Supporting Run-time Monitoring of UML-RT through Customizable Monitoring Configurations in PapyrusRT

by

Suchita Ganesan

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
September 2016

Copyright © Suchita Ganesan, 2016
Abstract

Model Driven Engineering uses the principle that code can automatically be generated from software models which would potentially save time and cost of development. By this methodology, a systems structure and behaviour can be expressed in more abstract, high level terms without some of the accidental complexity that the use of a general purpose language can bring. Models are the actual implementation of the system unlike in traditional software development where models are often used for documentation purposes only. However once the code is generated from the model, testing and debugging activities tend to happen on the code level and the model is not updated. We believe that monitoring on the model level could potentially facilitate quality assurance activities as the errors are detected in the early phase of development. In this thesis, we create a Monitoring Configuration for an open source model driven engineering tool called PapyrusRT in Eclipse. We support the run-time monitoring of UML-RT elements with a tracing tool called LTTng. We annotate the model with monitoring information to be used by the code generator for adding tracepoint statements for the corresponding elements. We provide the option of a timing specification to discover latency errors on the model. We validate the results by creating and tracing real time models in PapyrusRT.
Acknowledgments

It gives me great pleasure to express my deep gratitude to my supervisor Professor Dr. Juergen Dingel, for providing me the opportunity to study and do research under his invaluable guidance throughout my master’s degree. In addition to his motivation, patience and immense knowledge, I would like to thank him for his constant encouragement during my research work, the writing of this thesis and for the patient reviews thereafter.

My sincere and special thanks to my research lab mates who all provided me their support and motivation throughout this research. My lab has been a second home to me and it was a pleasure to work with my project partners Leo Jweda and Nondini Das right from taking Master’s courses together, project implementation, reviews and presenting demos at Montreal. I would like to thank Nicolas for his great help in writing the conference research paper and also his immense support during the project demos. I would also like to thank my Software Technology lab members Eric, Doug and Tuhin whom I have had the pleasure of socializing with and for making my research work experience fun and exciting.

I would like to extend my special thanks to my industrial sponsors for their support. Without them this work would not have been possible. Thanks to Ericsson Canada, Zeligsoft and LTTng for sponsoring and supporting this research work. A
huge thanks to Ernessto Posse who has been a big help to our research and used his valuable time to help us with any issues or difficulties that we faced.

Also I would like to put on record, Queen’s University as a whole has certainly been a contributing factor to the amazing times and experiences I have had here for the last two years. I thank one and all whom I came across and helped me here in my journey to Master’s degree.

Last but not the least, I am indebted to my parents for their continual support, for their belief in my abilities and for their encouragement to pursue this degree. My thanks are also due for all my friends and relatives who provided complete moral support to pursue my research ambitions.
# Contents

Abstract i

Acknowledgments ii

Contents iv

List of Tables vi

List of Figures vii

Chapter 1: Introduction 1

1.1 Motivation ......................................................... 3
1.2 Research Problem Statement ........................................ 4
1.3 Thesis Organization .............................................. 5

Chapter 2: Background 7

2.1 Real-time Software Systems ........................................ 7
2.2 Model Driven Development with UML ................................ 8
  2.2.1 Profile ......................................................... 9
  2.2.2 UML-RT ....................................................... 11
2.3 Model Driven Development Open Source Tools ......................... 16
  2.3.1 Eclipse IDE .................................................... 16
  2.3.2 Eclipse Plugin Development .................................... 17
  2.3.3 Papyrus ......................................................... 17
  2.3.4 PapyrusRT ..................................................... 17
2.4 Tracing Generated Code ........................................... 20
  2.4.1 LTTng .......................................................... 20
  2.4.2 Trace Compass ............................................... 23
2.5 Related Work ..................................................... 25
  2.5.1 Monitoring and Simulation for Real-Time Software Systems Using Different Approaches ........................................ 25
  2.5.2 Sequence Diagrams for Monitoring Real-Time UML Models ... 27
# List of Tables

2.1 Types of Ports [17] .................................................. 14

2.2 State Machine Major Elements [17] ............................... 15

2.3 Comparison of DEVS and UML-RT Frameworks [19] .......... 27
## List of Figures

1.1 Cost of fixing problems .............................................. 4
1.2 Traditional Software Monitoring (a) versus Model Monitoring (b) ... 5

2.1 Example of UML Profile [16] ........................................ 10
2.2 Capsule Diagrams ................................................... 12
2.3 Port types [17] ........................................................ 13
2.4 Protocol in UML-RT .................................................. 15
2.5 State Machine Diagram ............................................. 16
2.6 PapyrusRT window ................................................... 19
2.7 Tracing session [2] ..................................................... 21
2.8 Trace Compass view [6] ............................................. 24
2.9 Statechart diagram for brewer controller [19] ......................... 26
2.10 DEVS Model Simulation [19] .................................... 26
2.11 class diagram of PBX System [23] ................................ 28
2.12 Layers of runtime reflection framework for an application [24] .... 30

3.1 Implementation Work flow .......................................... 33
3.2 Trace Display ......................................................... 37
3.3 Monitoring Configuration Class Diagram .......................... 40
3.4 Behavioral Elements .................................................. 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Monitoring View Extension</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Monitoring View in Eclipse</td>
<td>45</td>
</tr>
<tr>
<td>3.7</td>
<td>Monitoring View zoomed in</td>
<td>45</td>
</tr>
<tr>
<td>3.8</td>
<td>Referencing metaclasses</td>
<td>46</td>
</tr>
<tr>
<td>3.9</td>
<td>LTTng Profile</td>
<td>47</td>
</tr>
<tr>
<td>3.10</td>
<td>Selecting the elements in PapyrusRT</td>
<td>49</td>
</tr>
<tr>
<td>3.11</td>
<td>Attaching LTTng Profile to the model upon element selection</td>
<td>50</td>
</tr>
<tr>
<td>3.12</td>
<td>UML-RT Profile in PapyrusRT</td>
<td>51</td>
</tr>
<tr>
<td>3.13</td>
<td>LTTng Stereotype applied to the state and boolean attribute set to true</td>
<td>52</td>
</tr>
<tr>
<td>3.14</td>
<td>Selecting the element pair in PapyrusRT</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Ping Pong Structure Diagram</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Pinger State Machine Diagram</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>Ponger State Machine Diagram</td>
<td>58</td>
</tr>
<tr>
<td>4.4</td>
<td>Monitored elements are shown in purple</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>Widget Production Top Capsule Structure Diagram</td>
<td>60</td>
</tr>
<tr>
<td>4.6</td>
<td>Widget Production Control Software state machine diagram</td>
<td>60</td>
</tr>
<tr>
<td>4.7</td>
<td>Widget Production Workstation and Robot state machine diagrams</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>States are monitored</td>
<td>62</td>
</tr>
<tr>
<td>4.9</td>
<td>Rover Architecture</td>
<td>63</td>
</tr>
<tr>
<td>4.10</td>
<td>Raspberry Pi 3 GPIO diagram [4]</td>
<td>64</td>
</tr>
<tr>
<td>4.11</td>
<td>Rover Diagram [4]</td>
<td>65</td>
</tr>
<tr>
<td>4.12</td>
<td>Rover Top Capsule structure diagram</td>
<td>66</td>
</tr>
<tr>
<td>4.13</td>
<td>Rover Engine Controller state machine diagram</td>
<td>66</td>
</tr>
<tr>
<td>4.14</td>
<td>Rover Detection state machine diagram</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

With the advancements in hardware and software, computers have no doubt changed the way we live. This rapid advancement has put pressure on software developers to deliver highly complex software efficiently. Modern software development is a complex process which typically involves several people working on individual parts of the entire project [12]. One of the reasons behind the difficulty in developing complex software is the large conceptual gap between the requirements and the implementation. In order to bridge this gap, the Model Driven Engineering (MDE) approach uses models or more abstract representations of the software. Models can describe complex systems at multiple levels of abstraction and from a variety of perspectives [14]. Although the use of models in traditional engineering disciplines has generally proven to be useful, it is still not widely accepted in the case of software engineering [22]. MDE is a software engineering discipline in which models are central artifacts of development which are used to construct the system, communicate design decisions, and generate other design artifacts [26]. MDE is based on three main principles:

1. increasing abstraction through the use of models, e.g., in the form of UML diagrams,
2. leveraging automation via code generation from models,

3. improving analysis capabilities through executable models.

