metric 2 (Performance of edit operations).** Based on the results of EXP-2, Table 7.2 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% and 92%, 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. Also, our experiment of the execution of actions in interpreted and compiled mode shows that, not surprisingly, the interpretation of actions is 70% slower than the execution of their compiled versions. Note
Table 7.2: Performance of edit operations using our approach and live program

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

*Comps.: Components, Ours: Our approach, Prog.: Live Programming
Trans.: Transition, Ratio: Prog./Ours, Rem.: Remove*

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.

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 [132, 133], users start noticing latency at 100ms and become distracted at 500ms). However, the execution of actions after the first edit downgraded around 70%.

**Metric 3 (Performance overhead of our approach on the generated program).**

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 modelling causes negligible performance overhead w.r.t. to the provided services.

**Metric 4 (Memory overhead of our approach on the generated program).** 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.

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 modelling 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.

7.6 Summary

In this chapter, we have proposed a conceptual framework for supporting live modeling in the context of the model execution 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 Live-UMLRT which 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.
Conclusion and Future Work

Providing proper support for debugging models at model-level is one of the main barriers to a broader adoption of Model Driven Development (MDD) [127]. Existing MDD tools do not provide proper support for debugging system behaviour at model-level. In this thesis, we have proposed three novel approaches to advancing the existing model-level debugging techniques and tools: (1) platform-independent model-level debugging, (2) debugging/executing of partial models, and (3) live modeling. As proof of concept, we have created a prototype for each of the proposed approaches. We also have evaluated the prototypes experimentally, and discussed their costs and benefits. All of the proposed approaches are formulated at model-level using modeling notations, and are implemented using model transformation techniques that validate our thesis statement:
Thesis Statement: The problem of model-level debugging can be addressed using model transformation techniques at model-level. This allows the support of model-level debugging in a way that is compatible with the semantics of the modeling languages, and is independent of programming languages used for the generated source code and program debuggers.

In the remainder of this section, we summarize the main chapters of the thesis, and then discuss promising avenues for future research.

1. **Platform-independent model-level debugging:** We introduced a novel approach to platform-independent model-level debugging that formulates debugging services at model-level and implements them using model transformations. We have implemented a model-level debugger of UML-RT (i.e., MDebugger) that embodies the proposed approach. Our experimental analysis shows that the performance overhead of our approach is in the order of microseconds, while the size overhead is comparable with that of GDB, the GNU Debugger. (Chapter 5)

2. **Execution of partial models:** We proposed a conceptual framework for the execution of partial models. First, a static analysis that respects the execution semantics of models is applied, to detect elements of models that cause problems for the execution. Second, using model transformation techniques, the models are refined automatically, mainly by adding decision points where missing information can be supplied. Third, refined models are executed, and when the execution reaches the decision points, it uses inputs obtained
either interactively or by a script that captures how to deal with partial elements. We have developed an execution engine of partial UML-RT models (PMExec) that embodies our proposed framework. We evaluated PMExec based on several use cases that show that the static analysis, refinement, and application of user input can be carried out with reasonable performance, and that the overhead of approach, which is related to increasing the complexity of models by the refinement, is manageable. (Chapter 6)

3. **Live modeling:** We proposed a framework for live modeling (i.e., updating the model during the execution) that is completely independent of any live programming support offered by the target language. This independence is achieved with the help of model transformations that equip the model with support for, e.g., debugging and state transfer, both of 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 framework in the context of UML-RT and created a Live-UMLRT that provides a full set of services for live modeling of UML-RT state machines. We have evaluated the prototype on several use-cases. The evaluation shows that (1) code generation, transformation, and state transfer 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. (Chapter 7)
8.1 Future Work

Addressing model-level debugging of distributed system in the context of MDD

In this thesis, we only focused on the model-level debugging of centralized systems. Thus, extending this work to model-level debugging of the distributed systems is demanding, as new challenges need to be addressed, including but not limited to (1) Efficient generation and collection of execution traces from possibly many distributed nodes (2) Aggregation of traces and ordering them consistently with the real execution of the system, which requires efficient mechanisms to deal with the observability problem, as discussed in Chapter 2.

In this context, I have been collaborating with two of my lab fellows on a novel technique to order the collected traces of a distributed system using abstract interpretation techniques that do not require logical/physical timestamps[43].

Efficient model-level debugging using live modeling services

Our solutions for platform independent model-level debugging (Chapter 5) and partial model execution (Chapter 6) use model instrumentation at design time that increases the complexity of the models and downgrades the performance of systems being debugged/ executed. With the help of Live modeling (Chapter 7) that allows updating of the models’ elements during the execution, the instrumentation can be postponed and applied at run-time whenever it is needed. We hypothesize that applying instrumentation at run-time can increase the efficiency of our solution significantly. However, we left the validation of the hypothesis to future work.
8.1. FUTURE WORK

Model sketching

As discussed in Chapter 3.2.2, program synthesis techniques use a partial program (i.e., a program with holes) along with a specification, test harness, or reference implementation, and try to fill the holes using synthesis techniques. In the same line with this type of work, we can apply synthesis techniques to complete the missing part of the partial models. In this context, PMExec can be used and extended for exhaustive execution (i.e., exploration of possible choices) of partial models and testing of the synthesized models.

Model mocking framework

While in the code-base development context, there are several mocking frameworks (e.g., Mockito, EasyMock, JMock, Opmock, etc.) that can be used to simulate components of a system [172] for testing, there is a lack of facilities, guidelines, and frameworks in the context of MDD, to help create mockers [173]. In this context, PMExec can be extended for the mocking and automation of unit testing. More specifically, the scripting languages of PMExec can be extended by constructs that are required for mocking and testing.

Self Adaptation

Finally, our focus on Chapter 7 was live modeling, but the proposed framework and implementation would also be of value for work on, e.g., model-based development of self-adaptive systems and models at run-time. More specifically, a similar solution can be proposed for the development of self-adaptive systems
and models at run-time in the context of other modeling languages, or development of self-adaptive systems and models at run-time can be addressed using our implementation in the context of the UML-RT language.
Bibliography


   devblogs.microsoft.com/cppblog/c-edit-and-continue-in-visual-
   studio-2015/.

   gdb.pdf.gz.


   papyrus-rt.

   moji1.github.io/EEE499/.


http://flux.cs.queensu.ca/mase/.


[26] PMExec.  
https://bitbucket.org/moji1/partialmodels.git.

[27] ProgmaDev.  


[29] Stateflow - MATLAB and Simulink.  

http://www.sdl-forum.org/SDL/.

http://www.omgsysml.org/.


[34] Trail: the reflection API. https://docs.oracle.com/javase/tutorial/reflect/.


[68] Hoa Khanh Dam, Alexander Reder, and Alexander Egyed. Inconsistency resolution in merging versions of architectural models. In *Proceedings of


[164] Sagar Sen, Benoit Baudry, and Doina Precup. Partial model completion in model driven engineering using constraint logic programming. In 17th International Conference on Applications of Declarative Programming and
Knowledge Management (INAP 2007) and 21st Workshop on (Constraint), page 59, 2007.


