Using LTTng for runtime monitoring of models of real-time embedded systems

by

Nondini Das

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
October 2016

Abstract

Model-Driven Development (MDD) is used for reducing the complexity of a software development process. One of the principal features of MDD which can make it very effective is the support for automatically generating code from the modeling artifacts. An example of MDD is the development of complex real time embedded software systems using the real time profile of UML (UML-RT). Development of this software is difficult mainly due to the requirement to satisfy timing constraints in a resource-constrained environment. Determining the correctness of this requirement is very important for ensuring the integrity and reliability of a real time software system. This research focuses on examining the correctness of timing information related to UML-RT models. The Linux Trace Toolkit: next generation (LTTng) is used for monitoring an executable real-time application, where the code is generated from UML-RT models using the open-source Papyrus-RT tool. Some of the key research outcomes include the ability to trace a user application, to read a trace file, display the trace results on the model level and display associated timestamps in textual form through the implementation of an Eclipse plugin. In addition, support is also provided to verify the actual timing information of a trace file against the desired user input. This feature enables users to find out the occurrence of any timing delay. Finally, three case studies are conducted using the prototype Eclipse plugin.

Acknowledgments

First and foremost, I would like to extend my sincerest gratitude to my supervisor, Dr. Juergen Dingel, for giving me the opportunity to work in an excellent atmosphere. His proper guidance and insightful comments on my work throughout this thesis showed me the right direction. This thesis would not have been possible without his constant support.

Then I would like to extend my sincere appreciation to my husband Tuhin Das for staying by my side all the time and giving me all the support and mental strength I needed during the stressful time.

I would also like to thank my project partners Leo Juwaidah and Suchita Ganesan who were very supportive, co-operative and friendly throughout the project period.

Next, I would like to thank Ernesto Posse, who is an employee at Zeligsoft, Canada. Whenever we have an issue with the Papyrus-RT tool, Ernesto was there to help us.

I would also like to thank my lab mates David Andrews, Amal Khalil, Mark Fischer, Mojtaba Bagherzadeh, Reza Ahmadi and Raquel Oliveira for their insightful feedback during the group meetings. I would like to extend special thanks to my colleague Nicolas Hili whose assistance helped me solve a number of technical problems during the completion of the thesis.

I would also like to thank my other lab mates Eric Rapos, Doug Martin and Nafisa

Kahani from the Software Technology Lab for making the lab environment very nice and enjoyable.

Last but not least, I would like to thank my family members, my mother, in-laws, relatives and friends for giving me constant support throughout my graduate life at Queen’s.

Contents

List of Tables

List of Figures

List of Listings

List of Acronyms

CTF Common Trace Format.

GPIO General Purpose Input/Output.

JDK Java Development Kit.

LTTng Linux Trace Toolkit: next generation.

MBE Model Based Engineering.

MDA Model Driven Architecture.

MDD Model-Driven Development.

MDE Model Driven Engineering.

Papyrus-RT Papyrus for Real-Time.

RTES Real-Time Embedded System.

SDK Software Development Kit.

UML-RT UML for Real-Time.

Chapter 1

Introduction

1.1 Motivation

Modern software systems can be extremely large and complex (e.g., [9]). It is hard to develop efficient software applications using traditional code based processes. The amount of difficulties increases when it comes to test the software and refine possibly millions of lines of code after finding bugs in a product. These challenges have motivated the development of model driven development (MDD). A major benefit of using MDD is the possibility of developing large, efficient software systems without having to code millions of lines. This advantage encourages software developers to use MDD as the means of developing and refining large and complex software systems [31].

Similar to other software systems, proper quality assurance of modern real-time embedded systems (RTES) is important. Software systems in telecommunications, aerospace and defense are usually very large and extremely complex. This makes the quality assurance of these systems very difficult. Missing a time interval or experiencing a slight delay can cause extreme problems in RTES. As Bran Selic, the creator of the real-time profile of UML (UML-RT), mentioned in [61], "The only characteristic common to all real-time software systems is timeliness; that is, the requirement to respond correctly to inputs within acceptable time intervals". As an example, in a telecommunication system, any violation in timing requirements can cause very poor user satisfaction. In hard real-time systems, for example, in aerospace or medical devices, the consequence can be fatal as such violations can lead to significant injuries, or even to loss of human lives.

This research focuses on quality assurance of RTES developed using UML-RT. Quality assurance typically has two phases: detecting the source of the problem and providing a solution to the problem. We have worked on the detection of problems via runtime monitoring.

1.2 Problem statement

One of the main advantages of developing real-time embedded systems using MDD is the automatic generation of code using MDD tools. But quality assurance of these real-time systems can be challenging. This is because if a user faces an issue in a real-time system, it is hard to find out the exact location in the model where the problem has occurred. It is almost impossible to modify the generated code for solving a problem without possibly introducing a wide number of new problems because the automatically generated code is usually not easy to understand. Therefore, it is important to find the source of problem on the model level. One way to do this is by tracing the generated code while running a real-time system. There are some real-time modeling tools which provide tracing facilities. For instance, IBM Rational RoseRT [6] and IBM RSA-RTE [5] are tools for creating models for real-time systems, and automatically generate traces after executing models. But these are proprietary tools and not suitable for non-commercial purposes such as research and academic usage. Recently, an open-source tool Papyrus-RT has been introduced for developing models of real-time embedded systems [59]. This research is conducted to contribute to the development of Papyrus-RT through providing support for run-time monitoring of UML-RT model execution.

1.3 Overview of the Proposed Approach

While conducting this research, we have focused on validating the timing requirements of a real-time system. The goal is to find out if there is any timing delay in the system that violates the requirements. We propose an approach for monitoring runtime information of real-time embedded systems using the LTTng tool [10].

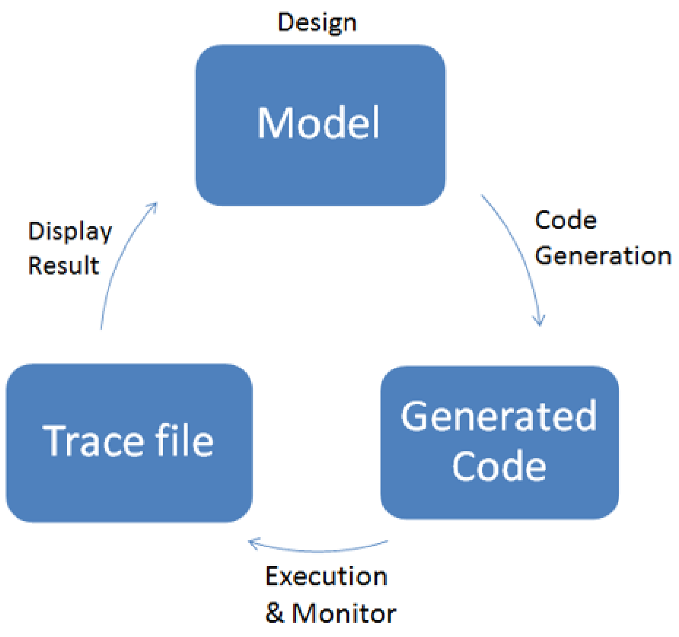


Figure 1.1: Proposed approach overview

Fig. 1.1 shows an overview of the proposed approach. We have used the Papyrus-RT tool for designing a model and generating code from it. We have used the LTTng tool to detect any timing delay or latency by monitoring the execution of the generated code. This gives us traces with timing information which we can display on the model level. Thus, with this approach we can detect if there is any delay and we can view the associated model element where the problem occurs.

1.4 Hypothesis

We hypothesize that it is possible to use open source monitoring tools such as LTTng to monitor the execution of code generated from models and to display the resulting trace on the model level.

1.5 Organization of Thesis

Chapter 2 presents some background discussion. We discuss Model-driven development (MDD), real-time embedded systems, the modeling language UML-RT, the MDD tool Papyrus-RT, the tracing tool LTTng and the language Xtend.

Chapter 3 discusses some related work. We present similarities and differences between our work and the existing works.

We start Chapter 4 by giving an overview of the whole project. Then we present a brief description of the first two parts of the project: monitoring configuration and code generation. This is followed by a detailed description of our work, i.e., "Tracing using LTTng and Trace Display."

Chapter 5 shows the proof of concept of our work by presenting three case studies.

Finally, in Chapter 6, we summarize our work, followed by discussing its limitations. We also present some possible future work.

Chapter 2

Background

This chapter discusses some background materials regarding MDD, real-time embedded systems, UML-RT, Papyrus-RT, LTTng and Xtend.

2.1 Definitions of Model-Driven Development and Relevant Terminology

The term *model-driven development (MDD)* is defined in a number of ways in the current state of the art.

The ability of MDD to transform a conceptual model into a working real world application is captured in the definition provided in [57]. According to the authors in [57], "Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing".

The power of MDD in dealing with the complexity of software development is highlighted in the definition provided by Atkinson and Kuhne. As they mentioned in [30], "Today's object-oriented languages let programmers tackle problems of a complexity they never dreamed of in the early days of programming. Model-driven development is a natural continuation of this trend. Instead of requiring developers to spell out every detail of a system's implementation using a programming language, it lets them model what functionality is needed and what overall architecture the system should have".

On the other hand, the definition provided by Brent et al. in [50] talks about the shifting of software development abstraction with the advent of MDD. According to them, "Model-driven development (MDD) is a software-engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle" [50].

As a summary of these different viewpoints, MDD is a development process where models are the primary artifacts of development rather than code. A model in this context is a reasonable set of consistent elements to describe something (e.g., a network, bank, automobile, or telecommunication system) built for some purpose that is consistent with a particular form of analysis. For example, models can be used for communicating ideas between people and machines, checking of completeness, analysis of concurrent systems with respect to race conditions, generation of test cases, analyzing the viability in terms of estimating the cost of development and the feasibility of transformation into an implementation [57]. In MDD, developers do not need to write thousands of lines of code for implementing complex software systems because of the possibility of generating code and documentation automatically from the models with the help of the MDD process [37, 51, 52, 62].

There exist some other acronyms closely related to MDD, i.e. MDA, MDE and MBE. The definitions of these acronyms are as follows:

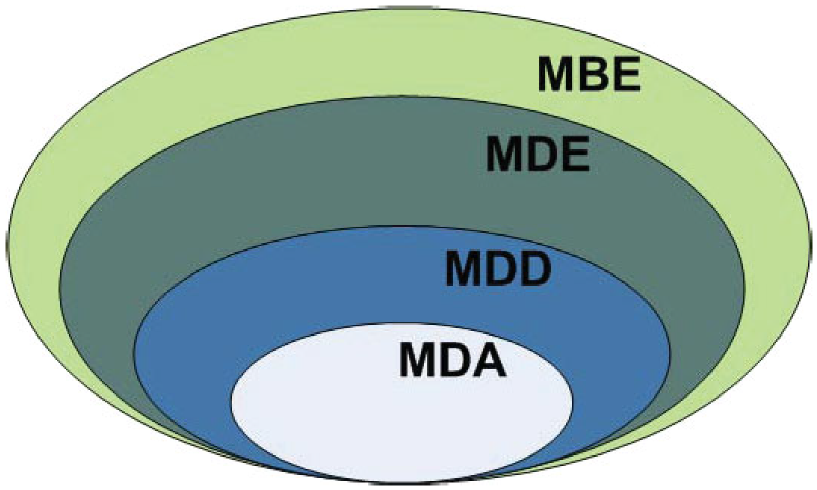


Figure 2.1: Relationship between the different MD* acronyms [37]

- Model Driven Architecture (MDA): MDA is a subset of MDD because, by definition, MDA is the particular vision of MDD which has been proposed by the Object Management Group (OMG) [14, 15] and also depends on the use of OMG standards [37].

- Model Driven Engineering (MDE): MDE contains a number of model-based tasks of a complete software engineering process including model-based evolution of the system, model-driven reverse engineering of a legacy system and so on. Therefore, MDE can be regarded as a superset of MDD [37].

- Model Based Engineering (MBE): MBE is known as a softer version of MDE. It is a process where software models play an important role even though they are not necessarily the key artifacts of the development process. For example, the designers specify the domain models of the system in the analysis phase of a development process; but then these models are directly handed out to the programmers as blueprints to manually write the code. Therefore, in this case, although models still play an important role, they are not the central artifacts of the development process. MBE is considered as a superset of MDE [37].

Fig. 2.1 shows a graphical overview of the relations among different model based development processes.

In our research, among these approaches we have focused on model-driven development approach where MDD has been used to generate code for a real-time embedded system. The following section provides background on real-time embedded systems.

2.2 Real-Time Embedded System

To talk about real-time embedded systems, we should present the definitions of embedded system and real-time system.

In the computing world, the term "embedded system" means an electronic system that is designed for performing a dedicated function and is often embedded within a larger system [46]. The computing power of an embedded system is a fragment of a larger system and it does not provide some standard computing service to users as its primary job. For example, a desktop computer is not an embedded system (unless it is within a device) because it can be used for multiple purposes and provides standard computing services. However, a computerized microwave oven or a VCR is an example of an embedded system as the embedded computing power of these systems is part of a larger system and dedicated for a specific function [44].

If a system needs to respond to a service request within a certain amount of time, it can be considered a real-time system [46, 38, 53]. In a real-time system, a real-time computing constraint gets attached to all the tasks imposed by each of the incoming service requests. The associated real-time computing constraint is known as the timing constraint of the related task. The timing constraint of a task is usually specified in terms of its deadline, the time instant by which the execution of the task needs to be completed. A timing constraint can be either a hard or a soft constraint depending on the consequences of missing a task deadline. The consequence of a missed deadline in a hard timing constraint is fatal. A late response in hard real-time systems, i.e., in automobile systems or medical devices, is usually unacceptable and becomes useless. In contrast, in a soft real-time system, i.e., in an audio-video chatting software, the consequence of a missed deadline is undesirable but tolerable. Although it would result in degraded quality, a late response in these systems is still useful as long as it is received within some acceptable range of time. An actual system may have both soft and hard timing constraints. In a soft real-time system, all the tasks have soft timing constraints, whereas, for a hard real-time system, the key tasks of the system need to have hard timing constraints [46].

Therefore, a real-time embedded system means a real-time system which is designed to be embedded within some larger system [46].

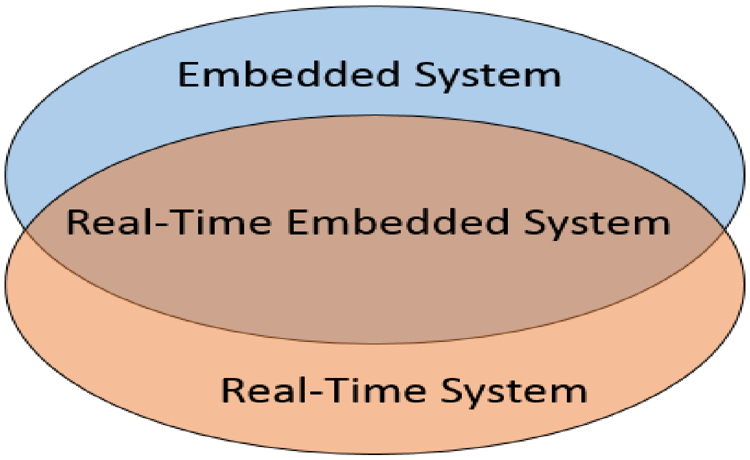


Figure 2.2: System Classification (Adapted from [46])

Fig. 2.2 shows a classification of the above mentioned systems where real-time embedded systems lie in the intersection between the real-time and embedded system.

A comprehensive definition of real-time embedded system is given by the authors of [48]. According to them, "Real-time and embedded systems are computer-based systems that interact with the physical world. This means that they are not only coupled to the physical world but that they are also constrained by the physical capacities of their underlying hardware and/or software platforms".

Developing large real-time embedded systems is a difficult task especially when it comes to manual coding because it increases the risk of bugs and failure. On the other hand automatic code generation from models could be a solution for this kind of problem. Thus, model-driven development (MDD) has been proposed to facilitate the development of real-time embedded systems (e.g., [31, 36]).

There is a number of modeling languages used for MDD, for instance, UML, SysML, SDL, ORM [12].

In our research we have used the real-time profile of UML (UML-RT) for capturing the structure and behaviour of an embedded real-time system.

The following section will discuss UML-RT and Papyrus-RT, a tool that we have used for developing UML-RT models.

2.3 UML-RT and Papyrus-RT

2.3.1 UML-RT

UML (Unified Modeling Language) is a graphical language to specify, visualize, construct, and document software systems [24]. A profile in UML is used for customizing UML models for particular domains and platforms by providing a generic extension mechanism. Extension mechanisms allow the refinement of the standard semantics in a strictly additive manner and it prevents them from contradicting standard semantics [60, 18, 29]. UML-RT is the real-time profile of UML. UML-RT provides a unified framework to model and analyze real-time systems. To be precise, for facilitating the modeling of run-time structures, UML-RT adds five stereotypes to standard UML, where a *stereotype* is a mechanism to classify or brand a model element and introduce a new type of modeling element [29, 23]. The five stereotypes are: capsule, port, protocol, protocol role and connector [40]. We present a brief discussion of these stereotypes in the following paragraphs.

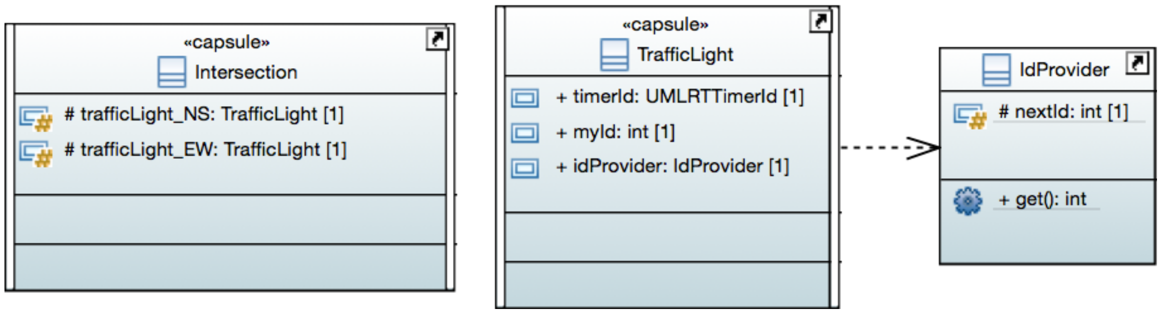

«capsule»
Intersection
# trafficLight_NS: TrafficLight [1]
# trafficLight_EW: TrafficLight [1]
«capsule»
TrafficLight
+ timerId: UMLRTTimerId [1]
+ myId: int [1]
+ idProvider: IdProvider [1]
IdProvider
# nextId: int [1]
+ get(): int

Figure 2.3: UML-RT capsule and class notation on class diagrams [17]

A UML-RT model consists of passive classes and capsules.

Passive classes are similar to classes in any object-oriented language, i.e., Java. Passive classes can be used as properties of a UML-RT capsule. They can also be used for message data parameters. In addition to defining operations, the behaviour of a passive class can also be represented using a state-machine. A state-machine is a graph of states and transitions that is used for describing the response of an object of a given class to the receipt of outside stimuli [17, 13]. Fig. 2.3 shows a passive class named “IdProvider” connected with a capsule through a dotted arrow which indicates that “IdProvider” is used by the capsule.

In contrast, *capsule* elements in UML-RT are used to represent independent flows of control in a system. They are known as the fundamental modeling element in UML-RT development. Capsules have some similar properties as classes [32] of UML diagrams. For instance, capsules can contain operations and attributes. They can also contain properties like dependency, generalization, and association relationships. Capsules can also contain one or more sub-capsules [13].

An *attribute* is a variable which can be specified with a type, an initial value and multiplicity. In Fig. 2.3 "myId" is a variable having integer type and also is an attribute of the TrafficLight capsule.

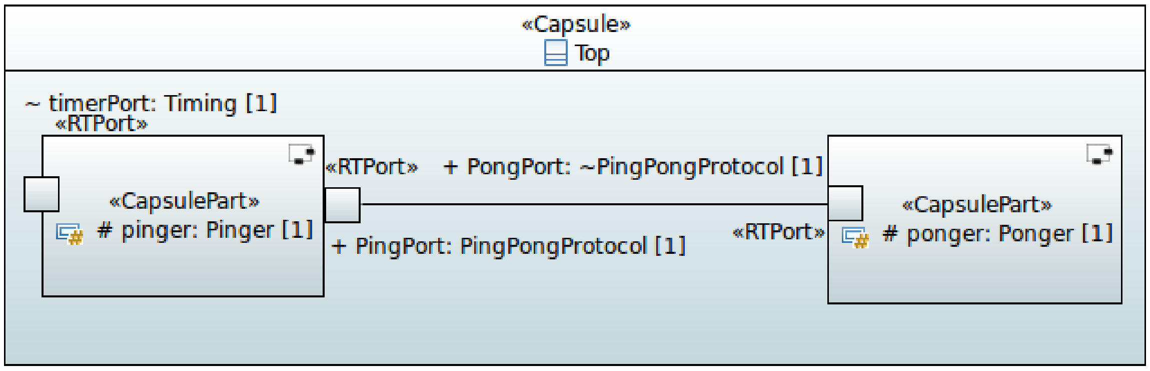


Figure 2.4: Structure diagram of a UML-RT capsule

In addition to these properties, a capsule can also contain a structure diagram and a behavioural diagram. A discussion of these capsule properties is presented in the following subsections.

Structure diagram

The structural aspects of a capsule are represented in a structure diagram, which depicts the elements that the capsule owns along with their inter-connections (see Fig. 2.4). A capsule can have capsule parts, ports and connectors.

A *capsule part* is an instance of a capsule which is contained by another capsule (see Fig. 2.4).

Ports are objects for sending and receiving messages to and from capsule instances. They are owned by the capsule instance because they are created and destroyed along with the owner capsule. Each port has its identity, which is distinct from the identity and state of its owning capsule instance. The main benefit of having message-based interfaces in UML-RT is the separation of the capsule instance from the outside environment. A capsule has no knowledge of its context outside the message interfaces which makes it much more flexible and robust than regular objects [13]. An example of a message port is shown in Fig. 2.4 where Pinger capsule has a port named "PingPort" and Ponger capsule's port is named as "PongPort".

Types of port can be categorized in multiple ways depending on the visibility, the connector types and the message termination points.

Visibility:

- Public - Ports that are part of a capsule interface are known as public ports. In a capsule structure diagram these ports are shown as located on the capsule boundary. Public ports may be visible from both outside and inside of a capsule instance.

- Protected - Protected ports are used for connecting capsules to the contained capsule roles. In contrast to their public counterparts, these ports are not visible from the outside of a capsule since they are not part of the capsule interface.

Connector type:

- Wired - Wired ports must be connected by a connector to other ports for communicating messages. In a capsule structure diagram, these are the ports that are graphically connected to other ports.

- Non-wired - Non-wired ports are used for modeling dynamic communication channels. In contrast to wired ports, graphical connectors are not used for connecting these ports to other port instances. Rather, non-wired connectors are created and destroyed dynamically during run-time.

Message Termination Point:

- Relay - Relay ports are by nature public and wired implicitly. They are used for modeling connections that pass signal events directly to protected capsule components without being processed by the capsule itself. All the signal events arriving at a relay port can be lost if the port is not connected to an internal component. In general, relay ports can be used for exporting the interfaces of the contained capsule roles.

- End - Unlike relay ports, end ports can be public or protected, wired or non-wired. Messages sent to an end port are intended to be processed directly by the capsule behaviour. End ports are the final destination of all signal events communicated in UML-RT.

For the specification of messages that can be sent to and from a capsule port, a port is associated with a protocol role. A protocol role is the specification of a set of messages that can be received (in) and sent (out) from the port. Essentially, a protocol role defines a port type.

Protocol: A protocol is a communication pattern that represents a set of messages that can be exchanged between two capsule instances. Basically, it is a contractual agreement portraying the valid types of messages that can be communicated among

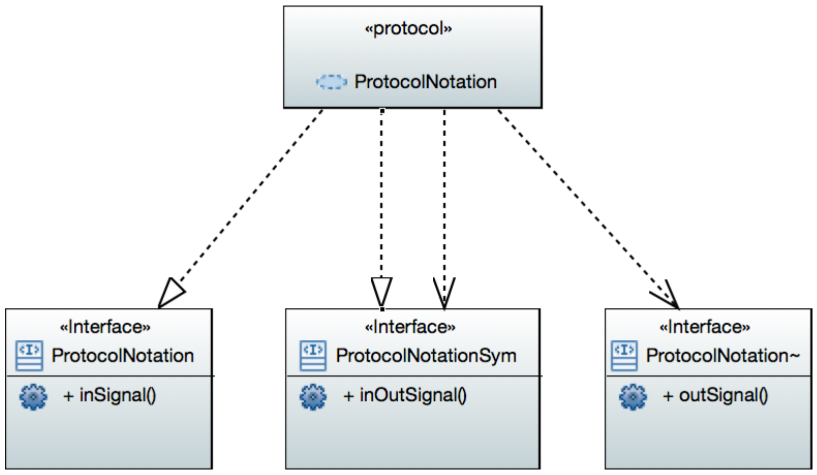


Figure 2.5: UML-RT protocol description (class diagram) [17]

the participants in the protocol. A set of participants, each of which plays a specific role, can be associated with a protocol. Each such protocol role is represented by a unique name and a specification of messages that can be received by that role as well as a specification of the messages that can be sent by that role (either set could be empty).

Therefore, a set of valid signals can be specified along with their directions using a UML-RT protocol. A signal in this context is a message that can be sent either synchronously, or asynchronously. Protocols consist of a list of incoming (provided) and outgoing (required) signals as well as any associated data parameters [13]. Fig. 2.5 shows a class diagram of protocols having a number of incoming/outgoing signals where the left two arrows indicate incoming signals and the two arrows on the right hand side mean outgoing signals.

The purpose of *connectors* is to capture the key communication relationships between capsule roles. They interconnect capsule roles that have similar public interfaces, a.k.a. ports. A key feature of connectors is their ability to interconnect only the compatible ports. In Fig. 2.4, a connector is shown with a solid line connecting two ports.

State-machine diagram

A state-machine [61, 63] and its components are used for describing the behavioural aspects of a UML-RT capsule. The diagram that contains a UML-RT state-machine is called a state-machine diagram (see Fig. 2.6).

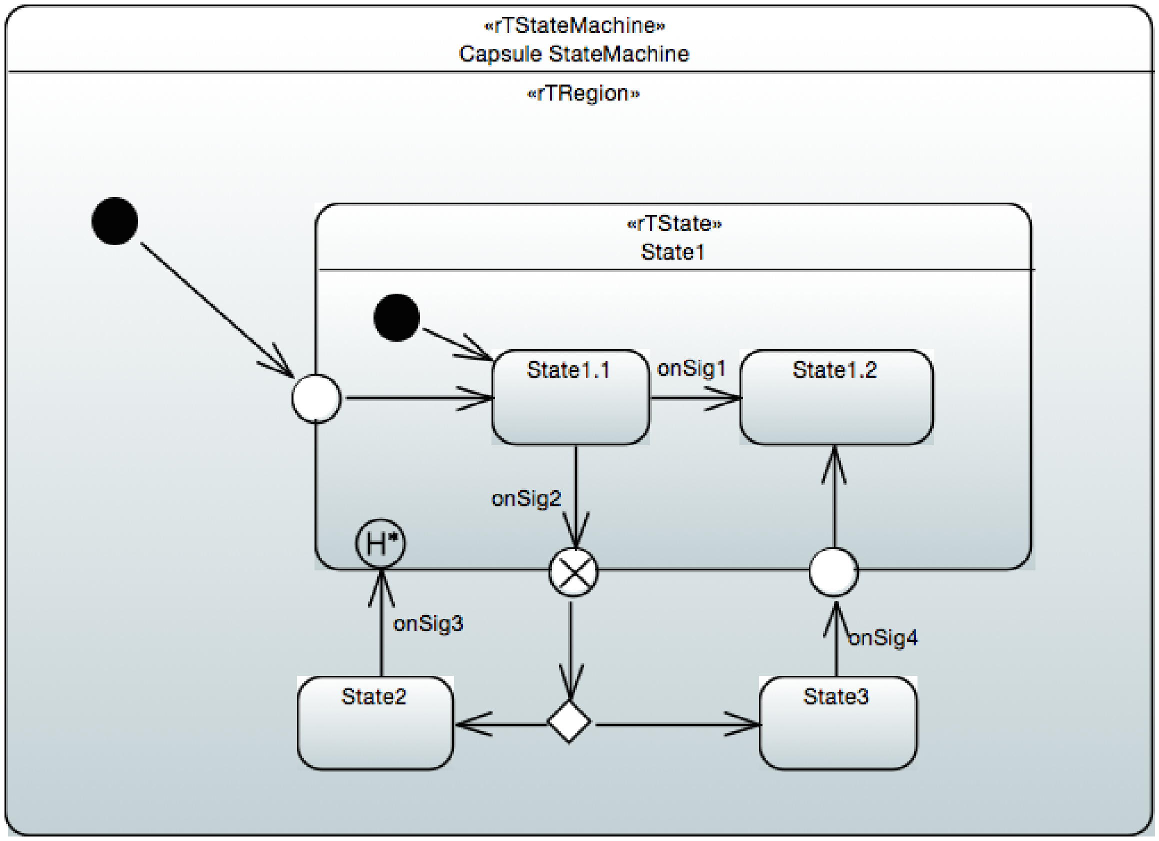


Figure 2.6: UML-RT Capsule State-machine Diagram [17]

In other words, a state-machine diagram is used for describing the life history of objects of a capsule. A state-machine can be comprised of states, transitions and action code for model execution.

