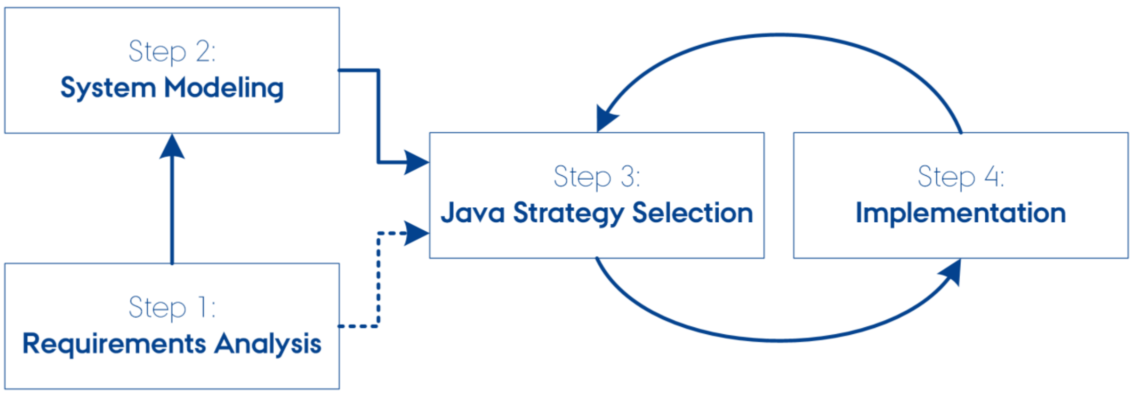
A METHODOLOGY FOR TRANSFORMING JAVA APPLICATIONS TOWARDS REAL-TIME PERFORMANCE

Electrical and Computer Engineering
Technical Report ECE-TR-11



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: A Methodology for Transforming Java Applications Towards Real-Time Performance

Subtitle: Electrical and Computer Engineering
Series title and no.: Technical Report ECE-TR-11

Authors: Mads von Qualen and Martin Askov Andersen
Department of Engineering - Electrical and Computer Engineering, Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2013 **Pages:** 139
Editing completed: December 2012

Abstract: The development of *real-time* systems has traditionally been based on low-level programming languages, such as C and C++, as these provide a fine-grained control of the applications temporal behavior. However, the usage of such programming languages suffers from increased complexity and high error rates compared to high-level languages such as Java. The Java programming language provides many benefits to software development such as automatic memory management and platform independence. However, Java is unable to provide any real-time guarantees, as the high-level benefits come at the cost of unpredictable temporal behavior.
This thesis investigates the temporal characteristics of the Java language and analyses several possibilities for introducing real-time guarantees, including official language extensions and commercial runtime environments. Based on this analysis a new methodology is proposed for *Transforming Java Applications towards Real-time Performance* (TJARP). This method motivates a clear definition of timing requirements, followed by an analysis of the system through use of the formal modeling language VDM-RT. Finally, the method provides a set of structured guidelines to facilitate the choice of strategy for obtaining real-time performance using Java. To further support this choice, an analysis is presented of available solutions, supported by a simple case study and a series of benchmarks.
Furthermore, this thesis applies the TJARP method to a complex industrial case study provided by a leading supplier of mission critical systems. The case study proves how the TJARP method is able to analyze an existing and complex system, and successfully introduce hard real-time guarantees in critical sub-components.

Keywords: Real-time, Java, VDM-RT, Methodology, TJARP

Supervisor: Peter Gorm Larsen

Please cite as: Qualen, M.v. and Andersen, M.A., 2013.
A Methodology for Transforming Java Applications Towards Real-Time Performance. Department of Engineering, Aarhus University. Denmark. 139 pp. - Technical report ECE-TR-11.

Cover image: Mads von Qualen and Martin Askov Andersen

ISSN: 2245-2087

A METHODOLOGY FOR TRANSFORMING JAVA APPLICATIONS TOWARDS REAL-TIME PERFORMANCE

Mads von Qualen and Martin Askov Andersen

Aarhus University, Department of Engineering

Abstract

The development of *real-time* systems has traditionally been based on low-level programming languages, such as C and C++, as these provide a fine-grained control of the applications temporal behavior. However, the usage of such programming languages suffers from increased complexity and high error rates compared to high-level languages such as Java. The Java programming language provides many benefits to software development such as automatic memory management and platform independence. However, Java is unable to provide any real-time guarantees, as the high-level benefits come at the cost of unpredictable temporal behavior.

This thesis investigates the temporal characteristics of the Java language and analyses several possibilities for introducing real-time guarantees, including official language extensions and commercial runtime environments. Based on this analysis a new methodology is proposed for *Transforming Java Applications towards Real-time Performance* (TJARP). This method motivates a clear definition of timing requirements, followed by an analysis of the system through use of the formal modeling language VDM-RT. Finally, the method provides a set of structured guidelines to facilitate the choice of strategy for obtaining real-time performance using Java. To further support this choice, an analysis is presented of available solutions, supported by a simple case study and a series of benchmarks.

Furthermore, this thesis applies the TJARP method to a complex industrial case study provided by a leading supplier of mission critical systems. The case study proves how the TJARP method is able to analyze an existing and complex system, and successfully introduce hard real-time guarantees in critical sub-components.

Acknowledgements

Several people have assisted the work of this thesis for which we are grateful. We would especially like to thank our academic supervisor Peter Gorm Larsen for his involvement and advice during this thesis and Master's degree. We would also like to thank Terma A/S, including Mikkel Elkjær Hede, for providing an industrial case study and their dedication to this thesis. Special thanks go to our industrial supervisor, Thomas Gjørup from Terma, for his contribution to the case study and general support.

Additionally we would like to thank Atego, and especially Tom Grosman, for providing us with access to, and support for, the Atego PERC products.

Finally, we would like to thank our families and friends for their patience and support.

Table of Contents

Abstract i

Acknowledgements iii

Table of Contents v

List of Figures viii

List of Tables x

Chapter 1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Motivation . . . 2
- 1.3 Thesis Goals, Approach and Scope . . . 3
  - 1.3.1 Goals . . . 3
  - 1.3.2 Approach . . . 3
  - 1.3.3 Scope . . . 3
- 1.4 The Method . . . 4
- 1.5 Case Studies . . . 5
  - 1.5.1 Car Controller . . . 5
  - 1.5.2 T-Core . . . 5
- 1.6 Reading Guide . . . 7
  - 1.6.1 Structure . . . 7

Chapter 2 Java and Real-Time 11
- 2.1 Introduction . . . 11
- 2.2 Scheduling . . . 12
  - 2.2.1 Real-Time Systems . . . 12
  - 2.2.2 Standard Java . . . 14
- 2.3 Synchronization . . . 15
  - 2.3.1 Real-Time Systems . . . 16
  - 2.3.2 Standard Java . . . 16
- 2.4 Memory Management . . . 17
  - 2.4.1 Real-Time Systems . . . 18
  - 2.4.2 Standard Java . . . 18
- 2.5 Discussion . . . 21

Chapter 3 Real-Time Extensions for Java 23
- 3.1 Introduction . . . 23

3.2 The Real-Time Specification for Java . . . 23
3.2.1 Scheduling . . . 24
3.2.2 Synchronization . . . 25
3.2.3 Memory Management . . . 25
3.3 Safety Critical Java . . . 27
3.3.1 Scheduling . . . 27
3.3.2 Synchronization . . . 27
3.3.3 Memory Management . . . 28
3.4 Discussion . . . 28

Chapter 4 Real-Time Requirements **31**
4.1 Introduction . . . 31
4.2 Sub-steps of the TJARP Method . . . 32
4.3 Requirements in Real-Time Systems . . . 32
4.3.1 Functional vs. Non-Functional . . . 32
4.3.2 Types of Timing Requirements . . . 33
4.4 The Car Controller Example . . . 34
4.4.1 The Functional Requirements . . . 34
4.4.2 The Non-Functional Requirements . . . 35
4.5 Discussion . . . 35

Chapter 5 Modeling Real-Time Systems **37**
5.1 Introduction . . . 37
5.2 Sub-steps of the TJARP method . . . 38
5.3 Tool Support . . . 39
5.4 Modeling with VDM-RT . . . 40
5.4.1 Modeling System Structure . . . 40
5.4.2 Introducing Concurrency . . . 41
5.4.3 Analyzing Timing Constraints . . . 42
5.4.4 Design Space Exploration . . . 44
5.5 Discussion . . . 45

Chapter 6 Towards Real-Time Java **47**
6.1 Introduction . . . 47
6.2 Sub-steps of the TJARP method . . . 48
6.3 Optimization of Standard Java . . . 49
6.3.1 Development . . . 49
6.3.2 Deployment . . . 50
6.3.3 Runtime Environment . . . 51
6.4 Analysis and Benchmark of Java Virtual Machines . . . 53
6.4.1 Basis for Comparison of Java Virtual Machines . . . 54
6.4.2 Benchmark Approach . . . 55
6.4.3 Oracle HotSpot . . . 57
6.4.4 Atego PERC Ultra . . . 59
6.4.5 Aicas JamaicaVM . . . 62
6.5 Utilizing the Real-Time Specification for Java . . . 64
6.5.1 Scheduling . . . 65
6.5.2 Synchronization . . . 65
6.5.3 Memory Management . . . 66

6.6 Discussion . . . 67
6.6.1 Optimizing Standard Java . . . 67
6.6.2 Substituting JVM . . . 67
6.6.3 Applying the RTSJ . . . 69

Chapter 7 Case Study: Terma T-Core **71**
7.1 Introduction . . . 71
7.2 Requirements Analysis . . . 73
7.2.1 Functional Requirements . . . 73
7.2.2 Non-Functional Requirements . . . 74
7.3 System Modeling . . . 75
7.3.1 Modeling System Structure . . . 75
7.3.2 Introducing Concurrency . . . 77
7.3.3 Analyzing Timing Constraints . . . 79
7.3.4 Design Space Exploration . . . 80
7.4 Java Strategy Selection and Implementation . . . 81
7.4.1 The Track Management Component . . . 81
7.4.2 The Engagement Manager Component . . . 83
7.5 Discussion . . . 86

Chapter 8 Concluding Remarks and Future Work **89**
8.1 Introduction . . . 89
8.2 Achieved Results . . . 89
8.2.1 Step 1: Requirements Analysis . . . 90
8.2.2 Step 2: System Modelling . . . 90
8.2.3 Step 3 and 4: Selecting and Implementing Java Strategy . . . 91
8.2.4 Applying the TJARP Method on the T-Core Case Study . . . 92
8.3 Future Work . . . 93
8.3.1 Letting the TJARP Method Further Exploit the VDM Model . . . 93
8.3.2 Utilize VDM Modeling for Designing RTSJ Applications . . . 93
8.3.3 Extending the TJARP Method with Support for Mission Critical Systems 93
8.3.4 Extending the TJARP Method with Support for Distributed Nodes . . . 94
8.3.5 Further Exploiting the VDM Language and Tool Support . . . 94
8.3.6 Improvements to the T-Core Case Study . . . 94
8.4 Personal Learning Outcomes . . . 95
8.5 Final Remarks . . . 96

Appendices **107**

A Terminology **109**

B Case Study Details **111**

C Overture Real-Time Log Viewer **123**

D Java Virtual Machine Analysis **127**

E Distributed Computing **137**

List of Figures

1.1 Overview of the TJARP method. 4
1.2 Entity Overview of the Car Controller example 6
1.3 Entity Overview of the T-Core Track Management System 6
1.4 Overview of the thesis structure 9

2.1 Scheduling in standard Java 15
2.2 Synchronization in standard Java 17
2.3 Jitter experienced by the `CruiseController` thread 20

3.1 Relationship between standard Java threads and RTSJ threads 24
3.2 Legal and illegal references between memory areas in RTSJ 26

4.1 Step 1 of the TJARP method 31

5.1 Step 2 of the TJARP method 37
5.2 Structural class diagram of the Car Controller model 41
5.3 Example of a conjecture violation for the Car Controller example 44

6.1 Step 3 and 4 of the TJARP method 47
6.2 Transitions between sub-steps in step 3 and 4 48
6.3 Car Controller Memory Profile 52
6.4 Jitter distribution for the HotSpot test three 59
6.5 Jitter distribution for Atego PERC Ultra test three 61
6.6 Jitter distribution for the JamaicaVM in test three with RTSJ memory and threads 64
6.7 Car Controller example redesigned for RTSJ 65
6.8 Scheduling in RTSJ 65
6.9 Synchronization in RTSJ 66
6.10 Comparison of the Oracle HotSpot, the Atego PERC Ultra and the Aicas JamaicaVM 68

7.1 The four steps of the TJARP method 71
7.2 Components of the T-Core Case Study 72
7.3 Simplified class diagram of the T-Core VDM-RT model 76
7.4 Sequence diagram of a track updated and engaged 78
7.5 Track Load Tests - PERC Ultra vs. HotSpot 82
7.6 Engagement Manager Class Diagram 84
7.7 Ideal Timing for the Engagement Manager Component 86
7.8 Total Handling Duration for the Engagement Manager Component 86

8.1 Degrees of real-time performance in the T-Core case study components 92

B.1 Class diagram of the Car Controller software . . . 112
B.2 Structural class diagram of the Car Controller model . . . 113
B.3 Sequence diagram for track creation in the model . . . 114
B.4 Evaluation Duration for the `TrackEvaluationHandler` . . . 120
B.5 Evaluation/Communication Delay Between the `TrackEvaluationHandler` and the `WeaponComHandler` . . . 120
B.6 Weapon Communication Total Duration for the `WeaponComHandler` . . . 121
B.7 Weapon Communication Period for the `WeaponComHandler` . . . 121

C.1 Old RT Log Viewer . . . 124
C.2 New RT Log Viewer . . . 124
C.3 Class diagram of the new RTLV design . . . 126
C.4 New vs. old RTLV load times . . . 126

D.1 CPU profile of the SPECjvm2008 Compress benchmark . . . 130
D.2 Memory profile of the SPECjvm2008 Compress benchmark . . . 130
D.3 Jitter distribution for the HotSpot test two . . . 133
D.4 Jitter distribution for the HotSpot test three . . . 133
D.5 Jitter distribution for the PERC Ultra test two . . . 133
D.6 Jitter distribution for the PERC Ultra test three . . . 134
D.7 Jitter distribution for the JamaicaVM test two with RTSJ . . . 134
D.8 Jitter distribution for the JamaicaVM test three with RTSJ . . . 134
D.9 Jitter distribution for the JamaicaVM test three with RTSJ and real-time RT Linux priorities . . . 135

List of Tables

2.1 Jitter statistics for the `CruiseController` thread 21

6.1 Comparison of jitter on HotSpot, with default and optimized settings and the PERC Ultra 52
6.2 Overview of applied benchmark tests 56
6.3 Oracle HotSpot Grades 57
6.4 Test results for Oracle HotSpot based on samples 500-5000 58
6.5 Atego PERC Ultra Grades 59
6.6 Test results for Atego PERC Ultra based on samples 500-5000 61
6.7 Aicas JamaicaVM Grades 62
6.8 Test results for Aicas JamaicaVM based on samples 500-5000 63
6.9 Jitter statistics for the `CruiseController` thread 66

7.1 Engagement Manager Test Results, using the JamaicaVM 85

D.1 Configuration parameters for GC optimizing the HotSpot JVM 127
D.2 Configuration parameters for GC optimizing the PERC Ultra JVM 128
D.3 Scale for rating the attributes of the individual JVMs 129
D.4 Configuration parameters for the SPECjvm2008 tests 131
D.5 Configuration parameters for the Collision Detector benchmark 132

Chapter 1

Introduction

This chapter describes the context of this thesis, and presents the background, motivation as well as goal and purpose of the completed work. This chapter sets the stage for the remaining chapters.

