ADDING RUN-TIME MONITORING TO UML-RT
BY MODIFYING THE PAPYRUS-RT CODE GENERATOR

by

LAITH JUWAIDAH

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
August 2016

Copyright © Laith Juwaidah, 2016

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.
Abstract

In Model-Driven Engineering (MDE), the developer creates a model using a language such as Unified Modeling Language (UML) or UML for Real-Time (UML-RT) and uses tools such as Papyrus or Papyrus-RT that generate code for them based on the model they create. Tracing allows developers to get insights such as which events occur and timing information into their own application as it runs. We try to add monitoring capabilities using Linux Trace Toolkit: next generation (LTTng) to models created in UML-RT using Papyrus-RT. The implementation requires changing the code generator to add tracing statements for the events that the user wants to monitor to the generated code. We also change the makefile to automate the build process and we create an Extensible Markup Language (XML) file that allows developers to view their traces visually using Trace Compass, an Eclipse-based trace viewing tool. Finally, we validate our results using three models we create and trace.
Acknowledgments

First and foremost, I would like to thank my parents without whom I would not be here. They supported me throughout my life and they helped bring me to Canada. If it wasn’t for them, I would not be doing this research.

I would like to thank my supervisor, Prof. Juergen Dingel, for guiding me during my time at Queen’s. Prof. Dingel supervises his students in a way where he provides them with the help and direction they need without making them feel like they lack control over their research or that they are being spoon-fed.

Next, I would like to mention my project partners Suchita Ganesan and Nondini Das with whom I worked closely throughout my Master’s. They are hardworking, helpful, and quick learners, a combination that is valuable in any team member.

I cannot not thank Ernesto Posse enough. Ernesto is an employee at Zeligsoft, the company that develops Papyrus-RT which we work mainly with. He has managed to substitute Stack Overflow when we needed it the most.

Finally, I would like to acknowledge the unique friends I had the pleasure of knowing over the past two years. They have taught me a lot and have provided me with company that helped me maintain my work-life balance. I appreciate everything they have done for me including distracting me (sometimes on purpose). Eric, Doug, Baharm, Ben, Esmesse, Tuhin, and Nicolas. You will be missed.
## Contents

Abstract i
Acknowledgments ii
Contents iii
List of Tables v
List of Figures vi
List of Listings viii
Glossary ix

### Chapter 1: Introduction 1

1.1 Motivation ........................................... 1
1.2 Hypothesis .......................................... 2
1.3 Contributions ....................................... 3
1.4 Organization of Thesis ............................. 4

### Chapter 2: Background 5

2.1 MDE .................................................. 5
2.2 UML-RT ............................................. 7
2.3 Papyrus-RT ......................................... 12
2.4 LTTng ............................................... 13
2.5 Related Work ....................................... 14
2.6 Model-Level Monitoring ........................... 18
   2.6.1 Monitoring Specification UI ................. 18
   2.6.2 Modifying the Papyrus-RT Code Generator 21
   2.6.3 Displaying Trace Results ................. 23

### Chapter 3: Modifying the Papyrus-RT Code Generator 26
List of Tables

3.1 Table summarizing how monitoring other elements would be done. . . 45
# List of Figures

2.1 Class diagram for the Ping Pong model. ......................... 8
2.2 The messages contained in the PingPongProtocol protocol. .... 9
2.3 Structure diagram of the top capsule in the Ping Pong model. ... 9
2.4 Pinger capsule state machine in the PingPong model. ........... 10
2.5 Ponger capsule state machine int he PingPong model. ........... 11
2.6 Papyrus-RT Screenshot ........................................... 13
2.7 A class diagram of the monitoring configuration. ................. 19
2.8 The Running state before monitoring it. .......................... 20
2.9 The Running state’s profiles before monitoring it. ............... 20
2.10 The Running state after monitoring it. ........................... 21
2.11 The Running state’s profiles after monitoring it. ............... 21
2.12 The Eclipse view used to display the trace results in Papyrus-RT. 24
2.13 A highlighted state once the modeller chooses to display the trace results. 25

3.1 The code generation process used by the Papyrus-RT code generator. 26
3.2 Class diagram for the Papyrus-RT code generator. ................ 28
3.3 An overview of the process of generating code after the modifications. 37
3.4 LTTng Profile .......................................................... 39
3.5 Colour wheel ........................................................... 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Pinger capsule state machine.</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>Ponger capsule state machine.</td>
<td>52</td>
</tr>
<tr>
<td>4.3</td>
<td>The UI used to attach the relevant <strong>LTtngProfile</strong> stereotype to the selected element.</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Widget Production model class diagram.</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Widget Production Control Software capsule state machine.</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>Widget Production Robot capsule state machine.</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>Widget Production Workstation capsule state machine.</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>Rover Architecture</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td>Rover Wiring Diagram</td>
<td>58</td>
</tr>
<tr>
<td>4.10</td>
<td>Rover model as seen in the Papyrus-RT model explorer panel.</td>
<td>59</td>
</tr>
<tr>
<td>4.11</td>
<td>Structure diagram of the Top capsule of the rover model.</td>
<td>60</td>
</tr>
<tr>
<td>4.12</td>
<td>The Engine capsule in the rover model.</td>
<td>61</td>
</tr>
<tr>
<td>4.13</td>
<td>The capsule in the rover model responsible for communicating with the distance sensor.</td>
<td>61</td>
</tr>
<tr>
<td>4.14</td>
<td>The capsule in the rover model responsible for communicating with the temperature sensor.</td>
<td>62</td>
</tr>
<tr>
<td>4.15</td>
<td>The Control Software capsule in the Rover model.</td>
<td>63</td>
</tr>
<tr>
<td>4.16</td>
<td>The Engine capsule in the rover model with its states monitored.</td>
<td>63</td>
</tr>
<tr>
<td>4.17</td>
<td>The Rover model's trace results displayed in Eclipse.</td>
<td>64</td>
</tr>
<tr>
<td>4.18</td>
<td>The Engine capsule in the rover model with a state being active.</td>
<td>64</td>
</tr>
</tbody>
</table>
## List of Listings

3.1 Code generated from the Pinger capsule. ......................... 29
3.2 Code generator modification for states. ......................... 43
3.3 Generated state code ........................................... 44
Glossary

• Application Programming Interface (API)

• C/C++ Development Tooling (CDT)

• GNU Debugger (GDB)

• GNU’s Not Unix (GNU)

• General Purpose Input/Output (GPIO)

• Hue, Saturation, Brightness (HSB)

• Linux Trace Toolkit: next generation (LTTng)

• Model-Driven Engineering (MDE)

• Pulse Width Modulation (PWM)

• Unified Modeling Language (UML)

• UML for Real-Time (UML-RT)

• Extensible Markup Language (XML)

• executable, translatable UML (xtUML)
Chapter 1

Introduction

1.1 Motivation

Model-Driven Engineering (MDE) is a software development methodology where the developer creates a model of the system and the code is generated for them automatically (see Section 2.1). Using models to develop software can allow developers to create more sophisticated software due to the abstraction it provides. While traditional development tools such as debuggers can be used on the generated code, using them presents a challenge because it requires the modeller to understand the generated code. For example, modellers can use GNU Debugger (GDB) to place breakpoints but they need to understand the generated code to know what part of the code corresponds to what part of the model to know where to place the breakpoints.

MDE can be used with any modelling language given the tools that support it. While most developers are familiar with Unified Modeling Language (UML), UML for Real-Time (UML-RT) is a bit less well known. UML-RT is a variation of UML that supports real-time applications such as communication that have strict timing requirements.
1.2. HYPOTHESIS

MDE also has development tools of its own such as model execution where the model is executed directly without generating code. Model execution has benefits of its own such as reducing the time between development and seeing the results but it has its drawbacks too. One such drawback is that it cannot be used to determine timing and real-time characteristics on the hardware on which the software will be deployed because the execution happens on the development machine.

The development tool we focus on is tracing. Tracing allows developers to gain insights into the workings of their application as it runs on the system providing them with information such as which events occur and at what time. This is particularly useful for UML-RT development because it provides timing information which can be used to ensure the model fits its timing requirements.

The problem is that tracing tools such as Linux Trace Toolkit: next generation (LTTng) require the developer to add tracing statements in their code before compiling. This is a challenge to MDE since code is generated automatically and developers do not write it themselves.

1.2 Hypothesis

We hypothesize that it is possible to add support for run-time monitoring to an MDE tool such as Papyrus-RT with reasonable efforts by modifying the code generator to insert tracing statements in the generated code for the elements the modeller wants to be monitored.
1.3 Contributions

We make the following contributions in this thesis: identification of approach and its strengths and weaknesses, identification of problems in the tools used, and illustrating the approach on three case studies.

In trying to figure out how to approach the problem we are trying to solve, we are faced with a lot of choices some of which lead to entirely different approaches to the problem. We study these choices carefully and make an informed decision. We discuss the different choices we have to make throughout the thesis along with their benefits and drawbacks, and why we choose the approach we end up using.