The main goal of Model Driven Engineering is to increase developer productivity by allowing them to describe the system through artifacts (models) that are more high-level and easier to use than code. It is centered on the construction and analysis of formal models of the software to be built. The software developers are shielded from the complexities of the underlying platform. Code becomes secondary because it is generated automatically from these models. This approach can lead to increased productivity by simplifying the process of design.

Models are artifacts expressed on a higher level of abstraction than code and often expressed in some custom-made domain specific language. Combined with a high degree of automation (e.g., to generate code from models), this development style can work well in many domain areas such as telecommunications, automotive, aerospace and business. It is effective when the models are clear from the point of view of a user who is familiar with the domain, and if they can serve as a basis for implementation of the software. It can help reduce errors while developing complex software.

Many proven MDE methods and tools have evolved over time. In the 1980s, CASE tools (computer aided software engineering tools) were introduced which focused on providing the developers with methods and tools to depict software systems using a graphical general purpose language representation. Different tasks involving these representations such as correctness analysis, transformations to and from code were enabled for the developer. However these tools were not successful due to reasons such as [10]:

1.1. MOTIVATION

1. the inability to scale since the tools did not support concurrent engineering (method of designing and developing products in which different stages run simultaneously)

2. generated code was harder to understand and maintain due to poor mapping of general-purpose languages onto the underlying platforms

3. models were merely used for documentation purposes and focus was more on coding in the implementation phase

The advancement of languages and platforms in the past years has raised the level of software abstraction hence alleviating one impediment to earlier CASE efforts [29]. At commercial and research levels several tools are available or in development.

1.1 Motivation

Software is always prone to some errors. When an error occurs in the requirements phase and is discovered during the deployment phase then the cost of fixing the error is much higher than if the error was detected during the requirements phase. Studies have shown that reworking a software requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the problem in the requirements stage [5]. Figure 1.1 represents an informal illustration of how the cost of fixing a bug goes up, the longer it goes unnoticed. The cost of fixing a bug can become exponential if the error had occurred in the requirements phase of development [1].

The first step in software development is the requirements phase in which the developer collects the requirements and then proceeds with designing the system.
Proper analysis leads to proper design and therefore ensures that the software developed through the phases will meet the user requirements. In the MDE approach the developer designs models from which code is automatically generated. However, once the code is generated, the validation part tends to be done on the code rather than validating the model. Analysis of models can help detect bugs earlier in development and thus can help reduce cost. Current open source modeling tools such as IBM RSA-RTE and IBM Rhapsody lack an integrated solution for model based monitoring.

1.2 Research Problem Statement

As depicted in Figure 1.2, in traditional software development, the design phase merely makes use of UML diagrams for documentation and design specifications. The coding phase tends to be of greatest importance. The developer focuses on the code that is developed from the models. Testing, tracing and debugging activities are performed on the code and the design specifications are not updated when changes are made to the code. Instead of this approach, we propose a monitoring mechanism.
1.3. THESIS ORGANIZATION

in which the executable models are monitored and traced for debugging purposes. This can ensure that the developer focuses on the models as they are the actual implementation of the system. As a result, the design and code will be in sync.

We use the Eclipse plugin PapyrusRT for real-time modeling and LTTng for tracing [3, 2]. The main goal of this research is to provide the support for monitoring UML-RT models in PapyrusRT. We allow the specification of a monitoring configuration for PapyrusRT which describes which events can be monitored in the real time model. We provide a timing constraint option for the user to be able to input a timing specification which is later validated. This is used for detecting latency errors in a real time model.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 discusses the literature related to this research. Chapter 3 discusses the concepts and implementation details for adding
support for monitoring configurations to PapyrusRT. Chapter 4 presents a set of case studies to validate the support for monitoring of UML-RT models. Chapter 5 concludes the thesis, describes the limitations of the thesis and presents ideas for future work.
Chapter 2

Background

This research is in the field of Model Driven Engineering for software and also deals with the importance of run-time monitoring of UML-RT models. This chapter discusses the literature related to the thesis and is organized as follows: the description of real-time embedded systems and their development is presented in Sections 2.1 and 2.2, respectively, the state of the art tools for creating models and tracing code are described in Sections 2.3 and 2.4, respectively and Section 2.5 defines the source of related work that has been done in the field.

2.1 Real-time Software Systems

Real-time software systems possess a common characteristic, which is timeliness; that is, the ability to respond properly to inputs within correct time intervals. These systems are most commonly used in telecommunications, aerospace, defense and automotive industries where software tends to be large and complex. The correctness of the system behavior depends not only on the logical results of the computations but also on the physical time when these results are produced. The sequence of outputs during the execution of a system is what is known as system behavior. A real-time
2.2. MODEL DRIVEN DEVELOPMENT WITH UML

Computer system is part of a larger system. It must react to stimuli from its environment within time intervals enforced by its environment. The instance when the output must be produced is called the deadline. If the output is of use even after a deadline has passed then the system is a soft real-time system. Otherwise if the output is not produced before the deadline which causes severe consequences, then the system is a hard real-time system. It is essential for such systems to have a well-defined architecture and also make sure they are fault tolerant [30, 20].

A good example of a real-time system is an engine controller in an automobile engine. The task of the engine controller is to calculate the proper amount of fuel and the exact time at which the fuel must be injected into the combustion chamber of each cylinder. The timing and amount of fuel is determined by various parameters such as the intentions of the driver, current load on the engine, the condition of the cylinders and so on. A sensor is present to indicate when the valve is opened in the cylinder to compensate for any latency issues. This example demonstrates the need for highly precise temporal control. Due to the increasing complexity of software, there is a need to apply advanced design methods and technologies which could simplify and enhance the reliability of real-time software design and implementation [22].

2.2 Model Driven Development with UML

Graphical design notations have proven effective in communicating and understanding an idea whether it is a software system or business process. Within the object oriented development community, the Unified Modeling Language (UML) has become a standardized graphical notation. MDE is centered on the development and use of the popular modeling language UML. Several modeling tools support UML and many
developers are familiar with the language. The syntax of every model is defined by its metamodel [13]. UML consists of a set of diagrams for representing two different views of a system model which are:

1. Structural or Static View: This view emphasizes the structure of a system through the use of class diagrams and composite structure diagrams. It models the system by using objects, attributes, behaviors and relationships.

2. Behavioral or Dynamic View: This view emphasizes the behavior of the system through the use of sequence diagrams, activity diagrams and state machine diagrams. It models the system by showing collaborations among objects.

2.2.1 Profile

One of the main challenges of any unified modeling language is dealing with the diversity existing across the different application domains [22]. In order to support usage of UML model elements in a particular modeling context, UML profiles are used. They describe how UML model elements can be extended in order to suit the needs of a particular domain. In a profile, UML model elements are extended using stereotypes that define additional element properties [15]. When we need to define a new language that either should add constraints or restrict the number of UML elements we don’t need to develop a new language from scratch. A UML profile is a packaged stereotype which can be created by either extending a metamodel or another profile. It is defined by three basic mechanisms: stereotypes, constraints and tagged values [16].