States:

A state represents a situation during the life time of an object where certain incoming events can be processed. A state-machine is usually composed of a top state, which can itself contain any number of other states. A state can have the following elements:

Name - A name must be associated with a state so that it can be distinguished from other states in the same context.

Entry/Exit actions - An entry action is executed whenever a state is entered, regardless of which incoming transition has been triggered. Similarly, an exit action is executed whenever we leave the state using any outgoing transition.

The entry point of a state-machine is represented using an *initial state* which can have only one outgoing transition, the initial transition. An initial state is represented using a small filled circle (see Fig. 2.6).

A state can be composed of a number of other states, called substates. This allows modelers to handle the complexity of a capsule state-machine by abstracting away detailed behaviour into multiple levels. If a state does not have any substate inside, it is called a simple state. In contrast, a composite state can have any number of substates. State elements can be nested to any hierarchical level [60, 13]. Fig. 2.6 shows a capsule state-machine diagram with three states in addition to the initial state. Among these three states, "State1" is a composite state as it consists of a number of other states.

Transitions: A transition represents a relationship between two states: a source state and a destination state. When an object in the source state receives a specified event and certain conditions are met, the behaviour will move from the source state to the destination state through the execution of the associated transition [13]. Fig. 2.6 depicts a number of transitions that are represented by a solid line with an arrow. For instance, in Fig. 2.6, “onSig1” is a transition which can be executed if the capsule receives the specified message while it is in the “State1.1” state and the execution will take the system to the “State1.2” state.

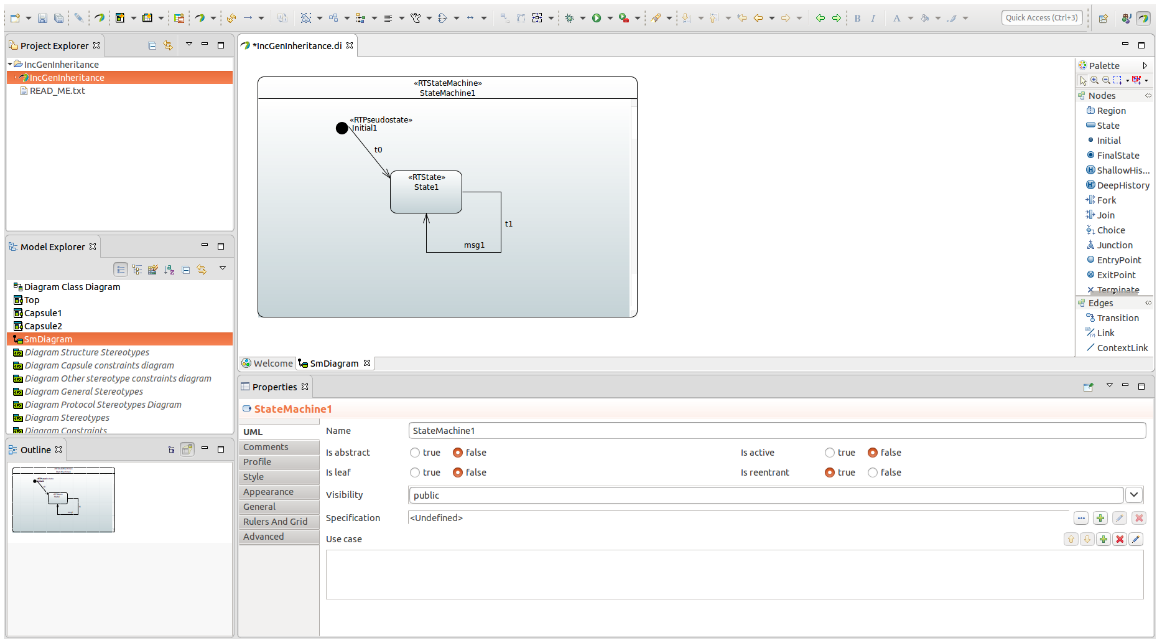
To summarize, UML-RT is a standard language for modelling real-time embedded systems. There are several tools for developing UML-RT models. IBM RoseRT and IBM RSA-RTE are well-known proprietary tools for UML-RT development. In our research we have worked with an open source tool called Papyrus-RT.

2.3.2 Papyrus-RT

Papyrus-RT is an industrial-scale, complete modeling environment for developing complex, large real-time embedded cyber-physical software systems [17].

It is a new open-source implementation of a complete UML-RT development environment including a graphical modelling environment, a code generator and a run-time system. Papyrus-RT allows UML-RT community to develop models that are executable. Papyrus-RT is implemented on top of Papyrus, a well-known UML modelling environment on Eclipse [16] [59].

In our research we have worked with a tool called “LTTng” to monitor the execution of code generated from UML-RT models. We will talk about LTTng in the next section.



StateMachine1

Name: StateMachine1
Is abstract: true [✓ false]
Is active: true [✓ false]
Is leaf: true [✓ false]
Is reentrant: [✓ true] false
Visibility: public
Specification: <Undefined>
Use case

Figure 2.7: Screenshot of Papyrus-RT tool

2.4 LTTng Tracing Tool

LTTng stands for "Linux Trace Toolkit: next generation" [10]. It is used for tracing runtime information of a software application. Tracing is a method used for understanding the execution behaviour of a running software system. Any software used for tracing is known as a tracer and is theoretically similar to a tape recorder. It may be possible to trace both the user application and the operating system at the same time. This can create an opportunity to detect a wide range of problems. Often tracing is being compared to logging. While there are differences between tracers and loggers, tracers are designed for recording very low-level events that happen much more frequently than log messages. Logging is suitable for very high-level analysis of less frequent events [10].

LTTng is a highly efficient open source software package used for tracing the Linux kernel, user applications and libraries at the same time [10, 64]. While other tracing tools can slow down the traced software significantly, LTTng is known for keeping the runtime overhead to a minimum [64, 43].

2.4.1 Core Concepts of LTTng

This section discusses four primary concepts that need to be dealt with while using the LTTng tool: Tracing session, domain, channel and event.

Tracing Session

A tracing session is just like any other session e.g., an website session. While tracing using LTTng everything happens in the scope of a tracing session. It is also known as a container of domains, channels and events [10].

Table 2.1 presents commands for managing a session lifecycle.

Table 2.1: Commands for managing tracing session [4]

Command	Description
create *NAME*	To create a session with a given *NAME*.
set-session *NAME*	Used for switching between sessions, setting current to *NAME*.
start	To start tracing
stop	To stop tracing
destroy *NAME*	To destroy the session with *NAME*. The option -a or –all can be used to destroy all the sessions.
list *NAME*	Used to show information regarding the session with a given *NAME* or to list all the sessions if *NAME* is already omitted

The number of tracing sessions is not limited. A user can create as many tracing sessions as he/she wants [10].

Domain

Essentially, a *domain* means a type of tracer. The LTTng project uses a tracing *domain* as the official term for designating a tracer category. At present, there are *five* known domains: Linux kernel, user space, java.util.logging (JUL), log4j and Python. All the five domains support some unique features that are not available in their counterparts. For example, the dynamic function entry/return instrumentation is currently supported only in the kernel domain but yet to be added in the other domains [10].

Channel

A channel is known as a set of events with a fixed set of parameters and some potential context information. Within a tracing session, for each domain, channels have unique names. A given event is always registered to one or more channels. It is also possible to individually enable or disable channels. Any event that occurs in a disabled channel would never be recorded. An underlying role of a channel is to maintain a shared ring buffer where events are finally recorded by the tracer and consumed by a consumer daemon. An internal ring buffer has many sub-buffers of equal size [10].

Event

The term *event* in LTTng can have three different meanings depending on the context it is used in. While tracing, an event is like a point in space-time. In the context of tracing, the term *space* is a collection of all executable positions of a compiled application that can be used by a logical processor. An event occurs if an instrumentation point is encountered while the program is executed by the processor. In contrast, when the term *event* is used in the context of a recorded trace, it means a recorded event. In a third context which involves the configuration of a tracing session, *enabled events* refer to a set of specific rules which may lead to the transformation of actually occurring events to recorded events. As discussed in the previous section, an event is always registered to at least one channel and can be willingly enabled or disabled. Even though the channel an event is registered to is enabled, if the event itself is disabled, it would never be recorded [10].

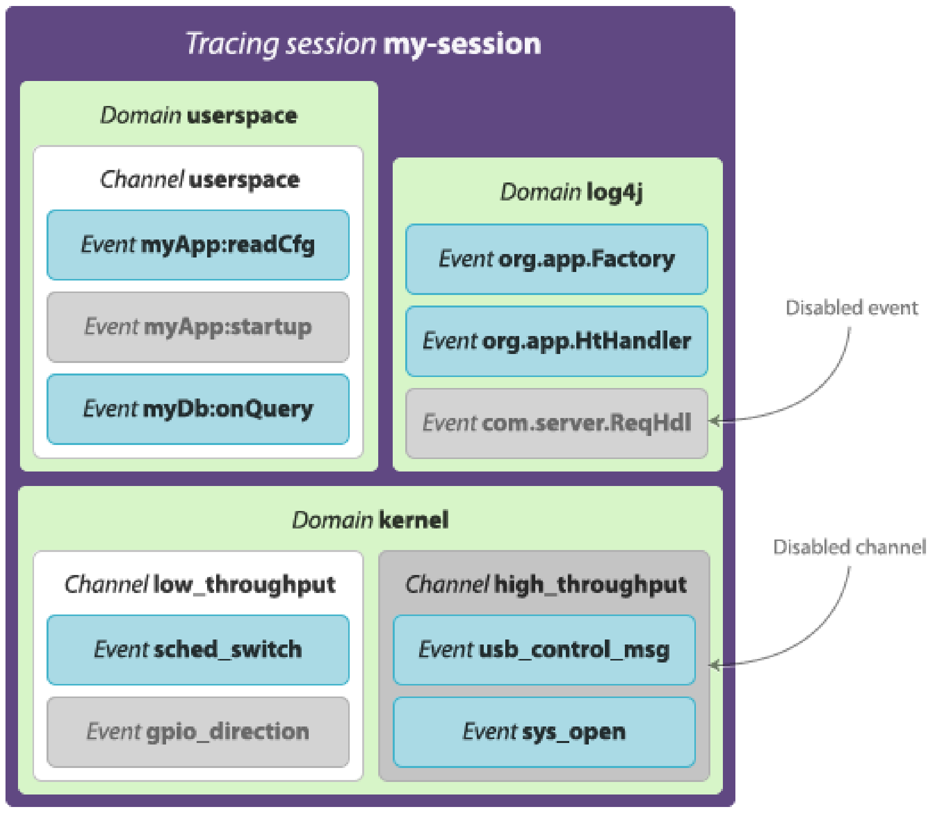


Figure 2.8: Overview of the core concepts of LTTng [10]

Fig. 2.8 shows the overview of the core concepts of LTTng. As we can see in this figure, a tracing session can contain a number of domains, channels and events.

2.4.2 Tracepoint

The concept of *tracing* is similar to have printf() commands at some locations in the source code. Instead of using printf commands, LTTng uses tracepoints which are quite faster and more flexible. A tracepoint represents a probe that needs to be placed manually in the source code for enabling the tracing capability where a probe [7] is like a sensor that can be placed in the code [10].

The format of a tracepoint must have to be defined before using it in the source code. For doing this, a tracepoint provider needs to be created which consists of two files: a tracepoint provider header file and a tracepoint provider definition file. To be valid, each of the tracepoints needs to have the following fields:

- A provider name, which is called the "scope" of the tracepoint
- A tracepoint name
- A each argument of the tracepoint call, the name and type of the argument
- A list of fields, which will be the actual fields of the recorded events for the tracepoint

LTTng uses static tracepoints which can be activated at runtime for recording information about a program execution. The connected probe is called when the program execution hits an active tracepoint and the execution continues when the probe completes execution. A key advantage of using static tracepoints is the ability to extract a huge amount of data with the lowest possible overhead [64].

2.4.3 Instrumentation

Instrumentation is the procedure of adding probes in source code. As mentioned previously, this can be done manually by writing tracepoint commands in the source code, or by automatically using dynamic probes. Users can also trace some-thing that is already instrumented. For example, the Linux kernel is comprehensively instrumented; therefore, users can trace it without caring about placing probes into it [10].

2.5 Xtend

Xtend is a statically-typed programming language which can be compiled into human readable Java code [25]. Xtend code can access all the Java libraries. This feature allows users to take the advantage of using Xtend and Java together [33]. The library of Xtend is just a small layer that can provide useful utilities and extensions on top of the Java Development Kit (JDK) [25]. In addition, comprehensive tool support available in Eclipse that includes a number of useful features such as refactoring and debugging reduces the user effort of development using Xtend [3].

Listing 2.1: A simple hello world program in Xtend [25]

```
class HelloWorld {
  def static void main(String[] args) {
    println("Hello World")
  }
}
```

Listing 2.1 represents an example of the simple “Hello world” program written in Xtend language. Here, the keyword *def* is used to declare a method. Similar to Java it is mandatory to define a class and a main method as the indication of the entry point for an application.

Listing 2.2: Generated hello world program in Java [25]

```
// Generated Java Source Code
import org.eclipse.xtext.xbase.lib.InputOutput;

public class HelloWorld {
  public static void main(final String[] args) {
    InputOutput.<String>println("Hello World");
  }
}
```

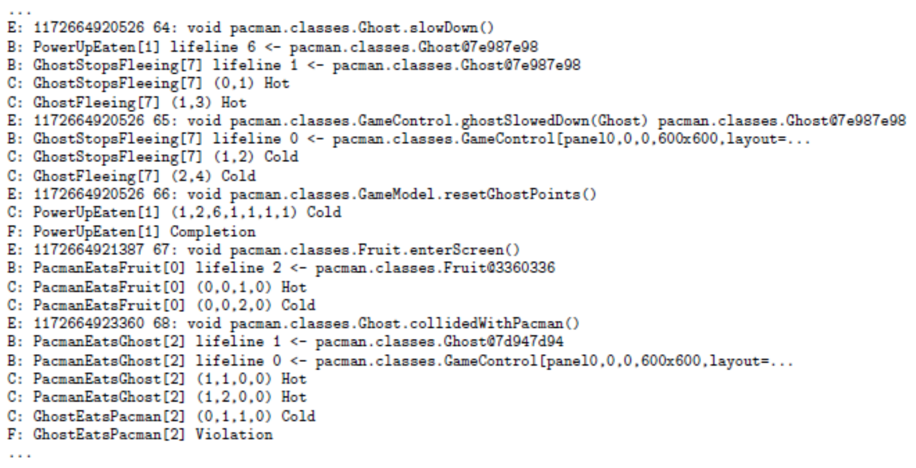
Eclipse automatically translates all the Xtend classes to Java source code after installing the SDK in the system [25]. Listing 2.2 shows an example of generated Java source code from an Xtend program.

Chapter 3

Related Work

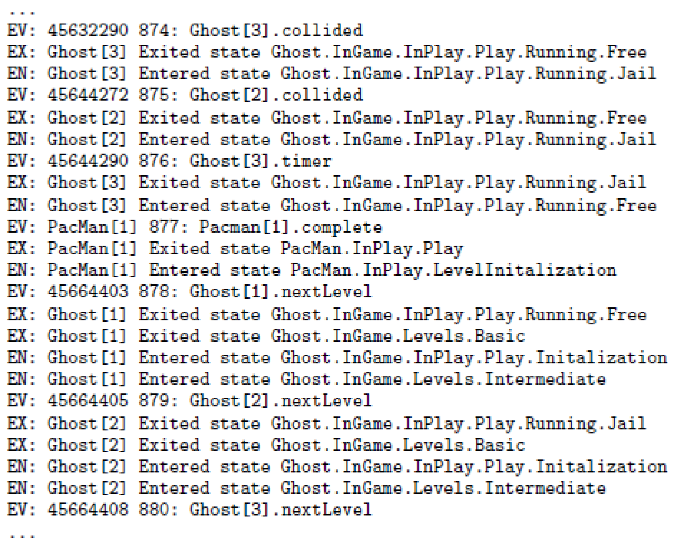
In this chapter, we present a number of related works along with their similarities and dissimilarities with our research.

In [55], the author introduces model based traces and presents two types of model-based traces: inter-object scenario-based traces (see Fig. 3.1) and intra-object state-based traces (see Fig. 3.2). Model-based traces focus on providing insight into an execution of a program at an abstraction level defined by a model. This enables the use of dynamic analysis in model-driven engineering.



```
...
E: 1172664920526 64: void pacman.classes.Ghost.slowDown()
B: PowerUpEaten[1] lifeline 6 <- pacman.classes.Ghost@7e987e98
B: GhostStopsFleeing[7] lifeline 1 <- pacman.classes.Ghost@7e987e98
C: GhostStopsFleeing[7] (0,1) Hot
C: GhostFleeing[7] (1,3) Hot
E: 1172664920526 65: void pacman.classes.GameControl.ghostSlowedDown(Ghost) pacman.classes.Ghost@7e987e98
B: GhostStopsFleeing[7] lifeline 0 <- pacman.classes.GameControl[panel0,0,0,600x600,layout=...
C: GhostStopsFleeing[7] (1,2) Cold
C: GhostFleeing[7] (2,4) Cold
E: 1172664920526 66: void pacman.classes.GameModel.resetGhostPoints()
C: PowerUpEaten[1] (1,2,6,1,1,1,1) Cold
F: PowerUpEaten[1] Completion
E: 1172664921387 67: void pacman.classes.Fruit.enterScreen()
B: PacmanEatsFruit[0] lifeline 2 <- pacman.classes.Fruit@3360336
C: PacmanEatsFruit[0] (0,0,1,0) Hot
C: PacmanEatsFruit[0] (0,0,2,0) Cold
E: 1172664923360 68: void pacman.classes.Ghost.collidedWithPacman()
B: PacmanEatsGhost[2] lifeline 1 <- pacman.classes.Ghost@7d947d94
B: PacmanEatsGhost[2] lifeline 0 <- pacman.classes.GameControl[panel0,0,0,600x600,layout=...
C: PacmanEatsGhost[2] (1,1,0,0) Hot
C: PacmanEatsGhost[2] (1,2,0,0) Hot
C: GhostEatsPacman[2] (0,1,1,0) Cold
F: GhostEatsPacman[2] Violation
...
```

Figure 3.1: Part of a textual representation of a scenario-based trace of PacMan [55]



```
...
EV: 45632290 874: Ghost[3].collided
EX: Ghost[3] Exited state Ghost.InGame.InPlay.Play.Running.Free
EN: Ghost[3] Entered state Ghost.InGame.InPlay.Play.Running.Jail
EV: 45644272 875: Ghost[2].collided
EX: Ghost[2] Exited state Ghost.InGame.InPlay.Play.Running.Free
EN: Ghost[2] Entered state Ghost.InGame.InPlay.Play.Running.Jail
EV: 45644290 876: Ghost[3].timer
EX: Ghost[3] Exited state Ghost.InGame.InPlay.Play.Running.Jail
EN: Ghost[3] Entered state Ghost.InGame.InPlay.Play.Running.Free
EV: PacMan[1] 877: Pacman[1].complete
EX: PacMan[1] Exited state PacMan.InPlay.Play
EN: PacMan[1] Entered state PacMan.InPlay.LevelInitalization
EV: 45664403 878: Ghost[1].nextLevel
EX: Ghost[1] Exited state Ghost.InGame.InPlay.Play.Running.Free
EX: Ghost[1] Exited state Ghost.InGame.Levels.Basic
EN: Ghost[1] Entered state Ghost.InGame.InPlay.Play.Initalization
EN: Ghost[1] Entered state Ghost.InGame.Levels.Intermediate
EV: 45664405 879: Ghost[2].nextLevel
EX: Ghost[2] Exited state Ghost.InGame.InPlay.Play.Running.Jail
EX: Ghost[2] Exited state Ghost.InGame.Levels.Basic
EN: Ghost[2] Entered state Ghost.InGame.InPlay.Play.Initalization
EN: Ghost[2] Entered state Ghost.InGame.Levels.Intermediate
EV: 45664408 880: Ghost[3].nextLevel
...
```

Figure 3.2: Part of a textual representation of a state-based trace of PacMan [55]

Similar to this work, our research involves run-time monitoring of automatically generated code from models and representation of traces in textual form. However, our work focuses on collecting traces during the execution of real-time models, i.e., UML-RT models. Timing is a very important aspect of real-time software modeling. Therefore, the traces we generated contain timing information which enables us to detect even very small latency issues. One of the important challenges mentioned in [55] for model-based trace generation is the minimization of runtime overhead. For our research we have used the LTTng tool which has the advantage of having low runtime-overhead [64, 43].

Automated transformation of models into executable code for enabling rapid system development is becoming increasingly common. In [49], an architecture has been proposed for debugging models that execute on target systems or in dedicated rapid-prototyping environments. This allows the definition of various debugging perspectives and views independent of the actual execution platform. Similar to their approach, we have also used automated model transformation for generating executable code. However, while our work focuses on using traces to monitor the execution of generated code and to verify the timing requirement specifications, their effort involves animating model execution in the context of model debugging.

A self-adaptive software (SAS) system needs to modify its behaviour at runtime in order to respond to the changes within the system or in its execution environment. One of the key challenges related to the development of these systems is quality assurance. In contrast to traditional software development where correctness of the software can be assured through a variety of activities and processes performed at development time such as, design analysis and testing, developing SAS may need additional assurance tasks at runtime with little or no human intervention. A promising approach for managing complexity in a runtime environment is using adaptation mechanisms that exploit software models. This approach is referred to as models@run.time, a.k.a. M@rt [34]. The authors in [39] present a research agenda regarding quality assurance using M@rt. As they point out, a key challenge for the software engineering community is to develop runtime assurance techniques for SAS that provide high performance, high confidence, and reconfigurable operation in the presence of uncertainties. They present a review of the relevant works that focus on the use of models at runtime during the adaptation process. In our work, we have also used models for quality assurance. However, we have mainly used models for generating executable code with monitoring features. In addition, we have used models for displaying trace results through animating the model elements.

In [35], the authors introduce an approach to utilize runtime models for reflecting the state of an interactive system and modifying its underlying configuration. The use of executable models allows them to build systems that are aware of context information in addition to their own state. The approach has been applied for building adaptive user interfaces by integrating a task and a context model with a set of additional user interface models. This approach creates a feedback loop between model and UI where external stimulations influence the model execution, thus allowing the dynamic alteration of the user interfaces. Our work has some similarities with this effort in the sense that our UI also communicates with the underlying model and exchanges information. This mechanism allows us to map the trace results back to the model by highlighting the relevant modeling elements.

In another related work [54], a discussion on runtime verification is presented. In addition to defining the terminology, the authors present a comparison of runtime verification with well-known verification techniques, i.e., model checking and testing. According to their discussion, runtime verification is a process of dealing with verification techniques that allow us to check if an execution of a system satisfies or violates a given correctness property.

There is also some work done that involves highlighting, animating and simulating of model elements. In [47] [45] [58], several techniques have been proposed for animating and simulating a process. Similar to these efforts, our work also involves highlighting and animating model elements. However, our research is based on "LT-Tng" traces that have been collected during code execution. We only highlight a model element that is being traced.

There is also some recent work done on model execution. For instance, Moka [27] and Moliz [56] are recent tools for model debugging and animating model execution.

Both of the environments have an fUML-based customizable model execution engine [26]. Our method is different from their work due to the distinct nature of the executable models. In our work, we have generated code from models and then the generated code has been executed rather than executing the model itself. And our animation process depends on the traces we retrieve from the executing code. This allows users to visualize only the states or transitions they want to monitor.

There are also a number of industrial tools available for modeling, monitoring, simulating and visualizing electronic control systems in, e.g., the automotive and aerospace domains. For instance, IBM Rational RoseRT [6], IBM RSA-RTE [5], National Instruments (LabVIEW) [8], MathWorks (Simulink) [21] and dSPACE [2]. Among these tools, [5] and [6] can be used for developing UML-RT models and support runtime monitoring of these models to a certain extent. But these tools are proprietary software and it makes their use limited for academic and research purposes. In contrast, our research is completely based on an open source platform, and thus, highly customizable.

In a related work, we have discussed the monitoring of real-time embedded systems along with animation, simulation and integrated debugging [41]. In another related work [28], we have described a case study of our work for runtime monitoring of a Rover.

Chapter 4

Using LTTng to check real-time specifications of models of real-time embedded systems

As mentioned in previous chapters, one of the key factors in the design of a real-time embedded system (RTES) is the proper implementation of timing requirements. Once a problem occurs in a RTES, it is not easy to find out the source of the problem. In MDD, it is possible to generate code for RTES automatically from conceptual models using a number of UML-RT development tools. For this research, we have focused on a particular tool called Papyrus-RT and have added support for tracing UML-RT models in Papyrus-RT. With the provision of this support, it is now feasible to detect timing violations.

In the beginning of this chapter we present an overview of the project. This is followed by a brief discussion of two of the three main sections of this project which have been carried out by two other members. Then, in Section 4.4, we present a detailed discussion on the part we have worked on, *tracing using the LTTng tool along with trace display.* Finally, in Section 4.5, we discuss the implementation details.