Throughout our work, we use plenty of open source tools (Papyrus-RT, LTTng, make, etc...). As we use these tools we encounter bugs some of which are minor while others impede our progress until they are fixed. We report the bugs and sometimes even submit fixes too, both to speed up the process and to give back.

Towards the end of the research, we develop a rover (see Section 4.4). The rover we create is a platform that carries two motors that are attached to wheels, a Single-board Raspberry Pi running Linux on which we run the compiled code and do the tracing. This project will be used for future research by our group.

Finally, we create three different models and use them both to test and to demonstrate what we have created. The first model is a simple model that sends messages back between two components. The second model simulates a factory with workstations that create widgets and robots that deliver them. The final model is a more sophisticated one that is used to interface with the hardware of a rover that avoids obstacles.
1.4 Organization of Thesis

In Chapter 2 we provide a background for our work where we describe some of the tools and technologies we build on and/or utilize as well as a discussion of work related to this research done in the past.

In Chapter 3 we describe the research work done in detail. We start by describing the existing code generator, then we discuss the requirements and restrictions that influence the decisions we make, after that we describe the actual changes made to the code generator, finally, we discuss the other changes that needed to be made to supplement the changes made to the code generator.

In Chapter 4 we present three models we use throughout our work to test new features as we implement them: Ping Pong, Widget Production, and the Rover.

Finally, in Chapter 5 we finish by summarizing the work done, a discussion of the limitations, and possible future work that can extend the benefit of the research done here.
Chapter 2

Background

2.1 MDE

Generally speaking, software development goes through different phases be it in a linear or iterative fashion. In the first phase, gathering requirements, the expectations for the systems are written down detailing things like expected behaviour, expected output, functionality, restrictions, and every other aspect of the system that the developer might need to know about when they design the system. The second phase is design. In this phase the details of how the system is going to be implemented are decided on including classes, interfaces, and user interface. The third phase is implementation. In this phase the designs created in the previous phase are used to write the code for the system. Finally, the system is tested to ensure that it meets the requirements and that there are no bugs.

In contrast, MDE merges the design and implementation phases into one. In MDE, code is generated directly from the model, or, some might say, the model is the code [10]. In the design phase, the UML is used to design the system. UML is a collection of languages and types of diagrams that describe designs of both the structure and
2.1. MDE

the behaviour of the system. Structurally, UML can describe classes including their public and private methods and variables, how the different components of the system interact with each other, and the structure of the system once it is deployed [13]. Behaviourally, UML can describe use cases and timing among other ways of specifying how the system functions.

MDE has many advantages over traditional software development [21]. First, it ensures that the implementation matches the design. This is because the code is generated automatically from the design. If the developer changes the design, that automatically changes the implementation of the system because the code is generated directly from the design. In contrast, in traditional software development if the design needs to change for any reason be it that the design is not permitted by the language, it is deemed inefficient, there are bugs in it, it does not follow the requirements, the requirements change, or any one of many other reasons the design needs to change after it has been created. If the design needs to be changed there are two options: keep the original design, or implement the system differently and update the design to reflect the changes. In industrial software development however, unless the system is highly sensitive, most of the time, design artifacts are abandoned soon after they are created and they become useless as the system changes.

The second advantage is that bugs are discovered earlier in the development process. This is because the design itself can be tested for bugs which means that if bugs are discovered early the design and the generated code are both fixed at the same time so there is no discrepancy between the design and the implementation of the application. Discovering bugs early has the benefit of reducing cost since the cost of fixing bugs increases as development progresses. If bugs can be prevented from happening in the
first place, that would save debugging time and thus money. Similarly, if a bug makes it to release, then more time needs to be spent maintaining the software costing more money.

Additionally, MDE forces the developer to think about design early. The developer cannot simply sit down and write code. Instead, the developer has to design the system forcing them to think about the architecture, interfaces, interactions, encapsulation, etc... All of which are things that can easily be deferred until a later stage of development when writing code directly is an option.

MDE is far from perfect though. The biggest problem that faces MDE is the lack of popularity which results in a lack of documentation and tooling. The number of MDE development tools pales in comparison to generic development environments for standard code-writing development.

2.2 UML-RT

UML-RT is a subset of UML that is meant to be used with real-time applications [19]. Real-time applications differ from ordinary applications in that responses have deadlines that have to be met, for example, after a button is pressed, the feedback must be within 15 milliseconds.

UML-RT adds a few features that are not present in UML. The first feature that UML-RT adds is capsules which are active classes [28]. Capsules have ports that can be attached to capsules which allows them to communicate with other capsules through set protocols. Protocols have incoming and outgoing messages. One port’s incoming messages are the other port’s outgoing messages. This is specified by setting one port to be conjugated which means that its messages are the opposite of what the
protocol specifies, this is in contrast to a base port where the port receives and sends messages as specified by the protocol. Messages can carry data with them which can either be of primitive types (integers, booleans, strings, etc...) or of custom data types which are defined as classes.

Figure 2.1 shows the class diagram for the Ping Pong model (see Section 4.2). The Ping Pong model has three capsules: Top, 
Pinger, and Ponger. The Top capsule is the top-level capsule of the system and it contains everything in the system, in this case that is the Pinger and Ponger capsules. Each capsule contains a port that allows it to communicate with the other capsule which is why they are of the same type: PingPongProtocol.

![Class Diagram](image)

Figure 2.1: Class diagram for the Ping Pong model.

The Ping Pong model also contains the PingPongProtocol that is used to communicate between the two capsules. As Figure 2.2 shows, the PingPongProtocol contains two messages: ping which is an outgoing message that can carry an integer and pong which is an incoming message that can also carry an integer.
2.2. UML-RT

UML-RT also comes with a three types of ports that can be used by the modeller. The first of which is a timer port. Timer ports allow the user to set them to send a message over a certain port either once or repeatedly at a certain interval using the `informIn()` and `informEvery()` message respectively. The second type of port are log ports which allow the modeller to output text to the console similar to `printf()` using the `log()` message. Finally, there are frame ports which provide helper functions for manipulating capsules. Figure 2.3 shows the `Pinger` capsule which, in addition to the port it uses to communicate with the `Ponger` capsule, also contains a timer port which causes the `Pinger` capsule to send a message whenever the timer times out.

Figure 2.2: The messages contained in the PingPongProtocol protocol.

Figure 2.3: Structure diagram of the top capsule in the Ping Pong model.

Capsules’ behaviour is specified using state machines which contain states, transitions, and a few other elements. States can have action code either when the state is entered or exited or both. Action code is code in the language that the code will be
generated in and it can be used for sending messages which we will discuss shortly, setting attribute values, or any other action that need to be done in the language such as interacting with hardware (see Subsection 4.4 for an example). Action code is put as is in the resulting code where that action is supposed to happen. States can also contain other state machines with their own states and transitions. There can be transition from the parent state to a child state and vice versa. Figure 2.4 shows the state machine for the Pinger capsule in the Ping Pong model. The Pinger state machine contains one state called Running and three transitions.

State machines can also contain choice points which act like if statements to determine which transitions are taken based on the condition they have. For example, if you have messages that are arriving at set intervals with data in them, you could use a choice point to decide what action to do next based on what the values are. See Subsection 4.4 for an example.

Figure 2.4: Pinger capsule state machine in the PingPong model.

In addition to states, state machines can have transitions which connect states to
other states. Transitions can have action code in them that executes when they are taken. Transitions happen when a certain message is received on a certain port. Data sent with the message that triggers the transition is only available in the transition’s action code which means it needs to be stored somewhere to be usable later (see below to learn about attributes). Transitions can have guards attached to them which are conditions that determine whether the transition should be taken or not. Guards can test things like message values, attribute values, etc...

Capsules, like classes, can contain attributes. Attributes are variables whose values can be set, changed, and read. Attributes can have default values which can be set during development and be read during execution, alternatively, they can be used to keep track of things during execution such as timer IDs or data that arrives with a message.

![Figure 2.5: Ponger capsule state machine int he PingPong model.](image)

An important principle of UML-RT is run to completion. Run to Completion means that once a transition is triggered all the action codes that are between the
current state and the next one get executed at once without interruption. For example, if state A has a transition to state B and the transition is triggered then the exit code for state A is executed followed by the action code for the transition, and finally, the entry code for state B. Figure 2.5 shows the state machine for the Ponger capsule. In this case, when the onPing transition is taken, three action code segments are executed: the Running state’s exit code, the onPing transition’s action code, and the Running state’s entry code. Those three action codes are ran together without interruption.

UML-RT also supports artifacts. Artifacts provide the developer with a header (.h) and implementation (.c) pair the content of which are generated as-is. See Subsection 3.3.3 and Section 4.4 for examples of artifacts in use.

2.3 Papyrus-RT

Papyrus is an industrial-strength open source MDE tool [5] which runs on the open source Eclipse IDE [7] and supports modelling in both UML and SysUML. Papyrus has several advantages over other industrial tools such as long-term support, commercial support, and no license fees. Because Papyrus is open source users can add features if they so desire.

Papyrus-RT is the real-time version of an open source Eclipse plugin called Papyrus [6]. In addition to allowing the user to create UML diagrams within Eclipse, it adds the ability to generate code from the diagrams. Papyrus-RT takes features and adds UML-RT support.