1.1. Background

The Java language has become one of the most popular programming languages in use since its introduction in the early nineties [TIOBE12]. The language has had an enormous impact on the software industry and is currently deployed in millions of systems ranging from large enterprise applications to low-end embedded devices [Higuera-Toledano&12]. Research has shown how Java is still increasing in popularity and use, both in the industry but also within academia [Chen&05]. The success of the Java programming language is highly due to its platform independence as emphasized by the Java mantra, *"Write once, run everywhere"*. Other compelling advantages include an object-oriented programming model, extensive library support, type safety, automatic memory management and build-in support for multithreading and distributed programming. However a domain still dominated by more low-level languages such as C and C++, and where Java is still trying to gain acceptance, is within the area of *real-time* systems.
The success or failure of real-time systems depend not only on their functional behavior, but also on their ability to meet critical deadlines, e.g. a missed or premature deadline in a medical pacemaker may have catastrophic consequence. A common denominator for real-time systems is that they are subject to real-world timing constraints [IBM07], where "real-time" is not a measure of being "real fast" but the guarantee of being "equally fast". This concept can further be divided into soft and hard real-time, where the first describes a system which might accept a result after a missed deadline, but the usefulness degrades as the deadline is passed. For hard real-time systems a missed deadline equals total system failure and is unacceptable. Real-time applications are often central components in *Mission Critical* systems where human life and significant costs are at stake. Such systems have a natural need for being reliable, and are often subject to strict certifications requirements, thus reducing complexity, and thereby the number of potential errors is of extra importance.
Real-time software practitioners have traditionally preferred low-level programming such as C or C++ due to a higher degree of control and predictability of timing behavior. However, given the complexity of such languages, they are more prone to errors and developers have proven to be less productive than those using high-level programming languages such as Java. Java developers are reported to be up to 200 percent more productive (in terms of lines of code per minute) and with

only half as many bugs (per lines of code) as C++ developers [Phipps99]. This makes using Java for development of real-time systems (hard or soft) attractive, especially considering the many benefits and widespread usage. However, Java suffers from non-deterministic temporal behavior, severely limiting its usage within the real-time domain.

Java is essentially two things: a programming language and a runtime environment. The first contribute to the non-deterministic temporal behavior through language facilities, such as dynamic class loading and automatic memory management. The latter adds unpredictable behavior through native code compilation, synchronization mechanisms, unpredictable scheduling policies etc.

For more than a decade the Java community has pushed towards specifications and optimization techniques for improving temporal behavior of the Java language [JSR001, Mikhalenko06]. This work has inspired official ⌜*Java Specification Requests*, such as the *Real-time Specification for Java* (RTSJ), and later the *Safety Critical Java* (SCJ). Both specifications are important contributions but a number of open problems have limited its widespread acceptance, leaving a fragmented community motivated by specific commercial interests on one side, and broader diverse academic approaches on the other [Plsek09].

1.2. Motivation

The primary motivation behind this thesis is the authors' wide interest within the field of real-time software engineering. Both authors' have attended courses within the field, where especially the courses *Modeling of Mission Critical Systems* and *Architecture and Design of Embedded Real-Time System* have served as an inspiration. The project proposal was initiated by a request from the Danish high-tech company Terma A/S. Terma is a leading supplier of mission critical software systems used in extreme environments, where the ability to guarantee predictability and determinism at certain points in time is crucial. Terma has for several years used their *Flexible Software Platform for Combat Management Systems* (T-Core) as a foundation for various command and control systems. Several measures have been taken in order to optimize the platform for real-time performance. Terma wish to further investigate the possibilities for introducing temporal guarantees in certain sub-components of the T-Core framework. The company has, as many other members of the industry, faced challenges when trying to choose the correct strategy for isolating, analyzing and optimizing critical Java components in order to achieve real-time performance. Currently Terma implements critical components, with hard real-time requirements, in low-level programming languages such as C and C++. Terma wish to utilize Java for implementing these components in order to reduce the required workload, and take advantage of the high-level benefits provided by the language. Therefore Terma have provided a real-life case study to apply the work of this thesis.

It is clear from the many benefits of Java that bridging the gap between existing real-time techniques and Java would be very attractive. The initial purpose of this thesis was to investigate if, and how, an existing Java implementation could be optimized for real-time performance. From review of literature, it quickly became clear that real-time guarantees with Java were achievable, but the options were many and unclear. Many official proposals and commercial products try to address the challenge of achieving real-time performance using Java. Furthermore, the task of defining requirements with emphasis on timing constraints is difficult and often subject to various interpretations. This led to a change in focus, towards defining a new methodology for facilitating the process of transforming Java applications towards real-time performance.

1.3. Thesis Goals, Approach and Scope

This section presents the goals of this master's thesis, the approach for achieving them and limits the scope of the work.

1.3.1 Goals

The main goals of this thesis are:

1. **To provide an overview of available real-time Java technologies through evaluation and comparison, which will assist the choice of the optimum strategy towards achieving real-time performance.**

2. **To propose a methodology which will facilitate the process of introducing real-time performance in existing Java applications.**

In addition, the authors of this thesis have personal goals of improving their skills and knowledge within the fields of real-time systems as well as in Java and the technologies for real-time Java.

1.3.2 Approach

In order to achieve the goals set out for this thesis, a three phased approach was used.

Phase 1 – Research: Initially a research phase was performed. Here relevant research was examined in order to uncover the challenges faced when developing traditional real-time systems. Furthermore, the problems preventing standard Java from providing real-time guarantees were studied. Finally, a survey of available real-time Java solutions on the market was conducted. The knowledge gained through this phase provided a strong basis for the activities in the following phase.

Phase 2 – Elaboration: Secondly an elaboration phase was carried out. Here the methodology mentioned in the thesis goals was developed, and a supporting analysis was performed. The methodology was based on real-time requirements and formal modeling of the Java application in question. This helped ensure a clear understanding of what can be achieved before choosing a real-time Java strategy.

Phase 3 – Evaluation: Finally an evaluation phase was performed in order to evaluate the methodology developed during phase 2. The evaluation was done by utilizing the methodology on a real-life case study and then assess the obtained outcome.

1.3.3 Scope

As the subject of real-time Java is vast and the possibilities many, a few limitations have been set up in order to bound the amount of work associated with this master's thesis.

Distributed systems: This thesis will focus on how to achieve real-time performance on a single computing node. However, distributed systems are considered in this thesis, nevertheless the challenges faced when trying to provide real-time guarantees across a communication link are out of the scope of this thesis.

New Java Applications: The methodology developed through this thesis will focus on how to introduce real-time in existing Java-based applications. The development of new real-time Java applications is therefore out of the scope of this thesis. However, although the methodology assumes an existing application, the experiences and results of this thesis will still be highly relevant when developing new applications.

Embedded Systems: Many real-time Java approaches are targeted embedded systems. This thesis will however, focus on computer systems in general. That said, the methodology or parts of it may well be applicable to embedded systems.

Modeling Techniques: The methodology developed through this thesis will make use of the formal modeling language VDM. Based on the authors' prior knowledge and interest in VDM, this has been preferred as modeling technique in this thesis, even though alternatives exist.

1.4. The Method

This section provides an overview of the proposed methodology for *Transforming Java Applications towards Real-time Performance* (TJARP). The overall goal of the TJARP method is to facilitate the process of introducing real-time behavior in existing Java-based applications. This is done through four steps, each comprised of a set of sub-steps. An overview of the four steps and how they are related can be seen in figure 1.1.

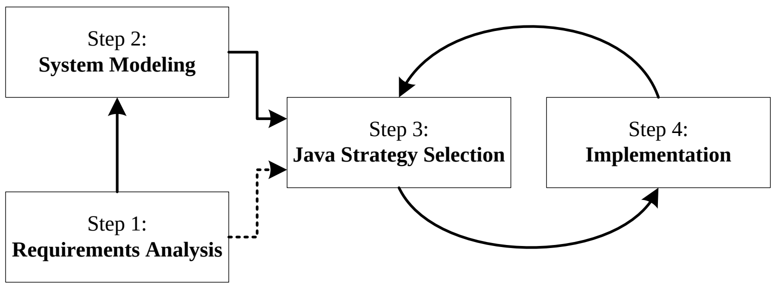


Figure 1.1: Overview of the TJARP method.

The dotted arrow between step 1 and step 3 indicates that under certain circumstances it is possible to go directly from step 1 to step 3 and thus skipping step 2. The motivation for skipping the second step is further described in section 4.5. The arrows between step 3 and step 4 show that these steps are repeated iteratively until a satisfactory result is obtained. A description of each step is given here along with their individual goals:

Requirements Analysis: This step is concerned with defining the desired real-time properties of the system through requirements. The existing Java application and associated functional requirements will be supplemented by a set of non-functional timing requirements in this step.

System Modeling: This step deals with modeling the most essential parts of the system, which requires real-time performance using VDM-RT. The model will help gain a better understanding of the system and the desired real-time properties. Furthermore, the model will assist in identifying potential design flaws and bottlenecks which could potentially impact the real-time performance. Finally the model is useful for doing early design space explorations in order to choose the best possible architectural design.

Java Strategy Selection: This step will facilitate the choice of strategy for obtaining real-time performance using Java, based on the requirements and the VDM-RT model. This step can be revisited after step 4 if the desired result was not obtained through the strategy already used. Hence, results from earlier tests can be used as input to this step as well.

Implementation: This is the final step of the method. It is concerned with implementing the changes to the application using the real-time Java strategy chosen in step 3. Afterwards the results are evaluated, and if they are satisfactory, compared to the requirements defined in step 1, then the use of the method is completed. Otherwise, step 3 is revisited using the newly obtained results and knowledge as input.

The details of all steps and their sub-steps will be described in chapters 4 to 6.

1.5. Case Studies

Two case studies are used in this thesis. One is a simple fictional case study called the *Car Controller* example which is used throughout this thesis to emphasize important points and exemplify use of the TJARP method. The second case study is more complex and based on the real-life system T-Core provided by Terma. This case is used to demonstrate how the TJARP method can be applied on a complex industrial case. The Car Controller example is introduced in section 1.5.1 while T-Core is introduced in section 1.5.2.

1.5.1 Car Controller

The Car Controller application is an imaginary real-time system imitating the behavior of a car. The system must provide the user with automatic cruise control, which samples the current speed and regulates the engine in order to achieve the desired cruise speed. Together with monitoring brake and gas pedals the system contains a navigation display and a cruise-control switch. The example abstracts away many details in order to focus purely on important real-time parts. An overview of the Car Controller example is illustrated in figure 1.2.
The Car Computer is the processing unit of the system where a Java application is executing and interfacing to several peripherals found in the car. Input is received from the gas pedal and the brake pedal and it is the job of the Car Computer to adjust the speed of the engine and the pressure of the brakes accordingly. The Car Computer also receives input from a cruise control switch which is able to enable and disable the cruise controller and configure the desired cruise speed. The Car Computer must then monitor the speed of the engine and change it accordingly. Finally the Car Computer is responsible for updating the in-car display with navigation information.
In addition to the overview presented in figure 1.2 a class diagram describing the software elements of the Car Computers Java application is provided in appendix B.

1.5.2 T-Core

T-Core is a framework developed by Terma, which is a distributed and component based *Combat Management System* platform [Terma11]. T-Core is developed in Java and forms the basis for

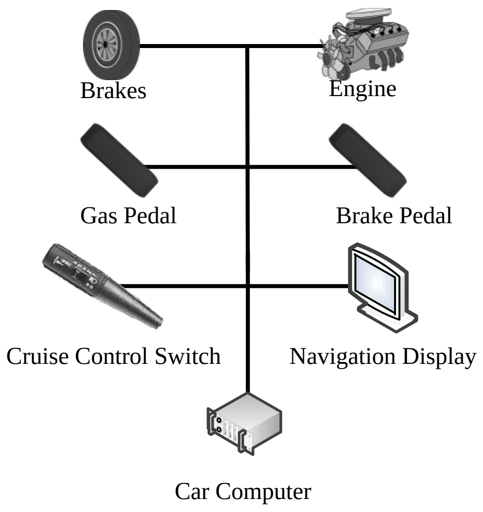


Figure 1.2: Entity Overview of the Car Controller example

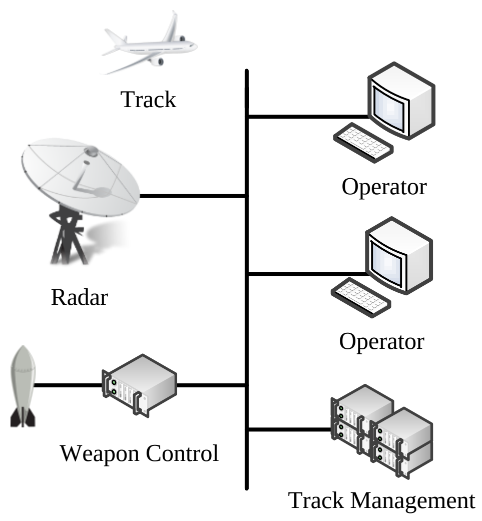


Figure 1.3: Entity Overview of the T-Core Track Management System

naval and ground command and control systems provided by Terma. The configuration of T-Core used for the case study in this thesis is illustrated in figure 1.3.

This case study focuses on the Track Management (TM) component of the T-Core framework. The TM component creates a full picture of the surroundings based on track information received from radars such as aircrafts or missiles. The information received from different sources is processed and distributed to other parts of the system. Operators will for instance be able to monitor the current situation and assist the TM component in making decisions about tracks. Finally, in this case study, the T-Core configuration also contains a Weapon Control component which is able to engage tracks based on information received from the TM component.

Some of Terma's customers are interested in having hard real-time guarantees in certain components of the T-Core framework. Therefore Terma wishes to investigate possibilities for combining non real-time components with hard real-time components on the same node while still providing timing guarantees. This is demonstrated in this case study by constraining the Weapon Control component with hard real-time requirements while the TM component is deployed on the same node performing soft real-time tasks.

1.6. Reading Guide

Throughout this thesis a few special notations and conventions are used to improve the readability:

References All external references (books, articles, technical reports etc.) are placed in brackets, labeled with the surname of the author followed by the year of publication, e.g. [Baker06]. If the referenced work has multiple authors the reference will use the surname of the first listed author followed by an ampersand before the year of publication, e.g. [Bacon&03].

Emphasis Words, sentences or names with special relevance are emphasized by the use of an *italic* typeface.

Special Terms Special technical terms not explained within the text are marked with tau and explained in the terminology found in appendix A, e.g. $^{\tau}$*Worst-Case-Execution-Time*.

Keywords Programming keywords (VDM, Java, etc.) within the text, as well as in diagrams and illustrations, are written in **`teletext and boldface`**, where instructions and class names are written in `plain teletext`.

Listings Model and code listings are marked in special styles, where VDM model is listed as shown in listing 1.1 and Java code is listed as shown in listing 1.2.

```
public ActivatePedal : () ==> ()
ActivatePedal() == Car`controller.HandleSpeedPedal();
```

Listing 1.1: Example of a VDM code block

```
public void handleAsyncEvent()  {
engine.setSpeed(Speed.Max); }
```

Listing 1.2: Example of a Java code block

If not specified otherwise, the term *"standard Java"* refers to the Oracle Java Platform, Standard Edition executing within the Oracle HotSpot Runtime Environment 1.6. All test are carried out on a set of *Lenovo Thinkpad X300* PC's with an Intel Core 2 Duo and two gigabytes of RAM. The operating system is *Fedora* 17 with the *Linux RT PREEMPT* patch. Test results are summarized in tables, often indicating jitter values as maximum, average or standard deviation which are based on datasets with absolute values.

1.6.1 Structure

This thesis is organized into eight chapters and five appendices. Figure 1.4 (page 9) provides a graphical overview of the thesis structure, where chapters are connected with arrows indicating the suggested reading order. Dotted lines indicate a loose connection, such as a reference to an

appendix or a chapter which can be skipped if the reader possesses prior knowledge about the presented subject.
Chapter 2 and 3 provides the theoretical foundation and introduces relevant terms, concepts and technologies.