4.1 Project Overview

This section gives an overview of the project we have worked on. This project comprises three distinct, but related sections:

- Monitoring Configuration
- Generating code from models
- Tracing of automatically generated code and displaying trace results

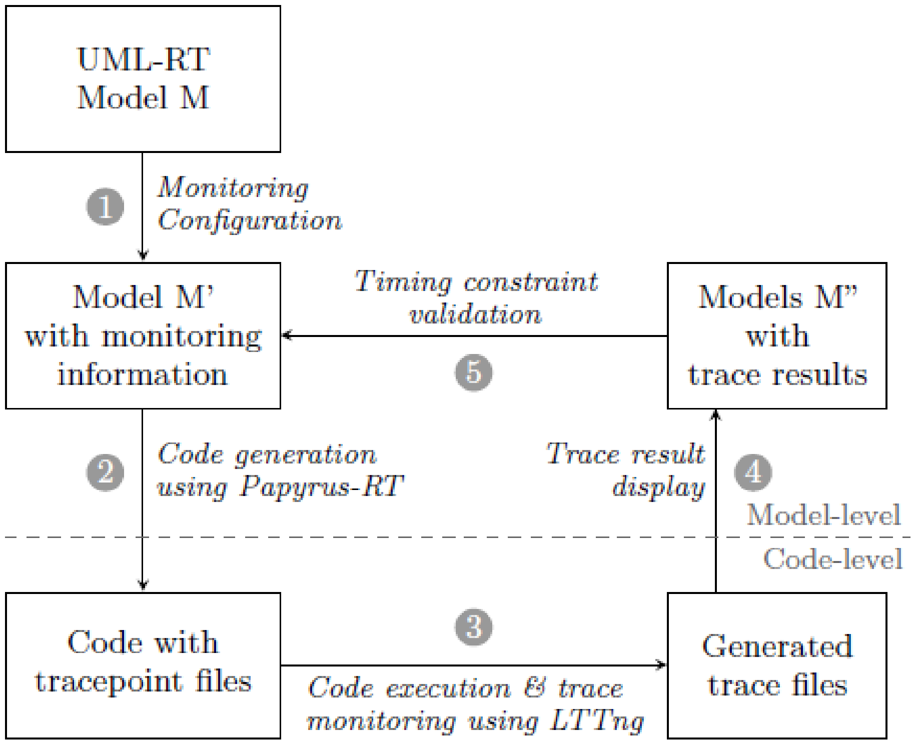

UML-RT Model M
1
Monitoring Configuration
Model M' with monitoring information
Timing constraint validation
5
Models M" with trace results
2
Code generation using Papyrus-RT
Trace result display
4
Model-level
Code-level
Code with tracepoint files
3
Code execution & trace monitoring using LTTng
Generated trace files

Figure 4.1: Monitoring Overview [41]

Each of these sections has one or more subsections. Among the structural and behavioural properties of a UML-RT model, this research focuses on monitoring only the behavioural components. Fig. 4.1 presents an overview of this project. As we can see in this figure, the project cycle begins with a UML-RT model named M in Papyrus-RT. With the addition of monitoring information related to, for instance, a particular state or transition of M, an updated model M' is generated. Then, a modified code generator is used for generating code from the M' model using the Papyrus-RT tool. As monitoring information is added in the first step, we get code with tracepoint files, which are necessary for tracing the generated code with LTTng. We compile the generated code using a make file [11], which is customized to compile code with the LTTng library and tracepoint files. The make file is automatically generated along with the code. After compilation we run the code and trace it using LTTng. Once tracing is done, we get the trace file. We read the trace file and display the trace result on the model level by highlighting the model element. We also display the traces in a textual form. In addition, we have also implemented live tracing which can be used for tracing generated code and displaying traces in the model level at the same time i.e., the execution does not have to have terminated before displaying the trace in the model can begin. Therefore, if a user identifies a problem in the model while observing the traces, s/he can modify the model accordingly. This process allows a model to be refined appropriately. It is also possible to apply this process multiple times.

4.2 Monitoring Configuration

For tracing code automatically generated from a model, tracing information needs to be attached to the model. This is done in this project by adding monitoring information within model elements that are to be traced. With the help of a UI

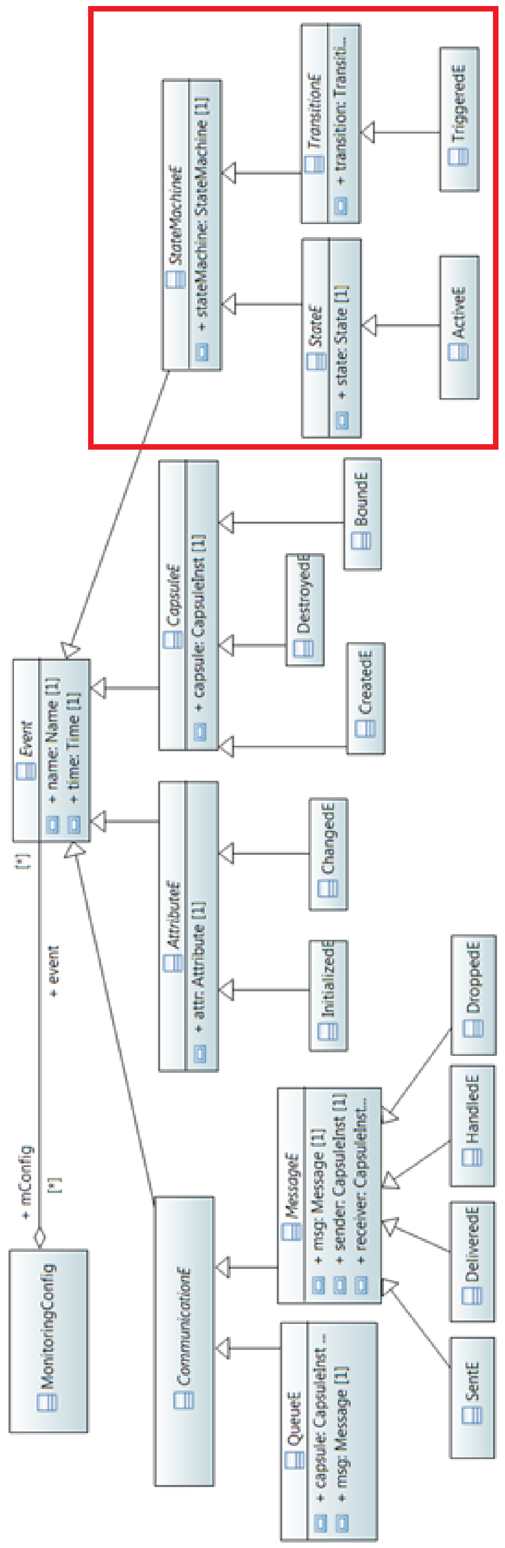


Figure 4.2: Class diagram of monitoring configuration

called the *monitoring configuration UI*, model elements that need to be monitored can be selected.

The purpose of the monitoring configuration is to collect a list of events that are intended to be monitored. Fig. 4.2 shows the class diagram of the events that could be monitored in a model. In this research, we have only worked on monitoring the behavioural aspects of the model. Therefore we only monitor active states and triggered transitions of the state-machine of a capsule which is indicated in the red rectangular box in Fig. 4.2.

To make the generated code traceable we have added monitoring information within the selected model elements, which is done by adding a monitoring profile to model elements. The user can also select a pair of model elements and associate a time value with it which can then be used to validate timing constraints.

4.2.1 Monitoring Profile

To monitor a selected element, the information related to the selected element is stored in a UML profile. In our case, we call it *monitoring profile*. Profiles can be attached to a model. Once a profile is attached to a model, the stereotypes it contains can be attached to the appropriate elements of the model. These stereotypes can have attributes which can also be added to the model elements.

In our research, a monitoring profile named “LTTng profile” is used for attaching monitoring information to model elements. Fig. 4.3 shows the profile we have used to monitor behavioural elements of a model. Here, the profile contains three stereotypes for monitoring three types of model elements. Each of the three stereotypes contains one Boolean attribute named “isMonitored”. When any state/transition is selected

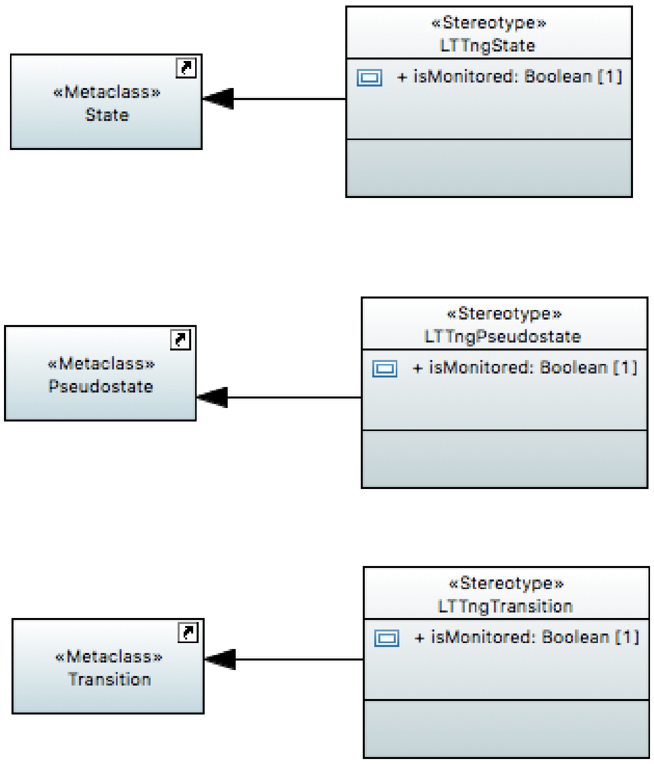


Figure 4.3: LTTng Profile

for monitoring through the monitoring configuration UI, the appropriate stereotypes are attached to the model element and the Boolean attribute "isMonitored" is set to

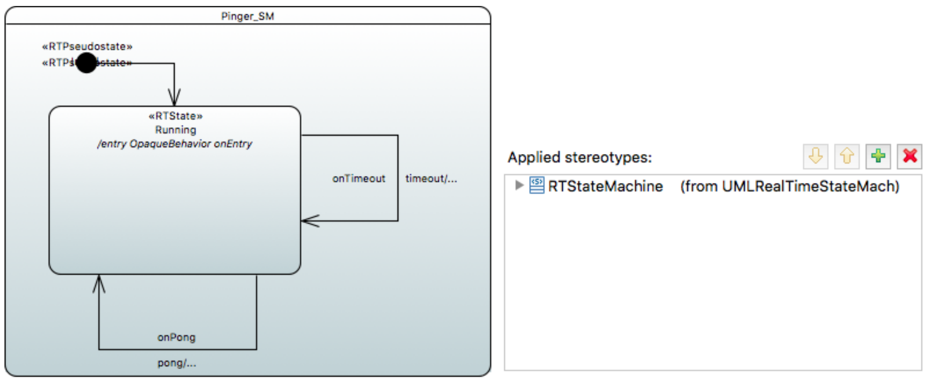
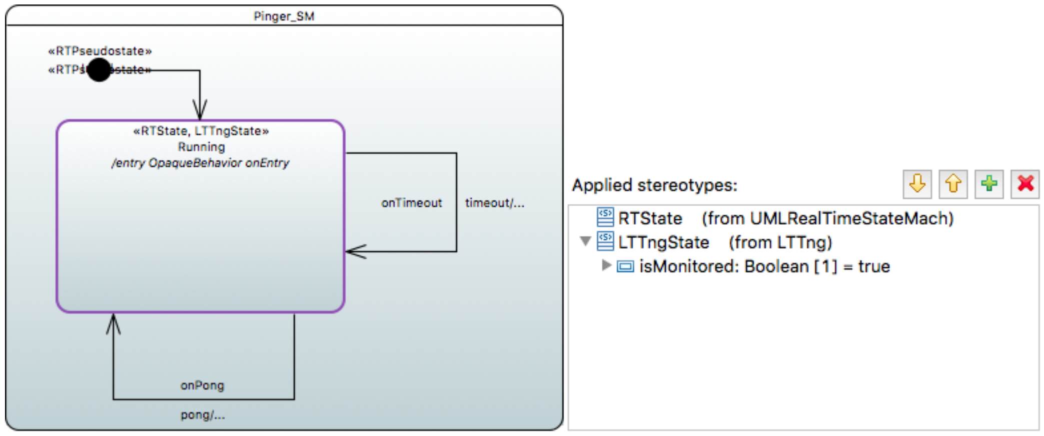


Figure 4.4: The Running state and properties before adding monitoring information



RTState (from UMLRealTimeStateMach)
LTTngState (from LTTng)
isMonitored: Boolean [1] = true

Figure 4.5: The Running state and properties after adding monitoring information

"true". This indicates the code generator to generate traceable code for LTTng. For example, Fig. 4.4 presents a state-machine named "Pinger_SM".

Fig. 4.5 shows the view of the "Running" state and its applied stereotype properties. As we can see in this figure, the "LTTngState" stereotype is applied and the Boolean attribute is set to *true*. The purple border-color of the "Running" state indicates the attachment of monitoring information to the state. This makes it visually different from a state which does not have any attached monitoring information.

4.2.2 Creating pair of model elements

Another feature allows the user to validate a desired timing specification. This can be done by selecting a couple of elements (e.g., a state or a transition) from a state-machine to be monitored and creating a pair out of the selected elements. The pair

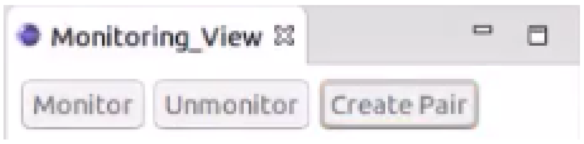
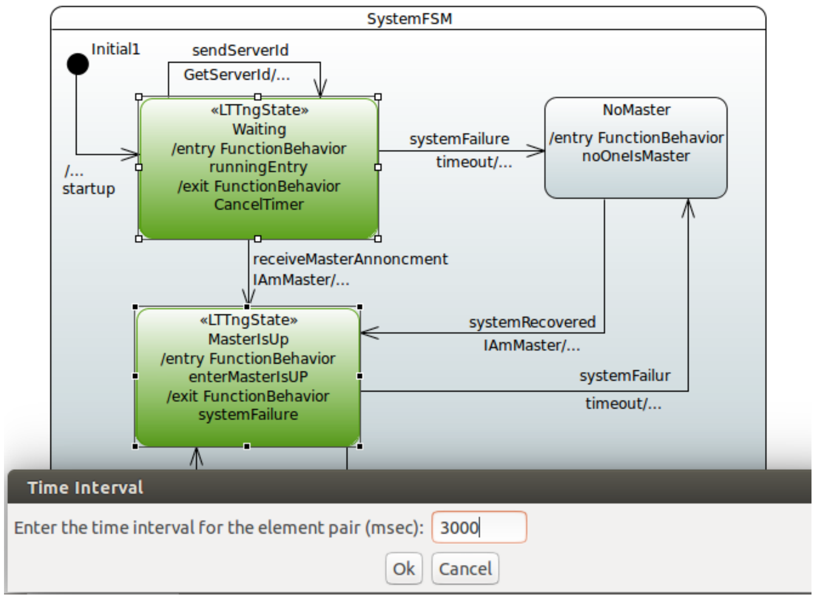


Figure 4.6: UI for supporting monitoring configuration



Enter the time interval for the element pair (msec): 3000

Figure 4.7: Dialog for inputting a time interval between two selected model elements

can be created with the specification of a time duration in milliseconds. This allows us to verify if the selected monitoring events occur within the given time difference. For example, this feature can be used for checking whether transitioning from a state S1 to another state S2 takes more than 3 seconds or not. The specification file is added to the model in the same directory and it contains the selected state and transition names, the name of the owner capsule and the specified time value in milliseconds. Fig. 4.6 shows the Eclipse UI of the implemented plugin for creating a pair of monitored events. Here, a button “Create Pair” is used to generate a specification file with the information of the selected model elements and the given time value. Fig. 4.7 shows the dialog allowing the user to input the timing value. The two states are highlighted with green background to indicate that they have been selected for verifying a time constraint.

4.3 Code Generation

Code generation is the process of generating code from models. The Papyrus-RT code generator is designed in the way that can be extended. This feature has allowed us to extend the code generator of Papyrus-RT for generating monitoring information along with the code of models. As we discussed in Chapter 2, tracepoint files are necessary for tracing the generated code with LTTng, therefore the code generator is extended to support tracing using LTTng. The monitoring configuration UI is used for selecting elements to be monitored. The selected elements become the user input for the code generation process.

Listing 4.1: Sample code generated from model

```
Capsule_Workstation::State Capsule_Workstation::state_____top__Workstation_Producing( const
    UMLRTMessage * msg )
{
    switch( msg->destPort->role()->id )
    {
    case port_ProductionTimer:
        switch( msg->getSignalId() )
        {
        case UMLRTTimerProtocol::signal_timeout:
            actionchain_____top__Workstation_finished__ActionChain4( msg );
            return top__Workstation_Standby;
        default:
            this->unexpectedMessage();
            break;
        }
        return currentState;
    default:
        this->unexpectedMessage();
        break;
    }
    return currentState;
}
```

The customized code generator of Papyrus-RT takes a UML-RT model and outputs a C/C++ Development Tooling (CDT) project. CDT is known as a set of tools that is provided by Eclipse to facilitate C and C++ development. The generated code using the customized code generator also contains tracepoint (.tp) files for all the events that have been selected for monitoring. The tracepoint() calls are added to the code generated from the selected model elements.

Listing 4.2: Generated Code with Tracepoints

```
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;
}
```

For instance, Listing 4.1 shows the code generated for the "Producing" state of the "Workstation" capsule without having any monitoring information attached. On the other hand, Listing 4.2 shows the code generated with tracepoint() calls, where tracepoint() calls are highlighted in red. The first tracepoint() call indicates the active state and triggers the associated tracepoint file when the state is entered. This function executes every time this state gets activated. The other tracepoint() call is placed to indicate the triggered transition and this tracepoint function executes every time the transition gets triggered. These two tracepoint() calls are added because they are selected in the monitoring configuration UI at the very first step. The generated code is then compiled with the tracing feature using a make (.mk) [11] file, which is also customized for compiling code with LTTng tracepoints and automatically generated with the code using the customized code generator of Papyrus-RT.

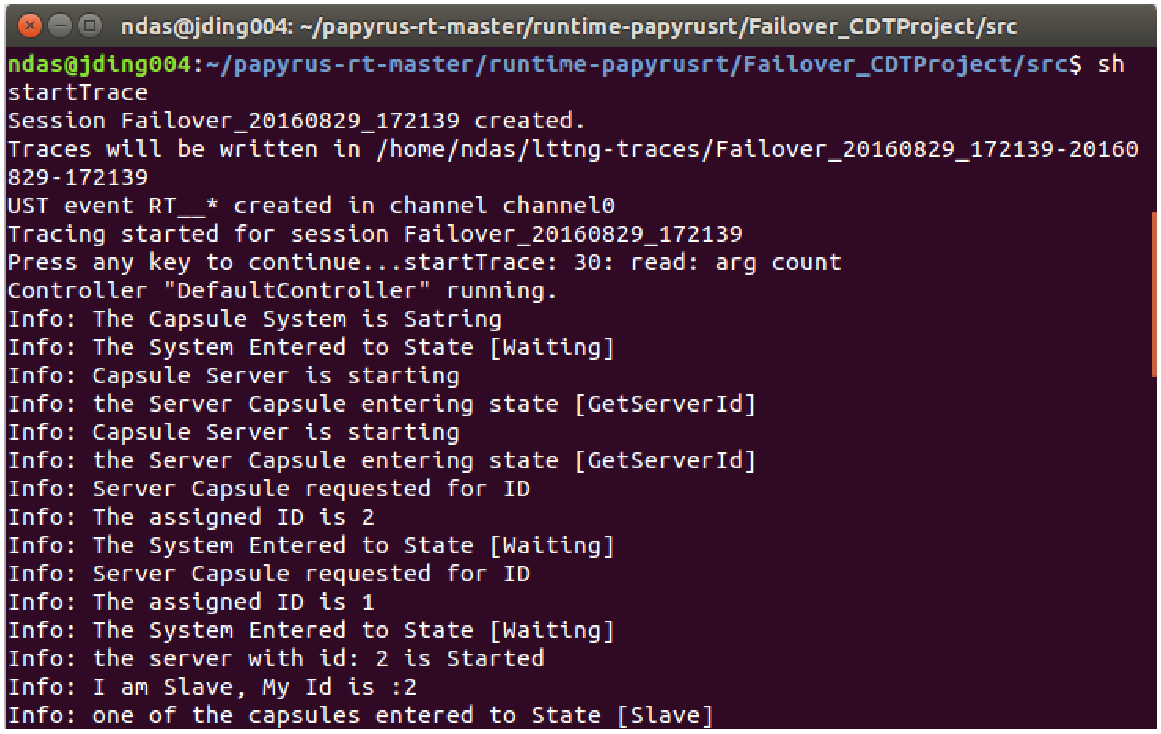
4.4 Tracing using LTTng and Trace Display

Tracing in RTES is a technique to monitor a running real-time embedded system. In this section, we describe the steps for tracing generated code and displaying trace results. Our research focuses on this part of the project.

In addition to the code, two scripts are generated that contain all the LTTng commands are necessary for tracing and need to be run to monitor the execution. One of these scripts is used to start tracing and executing the generated code, while the other one is used to stop tracing. Once the code is generated and compiled, we execute the code and trace it through the use of the scripts. This makes it very easy to trace the generated code, because users do not have to give a list of LTTng commands for tracing.

Fig. 4.8 displays the execution of the code and the use of the script to start tracing. Here, "startTrace" is the name of the script used to start tracing. The script contains LTTng commands for starting a *session*, enabling the associated user space *events*, starting the process of *tracing* and starting the execution of the generated code.

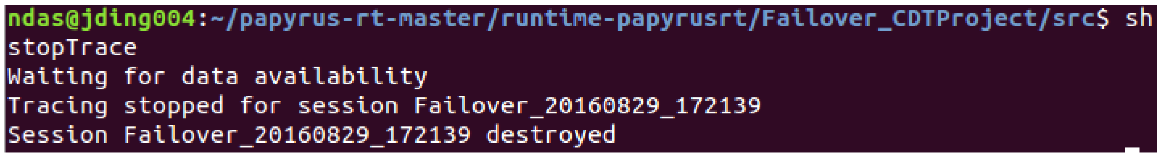
The trace is written to a file on a specific path. Once tracing is started, we can stop tracing at any point. We use the generated script to stop tracing and to destroy the


ndas@jding004: ~/papyrus-rt-master/runtime-papyrusrt/Failover_CDTProject/src

```
ndas@jding004:~/papyrus-rt-master/runtime-papyrusrt/Failover_CDTProject/src$ sh
startTrace
Session Failover_20160829_172139 created.
Traces will be written in /home/ndas/lttng-traces/Failover_20160829_172139-20160
829-172139
UST event RT__* created in channel channel0
Tracing started for session Failover_20160829_172139
Press any key to continue...startTrace: 30: read: arg count
Controller "DefaultController" running.
Info: The Capsule System is Satring
Info: The System Entered to State [Waiting]
Info: Capsule Server is starting
Info: the Server Capsule entering state [GetServerId]
Info: Capsule Server is starting
Info: the Server Capsule entering state [GetServerId]
Info: Server Capsule requested for ID
Info: The assigned ID is 2
Info: The System Entered to State [Waiting]
Info: Server Capsule requested for ID
Info: The assigned ID is 1
Info: The System Entered to State [Waiting]
Info: the server with id: 2 is Started
Info: I am Slave, My Id is :2
Info: one of the capsules entered to State [Slave]
```

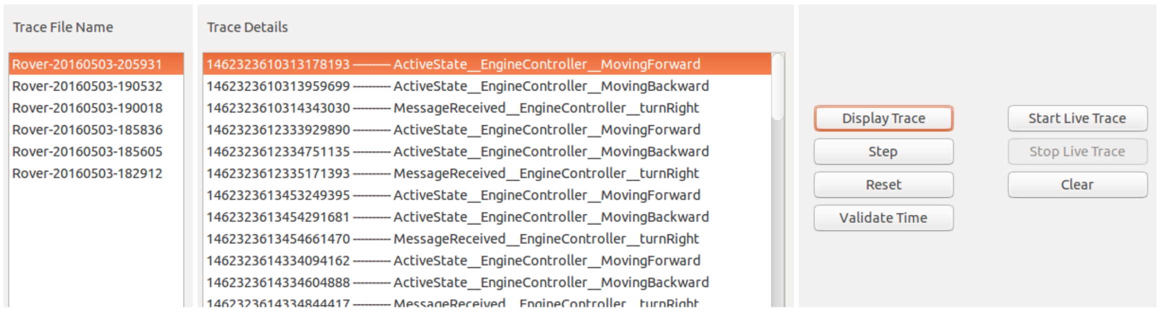
Figure 4.8: Tracing of generated code

current session (see Fig. 4.9). Once the trace file is generated, we use it to display the trace results on the model level. There is different information that can be displayed. There are also many ways to display trace results. In our research we have used two types of visualization. One is to display the traces in textual form and the other one is to display traces through the highlighting of model elements.



```
ndas@jding004:~/papyrus-rt-master/runtime-papyrusrt/Failover_CDTProject/src$ sh
stopTrace
Waiting for data availability
Tracing stopped for session Failover_20160829_172139
Session Failover_20160829_172139 destroyed
```

Figure 4.9: Running a script to stop tracing



```
Rover-20160503-205931
Rover-20160503-190532
Rover-20160503-190018
Rover-20160503-185836
Rover-20160503-185605
Rover-20160503-182912
```

```
1462323610313178193 ——— ActiveState__EngineController__MovingForward
1462323610313959699 ——— ActiveState__EngineController__MovingBackward
1462323610314343030 ——— MessageReceived__EngineController__turnRight
1462323612333929890 ——— ActiveState__EngineController__MovingForward
1462323612334751135 ——— ActiveState__EngineController__MovingBackward
1462323612335171393 ——— MessageReceived__EngineController__turnRight
1462323613453249395 ——— ActiveState__EngineController__MovingForward
1462323613454291681 ——— ActiveState__EngineController__MovingBackward
1462323613454661470 ——— MessageReceived__EngineController__turnRight
1462323614334094162 ——— ActiveState__EngineController__MovingForward
1462323614334604888 ——— ActiveState__EngineController__MovingBackward
1462323614334844417 ——— MessageReceived__EngineController__turnRight
```

Figure 4.10: Display of traces in textual form

Textual form of trace display

An LTTng trace contains a wide range of information, e.g., a timestamp, start time, end time, *string* field and so on. In our work, we have displayed the timestamp value along with the *string* field, which contains information about monitored element, i.e., the type and name of the monitored element.

We have developed an Eclipse based plugin which displays three columns. The leftmost column shows the names of the trace files after selecting a model in the Eclipse project explorer window. Fig. 4.10 shows the trace display in textual format. Upon selecting any particular trace from the leftmost column, the middle column shows the details of that trace file. Information presented in the middle column gives us an idea about the currently active model element at any particular time. The timestamp information represents the exact time when the event occurred, e.g., when a state became active or when a transition was taken. The timestamp information is represented in nanoseconds. The rightmost column contains *buttons* for user interaction. The "Display Trace" button is used to start displaying traces in textual format and on the model level by highlighting corresponding elements. Then user can go through the trace and display the events in it one by one by using the "Step" button.

The "Reset" button is used for going to the initial position which does not have any elements highlighted. The "Validate Time" button is used for checking the validity of a given time constraint. In our research, we have also worked on *live* tracing where the user can see traces as they are being collected. To start viewing live traces we have added the "Start Live Trace" button. The "Stop Live Trace" button is used to stop the display of live traces and the "Clear" button is used to clear the text area.