Figure 2.6 shows a screenshot of Papyrus-RT. On the left there are three panels. Panel A displays a list of the projects in the current workspace. Panel B shows the
2.4. LTTNG

LTTng is a tracing tool used to trace both the Linux kernel and applications running on Linux. Tracing provides the developer with the ability to keep track of certain events as they occur which gives them insights about their application and allows them...
2.5. RELATED WORK

to make decisions based on that information. Tracing tracks and extracts information without adding too much overhead to the application which gives the developer a more accurate insight into the actual performance of the application.

Tracing in LTTng is done by adding a call to the `tracepoint()` function provided by LTTng [3]. In order for the `tracepoint()` function to work, there needs to be a pair of header and implementation files (.h and .c respectively), and the header file needs to be included in the application that is being traced. An important detail about the implementation file is that it needs to have two `#define` lines before the `#include` line at the top for it to work as they are required by LTTng but cannot be generated by the Papyrus-RT code generator. See Subsection 3.3.6 for more details.

2.5 Related Work

There has been quite a bit of research related to the work we are doing here. In this section we take a look at some of that research and compare and contrast it to what we are doing.

Philipp Graf and Klaus D. Müller-Glaser [19] provide a new way to approach debugging MDE applications. They start by describing how software for embedded systems is developed using models, then they compare and contrast debugging and simulation, they then give an overview of the process going from a model to source code to an executable, after that they present an example that illustrates executable stateflow diagrams, finally, they present the architecture that results from their approach before demonstrating a prototype of model debugger.

In their paper, Martin Leucker and Christian Schallhart [23] define runtime verification and, compare well-known verification techniques. They also present applications
in which runtime verification is particularly useful. Additionally, they discuss the differences and similarities between extensions of runtime verification.

Adam Prout et al. present a “semantically configurable code-generator generator” [27]. They make the argument that semantically configurable MDE is a compromise between general-purpose notation and domain-specific languages. Code generator configurability is important to our research since without it we would not be able to make the changes we need to add the features we want.

Shahar Maoz presents his work on runtime tracing of models [24]. His work is similar to ours in that they are trying to monitor events that occur while the applications runs. However, his research differs from ours since he focuses on scenario-based models and traces which we do not deal with scenarios at all. Additionally, we try to take an approach that tries to keep the application as light as possible. Furthermore, our work focuses specifically on UML-RT.

Omar Badreddin et al. [12] present their work which attempts to solve a problem caused by composite state machines which is that the resulting code is large and complex. The approach they take is to convert the composite state machine into several simple state machines. They use Umple to implement their solution.

Gordon Blair, Nelly Bencomo, and Robert B. France introduce and discuss what they call models@run.time. models@run.time is an attempt to make models more abstract. They first about what models@run.time is and why it is important, then they discuss what counts as models@run.time before concluding with some research challenges.

Betty Cheng et al. [14] try to address the challenge of ensuring that software that modify their behaviour during runtime fulfills its requirements. Traditional tools used
to ensure the software meets the requirements such as testing are not enough and, therefore, some tasks need to be performed during runtime. They attempt to use models@run.time to solve the problem by gathering data at runtime specifically to address the problem of assurance.

Shang-Wen Cheng et al. [15] attempt to use software architectural models to support dynamic adaptation in which the software adapts to external elements. Using models allows for decisions to be made based on a global perspective of the system. They conclude by illustrating the application of their idea focusing on adaptations caused by performance-related factors.

Eladio Domínguez et al. [18] try to present an overview of published works that propose solutions for generating code from state machine specifications. They systemically review 53 published studies which they classify into two groups: ones that are based on patterns and ones that are not. They find that most of the works they analyzed are based on design patterns and most do not provide an implementation strategy.

Stefan Heinzmann [20] presents a way to create hierarchical state machines in C++. This is in contrast to flattening hierarchical state machines to flat state machines which makes it hard to connect the statechart to the implementation. He goes to explain how different components such as states, events, and actions are implemented.

Anna Derezińska and Romuald Pilitowski [17] use Framework for eXecutable UML to transform UML models to C# code. Framework for eXecutable UML is made up of two components: a run time library and a code generator. The code is generated for both the structural and behavioural elements of UML. They also check the state machines and classes for correctness. They conclude by discussing testing the solution
on a few UML models.

Alexander Knapp, Stephan Merz, and Christopher Rauh [22] describe HUGO/RT, a prototype tool that automatically verifies whether UML state machines match timing specified in UML collaborations. State machines get compiled into timed automata. A model checker is used to check whether the model matches the specification.

Jauhar Ali [11] published a study with the aim of finding an easy way to generate Java code from hierarchical state machines. He implements state machines using Java Enums. His approach is to represent each state with an enum-value. Hierarchical states are implemented by linking enum-values to one another.

Jilles van Gurp and Jan Bosch [29] discuss implementing finite state machines in object-oriented languages with the intent to replace the state pattern which is often used for this purpose. They present an alternative approach and a framework that implements it. Additionally, they implement a tool that automates the configuration of the framework.

Iftikhar Azim Niaz and Jiro Tanaka [25] propose using design patterns in object-oriented languages such as Java to generate code from statecharts. Their approach is to turn each state into a class encapsulating its actions and transitions. This eliminates the need to use conditions for transitions making the code more readable, and thus maintainable.

There is no shortage of research in this area. MDE has been around for years now and it has consistently been increasing in popularity. While this work is similar to other work in the literature, it is still unique and novel enough to be justifiable.
2.6 Model-Level Monitoring

2.6.1 Monitoring Specification UI

There needs to be a way for the modeller to specify which elements of the model they want to be monitored. There are multiple ways to achieve this. The first choice that needs to be made is how to select the element. One way is to read the Extensible Markup Language (XML) file in which the model is stored, listing the elements in a dropdown box and asking the modeller to select the element they want from there. This method has some disadvantages. The first of which is that if there are elements that have the same name in different parts of the model they would be indistinguishable from each other in the list. The other problem is that the modeller has to specify the path of the XML file which is inconvenient. This is done initially but it is then with a more user-friendly method. That method, which is used right now, is to listen to selection events on the diagrams and get the UML element from the diagram element that is selected. Events can be listened to and when an event is fired, a function referred to as the event handler is called.

The second choice that needs to be made is how to provide feedback that an element is set to be monitored. It seems obvious that changing the appearance of the element is the way to go here. Initially, this is done by modifying the XML file but that requires refreshing the diagram every time which takes a few seconds. That approach is then replaced with Application Programming Interface (API) calls that change the appearance of elements right away.

There also needs to be a way to communicate the modeller’s selection to the code generator. A few methods are considered briefly (see Subsection 3.3.2) but one emerges as the obvious choice fairly quickly and that is to use a UML profile to store this
information. Profiles provide stereotypes which can be attached to element in the model the profile is attached to and stereotypes add attributes to the elements they are attached to. A profile is created for each element. When the modeller selects an element to be monitored a profile is attached to the model, a stereotype is attached to the selected element, and a boolean attribute in the stereotype is set to true. Once again, this is initially done by editing the XML file but is later replaced by API calls.

Figure 2.7 shows a class diagram of the different elements that can be monitored. The top component in the diagram is the Monitoring Configuration. The monitoring configuration refers to the collection of data that is needed to know which elements to monitor and how. While, right now we can only monitor two elements (states and transitions) which are marked by the purple box in the figure, in the future, our work can be expanded to support other types of elements.

Figure 2.8 shows a capsule with a state in it called Running which appears normal with a black border while Figure 2.9 shows the stereotypes attached to the Running state where only the RTStateMachine stereotype is attached to it. After the Running state is set to be monitored, its appearance changes to make it easy for the modeller to identify which elements are monitored and which ones are not. As Figure 2.10 shows,
the Running state now has a purple border that is thicker than usual. Additionally, the Running state now has the LTTngState stereotype attached to it as can be seen in Figure 2.11.

An additional functionality that is implemented is timing constraints to help modellers verify time intervals between two events. The modeller can choose two elements and then selects the time constraint they want to set, this data is written to a text file in the model’s directory. After the code is compiled and traced (see
Subsection 2.6.3 for details), the modeller can verify whether the two events they select occur within the time constraint they set or not.

2.6.2 Modifying the Papyrus-RT Code Generator

The code generator needs to be modified to put statements required by LTTng in the right place in the code so that when the code is run LTTng collects information about the element that the user chose to be monitored. The code generator first checks
whether the element that is being processed has a custom profile associated with it and whether the attribute value is set to true. Once it is determined that the element needs to be monitored, a few things need to happen.

The first thing is to put an LTTng tracing statement in the appropriate part of the code that is associated with the element. This is to tell LTTng when that particular event (the element being active) occurs. The second thing that needs to happen is to generate .c and .h files that are required by LTTng. However, due to how LTTng requires the .c and .h files to be written and due to limitations of the Papyrus-RT code generator, it is not possible to generate the .c and .h files directly. Instead, intermediate .tp files are generated. Finally, the make file needs to be modified so it generates the .c and .h files from the .tp files so they are ready to be included in the other .c files of the project.