Chapter 2 This chapter describes the characteristics of real-time systems and why Java is unable to achieve real-time guarantees. This chapter can be skipped if the reader holds prior knowledge within this area.

Chapter 3 This chapter presents two specifications extending the Java language, the RTSJ and the SCJ. Both specifications are analyzed and related to the topics described in chapter 2.

The following chapters describe the four steps of the TJARP method including the motivation for each step and concrete examples of the steps applied to the simple Car Controller case study.

Chapter 4 This chapter presents the first step of the TJARP method. Focus is on definition of real-time requirements with emphasis on describing events with temporal constraints.

Chapter 5 This chapter presents the second step of the TJARPmethod, and introduces the formal modeling language VDM-RT as an important tool for analyzing real-time requirements.

Chapter 6 This chapter presents the third and fourth step of the TJARP method, and provides the reader with a detailed overview of some of the implementations available for real-time using Java. Several techniques are presented and technical solutions are tested and compared.

In **chapter 7** the TJARP method is evaluated by applying the steps explained in chapters 4-6 to a complex industrial case study. Finally **chapter 8** provides a conclusion of the work described within this thesis, where the achieved results are analyzed and discussed as well as potential additions for future works are included.
Five appendices are referenced within this thesis.

Appendix A This appendix presents the terminology used within this thesis.

Appendix B This appendix contains detailed information of the Car Controller and Terma T-Core case studies.

Appendix C This appendix contains a technical descriptions of the additions to the Overture RT Log Viewer.

Appendix D This appendix contains additional details of the analysis and benchmarks applied in chapter 6.

Appendix E This appendix describes some of the possibilities for introducing real-time performance across distributed nodes.

Additionally all implementation specific files, such as VDM-RT and Java source code, is located on the attached CD.

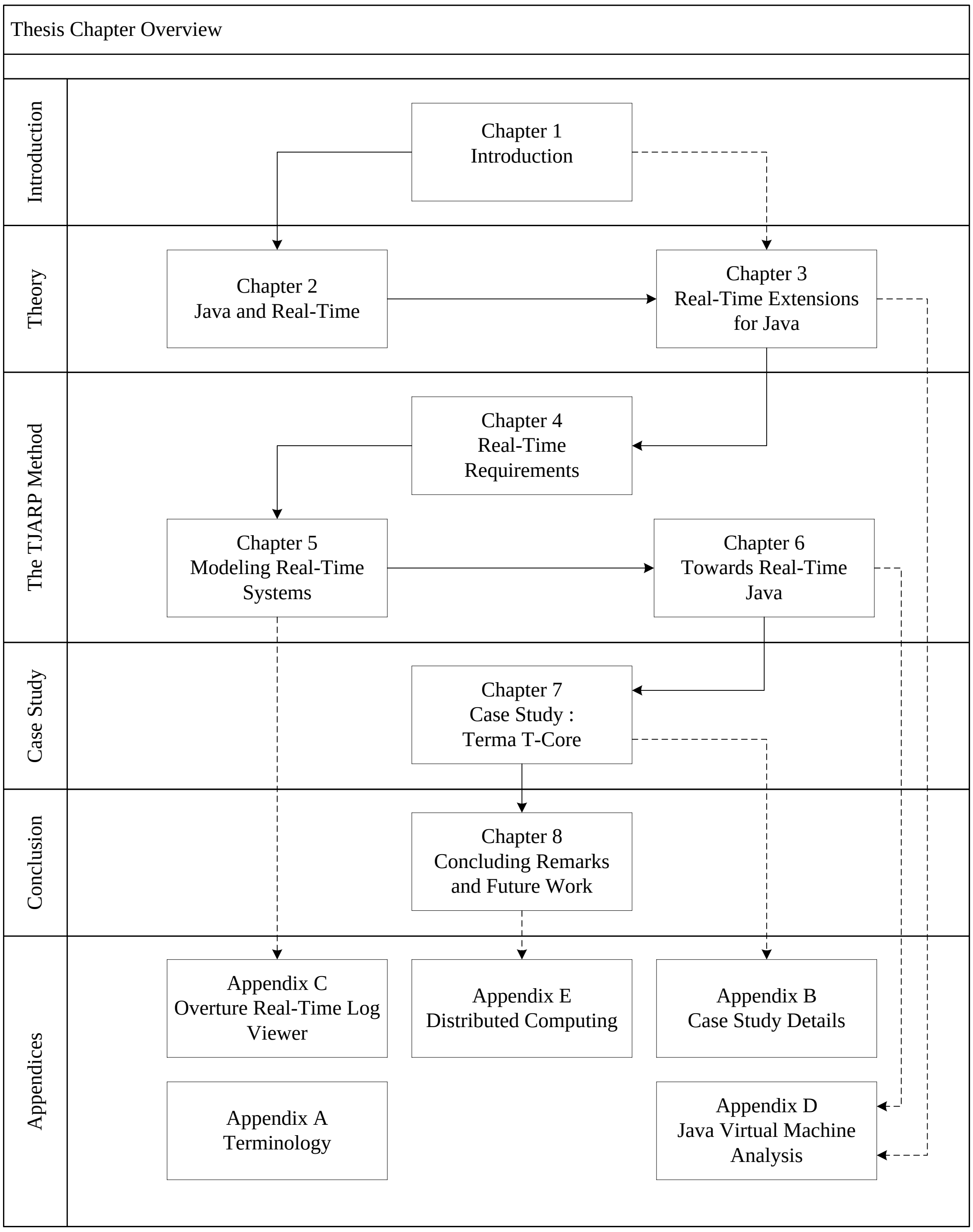


Figure 1.4: Overview of the thesis structure

Chapter 2

Java and Real-Time

This chapter describes important topics which need to be considered in order to achieve real-time performance in software systems. It is described how these topics are handled in traditional real-time systems as well as in standard Java. The points made through this chapter will be emphasized by use of the Car Controller example presented in section 1.5.1. The theory introduced in this chapter also serves as background knowledge for the rest of this thesis. Therefore readers who are already experienced within real-time systems and standard Java may want to skip this chapter or parts of it.

2.1. Introduction

In order to achieve real-time performance in computer systems, a number of important topics need to be considered. These topics include *scheduling*, *synchronization* and *memory management*. Each of these has great influence on timeliness and predictability which are required properties of real-time systems. For the same reason these topics has been subject to research for many years and solutions to common problems within each of them exist. Some of these common problems and their solutions will be touched upon in this chapter. The solutions however often require low-level programming languages and special operating system features. Therefore in order to increase productivity, portability, etc. the idea of achieving real-time performance using Java is tempting. However, it is not possible to achieve real-time performance using standard Java and this chapter seeks to investigate the challenges preventing this.

The three topics scheduling, synchronization and memory management are described in sections of their own, from section 2.2 to section 2.4. Each section is divided into two sub-sections. The first sub-section describes how the topic is traditionally approached in real-time systems, covering the common challenges and solutions. The second sub-section describes challenges preventing standard Java from achieving real-time performance, within the particular topic, which are outlined and exemplified using the Car Controller example (See section 1.5.1). Finally section 2.5 discusses the theory presented throughout this chapter.

2.2. Scheduling

In computing the process of determining which threads gets to execute at what times is called scheduling. If several threads, sharing the same processing unit, are eligible for execution at the same time, the scheduler must be able to choose which thread to execute first. Many algorithms for making such choices exist, each addressing different properties which could be desirable for a particular system. Examples of scheduling algorithms are:

First-in-first-out: This algorithm queues threads according to the time they got eligible for execution. When a thread gets to run it can run until it completes, blocks or gives up the processor itself. The benefit of this algorithm is simplicity. The drawback is performance if threads choose to occupy the processor for long periods of time.

Round Robin: This scheduler focuses on fairness among threads. This is done by giving an equal amount of execution time (time slice) to all threads in a fixed order. If a thread do not finish within its time slice the scheduler $^{\tau}$preempts it from the processor in favor of the next thread. An advantage of this approach is the fairness which avoids $^{\tau}$starvation of threads. However, the lack of prioritization between threads will potentially cause less important threads to delay more important threads.

Priority scheduling: In this approach each thread is assigned a priority. The scheduler executes the highest priority thread which is also eligible for execution. A benefit is that the latency of high priority threads is minimized i.e. the time from the thread gets eligible for execution until it completes. A drawback is the possibility of high priority threads starving low priority threads.

Below section 2.2.1 will describe how scheduling is approached in traditional real-time systems and section 2.2.2 will describe the approach taken in standard Java.

2.2.1 Real-Time Systems

In real-time systems timeliness and predictability are properties of great importance which the scheduler needs to facilitate. The scheduler must help guarantee that both periodic and aperiodic threads are able to meet their deadlines under all specified circumstances. In order to achieve this, the real-time schedulers are usually supported by various analysis techniques applied at design time. Section 2.2.1.1 and 2.2.1.2 gives examples of two common real-time schedulers and their corresponding analysis techniques. Section 2.2.1.3 briefly describes the challenges faced when doing real-time scheduling in a multiprocessor environment.

2.2.1.1 Cyclic Scheduling

A simple technique for ensuring that all deadlines are met has resemblance to the round robin scheduler and is called cyclic scheduling [Baker&88]. This scheduling algorithm depends on an analysis of all threads at design time, in order to determine their execution times. These times are then used to create a static schedule where all threads are given fixed periodic time slices, which ensure their timeliness. Additional time slices can be allocated in the schedule in order to serve aperiodic threads, which are triggered by external events instead of progression in time. Advantages of this solution are its simplicity and predictability. Ideally the scheduler does not have to make any decisions at runtime as everything is planned a priori. If the $^{\tau}$*Worst-Case-Execution-Times* of all threads are well-known then the resulting static schedule will be able to guarantee that all deadlines are met. The main issue of this solution is to come up with a static schedule satisfying all deadlines given a set of threads, their execution times, and deadlines. This is known to be an $^{\tau}$NP-hard problem and there is no guarantee that such a static schedule exists [Mok83]. Another drawback is the inflexibility of this approach e.g. whenever a new thread is added or an existing thread changes execution time a new static schedule must be produced.

2.2.1.2 Scheduling using Rate-Monotonic Analysis

A highly influential principle for scheduling in real-time systems is the *Rate-Monotonic Analysis* or just rate-monotonic scheduling [Liu&73]. This approach is also able to ensure that all deadlines are met but do not rely on a static schedule in order to achieve this. Instead all threads are assigned fixed priorities using a method called the *rate-monotonic priority assignment*. Intuitively priorities would be assigned to threads according to their perceived importance. However, this method assigns higher priorities to threads with higher periodic frequencies. Using this method a given set of threads is $^{\tau}$schedulable if they satisfy equation 2.1.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{2.1}$$

Where n is the number of threads while C_i and T_i are the execution time and the period of thread i respectively. The $^{\tau}$*processor utilization* of the entire set of threads is given by the left side of equation 2.1, while the right side is the least upper bound of processor utilization in order to guarantee schedulability. If the set of threads does not satisfy the equation then the threads may still be schedulable but further analysis is required in order to ensure this [Lehoczky&89]. The Rate-Monotonic Analysis is done at design time, at runtime the scheduling is done by a fixed-priority preemptive scheduler. This scheduler ensures that the currently executing thread is always the one with the highest priority of the threads eligible for execution. A thread is preempted if another thread with higher priority gets eligible for execution. A main advantage of rate-monotonic scheduling is that whether the set of threads are schedulable or not depends on the entire sets utilization of the processor. This means that periodic threads can be executed independently while deadlines are still guaranteed. Also if a new thread is added to the system only a recalculation of the sets processor utilization is required to check whether it is still schedulable. The weakness of this approach is that it suffers from some rather restrictive assumptions e.g. threads must be completely independent of each other and have constant execution times. Rate-monotonic scheduling has however served as a starting point for more research which has been able to relax these assumptions [Sha&86, Sha&90, Rajkumar89, Rajkumar91].

2.2.1.3 Scheduling in Multiprocessor Systems

The real-time scheduling techniques described so far assumes that a single processor is shared among all threads. However, as multiprocessor systems have become the norm also within real-time systems, the scheduling algorithms and analyses must also be able to support additional processors. Further complexity is added to scheduling when working in a multiprocessor environment. This is because, in addition to deciding which threads gets to execute when, the scheduler must also consider which processor to use for the execution.

Unfortunately the research within multiprocessor scheduling has yet to catch up with that of uniprocessor scheduling [Baker06]. Therefore, no optimal and agreed upon solution currently exists to real-time scheduling in multiprocessor environment. However, in general three different approaches are used by multiprocessor schedulers for choosing where to execute a given thread [Higuera-Toledano&12, Davis&11]:

Fully Partitioned Scheduling: This approach assigns each thread to a single processor where it will always be executed. The main advantage of this solution is that after assigning threads to processors, the well-known uniprocessor scheduling algorithms and analyses can be applied per processor. A drawback is that a processors idle time cannot be utilized by threads on other processors which may be busy.

Global Scheduling: This approach allows all threads to execute on all processors. An advantage of this approach is that idle time on one processor can be utilized by all threads in the system. The drawback is the added complexity and the overhead introduced when moving threads between processors.

Clustered Scheduling: This approach is a hybrid of the two techniques just mentioned. Each thread is assigned to a subset of all processors called a cluster. Within each cluster the threads are scheduled using global scheduling. This can be an advantage if groups of processors share the same local memory, thus the overhead of moving threads are minimized.

2.2.2 Standard Java

Standard Java has a single thread class which can be used for implementing multi-threaded applications. A standard Java thread can have one of 10 different priority levels. However the original Java Language Specification (JLS) [Gosling&96] does not specify any algorithm for scheduling these threads. The closest the specification gets to mentioning scheduling is the following vaguely formulated paragraph taken from section 17.12 describing threads:

> When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

More recent Java Language Specifications say even less on the subject of scheduling. The reasons for this are probably the major design goals of making Java independent of hardware and Operating Systems (OS). This implies that the scheduling mechanisms actually used in standard Java are decided by the vendors of Java Virtual Machines (JVM). However it seems to be common practice among JVM implementations to create a 1:1 mapping between Java threads and native threads of the underlying OS [Oracle08]. The 10 different priority levels are typically mapped to priorities of native threads and scheduling can then be done by the underlying OS. This is also why standard Java does not make any particular considerations about scheduling when working in a multiprocessor environment as this responsibility is left to the OS scheduler.

The Car Controller Example

In order to further investigate the scheduling mechanism in standard Java, the Car Controller example has been used. A test scenario, where the brake pedal is activated around the same time as the navigation software is updating the car display image, has been executed several times. The results show that standard Java is indeed unpredictable in its scheduling of threads and does not respect thread priorities. Sometimes the brakes are activated immediately, other times the brake activation is delayed due to the navigation software. The result of a test run which yielded a particular bad result has been illustrated in figure 2.1.

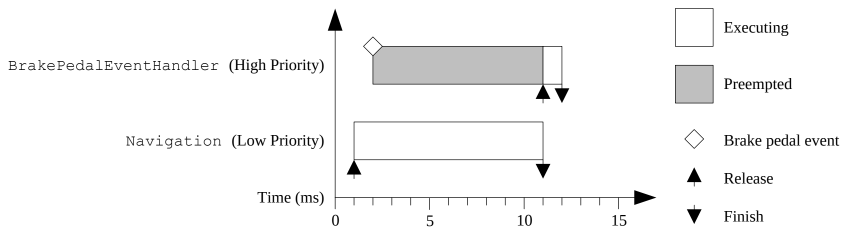

BrakePedalEventHandler (High Priority)
Navigation (Low Priority)
Time (ms)
0
5
10
15
Executing
Preempted
Brake pedal event
Release
Finish

Figure 2.1: Scheduling in standard Java

The `Navigation` thread starts updating the display at time t=1. The brake pedal is activated at time t=2 and makes the `BrakePedalEventHandler` thread eligible for execution. Although the `BrakePedalEventHandler` thread has a higher priority than the `Navigation` thread, the `Navigation` thread is allowed to preempt the `BrakePedalEventHandler` thread for 9 ms, before it is allowed to execute.