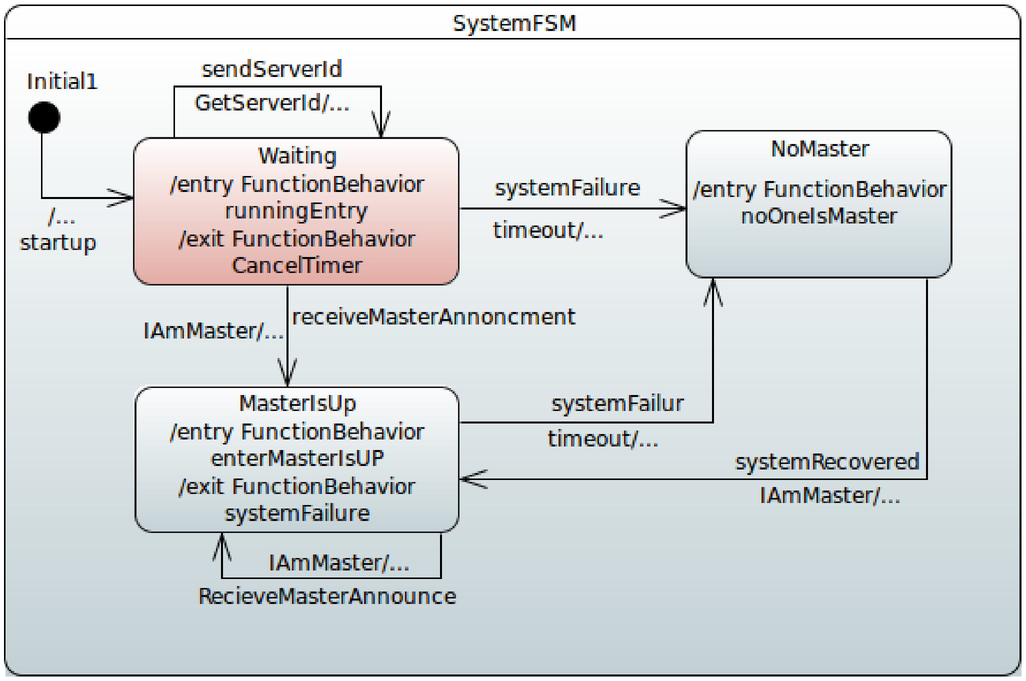
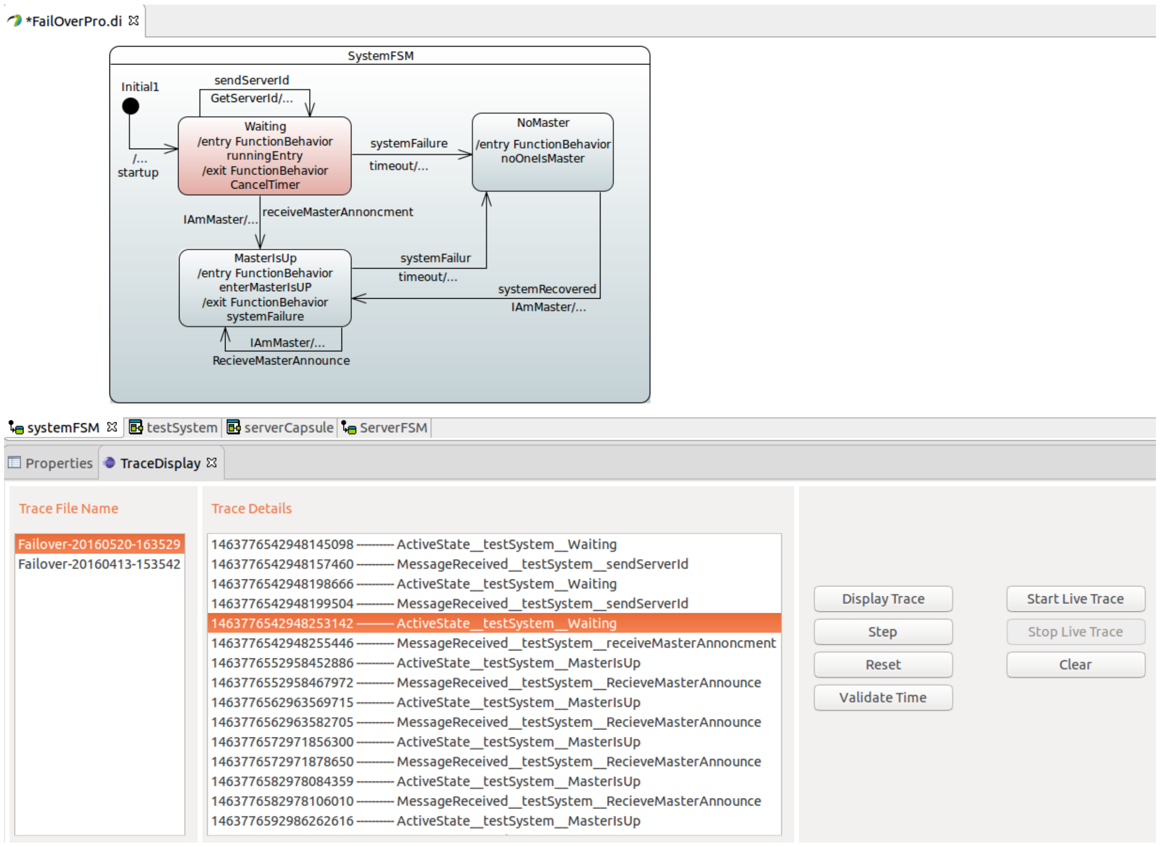


Figure 4.11: Display of traces in Model level

Model level Display of Traces

To make the display more user friendly we show traces on the model level by highlighting the model elements involved in the events in the trace. Fig. 4.11 shows the state-machine of a capsule where the "Waiting" state is highlighted in Red color to indicate that the trace file contains information about the “Waiting” state. The textual and the model level view are synchronized. Therefore, when a user selects the “Display Trace” button, both textual and model level elements get highlighted.



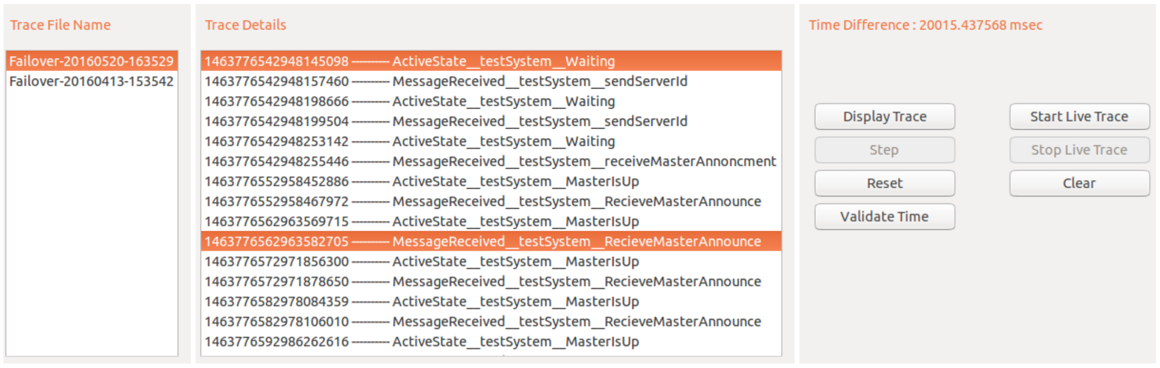
Failover-20160520-163529
Failover-20160413-153542

```
1463776542948145098 --------- ActiveState__testSystem__Waiting
1463776542948157460 --------- MessageReceived__testSystem__sendServerId
1463776542948198666 --------- ActiveState__testSystem__Waiting
1463776542948199504 --------- MessageReceived__testSystem__sendServerId
1463776542948253142 --------- ActiveState__testSystem__Waiting
1463776542948255446 --------- MessageReceived__testSystem__receiveMasterAnnoncment
1463776552958452886 --------- ActiveState__testSystem__MasterIsUp
1463776552958467972 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776562963569715 --------- ActiveState__testSystem__MasterIsUp
1463776562963582705 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776572971856300 --------- ActiveState__testSystem__MasterIsUp
1463776572971878650 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776582978084359 --------- ActiveState__testSystem__MasterIsUp
1463776582978106010 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776592986262616 --------- ActiveState__testSystem__MasterIsUp
```

Figure 4.12: Display of offline trace

4.4.1 Offline Tracing

Offline tracing means tracing that is not live, i.e., that the execution of the monitored code and the collection of traces is completed before the display of the trace. While tracing offline, traces cannot be seen at the time of arrival. Consequently, execution and tracing need to be stopped through user interaction for viewing the offline trace results. Fig. 4.12 shows the display of traces in offline mode where user can see traces both textually and on the model level by using related buttons. It is also possible to select any row from the middle column of the textual view which associates the model element with the row highlighted.



```
Failover-20160520-163529
Failover-20160413-153542
```

```
1463776542948145098 --------- ActiveState__testSystem__Waiting
1463776542948157460 --------- MessageReceived__testSystem__sendServerId
1463776542948198666 --------- ActiveState__testSystem__Waiting
1463776542948199504 --------- MessageReceived__testSystem__sendServerId
1463776542948253142 --------- ActiveState__testSystem__Waiting
1463776542948255446 --------- MessageReceived__testSystem__receiveMasterAnnoncment
1463776552958452886 --------- ActiveState__testSystem__MasterIsUp
1463776552958467972 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776562963569715 --------- ActiveState__testSystem__MasterIsUp
1463776562963582705 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776572971856300 --------- ActiveState__testSystem__MasterIsUp
1463776572971878650 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776582978084359 --------- ActiveState__testSystem__MasterIsUp
1463776582978106010 --------- MessageReceived__testSystem__RecieveMasterAnnounce
1463776592986262616 --------- ActiveState__testSystem__MasterIsUp
```

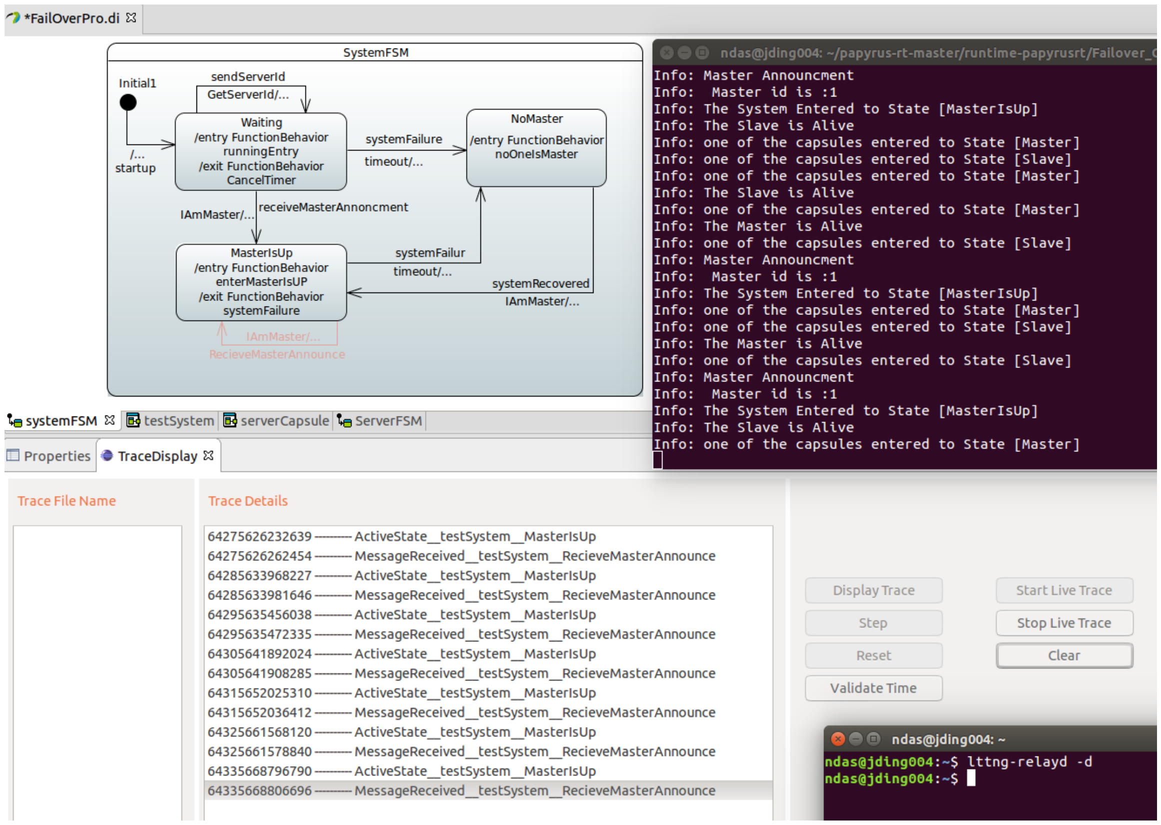
Figure 4.13: Display of time difference through selecting two lines from trace details

Also, selecting two rows from the trace details (i.e., middle column of the textual view) allows us to view the time difference between the selected trace lines. Fig. 4.13 shows the time difference between two selected trace lines. The timestamp shown here was originally in nanoseconds. To make it more user friendly we converted it to milliseconds while displaying the time difference.

4.4.2 Online/Live Tracing

In online/live tracing users can view traces while trace elements are arriving. In our research, we have implemented a feature to start live tracing and read live traces. Similar to the representation of offline traces, we display the live traces both in textual form and on the model level.

Live traces can also be remote traces where users can send trace data from one machine to another. To do this, users need to start a LTTng *relay daemon* [10] in the



```
Info: Master Announcment
Info:  Master id is :1
Info: The System Entered to State [MasterIsUp]
Info: The Slave is Alive
Info: one of the capsules entered to State [Master]
Info: one of the capsules entered to State [Slave]
Info: one of the capsules entered to State [Master]
Info: The Slave is Alive
Info: one of the capsules entered to State [Master]
Info: The Master is Alive
Info: one of the capsules entered to State [Slave]
Info: Master Announcment
Info:  Master id is :1
Info: The System Entered to State [MasterIsUp]
Info: one of the capsules entered to State [Master]
Info: one of the capsules entered to State [Slave]
Info: The Master is Alive
Info: one of the capsules entered to State [Slave]
Info: Master Announcment
Info:  Master id is :1
Info: The System Entered to State [MasterIsUp]
Info: The Slave is Alive
Info: one of the capsules entered to State [Master]
```

```
64275626232639 -------- ActiveState__testSystem__MasterIsUp
64275626262454 -------- MessageReceived__testSystem__RecieveMasterAnnounce
64285633968227 -------- ActiveState__testSystem__MasterIsUp
64285633981646 -------- MessageReceived__testSystem__RecieveMasterAnnounce
64295635456038 -------- ActiveState__testSystem__MasterIsUp
64295635472335 -------- MessageReceived__testSystem__RecieveMasterAnnounce
64305641892024 -------- ActiveState__testSystem__MasterIsUp
64305641908285 -------- MessageReceived__testSystem__RecieveMasterAnnounce
64315652025310 -------- ActiveState__testSystem__MasterIsUp
64315652036412 -------- MessageReceived__testSystem__RecieveMasterAnnounce
64325661568120 -------- ActiveState__testSystem__MasterIsUp
64325661578840 -------- MessageReceived__testSystem__RecieveMasterAnnounce
64335668796790 -------- ActiveState__testSystem__MasterIsUp
64335668806696 -------- MessageReceived__testSystem__RecieveMasterAnnounce
```

```
ndas@jding004:~$ lttng-relayd -d
ndas@jding004:~$ 
```

Figure 4.14: Live tracing

receiver machine. Fig. 4.14 illustrates live tracing. In this figure, the bottom right console is for starting the LTTng relay daemon in the receiver part. The top right console shows the starting of live traces. Here, "localhost" is used as the destination of traces, which means the traces are being sent to the same machine. The Eclipse view starts showing the traces as they arrive after clicking the "Start Live Trace" button.

4.5 Implementation

This section describes our implementation of tracing using LTTng and trace display.

4.5.1 Implementation overview

In the beginning, we have implemented a method to automatically generate two script files along with the code. We have created an Eclipse plug-in (TraceDisplay, in Fig 4.15) to read and display the trace files. Our plug-in is basically an Eclipse view which can be visualized in an Eclipse instance. It is possible to list all the trace file names and traced events for a selected model element in the implemented Eclipse view. We have also implemented support for validating the traced information in accordance with a given time constraint.

For creating the Eclipse plug-in view we have written a class (EclipseView, in Fig 4.15) which inherits from the following class in Eclipse framework:

- org.eclipse.ui.part.ViewPart

A form has been created inside the Eclipse view for listing traces and to support user interaction. For the creation of the form and to add buttons and labels for user interaction, we have used the following packages:

- org.eclipse.swt.widgets
- org.eclipse.ui.forms.widgets

For selecting a particular model in the Eclipse project explorer window and get the full path of it, we have used:

- org.eclipse.core.runtime.IPath
- org.eclipse.jface.viewers
- org.eclipse.ui.ISelectionListener

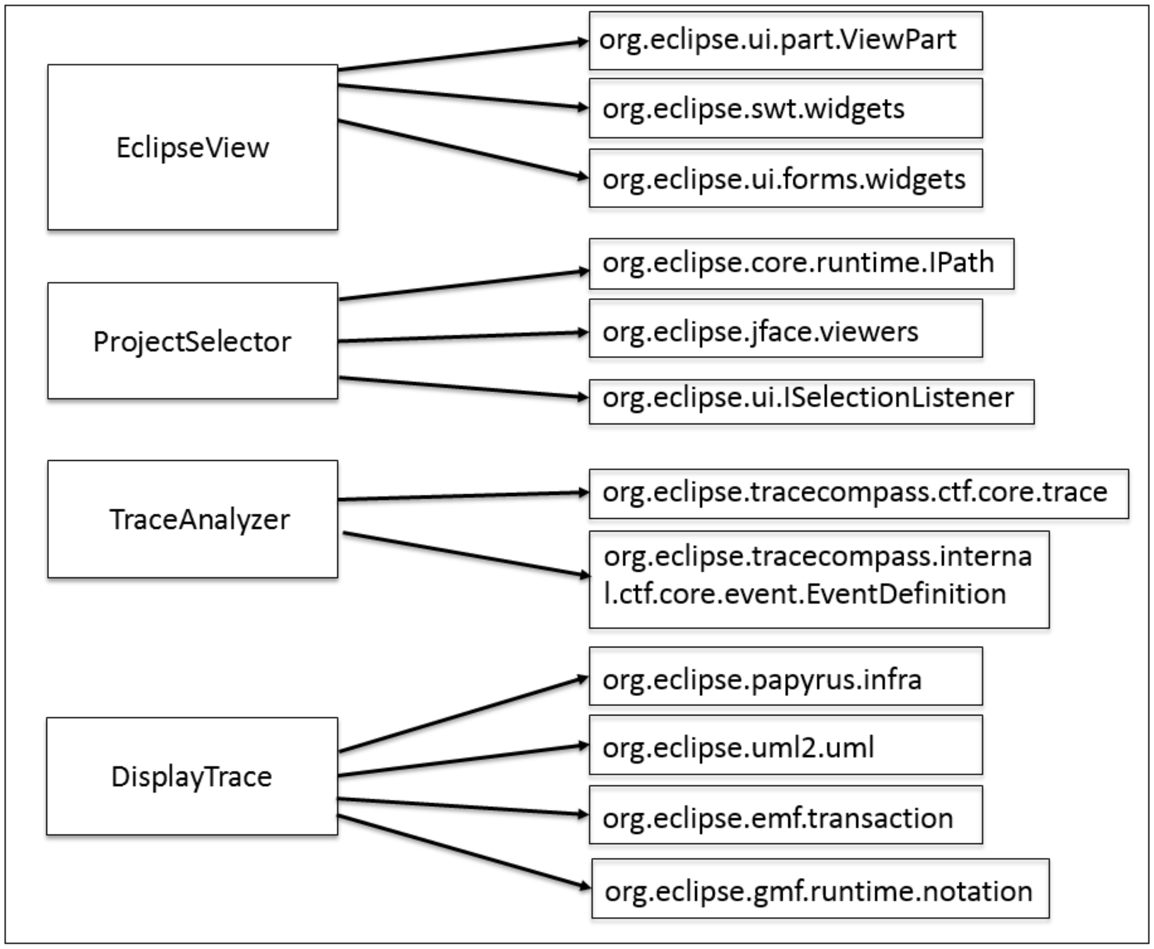

EclipseView
org.eclipse.ui.part.ViewPart
org.eclipse.swt.widgets
org.eclipse.ui.forms.widgets
ProjectSelector
org.eclipse.core.runtime.IPath
org.eclipse.jface.viewers
org.eclipse.ui.ISelectionListener
TraceAnalyzer
org.eclipse.tracecompass.ctf.core.trace
org.eclipse.tracecompass.internal.ctf.core.event.EventDefinition
DisplayTrace
org.eclipse.papyrus.infra
org.eclipse.uml2.uml
org.eclipse.emf.transaction
org.eclipse.gmf.runtime.notation

Figure 4.15: Implementation overview of "TraceDisplay" plug-in (Left column represents the classes inside the plug-in, right column represents the imported packages in the associated classes)

For extracting trace event details from a trace file we have written a class "Trace-Analyzer" (see Fig. 4.15) using the "Trace Compass" library for reading the generated traces. We have used the following packages while writing this class:

- org.eclipse.tracecompass.ctf.core.trace
- org.eclipse.tracecompass.internal.ctf.core.event.EventDefinition

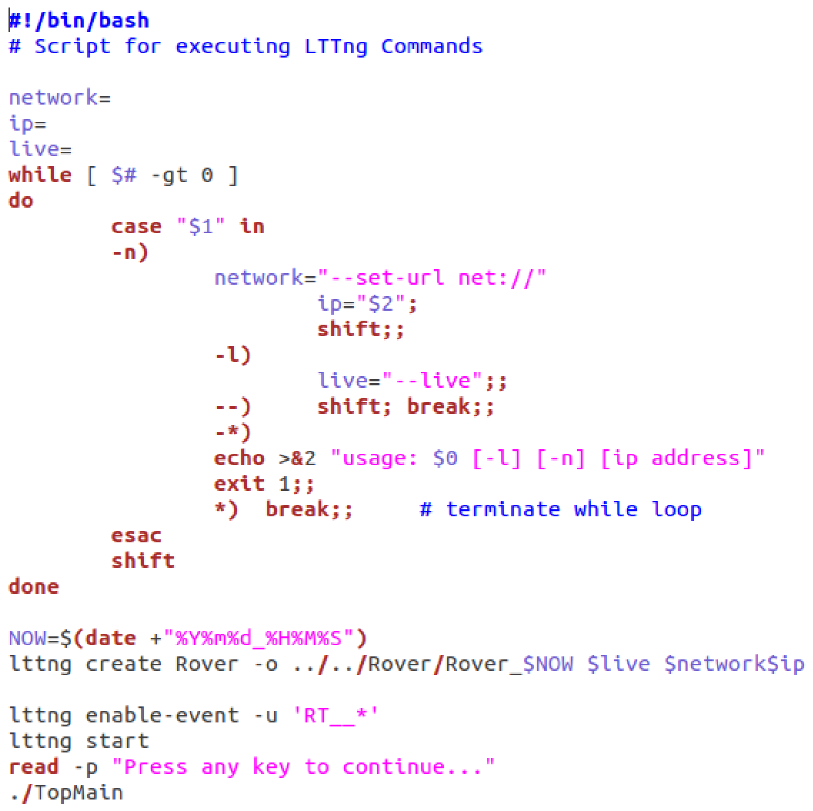
Once all the trace events are enlisted on the eclipse view, we need to retrieve the exact model element for showing the traces on the model. We have collected the exact model element using the "org.eclipse.papyrus.infra" and the "org.eclipse.uml2.uml" packages. To change the color of the model element we have used following packages:

- org.eclipse.emf.transaction
- org.eclipse.gmf.runtime.notation

An overview of the packages we have used for doing some particular tasks is presented in Fig. 4.15.

4.5.2 Script Generation

We have used Xtend to generate scripts automatically along with the code. Initially we have prepared a *shell script* [20] with all the LTTng commands to start and stop tracing. Then, we have used Xtend for generating the script with all the required LTTng commands (see Appendix A). Fig. 4.16 shows a screenshot of an automatically generated script for the Rover model. Here, the "lttng create" command is used for creating a tracing session. This is followed by the specification of the session name. We have used the model name for naming the session. Command line parameter "-o" followed by a path indicates the output path of the trace. The output path is automatically generated by our plugin from the path of the selected model. We have used user input for live and network tracing. Once a session is created, all the events that are marked for tracing become enabled by the use of the appropriate LTTng command. Then, tracing is started using the "lttng start" command and it can be used to trace the program that is being executed. It is worth noting here that we start tracing before executing the generated code. This protects us from missing any event that can occur between the start of the tracing and the execution.



```
#!/bin/bash
# Script for executing LTTng Commands

network=
ip=
live=
while [ $# -gt 0 ]
do
        case "$1" in
        -n)
                network="--set-url net://"
                        ip="$2";
                        shift;;
                -l)
                        live="--live";;
                --)     shift; break;;
                -*)
                echo >&2 "usage: $0 [-l] [-n] [ip address]"
                exit 1;;
                *)  break;;     # terminate while loop
        esac
        shift
done

NOW=$(date +"%Y%m%d_%H%M%S")
lttng create Rover -o ../../Rover/Rover_$NOW $live $network$ip

lttng enable-event -u 'RT__*'
lttng start
read -p "Press any key to continue..."
./TopMain
```

Figure 4.16: Screenshot of a generated script for Rover model

We have also generated a script named “stopTrace” to stop tracing. This script contains only two commands. One is “lttng stop” to stop tracing and the other one is “lttng destroy” to destroy the current LTTng session.

4.5.3 Reading of Trace

Reading a trace file is the process of reading offline and online traces. Initially it has been done by selecting the path of the trace file by the user. This allowed us to read only offline traces. Later, we have extended our plugin to read both offline and online traces without requiring the user to specify the path manually.

Offline

To read offline traces, the path of the model directory is retrieved by selecting the model name in the project explorer within the Eclipse instance. Once the path of the model directory is specified, we analyze if the directory contains any trace files. If any trace file is found, we display the name of the trace file in the leftmost column of the plugin view. Then, upon selecting of a trace file name from the leftmost column we get the exact path of the trace file.

LTTng relies on the Common Trace format (CTF) [42], an optimized format for producing and analyzing large amounts of data, to produce trace files with a low overhead [43]. We have used the “Trace Compass” [22] library to read these trace files. To read the traces of a selected trace file we go through the events one by one and read the timestamp of that event. We also read the “String field” attribute which contains the type of trace, the capsule name and the name of the model element involved (e.g., the state or transition). We merge the timestamp and string field information and store it in a list.

Online

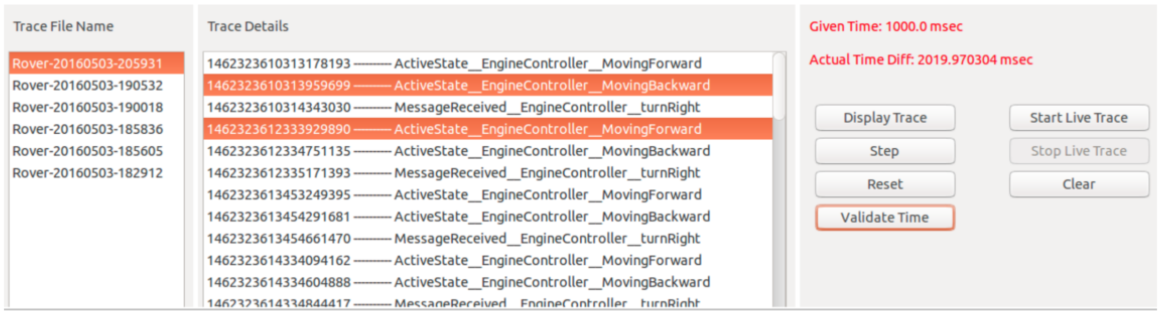
For reading online traces first we need to start the relay daemon on the receiving end. This is done with the “lttng-relayd -d” command. Here, “-d” is used to run the relay daemon in the background. After starting relay daemon, we have used “Babeltrace” [1] to read live traces. We use the “Start Live Trace” button in the plugin UI for reading live traces. Similar to offline traces, we read the timestamp value and the string field and then we merge both and store them in a list.

4.5.4 Display Traces

Once trace files are read, the next part is to display traces. Displaying traces in the textual view is done by printing all the traced information in the middle text area of the view part. In contrast, displaying traces on the model level is done by highlighting the associated model elements. At first, we implemented the latter by changing the contents of the “notation” file which is associated with the model. That was not user-friendly as it requires reloading the Eclipse window to show the changes. Later, for improving the user experience, we have used the setFillColor property of the model element to highlight it.

4.5.5 Time constraint validation

The main focus of this research is to discover time constraint violations. To do this, we have read the specification file containing user-specified timing value for any two elements (state/transition) from a state-machine. The specification file is made available in a directory similar to the model. Therefore, we get the path of the specification file by selecting the model in the project explorer window in Eclipse. The format of the contents of the specification file is “type of element(state or transition)_capsule



```
Rover-20160503-205931
Rover-20160503-190532
Rover-20160503-190018
Rover-20160503-185836
Rover-20160503-185605
Rover-20160503-182912
```

```
1462323610313178193 ——— ActiveState__EngineController__MovingForward
1462323610313959699 ——— ActiveState__EngineController__MovingBackward
1462323610314343030 ——— MessageReceived__EngineController__turnRight
1462323612333929890 ——— ActiveState__EngineController__MovingForward
1462323612334751135 ——— ActiveState__EngineController__MovingBackward
1462323612335171393 ——— MessageReceived__EngineController__turnRight
1462323613453249395 ——— ActiveState__EngineController__MovingForward
1462323613454291681 ——— ActiveState__EngineController__MovingBackward
1462323613454661470 ——— MessageReceived__EngineController__turnRight
1462323614334094162 ——— ActiveState__EngineController__MovingForward
1462323614334604888 ——— ActiveState__EngineController__MovingBackward
1462323614334844417 ——— MessageReceived__EngineController__turnRight
```

```
Given Time: 1000.0 msec
Actual Time Diff: 2019.970304 msec
```

Figure 4.17: Time constraint validation

name_state/transition name". For checking the violation of a time constraint, we retrieve the particular lines of trace using the names of the capsule and the state/-transition of the model element. After calculating the time difference, we compare it with the given time value. If the actual time difference does not exceed the given time value, we display "Validated". Otherwise, we display the given time difference and actual time difference in milliseconds. If there is a violation of the time constraint, we also highlight the associated trace lines in the textual view. Fig. 4.17 shows the view of the trace display after finding a violation for a given time constraint.