In addition to the main features above, a few additional features were added to enhance the user experience and to add additional analysis capabilities. The first of these is to generate a script that automates the invocation of LTTng and the executable. This makes the process of monitoring transparent since one of our goals is to maximize automation. The second feature is the generation of an XML file that can be used by a tool called Trace Compass which is used to view LTTng traces. The XML file contains a list of the the types of events that can occur along with their colours. This XML file, in addition to the trace information gathered by running the generated code, can be used to visualize the the execution of the application providing valuable information about timing.
2.6.3 Displaying Trace Results

Once the generated code has been run and traced, the information gathered needs to be displayed on the model. There is a lot of information that can be displayed and a lot of ways to display that information. For now, only a “step-through” feature is implemented, which allows the user the step through the execution of the application with each element getting highlighted as it becomes active. There are two main parts to doing this: The first is to read the monitoring information, and the second is to display that information on the model.

Reading the tracing information was initially done by asking the user to specify the path to the mounting information, later the tracing script generated along with the code was modified to move the monitoring information to a set location so that it is easy to find. Once the information is located, it is read using an external library.

Once the information is read it needs to be displayed on the model. It was obvious early on that since which elements are set to be monitored is indicated by changing the colour of the elements in the diagram, it is appropriate to use colour to indicate active elements in this phase too. This was initially done by modifying the XML files and refreshing the diagram every time the user steps through a new element. This was later replaced by API calls and the refresh is no longer necessary.

Additional features that are added later include a way for the user to specify time constraints between two monitored model elements. Once the user selects a pair of elements that they want to set a time constraint on, they are prompted for a time constraint. The elements and the time are written to a text file. The text file is then read and in combination with the tracing information can be validated and the result is displayed on the model.
2.6. MODEL-LEVEL MONITORING

Figure 2.12: The Eclipse view used to display the trace results in Papyrus-RT.

Figure 2.12 shows the Eclipse view created for the purpose of displaying trace results in the model. The left-most panel contains a list of traces that were collected for the current model. The middle panel displays a list of all the events that are in the trace that is selected in left panel. The right panel has a few controls that allow the modeller to control the display of information on the model. The “Display Trace” button highlights the first event and its corresponding element (state or transition) in the diagram as can be seen in Figure 2.13. Once the trace is displayed the modeller can step through the events sequentially which highlights them in the diagram. The reset button removes all highlighting from the diagram so it does not distract the modeller once they are done looking at the trace results. The “Validate Time” button allows the modeller to verify that the time constraints they set up hold; the results are displayed at the top of the right panel in red. There are also buttons for controlling live tracing which allow the modeller to receive events from the executed application as they occur.
Figure 2.13: A highlighted state once the modeller chooses to display the trace results.
Chapter 3

Modifying the Papyrus-RT Code Generator

3.1 Papyrus Code Generator

The code generator in Papyrus-RT takes the UML-RT model that the modeller creates in Papyrus-RT (visually or textually) and outputs a C/C++ Development Tooling (CDT) project. CDT is the set of tools provided by Eclipse to facilitate C and C++ development. Eclipse also has its own format for projects. CDT projects contain information about the source code, paths, build instructions, and many more things that make working with a set of related source files easier. The code generator outputs a CDT project that is ready for compilation.

Figure 3.1: The code generation process used by the Papyrus-RT code generator.

As Figure 3.1 [1] shows, the code generator first transforms the UML-RT model to xtUMLrt. xtUMLrt is a meta-model that serves many purposes. Since xtUMLrt simplifies UML it simplifies the code generator itself. However, since it supports both executable, translatable UML (xtUML) and UML-RT, it allows future extension to
Another model that the Papyrus-RT code generator utilizes during code generation is the C++ meta model. The C++ meta model represents the C++ language. There are elements that represent functions, function calls, classes, variables, expressions, etc... Anything that can be written in C++ can be expressed in the C++ meta model.

Next, the Papyrus-RT code generator translates the xtUMLrt model to the C++ meta model. The purpose of this step is to separate the code generator from things like file formatting. By generating a meta model representing the C++ language then generating the actual code from the model, the code generator is not affected by changes in formatting requirements or changes to the syntax of the C++ language. In this phase, the Papyrus-RT code generator creates the classes and functions that will end up in the generated code. When the code generator generates code for a state it creates a function and puts a switch statement in it with a case for every transition going out of that state. When the state becomes active, this function is called. When a message arrives, it runs the case for the transition that is supposed to be triggered by it. See Subsection 3.3.4 for an example of a state’s generated code.

Finally, the Papyrus-RT code generator generates C++ code from the C++ meta model taking care of things such as make file generation (see Subsection 3.3.6 for more details about makefiles).

Figure 3.2 [26] shows the architecture diagram of the Papyrus-RT code generator. UMLRTGenerator starts the code generation process which is initiated by the UI. CppCodeGenerator is the class that controls the entire code generation process. UML2ChangeTracker keeps track of model changes to facilitate incremental code generation and avoid regenerating code unnecessarily. UML2xtumlrtModelTranslator is
Figure 3.2: Class diagram for the Papyrus-RT code generator.

controlled by UML2ChangeTracker and it translates UML-RT to xtUMLrt. GeneratorManager controls the individual classes that translate different elements from xtUMLrt to the C++ meta model. AbstractCppGenerator is the base class for the individual code generator classes for specific UML-RT elements which are controlled by GeneratorManager. From AbstractCppGenerator six concrete code generator classes are inherited: BasicClassGenerator generates passive classes, SerializableClassGenerator generates serializable classes, CapsuleGenerator generates capsules, ProtocolGenerator generates protocols, StateMachineGenerator generates state machines, and CompositionGenerator generates deployment structure. CppCodePattern generates and tracks C++ elements for xtUMLrt elements.

As an example, the Ping capsule we saw in Section 2.2 generates the code in Listing
3.1 which is presented here to give context to what follows. The generated code is a C++ class. The C++ class contains a function representing action codes in the state machine. The C++ class also contains functions that call the action code functions in the order they are to be executed. For example, the function that is executed when the onPong transition is taken calls the function that contains the action code for the onPong transition followed by the function that contains the Running state’s entry action code. The C++ class also contains functions representing the states in the state machine. Each state function contains a switch statement containing a case statement for every transition going out of that state.

Listing 3.1: Code generated from the Pinger capsule.

```cpp
#include 'Pinger.hh'

#include 'PingPongProtocol.hh'
#include 'umlrcommsportrole.hh'
#include 'umlrtmessage.hh'
#include 'umlrslot.hh'
#include 'umlrttimerprotocol.hh'
#include <stddef>
#include 'umlrtcapsuleclass.hh'
#include 'umlrtframeservice.hh'

class UMLRTRtsInterface;

class UMLRTRtsInterface;

Capsule_Pinger::Capsule_Pinger( const UMLRTCapsuleClass * cd, UMLRTSlot * st, const UMLRTCommsPort * * border, const UMLRTCommsPort * * internal, bool isStat )

    Capsule_Pinger::Capsule_Pinger( const UMLRTCapsuleClass * cd, UMLRTSlot * st, const UMLRTCommsPort * * border, const UMLRTCommsPort * * internal, bool isStat )

stateNames[top__Running] = 'top__Running';
stateNames[SPECIAL_INTERNAL_STATE_UNVISITED] = '<uninitialized>';
3.1. PAPYRUS CODE GENERATOR

```c
void Capsule_Pinger::bindPort( bool isBorder, int portId, int index )
{
    if( isBorder )
        switch( portId )
        {
            case borderport_PingPort:
                UMLRTFrameService::sendBoundUnbound( borderPorts, borderport_PingPort, index, true );
                break;
            case borderport_timerPort:
                UMLRTFrameService::sendBoundUnbound( borderPorts, borderport_timerPort, index, true );
                break;
        }
}

void Capsule_Pinger::unbindPort( bool isBorder, int portId, int index )
{
    if( isBorder )
        switch( portId )
        {
            case borderport_PingPort:
                UMLRTFrameService::sendBoundUnbound( borderPorts, borderport_PingPort, index, false );
                UMLRTFrameService::disconnectPort( borderPorts[borderport_PingPort], index );
                break;
            case borderport_timerPort:
                UMLRTFrameService::sendBoundUnbound( borderPorts, borderport_timerPort, index, false );
                UMLRTFrameService::disconnectPort( borderPorts[borderport_timerPort], index );
                break;
        }
}

void Capsule_Pinger::inject( const UMLRTMessage & message )
{
    msg = &message;
    switch( currentState )
    {
        case top__Running:
            currentState = state_____top__Running( &message );
            break;
        default:
            break;
    }
}

void Capsule_Pinger::initialize( const UMLRTMessage & message )
{
    msg = &message;
    actionchain_____action_____PingPong__Pinger__Pinger_SM__Region1__initial( &message );
    currentState = top__Running;
```
3.1. PAPYRUS CODE GENERATOR