In order to illustrate the concepts we explain an example as illustrated in Figure 2.1. Suppose we want to add weights and colours to our UML models. First two
Figure 2.1: Example of UML Profile [16]

stereotypes are defined —Coloured and Weighed— which indicates that both UML classes and associations can be coloured. The metamodel elements are indicated by classes stereotyped «metaclass». Extensions are indicated by the arrow pointing to the extended class. Constraints can be associated with stereotypes that impose restrictions on the corresponding metamodel elements. These constraints can be expressed in any language such as the Object Constraint Language (OCL) to enforce properties creating a well-formed model. Tagged values are additional meta-attributes that are attached to a metaclass of the metamodel extended by a profile. In the example the stereotype «Weighed» has an associated tagged value named “weight” of type Integer [16]. A UML profile is defined using a set of these extension mechanisms and grouped into a UML package stereotyped «profile» [16].
2.2. MODEL DRIVEN DEVELOPMENT WITH UML

2.2.2 UML-RT

Specifying the design of a distributed system is a difficult task which involves data, behaviour, intercommunication and architectural aspects of the model. UML and ROOM (Real-Time Object-Oriented Modeling language) have been combined into UML for Real-Time (UML-RT). By the use of stereotype mechanisms, UML-RT introduces four new constructors which are capsule, protocol, port and connector [27, 28].

UML-RT Elements

Capsule

The real-time system is modeled as multiple communicating active objects called capsules. A capsule is an active class which has attributes, methods, and to model its dynamic behaviour it has a state machine. Figure 2.2 depicts a simple capsule which has an attribute and three different ports. A capsule can communicate with the outside world using ports. Implementation of a capsule can be seen as a mix of C++ classes and active capsules. A capsule has two types of diagrams.

1. State Machine Diagram: The behaviour of a capsule is modeled as a traversal of a graph of states connected with transitions as seen in Figure 2.2 (b). The state machine formalism is an object based variant of Harel statecharts. Harel statecharts are expressive diagrams that can describe complex behaviour through a finite collection of states and transitions. They are convenient for describing reactive systems since they deal with the notions of hierarchy, concurrency and communication [18].
2.2. MODEL DRIVEN DEVELOPMENT WITH UML

2. Structure Diagram: This is similar to a collaboration diagram which is used for specifying the pattern of communication with other capsules. This is depicted in Figure 2.2 (a).

Port

Ports are objects whose purpose is to act as boundary objects for a capsule instance [28]. Capsules can only communicate by sending messages through their ports. Ports are created and destroyed by capsules so they have a composition relationship with capsules in the metamodel of UML RT. Ports can communicate with one another only if they are defined by a common protocol or by two different protocols having
the same definition. The different types of ports are listed below in Table 2.1 and are depicted in Figure 2.3.

![Figure 2.3: Port types](image)

**Protocol**

Protocols define the communication rules between capsules. It is a specification of a set of messages received (in) and sent (out) from the port. It is a stereotype of a UML collaboration. Each capsule role typically has an associated protocol for every other capsule role with which it associates. Figure 2.4 depicts an example of a protocol in UML-RT. The “RobotProtocol” consists of three signals through which the “Control Software” and “Robot” can communicate with each other through the ports “CS2R” and “R2CS”, respectively.

**State Machine**

System behaviour is modeled through the state machines of the capsules. There can be at most one state machine per capsule. As a state machine grows larger it
Table 2.1: Types of Ports [17]

becomes important to be able to decompose states using sub-state machines. States
which contain a sub-state machine are called composite states, and a state machine
with composite states is a hierarchical state machine [28].

Table 2.2 describes the major elements of state machines and their roles.

An example of a state machine diagram is depicted in Figure 2.5 which contains
four states and transitions between the states. Transitions define the relationship
between the source and destination state. The triggers for the transitions between the
states are defined by the protocol. All state machine formalisms universally assume
that a state machine completes processing of each event before it can start processing
the next event which is known as “run to completion semantics”. In Figure 2.5, the
2.2. MODEL DRIVEN DEVELOPMENT WITH UML

Figure 2.4: Protocol in UML-RT

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>A state is a particular condition that a system is in at a specific time. Every state machine at least has one state which is the initial state. A state can can have sub states.</td>
</tr>
<tr>
<td>Transition</td>
<td>A transition is a directed relationship between a source vertex and a target vertex. In a state machine it is the event which makes a state move from one to another.</td>
</tr>
<tr>
<td>Guard</td>
<td>A guard is a constraint that provides a fine-grained control over the firing of the transition.</td>
</tr>
<tr>
<td>Choice Point</td>
<td>A choice point is a dynamic conditional branch where a condition is evaluated and based on the output the respective path is taken.</td>
</tr>
<tr>
<td>Trigger</td>
<td>A trigger is the event that causes the transitions to be taken.</td>
</tr>
</tbody>
</table>

Table 2.2: State Machine Major Elements [17]

‘startUp’ state changes to the ‘produce’ state when the ‘start’ transition is triggered.
The ‘produce’ state changes to the ‘deliver’ state when the ‘deliverMe’ transition is
triggered. The ‘deliver’ state changes towards the ‘produce’ state when the ‘goAgain’ transition is triggered. This transitioning between ‘produce’ and ‘deliver’ occur until the ‘stop’ transition is triggered. The capsule goes to the ‘shutDown’ state as soon as the ‘stop’ transition is triggered. This transition is based on a timing port and is triggered after a finite amount of time.

![State Machine Diagram](figure2.5.png)

**Figure 2.5: State Machine Diagram**

### 2.3 Model Driven Development Open Source Tools

Due to the advancements in design and development for real-time embedded systems using MDE, the need for providing an efficient tooling support is quite a challenge. The development of large collaborative environments such as the Eclipse ecosystem, has made it easier to reuse and build tools [21].

#### 2.3.1 Eclipse IDE

Eclipse is an open source software project that provides a highly integrated tool platform. The work in Eclipse includes a generic framework for tool integration and
2.3. MODEL DRIVEN DEVELOPMENT OPEN SOURCE TOOLS

a Java development environment built using it. The development work in Eclipse is divided into numerous top-level projects which include the Eclipse Modeling Project which promotes model-based development technologies [31].

2.3.2 Eclipse Plugin Development

In Eclipse, the basic unit of function, or a component is called a plug-in. The Eclipse platform and tools that extend it are collections of plugins. Each plugin contributes some functionality that is either invoked by the user or extended by other plug-ins [31].

2.3.3 Papyrus

Papyrus is an open source general purpose UML2 graphical modeler based on the Eclipse environment. It provides code generation (C, C++, Java) establishing it as an MDE tool. It leverages various Eclipse software components to provide an efficient graphical editor for UML2 [21]. It currently supports the fundamental UML diagrams. As mentioned before, profiles are a key feature of UML that allow a user to adapt the language to a specific domain or process. Papyrus supports the creation of new profiles in order to define new modeling concepts which makes Papyrus highly customizable.

2.3.4 PapyrusRT

PapyrusRT is the real time version of Papyrus. It is an industrial tool which provides a complete development environment for developing real-time, distributed, reactive or embedded systems. It supports the concerns related to designing systems such as modeling, code generation, run-time support, schedulability, advanced debugging
facilities and performance analysis. It is an open source tool designed to support extension. It also provides specific perspectives which facilitate the development of UML-RT models as well the validation of these models to ensure they are well-formed.