Chapter 5

Proof of Concept

This chapter presents an proof of concept of our implementation through conducting three case studies. We have used two Papyrus-RT models, the PingPong and the Workstation, for exercising two features of our implementation: offline tracing and trace display. In addition, we have used another Papyrus-RT model, namely the Rover model, to apply the live tracing feature of the implemented plugin on a larger model. We present a brief discussion of the structural and behavioural design of the three models in the following three sections. We also present the result of tracing these models using our plugin.

5.1 PingPong

Pingpong is a very simple model available in the sample model repository of the Papyrus-RT distribution. It has a Top capsule which contains two other capsules: Pinger and Ponger. Fig. 5.1 shows the structure diagram of the Top capsule of the PingPong model. As we can see in this figure, the Pinger and the Ponger capsules can communicate with each other through a protocol named “PingPongProtocol”. They send signals to each other using the real-time ports they own.

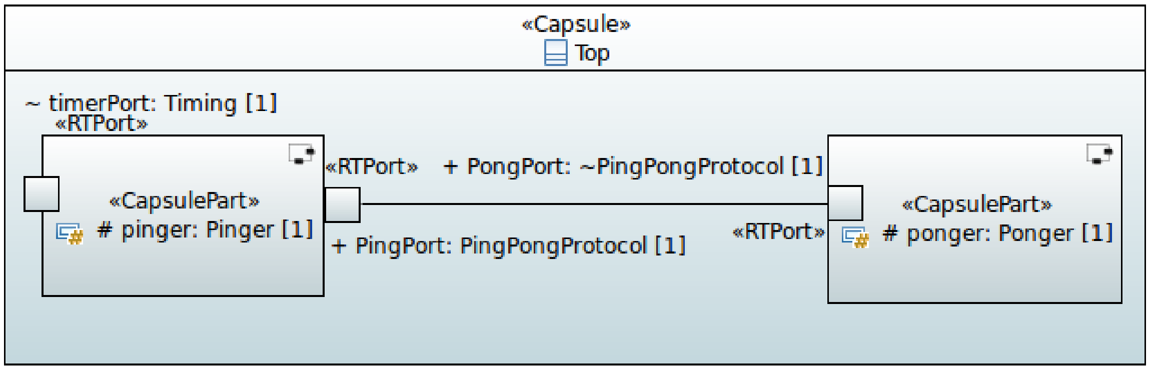


Figure 5.1: Structure diagram of PingPong model

Both Pinger and Ponger have their own state-machine behaviour. Fig. 5.2 shows the state-machine diagram of the Pinger capsule. Pinger has a state named "Running" and can send a "onPing" signal to the Ponger capsule when it is in the "Running" state. The "Running" state has a self transition which can be triggered if it receives the "onPong" signal from the Ponger capsule. It has another self-transition which

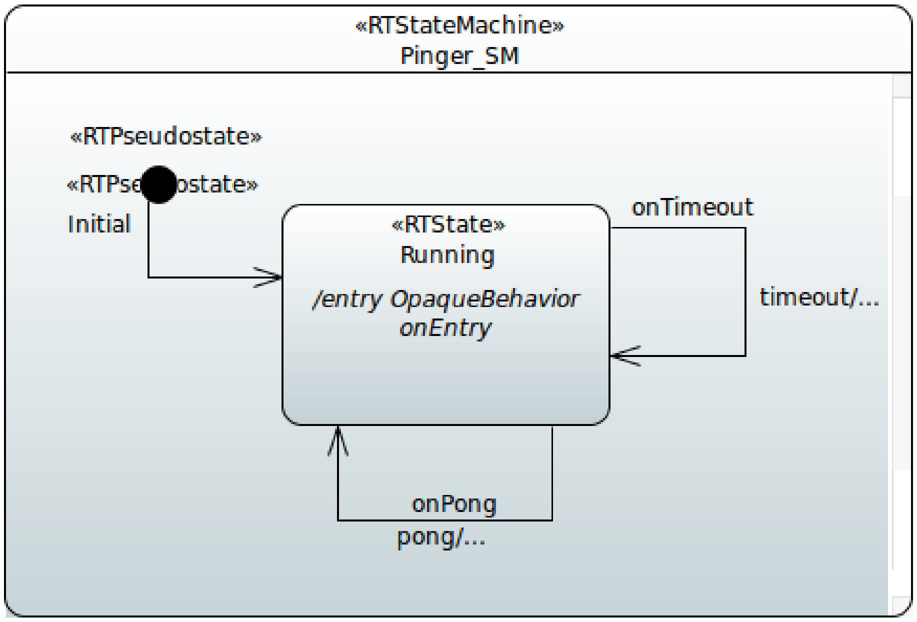


Figure 5.2: State-machine diagram of Pinger Capsule

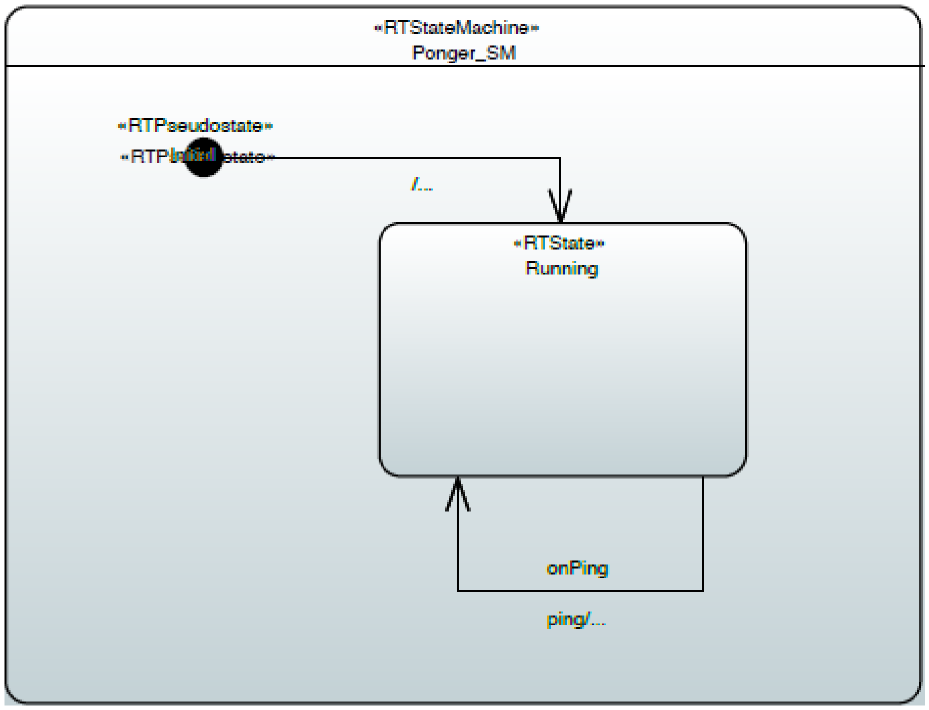
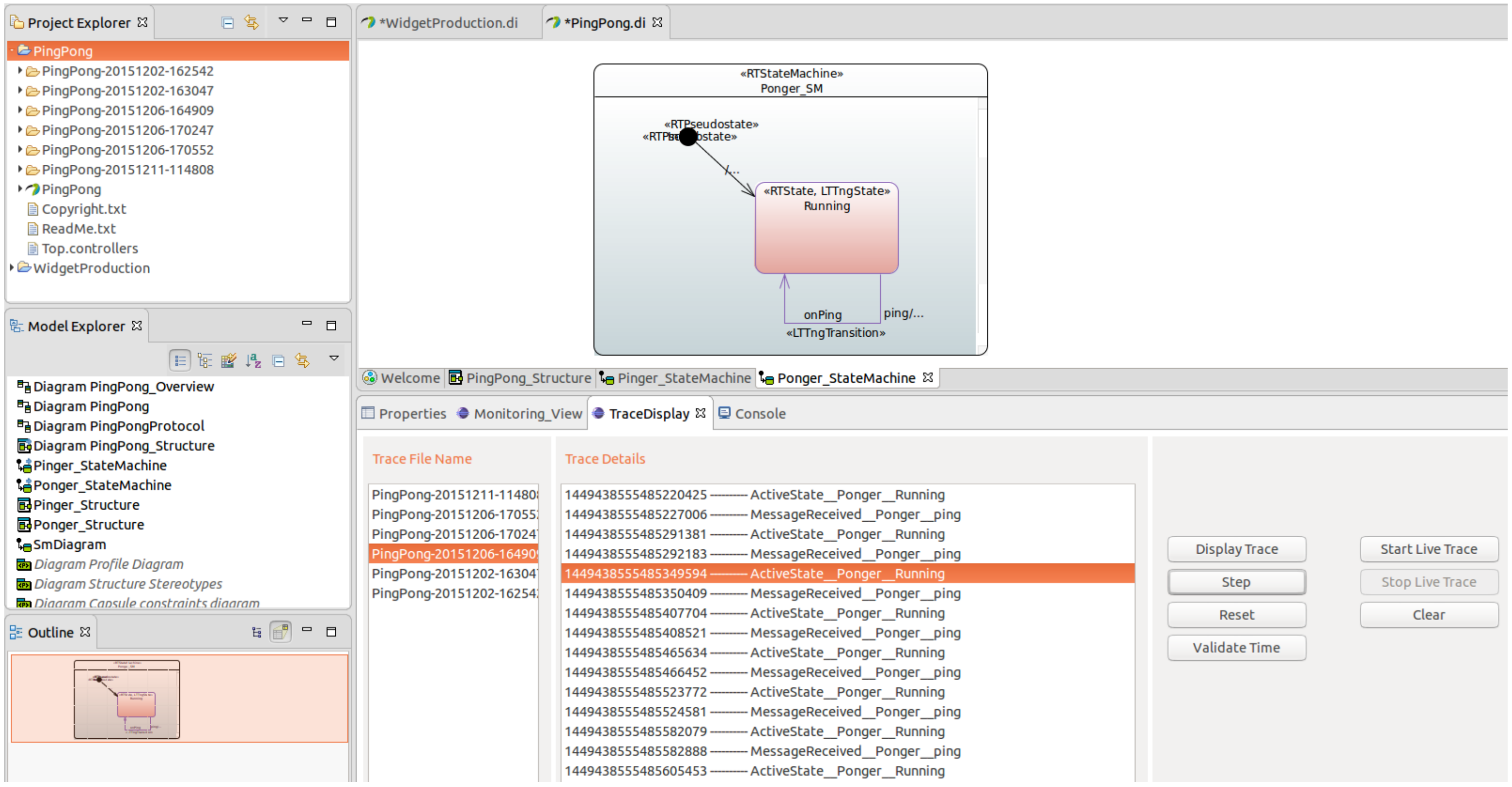


Figure 5.3: State-machine diagram of Ponger Capsule

can be triggered through the receipt of a timeout signal from a stop-timer intended to cease the signal interactions (see Fig. 5.2).

On the other end of the signal communication, we have the Ponger capsule. The behaviour of Ponger is defined in a state-machine which also has a state named “Running”. Ponger expects a signal named “onPing” from the Pinger capsule. Once it receives it when it is in the “Running” state, the “onPing” self-transition gets triggered and Ponger sends out the “onPong” signal to the Pinger capsule. Fig. 5.3 shows the state-machine behaviour of the Ponger capsule.

As part of the proof of concept, we have monitored the states and transitions of the Ponger state-machine in the PinPong model. Fig. 5.4 shows the display of traces in the plugin for the PingPong model with the associated state element and corresponding trace event at a particular point being highlighted. Clicking on the “Step” button



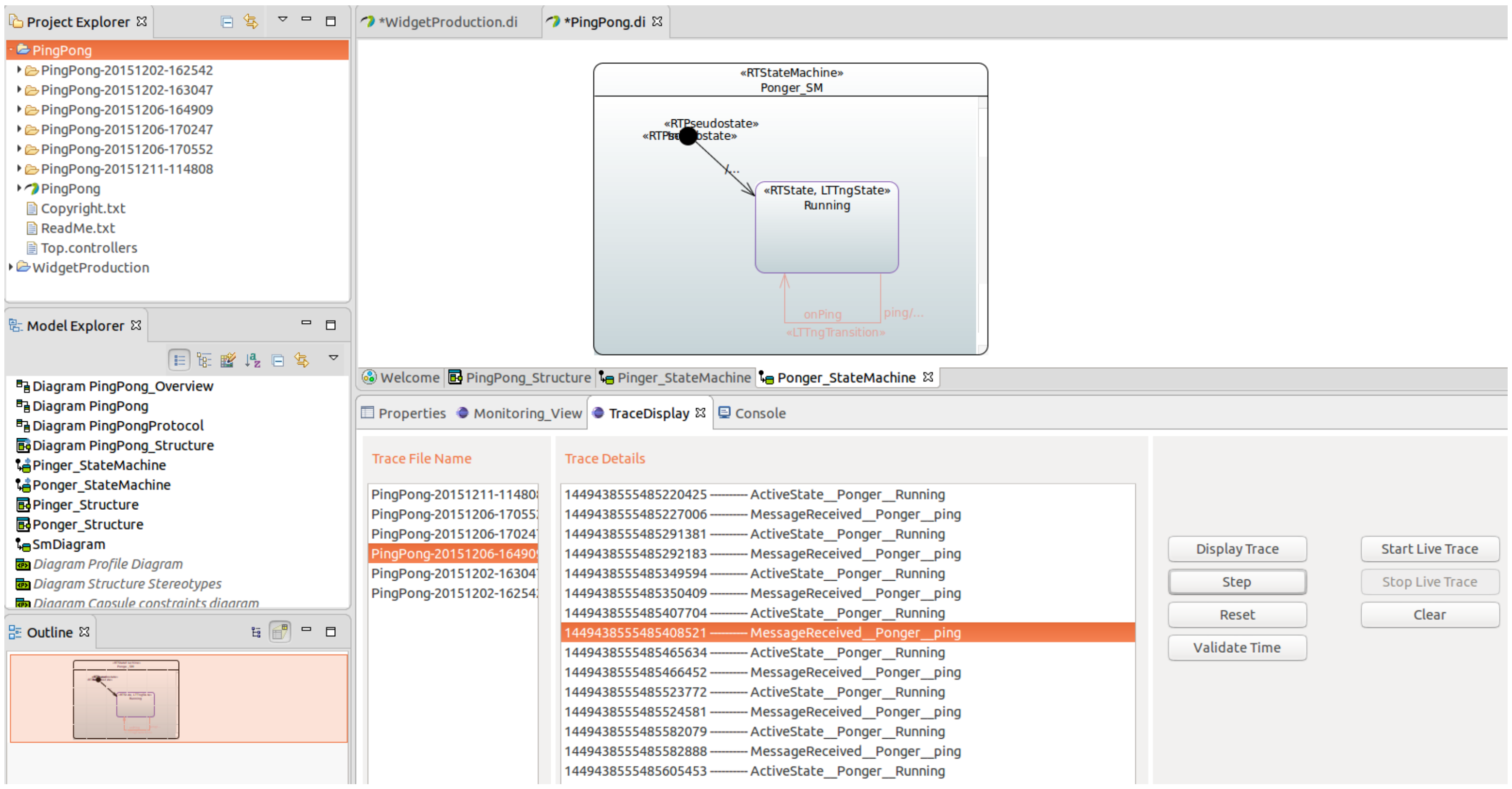
Trace File Name

```
PingPong-20151211-11480
PingPong-20151206-17055
PingPong-20151206-17024
PingPong-20151206-16490
PingPong-20151202-16304
PingPong-20151202-16254
```

Trace Details

```
1449438555485220425 -------- ActiveState__Ponger__Running
1449438555485227006 -------- MessageReceived__Ponger__ping
1449438555485291381 -------- ActiveState__Ponger__Running
1449438555485292183 -------- MessageReceived__Ponger__ping
1449438555485349594 -------- ActiveState__Ponger__Running
1449438555485350409 -------- MessageReceived__Ponger__ping
1449438555485407704 -------- ActiveState__Ponger__Running
1449438555485408521 -------- MessageReceived__Ponger__ping
1449438555485465634 -------- ActiveState__Ponger__Running
1449438555485466452 -------- MessageReceived__Ponger__ping
1449438555485523772 -------- ActiveState__Ponger__Running
1449438555485524581 -------- MessageReceived__Ponger__ping
1449438555485582079 -------- ActiveState__Ponger__Running
1449438555485582888 -------- MessageReceived__Ponger__ping
1449438555485605453 -------- ActiveState__Ponger__Running
```

Figure 5.4: Trace display of the PingPong model



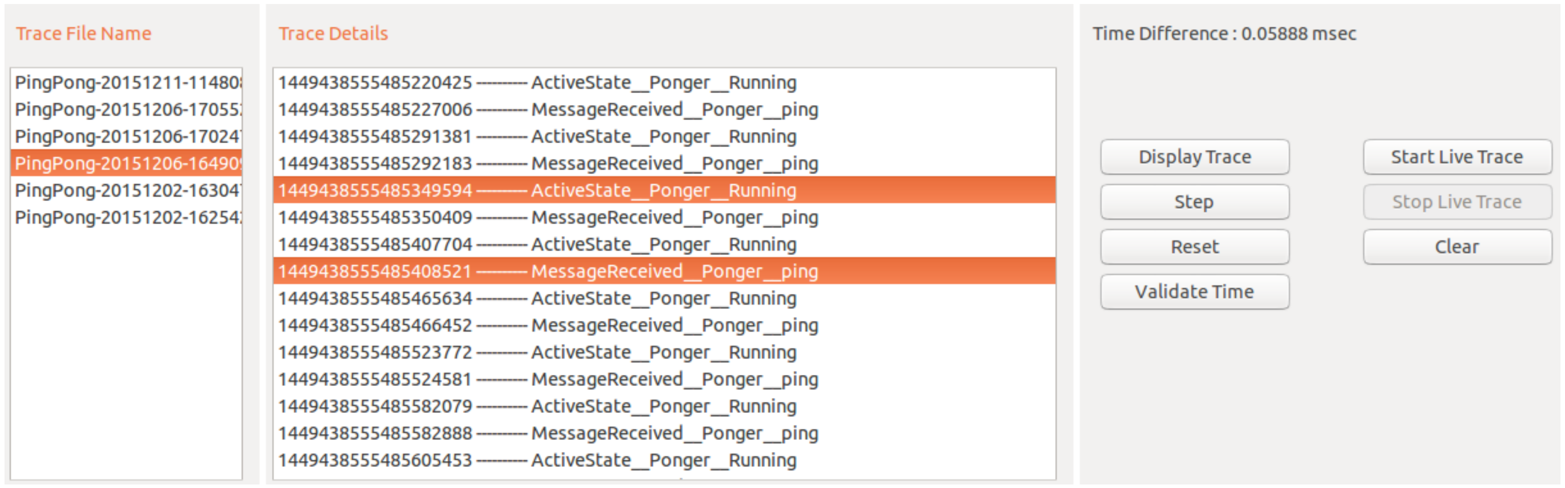
Trace File Name

```
PingPong-20151211-11480
PingPong-20151206-17055
PingPong-20151206-17024
PingPong-20151206-16490
PingPong-20151202-16304
PingPong-20151202-16254
```

Trace Details

```
1449438555485220425 -------- ActiveState__Ponger__Running
1449438555485227006 -------- MessageReceived__Ponger__ping
1449438555485291381 -------- ActiveState__Ponger__Running
1449438555485292183 -------- MessageReceived__Ponger__ping
1449438555485349594 -------- ActiveState__Ponger__Running
1449438555485350409 -------- MessageReceived__Ponger__ping
1449438555485407704 -------- ActiveState__Ponger__Running
1449438555485408521 -------- MessageReceived__Ponger__ping
1449438555485465634 -------- ActiveState__Ponger__Running
1449438555485466452 -------- MessageReceived__Ponger__ping
1449438555485523772 -------- ActiveState__Ponger__Running
1449438555485524581 -------- MessageReceived__Ponger__ping
1449438555485582079 -------- ActiveState__Ponger__Running
1449438555485582888 -------- MessageReceived__Ponger__ping
1449438555485605453 -------- ActiveState__Ponger__Running
```

Figure 5.5: Trace display by highlighting a transition of the PingPong model



Time Difference : 0.05888 msec

```
PingPong-20151211-11480
PingPong-20151206-17055
PingPong-20151206-17024
PingPong-20151206-16490
PingPong-20151202-16304
PingPong-20151202-16254
```

```
1449438555485220425 --------- ActiveState__Ponger__Running
1449438555485227006 --------- MessageReceived__Ponger__ping
1449438555485291381 --------- ActiveState__Ponger__Running
1449438555485292183 --------- MessageReceived__Ponger__ping
1449438555485349594 --------- ActiveState__Ponger__Running
1449438555485350409 --------- MessageReceived__Ponger__ping
1449438555485407704 --------- ActiveState__Ponger__Running
1449438555485408521 --------- MessageReceived__Ponger__ping
1449438555485465634 --------- ActiveState__Ponger__Running
1449438555485466452 --------- MessageReceived__Ponger__ping
1449438555485523772 --------- ActiveState__Ponger__Running
1449438555485524581 --------- MessageReceived__Ponger__ping
1449438555485582079 --------- ActiveState__Ponger__Running
1449438555485582888 --------- MessageReceived__Ponger__ping
1449438555485605453 --------- ActiveState__Ponger__Running
```

Figure 5.6: Time difference between two traced events in the PingPong model

allows us to see the next trace event along with the highlighted element being updated (see Fig. 5.5). Fig. 5.6 shows the time difference after selecting two events from the middle column of the textual display.

5.2 Widget Production

The Widget Production model is the second model we designed to test our implementation features. The Top capsule of this model contains four other capsules: ControlSoftware, ProductionLine, Workstation and Robot. Fig. 5.7 shows the structure diagram of the Top capsule of this model.

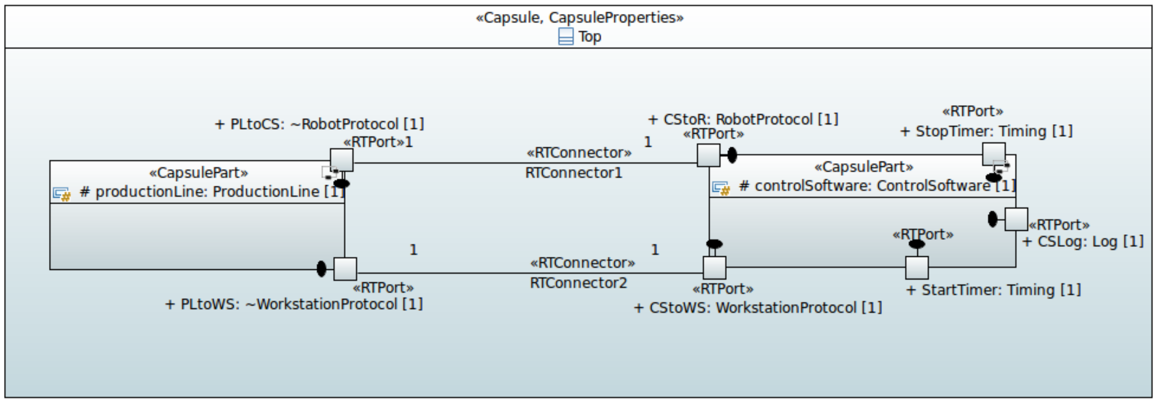

«Capsule, CapsuleProperties»
Top
+ PLtoCS: ~RobotProtocol [1]
«RTPort»1
«CapsulePart»
# productionLine: ProductionLine [1]
«RTConnector»
RTConnector1
+ CStoR: RobotProtocol [1]
«RTPort»
1
«RTPort»
+ StopTimer: Timing [1]
«CapsulePart»
# controlSoftware: ControlSoftware [1]
«RTPort»
+ CSLog: Log [1]
«RTPort»
1
«RTConnector»
RTConnector2
1
«RTPort»
+ PLtoWS: ~WorkstationProtocol [1]
«RTPort»
+ CStoWS: WorkstationProtocol [1]
+ StartTimer: Timing [1]

Figure 5.7: Structure diagram of the Widget production model

The ControlSoftware capsule is the heart of the widget production system. It coordinates the activities on the production line by directing when to produce a widget and when to deliver a widget by sending signals through protocols. The ProductionLine capsule is a container capsule which encapsulates the structure and behaviour of the Workstation and Robot capsules. The Workstation capsule controls the actual manufacturing of widgets (see Fig. 5.9) and the Robot capsule controls the delivery of robots (see Fig. 5.10).