```c
const char * Capsule_Pinger::getCurrentStateString() const
{
    return stateNames[currentState];
}

void Capsule_Pinger::entryaction____PingPong__Pinger__Pinger_SM__Region1__Running__onEntry(const UMLRTMessage * msg )
{
    // the following code has been generated
    #define rtdata ( (void *)msg->getParam( 0 ) )
    // generated code ends
    // the following code has been generated
    #undef rtdata
    // generated code ends
}

void Capsule_Pinger::transitionaction____PingPong__Pinger__Pinger_SM__Region1__initial__onInit(const UMLRTMessage * msg )
{
    // the following code has been generated
    #define rtdata ( (void *)msg->getParam( 0 ) )
    // generated code ends
    std::cout << getName() << " :/uni2423timer/uni2423started " << std::endl;
    timerPort.informIn( UMLRTTimespec( 1, 0 ) );
    // the following code has been generated
    #undef rtdata
    // generated code ends
}

void Capsule_Pinger::transitionaction____PingPong__Pinger__Pinger_SM__Region1__onPong__onPong(const UMLRTMessage * msg )
{
    // the following code has been generated
    #define umlrtparam_param ( *(int *)msg->getParam( 0 ) )
    #define rtdata ( (int *)msg->getParam( 0 ) )
    // generated code ends
    std::cout << getName() << " :/uni2423pong(/uni2423" << rtdata << "/uni2423)/uni2423received ,/uni2423sending/uni2423ping" << std::endl;
    PingPort.ping( rtdata + 1 ).send();
    // the following code has been generated
    #undef rtdata
    #undef umlrtparam_param
    // generated code ends
```
3.1. PAPYRUS CODE GENERATOR

void Capsule_Pinger::
transitionaction_____PingPong__Pinger__Pinger_SM__Region1__onTimeout__onTimeout(const UMLRTMessage * msg )
{
  // the following code has been generated
  /* UMLRT-CODEGEN:platform://resource/PingPong/PingPong.uml#i0kEzZcSWp5V1YWzXj */
  #define rtdata ( (void *)msg->getParam( 0 ) )
  // generated code ends
  std::cout << getName() << " :/uni2423timeout ,/uni2423sending/uni2423ping(/uni24230/uni2423)" << std::endl;
  PingPort.ping( 0 ).send();
  // the following code has been generated
  #undef rtdata
  // generated code ends
}

void Capsule_Pinger::actionchain_____action_____PingPong__Pinger__Pinger_SM__Region1__initial(const UMLRTMessage * msg )
{
  transitionaction_____PingPong__Pinger__Pinger_SM__Region1__initial__onInit( msg );
  entryaction_____PingPong__Pinger__Pinger_SM__Region1__Running__onEntry( msg );
}

void Capsule_Pinger::actionchain_____action_____PingPong__Pinger__Pinger_SM__Region1__onPong(const UMLRTMessage * msg )
{
  transitionaction_____PingPong__Pinger__Pinger_SM__Region1__onPong__onPong( msg );
  entryaction_____PingPong__Pinger__Pinger_SM__Region1__Running__onEntry( msg );
}

void Capsule_Pinger::actionchain_____action_____PingPong__Pinger__Pinger_SM__Region1__onTimeout(const UMLRTMessage * msg )
{
  transitionaction_____PingPong__Pinger__Pinger_SM__Region1__onTimeout__onTimeout( msg );
  entryaction_____PingPong__Pinger__Pinger_SM__Region1__Running__onEntry( msg );
}

Capsule_Pinger::State Capsule_Pinger::state_____top__Running(const UMLRTMessage * msg )
{
  switch( msg->destPort->role ()->id )
  {
    case port_PingPort:
      switch( msg->getSignalId() )
      {
        case PingPongProtocol::signal_pong:
          actionchain_____action_____PingPong__Pinger__Pinger_SM__Region1__onPong( msg );
          return top__Running;
        default:
          this->unexpectedMessage();
          return top__Running;
      }
  }
3.1. PAPYRUS CODE GENERATOR

```c
static const UMLRTCommsPortRole portroles_border[] = {
    { Capsule_Pinger::port_PingPort, 'PingPongProtocol', 'PingPort', '.', 1, true, false, false, false, false, true },
};
```
3.2 Implementation Requirements

In this section we will discuss the requirements and limitations that dictate the details of the implementation.

3.2.1 Performance

The goal of this thesis is to help modellers track of the performance of their applications once the code is compiled and running whether it is on the hardware it will eventually run on or simply on their computer. Developers can set different restraints for different hardware to account for the difference in processing power. Regardless of where modellers want to run the compiled code, they would like to know where delays happen which means that we need to minimize the impact monitoring has on the performance of the application.
There are a few ways to monitor what is going on in the application as it runs and keeps track of time. The first of which is to make calls to a function that outputs the name of the event and the current system time to a file which can be read later. The second option is to use a monitoring tool such as LTtng to monitor the application as it runs. Since tracing tools are meant to have minimal impact on the performance of the application (see Subsection 3.3.3), and they provide more information than just a time stamp, they are the superior choice.

3.2.2 Maintaining the Code Generator’s Semantics

The code generator has been designed to generate code to reflect how the UML-RT model is meant to be run. Our changes to the code generator cannot change the behaviour of the generated code. For instance, functions that are generated for states contain switch statements. We cannot alter that in order to monitor the behaviour. We can add to the generated code (assuming we do not hinder performance too much (see Subsection 3.2.1) as long as it does not alter the execution of the application. We cannot remove code since it alters the existing behaviour of the generated code. Additionally, removing code changes the behaviour of the generated code which means that tracing it now does not give us an accurate reflection of the actual performance of the application which is the goal of this project in the first place (see Section 1.1).

3.2.3 Naming Requirements

Since modifying the code generator is only a part of a bigger project, we have to take that into account. As mentioned in Subsection 2.6.3, after the code is generated, compiled, and traced, the results of the trace are supposed to be displayed on the
3.2. IMPLEMENTATION REQUIREMENTS

model. In order to display trace results on the model, we have to find a way to make it possible to link an event that occurs during execution back to the element that generates it.

There are multiple ways to link an event back to the element that generates it. As discussed in Subsection 2.6.1, parsing the XML file for elements and using their unique IDs could work here but we face the same problems we discussed in that subsection, namely, that we need to specify the path to the XML and that manipulating the XML file could easily break the model since there are no protections in place to ensure validity.

The other option, which is the one we end up using, is to include the path to the element in the model in the event name. For instance, the name of the event that indicates that the Running state in the Pinger capsule we saw in Section 2.2 would be ActiveState__Workstation__Workstation_Producing where ActiveState indicates the type of event, Workstation is the name of the capsule, and Producing is the name of the state. This makes it possible to know what happened and which element it happened to.

3.2.4 Trace Compass Visualization

Trace Compass is an application that allows users to view and analyze traces in a graphical way [8]. Once the tracing data is gathered it is displayed back on the model by highlighting the active state and providing the user with timing information. However, in some cases, this is not enough and the modeller wants to visualize the execution of the application differently. For such situations, we can add support for Trace Compass.
Two things are required to visualize traces in Trace Compass. The first is the trace data. The second is an XML file that describes the trace which we create during code generation using Xtend. The XML file must contain a list of the possible events, and a list of the colours that correspond to each one to be used to display events of that type.

3.3 Implementing the Changes

3.3.1 Overview

Figure 3.3: An overview of the process of generating code after the modifications.

Figure 3.3 shows an overview of the process that goes on when the modeller generates code using the modified code generator. We use 1s and *s to represent multiplicities between the components (one to one, one to many, and many to many). The model which is the internal representation in Eclipse contains the monitoring
3.3. IMPLEMENTING THE CHANGES

configuration which is represented using the monitoring profile (see Subsection 3.3.2). The original part of the code generator generates code as it normally does while the modified code generator extracts event names from the monitoring configuration, generates a .tp file for each event type (see Subsection 3.3.4), includes the header files required to trace using lttng (see Subsection 3.3.3), and generates a .xml file to be used for visualization in Trace Compass (see Subsection 3.2.4). The modified code generator also outputs a modified version of the makefile which includes instructions for compiling code instrumented with lttng. When the developer compiles the generated code using the makefile it generates the header (.h) and implementation (.c) files from the .tp files using the lttng-gen-tp files and then compiles the header and implementation files it normally does (see Subsection 3.3.6). The executable is ready to be traced using lttng. The developer can then run and trace their application which outputs an lttng trace. The lttng trace can be displayed on the model (see Subsection 2.6.3) or used in combination with the generated XML file to visualize the results in Trace Compass (see Subsection 3.3.7).

3.3.2 Monitoring Profile

As discussed in Section 2.6, modifying the code generator is only one of three parts in this project. Therefore, it was necessary to find a way for the UI to interface with the code generator. The UI informs the code generator which elements need to be monitored so it knows which elements need to have tracing statements.

We briefly considered using a text file to store that data. However, this has many drawbacks. The first of which is that it is not obvious how we will represent the different elements in the file. We could use names but what if the user renames the
3.3. IMPLEMENTING THE CHANGES

state? Then the text file will point to a state that does not exist and we have no way of knowing which state is supposed to be monitored. Direct access to the XML file representation is not an elegant solution since it does not do things the proper way (through Papyrus APIs); we are not supposed to access the XML file, the ids are not meant to be used externally. Accessing the XML file directly is discouraged because there are no safeguards to keep the model safe therefore a simple mistake can break the model and make it unusable.

The solution we settled on was to use a UML profile to indicate which elements are to be monitored. Since the Papyrus API supports applying profiles, updating the UI to allow the user to indicate which elements are to be monitored is easy.

![Figure 3.4: LTtng Profile](image)

A UML profile adds the ability to add stereotypes to elements. Stereotypes add additional attributes to elements. As can be seen in Figure 3.4, we added two stereotypes `LTtngState` and `LTtngTransition` which add the `isMonitored` boolean attribute to their corresponding elements.

We could have used one stereotype, say, `LTtngElement` to indicate that an element is monitored. Similarly, we could have done without the attribute since it is just a boolean so the presence or absence of the stereotype would do just fine. However, we decided to do things this way as a preparation for future work. We realized that this
is a very basic way of monitoring elements. While simply indicating whether an element is to be monitored or not is sufficient for now to prove that this can be done, it is nowhere near unlocking the full potential of monitoring MDE applications on the model level. There is much more that can be done to make monitoring MDE applications more powerful. For instance, the user could specify values to be passed as parameters in `tracepoint()` calls (such as strings, integers, etc...) which can be later displayed back on the model. Alternatively, when monitoring attributes to keep track of their values as they change is implemented, the user might want to specify an interval during which attribute values are monitored in which case there would need to be a numerical attribute in the stereotype to specify that.

These are a few examples of what could be done in the future that require specifying and reading attribute values in the stereotype. That is why we decided to use an attribute in our stereotype to specify which elements are monitored.

### 3.3.3 Tracing with LTTng

As seen in Section 2.4, LTTng allows developers to track events as they occur in their applications by adding `tracepoint()` calls and including header and implementation files in their code. In order to use LTTng in our generated code, we need to do two things: add the `tracepoint()` calls in the generated code, and generate and include the required header and implementation files.

The header file along with its implementation file can be created in two ways: They can either be written by hand or generated automatically from a `.tp` file using the `lttng-gen-tp` tool that comes with LTTng. To generate a header and implementation
pair using the code generator we would use a UML-RT artifact from which a header
and implementation pair is generated that is ready to be included in code generated
for other components such as capsules. To generate a .tp file we would use an Xtend
file.

There is a limitation in the code generator that prevents any code from being
added before \#include lines at the top of files generated by artifacts. This limitation
prevents us from using the first method thus forcing us to generate .tp files and then
using the lttng-gen-tp tool to generate the header and implementation files from it.

Xtend is a language that is compiled to Java. It has a few features that make it
superior to Java when it comes to generating files such as template expressions which
handle white space in a much better fashion [9]. Xtend allows the developer to create
templates with place holders that are placed as is in the compiled code. Another
option to generate the .tp files is Java. We did not spend much time deciding on the
best approach here since the Papyrus-RT code generator uses Xtend to generate files
of similar nature, it was logical to use the same tool.

.tp files define the name of the lttng event along with the names and types
of arguments that go along with it. For our purposes, we do not have a use for
tracepoint() event arguments. However, we do support them just in case they are
needed in the future. As a results, .tp files only differ in the names of events they
represent.

3.3.4 Monitoring Behavioural Elements

In this subsection, we will explain how we implement the monitoring of the behavioural
elements that the modeller chooses. The behavioural elements in UML-RT are states
and transitions. The first thing we need to figure out is what it means for a state or a transition to become active.

Listing 3.2 shows one of the additions we make to the code generator, particularly, the part responsible for adding `tracepoint()` calls to states which is overriding a function in the `StateMachineGenerator` class. This cost listing is provided here to give an example of the modifications we make to the code generator. The function first checks if the state has the `LTTngState` stereotype is attached to it and if the `isMonitored` boolean attribute in the stereotype is set to true, if that is the case then it proceeds to add the `tracepoint()` call to the state’s generated code. In order to add the `tracepoint()` call to the state’s generated code, the code generator creates an instance of the C++ meta model function (Macro) specifying what arguments it accepts. Once the Macro is created, we utilize it by adding an instance of MacroExpression which is a function call and provide it with values for its arguments. Once we have an instance of a function call, we add it to the function that represents the state we are generating. Finally, we add an `#include` line to include the header file which we will generate later from the `.tp` file (see Subsection 3.3.6).
3.3. IMPLEMENTING THE CHANGES

Listing 3.2: Code generator modification for states.