PapyrusRT has an built-in code generator that translates the UML-RT model into C++ code. The structural elements such as capsules, protocols and classes are translated into C++ code based on the stereotypes applied to the elements. The behavioral elements are translated into an intermediate model that is then translated into optimized code. Based on user-provided thread allocation information, the code generator will also allocate capsules, or groups of capsules, to threads. A make file is provided to compile the code and link it with the UML-RT run time library, and any other external libraries used by the application [3].
2.3. MODEL DRIVEN DEVELOPMENT OPEN SOURCE TOOLS

Figure 2.6: PapyrusRT window
Figure 2.6 shows the PapyrusRT project window in Eclipse. In Eclipse there are several views and editors which can be created for specific purposes. View 1 in the figure is the project explorer view to import or create the PapyrusRT projects in the workspace. Eclipse offers different perspectives which change the Active Editor window according to the perspective. View 2 in the figure shows the multi editor view and displays the selected project diagrams. Currently in the figure the Pinger state machine is displayed. View 3 displays the palette from which the UML-RT elements can be created in the model. View 4 is the model explorer view which is used to navigate to all model elements and the diagrams. View 5 is the properties view which shows the properties of the selected elements either from the multi editor view or from the model explorer view.

2.4 Tracing Generated Code

Software tracing is a useful way to record information about program execution in order to diagnose problems and improve performance. Tracing is suitable for applications in which several processes interact with the operating system. In comparison to logging, tracing typically records events that occur frequently. Good tracing software must be optimized so that the impact of tracing performance is minimized.

2.4.1 LTTng

Linux Trace Toolkit Next Generation (LTTng), is an open source tracing tool which allows tracing of kernel and user space applications as well as viewing, analyzing and live streaming of traces. LTTng is efficient since it has less overhead on production systems by using per-CPU buffering, read-copy-update (RCU) data structures and
an efficient binary format which is the Common Trace Format (CTF). It is flexible and easy to use as it collects events from the kernel as and when they happen. It is possible to trace user space applications or define your own tracepoints. LTtng offers local and remote tracing of small and large systems [2]. Figure 2.7 describes a tracing session. Everything happens in the scope of a tracing session. The core concepts of tracing using LTtng are discussed below.

**Tracing Session**

It is a container of state and everything happens in the scope of the tracing session just like a bank website session. The traces generated during a session may be sent over a network or saved to disk.
2.4. TRACING GENERATED CODE

Domain

The domain is used to appoint a tracer category. There are several domains provided by LTTng such as Linux kernel, user space, java.util.logging, log4j and Python. Depending on the user’s objective, a specific type of tracer must be targeted. For example, in the case of using kernel and user space tracers, both support named tracepoints. While enabling an event the type of tracer must be specified since both domains might have existing events with the same name. Certain features are supported for some domains like for example the dynamic function entry/return instrumentation is supported only for the kernel domain. In our research work we make use of the user space domain to define tracepoints in the PapyrusRT code.

Channel

A channel is given by a set of events with specific parameters and added content information. Within a tracing session, channels have unique names. Any given event is always registered with one channel. A channel may be enabled or disabled at any given point. Channels are implemented using a shared ring buffer where events are eventually recorded by a tracer and consumed by a consumer daemon.

Event

An LTTng event can be defined in multiple ways depending on the context of its use. During tracing, the event is a point in space-time. When a processor executes a program and an instruction point or probe is encountered then an event is said to occur. An event is always registered to one or more channels and may be enabled or disabled per channel as required by the user [2].
2.4.2 Trace Compass

Trace Compass is an open source Java tool integrated with LTTng, for viewing traces generated by LTTng. It extracts useful information from the traces and represents them in the form of graphs, views or metrics. Trace Compass provides the support for large traces, a built-in CTF parser to parse the traces (binary files), efficient searching and filtering of events. Trace Compass is available as an Eclipse plugin as well as a stand alone RCP application.

Figure 2.8 shows the Trace Compass view in Eclipse which displays the traces from a selected trace folder. The information collected from the trace as explained before consists mainly of a timestamp (when the event occurred), channel, event type and contents. Contents are the fields or payloads of an event. From the trace information a histogram is drawn based on this information and shown in the Trace Compass view. The histogram view displays the event density with respect to time in traces. The benefit of Trace Compass is that it has built in trace analyzers and viewers which derive useful information from the raw data along with knowledge of the traced program.
Figure 2.8: Trace Compass view [6]
2.5. RELATED WORK

2.5 Related Work

In this section, the previous efforts to monitor UML-RT models in IBM Rational RoseRT are explained. Also the significance of sequence diagrams is explained in the context of monitoring real time models.

2.5.1 Monitoring and Simulation for Real-Time Software Systems Using Different Approaches

In the work of Dongping et al., two existing simulation approaches used for real time software are described and compared. Models can be described in a system-theoretical way or by using object oriented languages for designing software and simulation modeling. Discrete-Event System Specification (DEVS) is a system-theoretic modeling approach which can be used to develop simulation models that exhibit concurrent and distributed behaviour. Similarly, UML-RT can be used to develop software models that can be implemented and executed. These software models are not quite suitable for the purpose of simulation modeling and the simulation models may not be adequate enough to serve as software design blueprints which is the importance of their work to compare both approaches [19].

The authors explain the differences by developing a coffee machine example with a focus on the treatment of logical and physical time. The coffee machine software controls the hardware operations. Scenarios such as brewing coffee are modeled and graphical user input is used to simulate the behaviour of the Coffee Machine. The modeling of the brewer controller behaviour using the UML-RT approach is shown in Figure 2.9. In UML-RT the trigger events in a capsule’s statechart are defined with a port and signal pair. UML-RT provides run-to-completion semantics for a statechart.
to process a single event at a time. UML-RT thus supports concurrency for executing multiple capsules simultaneously however cannot guarantee processing of events using
2.5. RELATED WORK

priority settings. Figure 2.10 (a) describes the external transition function for the CM Brewer. Figure 2.10 (b) describes the structural model of the CM Controller with the specifications of the input and output ports as DEVS supports communication among multiple ports. Due to the fan-in and fan-out couplings, the same messages which are sent from one or two atomic models need to be identified based on their senders. This process is not required in UML-RT since there is no constraint on port communication as long as input messages go to input ports and output messages go to output ports [19]. Table 2.3 summarizes the most important differences in both approaches.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>DEVS</th>
<th>UML-RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling Framework</td>
<td>Based on systems and theory; provides multiple levels of specifications and abstractions</td>
<td>Based on object-orientation; provides different kinds of model abstractions</td>
</tr>
<tr>
<td>Behavior Specification</td>
<td>Model’s behavior specified by internal/external transitions, output and time advanced functions</td>
<td>Statechart is used to specify behavior of capsules</td>
</tr>
<tr>
<td>Timing and transition</td>
<td>Provides timing for atomic models as well as a global time. Transition may take zero to infinity.</td>
<td>Transition time is negligible but must be greater than zero</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison of DEVS and UML-RT Frameworks [19]

2.5.2 Sequence Diagrams for Monitoring Real-Time UML Models

The work of Marc Lettari et al., describe how sequence diagrams can be used for both monitoring and testing functional and real-time requirements of an executable UML design [23]. The authors demonstrate their work using a PBX telephone system as an example test case. The PBX system consists of a Telephone, Line, Connection and
2.5. RELATED WORK

Figure 2.11: class diagram of PBX System [23]

Call Router as seen in Figure 2.11. The authors believe that sequence diagrams are well suited for testing because they can be used to capture relevant use cases of the desired system. The graphical nature of sequence diagrams is useful to communicate the requirements between users and developers and also real-time properties can be expressed easily. The authors propose an additional feature which is the capability to specify when a scenario described in a sequence diagram should be activated. This capability is useful to determine when the system is to be monitored or to generate certain inputs [23].

2.5.3 Model Based Traces

The work of Shahrar Maoz et al., presents model-based traces which trace behavioral models of a system’s design during execution time [25]. The authors suggest that sequence diagrams and statecharts can be used as a higher level of abstraction over execution traces. For a given program P and a behavioral model M there can be a
model-based execution trace which records a run $r$ of $P$ at the level of abstraction induced by $M$. The trace generation is advantageous in the sense that the monitored code is automatically generated from the models and the code of the system under investigation is obvious to the models that are watching it. However their approach suffers from higher overhead and poor scalability in terms of trace length and model size. The authors propose model-based runtime configurations as an analysis method to compare traces of different runs of the same system. They make use of Tracer which is a prototype tool for the visualisation of model-based traces [25].