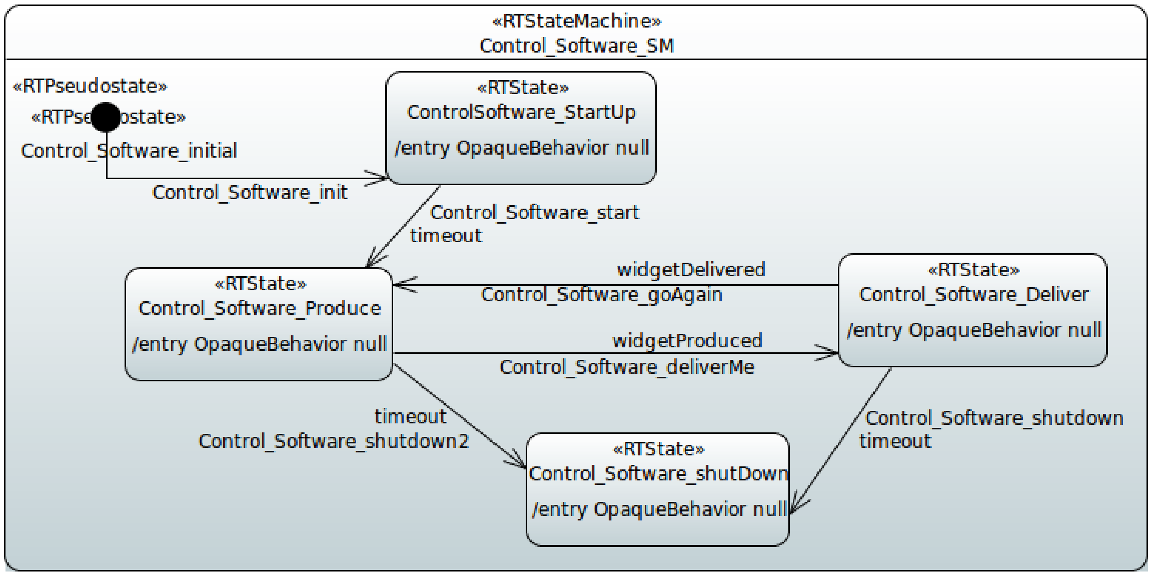


Figure 5.8: Statemachine diagram of the ControlSoftware capsule

Fig. 5.8 shows the state-machine diagram of the ControlSoftware capsule having four states. Initially it waits in the "ControlSoftware_StartUp" state to initialize other elements and startup the system. At this point, the "Workstation" and the "Robot"

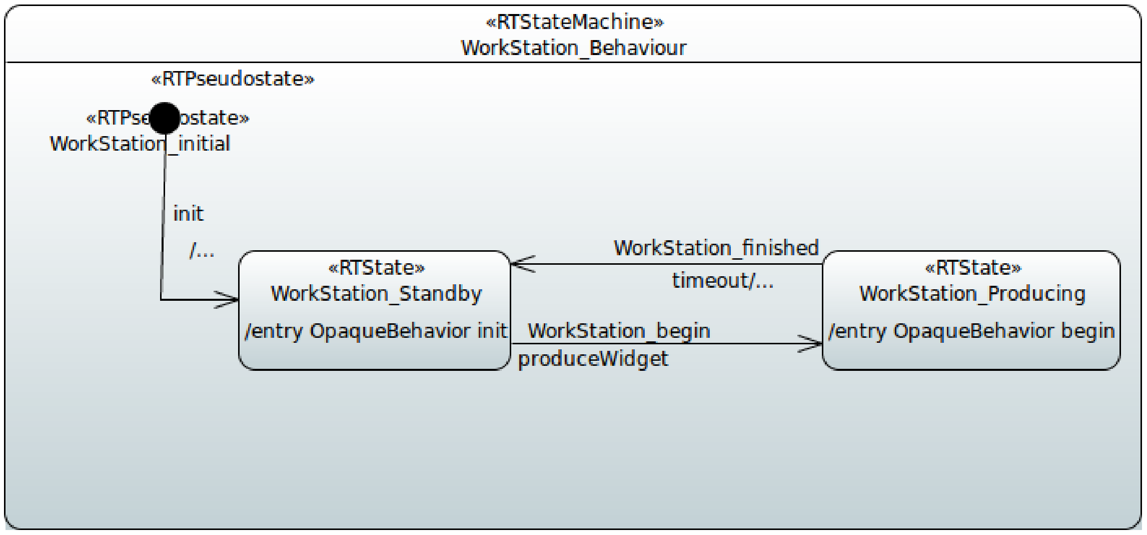


Figure 5.9: Statemachine diagram of the Workstation capsule

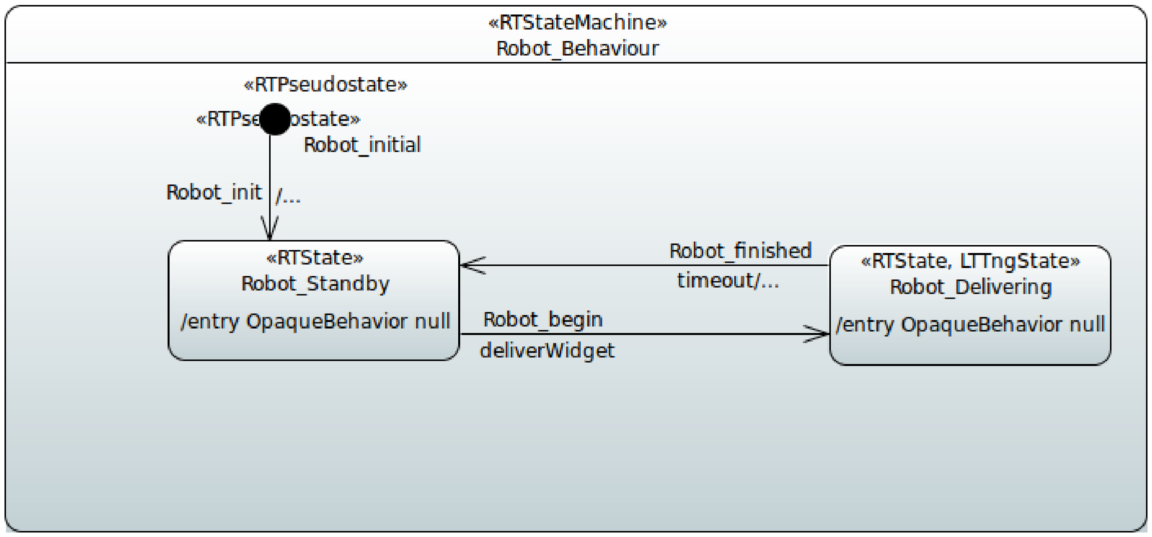
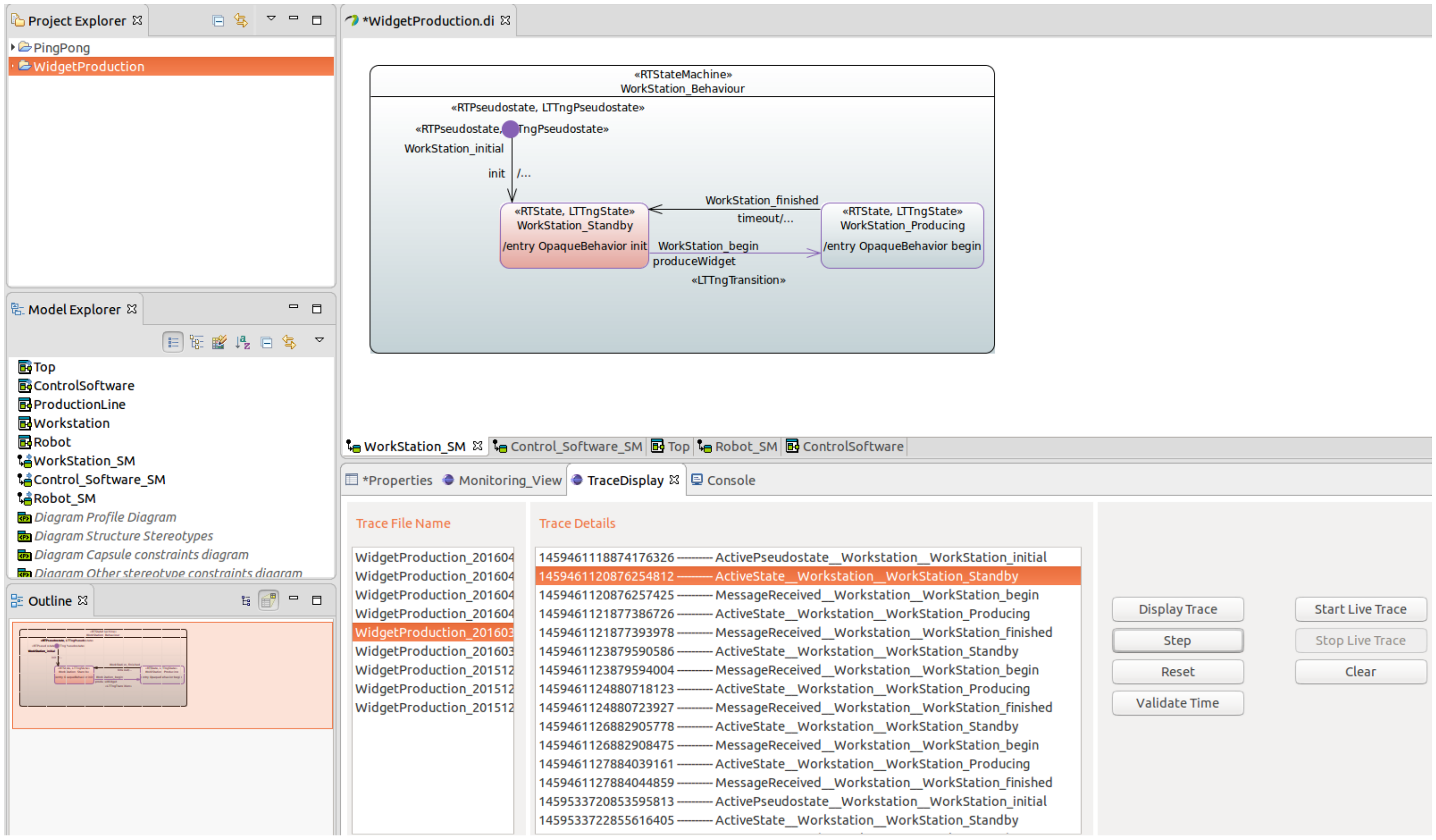


Figure 5.10: Statemachine diagram of the Robot capsule

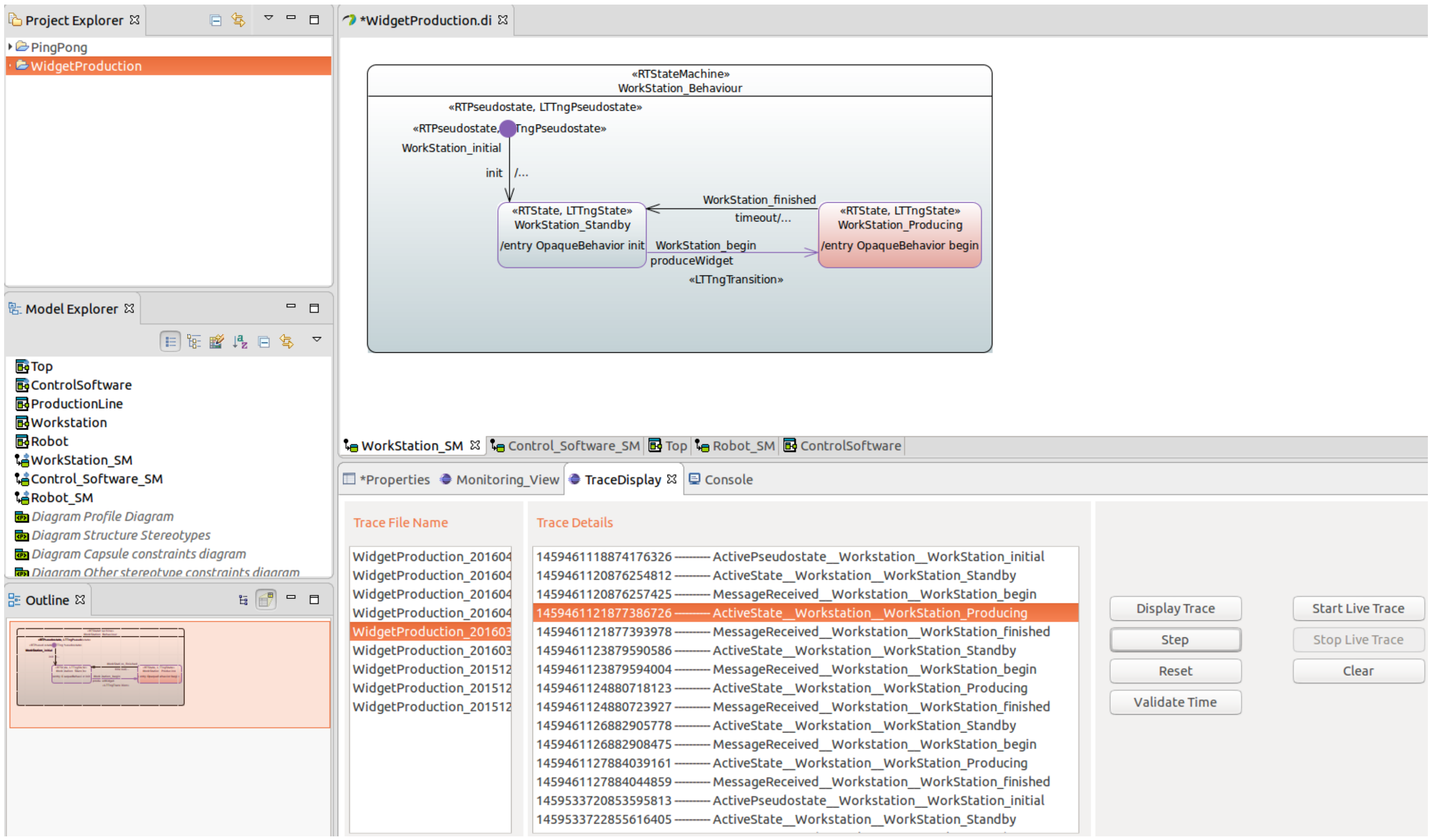
state-machines wait in their respective Standby states for a signal from the Control-Software state-machine for proceeding to operating mode. Once the startup timer of the ControlSoftware capsule fires, it enters the “Control_Software_Produce” state and sends a signal to the Workstation to produce widgets. After producing the widget, the Workstation sends a signal back to the ControlSoftware where the “widgetProduced” transition gets triggered for taking it to the “Control_Software_Deliver” state. Then the ControlSoftware sends a signal to the Robot state-machine to deliver the widget and then it waits for a signal from the Robot state-machine. Once it receives the signal, the “widgetDelivered” transition gets triggered and the ControlSoftware enters the “Control_Software_Produce” state again. The whole system ends after the ControlSoftware receives a shut down signal which gets triggered after a predefined time interval.

The trace display of the Workstation state-machine is demonstrated in Fig. 5.11 and Fig. 5.12. Fig. 5.13 and Fig. 5.14 show two cases demonstrating the conformance



Trace File Name	Trace Details
WidgetProduction_201604	1459461118874176326 ——— ActivePseudostate__Workstation__WorkStation_initial
WidgetProduction_201604	1459461120876254812 ——— ActiveState__Workstation__WorkStation_Standby
WidgetProduction_201604	1459461120876257425 ——— MessageReceived__Workstation__WorkStation_begin
WidgetProduction_201604	1459461121877386726 ——— ActiveState__Workstation__WorkStation_Producing
WidgetProduction_201603	1459461121877393978 ——— MessageReceived__Workstation__WorkStation_finished
WidgetProduction_201603	1459461123879590586 ——— ActiveState__Workstation__WorkStation_Standby
WidgetProduction_201512	1459461123879594004 ——— MessageReceived__Workstation__WorkStation_begin
WidgetProduction_201512	1459461124880718123 ——— ActiveState__Workstation__WorkStation_Producing
WidgetProduction_201512	1459461124880723927 ——— MessageReceived__Workstation__WorkStation_finished
	1459461126882905778 ——— ActiveState__Workstation__WorkStation_Standby
	1459461126882908475 ——— MessageReceived__Workstation__WorkStation_begin
	1459461127884039161 ——— ActiveState__Workstation__WorkStation_Producing
	1459461127884044859 ——— MessageReceived__Workstation__WorkStation_finished
	1459533720853595813 ——— ActivePseudostate__Workstation__WorkStation_initial
	1459533722855616405 ——— ActiveState__Workstation__WorkStation_Standby

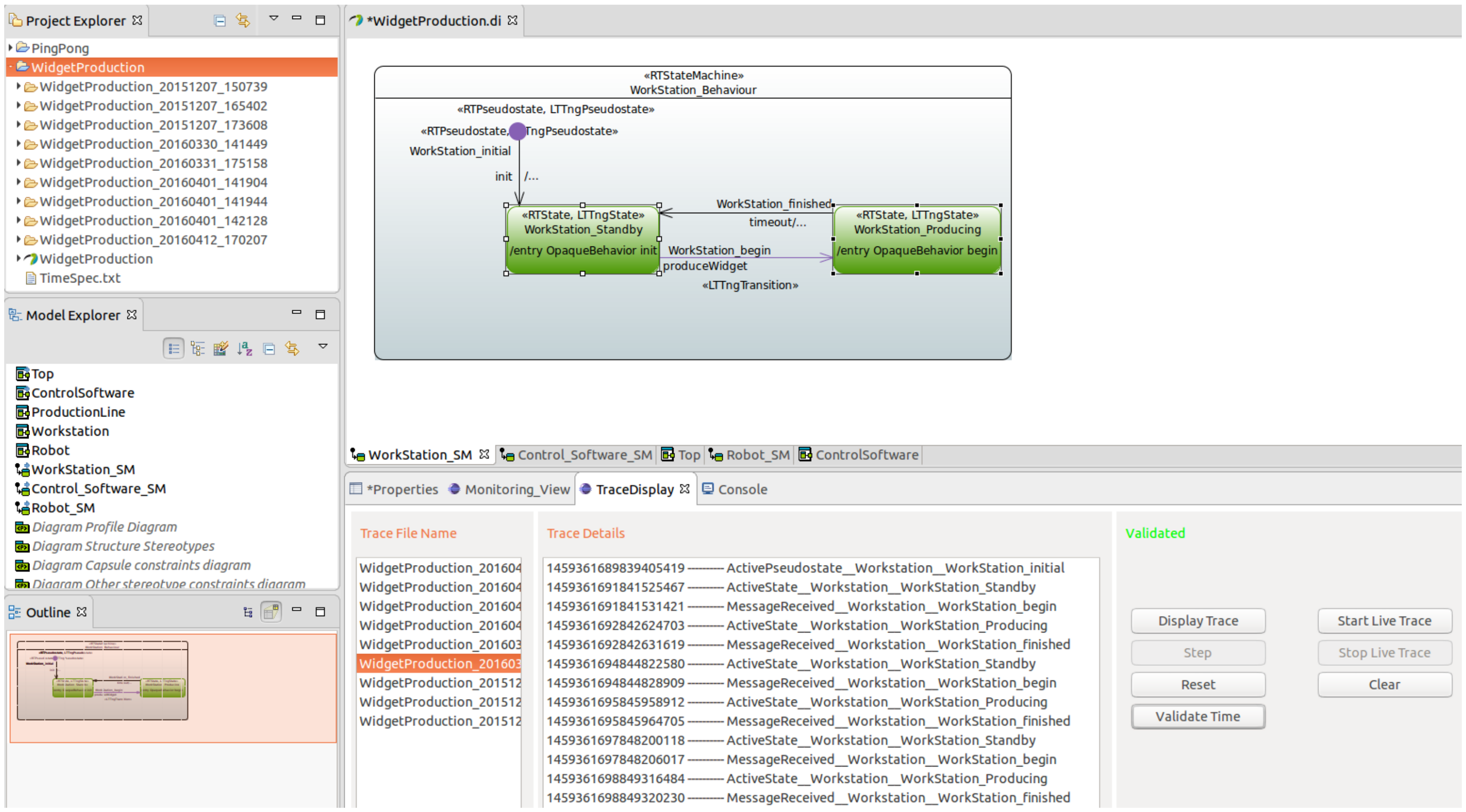
Figure 5.11: Trace display in the Widget Production model



```
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201603
WidgetProduction_201603
WidgetProduction_201512
WidgetProduction_201512
WidgetProduction_201512
```

```
1459461118874176326 -------- ActivePseudostate__Workstation__WorkStation_initial
1459461120876254812 -------- ActiveState__Workstation__WorkStation_Standby
1459461120876257425 -------- MessageReceived__Workstation__WorkStation_begin
1459461121877386726 -------- ActiveState__Workstation__WorkStation_Producing
1459461121877393978 -------- MessageReceived__Workstation__WorkStation_finished
1459461123879590586 -------- ActiveState__Workstation__WorkStation_Standby
1459461123879594004 -------- MessageReceived__Workstation__WorkStation_begin
1459461124880718123 -------- ActiveState__Workstation__WorkStation_Producing
1459461124880723927 -------- MessageReceived__Workstation__WorkStation_finished
1459461126882905778 -------- ActiveState__Workstation__WorkStation_Standby
1459461126882908475 -------- MessageReceived__Workstation__WorkStation_begin
1459461127884039161 -------- ActiveState__Workstation__WorkStation_Producing
1459461127884044859 -------- MessageReceived__Workstation__WorkStation_finished
1459533720853595813 -------- ActivePseudostate__Workstation__WorkStation_initial
1459533722855616405 -------- ActiveState__Workstation__WorkStation_Standby
```

Figure 5.12: Trace display in the Widget Production model by highlighting the Producing state



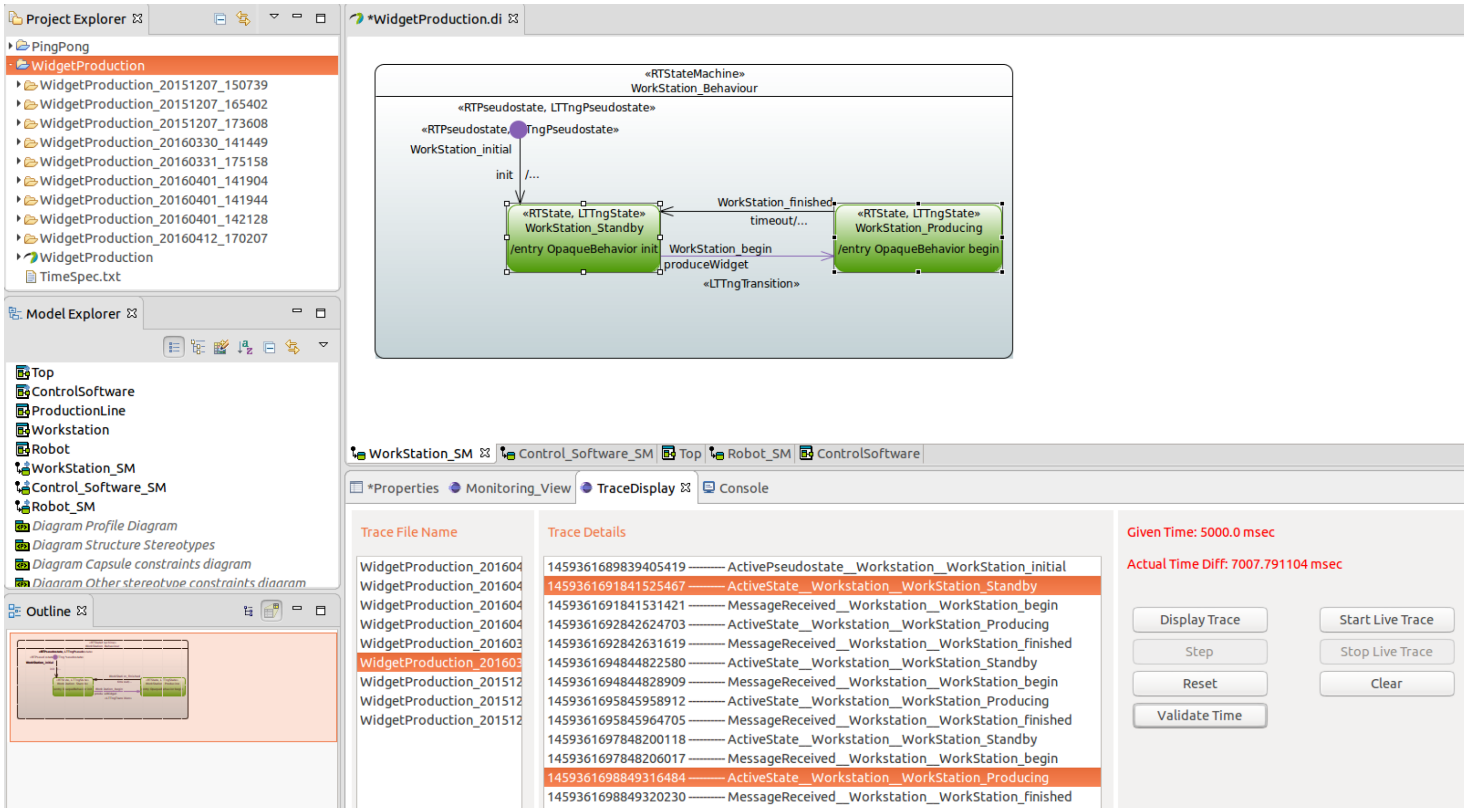
Trace File Name

```
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201603
WidgetProduction_201603
WidgetProduction_201512
WidgetProduction_201512
WidgetProduction_201512
```

Trace Details

```
1459361689839405419 -------- ActivePseudostate__Workstation__WorkStation_initial
1459361691841525467 -------- ActiveState__Workstation__WorkStation_Standby
1459361691841531421 -------- MessageReceived__Workstation__WorkStation_begin
1459361692842624703 -------- ActiveState__Workstation__WorkStation_Producing
1459361692842631619 -------- MessageReceived__Workstation__WorkStation_finished
1459361694844822580 -------- ActiveState__Workstation__WorkStation_Standby
1459361694844828909 -------- MessageReceived__Workstation__WorkStation_begin
1459361695845958912 -------- ActiveState__Workstation__WorkStation_Producing
1459361695845964705 -------- MessageReceived__Workstation__WorkStation_finished
1459361697848200118 -------- ActiveState__Workstation__WorkStation_Standby
1459361697848206017 -------- MessageReceived__Workstation__WorkStation_begin
1459361698849316484 -------- ActiveState__Workstation__WorkStation_Producing
1459361698849320230 -------- MessageReceived__Workstation__WorkStation_finished
```

Figure 5.13: Time constraint validation in the Widget Production model



```
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201604
WidgetProduction_201603
WidgetProduction_201603
WidgetProduction_201512
WidgetProduction_201512
WidgetProduction_201512
```

```
1459361689839405419 -------- ActivePseudostate__Workstation__WorkStation_initial
1459361691841525467 -------- ActiveState__Workstation__WorkStation_Standby
1459361691841531421 -------- MessageReceived__Workstation__WorkStation_begin
1459361692842624703 -------- ActiveState__Workstation__WorkStation_Producing
1459361692842631619 -------- MessageReceived__Workstation__WorkStation_finished
1459361694844822580 -------- ActiveState__Workstation__WorkStation_Standby
1459361694844828909 -------- MessageReceived__Workstation__WorkStation_begin
1459361695845958912 -------- ActiveState__Workstation__WorkStation_Producing
1459361695845964705 -------- MessageReceived__Workstation__WorkStation_finished
1459361697848200118 -------- ActiveState__Workstation__WorkStation_Standby
1459361697848206017 -------- MessageReceived__Workstation__WorkStation_begin
1459361698849316484 -------- ActiveState__Workstation__WorkStation_Producing
1459361698849320230 -------- MessageReceived__Workstation__WorkStation_finished
```

```
Given Time: 5000.0 msec
Actual Time Diff: 7007.791104 msec
```

Figure 5.14: Violation of time constraint in the Widget Production model

and violation of a timing requirement specification in our plugin.

5.3 Rover

We have created the Rover model in Papyrus-RT to test live and remote monitoring of real-time embedded systems. The rover we have is a small vehicle with two motors that can move in different directions. It is built using a Raspberry-Pi platform running a Linux OS. Fig. 5.15 shows the physical Rover we have used for testing the live tracing concept.

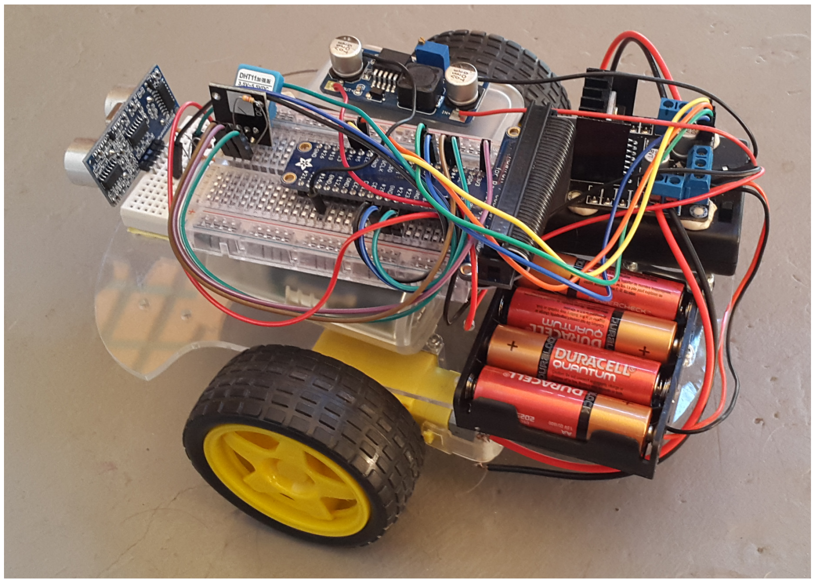


Figure 5.15: The Rover

The Rover behavior is represented using UML-RT state-machine in Papyrus-RT. Following the behavior specification, Rover moves forward until detecting an obstacle. Once it detects an obstacle, it tries to avoid the collision by making a turn of 90 degrees and then it starts moving in forward direction again. The code is generated using the Papyrus-RT tool and Raspberry-Pi is used for executing the code.