From a real-time system's perspective it is problematic that thread priorities are not strictly adhered to. Furthermore, as the scheduling may or may not be done by the OS, it would take a thorough investigation of the JVM implementation, and the underlying OS, in order to reason about how the threads of an application would be scheduled. Clearly more strict semantics and guidelines are needed in order to obtain real-time scheduling and ensuring deadlines.

2.3. Synchronization

Synchronization is used to coordinate concurrent threads, sharing a common resource, in order to avoid undesirable $^{\mathcal{T}}$*race conditions*. If a memory area is shared then synchronization helps ensure data integrity, or if a hardware component is shared then synchronization can provide exclusive access to this component. Code blocks accessing shared resources, and hence requiring synchronization, are called critical sections. In order to protect critical sections different locking primitives are used e.g. $^{\mathcal{T}}$*semaphores* [Dijkstra68], $^{\mathcal{T}}$*mutexes* [Dijkstra65], $^{\mathcal{T}}$*monitors* [Hoare74] etc. They all help ensure that critical sections are only entered by a limited number of threads. Usually only one thread is allowed inside a critical section at a time. This is called *mutual exclusion*. If more threads attempt to enter the critical section these will be blocked until the critical section is again unlocked.

A common problem with thread synchronization is *unbounded priority inversion*, which is a situation where a high priority thread is blocked by a low priority thread for an unknown amount of time [Sha&90]. This can occur if a resource is shared between a low priority thread and a high priority thread. If the low priority thread already has acquired the resource and the high priority thread then tries to do the same, the high priority thread will have to wait until the low priority thread releases the resource. The waiting time endured by the high priority thread can be prolonged by medium priority threads which will preempt the low priority thread while it is still holding the lock for the shared resource.

Another problem when doing thread synchronization is *deadlocks* [Coffman&71]. These can occur if two or more threads try to gain access to a critical section while already executing within one. Then for instance, if two threads each try to access the critical section held by the other thread they will wait forever and a deadlock has occurred.

Section 2.3.1 describes how the synchronization challenges are handled in traditional real-time systems, while section 2.3.2 describes how thread synchronization is achieved in standard Java.

2.3.1 Real-Time Systems

The problem with unbounded priority inversion outlined above is particular dangerous in real-time systems. The non-deterministic delay incurred on the high priority thread will make it impossible to predict if its deadline will be met. Here the most common solutions to the problem used in real-time systems are described:

Non-preemptive Critical Sections Protocol [Mok83]: This is a simple approach where a thread cannot be preempted as soon as it is executing within a critical section. This ensures that the thread will finish the critical section as fast as possible. Also deadlocks are prevented if this protocol is applied in a uniprocessor environment. The drawback is that other higher priority threads may be delayed unnecessarily even though they do not share the same critical section.

Priority Inheritance Protocol [Sha&90]: With this approach a low priority thread holding the lock for a critical section can have its priority raised temporarily. This happens when a higher priority thread tries to access the same critical section. Then the low priority thread inherits the priority level of the higher priority thread until it finishes the critical section. This solution is not perfect as the high priority thread will still have to wait for the low priority thread to finish the critical section. However using this approach there is an upper limit to the amount of time the high priority thread can be blocked by lower priority threads.

Priority Ceiling Protocol [Sha&90]: This approach assigns a priority ceiling to each critical section which equals the priority of the highest priority thread accessing the particular section. When a thread tries to enter a critical section it is checked whether the thread has a higher priority than the priority ceilings of all critical sections currently executed. If this is the case then the thread is allowed to execute the critical section. Otherwise the thread is blocked and other threads executing critical sections inherit the priority of the blocked thread. This solution is proved to minimize the upper bound on blocking time endured by high priority threads caused by lower priority threads. Additionally this approach also solves the deadlock problem.

2.3.2 Standard Java

In standard Java, synchronization is done by use of the **`synchronized`** keyword. Methods within a class declared with the **`synchronized`** keyword are executed with mutual exclusion. This functionality makes Java classes act like monitors i.e. each object instance has an associated mutex which must be obtained before the synchronized methods can be executed. Another way of using the **`synchronized`** keyword is as a statement. This is exemplified in listing 2.1.

```
	synchronized(obj){
		Body of statements...
	}
```

Listing 2.1: The **`synchronized`** statement

Using the **synchronized** statement, ensures that the body of the statement is not executed until the mutex associated with the object instance `obj` has been obtained. When the body is exited the mutex is released again.
Standard Java does not explicitly handle the priority inversion problem. As described in section 2.2.2, JVM implementations usually rely on the underlying OS for enforcing thread priorities and scheduling. Therefore mechanisms for avoiding priority inversion and deadlocks rely both on the mapping done by the JVM to the OS and the OS itself.

The Car Controller Example

The Car Controller example has been used to check if priority inversion is possible in standard Java. The shared resource in the test case is the `Engine` which must be accessed by both the `CruiseController` thread and the `GasPedalEventHandler` thread in order to change speed. The priority of the `CruiseController` thread has been lowered in this particular test in order to provoke the situation where priority inversion can arise, in all other tests the thread has a high priority. Figure 2.2 shows the result of a test scenario where the low priority `Cruise-Controller` thread gains access to the engine just before the high priority `GasPedalEvent-Handler` thread is scheduled.

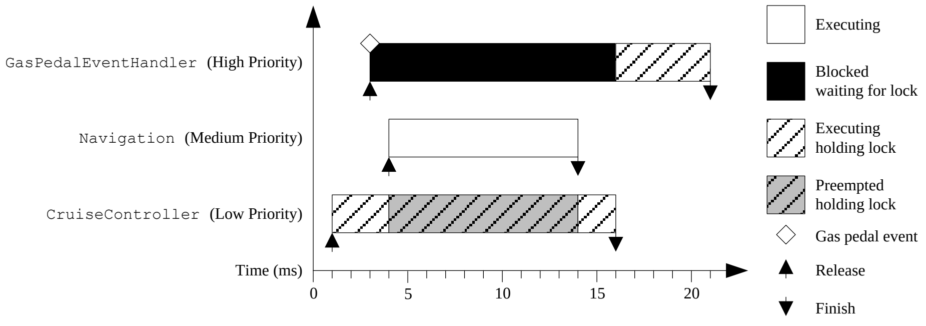


Figure 2.2: Synchronization in standard Java

The result is that the `GasPedalEventHandler` thread is blocked until the `CruiseCon-troller` thread finishes the speed change and releases the lock on the engine. The `GasPedal-EventHandler` thread is further delayed because the medium priority `Navigation` thread preempts the `CruiseController` while it is holding the lock.
This shows that the unbounded priority inversion problem indeed exists in the standard Java implementation. In order to obtain real-time performance this problem must be dealt with, possibly using the techniques mentioned in section 2.3.1.

2.4. Memory Management

Memory access is essential for most computer applications in order to store instructions and data e.g. state information. This must be supported by the environment in which the program executes to ensure safe and correct behavior. Most programming languages provide the developer with access to both static (allocated at compile-time) and dynamic memory (allocated at run-time). The latter is by far the most complex and requires the developer, the environment, or both, to handle used *and* available memory while executing the application. The term *memory management* cov-

ers several different aspects of dynamic memory within computer science, ranging from operating system level to application level. This section will discuss some of the challenges that standard Java applications, and the corresponding JVM, encounters when dealing with dynamic memory management.
Section 2.4.1 will describe how memory management is handled in traditional real-time systems. Section 2.4.2 describes how memory management is handled in standard Java, and why this is not suitable for real-time systems.

2.4.1 Real-Time Systems

In most low-level programming languages such as C and C++, the task of managing dynamic memory is explicit and left to the programmer. This includes manually allocating dynamic memory and deallocating when it is no longer needed. In complex system this can be a difficult task and is often prone to errors, leading to unexpected program behavior or even crashes. Despite the challenges, these languages are often the preferred solution in embedded and real-time systems [Hertz&05]. They allow the developer to be 'close to the metal', and provide a deterministic behavior where the developer is able to predict exactly the time it takes for the system to make the requested memory available. However when using explicit memory management the developer has to manually deal with the following challenges:

Dangling pointers: Also known as dangling references or wild pointers. This problem arises when an object is deallocated while one or more objects still hold a reference/pointer to it. If one of these objects tries to dereference the now deallocated object, it will result in unpredictable behavior.

Memory leak: This occurs when an application continuously consumes memory through allocation, but does not deallocate unreferenced objects. Memory leaks may potentially decrease the performance of the system or in worst case result in the system running out of available memory.

Fragmentation: Memory fragmentation is due to dynamic memory being allocated in separate memory segments (chunks) over time. The developer must consider situations where N bytes of memory is needed but all available memory segments are of size less than N, even if the total amount of available memory is greater than N. Fragmentation may result in the system running out of available memory even though the total size of free memory segments is sufficient. This can be solved by linking the fragmented memory segments, however this will degrade the performance of operations made on the linked memory area.

Several methods and design principles have been proposed in the literature to help developers through some of the above mentioned challenges [Crocker10], e.g. by never releasing unused memory or adding object reference counting. These solutions however, introduce extra complexity and new challenges, such as insufficient memory.

2.4.2 Standard Java

High-level programming languages such as Java try to relieve the programmer from these complex tasks by providing automatic memory management. This includes keeping track of all allocated memory and recognizing when segments are no longer needed (referenced) by the active program and then deallocating it. This mechanism is referred to as *Garbage Collection* and has been the subject of intensive research within computer science for more than fifty years [McCarthy60].

Garbage collection is an essential feature of modern high-level object-oriented programming languages.
The benefits of garbage collection are indisputable, but for years it has been an ongoing discussion in the software community whether explicit memory management or automatic memory management can achieve the highest performance. Garbage collection introduces extra overhead both in space and time [Hertz&04, Hertz&05]. However, using an algorithm tailored towards the application needs and behavior, it is possible to achieve both the benefits of automatic memory management and high performance. However, most garbage collection algorithms suffer from lack of determinism. As dynamic memory in Java is handled by the *Garbage Collector* (GC) the two terms are used interchangeably in the following.
Today many different types of garbage collection algorithms are available for Java applications [Venners99] each with its own characteristics designed for different situations and types of applications. Below sections 2.4.2.1 and 2.4.2.2 will discuss some of the basic principles employed in most garbage collection algorithms.

2.4.2.1 Detecting Garbage

A critical feature of all GCs is the process of determining which objects in memory can be considered garbage, and which objects should remain in memory. Objects that are reachable by the active program is said to be *live*, and is often determined by examining a distinguished set of objects, known as root objects. The definition of root objects in JVMs is vendor specific, but typical root objects are reachable through references located on the call stack, such as class variables and global variables. By traversing references from root objects the GC is able to determine the $^{\tau}$*transitive closure* of the reference-graph which includes all reachable objects. This type of strategy is referred to as *tracing collectors*. Another strategy also employed in some garbage collectors is *reference counting*, where access to objects in memory, is provided by an extra layer of abstraction [Dawson08, Levanoni&06]. This layer holds a counter for each object which is incremented every time a reference is created and decremented every time a reference is destroyed or set to another object and when the counter reaches zero the referenced objects may be deallocated. This strategy does however introduce another challenge: how to deal with the case where two objects reference each other, but are not reachable from any other object? Then their reference count will remain at one, but they can legally be considered garbage – a situation known as *cyclic references*. This is not a problem for the tracing collectors, but reference counting collectors must adapt another approach such as the one used by the train algorithm [Venners99] in order to deal with cyclic references.

2.4.2.2 Collecting Garbage

Observations of memory behavior in high-level programming languages such as Java has introduced *the weak generational hypothesis* [Lieberman&83] which states that most allocated objects are not alive for long, and there are few references from older to younger objects. This hypothesis is exploited by *generational collectors* which tries to address the inefficiency of collecting the entire heap in every run, by separating the heap into generations. Younger objects, which are more likely to be collected early, is placed in one memory area (often named *eden space*, *nursery* or *young space*) which is collected more often. When an object has survived a number of collections it may be promoted to another memory area (often named *tenured* or *old space*) which due to the hypothesis does not require collections as often. This strategy is used in several virtual machines, such as the Oracle HotSpot JVM [Sun06]. A major benefit of separating the memory into different areas is the ability to collect an entire area in one sweep, which also allows for fast allocation.

This is possible since memory is allocated in contiguous blocks, and new areas can be allocated by a simple *bump-the-pointer* technique. Where the end of the previously allocated object in the young space is saved, and used as a starting point for the next allocated object.
Garbage collection requires access to the heap while the application is active, therefore synchronization of memory areas is needed. This is often done with a *stop-the-world* approach where execution of the application is completely suspended while collection is in progress. This is tolerable in many applications without real-time deadlines, however in critical systems this is unacceptable.
When executing in a multiprocessor environment it is possible to split garbage collection into subtasks running in parallel, a collection strategy referred to as a *parallel collection* [Sun06]. This minimizes the suspension time endured by the application during each garbage collection but does not eliminate it.
Timing is not the only consideration when choosing a garbage collection strategy. Many algorithms introduce extra scheduling and memory overhead, therefore the choice of garbage collector is always a tradeoff between several performance metrics such as throughput, block time, frequency etc. [Sun06].

The Car Controller Example

To investigate the impact of garbage collection on the timeliness of threads in standard Java, the Car Controller example has been used. The `CruiseController` thread has a high priority, and its job is to monitor the speed of the car, with a period of 50 ms, and alter it if needed. Concurrently the lower priority `Navigation` thread updates the car display with a period of 33 ms. This operation generates a significantly amount of garbage and triggers invocation of the garbage collector.
To ensure that the jitter observed during the test is indeed caused by garbage collection, another reference-test has been included. The only difference in this additional reference-test is that the `Navigation` thread does not generate any garbage.
The results of both tests can be seen in figure 2.3 and table 2.1. The table indicates the jitter values for the `CruiseController` thread which are the deviation from the expected period (50 ms).

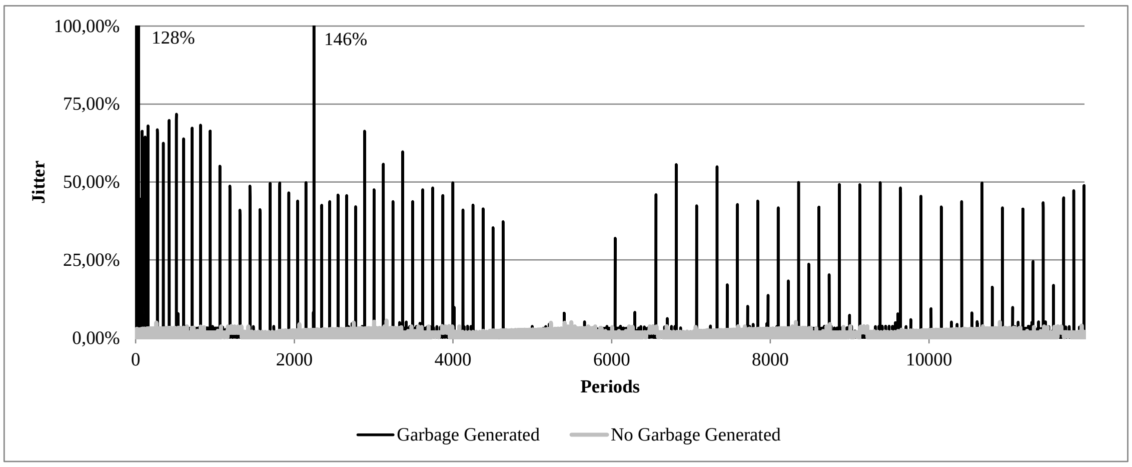


Figure 2.3: Jitter experienced by the `CruiseController` thread

When no garbage is generated there is a maximum jitter of 5,54% from the 50 ms thread period. This may suffice as soft real-time in some systems. But when the garbage generation is enabled

Thread Period (ms):	50,00	**Maximum**	**Average**	**Std. Deviation**
Garbage Generated:	**Jitter (ms):**	73,26	0,76	2,81
	Jitter (%):	146,52%	1,52%	5,61%
No Garbage Generated:	**Jitter (ms):**	2,77	0,47	0,45
	Jitter (%):	5,54%	0,95%	0,91%

Table 2.1: Jitter statistics for the `CruiseController` thread

the predictability of the `CruiseController` thread deteriorates significantly and the maximum deviation gets as high 146,52%. The standard deviation when garbage is generated is also significantly higher than without garbage generation, which illustrates the degree of unpredictability caused by the garbage collector.