2.5.4 Adaptive User Interfaces

The work of Marco Blumendorf et al., describes how models can be utilized at run time and a close interconnection with the running system can be achieved [8]. Executable models make it possible to build systems which are aware of context information. The authors demonstrated the approach by building adaptive user interfaces through the integration of a context model with a set of additional user interface models. The context model provides means for reasoning about the current situation of the application and user at run time. In order to build a context model which reflects the state of the environment at run time the connection of the model with real world context sensors and actors available in the environment are required. The task model reflects the state of a system on a higher level of abstraction which helps to keep track of the user interaction on a higher level of abstraction. The context model can be connected with the task model to create a context-adaptive application [8].
Figure 2.12: Layers of runtime reflection framework for an application [24]

2.5.5 **Runtime Verification**

Software failure can be defined as a deviation between the observed behaviour and the required behaviour of the software system [24]. Software verification is comprised of all techniques suitable to show that a system satisfies its specification such as through model checking or theorem proving. More specifically runtime verification deals with the techniques that allow to check whether the run of a system satisfies a given correctness property. In the work of Martin Leucker et al., it is described how a system can be understood as a possibly infinite sequence of the system’s states which are formed by variable assignments or sequence of actions that a system is currently emitting [24]. To check whether an execution meets a correctness property a monitor is typically used. A monitor is a device that reads a finite trace and produces a certain verdict. Working on a current execution of a system is defined as online monitoring and working on a recorded execution of a system is defined as offline monitoring. The development of reliable systems can potentially be achieved using a monitoring layer followed by a diagnosis layer and mitigation layer as depicted
in Figure 2.12. The logging layer records the system events and provides them in a suitable format to the monitoring layer. The monitoring layer detects the fault without affecting the behaviour of the system. The diagnosis layer identifies the failures and the mitigation layer is then used to reconfigure the system to mitigate the failures. Typically models reflect the most important aspects of the corresponding implementation, hence checking the model correctness gives useful insights to the implementation [24]. In the case of safety critical systems this approach is useful to monitor behaviour to double check that everything goes well.

Our implementation work differs from the related works as we monitor real-time systems using a tracing tool. Our tool environment is completely open source. As explained in the related works, the model level checking is essential as making changes on the model level would automatically reflect on code level. We trace the generated code and integrate these traces on the model level. These traces would be valuable for validation and debugging purposes. We provide a monitoring configuration with a set of valid events that can be monitored.
Chapter 3

Monitoring Configuration

In this chapter, the Monitoring Configuration is explained in detail. The overall vision of monitoring UML-RT elements with LTTng is explained in Section 3.1. The elements of the Monitoring Configuration are described in Section 3.2. The implementation details are elaborated in Section 3.3.

3.1 Monitoring and Simulation for Quality Assurance

Real-time software can be large and complex with respect to software design, development, testing and operation [11]. Not being able to identify design flaws that cause problems until late in the software development life cycle is a significant contributor to inflated expense and delay [29]. Monitoring and simulation are important activities as they can help designers validate model quality under different operating conditions. MDE aims to facilitate development by providing high-level, abstract views of the system. Since code is generated automatically from these models, monitoring and debugging should occur on the model level which will help developers detect and rectify errors in the early phases of development. Moreover, developers create these models and thus are familiar with them. So it is natural to want to present information about
3.1. MONITORING AND SIMULATION FOR QUALITY ASSURANCE

system executions in terms of these models.

Observation could aim at high-level correctness properties and low-level resource consumption properties. Tracing the code generated from the model and displaying the traces on the model level in a suitable way for the developers to identify errors can be helpful during development of real-time systems. For example the monitoring and simulation of an ATM controller to determine if it returns the bank card in less than two seconds after three failed PIN input attempts can be of practical use.

Figure 3.1: Implementation Work flow

Figure 3.1 represents the work flow of the entire run-time monitoring system. Once the user creates the UML-RT model in PapyrusRT, the user can select desirable elements to be monitored through the Monitoring Configuration UI and the model is annotated with selected monitoring information. Using the PapyrusRT code generator which has been extended to support tracing with LTTng, the code is generated with the necessary trace point files. Then the code is traced using LTTng to generate
3.1. MONITORING AND SIMULATION FOR QUALITY ASSURANCE

trace files. Finally, the generated trace files are displayed on the model level. Support for live tracing of the model has also been implemented which shows a live step of the behavioral components in the real time model.

3.1.1 Monitoring Configuration

The code generated from models in PapyrusRT has to interact with LTTng in order for it to be traceable. The first step to monitor the models in PapyrusRT is for the user to specify which events of the model should be monitored. The monitoring configuration is a collection of events that should be monitored on the model. These events, in turn, affect the way the code generator works in order to produce code that allows LTTng to observe the selected events.

3.1.2 Code Generation with LTTng trace points

The PapyrusRT code generator was designed so that it can easily be extended. This feature has been used to extend the code generator to support tracing with LTTng. LTTng is suitable for monitoring to validate timing constraints since it provides accurate event time stamps in a minimally disruptive fashion and allows for both local and remote monitoring [10]. Local monitoring has lower overhead which makes it more suitable for timing analyses.

The code generated by the customized code generator contains the trace point information required by LTTng. The input for the code generator is from the Monitoring Configuration UI which notifies the code generator which elements have been selected for monitoring. The code is then compiled and results into an executable user-space application. A script which contains all LTTng commands necessary for
tracing is contained in the customized code generator. This script is invoked to monitor the user-space application. This can be used for tracing the user-space application many times. The implementation provides fully automatic support for code generation, execution of LTTng on the target platform for the production of traces and display of these traces in PapyrusRT.

Listing 3.1: Sample code generated

```cpp
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 ;
                }
                return currentState ;
                default :
                    this->unexpectedMessage() ;
                    break ;
            }
            return currentState ;
}
```

Listing 3.1 depicts sample code generated by the modified PapyrusRT code generator. The tracepoints are added to the elements that have been selected for monitoring which are seen in lines 3 and 10 in the code. In this example the state “Workstation__Producing” has been selected from the Monitoring Configuration UI.
3.1. MONITORING AND SIMULATION FOR QUALITY ASSURANCE

for monitoring.

3.1.3 Tracing Using LTTng and Trace Display

The traces generated by LTTng are displayed in the UML-RT model in PapyrusRT in a user friendly manner both textually and graphically. This is done by creating an Eclipse plugin with a view. The Figure 3.2 depicts the Eclipse view created to show the traces. It consists of three columns showing trace files, trace details and four buttons for user interaction. The plugin allows the user to select the model and upon selection the list of trace files associated with the model are displayed. The Trace Compass CTF parser is used for parsing all the traces. The UML-RT model in PapyrusRT can also be traced live, where traces are displayed in the view as and when they are generated. The UML-RT model is animated to display the flow of events and thus the final step of monitoring and simulation is achieved in PapyrusRT. The “Display Trace” button highlights the respective trace in the model in red. The “Step” button steps through the events in the trace by highlighting the appropriate elements in the model. For example in the figure the active state is highlighted in red color. The “Start Live Trace” button enables the live tracing feature in which the traces are highlighted in the model as and when they are received.
3.1. MONITORING AND SIMULATION FOR QUALITY ASSURANCE

Figure 3.2: Trace Display
3.2 Monitoring Configuration Description

The monitoring configuration is a collection of events that should be monitored in a UML-RT model for PapyrusRT. It describes the monitored events which the code generator will read. This impacts the shape of the code generated from the model.

3.2.1 Class Diagram

A class diagram describing monitoring configurations is depicted in Figure 3.3. It is a collection of events that are interesting to monitor. These events can be classified into four categories: Communication, Attribute, Capsule and State Machine events. On a broader level they can be viewed as Structural Elements and Behavioural Elements.