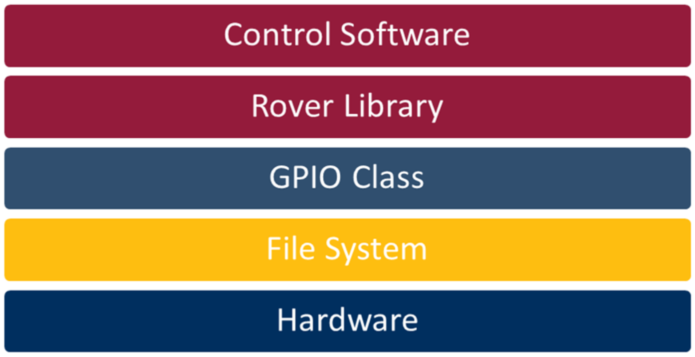
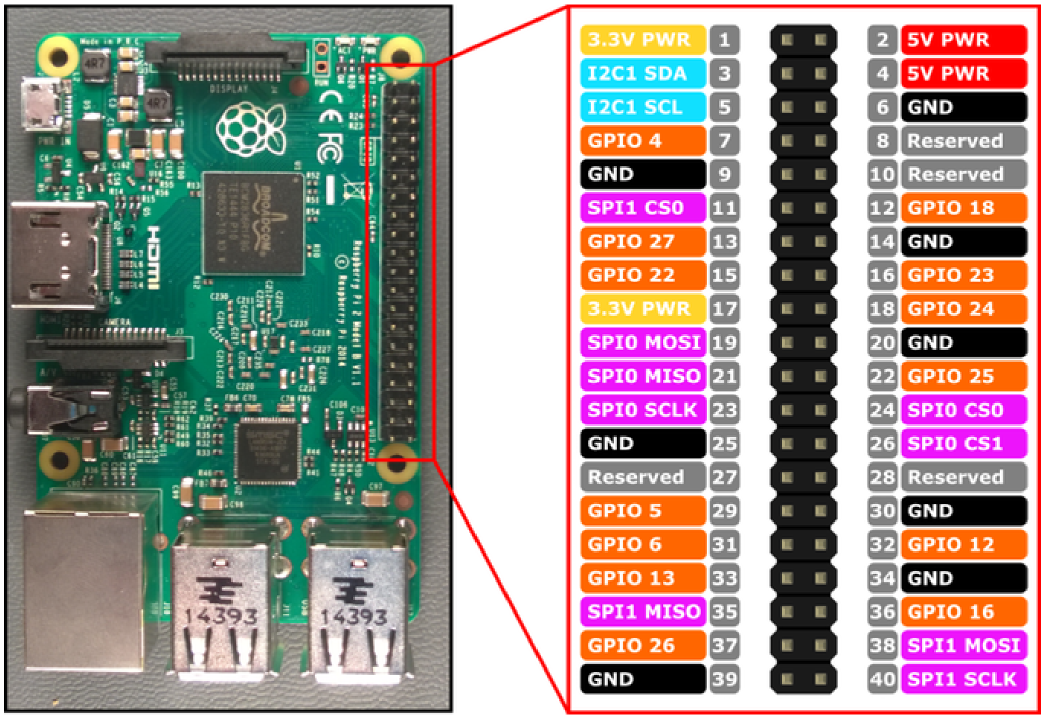


Figure 5.16: Rover Architecture

Now, let us go through the architecture of the Rover (see Fig. 5.16). The Rover architecture consists of 5 different layers. In the bottom-most, we have the hardware layer. A Raspberry Pi having 26 GPIO pins is used for representing this layer. 17 of these pins are used for connecting external devices such as sensors and actuators (see



3.3V PWR	1	2	5V PWR
I2C1 SDA	3	4	5V PWR
I2C1 SCL	5	6	GND
GPIO 4	7	8	Reserved
GND	9	10	Reserved
SPI1 CS0	11	12	GPIO 18
GPIO 27	13	14	GND
GPIO 22	15	16	GPIO 23
3.3V PWR	17	18	GPIO 24
SPI0 MOSI	19	20	GND
SPI0 MISO	21	22	GPIO 25
SPI0 SCLK	23	24	SPI0 CS0
GND	25	26	SPI0 CS1
Reserved	27	28	Reserved
GPIO 5	29	30	GND
GPIO 6	31	32	GPIO 12
GPIO 13	33	34	GND
SPI1 MISO	35	36	GPIO 16
GPIO 26	37	38	SPI1 MOSI
GND	39	40	SPI1 SCLK

Figure 5.17: Raspberry-pi GPIO Pin [19]

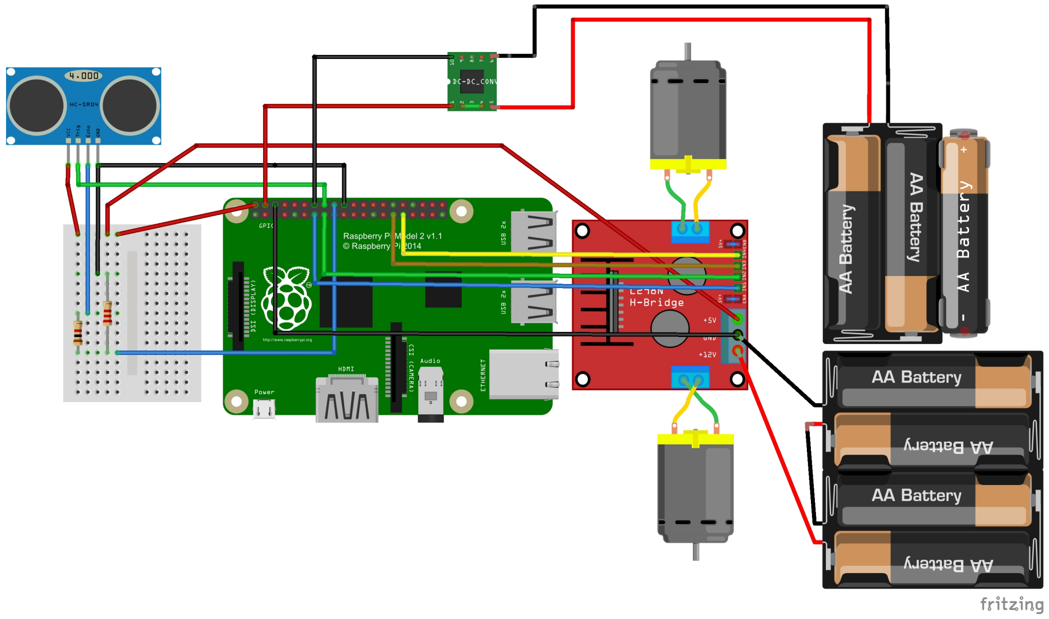


Figure 5.18: Rover Wiring Diagram [19]

Fig. 5.17). On top of the hardware layer, the Rover architecture has a file system which is powered by a real-time version of Linux. GPIO pins in the Raspberry Pi are used for providing users read/write access to the file system. These GPIO pins are controlled using a C++ wrapper class which represents the GPIO Class layer in Fig. 5.16. The wrapper class consists of the ”set” and ”get” methods which are used for setting and retrieving the pin values respectively. On top of the wrapper class, we have the Rover library layer. This library comprises different components of the physical Rover in the form of a number of UML-RT capsules. Finally, the Control Software layer, the topmost layer of the Rover architecture, represents the business logic of the application.

The assembly of the Rover platform is shown in Fig. 5.18. An important step of assembling the physical platform of the Rover is the proper selection of components. As we can see in Fig. 5.18, two step motors that are attached to wheels are embedded in the core component, the Raspberry Pi 3. A motor controller connected to the Raspberry Pi ensures the control of the Rover. Distance measurement with obstacles is done using an ultrasonic distance sensor. Different components are connected using a breadboard. Power is provided using two sets of batteries, whereas the receipt of 5 volts in the Raspberry Pi, which it needs to be powered on, is ensured with the use of a voltage regulator.

The composite structure diagram of the Top capsule is shown in Fig. 5.19. This capsule connects two other capsules: the ControlSoftware and the Rover capsule.

Fig. 5.20 shows the state-machine diagram of the ControlSoftware capsule. It is a key aspect of the model behavior as it represents the logic of the rover system. Instructions are designed to be sent from this capsule behavior to the Engine Controller to move forward until the distance between the rover and an obstacle is no more than one foot. Once the rover reaches such a point, it moves back, turns and goes back to a position where it moves forward again and checks the distance.

A capsule named "Detection" is used for communicating the distance sensor.

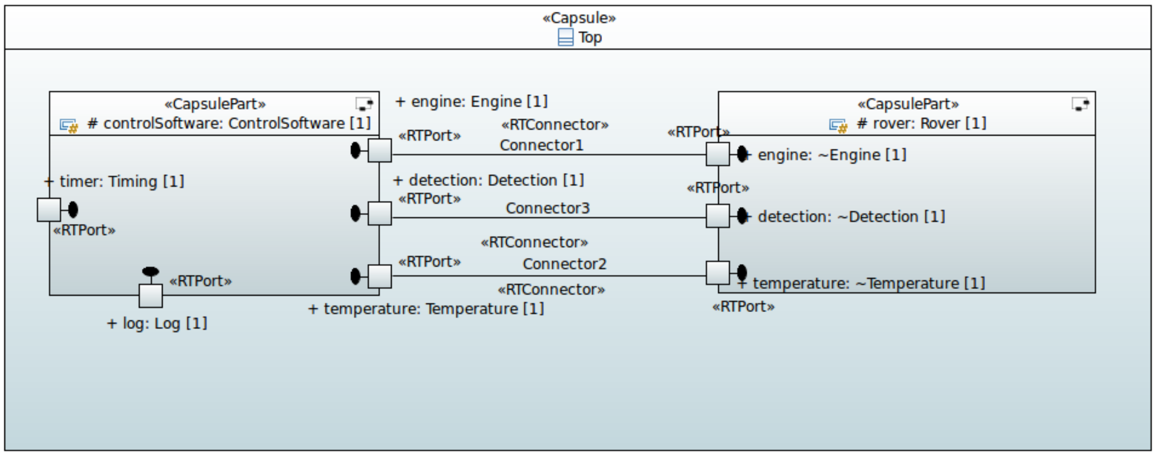


Figure 5.19: Structure diagram of Rover Top capsule

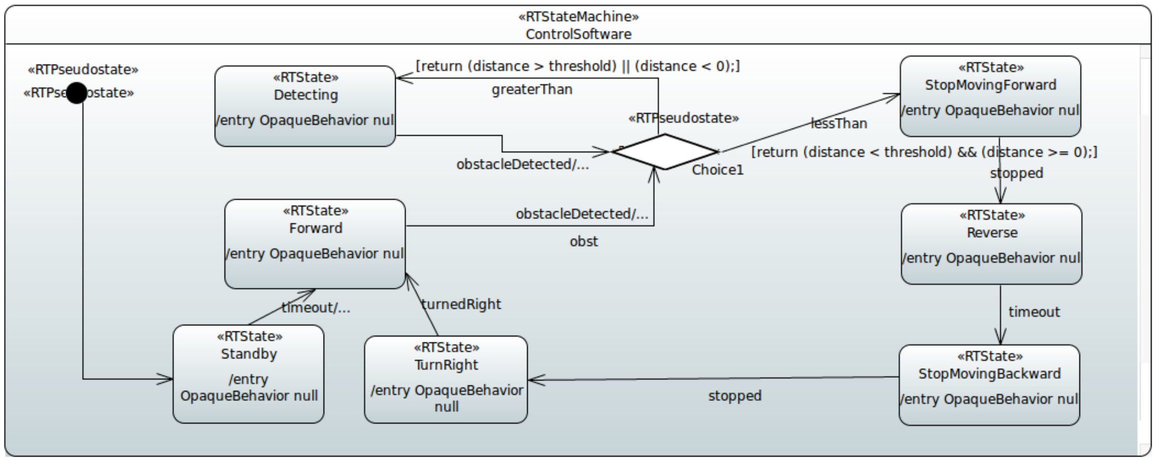

«RTStateMachine»
ControlSoftware
«RTPseudostate»
«RTPseudostate»
«RTState» Detecting /entry OpaqueBehavior null
[return (distance > threshold) || (distance < 0);]
greaterThan
«RTPseudostate»
Choice1
«RTState» StopMovingForward /entry OpaqueBehavior null
lessThan
[return (distance < threshold) && (distance >= 0);]
stopped
obstacleDetected/...
«RTState» Forward /entry OpaqueBehavior null
obstacleDetected/...
obst
«RTState» Reverse /entry OpaqueBehavior null
timeout/...
turnedRight
timeout
«RTState» Standby /entry OpaqueBehavior null
«RTState» TurnRight /entry OpaqueBehavior null
stopped
«RTState» StopMovingBackward /entry OpaqueBehavior null

Figure 5.20: Control Software state-machine diagram of the Rover model

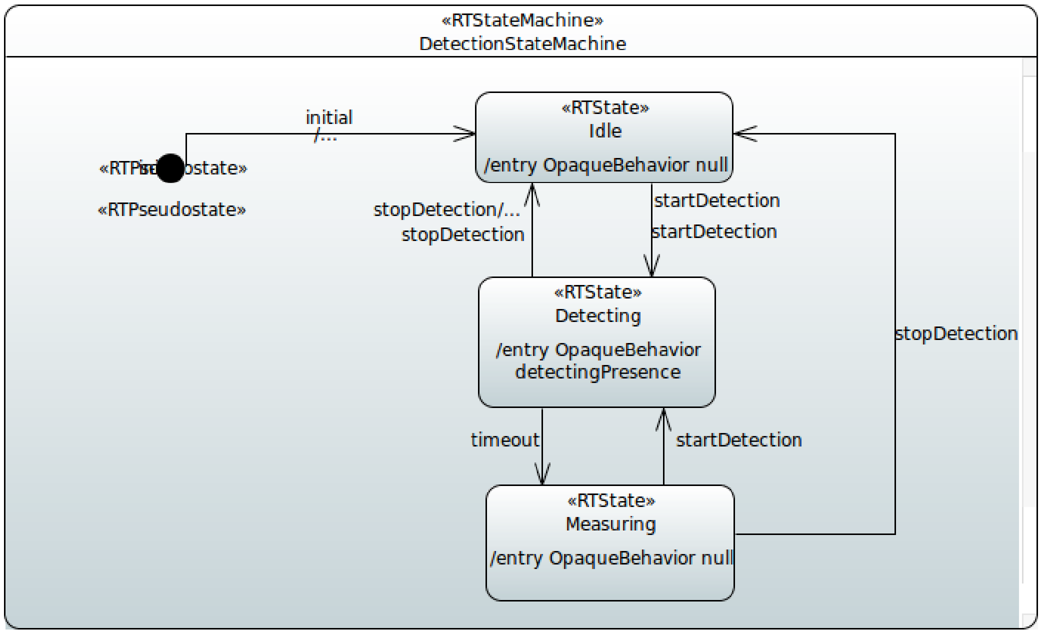


Figure 5.21: Detection state-machine diagram of the Rover model

Fig. 5.21 shows the behavior of the Detection capsule. The distance sensor sends out a pulse of ultrasonic sound and measuring the time it takes for the signal to return. It stops sending out the signals once the pulse comes back.

The Engine Controller capsule behavior is shown in Fig. 5.22. As the name implies, this capsule is responsible for controlling the engine. The execution of this capsule waits in the “Idle” state until an event is received which directs it to either forward or backward direction. Consequently, the execution moves to either “MovingForward” or “MovingBackward” state. Once one of these states becomes active, the system can come back to the “Idle” state only if it receives a stop signal. In the background, when the execution enters either “MovingForward” or “MovingBackward” state, the engine controller communicates with the file system through the rover library. This allows the control of the hardware which directs the actual movement of the physical component. Similarly, once the Engine Controller receives a signal directing it to either right or left, the execution goes to the “TurningRight” or “TurningLeft” state respectively. In either cases, the rover engine moves the corresponding motors which causes the actual turn. The transition of the rover engine back to the “Idle” state is done using the expiration event of a system timer.

We have used the Rover model for both offline and live monitoring.

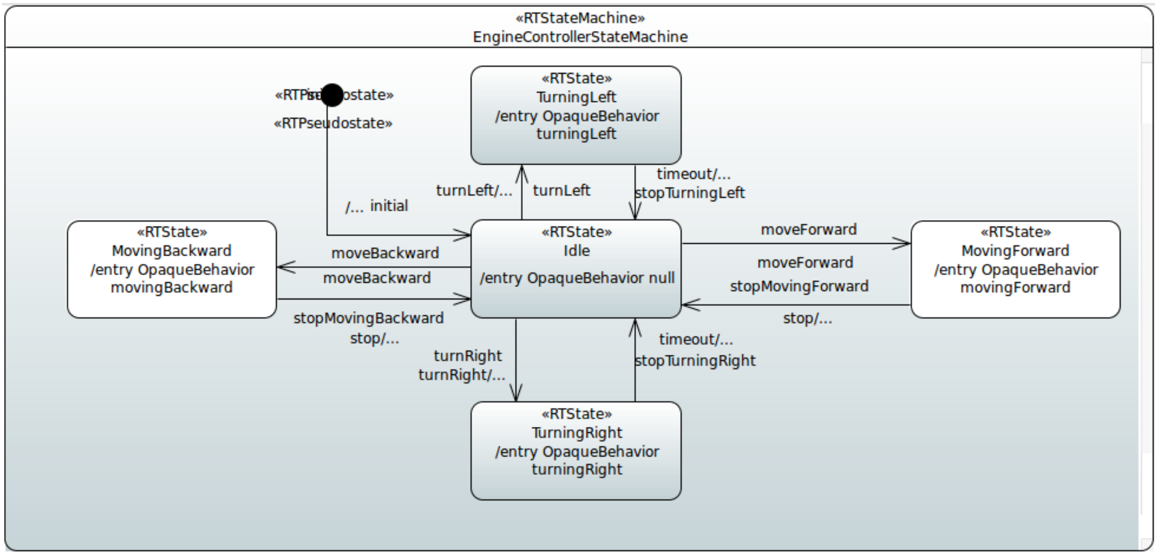
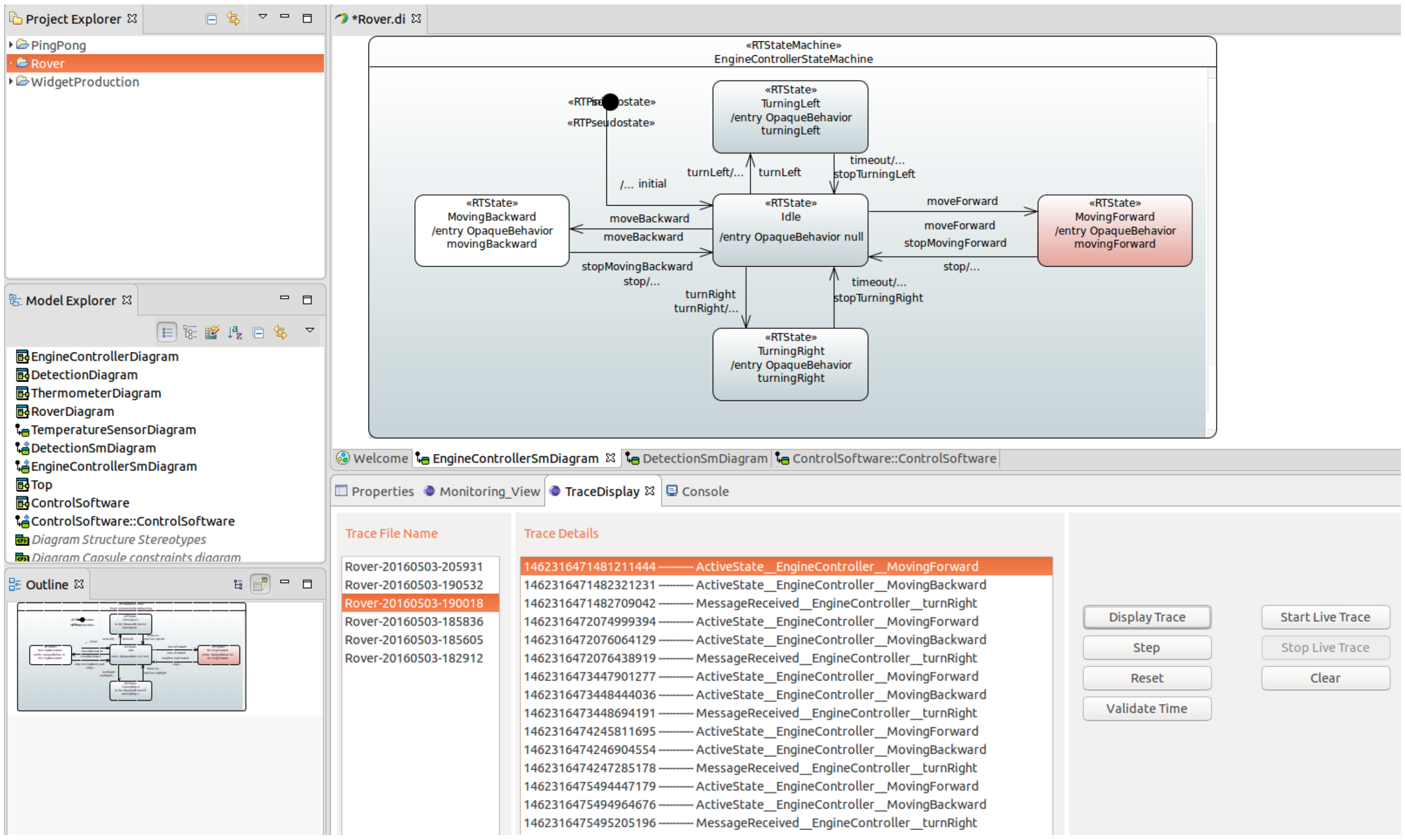

«RTStateMachine»
EngineControllerStateMachine
«RTPseudostate»
«RTPseudostate»
/... initial
«RTState» TurningLeft /entry OpaqueBehavior turningLeft
turnLeft/...
turnLeft
timeout/...
stopTurningLeft
«RTState» MovingBackward /entry OpaqueBehavior movingBackward
moveBackward
moveBackward
stopMovingBackward
stop/...
«RTState» Idle /entry OpaqueBehavior null
moveForward
moveForward
stopMovingForward
stop/...
«RTState» MovingForward /entry OpaqueBehavior movingForward
turnRight
turnRight/...
timeout/...
stopTurningRight
«RTState» TurningRight /entry OpaqueBehavior turningRight

Figure 5.22: State-machine diagram of Engine Controller


Project Explorer
PingPong
Rover
WidgetProduction
Model Explorer
EngineControllerDiagram
DetectionDiagram
ThermometerDiagram
RoverDiagram
TemperatureSensorDiagram
DetectionSmDiagram
EngineControllerSmDiagram
Top
ControlSoftware
ControlSoftware::ControlSoftware
Diagram Structure Stereotypes
Diagram Capsule constraints diagram
Outline
*Rover.di
«RTStateMachine» EngineControllerStateMachine
«RTPseudostate»
«RTPseudostate»
/... initial
«RTState» TurningLeft /entry OpaqueBehavior turningLeft
«RTState» MovingBackward /entry OpaqueBehavior movingBackward
«RTState» Idle /entry OpaqueBehavior null
«RTState» MovingForward /entry OpaqueBehavior movingForward
«RTState» TurningRight /entry OpaqueBehavior turningRight
turnLeft/...
turnLeft
timeout/... stopTurningLeft
moveBackward
moveBackward
stopMovingBackward stop/...
moveForward
moveForward stopMovingForward
stop/...
turnRight turnRight/...
timeout/... stopTurningRight
Welcome
EngineControllerSmDiagram
DetectionSmDiagram
ControlSoftware::ControlSoftware
Properties
Monitoring_View
TraceDisplay
Console
Display Trace
Step
Reset
Validate Time
Start Live Trace
Stop Live Trace
Clear

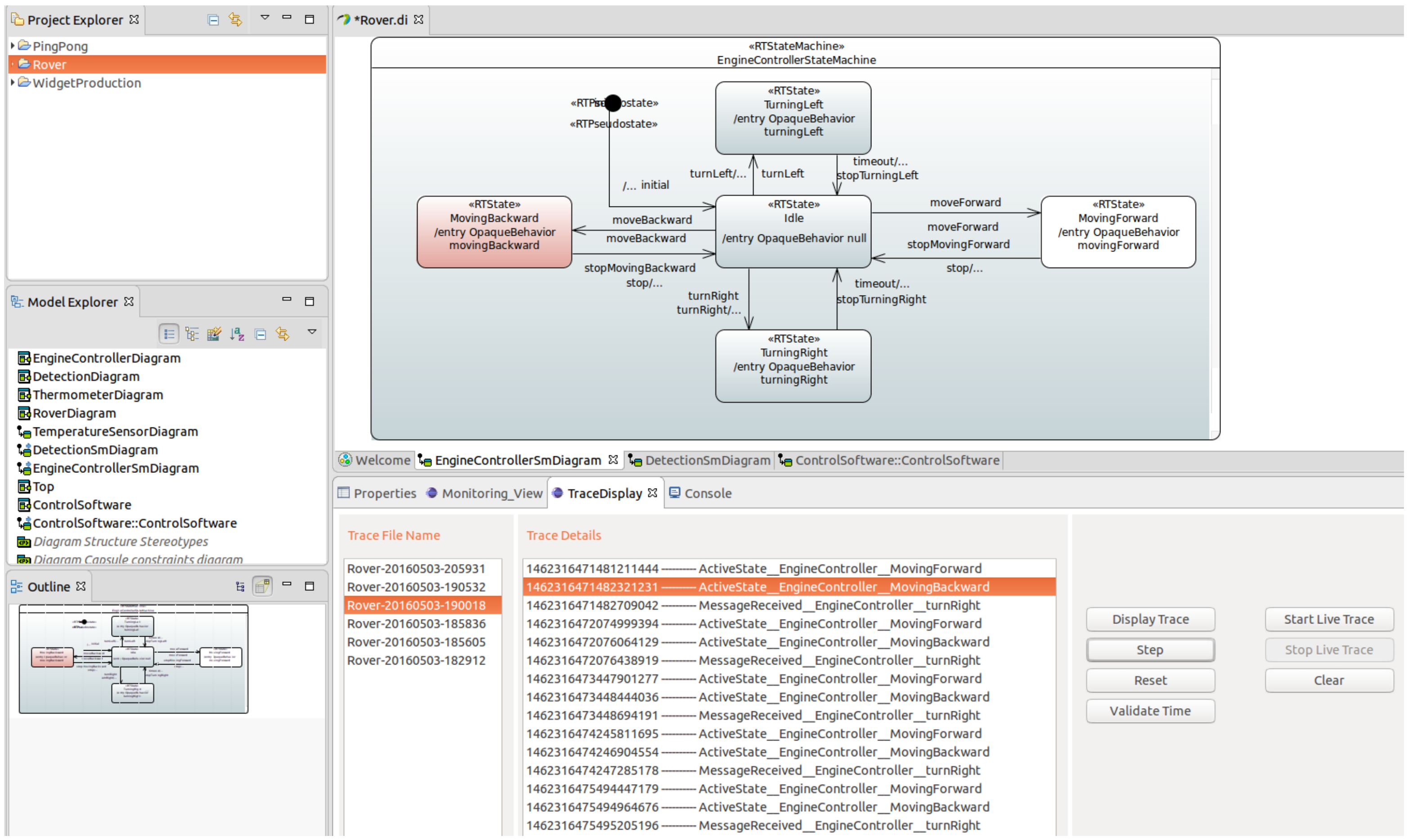
Trace File Name

```
Rover-20160503-205931
Rover-20160503-190532
Rover-20160503-190018
Rover-20160503-185836
Rover-20160503-185605
Rover-20160503-182912
```

Trace Details

```
1462316471481211444 ——— ActiveState__EngineController__MovingForward
1462316471482321231 ——— ActiveState__EngineController__MovingBackward
1462316471482709042 ——— MessageReceived__EngineController__turnRight
1462316472074999394 ——— ActiveState__EngineController__MovingForward
1462316472076064129 ——— ActiveState__EngineController__MovingBackward
1462316472076438919 ——— MessageReceived__EngineController__turnRight
1462316473447901277 ——— ActiveState__EngineController__MovingForward
1462316473448444036 ——— ActiveState__EngineController__MovingBackward
1462316473448694191 ——— MessageReceived__EngineController__turnRight
1462316474245811695 ——— ActiveState__EngineController__MovingForward
1462316474246904554 ——— ActiveState__EngineController__MovingBackward
1462316474247285178 ——— MessageReceived__EngineController__turnRight
1462316475494447179 ——— ActiveState__EngineController__MovingForward
1462316475494964676 ——— ActiveState__EngineController__MovingBackward
1462316475495205196 ——— MessageReceived__EngineController__turnRight
```