2.5. Discussion

The theory and test results presented throughout this chapter, makes it clear that standard Java is unable to provide any real-time guarantees. However, standard Java does not try to achieve real-time performance either, instead the JLS facilitates the Java mantra “Write once, run everywhere”. This leaves the JLS ambiguous and unclear on several points which are important to real-time performance e.g. scheduling of threads.
Even though standard Java seeks to be independent of the underlying OS, it cannot be guaranteed that a Java application will exhibit the same behavior when executed on different OS’s. This is due to differences in thread models of the underlying OS’s. Therefore, developers of multithreaded applications are discouraged to let the correctness of their application depend on thread priorities, in order to ensure portability [Bloch01, Sun03]. In other words the JLS provides thread priorities, but it is not possible to depend on them.
One of the challenges preventing real-time behavior presented through this chapter is that of supporting dynamic memory management, which is one of the dominating obstacles. In general two approaches are used to accommodate the problem of combining real-time performance and garbage collection. The first approach is to introduce memory areas which will not be garbage collected and hence the threads using this area will not be suspended by the GC. This however complicates the development of applications as these new memory areas must be taken into account. The second approach is to use a real-time garbage collection algorithm. These algorithms exhibit a highly deterministic behavior and can be preempted by application threads. Such an algorithm however adds to the complexity of the application and complicates the task of reasoning about real-time guarantees of an application.

Chapter 3

Real-Time Extensions for Java

This chapter introduces two official extensions proposed for improving the temporal behavior of Java. These are related to the topics described in chapter 2. The theory presented serves a basis for evaluation and discussion in chapters 6 to 8. Readers who are already familiar with these efforts and specifications may want to skip this chapter or parts of it.

3.1. Introduction

In attempt to standardize the approach for achieving real-time performance using the Java programming language, two specifications have been proposed. One is called the *Real-Time Specification for Java* (RTSJ), and another is called *Safety Critical Java* (SCJ) and is based upon the RTSJ. Their individual goals and solutions to the challenges preventing standard Java from achiving real-time performance (see chapter 2), will be presented in the following sections 3.2 and 3.3. Finally section 3.4 will discuss the advantages and disadvantages of the language extensions.

3.2. The Real-Time Specification for Java

Work on the RTSJ began in the late nineties, under the $^{\tau}$*Java Community Process* (JCP) after an initial $^{\tau}$*Java Specification Request* (JSR) had been filed by IBM [Bollella&00]. The request was accepted under the name JSR-001 and an expert group was put together consisting of people from both industry and academia [JSR001]. Their job was to produce the specification and in 2002 the first edition, RTSJ 1.0, was accepted by the JCP. The latest version is 1.0.2 and version 1.1 is currently under development.
The goal of the RTSJ is to extend the Java language to provide an *Application Programming Interface* (API) which facilitates the creation of real-time applications in Java. The expert group behind the RTSJ identified seven areas which needed to be enhanced, in order to achieve real-time performance in Java. Three of the areas equal those presented in chapter 2: *Scheduling*, *Synchronization* and *Memory Management*. The remaining four areas are: *Asynchronous Event Handling*, *Asynchronous Transfer of Control*, *Asynchronous Thread Termination* and *Physical Memory Access*. However, it is beyond the scope of this chapter to describe the RTSJ in its entirety. Furthermore, the four additional areas are either sub-areas or they do not directly influence real-time performance but are desirable features in many real-time systems. Therefore sections 3.2.1

to 3.2.3 describes how the RTSJ deals with the problems in the three areas presented in chapter 2.

3.2.1 Scheduling

The RTSJ defines two additional thread implementations besides the standard Java thread, which are called `RealTimeThread` (RT) and `NoHeapRealTimeThread` (NHRT). As the name suggests the NHRT cannot allocate memory on the heap. This ensures that threads of this type are never preempted by garbage collection, thus they are not affected by the temporal non-determinism introduced by garbage collection. The idea of the additional thread types is that it should be possible to have *non-real-time*, *soft real-time* and *hard real-time* tasks within the same Java Virtual Machine (JVM). Hence, the standard Java thread should be used for non-real-time tasks, while the RT should be used for soft real-time tasks and the NHRT for hard real-time tasks.
The RTSJ recognizes the need for a scheduling algorithm in order to achieve real-time performance. Therefore it defines a base scheduler which must be available in all RTSJ compliant JVM implementations. This scheduler is a fixed priority preemptive scheduler like the one used for rate-monotonic scheduling (see section 2.2.1.2). The scheduler works on threads through the `Schedulable` interface which is implemented by the aforementioned real-time threads. At least 28 different priority levels must be available for the real-time threads, including the 10 levels defined in standard Java. In order to support aperiodic tasks the RTSJ defines an `AsyncEventHandler` (AEH) which also implements the `Schedulable` interface. This means that aperiodic tasks are handled by the same scheduler and through the same interface as the real-time threads. The tasks implementing the `Schedulable` interface are, by the RTSJ referred to as *schedulable objects*. The relationship between the schedulable objects and the standard Java `Thread` class can be seen in figure 3.1.

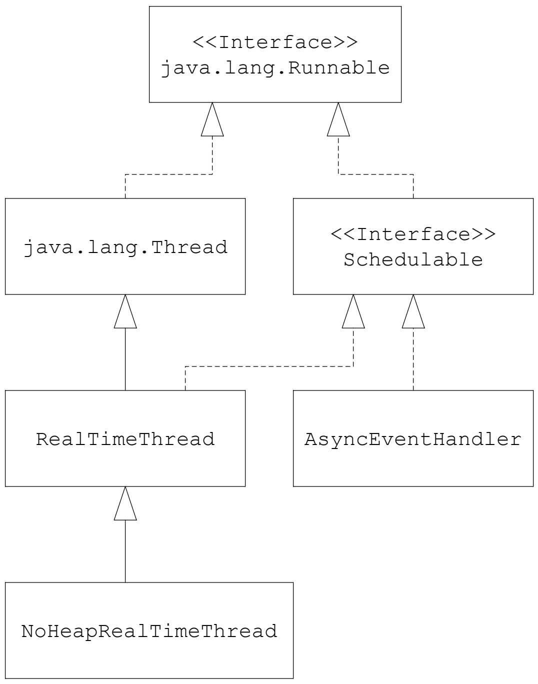


Figure 3.1: Relationship between standard Java threads and RTSJ threads

In addition to the base scheduler the RTSJ also provides the means for JVM vendors to implement their own scheduling algorithms. This is done by defining standard interfaces like the

`Schedulable` interface already mentioned, but also standard parameters to these schedulable objects. Parameters like `ReleaseParameters` and `SchedulingParameters` are able to specify the priority, period, deadlines, etc. for a thread. These generic parameters can then be used in the custom scheduling algorithms.
Until recently neither standard Java nor the RTSJ had addressed the issues of multiprocessor scheduling. However, in version 1.1 the RTSJ recognizes the existence of multiprocessor systems [Higuera-Toledano&12]. As there is no standard approach to multiprocessor schedulability analysis, the RTSJ 1.1 will not provide a base scheduler for multiprocessors, like the fixed priority preemptive scheduler is provided for uniprocessors. However, the API of the RTSJ is extended in order to support multiple processors by allowing developers to allocate threads to specific processors. Hence, facilitating the implementation of global or partitioned scheduling algorithms (see section 2.2.1.3). The main motivation behind supporting multiple processors in the RTSJ is the potential performance increase and the possibility of isolating real-time threads from non-real-time threads on separate processors.

3.2.2 Synchronization

The RTSJ requires that the priority inheritance protocol (see section 2.3.1), is implemented by default, in order to avoid the problem of unbounded priority inversion, when using the **`synchronized`** Java keyword. The protocol must be applied to both real-time threads and regular Java threads when doing synchronization. Optionally the RTSJ allows for implementation of the priority ceiling protocol and other mechanisms that JVM vendors may find appropriate. In addition to the default priority avoidance protocol the RTSJ also makes it possible to choose which protocol to use for a particular monitor.
It is allowed for real-time threads to synchronize with non-real-time threads in the RTSJ. However, a problem arises when a NHRT is synchronizing with a thread which can be preempted by garbage collection. Although the regular Java thread or RT which the NHRT is synchronizing with, can inherit the NHRT's higher priority it cannot be allowed to preempt garbage collection, as this could potentially cause the application to run out of memory. Therefore the NHRT can incur an unpredictable delay from garbage collection when synchronizing with other thread types. In order to solve this issue the RTSJ specifies a wait-free queue which allows communication between the NHRT and the other thread types. Calls to the enqueue and dequeue primitives used by the NHRT will always return immediately. However, this timing predictability comes at a price, as data can be lost if the wait-free queue gets full. Therefore the size of the queue must be chosen carefully.

3.2.3 Memory Management

The RTSJ specifies two additional memory areas besides the traditional garbage collected heap used in standard Java. These are called *Immortal Memory* and *Scoped Memory*. As mentioned in section 3.2.1 the NHRT cannot make use of heap memory and therefore in order to allocate memory it must use these two alternatives. Neither of the two memory areas can be used by standard Java threads.

Immortal Memory: This is a single memory area which is shared among the threads and event handlers defined by the RTSJ. Objects allocated from the immortal memory area are not deallocated until the JVM terminates. This provides the developer with the opportunity of manually allocating the objects needed ahead of time. However, managing these objects requires extra care, because they are potential sources of memory leaks. In return the immortal memory area never gets garbage collected. Therefore, object allocation or referencing operations done in immortal memory will not suffer from the unpredictable latency caused by the GC.

Scoped Memory: Several instances of this memory area can exist and these can be nested. Objects allocated in a scoped memory area are alive as long as the syntactic scope, where the objects were created, is active. Once the application leaves the scope, the entire scoped memory area is reclaimed. This makes scoped memory ideal for temporary short-lived objects. These areas are also free from garbage collection latency and hence provide higher predictability for the real-time threads using it.

These additional memory areas introduced by the RTSJ are accompanied by a set of rules, constraining how objects in the different memory areas are allowed to reference each other. These are needed in order to maintain the reference safety in Java, i.e. avoid dangling references. In general, objects residing in longer-lived memory areas must not hold references to objects residing in shorter-lived memory areas. The rules has been illustrated in figure 3.2.

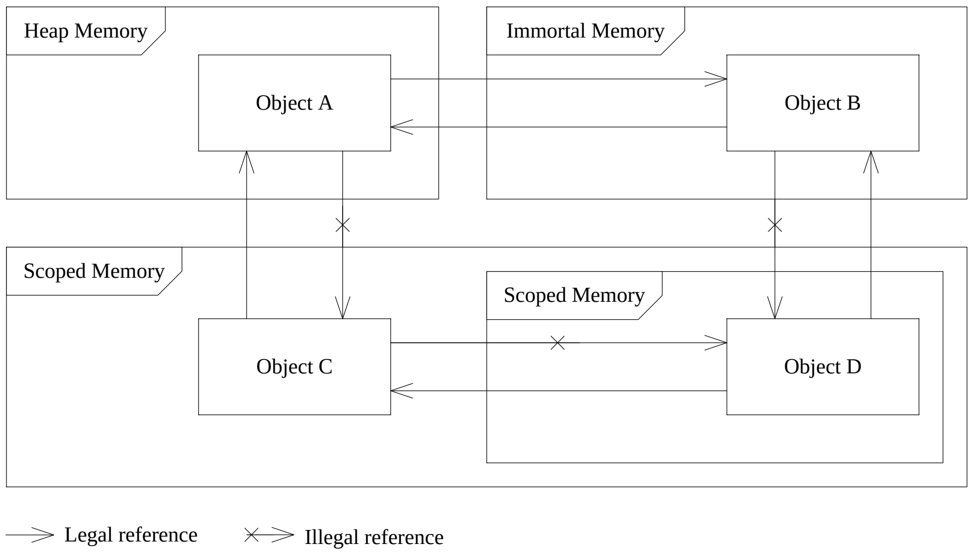


Figure 3.2: Legal and illegal references between memory areas in RTSJ

It can be seen that *Object A* and *Object B*, residing in heap and immortal memory respectively, are not allowed to reference objects residing in scoped memory areas. This is because *Object C* and *Object D* can potentially be reclaimed at any time. *Object C* resides in an outer scope while *Object D* resides in a nested inner scope. Therefore, *Object D* is able to reference *Object C* but the opposite is not allowed. This is because the inner scope is by definition shorter-lived than the outer scope it is nested within.

The RTSJ allows for the heap memory area to be garbage collected like in standard Java. However, it does not specify any specific garbage collection strategy. The rationale behind this decision is that no single optimal GC exists. The success of a garbage collection algorithm is highly dependent on the type of application i.e. the memory consumption patterns of the application. Therefore the RTSJ leaves it to the JVM vendors to choose the garbage collection strategies.

3.3. Safety Critical Java

The SCJ specification is a subset of standard Java and the RTSJ, which, as the name suggests, is targeted safety critical systems. The initial release of the SCJ specification is currently being developed as JSR-302 under the JCP. However, a draft of the specification is currently public available [Locke&11].
Safety critical systems are usually required, by certification authorities, to be certified before they can be put into operation. For instance, safety critical software used in avionics must be certified using the DO-178B certification standard [DO-178B][1]. SCJ specifies a subset of standard Java and the RTSJ, which eases the process of certification for safety critical systems written in Java.
The time and money spend on the certification process is highly dependent on the complexity of the software in question. Therefore, the SCJ specification defines three levels of compliance from 0 to 2, which allows for different levels of complexity in the application. Level 0 being least complex and level 2 being most complex.
Sections 3.3.1 to 3.3.3 will describe how the SCJ specification differs from the RTSJ when dealing with the problems presented in chapter 2, in the areas of scheduling, synchronization and memory management.

3.3.1 Scheduling

The SCJ specification introduces the concept of a *mission*, which consist of three phases called *initialization*, *execution* and *cleanup*. A mission has a fixed set of schedulable objects. Each compliance level defines a different set of rules on how missions and schedulable objects can be used.

Compliance level 0: Level 0 only allows a single mission, which is comprised of several instances of the `PeriodicEventHandler` (PEH) class. This is a SCJ defined subclass of the AEH defined by the RTSJ. The PEH's are executed in turn by a single thread of control, according to a predefined schedule. This scheduling technique and its characteristics was introduced as cyclic scheduling in section 2.2.1.1.

Compliance level 1: Level 1 also supports only one mission but allows for concurrency as each PEH executes within its own thread of control. Also aperiodic events are supported through instances of the `AperiodicEventHandler` (APEH) class. Both PEH's and APEH's are scheduled by the base scheduler defined by the RTSJ, which is a fixed-priority preemptive scheduler. These conditions make it possible to guarantee deadlines using schedulability analysis techniques like rate-monotonic analysis (see section 2.2.1.2).

Compliance level 2: Level 2 is the most complex level and allows for missions to be created and executed concurrently. Within each mission it is allowed to use the NHRT's as defined by the RTSJ, in addition to the PEH's and APEH's.