The Communication events can be of two types namely Queue and Message. Capsules are active objects with their own logical thread of control typically representing an active unit of computation. They communicate with other capsules through buffered asynchronous message passing. The real time model of UML-RT follows the run to completion semantics for each capsule. The capsule must execute the triggered action to completion as soon as it is triggered at its input port before processing the next message. Messages can be assigned priorities and queued in priority order. The event ‘SentE’ indicates when messages are sent to and from capsules and that can have several sub events. Once the message has been sent it must be delivered to the receiver capsule which is captured by the ‘DeliveredE’ event. After that the message is handled or if it does not reach the receiver capsule it is dropped which correspond to the ‘DeliveredE’ and ‘DroppedE’ events respectively. These communication events can be helpful to monitor and figure out any possible delays in communication between capsules. It can also help in schedulability analysis.
The Attribute Events can be used to monitor the initialization (InitializedE) and when the attribute is updated (ChangedE), i.e., the value of an attribute changes.

The capsule events can be used to monitor when the capsule instances are created or destroyed (DestroyedE) and also to which they are bound (CreatedE).

They consist of states and transitions. The state machine events can be monitored in UML-RT for detecting errors for, e.g., delay between creation of two capsule instances could be measured. We focus on when the state is currently active in the model and also when a transition is triggered.

The code generator is influenced by the monitoring configuration and is notified of the relevant events that need to be monitored. The presence of an element in the configuration would impact the code generated from the model and may also cause the generation of additional artifacts the element requires. The annotated model with the monitoring information is used to produce code that is ready for monitoring with LTTng.
3.2. MONITORING CONFIGURATION DESCRIPTION

Figure 3.3: Monitoring Configuration Class Diagram
For the implementation of this research work we have worked on monitoring the Behavioral elements for real time models in PapyrusRT as depicted in Figure 3.4. All capsules have an associated state machine in UML-RT. State machine diagrams consist mainly of states and transitions. Choice points are also present, however they currently cannot be monitored. A state represents a moment during the life of an object where it is ready to process certain events. Transitions show the relationship between a source and a destination state. They specify the conditions under which an object in the source state will change to the destination state. Pseudostates in PapyrusRT represent the initial state which is explicitly defined at the beginning of the state machine. According to our definition of the monitoring configuration we monitor when the selected states are active and the transitions are triggered. We monitor the elements by using LTTng tracepoints inserted by the code generator at the appropriate places. The elements are selected by the user and the real time model is annotated with this information. The code generator is notified of the StateMachine
3.3 Monitoring Configuration Implementation

This section describes the implementation details of the Monitoring Configuration in PapyrusRT.

3.3.1 Eclipse Views

An Eclipse application consists of several plug-ins. We create a plug-in in order to add our monitoring feature and extend the capabilities of PapyrusRT. For our implementation of the Monitoring Configuration UI we make use of the Eclipse View which is a visual component within the workbench. Usually Eclipse views are used to navigate a list of information or display properties for the active editor. Any modification made in the view is saved immediately. The extension point org.eclipse.ui.views allows plug-ins to add views to the workbench. Any plug-in that contributes a view must register the view in their plugin.xml. The plug-in should provide information about the view, such as implementation class, the category of its view and the name of the view.

Listing 3.2: Monitoring Configuration UI plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>

<extension point="org.eclipse.ui.views">
    <category
        name="Monitoring_Category"
        id="MONITORING_VIEW">
    </category>