Figure 5.23: Trace display in the Rover model



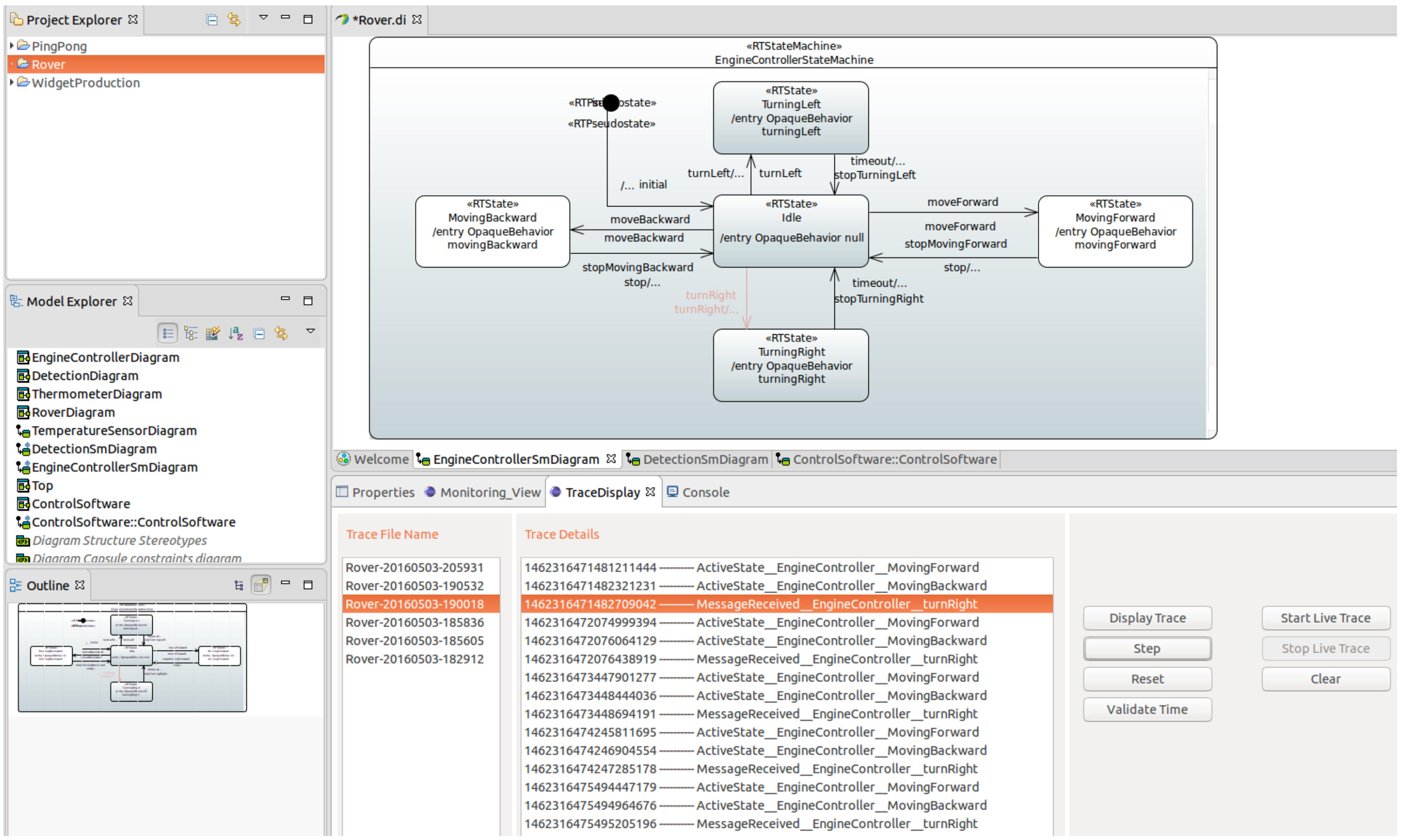
Trace File Name

Rover-20160503-205931
Rover-20160503-190532
Rover-20160503-190018
Rover-20160503-185836
Rover-20160503-185605
Rover-20160503-182912

Trace Details

```
1462316471481211444 ——— ActiveState__EngineController__MovingForward
1462316471482321231 ——— ActiveState__EngineController__MovingBackward
1462316471482709042 ——— MessageReceived__EngineController__turnRight
1462316472074999394 ——— ActiveState__EngineController__MovingForward
1462316472076064129 ——— ActiveState__EngineController__MovingBackward
1462316472076438919 ——— MessageReceived__EngineController__turnRight
1462316473447901277 ——— ActiveState__EngineController__MovingForward
1462316473448444036 ——— ActiveState__EngineController__MovingBackward
1462316473448694191 ——— MessageReceived__EngineController__turnRight
1462316474245811695 ——— ActiveState__EngineController__MovingForward
1462316474246904554 ——— ActiveState__EngineController__MovingBackward
1462316474247285178 ——— MessageReceived__EngineController__turnRight
1462316475494447179 ——— ActiveState__EngineController__MovingForward
1462316475494964676 ——— ActiveState__EngineController__MovingBackward
1462316475495205196 ——— MessageReceived__EngineController__turnRight
```

Figure 5.24: Trace display of the Rover model by highlighting the “MovingBackward” state



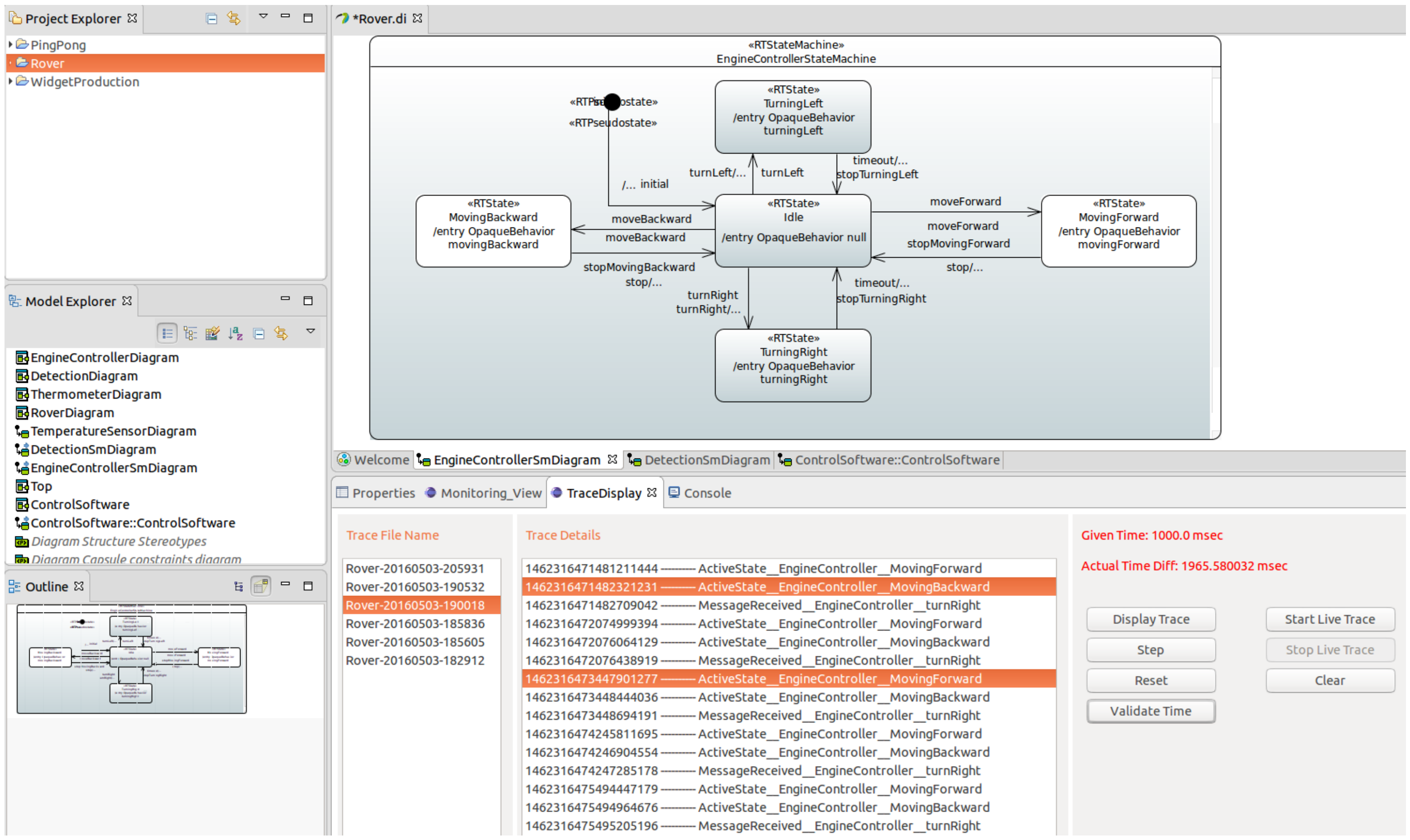
Trace File Name

```
Rover-20160503-205931
Rover-20160503-190532
Rover-20160503-190018
Rover-20160503-185836
Rover-20160503-185605
Rover-20160503-182912
```

Trace Details

```
1462316471481211444 ——— ActiveState__EngineController__MovingForward
1462316471482321231 ——— ActiveState__EngineController__MovingBackward
1462316471482709042 ——— MessageReceived__EngineController__turnRight
1462316472074999394 ——— ActiveState__EngineController__MovingForward
1462316472076064129 ——— ActiveState__EngineController__MovingBackward
1462316472076438919 ——— MessageReceived__EngineController__turnRight
1462316473447901277 ——— ActiveState__EngineController__MovingForward
1462316473448444036 ——— ActiveState__EngineController__MovingBackward
1462316473448694191 ——— MessageReceived__EngineController__turnRight
1462316474245811695 ——— ActiveState__EngineController__MovingForward
1462316474246904554 ——— ActiveState__EngineController__MovingBackward
1462316474247285178 ——— MessageReceived__EngineController__turnRight
1462316475494447179 ——— ActiveState__EngineController__MovingForward
1462316475494964676 ——— ActiveState__EngineController__MovingBackward
1462316475495205196 ——— MessageReceived__EngineController__turnRight
```

Figure 5.25: Trace display of the Rover model by highlighting the “TurnRight” transition



Rover-20160503-205931
Rover-20160503-190532
Rover-20160503-190018
Rover-20160503-185836
Rover-20160503-185605
Rover-20160503-182912

```
1462316471481211444 -------- ActiveState__EngineController__MovingForward
1462316471482321231 -------- ActiveState__EngineController__MovingBackward
1462316471482709042 -------- MessageReceived__EngineController__turnRight
1462316472074999394 -------- ActiveState__EngineController__MovingForward
1462316472076064129 -------- ActiveState__EngineController__MovingBackward
1462316472076438919 -------- MessageReceived__EngineController__turnRight
1462316473447901277 -------- ActiveState__EngineController__MovingForward
1462316473448444036 -------- ActiveState__EngineController__MovingBackward
1462316473448694191 -------- MessageReceived__EngineController__turnRight
1462316474245811695 -------- ActiveState__EngineController__MovingForward
1462316474246904554 -------- ActiveState__EngineController__MovingBackward
1462316474247285178 -------- MessageReceived__EngineController__turnRight
1462316475494447179 -------- ActiveState__EngineController__MovingForward
1462316475494964676 -------- ActiveState__EngineController__MovingBackward
1462316475495205196 -------- MessageReceived__EngineController__turnRight
```

Given Time: 1000.0 msec

Actual Time Diff: 1965.580032 msec

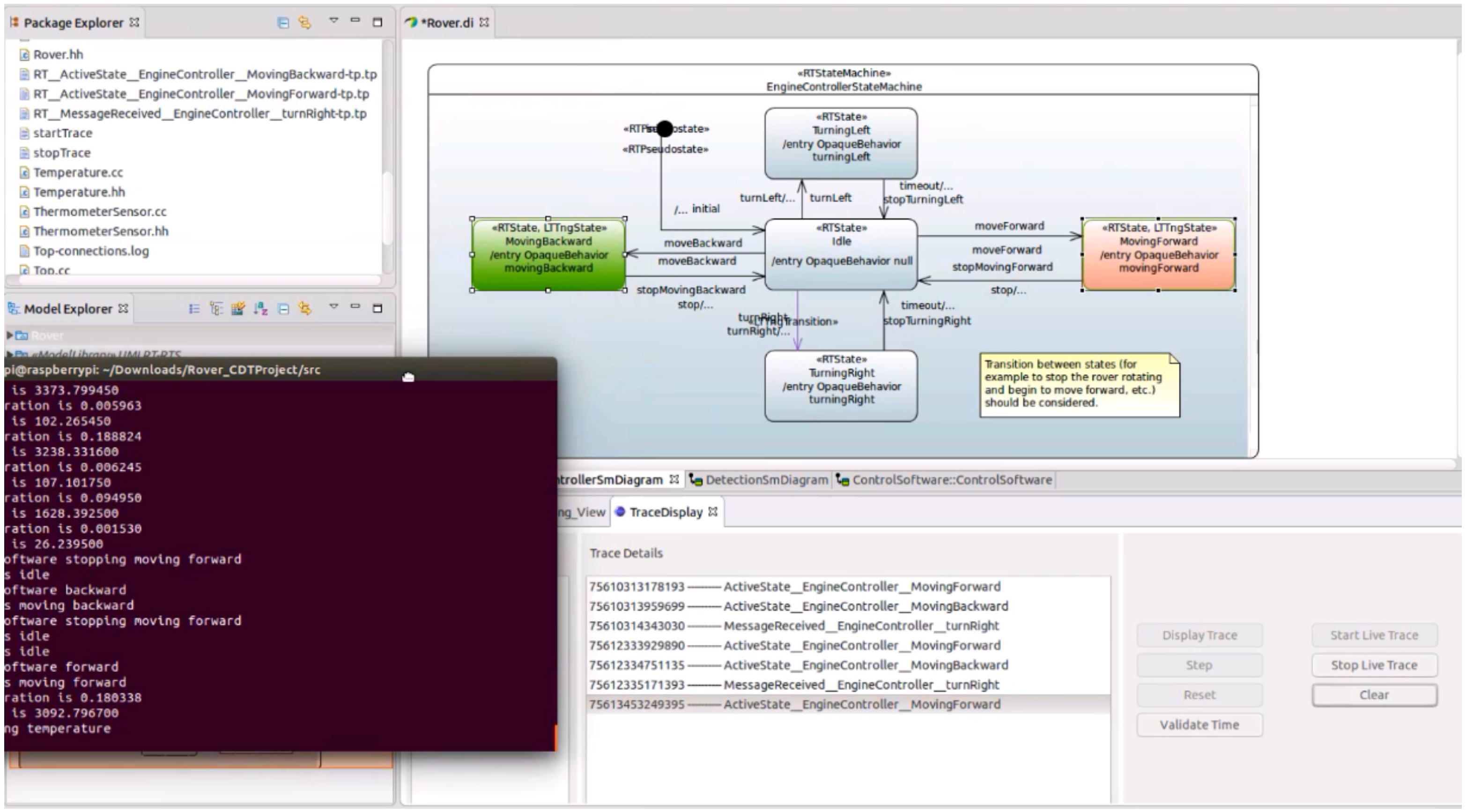
Figure 5.26: Violation of a time constraint in the Rover model

5.3.1 Offline Tracing

Fig. 5.23 shows the display related to the traces of the “MovingBackward” state, the “MovingForward” state and the “turnRight” transition of the EngineController state-machine. Clicking on the ”Step” button results in the display of subsequent events (see Fig. 5.24 and Fig. 5.25). Fig. 5.26 shows the violation of a given time constraint for two states of Engine Controller state-machine.

5.3.2 Online/Live Tracing

Fig. 5.27 shows the live traces of Rover model where the LTTng tool is used for tracing the model elements and for sending traces to the user machine over the network at the moment they arrive. Our plugin is capable of highlighting the associated model elements and the textual view at the same time for incoming trace events.



```
pi@raspberrypi: ~/Downloads/Rover_CDTProject/src
 is 3373.799450
ration is 0.005963
 is 102.265450
ration is 0.188824
 is 3238.331600
ration is 0.006245
 is 107.101750
ration is 0.094950
 is 1628.392500
ration is 0.001530
 is 26.239500
oftware stopping moving forward
s idle
oftware backward
s moving backward
oftware stopping moving forward
s idle
s idle
oftware forward
s moving forward
ration is 0.180338
 is 3092.796700
ng temperature
```

```
75610313178193 ——— ActiveState__EngineController__MovingForward
75610313959699 ——— ActiveState__EngineController__MovingBackward
75610314343030 ——— MessageReceived__EngineController__turnRight
75612333929890 ——— ActiveState__EngineController__MovingForward
75612334751135 ——— ActiveState__EngineController__MovingBackward
75612335171393 ——— MessageReceived__EngineController__turnRight
75613453249395 ——— ActiveState__EngineController__MovingForward
```

Figure 5.27: Screenshot of live traces of the Rover model

Chapter 6

Conclusion

6.1 Summary

The importance of quality management of complex software systems is significant as it can reduce the software maintenance cost remarkably. This research focuses on enhancing the quality of existing software models by detecting problems in the runtime behaviour. The model-driven development tool for real-time embedded systems Papyrus-RT is of interest to the modeling community because it is open source. Developers from different regions of world can contribute to the implementation of this tool. Moreover, LTTng is a great tool for monitoring runtime information with a minimum overhead. Therefore, we have used these tools for creating models for RTES and implementing monitoring.

The main goal of this research is to automate tracing of RTES and display trace results on the model level. We have focused on implementing a plugin to support monitoring of runtime information of RTES. To summarize our work, we have first created a plugin for automatically generating scripts with all the LTTng tracing commands. Support for reading the trace files is provided in the plugin. In addition, it is capable of displaying both offline and live traces in textual form and graphically in the model level. This gives us a way to verify any user defined time constraints and make it possible to find out the source of any delay in a running real-time embedded system.

6.2 Limitations

In this research, we have just provided a proof of our proposed concept. For real use of industrial system for validating timing constraints comprehensive observation of execution is important. Also it is important to know the factor of monitoring overhead introduced by LTTng.

In this research, we have only worked on two behavioural elements of a UML-RT model: state and transition. Runtime monitoring of other behavioural elements such as choice-points and junction-points are out of scope of this work. In addition, as we can see in Fig. 4.2, there are other model elements, i.e., structural components that might be monitored as well. Thus, our work will not be able to detect a problem if it involves structural design of a UML-RT capsule.

6.3 Future Work

There is a range of future work that can be done.

Till now there is no option of generating sequence diagrams in Papyrus-RT. The traces we collect contain the source and timing information of all the monitored events which can be used in future to generate a sequence diagram.

As we have mentioned while discussing the limitations of this work, runtime monitoring of only a subset of behavioural elements of a UML-RT capsule is currently supported. This can be enhanced by adding support for monitoring other behavioural elements as well as the structural components.

In addition, the traces we get can also be used for animating and simulating the model execution. Therefore, our plugin can be used for both tracing and simulating a real-time embedded system.

6.4 Conclusion

The importance of runtime monitoring is significant to ensure the correctness of the runtime behavior of complex real time embedded systems. This research is an attempt to examine the correctness of timing information related to UML-RT models using runtime traces generated by the open-source LTTng tool. We provide support for tracing a user application, reading a trace file, displaying the trace results on the model level, exploring the associated timestamps in textual form and verifying the actual timing information of a trace file against the desired user input. Some of the challenges we faced while doing this research include dealing with the unstable releases of Papyrus-RT tool as it was in the initial phase of development and managing Eclipse-based issues such as lack of backward-compatibility for the models created in an old Eclipse version. This research constitutes a first step to combining MDE and runtime monitoring and verification of timing constraints.

Bibliography

[1] Babeltrace. http://www.efficios.com/babeltrace. Accessed: 2016-08-25.

[2] dSpace. https://www.dspace.com/en/inc/home.cfm. Accessed: 2016-08-20.

[3] Five Things that make Xtend a great Language for Java Developers. http://www.sebastianbenz.de/5-Things-that-make-Xtend-a-great-Language-for-Java-developers. Accessed: 2016-06-31.

[4] Howto tracing with LTTng. https://www.ibm.com/developerworks/community/blogs/fe313521-2e95-46f2-817d-44a4f27eba32/entry/howto_tracing_with_lttng?lang=en. Accessed: 2016-07-09.

[5] IBM: Modeling real-time applications in RSARTE. https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W0c4a14ff363e_436c_9962_2254bb5cbc60/page/Modeling%20Real-Time%20Applications%20in%20RSARTE. Accessed: 2016-08-20.

[6] IBM: Rational Rose Real Time. ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rosert.html. Accessed: 2016-08-20.

[7] Instruments User Guide. https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/CreatingCustomInstruments.html. Accessed: 2016-06-19.

[8] LabVIEW System Design Software. http://www.ni.com/labview/. Accessed: 2016-08-20.

[9] Lines of code in Google. http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/. Accessed: 2016-07-18.

[10] The LTTng Documentation. http://lttng.org/docs/. Accessed: 2016-06-01.

[11] Makefiles. http://mrbook.org/blog/tutorials/make/. Accessed: 2016-08-16.

[12] Modeling language. https://en.wikipedia.org/wiki/Modeling_language. Accessed: 2016-06-25.

[13] Modeling Language Guide, Rational Rose Realtime - ibm.com. ftp://ftp.software.ibm.com/software/rational/docs/v2003/win_solutions/rational_rosert/rosert_modeling_language.pdf. Accessed: 2016-07-24.

[14] Object management group. http://www.omg.org/. Accessed: 2016-06-24.

[15] Object Management Group. https://en.wikipedia.org/wiki/Object_Management_Group. Accessed: 2016-06-24.

[16] Papyrus. https://wiki.eclipse.org/Papyrus. Accessed: 2016-08-01.

[17] Papyrus for Real Time (Papyrus-RT). https://projects.eclipse.org/projects/modeling.papyrus-rt. Accessed: 2016-07-24.

[18] Profile (UML). `https://en.wikipedia.org/wiki/Profile_(UML)`. Accessed: 2016-07-18.

[19] Rover. `https://www.hackster.io/peejster/rover-c42139`. Accessed: 2016-08-28.

[20] Shell Script. `http://linuxcommand.org/writing_shell_scripts.php`. Accessed: 2016-08-25.

[21] Simulink. `http://www.mathworks.com/products/simulink/`. Accessed: 2016-08-20.

[22] Trace Compass. `http://tracecompass.org/`. Accessed: 2016-08-25.

[23] UML Stereotypes. `http://www.sparxsystems.com/enterprise_architect_user_guide/10/standard_uml_models/stereotypedlg.html`. Accessed: 2016-07-23.

[24] Unified modeling language. `http://www.uml.org/`. Accessed: 2016-06-25.

[25] Xtend. `http://www.eclipse.org/xtend/documentation/index.html`. Accessed: 2016-06-01.

[26] Object management group (OMG). Semantics of a foundational subset for executable UML models (fUML). `http://www.omg.org/spec/FUML/1.2.1`, 2016.

[27] Papyrus: Moka overview. `http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`, 2016.

[28] Reza Ahmadi, Nicolas Hili, Leo Jweda, Nondini Das, Suchita Ganesan, and Juergen Dingel. Run-time Monitoring of a Rover: MDE Research with Open

Source Software and Low-cost Hardware. In *2nd International Workshop on Open Source Software for Model Driven Engineering (OSS4MDE'16)*, 2016. To appear.

[29] Sinan Si Alhir. *Guide to Applying the UML*. Springer Science & Business Media, 2006.

[30] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[31] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006.

[32] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on uml class diagrams. *Artificial Intelligence*, 168(1):70–118, 2005.

[33] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[34] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[35] Marco Blumendorf, Grzegorz Lehmann, and Sahin Albayrak. Bridging models and systems at runtime to build adaptive user interfaces. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 9–18. ACM, 2010.

[36] Matteo Bordin and Tullio Vardanega. Automated model-based generation of ravenscar-compliant source code. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 59–67. IEEE, 2005.

[37] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[38] Giorgio Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[39] Betty HC Cheng, Kerstin I Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A Müller, Patrizio Pelliccione, Anna Perini, Nauman A Qureshi, Bernhard Rumpe, et al. Using models at runtime to address assurance for self-adaptive systems. In *Models@ run. time*, pages 101–136. Springer, 2014.

[40] Shang-Wen Cheng and David Garlan. Mapping architectural concepts to uml-rt. 2001.

[41] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 36–43. ACM, 2016.

[42] Mathieu Desnoyers. Common trace format (CTF) specification (v1. 8.2). *Common Trace Format GIT repository*, 2012.

[43] Mathieu Desnoyers and Michel R Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[44] Bruce Powel Douglass. *Real time UML: advances in the UML for real-time systems*. Addison-Wesley Professional, 2004.

[45] Ken Edwards and Gabriel Wainer. Gatlas: Google earth visualization for atlas. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 213–220. Society for Computer Simulation International, 2011.

[46] Xiaocong Fan. *Real-time Embedded Systems: Design Principles and Engineering Practices*. Newnes, 2015.

[47] Victor Freire, Sixuan Wang, and Gabriel Wainer. Visualization in 3ds max for cell-devs models based on building information modeling. In *Proceedings of the Symposium on Simulation for Architecture & Urban Design*, page 9. Society for Computer Simulation International, 2013.

[48] Sébastien Gérard, Huascar Espinoza, François Terrier, and Bran Selic. 6 modeling languages for real-time and embedded systems. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 129–154. Springer, 2010.

[49] Philipp Graf and Klaus D Muller-Glaser. Gaining insight into executable models during runtime: Architecture and mappings. *IEEE Distributed Systems Online*, 8(3):1–1, 2007.

[50] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451, 2006.

[51] Zef Hemel, Lennart CL Kats, Danny M Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software & Systems Modeling*, 9(3):375–402, 2010.

[52] Zef Hemel, Lennart CL Kats, and Eelco Visser. Code generation by model transformation. In *International Conference on Theory and Practice of Model Transformations*, pages 183–198. Springer, 2008.

[53] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[54] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[55] Shahar Maoz. Model-based traces. In *International Conference on Model Driven Engineering Languages and Systems*, pages 109–119. Springer, 2008.

[56] Tanja Mayerhofer and Philip Langer. Moliz: A model execution framework for UML models. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, page 3. ACM, 2012.

[57] Stephen J Mellor, Tony Clark, and Takao Futagami. Model-driven development: guest editors' introduction. *IEEE Software*, 20(5):14–18, 2003.

[58] Mohammad Moallemi. *Real-Time and Embedded Systems Development based on Discrete Event Modeling and Simulation*. PhD thesis, Carleton University Ottawa, 2011.

[59] Ernesto Posse. Papyrusrt: Modelling and code generation (invited presentation). In *OSS4MDE@ MoDELS*, pages 54–63, 2015.

[60] Ernesto Posse and Juergen Dingel. An executable formal semantics for UML-RT. *Software & Systems Modeling*, 15(1):179–217, 2016.

[61] Bran Selic. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.

[62] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19, 2003.

[63] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-time object-oriented modeling*, volume 2. John Wiley & Sons New York, 1994.

[64] Dominique Toupin. Using tracing to diagnose or monitor systems. *IEEE Software*, 28(1):87, 2011.

Appendix A

Sources for Generating Scripts

Listing A.1: Start Trace Script Generator

```
class StartTraceScriptGenerator {
  def generate( String path, String mainFileName, String folderName )
    {

        val file = new File(path);

        val writer = new BufferedWriter( new FileWriter( file ) );
        writer.write( doGenerate( mainFileName, folderName ).toString )
        writer.close
    }

    def private doGenerate( String mainFileName, String folderName ) {
        '''
    #!/bin/bash
    # Script for executing LTTng Commands

    network=
    ip=
    live=
    while [ $# -gt 0 ]
    do
      case "$1" in
      -n)
         network="--set-url net://"
           ip="$2";
           shift;;
         -l)
           live="--live";;
         --) shift; break;;
         -*)
         echo >&2 "usage: $0 [-l] [-n] [ip address]"
             exit 1;;
         *)  break;; # terminate while loop
      esac
      shift
    done

    NOW=$(date +"%Y%m%d_%H%M%S")
    lttng create  folderName  -o ../../ folderName/folderName_$NOW $live
        $network$ip

    lttng enable-event -u 'RT__*'
    lttng start
    read -p "Press any key to continue..."
    ./ mainFileName
        '''
    }
}
```

Listing A.2: Stop Trace Script Generator

```
class StopTraceScriptGenerator {
  def generate( String path )
    {

        val file = new File(path);

        val writer = new BufferedWriter( new FileWriter( file ) );
        writer.write( doGenerate( ).toString )
        writer.close
    }

    def private doGenerate( ) {
        '''
    #!/bin/bash
    # Script for stopping LTTng Commands

    lttng stop
    lttng destroy
        '''
    }
}
```