3.3.2 Synchronization

A SCJ application of compliance level 0 has no need for synchronization, as all PEH's are executed by a single thread. It is however recommended to do synchronization in order to make the application compatible with compliance levels 1 and 2.
Concurrency is supported at compliance levels 1 and 2 and hence the SCJ specification requires that the unbounded priority inversion problem must be avoided. This must be done by use of

[1]The DO-178B will soon be replaced by a new revision called DO-178C

the priority ceiling protocol (see section 2.3.1). SCJ chooses to use the priority ceiling protocol rather than the priority inheritance protocol, which is the default mechanism in the RTSJ. This is because the priority ceiling protocol minimizes the maximum latency which can be incurred on high priority threads by low priority threads due to priority inversion. Furthermore the priority ceiling protocol prevents deadlocks when used on a uniprocessor system.

The SCJ specification only allows for use of the **`synchronized`** keyword when declaring methods. This means that it is prohibited to use the **`synchronized`** block statement (see section 2.3.2). The rationale behind this limitation is that it is easily observed if a method uses synchronization just by inspecting its interface.

3.3.3 Memory Management

The non-determinism introduced through garbage collection is unacceptable in a safety critical system. Therefore the SCJ specification disallows garbage collection and hence also the traditional heap used in standard Java and the RTSJ.

Instead the SCJ specification defines two subclasses of the scoped memory area defined in the RTSJ. These are called *mission memory* and *private memory*. Each schedulable object has a private memory area which is empty when the object is scheduled, as the private memory area is cleaned after each execution. A mission memory area is shared among all schedulable objects participating in a mission and allocations are usually only done during the missions initialization phase. Object residing in mission memory are only reclaimed during the cleanup phase of the mission, hence it acts as immortal memory to the schedulable objects comprising the mission.

The immortal memory area defined by the RTSJ, which spans the life of the JVM is also available in SCJ and can be shared among all schedulable objects in all missions.

3.4. Discussion

The benefits of the RTSJ are apparent, it provides solutions for many of the issues preventing standard Java from achieving real-time performance (see chapter 2). However, the RTSJ has received some amount of criticism, which is mainly targeted the complexity of its memory model compared to standard Java [Bacon&03, Nilsen07]. Especially the scoped memory area and the memory access rules imposed on it, has given rise to debate, because it has proven hard for developers to utilize in practice [Benowitz&03A]. This has resulted in several proposals for design patterns and methodologies which seek to ease the use of the RTSJ for developers and make it less prone to errors [Benowitz&03B, Pizlo&04, Dawson07, Plsek09].

It is hard to find evidence of industrial systems utilizing the RTSJ, despite being available for more than 10 years. This is either due to lack of general acceptance in the industry or that the systems using the RTSJ are confidential.

The SCJ specification defines a much more restricted environment than the RTSJ, in order to facilitate certification. This ensures that the software developer does not accidently refer Java libraries which is not optimized for real-time, which is a potential problem with the RTSJ. Also SCJ allows for easier static analysis which makes it possible to prove if the hard real-time deadlines will be met. However, this restricted environment also limits the usage of the SCJ specification to deeply embedded and less complex systems.

Usage of the SCJ specification has so far been limited to research systems due to, it being unfinished. The work on the specification suffers from the involved parties having different interests.

Supporters of the RTSJ want the SCJ to inherent more of its features, while manufactures of products similar to the SCJ wants the specification to support their features.

Chapter 4

Real-Time Requirements

This chapter describes the first step of the TJARP method and emphasizes the importance of considering not only functional, but also non-functional requirements for real-time systems. The TJARP method encourages a detailed requirement analysis in step one, before moving to either step two or directly to step three as illustrated by figure 4.1.

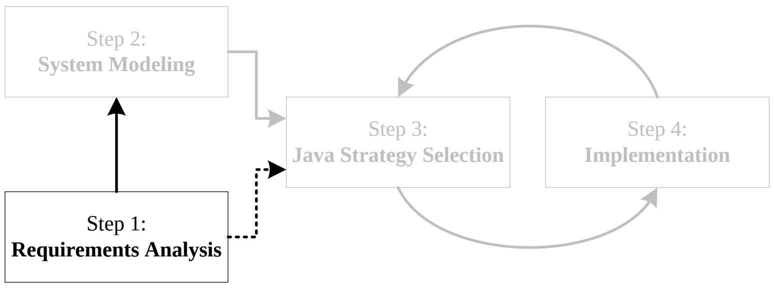


Figure 4.1: Step 1 of the TJARP method

4.1. Introduction

The importance of having consistent, feasible and unambiguous requirements when designing complex software systems cannot be questioned. Software engineers and architects tend to focus on functional requirements, i.e. what shall the system do and how should it behave in a variety of circumstances [Douglass01]. However, it is also of great importance to consider non-functional requirements when analyzing and designing real-time systems. These are also known as *Quality Attributes* or *Quality of Service* parameters.

This chapter briefly discusses the role of requirements in real-time systems, with emphasis on the different *types* of requirements relevant for defining timing constraints. Step 1 of the TJARP method contains a series of sub-steps which are explained in section 4.2. Then section 4.3 relates real-time requirements to functional and non-functional requirements and presents different types of timing requirements. Section 4.4 exemplifies how step 1 of the TJARP method can be applied to the Car Controller example. Finally the work presented in this chapter is discussed in section 4.5.

4.2. Sub-steps of the TJARP Method

The goal of step 1 in the TJARP method is twofold. First this step serves the purpose of determining the exact degree of real-time behavior, desirable for the system in question. Secondly, this step will result in a set of concrete timing constraints which will later be used to verify if the real-time system exhibits the desired temporal behavior. In order to achieve these goals the following sub-steps are recommended.

Sub-step 1.1 - Determine Degree of Real-Time Behavior: The first sub-step is to determine the required degree of real-time for the system. The choice between hard, soft or mixed real-time performance for a given system must be motivated by the original purpose of the system (business case). This usually results in a tradeoff between time, money and temporal guarantees. Hard real-time systems cannot make compromises on timing guarantees, therefore such implementations require extra care, which adds significantly to the amount of time and money spend during development.

Sub-step 1.2 - Re-assess Functional Requirements: As the TJARP method is targeted existing Java applications a set of functional requirements should be available before the TJARP method is applied. However, if the available requirements are insufficient or for instance a new sub-system has been added to the original, this sub-step serves to obtain a clear definition of the desired functionality. Furthermore, if the described functionality requires real-time performance this must be specified as either hard or soft for each requirement.

Sub-step 1.3 - Produce Non-Functional Requirements: During this step the functional requirements of the existing system is augmented with a set of timing constraints. Some non-functional requirements may not be directly associated with a functional requirement, and these must also indicate whether it is a hard or a soft real-time requirement.

4.3. Requirements in Real-Time Systems

The success or failure of a hard real-time system depends not only on its functional behavior, but also on its ability to meet critical deadlines, where a missed deadline has no value and can result in a total system failure. Whereas a soft real-time system can survive a missed deadline, but the usefulness of the result is degraded as the deadline is passed. These characteristics must clearly be reflected in the requirements of a real-time system.
Section 4.3.1 relates the requirements of real-time systems to functional and non-functional requirements and describes a few types of non-functional requirements, which are particular important to real-time systems. Section 4.3.2 presents timing requirements as a sub-category of non-functional requirements, and describes examples of relevant types of timing requirements for real-time systems.

4.3.1 Functional vs. Non-Functional

The requirements of a software system is usually separated into functional and non-functional requirements [Lauesen02]. The general definition of the two is as follows.

Functional: Functional requirements specify the functions of the system, where a system function F is defined as: F(input,state) $\rightarrow$ (output, new state)

Non-functional: Non-functional requirements specify how well the system should respond, i.e. in terms of how fast, how accurate etc.

This division can however lead to confusion when specifying timing requirements for real-time systems. It could be argued that timing constraints for hard real-time systems are functional, due to the definition of functional success of a hard real-time system being dependent on its temporal properties. Where timing requirements for soft or non-real-time systems should be categorized as non-functional (quality attributes). The ambiguity of these definitions has led to several different techniques for capturing timing requirements [Glinz05, Gilb97]. However, in this thesis timing requirements are defined as non-functional, hence a hard real-time systems ability to meet non-functional timing requirements has direct influence on the success of the system.
The process of eliciting functional requirements for real-time systems is not significantly different from that of other types of software systems. Therefore, this subject will not be covered here and focus is instead kept on the non-functional requirements.

Non-Functional Requirements

In order to assist software engineers and architects in analyzing non-functional requirements for complex software systems several methods, tools and workshops exist [Barbacci&08, Chung&09]. The following will describe categories of non-functional requirements which are particular relevant for real-time systems.[1]

Performance: For real-time systems the task of specifying performance is concerned with predictability, whether it is worst-case or average-case performance [Stankovic88]. Execution speed, latency, throughput etc. are examples of factors that must be considered when trying to define predictability. An example of such a non-functional requirement would be to specify that a given system must be able to guarantee an event response time of 1 ms, while the system is processing up to 500 events per second.

Jitter: This is an unexpected deviation from the expected time, such as the deadline of a periodic event. Jitter is an undesired property in real-time systems, and timing requirements should be specified with an upper bound on tolerable jitter.

Reliability: This is not directly related to timing. Nevertheless, it is an important factor to consider when specifying requirements for a real-time system. Reliability could for instance describe the system's ability to recover from a failure or the mean time between failures. For Java implementations this is an important motivation behind several implementations of JVMs and specifications.

4.3.2 Types of Timing Requirements

A subset of non-functional requirements, are timing requirements, which are particular important to real-time systems. Several types of timing requirements target the relationship between system events and the corresponding response. Such events may be triggered by input from the operating environment (stimuli), the passage of time (periodic) or even the absence of specific event at certain points in time [Wieringa03]. Such events may be described using one or more of the following constraints:

[1]Several other requirements, depending on the type of system, could be relevant such as *security, scalability, portability* etc.

Deadline: This specifies the time by which a responding event must be triggered based on a previous stimuli event. This can both be a fixed time or relative to the occurrence of the triggering event, for example the maximum allowed time between pressing the brake pedal in a car, until the brakes are activated.

Separation: This specifies the minimum allowed time between two events. Separation can be specified as a *required* separation, where the occurrence of the first event must always be followed by the second event, at or after the specified time. The initial and following event may be of the same type [Fitzgerald&07]. An example of such a separation constraint is the minimum time allowed between the increases in brake pressure of an elevator, to ensure a smooth reduction in speed towards the destination floor.

Minimum/Maximum: This specifies either the lower- or upper-bound on a specific attribute. They can be used in combination with other constraints or as independent criterions for a specific event. For example, the indicator light of a car might need to specify both a minimum and maximum toggle rate.

The above list is not intended to be exhaustive, but as an indication of what types of timing constraints may be relevant. Also these timing constraints are some of the most common within real-time system requirements.

4.4. The Car Controller Example

This section will apply the three sub-steps of step 1 from the TJARP method to the Car Controller example (see section 1.5.1). The task of the first sub-step 1.1 is to determine the degree of real-time behavior the Car Controller must meet. As the Car Controller is a highly simplified system it is rather trivial to determine that the system contains both soft and hard real-time parts. Timing requirements concerning the basic functionality of the car represented by its ability to accelerate and brake must be considered hard real-time. Compromises cannot be made on e.g. the amount of time from the brake pedal is pressed until the brakes are activated. Whereas the task of operating the in-car navigation display, must be considered soft real-time. Hence the Car Controller system requires mixed real-time behavior.
Once the choice between hard, soft or mixed real-time has been made, the functional requirements must be re-assessed and the non-functional requirements must be determined. The process of applying these sub-steps to the Car Controller example will be described in section 4.4.1 and 4.4.2 respectively.

4.4.1 The Functional Requirements

It is important to have a clear definition of the functional requirements before the non-functional timing requirements can be defined. Therefore the goal of sub-step 1.2 is to ensure that the functionality is well-defined before proceeding to sub-step 1.3. Each functional requirement should indicate if the specified functionality requires hard, soft or non-real-time performance.
For the Car Controller example a set of functional requirements has been stated below. Notice how each requirement is followed by a parenthesis, describing the desired degree of real-time performance for the functionality the requirement describes.

R1: When the gas pedal is pressed the system must increase the engine RPM proportional to the pedal pressure. (Hard real-time)

R2: When the cruise controller switch is activated, the current speed must be maintained by continuously adjusting the engine RPM, unless the car is stationary. (Hard real-time)

R3: When the brake pedal is pressed the system must activate the brakes with a pressure proportional to the pedal pressure. (Hard real-time)

R4: The system shall include a navigation display which indicates the current position of the car on a 2D map. (Soft real-time)

The functionality specified in requirements *R1-R3* are subject to hard real-time constraints, while requirement *R4* is considered a soft real-time requirement. These requirements will in the following section be augmented with non-functional requirements describing the desired temporal behavior.

4.4.2 The Non-Functional Requirements

Sub-step 1.3 is concerned with specifying the non-functional timing requirements for the real-time system in question. Below, a list of non-functional timing requirements for the Car Controller example is given. These requirements have been given identifiers which relates directly to the functional requirement which they constrain.

R2.1: The system must monitor the speed of the engine and with a period of 50 ms with a tolerable jitter of +/- 6% (3 ms).

R3.1: The system shall activate the brakes within 1 ms after the brake pedal is pressed.

R3.2: The time between two consecutive changes in brake pressure send to the brakes by the system must be separated by at least 0,1 ms.

R3.3: Requirements *R3.1* and *R3.2* must still hold while the system is processing up to 1000 events/second from the brake and gas pedals.

R4.1: The navigation screen must be updated with a period of 33 ms with a tolerable jitter of +/- 15% ($\sim$ 5 ms).

Note how *R3.1-R3.3* extends the functional requirements *R3* with non-functional hard real-time requirements. While *R4.1* extends *R4* with a non-functional soft real-time requirement, further specified using an average value.

4.5. Discussion

The first step of the TJARP method described in this chapter, emphasizes the importance of understanding requirements for real-time systems. Focus is on non-functional requirements describing temporal behavior, because the available literature made it clear that many different interpretations of real-time requirements exist. Where, even the general perception of time bounds has been subject to extensive research [Walkup&94]. This chapter therefore described several categories of timing specific events, and how they can be constrained by different types of requirements.
Several other techniques for capturing and describing real-time requirements exist, where especially timing requirements for hard real-time systems are often based on the use of formal specification languages. These may include use of linear-time temporal logic [Manna&92], computational tree logic [Clarke&86] or graphical interval logic [Ramakrishna&96]. The use of formal

specification of requirements is beyond the scope of this thesis, as the purpose of this chapter is to provide a common understanding of timing requirements and their nature.
Furthermore, this chapter does not focus on business requirements which are a significant factor for industrial projects. Such requirements are assumed to be determined by other stakeholders and several processes exist to help prioritize different categories of requirements such as the *Cost-Value Approach* [Karlsson97].
The determined requirements must be specified in order to progress to either step 2, or step 3 in the TJARP method. When modeling the system in step 2, these requirements serve as a validation base, and as a mean for identifying timing bottlenecks and other areas of concern. The requirements also serve as the basis for choosing the optimal Java strategy in step 3. However if the requirements and the business goal (time and money) of the system does not require hard real-time or the system in general can be considered overly simple, the benefits of step 2 does not match the extra workload, and a direct transition to step 3 is advised.