    <view
        name="Monitoring_View"
```
3.3. MONITORING CONFIGURATION IMPLEMENTATION

Listing 3.2 describes the Monitoring configuration UI plugin.xml extensions. The category is the “Monitoring Category” and the name of the view is “Monitoring View” as shown in Figure 3.5. While creating an Eclipse plug-in view the default extensions...
3.3. MONITORING CONFIGURATION IMPLEMENTATION

used are:

- org.eclipse.ui.views
- org.eclipse.ui.perspectiveExtensions
- org.eclipse.help.contexts

In addition to these extensions, we also use the PapyrusRT plug-ins in order for our view to interact with the PapyrusRT multi-diagram editor. The required dependencies that should be imported for our plugin are:

- org.eclipse.papyrus.infra.core
- org.eclipse.papyrus.views.modelexplorer
- org.eclipse.uml2.uml
- org.eclipse.papyrus.infra.core.sasheditor

To create a web-like user interface we used the org.eclipse.ui.forms plug-in which is based on SWT and JFace. The class FormToolkit serves as a factory for the creation of the required user interface elements. The FormToolkit provides a Form which serves as the frame for the user interface elements. They provide a body which can be accessed via the getBody() method which contains other user interface elements.

Figure 3.6 shows the Monitoring Configuration view in Eclipse. In order to make it user friendly, the user is provided with 3 buttons to monitor the model as shown in Figure 3.7. The user selects the elements that should be monitored from the model in the Model Editor window. The buttons “Monitor” and “Unmonitor” are enabled in the view when the user selects the model. The user can only select the states, transitions
or pseudo-states for monitoring purpose. The “Create Pair” button is enabled when the user selects exactly two elements. The details of the button functionality are explained in the subsequent sections.

### 3.3.2 LTTng Profile and Boolean Stereotypes

A profile is a restricted form of a metamodel that should always be related to a referenced metamodel, for example UML. Any profile created cannot be used without its reference metamodel. It defines a limited capability to extend metaclasses of the
reference metamodel via stereotypes. A profile that we create is made to reference metaclasses from metamodels by creating an import relationship between the profile and the reference metaclass. In Figure 3.8 we create a package import to the UML-RealTimeStateMach for our LTTng Profile. In order for the code generator to know which elements have been marked for monitoring, we created a custom profile called “LTTng Profile” which consists of three stereotypes named “LTTngState”, “LTTng-Pseudostate” and “LTTngTransition” as depicted in Figure 3.9. They extend their respective metaclasses and each have a boolean attribute to indicate whether that element has been marked or not. The reason we created separate stereotypes for each element was for the support for future enhancements for monitoring other aspects of the elements.

![Figure 3.8: Referencing metaclasses](image)

The behavioral elements that should be monitored are selected in PapyrusRT as shown in Figure 3.10. Obtaining the current element selection of the user involves
listening to the selection using the ISelectionListener instance. This is used to obtain the Selection object containing a collection of Graphical Objects. In order to obtain the underlying domain object (UML) we use the Papyrus method “getSelectedUmlObject()”. In order to apply the profile to the model, we require the URI instance having the path of the LTTng profile (“pathmap://LTTNG_PROFILE/LTTng”). We then make use of this instance to obtain the profile and attach it to the UML-RT model in PapyrusRT. Once a profile has been applied to a model, the stereotypes defined in the profile can be applied to the instances of the appropriate metaclasses. Once a stereotype is applied to an element, that element is effectively extended with the properties that are defined as part of the stereotype.

The “Monitor” button applies the LTTng profile to the model which can be seen
in Figure 3.11. Once the LTTngProfile is applied to the model the stereotype corresponding to the selected element is attached and the boolean attribute “isMonitored” is set to true. The “Unmonitor” button removes the Stereotype from the elements.

By default in PapyrusRT the RT profile is applied to the model as shown in Figure 3.12. The “LTTng” stereotype corresponding to an element is applied upon selection. This is seen in Figure 3.13. The code generator is thus notified of the marked elements and will add tracepoint statements to the appropriate places in the generated code.
3.3. MONITORING CONFIGURATION IMPLEMENTATION

Figure 3.10: Selecting the elements in PapyrusRT
3.3. Monitoring Configuration Implementation

Figure 3.11: Attaching LTTng Profile to the model upon element selection.
3.3. MONITORING CONFIGURATION IMPLEMENTATION

Figure 3.12: UML-RT Profile in PapyrusRT
Figure 3.13: LTTng Stereotype applied to the state and boolean attribute set to true.
3.3.3 Time Constraints for Element Pairs

In order to support the checking of time constraints in PapyrusRT, we have implemented the “Create Pair” button which is enabled when the user selects two elements in PapyrusRT. The user is prompted with a window to enter the expected time interval for the element pair that has been selected as shown in Figure 3.14. This time interval is entered in milliseconds. A text file is generated in the folder of the model with the entered information, i.e., both the elements and the time interval are entered. This file is later used after tracing with LTtng to validate if the expected time interval for the elements is valid or not.
3.3. MONITORING CONFIGURATION IMPLEMENTATION

Figure 3.14: Selecting the element pair in PapyrusRT
3.3.4 Highlighting Monitored Elements

The Cascading Style Sheet (CSS) component of PapyrusRT allows for the definition of custom style sheets for UML-RT diagrams. In order to notify the user which elements are selected for monitoring we created a CSS Theme in PapyrusRT called “LTTng” highlighting the monitored elements in purple. The Stylesheet is a text file with .css extension. Once the theme is defined it requires an extension point with two entries: Theme definition and a Theme contribution. The style sheet checks the applied stereotypes of the elements to highlight them. The selected element pair for time constraint validation is highlighted in Green.
Chapter 4

Proof of Concept

This chapter describes the use of the implementation on three examples. Section 4.1 describes the models created in PapyrusRT for validation purposes. Section 4.2 and 4.3 explain the process of live tracing of real time models and time constraint validation in PapyrusRT.

4.1 Description of Sample Models

In order to test the monitoring of real time models with LTTng we create several real time models varying in complexity. These models are created using PapyrusRT in Eclipse.

4.1.1 Ping Pong

The PingPong model is a simple model involving two capsules Pinger and Ponger and communication between these two capsules. Figure 4.1 describes the structure of the model. In UML-RT capsules are the main building blocks and there must always be a Top capsule that represents the system. All the other capsules are contained in the Top capsule. The two capsules namely Pinger and Ponger are connected through a
4.1. DESCRIPTION OF SAMPLE MODELS

PingProtocol which specifies the valid signals that can be sent from either capsules through real time ports.

Figure 4.1: Ping Pong Structure Diagram

Figure 4.2 and Figure 4.3 describe the behavior of the two capsules. The “Pinger” capsule is rather simple as it has a self-transition “onPong” which is triggered on receiving the “pong” signal from “Ponger”. It also has another transition “onTimeout” to stop the game. The “Ponger” state machine is similar to that of the “Pinger” and receive “ping” messages.

Figure 4.2: Pinger State Machine Diagram
Figure 4.3: Ponger State Machine Diagram

Figure 4.4 shows the selection of the behavioral elements that should be monitored. The code is generated by the PapyrusRT code generator which contains the respective tracepoint files for the annotated elements. For this model, we do not do time constraint validation and also restrict the monitoring of only one state machine. This case study validates the methodology of being able to monitor the required elements in a state machine. The elements that were annotated were traced with LTTng.
Figure 4.4: Monitored elements are shown in purple.
4.1. DESCRIPTION OF SAMPLE MODELS

4.1.2 Widget Production

The Widget Production model is the second case study we created in PapyrusRT. This involves a little more complexity than the Ping Pong model. This model consists of a workstation, robot and a control software capsule. The top capsule depicts the structure diagram of the model as seen in Figure 4.5. The control software is the control logic of the Widget Production model. This model illustrates a real time system.
4.1. DESCRIPTION OF SAMPLE MODELS

Figure 4.7: Widget Production Workstation and Robot state machine diagrams

in which a workstation produces the widgets as instructed by the control software. The Robot delivers the widgets produced by the WorkStation. The controller of the system is the control software which delegates the tasks to be performed by the two systems (Workstation and Robot). The WorkStation and Robot model are similar in the sense that both of them have two states as seen in Figure 4.7. The standby state is entered when one system is waiting for the other system. The WorkStation has a “producing” state which produces the widgets. The Robot has a “delivering” state which delivers the widgets produced by the WorkStation. They communicate with the
control software which manages the process flow. The control software has four states as seen in Figure 4.6. The initial “standby” state is for initializing tasks to start up the system. The system remains in standby for a few seconds to wait for all capsules to start up. The “Control _Software _Produce” state sends a signal to the workstation to produce widgets. Once the workstation produces the widgets it sends a signal back to the control software “widgetproduced”. This triggers “deliverMe” transition which causes the control software to enter the “Control _Software _Deliver” state. Here it sends out a signal to the robot to deliver the widgets. The robot in turn sends a message back to the control software upon delivery. The control software again goes back to the “produce” state. The system enters the shut down state after a definite time period in which all the capsules are notified by the control software of the shut down.

In order to monitor the behaviour of the system, we monitor the states “Workstation _Producing” and the “Robot _Delivering” states. In this example we demonstrate
the monitoring of elements in different state machines unlike the first example in which we monitored the elements of only a single state machine. In Figure 4.8 the monitored elements are highlighted. The annotated model is then used by the PapyrusRT code generator to generate the tracepoint files for the annotated elements. We execute the generated code and trace it with LTTng. The traces are read by the monitoring plugin in PapyrusRT and the active states are highlighted through the trace display UI. This example shows the use of monitoring configuration for a model with more complexity.

4.1.3 Rover

![Rover Architecture](image)

Figure 4.9: Rover Architecture

In order to validate the live monitoring of real time models in PapyrusRT, we created a UML-RT model in PapyrusRT to model the execution of a Rover. A Rover is a small vehicle driven by two motors to move forwards and backwards and rotate. It has two sensors to collect data from the environment (detection of obstacles and temperature). The Rover has been built using a Raspberry-Pi platform running a
4.1. DESCRIPTION OF SAMPLE MODELS

Figure 4.10: Raspberry Pi 3 GPIO diagram [4]

real-time version of Linux. The code generated from the model in PapyrusRT is run on the Pi. The Rover is programmed to move forward until it detects an obstacle. To avoid the obstacle, it moves a few steps backwards, turns right 90 degrees and then moves forward. During the execution the temperature information is collected.