```java
protected override
create super.generateStateFunc(state)
generateStateFunc(State state) {
    val b = LTTngGenerationProperties.getLTTngStateIsMonitored(state)
    val isMonitored = b != null & b.booleanValue
    if (isMonitored) {
        val thisCapsuleElementList = cpp.getElementList(CppCodePattern.Output.
            CapsuleClass, capsuleContext)

        var List<TPFileArgument> args = new ArrayList
            args.add(new TPFileArgument("my_string", "string"))

        val state_name = "RT__ActiveState__" + thisCapsuleElementList.name.folderName + "__" + state.name.substring(state.name.lastIndexOf(":" + 1)

        // Generate TP file
        new TPFileGenerator().generate(cpp.outputFolder.getAbsolutePath() + "/" + state_name + "−tp.tp", state_name, args);
        cpp.tpFiles.add(new FileName(state_name + "−tp.tp"));

        // declare the tracepoint macro
        val tracepoint_macro = new Macro("tracepoint")
        tracepoint_macro.addParameter(new MacroParameter("provider")
            tracepoint_macro.addParameter(new MacroParameter("name")

        for (arg : args) {
            tracepoint_macro.addParameter(new MacroParameter(arg.name)
        }

        // we don't need to add this macro object to any element list, because we can hard-code the name of the header file where it is defined
        // Define the macro expansion call
        val call = new MacroExpansion(tracepoint_macro)
        call.addArgument(new ExpressionBlob(state_name + "_provider")
            call.addArgument(new ExpressionBlob(state_name + "_tracepoint")
            call.addArgument(new StringLiteral(state_name)

        // Add the call to the state function's body (at the beginning)
        body.add(0, new ExpressionStatement(call))

        // Add the #include to the capsule's element list
        thisCapsuleElementList.addDefnDependency(new DependencyBlob("#include\":" + state_name + "−tp.h"")
    }
```
3.3. IMPLEMENTING THE CHANGES

Listing 3.3 shows the code generated for a state called Producing in a capsule called Workstation which is contained in a capsule called top. The Producing state has one transition going out of it called finished that is triggered by the ProductionTimer timer’s timeout message. Once inside the case block, the function that the code generator creates which contains all the code that needs to be run in the run to completion sequence (the state’s exit code, the transition’s code, the other state’s entry code, etc...) is called.

Listing 3.3: Generated state code

```
Capsule_Workstation::State Capsule_Workstation::state_____top__Workstation_Producing( const UMLRTMessage * msg )
{
    tracepoint ( ActiveState__Workstation__Workstation_Producing_provider ,
    ActiveState__Workstation__Workstation_Producing_tracepoint , "ActiveState__Workstation__Workstation_Producing" ) ;