Chapter 5

Modeling Real-Time Systems

This chapter describes the role of the formal modeling language VDM-RT in the second step of the TJARP method as illustrated in figure 5.1. This chapter provides a series of guidelines for utilizing the many benefits provided by VDM-RT, and relates these to the requirements determined in the first step as described in chapter 4

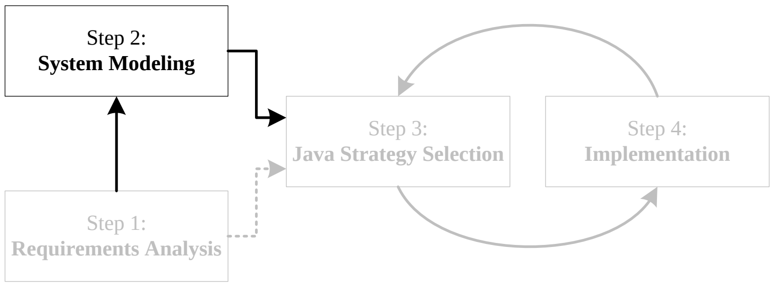


Figure 5.1: Step 2 of the TJARP method

5.1. Introduction

To increase the level of confidence that a real-time system will be able to meet its timing constraints, these systems are traditionally tested intensively during and after development. However, practical experience suggests that the development process and final product can benefit from re-allocating some of the time spend later in the development process, e.g. during testing. Instead some of the time should be spend on producing and evaluating an intermediate representation of the system. An example of such an intermediate representation is a formal model specified using the Vienna Development Method (VDM) language [Larsen&10a].
By doing initial tests on a VDM model, of the real system, it is possible to identify design flaws and bottlenecks before work on the real system is initiated. An advantage of working with a model instead of the actual system is that the model is an abstract representation simplifying many details. This allows the model to only focus on important aspects, for instance the timing constraints of a real-time system. The simplicity of the model allows it to be built using much less effort than the actual system, which enables software architects to explore different strategies for implementation and deployment early in the development process.

VDM supports modeling of temporal aspects through the VDM Real-Time dialect called *VDM-RT*. Useful features are provided by VDM-RT which allows for extending the model with a notion of time. For instance, it is possible to specify durations of operations, execution speed of processors, periods and jitter of threads, etc. Additionally, VDM-RT allows for modeling of distributed architectures by supporting deployment of models across several CPUs.
This chapter will describe how to take advantage of VDM-RT when applying the TJARP method[1]. The parts of the VDM-RT language which are of particular interest to the method and real-time systems will be introduced. However, it is expected that the reader has basic knowledge about VDM as this chapter will not introduce the language in its entirety[2].
Section 5.2 outlines the sub-steps of step 2 in the TJARP method. Section 5.3 presents the available tool support for VDM-RT, while section 5.4 exemplifies how step 2 of the TJARP method is applied to the Car Controller example. Finally section 5.5 discusses the work presented in this chapter.

5.2. Sub-steps of the TJARP method

Before creating a VDM model, it is important that the purpose of the model is clearly defined. When applying the TJARP method, the purpose of the model will typically be to investigate the temporal properties of critical components in the system in question, as well as explore alternative architectures for achieving optimal timing. The choice of components, which the model should focus on, must be motivated by the non-functional requirements developed through step 1.
When the critical parts of the system have been selected for modeling, the sub-steps listed below will help build the model in an incremental fashion. For each sub-step, the models level of detail will be increased. At the end of each sub-step, the ability to execute a VDM model is taken advantage of, by executing the current increment of the model and evaluating it. The evaluation is done with regards to relevant requirements, with emphasis on timing requirements. All requirements referred to in the following are products of step 1 of the TJARP method.

Sub-step 2.1 - Define Structure: First the top level classes of the existing system are selected and modeled. The choice of classes to model must be motivated by the purpose of the model. The functionality of the classes is augmented with invariants and preconditions according with the functional requirements. Finally, the model is executed sequentially and unit-tested in order to verify its compliance with the functional requirements.

Sub-step 2.2 - Introduce Concurrency: The next step is to introduce concurrency in to the model by identifying and modeling active classes along with their shared resources. These shared resources must then be protected by synchronization primitives. The model is then executed again to ensure the functionality of the model behaves as specified after concurrency has been introduced.

Sub-step 2.3 - Introduce Timing: Here the model is extended by giving it a notion of time. The timing requirements are used as basis for setting up timing constraints on the model. The model is then executed in order to gain confidence that no timing requirements are violated.

Sub-step 2.4 - Explore Design Space: This sub-step serves the purpose of finding the best compliance with the requirements from step 1. This is done by experimenting with different combinations of timing properties and deployments to alternative hardware architectures.

[1]The approach taken in the sub-steps of step 2 are inspired by work done in: [Larsen&09]
[2]Recommended sources for information on VDM are: [Larsen&10a, Larsen&10b, Larsen&10c]

5.3. Tool Support

Several tools exist for developing and evaluating VDM models. One example is *VDMTools*, which is a commercially available tool with various features [VDMTools]. A particular interesting feature, to the TJARP method, is the possibility of reverse engineering existing Java code in order to generate the basis for a model. However, VDMTools does not provide support for the VDM-RT dialect, required by the TJARP method for modeling real-time systems, and hence it is not ideal for use with this method.
An alternative tool is called *Overture* which is an open source application build on top of the Eclipse IDE [Larsen&10b]. Overture makes it possible to debug the VDM model using the Eclipse debugging perspective, which among other things allows for use of breakpoints. This allows for stopping the execution of the model and inspecting its current state, which is a valuable feature during development. In addition, Overture supports unit-testing which enables the developer to ensure continuously correctness of the unit tested parts of the model during development.
A feature of particular interest to the TJARP method, which can be used to evaluate models of real-time systems with the Overture tool, is called the RT Log Viewer. This allows developers to inspect and get an overview of the behavior of the model and identify any violated timing constraints.

RT Log Viewer

The RT Log Viewer (RTLV) is able to provide a graphical representation of how a VDM-RT model behaves during execution [Ribeiro&11]. An example of the RTLV in action can be seen in figure 5.3 on page 44. Using the RTLV, it is possible to inspect how threads are scheduled, how they communicate and synchronize. This is a valuable tool when evaluating a VDM-RT model of a real-time system.
During the work on this thesis it became clear that the current implementation of the RTLV was insufficient for use with medium or larger sized models. The performance of the viewer got painfully slow as the execution trace of a model grew. Therefore, the internals of the RTLV was redeveloped during this thesis for improved performance and usability.
The old implementation relied on reading a text file containing all events from one execution trace. The execution of the simple Car Controller model generates more than 1000 events per second. This result in large text files, which combined with the old implementation, greatly limited the performance of the RTLV. Another problem was the internals of the RTLV trying to read all events and loop through them several times before illustrating the execution graphics to the user.
The performance problem was solved by saving all events as a binary stream of objects instead of text. This file is faster to read, and additionally, the algorithm used for drawing the resulting graphics was redesigned. The new algorithm only loops through the limited amount of events which are shown to the user. However, if a user jumps to a later point in time, the RTLV is required to determine the state of all visible objects at that specific time. The new design solved this by updating event information from the last referenced point in time, and up to the current point in time. The new design then only parses the minimum number of events required for drawing the visible section of the execution trace.
The new RTLV implementation design is based on a clear separation of logic, where the responsibility for reading and handling events is separated from the logic responsible for updating the user interface. This allows for easy maintenance and extensions in future releases. This re-implementation is further described in appendix C.
The result is a RTLV with the same interface and functionality but with a significant increase in performance, which extends RTLV's utilization on to larger models. This is highly relevant for

the TJARP method in order for applying it to complex industrial systems.

5.4. Modeling with VDM-RT

VDM-RT supports a structured approach for analyzing real-time systems, but before proceeding to the actual development of the model it is important to define a clear purpose of the model. The definition should keep an emphasis on the requirements of the system, and especially the non-functional (timing) requirements are of interest. The purpose of a model of the Car Controller example would be *to analyze and obtain an understanding of the temporal attributes of the system and help identify potential timing bottlenecks in the current design, and allow for exploration of an distributed architecture.*

This section describes how VDM-RT is used in the four sub-steps within the second step of the TJARP method. Section 5.4.1 describes how a model can be structured based on the existing system as motivated by sub-step 2.1. Section 5.4.2 introduces the concurrency concepts of VDM-RT and how they can be used to increase the level of confidence within the model. Section 5.4.3 describes how temporal constraints can be described directly within the model and validated through the available tools as the model is executed. Finally section 5.4.4 describes how the architectural properties of VDM-RT can be utilized to do an early design space exploration with respect to the purpose of the model. The Car Controller example is used throughout these sections to stress important points and provide concrete examples.

5.4.1 Modeling System Structure

VDM-RT allows for an object-oriented design, with classes, associations and concurrency support. The recommended approach is to start by selecting relevant classes from the existing system, and create similar classes within the model. The choice of classes to include depends on their relation to the purpose of the model. If classes only allow for trivial functionality they should be modeled as VDM *types* [Larsen11]. The model should contain a top-level class to present the entirety of the system, as well as an environment class to define deployment and the available architecture. The model at this stage should only consider the overall structure i.e. only the relevant classes and their associations. Algorithms and additional functionality unrelated to the purpose of the model can be ignored by raising the level of abstraction, for example through the use VDM keywords **`skip`** and **`is not yet specified`**.

Once the model structure is in place, additional functional constraints can be introduced. The VDM language and tool support allows for internal consistency check by use of *invariants, type check, pre-conditions* and *post-conditions*. This enables the developer to document important properties and constraints directly within the model and make it subject to runtime checks.

The Car Controller Example

Sub-step 2.1 is used for gaining an overall impression of the system architecture. For the Car Controller example the classes have been mapped to the VDM-RT model as shown in figure 5.2. Additional classes has been added such as the `CarEnvironment` class responsible for deployment of architectural properties, and the `World` class responsible for initiating the environment and loading specific scenarios. The level of detail in the model has been adapted according to the purpose of this model. Hence the complex behavior of classes such as the brakes or the engine is ignored as the level of abstraction is raised.

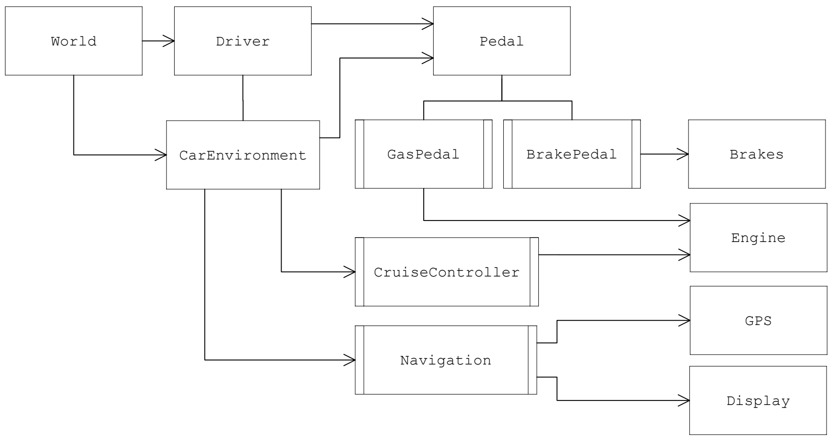


Figure 5.2: Structural class diagram of the Car Controller model

At this stage the model has no notion of time or concurrency. However the constraints of the functional requirements are introduced through the use of pre-conditions and invariants. In listing 5.1 an invariant located within the `CruiseController` class is illustrated. The invariant limits the operational behavior of the model to ensure that the automated cruise control is not activated while the car is stationary, as according to functional requirement R2 (see section 4.4).

```
    private cruiseActive : bool := false;
    private desiredSpeed : nat := 0;
inv (not cruiseActive) or (desiredSpeed > 0);
```

Listing 5.1: Invariant check on attributes within the `CruiseController` class

5.4.2 Introducing Concurrency

The initial structural model can be extended by introducing concurrency primitives to increase the level of detail and temporal confidence within the model. VDM-RT allows for modeling of active objects together with synchronization of shared objects. The basic concurrency primitives in VDM-RT are based on two types of threads:

Periodic: The concept of a periodic thread can be modeled by use of the **`periodic`** keyword, which ensures periodic invocation of the enclosed operation or statement. The keyword accepts four values: *period*, *jitter*, *delay* and *offset*. These describe the periodic interval between invocations, the allowed time variance, the minimum inter arrival distance between two invocations and the finally the timed delay of the first invocation.

Procedural: A procedural thread is defined by use of the **`thread`** keyword with an enclosed operation or statement, which is executed once the thread is started. The thread is executed sequentially and runs until completion.

Additionally, operations may be specified as asynchronous with the **`async`** keyword, which creates and activates a new procedural thread for execution of the specified operations once it is invoked.

Objects shared between multiple threads require synchronization mechanisms to ensure proper operation, which in VDM-RT is supported by *permission predicates*. Permission predicates are used to state rules for accepting the execution of operations invoked concurrently [Larsen&10a], and are defined by the **per** keyword. Permission predicates may refer to specific instance variables or *history counters*, where the latter allows the modeler to do more advanced synchronization modeling.

The Car Controller Example

Sub-step 2.2 motivates the introduction of concurrency. Four classes are identified in the Car Controller example as active classes, the `CruiseController`, the `Navigation`, the `GasPedal` and the `BrakePedal` classes. Operations within the pedal-classes are marked with the **async** keyword, as they are invoked through the communication bus by the `Driver` class. The `CruiseController` and the `Navigation` classes are implemented as **periodic** threads to model their periodic behavior. Listing 5.2 shows the periodic implementation of the `MonitorCruiseSpeed` operation monitoring and adjusting the current speed of the car. Notice how the period (50E6 ns) is assigned with a relatively low jitter value (100 ns).

```
public MonitorCruiseSpeed : () ==> ()
MonitorCruiseSpeed() ==
(
    dcl newRpm : nat := CalculateNewMotorRpm();
    if cruiseActive then engine.SetRPM(newRpm);
);

thread
    periodic(50E6,100,0,0)
        (MonitorCruiseSpeed)
```

Listing 5.2: Periodic scheduling of the `MonitorCruiseSpeed` operation

From the class diagram illustrated in figure 5.2, it is clear that the `Engine` class is a shared resource, thus requiring synchronization. This is achieved through a permission predicate, as shown in listing 5.3, with the history counters **act** and **fin**, representing the number of active and finished invocations of the operations. Notice this could alternatively be specified by using the **mutex** keyword.

```
sync
per GetRPM => #act(SetRPM) - #fin(SetRPM) = 0;
per SetRPM => #act(GetRPM) - #fin(GetRPM) = 0;
```

Listing 5.3: Permission predicates limiting the access to the `GetRPM` and `SetRPM` operations

5.4.3 Analyzing Timing Constraints

An important addition to the VDM-RT dialect for use within the real-time domain is the notion of time, which serves as a valuable property for introducing temporal constraints in the model. VDM-RT models the passing of time by use of *time-ticks* which correspond to one nanosecond on the real-life wall clock. The concept of time can be added to the model by use of two language constructs:

Duration: The **`duration`** keyword specifies a fixed value, used to increment the internal clock of the model, when executing the enclosed statement.

Cycles: The **`cycles`** keyword specifies a relative amount of clock-cycles used to calculate the increment of the internal clock when executing the enclosed statement. By using this statement the modeler can specify the execution time of a statement relative to the speed of the current `CPU`.

The Overture tool allows for specifying *validation conjectures* which are timing assertions validated as the models is executed [Fitzgerald&07]. The support for conjectures is still under development but has already proven a valuable tool for validating timing requirements. The currently supported validation conjectures are **`deadlineMet`** and **`separate`**. The first describes the maximum delay between the occurrence of a response event after a corresponding stimuli event. The latter describes the required separation in time between two events.