The architecture of the Rover consists of five layers as seen in Figure 4.9. The bottom layer is the Hardware that corresponds to the Raspberry Pi. The Pi embeds 26 GPIO pins, 17 of which are used to connect external devices such as sensors and actuators as seen in Figure 4.10. The File System layer is the file system in the Linux OS. Each GPIO pin in the Raspberry Pi corresponds to a file in the file system. The user can interact with a pin by reading the file’s value or writing into the file. The GPIO layer is a C++ class to ease the file access for controlling GPIO pins. The class contains “get” and “set” methods to change the value of the pin accordingly. The layer on top of the GPIO layer is the Rover library modeled in PapyrusRT. The
Rover library contains the business logic of the application. The Rover logic is the topmost layer which makes use of the Rover library. The Rover logic mainly is the control software through which the rover execution is controlled and modeled [7].

**Figure 4.11: Rover Diagram [4]**

For the hardware implementation we chose to work with Rasberry Pi 3 since it supports Linux OS. Figure 4.11 illustrates the assembly of the Rover platform. The Raspberry Pi 3 is the core component. It embeds two step motors attached to two wheels. An ultrasonic distance sensor is used to measure the distance of obstacles. A breadboard is used to connect the different compartments. Two sets of batteries are used to power the Raspberry Pi and the two motors. A voltage regulator is used to ensure that the Pi is getting 5 volts that it needs to be powered on [4].

Figure 4.12 describes the rover model in PapyrusRT. The Engine Controller and Detection state machines are the rover library which model the behaviour of the
rover movement. The Engine Controller consists of 5 states for movement as seen in Figure 4.13. The rover begins in the “Idle” state. It then transitions to the “Move Forward” state. Once it detects an obstacle it can choose to move to the other states depending on the rover control logic. The Detection state machine is responsible for detecting the obstacle and is depicted in Figure 4.14. It constantly is measuring the distance of the obstacles and sending the information to the rover controller. The
4.1. DESCRIPTION OF SAMPLE MODELS

Rover Control Software contains the logic of the rover movement. By separating the business logic of the rover, it provides flexibility to create models for the rover movement in different ways. We have modeled the rover control logic as follows.

Figure 4.14: Rover Detection state machine diagram

Figure 4.15: Rover Control Software state machine diagram
4.2. VALIDATION THROUGH LIVE TRACING

The top capsule of the Rover describes how the control software capsule is connected to the rover library. The control software state machine begins with the “Standby” state as seen in Figure 4.15. This state is used for setting up the pins and for loading all the model elements. After four seconds the rover goes into the “Forward” state. In order to detect the obstacle, the control software constantly receives distance data from the “Detection” capsule. We use a choice point to decide the action corresponding to the distance between the rover and the obstacle. If the distance is greater than 15 cms, then the Rover continues to remain in the “Forward” state. If the distance is less than the threshold, the Rover stops moving forward, goes into the “reverse” state for a few seconds. Then the rover turns right and moves into the “Forward” state. The detection of obstacles is constantly enabled so even while the rover moves backward the rover is detecting for obstacles.

4.2 Validation Through Live Tracing

LTTng supports live tracing of user applications. It also supports remote tracing which allows traces to be sent from one computer to another through the network. We tested our rover execution through live monitoring with LTTng. We monitor the execution of the Rover Engine Controller by selecting the “Moving Forward”, “Moving Backward” states and “turnRight” transition as seen in Figure 4.16 by using the Monitoring Configuration UI. The code generated by PapyrusRT now contains the tracepoints for the selected elements. We used this code on the Raspberry Pi 3 which controlled the rover. The code generated with the tracepoints was executed and traces were sent at run time back to the computer on which the rover model was created. The live tracing was enabled via wifi network connection. The live traces
were useful to visualize the execution of the model at run-time. The selected elements that were active during run time were highlighted in the model in PapyrusRT. As depicted in Figures 4.17, 4.18, and 4.19 the active elements are highlighted in red depending on the trace. This occurred at run-time so the execution of the live traces on the model happened dynamically.
4.2. VALIDATION THROUGH LIVE TRACING

Figure 4.16: Monitoring Rover model
Figure 4.17: Rover live tracing MovingForward state highlighted
4.2. VALIDATION THROUGH LIVE TRACING

Figure 4.18: Rover live tracing MovingBackward state highlighted
4.2. VALIDATION THROUGH LIVE TRACING

Figure 4.19: Rover live tracing turnRight transition highlighted
4.3. TIME CONSTRAINT VALIDATION

4.3 Time Constraint Validation

In order to validate the time constraints, we provided a timing input specification. A timespec file is generated which contains the element pair that has been selected. The contents of this file specifies the expected maximal delay between the occurrence of the first element and the second element. Once a trace has been created the user can check if the actual delay exceeds this threshold. We created a timing specification for the “Moving Forward” and “Moving Backward” states, the expected time difference being 1000ms as seen in Figure 4.20. The actual time difference between the occurrence of both states was found out to be 2019 msecs based on the information obtained by LTTng traces as seen in Figure 4.21. The developer of the real time model knows the time constraints that have been specified in the model. This technique of validating the timing information could prove useful to figure out latency errors, as many real time models contain time constraints. This latency information displayed on the model level is helpful for the developer of real time models to determine the exact delay between two events. This methodology can be extended to support monitoring of several events and the timing information could be helpful for enforcing guard conditions or other assertions on the model level [9].
Figure 4.20: Rover Time Specification
4.3. TIME CONSTRAINT VALIDATION

Figure 4.21: Rover Time Specification Validation
Chapter 5

Conclusions and Future Work

With the increasing complexity of software and software development, model driven engineering can potentially reduce complexity through the use of software models as the implementation of software. The developer creates the models with a thorough understanding of requirements. Code is automatically generated from these models. Although this methodology is widely used in certain domains, existing tools can be improved by giving developers more effective means for checking the correctness of models.

5.1 Summary

The most significant contribution of this thesis is the design and implementation support for a customizable monitoring configuration for an open source model driven development tool, PapyrusRT. The real time models can be traced with LTtng. The development of the Eclipse plugin enabled us to add a monitoring capability to PapyrusRT. The definition of a monitoring configuration class diagram specifying the possible events that can be monitored can be used to enhance the existing monitoring support. The monitoring configuration was implemented through the definition of a
custom profile by the name “LTTng” for supporting tracing with LTTng. The user can select the elements from the model and specify the elements that they need to monitor. The model is annotated with this information and the code generator is able to add the tracepoints to the elements that have been selected through the monitoring configuration UI. The UI also provides support for the input of a timing specification of an element pair. This is used to specify the expected time delay between two elements. In real time models it is possible to specify timing constraints (e.g., time it takes between two states or transitions to occur). Using this information, we can use the traces to check the actual time difference. This would potentially help developers to figure out any latency errors on the model level and rectify them early in the development phase. Once the model is traced the user can step through the traces through the trace display on the model level.

5.2 Conclusion

In this thesis we implement the support of monitoring behavioural elements in UML-RT models in PapyrusRT with LTTng. We also support the time constraint validation based on the timestamp difference in the traces. This implementation has been performed in Eclipse through the creation of plugins. We created three models of different complexity to test our monitoring configuration. We also have tested the live tracing of UML-RT models in PapyrusRT using the Rover model test case. The model elements annotated with the information are traced with LTTng. This feature in PapyrusRT can thus help developers to check the correctness of their models. The trace information is displayed on the model level so the developers need not look into the code that is generated from the model. This is our ultimate goal that through
model monitoring the software can be developed more efficiently and errors can be
detected early.

5.3 Future Work

This thesis presents the vision of all possible events which can be monitored in UML-
RT models. The support for more elements apart from the behavioural elements
discussed in this thesis can be extended. Being an open source tool, PapyrusRT has a
lot of potential for the addition of additional functionalities. Eclipse IDE is a suitable
environment to develop new extensions via plug-ins. This implementation supports
the specification of timing requirements however there is no enforcement of these
requirements. This can be further extended to support constraints and enforce them
on the model level. Support for monitoring the structural components of a UML-RT
model can lead to the correctness of designing real time models. Customizing the
code generator to support the additional monitoring capabilities has to be studied.

5.3.1 Model Based Debugging

In our vision of model based debugging, the traceability between the model and
the generated code must be ensured. For this the appropriate mapping information
must be collected during code generation. The key idea of adapting model based
debugging is through the exchange of the roles of the model and the actual system.
The model reflects the behaviour of the incorrect program. The test cases will specify
the anticipated result. This will undoubtedly be useful as model-level execution and
debugging would be useful for finding errors in the beginning stages of development.
Specifically it will be suited for discovering faults in the basic functionality of the
model. Requirement violations involving timing constraints are hard to detect on the model level because the way systems use computational resources varies based on the generated code. Code-level monitoring can detect such violations. The integration of both model and code level monitoring for debugging purposes has to be further studied. To achieve this integration, relevant code-level elements must be traceable to their model-level counterparts.

5.3.2 Model Refinement Using Run-time Monitoring Information

Run-time monitoring and testing are important activities in software development. They can help discover faults in the model through the observation of the code at run-time under different operating conditions. In the current implementation of this thesis tracing information is displayed on the model level, however, the traces are not analyzed for model refinement. The traces provide useful information that can be used for refining the model and also create assertions on the model. This requires further study of traces and how the model can be validated using this information.
BIBLIOGRAPHY

Bibliography