    switch( msg->destPort->role() ->id )
    {
        case port_ProductionTimer:
            switch( msg->getSignalId() )
            {
                case UMLRTTimerProtocol::signal_timeout :
                    tracepoint ( MessageReceived__Workstation__Workstation_finished_provider ,
                    MessageReceived__Workstation__Workstation_finished_tracepoint , "MessageReceived__Workstation__Workstation_finished" ) ;
                    actionchain_____top__Workstation.finished__ActionChain4( msg ) ;
                    return top__Workstation_Standby ;
                default:
                    this->unexpectedMessage() ;
                    break ;
            }
            default:
                this->unexpectedMessage() ;
                break ;
    }
    return currentState ;
}
```

There are two tracepoint() calls in this function. The first is at the very top. It is called whenever the function executes which happens every time the state gets activated. This tracepoint() call indicates that the state is active and triggers
the `ActiveState__Workstation__Workstation_Producing_tracepoint` tracepoint which indicates that the event of the producing state becoming active has occurred. The other `tracepoint()` call is placed inside the case that corresponds to the `ProductionTimer` timer’s `timeout` message. This means that it is run only when a message that triggers that transition is received. This ensures that that `tracepoint()` call is only run when the transition is taken.

The above `tracepoint()` calls are added based on which components are marked to be monitored. During the code generation for states and transitions, specifically the part that converts the xtUMLrt model to the C++ model, we check whether the `LTTngState` stereotype is attached to the current state or transition, and whether the `isMonitored` attribute is set to true. If both conditions hold, when we add the corresponding `tracepoint()` call.

### 3.3.5 Monitoring Other Elements

While our work only implements tracing for behavioural elements (states and transitions), it is worth discussing tracing some other elements. This will be done hypothetically, addressing potential pitfalls.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Place in code where the <code>tracepoint()</code> would go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message sent</td>
<td>In action code.</td>
</tr>
<tr>
<td>Attribute value initialized</td>
<td>Near capsule attribute declaration.</td>
</tr>
<tr>
<td>Attribute value changed</td>
<td>In action code.</td>
</tr>
</tbody>
</table>

Table 3.1: Table summarizing how monitoring other elements would be done.

Table 3.1 summarizes how we think future implementation of certain elements. Attributes’ values can only be changed during execution using action code. The fact that attribute value assignments can occur in any snippet of action code makes
monitoring attribute value changes particularly difficult because the code generator does not have a particular place to check whether an attribute value change is occurring or not. One way to go around this is to check the value of the attribute at set intervals which is far from perfect because important things can happen between the value captures that the modeller would have no access to. Another way to solve this problem is to parse every action code snippet, which is how behaviour is specified for states and transitions, and look for assignment statements and put a `tracepoint()` call after every occurrence of an assignment statement. This solution provides the most details but is harder to accomplish since it requires a parser rather than just a button. The other challenge is that there are no message elements in the model that we can attach stereotypes to. We can add a button next to the field where action code is written that adds `tracepoint()` calls directly to the action code, or we can allow developers to write the `tracepoint()` calls themselves.

Initial attribute values are set during development and are placed as-is in the generated code. `tracepoint()` calls can be added directly where initial values are set in the code.

Sending messages, for example, is done in action code follows a certain format which looks like this: `port_name.message_name().send()`. This is similar to how changing attribute values is done. Therefore, we can use a similar approach to monitoring changing attribute values (adding `tracepoint()` calls directly in the action code).

There are other events that may be monitored which we will not discuss here since the way to monitor them is not obvious and further research is needed. See Figure 2.7 for a list of possible events.
3.3. IMPLEMENTING THE CHANGES

3.3.6 Makefile

Make is a tool provided by GNU’s Not Unix (GNU) which automates the build process (converting source code to an executable) by specifying certain rules for certain files and applying said rules to all the files that fit the criteria [4]. Rules can be chained which allows for multi-level builds which is essential for our purposes.

As discussed in Subsection 3.3.3, since it is not possible to generate header and implementation files directly from the code generator we have to generate .tp files using Xtend and then generate the header and implementation files from them by utilizing the lttng-gen-tp tool.

Since the goal of this thesis is to help modellers do as much as possible in the model, it is important to automate as many processes as possible so that the modeller does not need to do much outside the model. As such, we need to automate the header and implementation file generation generation from the .tp files so the modeller does not need to do that manually.

Regardless of how we choose to implement this, we need to have a list of tracepoint() calls (and .tp files) in order to run the lttng-gen-tp tool since it requires the name of the .tp file as input. In order to have this list ready, we need to keep track of the tracepoint() calls throughout the generation process.

There are a few options that allow us to automate the process of generating header and implementation files from .tp files that we need to consider. The first is to generate a script that the user needs to run before running the makefile to generate the header and implementation files thus making them available for make to compile. The second option is to do the generation in the makefile. The latter option is superior to the former because it requires the modeller to run one fewer script to get their code
3.3. IMPLEMENTING THE CHANGES

3.3.7 Supporting Trace Compass

As discussed in Subsection 3.2.4, we need to generate an XML file containing a list of events and the colours that correspond to them. In order to do that, we use the list of `tracepoint()` calls to create the list of types of events since each trace point call represents one event type. For the list of colours we generate colours by dividing the colour wheel (see Figure 3.5) by the number of events we have and use those points on it to get the colour for each event, this way we ensure that each event has a distinct colour and that the colours are not too close to each other. This does not
3.3. IMPLEMENTING THE CHANGES

take into account the fact that the human eye is more sensitive to colours in the green part of the spectrum. More specifically, this is done by utilizing the Hue, Saturation, Brightness (HSB) colour model where the hue determines the colour itself (represented by the wheel) and ranges from 0 to 360 saturation determines the whiteness of the colour and ranges from 0 to 100%, while brightness determines the darkness of the colour [2]. We divide the range of the hue value by the number of events and those are the values we use for the hue part of the colour model.
Chapter 4

Proof of Concept

4.1 Overview

In this chapter we discuss different models we use throughout our work to evaluate the features we add to the Papyrus-RT code generator. The first model we discuss is the Ping Pong model. Ping Pong is a simple model which makes it ideal for testing features when they are first implemented since there is little that can go wrong. The second model is the Widget Production model which is meant to simulate a factory. We use Widget Production to evaluate our work once it has advanced a bit more. Due to its complexity relative to Ping Pong, it is ideal for demonstrating certain features such as monitoring active states since there is more than one state in a capsule which makes for an excellent visualization. Finally, we use the Rover model as a way to demonstrate all the features we have at once. It also serves the purpose of demonstrating the applications of our work such as monitoring hardware remotely and receiving the updates live as they happen.
4.2 Ping Pong

The Ping Pong model is a very simple model that is provided with Papyrus-RT’s source code. The Ping Pong model is used to test every new feature we implement because of its simplicity.

As Figures 4.1 and 4.2 show, the Ping Pong model consists of two capsules **Pinger** and **Ponger** and one protocol, **PingPongProtocol**, which contains two messages: **ping** and **pong**. The **Pinger** capsule sends a ping message to the **Ponger** capsule at set intervals which in turn sends a pong message back.

In the Ping Pong model, the developer can choose to monitor the Running state, the initial transition, the onTimeout transition, or the onPong transition in the Pinger capsule, and the Running state, the initial transition, or the onPing transition in the Ponger state.

Once the developer chooses which elements they want to monitor, they can use the
4.3. WIDGET PRODUCTION

UI in Figure 4.3 which activates automatically once an element is selected. Clicking the “Monitor” button attaches the applicable \texttt{LTTngProfile} stereotype to the selected element and sets the value of its \texttt{isMonitored} attribute to \texttt{true}. Figure 2.10 shows the \texttt{Running} state being set to be monitored.

4.3 Widget Production

The Widget Production model simulates a factory where there are workstations where widgets are produced and robots that deliver the produced widgets from the workstations to their final destinations in the factory. Both the workstations and the robots are managed by the control software which directs both the workstations and robots ensuring they work together seamlessly and shuts down at the end of the
4.3. WIDGET PRODUCTION

simulation.

The Widget Production model is used to evaluate

As can be seen in Figure 4.4, the Top capsule contains two capsules. The first capsule contained within Top is ProductionLine which represents the production line of the widget factory, that is why it contains both the Workstation capsule and the Robot capsule. Both the Workstation capsule and Robot capsule have an Idle state that they stay in until they receive a message that tells them to become active in which case they go to a state that represents them being active (Producing and Delivering respectively) where they set a timer to simulate performing the task then they return to the Idle state when they are done. The second capsule contained within top is the ControlSoftware capsule which contains all the logic that controls

Figure 4.4: Widget Production model class diagram.
both the widgets and the robots.

![Diagram of Widget Production Control Software state machine]

Figure 4.5: Widget Production Control Software capsule state machine.

Figure 4.5 shows the state machine for ControlSoftware. ControlSoftware first goes in the StartUp state which serves the purpose of allowing the other capsules to initialize so that they are ready to receive messages that ControlSoftware sends to them. ControlSoftware does nothing in that state other than setting a timer to timeout in a few seconds which takes ControlSoftware to the Produce state. Once in the Produce state, ControlSoftware sends a message to Workstation telling it to produce widgets. Once the Workstation capsule is done with its task it sends ControlSoftware a message telling it it is done delivering. This message causes ControlSoftware to go into the Delivering state which sends the Robot capsule a message telling it to deliver the widgets, once the Robot capsule is done it sends ControlSoftware a message telling it it is done which causes ControlSoftware to repeat the process again. The loop ends after 30 seconds which is when the system shuts down.
Figure 4.6 shows the Robot capsule state machine. The Robot capsule state machine contains two states: Standby and Delivering. The capsule starts in the Standby state and waits for the deliverWidget message which takes it to the Delivering state. The Delivering state’s action code sets a timer to time out in 1 second triggering the transition taking it back to Standby where it waits for another deliverWidget message.

Figure 4.7 shows the Workstation capsule with all its Standby and Producing states being monitored. Since the transition that goes from Producing to Standby is triggered by a timer which times out after 1 second, we can use monitoring to know for sure if Producing and Standby are becoming active within a reasonable period of time.
4.4 Rover

4.4.1 Overview

As a final demo, to illustrate what our research makes possible, we decided to use a rover. Our rover is a small vehicle with two motors to move it in different directions, and a single-board computer that runs Linux to keep all the applications and the logic on. The purpose of this rover is to demonstrate the applications of what we created. It does so by allowing us to show a real application. Additionally, we want to demonstrate receiving events in real-time and displaying them on the model as they come in. It also serves to demonstrate our vision [16].

Figure 4.8 shows the architecture of the rover we used. The very bottom layer is the hardware layer of the Rover which is described in detail in Figure 4.9. The second layer is the file system. The Raspberry Pi runs on Linux. The Raspberry Pi has 26 General Purpose Input/Output (GPIO) pins, 17 of which can be used to interact with hardware connected to them. The other nine consist of: two 5 volt pins, two
3.3 volt pins, and five ground pins. Each GPIO pin in the Raspberry Pi corresponds to a file in the file system. The user can interact with the pin by reading the file’s value or writing to it. The third layer is the GPIO library which is a C++ class that facilitates interacting with the file system for the purposes of interfacing with the pins. The three layers discussed so far which are indicated with light blue rectangles are in the model level while the next two layers we will discuss are not. The fourth layer is the rover library in which we have capsules corresponding to the different physical components of the rover along with a parent capsule that encapsulates the other capsules. The fifth layer is the logic for how the rover should behave which can be changed independently of the rover library.

4.4.2 Hardware

First, we have to decide what to use to control the rover. There are many single boards available that can be used to control the rover. However, the main two we consider are Arduino and Raspberry Pi. Arduino is a programmable controller. Arduino has many benefits such as support for Pulse Width Modulation (PWM) which would allow for variable motor speeds. The problem with Arduino is that it does not support
Linux which means it cannot run Papyrus-RT generated code. Raspberry Pi on the other hand does not support PWM but it runs Linux so we can compile and run code generated by Papyrus-RT. That is why, we settle for the Raspberry Pi as the brain for our rover. We choose to use Raspbian, a Linux distribution based on Debian that is created by the makers of the Raspberry Pi.

Figure 4.9: Rover Wiring Diagram

Figure 4.9 shows the wiring scheme for the rover. Component A is a Raspberry Pi 3, a credit-card-sized computer that we used as the controller for the rover. Component B is a battery case that holds the batteries which power the Raspberry Pi. Component C is a voltage regulator to ensure the Raspberry Pi is getting the 5 volts it needs to be powered on. Components D are motors that are attached to the wheels of the chassis. Component E is a motor controller that allows the Raspberry Pi to control the two motors in two different directions. Component F is the battery case that holds the
batteries that are used to power the motor controller and the motors. Component G is a distance sensor used to measure the distance in front of the rover.

4.4.3 Software

Figure 4.10 shows the elements of the rover model in the PapyrusRT model explorer. This includes the GPIO library, the rover library, and the logic for the rover's behaviour. The purpose of the rover library is to separate the capsules that control the rover from the logic which allows for interchangeable logic capsules which allows a different logic to be used with the same rover. It also allows for an interchangeable rover library which allows the same logic to be used with a different rover.

![Rover model as seen in the Papyrus-RT model explorer panel.](image)

Figure 4.10: Rover model as seen in the Papyrus-RT model explorer panel.

Figure 4.11 shows the top capsule of the rover. The top capsule connects the rover library to the control software capsule.

Figure 4.12 shows the capsule responsible for controlling the engine. It stays in the idle state until it receives a message telling it to either move forward or move
backward in which case it goes to the corresponding state and stays there until it receives the **stop** signal which takes it back to the **idle** state. When it enters the **MovingForward** or **MovingBackward** state it uses the rover library to communicate with the files system which in turn controls the hardware which moves the rover either forward or backward. Similarly, when the Engine capsule receives a messages telling it to move right or move left it goes to the **TurningRight** or **TurningLeft** states respectively. Once in one of those states it moves one of the motors which causes the rover to turn. It calculates the time it needs to turn based on the desired angle which is passed to it along with the message. It sets a timer to time out at that time which causes it to go to the **idle** state which stops the engine.

Figure 4.13 shows the capsule which interacts with the distance sensor. The distance sensor works by sending out a pulse of ultrasonic sound and timing how long it takes to get back. The distance sensor sends a signal for as long as the signal took to get back. To get the distance, we divide the time by two and multiply by the speed of sound. To simplify the implementation, we do this in C++ action code.

Figure 4.14 shows the capsule which interacts with the temperature sensor. The temperature sensor has a driver and a demo Python script that utilizes the driver to
4.4. ROVER

Figure 4.12: The Engine capsule in the rover model.

Figure 4.13: The capsule in the rover model responsible for communicating with the distance sensor.
print out the temperature and humidity. We use the demo Python script to get the data and parse its output.

Figure 4.14: The capsule in the rover model responsible for communicating with the temperature sensor.

Figure 4.15 shows the state machine of the capsule that contains the logic of the rover. The states in this state machine communicate with the rover’s components (the engine, distance sensor, and temperature sensor) through the top and the rover capsules. This capsule makes the rover move forward until it is within one foot of an obstacle then it moves back, turns, and goes back to moving forward and checking the distance.

Figure 4.16 shows the Engine capsule in the Rover library being with its Moving-Backward, MovingForward, TurningLeft, and TurningRight states being monitored.

Once we set these states to be monitored, we generate the code, transfer it to the Raspberry Pi on the Rover where we compile it, run it, and run LTTng so that it
sends the events as they happen to the development machine where the events are listed in an Eclipse view as shown in Figure 4.17.

When events arrive from the rover to the development machine, they are displayed in the Rover model as shown in Figure 4.18 where the MovingForward is highlighted in red to indicate that it is active.
4.4. ROVER

Figure 4.17: The Rover model’s trace results displayed in Eclipse.

Figure 4.18: The Engine capsule in the rover model with a state being active.
Chapter 5

Summary and Conclusions

5.1 Summary

The goal of this work is to maximize automation for modellers who are using UML-RT to develop embedded software and who want code to validate real-time constraints. We achieve this by modifying the code generator of Papyrus-RT, an open source Eclipse plugin that supports MDE in UML-RT.

We modify the code generator to include LTTng tracing statements in the generated code at places that correspond to certain events happening such as a state becoming active or a transition being taken. The generated code can then be compiled and traced using lttng which utilizes the tracing statements included by the code generator to provide insights about the code as it executes, including on the hardware it will eventually run on.

Once the execution is complete, the results of the trace are then parsed and displayed in Eclipse allowing the modeller to get information about the order in which the events occurred along with timing information about said events. If the modeller so chooses, they can display the information in Trace Compass by utilizing the XML
5.2. CHALLENGES

file the code generator creates to look at the tracing information visually and get a different perspective on their application.

5.2 Challenges

When this work started, Papyrus-RT was in version 0.6, a very early development version that was relatively unstable, bug-ridden, and feature-lacking. Since it is in early development, there is a lack of documentation and community around it. While in most situations, a developer in our situation might use Stack Overflow, that was not an option since there is no community around it yet due to it being new.

By the end of the work (just over a year later), Papyrus-RT was in version 0.8 and it had gone through 3 versions of Eclipse (Luna, Mars, and Neon). While this meant that Papyrus-RT was now a much more stable version, it forced us to adapt to a rapidly changing piece of software, changing our code base frequently slowed down development due to time spent adapting to changes made by the Papyrus-RT team that sometimes broke our code halting our progress for days on end. To reduce the effects of this, we tried to not upgrade often but sometimes we would report a bug and it would be fixed but to get the fix we would need to update thus forcing us to change our code again.

Understanding the semantics of UML-RT presented a challenge of its own. It is relatively simple to start learning and developing in UML-RT. However, in order to make the required changes, we need a deeper understanding in UML-RT because it guides the decisions we need to make. For example, deciding where to place the \texttt{tracepoint()} calls to know when a transition is taken we need to know everything that happens between when a message arrives and the transition is complete. While
it is possible to find one way of achieving a particular task and doing that without considering all the alternatives but that makes for bad research.

5.3 Limitations

As discussed in Subsection 2.6.1, the scope of this research is limited to the behavioural elements of UML-RT (states and transitions). While this is sufficient to show that what we are trying to achieve is doable to to support our hypothesis, it is nowhere near realizing the full potential of adding monitoring support for UML-RT models by modifying the code generator.

Eclipse functionality is supposed to be extended by creating plugins that inherit from other plugins if necessary and either replace their functionality or add to it. In order to make the changes needed to generate .tp files, we need to make modifications to the code generator that are not supported by the API. The StateMachineGenerator does not expose certain attributes that we need to access. Papyrus-RT developers are very careful about allowing developers to extend it and providing them with an official API because it is still in early development which means that they may need to change things in the future and they do not want developers to rely on APIs that might change. This means we cannot extend the code generator to make our changes which means that we need to modify the code generator’s source code. Modifying other Eclipse plugins’ source code is not how Eclipse’s functionality is supposed to be extended.
5.4 Future Work

We mention in Section 5.3 that one way to extend our work is to add support for monitoring other elements such as attribute value changes, message status updates, and other events shown in Figure 2.7. Monitoring these elements ought to be more challenging because not every event has a set position in the generated code where the `tracepoint()` call should be placed.

One area of focus for future research is dynamic monitoring. Dynamic monitoring refers to being able to enable and disable `tracepoint()` calls after the code has been generated, recompiled, or, even better, while the application is running without having to restart it. One way to achieve dynamic tracing without having to regenerate code is to generate all possible `tracepoint()` calls and providing a way to enable and disable them using some kind of input to the application such as a text file or command line arguments. One way to implement `tracepoint()` calls that can be enabled and disabled at runtime is to have the application communicate with a server every time a possible `tracepoint()` can occur to check whether it should activate that `tracepoint()` call or not. One disadvantage of this approach is that the communication with the server could slow down the performance of the application and invalidate the conclusions made from the results.
Bibliography