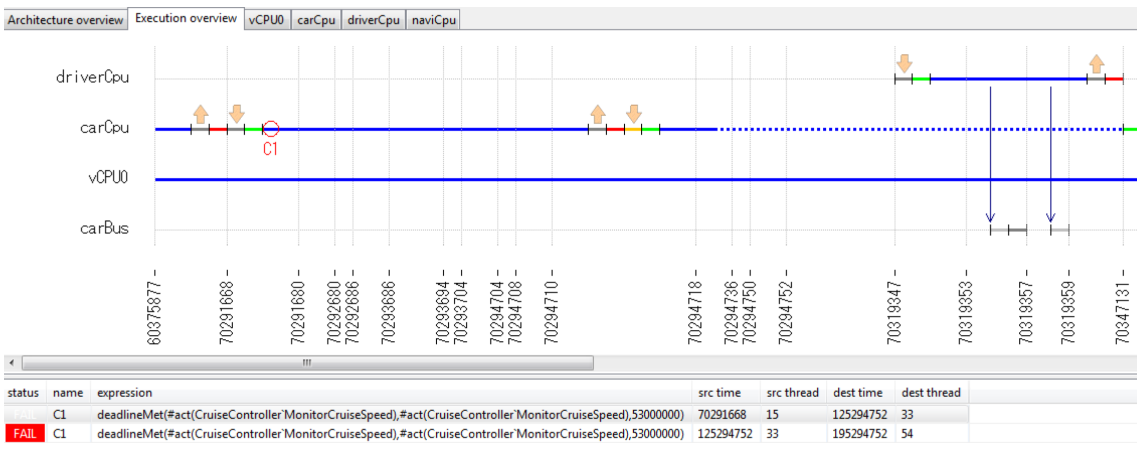
The Car Controller Example

Sub-step 2.3 introduces the notion of time into the model. Non-functional requirements can be mapped directly to validation conjectures specifying the occurrence of specific events, e.g. their deadline or their separation. An example of how requirement R2.1 (see section 4.4) of the Car Controller example is mapped directly to a validation conjecture is illustrated in listing 5.4. The conjecture validates the periodic invocations of the `MonitorCruiseSpeed` operation to be separated by no more than 53 ms (the duration of the periodic thread is set to 50 ms).

```
/* timing invariants
deadlineMet(#act(CruiseController`MonitorCruiseSpeed),
#act(CruiseController`MonitorCruiseSpeed), 53 ms);
*/
```

Listing 5.4: Conjectures for validating a timing-requirement for the Car Controller

The Car Controller example is further extended by using the **`duration`** keyword for modeling the duration of time for specific statements. In the Car Controller example, the operations within the `Display` class has been specified with durations of 22 ms in order to model the computational heavy task of updating the in-car display. Due to the periodic nature of the `Navigation` and `CruiseControl` threads, and the computational duration of the `Display` class operations, the conjecture shown in listing 5.4 is violated. This can be ascribed to the `Navigation` thread being scheduled, and updating the display, causing the `CruiseController` periods to be delayed with a total of 55 ms (33 ms thread period + 22 ms operation duration), which exceeds the maximum separation of 53 ms. Figure 5.3 shows how a validation conjecture is illustrated in the Overture RTLV, where the point in time of the violation is indicated by a red circle. The source and destination time of the conjecture is indicated, where the `CruiseController` was scheduled after 70.291.668 nanoseconds and then again at 125.294.752 nanoseconds giving a span of 55 ms.
Additional variation can be modeled for periodic threads by adjusting the jitter values, e.g. a jitter value of 5 ms for the `CruiseControl` thread also violates the aforementioned conjecture.



status	name	expression	src time	src thread	dest time	dest thread
FAIL	C1	deadlineMet(#act(CruiseController`MonitorCruiseSpeed),#act(CruiseController`MonitorCruiseSpeed),53000000)	70291668	15	125294752	33
FAIL	C1	deadlineMet(#act(CruiseController`MonitorCruiseSpeed),#act(CruiseController`MonitorCruiseSpeed),53000000)	125294752	33	195294752	54

Figure 5.3: Example of a conjecture violation for the Car Controller example

Adjusting the jitter value is another important tool for modeling the temporal non-determinism faced by many Java developers, and shows how a deviation in the periodic threads can prove important with respect to the purpose of the model.

5.4.4 Design Space Exploration

By using the distributed architectural features of VDM-RT it is possible to do an early design space exploration and evaluate the effect of different architectures on both functional and non-functional requirements. The VDM-RT dialect introduces static object deployment by additional `CPU` and `BUS` classes.

CPU: The `CPU` class represents a single physical processing unit where VDM objects can be deployed for execution. The `CPU` can be configured with different scheduling policies and with a fixed clock frequency.

BUS: The `BUS` class can be configured with different transmission policies and represents a communication channel between different `CPU`.

The `CPU` and `BUS` classes are to be created and configured within a central class with the same syntactical description as ordinary classes but with the use of the **`system`** keyword. If a distributed architecture is within the scope of the model, the timing policies of both `CPU` and `BUS` should be carefully considered, especially when used together with the **`cycles`** keyword.

The Car Controller Example

Sub-step 2.4 of the method encourages an investigation of the current architecture, with respect to the purpose of the model. By applying the architectural features of VDM-RT to the Car Controller model, the initial system is assigned to one single `CPU` instance. This causes the conjecture violation as described in section 5.4.3 since several active classes are deployed at the same `CPU`. VDM-RT allows for easy expansion of the system with an extra processing unit. The `naviCPU` is added and used for deployment of the `Navigation` and `Display` objects as illustrated in listing 5.5.

```
static public naviCpu : CPU := new CPU(<FP>,1E9);
static public carCpu : CPU := new CPU(<FP>, 1E9);

operations
public CarEnvironment: () ==> CarEnvironment
CarEnvironment () ==
(
    naviCpu.deploy(navigation);
    naviCpu.deploy(display);
    carCpu.deploy(cruiseControl);
```

Listing 5.5: Deployment of classes within the `CarEnvironment` constructor

The additional `CPU` instance ensures that the periodic processing of the `CruiseController` is not disturbed by scheduling of the `Navigation`.

5.5. Discussion

This chapter introduced the VDM-RT modeling language and its use in the second step of the TJARP method. It has been illustrated how VDM-RT can be used to provide detailed insight into the timing behavior of the system. This chapter, and this thesis in general, adopts the VDM-RT modeling language based on personal experiences of the authors and the support provided by the tool-development group located at Aarhus University. The ability of VDM-RT to simplify complex systems and promote understanding, reasoning and analysis, makes it a valuable tool before proceeding to step 3 of the TJARP method.
Modeling alternatives do exists, such as the SA/SD-RT [Wegener&98] for behavioral modeling, the Java PathFinder [Lindstrom&05] for analyzing runtime execution paths and the SARTS tool-chain for SCJ supported modeling [Bøgholm&08]. Many of these approaches cover specific areas of temporal analysis. However, by using the VDM-RT modeling language the authors has shown how the process of analyzing, understanding and checking temporal attributes of a system can be supported.
Some authors argue that a system is only able to provide hard real-time guarantees if all execution paths can be mathematically proven to adhere to the strict timing requirements [Thiele&00]. Such approaches are beyond the scope of this thesis but this is widely covered within the literature. The TJARP can be supported by extending this second step of the method with existing validation methods such as the work presented by Lu et al [Lu&11].
The general approach when designing systems with real-time constraints has traditionally been to do a thorough static analysis. These often require intensive knowledge of the runtime behavior of the system such a task deadlines, hardware response times etc. While both static and dynamic analysis often focuses on extracting certain attributes of a given system, formal models add the benefits of raising the abstraction level thus focusing on the important parts of the system with influence on the timing requirements.

Chapter 6

Towards Real-Time Java

This chapter serves as a base for choosing the best Java strategy and corresponding Java Virtual Machine (JVM), relative to the real-time constraints of the system as illustrated in figure 6.1. This relates directly to the third and fourth step of the TJARP method which encourages an iterative progress between the two, based on the requirements captured in step two. The Car Controller example is used throughout the chapter to give specific examples of optimization techniques.

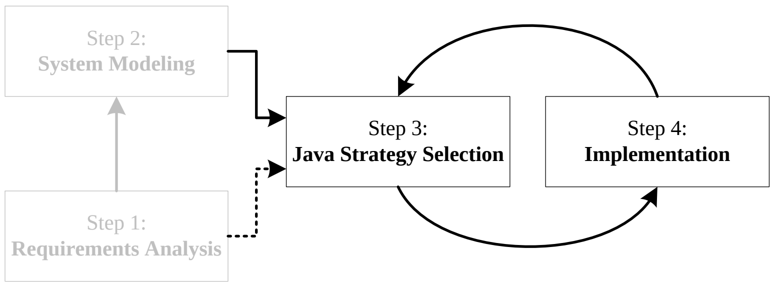


Figure 6.1: Step 3 and 4 of the TJARP method

6.1. Introduction

Several different strategies for achieving real-time performance using Java exist. Some are language specifications such as the RTSJ and the SCJ specification introduced in chapter 3, others are commercial products utilizing standard Java and proprietary real-time garbage collection algorithms. The choice of strategy or combination of strategies for achieving real-time performance in Java applications can be confusing and difficult. Therefore, this chapter will provide an overview, evaluation and comparison of some of the currently available solutions. This will serve as a basis for decisions made through steps 3 and 4 of the TJARP method, which are concerned with selecting and implementing the optimal real-time Java strategy.

This chapter starts by introducing the sub-steps of step 3 and 4 in the TJARP method in section 6.2. Then section 6.3 will describe possibilities for optimizing standard Java, towards real-time performance. In section 6.4 different Java Virtual Machines (JVM) will be evaluated on a set of qualitative and quantitative attributes. Afterwards, section 6.5 will exemplify usage of the RTSJ on the Car Controller example. Finally, section 6.6 will summarize and discuss the different real-time Java strategies described throughout this chapter.

6.2. Sub-steps of the TJARP method

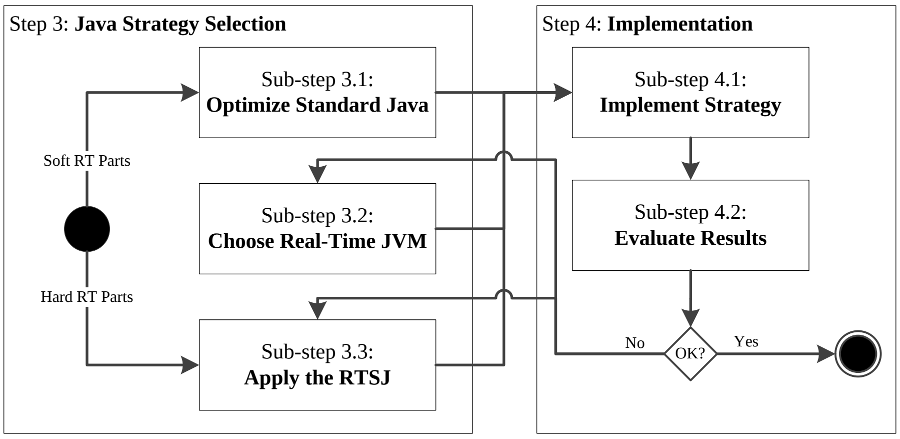

Step 3: Java Strategy Selection
Step 4: Implementation
Sub-step 3.1: Optimize Standard Java
Sub-step 3.2: Choose Real-Time JVM
Sub-step 3.3: Apply the RTSJ
Soft RT Parts
Hard RT Parts
Sub-step 4.1: Implement Strategy
Sub-step 4.2: Evaluate Results
No
OK?
Yes

Figure 6.2: Transitions between sub-steps in step 3 and 4

Through step 1 and 2, of the TJARP method, a clear understanding of the required degree of real-time performance, along with specific timing requirements for the system has been obtained. This knowledge is utilized in step 3 in order to choose an appropriate strategy for achieving real-time performance. Step 4 of the method is concerned with implementing the chosen strategy and evaluating the obtained results. The sub-steps are described in the following and an overview of the transitions between them is illustrated in figure 6.2. Soft real-time parts of the system should start at sub-step 3.1 while hard real-time parts can go directly to sub-step 3.3.

Sub-step 3.1 - Optimize Standard Java: The first sub-step is concerned with achieving soft real-time performance by optimizing the existing standard Java application. This can for instance be done by running the application through a ${}^{\tau}$*profiler*. This can potentially reveal areas of code which are inefficient as well as measuring the CPU usage, memory allocation patterns etc. of the application. The information obtained through profiling can then be used to apply techniques for code optimization and tweaking parameters of the current JVM.

Sub-step 3.2 - Choose Real-Time Java Virtual Machine: The second sub-step deals with optimization of the execution environment. This includes moving the application to a real-time optimized JVM, or enable real-time operation if it is supported by the current JVM.

Sub-step 3.3 - Apply the Real-Time Specification for Java: The final step recommends re-implementing the application or sub-parts of the application for compliance with the RTSJ. If the application or parts of it needs hard real-time performance, the previous two sub-steps can be skipped.

After any of the three sub-steps of step 3 has been performed the method transitions to step 4, in order to do the actual implementation and evaluate the results. The sub-steps of step 4 are further detailed in the following.

Sub-step 4.1 – Implement Strategy: In this sub-step the strategy decided upon in any of the three sub-steps of step 3 are implemented.

Sub-step 4.2 – Evaluate Results: Here the results obtained from implementing the changes decided upon in step 3 are evaluated. The system must be tested in order to gain confidence that the system will meet the timing requirements elicited in step 1. If the results are inadequate the method transitions back to the next sub-step of step 3.

Special considerations for mission critical systems for instance by use of the SCJ specification, are left out of the method intentionally. This is due to the immaturity of such Java-based solutions, e.g. the SCJ is unfinished and lacks supporting JVMs.

6.3. Optimization of Standard Java

The task of providing real-time guarantees using standard Java involves many challenges as described in chapter 2. However, even though standard Java is unable to provide any real-time guarantees, several options for optimization and fine-tuning are available. This is often a natural step before taking further actions towards achieving real-time guarantees, such as adopting language extensions as discussed in chapter 3. The following sections will describe some of these techniques, such as general programming guidelines in section 6.3.1, deployment techniques in section 6.3.2 and optimizations of the runtime environment in section 6.3.3. The techniques are described from a general perspective as these optimization techniques are often very application-specific, e.g. parameters for the garbage collector may vary depending on the memory allocation pattern of the specific application.

6.3.1 Development

Java syntax specific techniques for optimization of time and performance, have for several years included approaches such as using the **`final`** keyword to allow more compiler inlining, replacing virtual inheritance invocations with **`instanceof`** if-branches or avoiding **`foreach`** statements [Tene&05]. Several articles, mainly from the early days of Java, discouraged developers from using temporary objects, and promoted the creation of object pools to assist the garbage collector. Many of these approaches have since been proven to have no effect, or to be causing extra overhead and complexity [IBM04]. This is mainly due to general misunderstandings or the increasing ability of the JVM to do runtime optimization, and this is why the usage of so called "performance hacks" is discouraged and syntax optimization should be left to the compiler and the runtime environment.
It is however, of extra importance to have a general understanding of how specific Java language constructs affect performance and temporal behavior when developing real-time applications. Programming guidelines for avoiding timing complexity is still recommended such as avoiding nested loops when possible and writing "simple straightforward code" to aid the compiler [Oracle08]. The term *computational transparency* [Aicas12] is used to describe to what degree the computational effort, of a code sequence, is obvious to the developer. To heighten the degree of computational transparency the developer is encouraged to avoid, or at least understand the consequence of using specific Java constructs such as the following:

Implicit memory allocations: Statements such as string concatenations and array initializations will result in implicit memory allocations. For instance, adding two strings using the +

operator, will results in an implicit allocation of the `java.lang.StringBuilder` and an invocation of its `append` function to combine the two strings.

Final local variables: Use of the **`final`** keyword when assigning local variables is often used to provide access to the variable from an anonymous inner class. This access from the inner class will result in hidden set- and get-fields to be generated with an implicit invocation requiring memory access.

Class initializations: Java classes are explicitly initialized when using the **`new`** keyword, but are also implicitly initialized when they are first referenced e.g. by referencing any static field or static methods.

The above is not meant as an exhaustive list, but as an indication of how important it is for a real-time Java developer to understand the temporal and spatial attributes of the language semantics.

6.3.2 Deployment

The deployment process of Java applications is highly dependent on the JVM, but also the *Just-In-Time* (JIT) compiler, which generates native code based on Java intermediate files. The JIT compiler is a key feature for obtaining platform independence and runtime optimization. It is able to compile specific code segments once they are invoked. By analyzing the underlying hardware, the JIT compiler can optimize the native code for that specific platform. The nature of the JIT compiler introduces undesirable temporal non-determinism when compiling. However, several JVMs allow for application specific configuration of the JIT compilers and the corresponding classloaders used by the JIT compiler to locate classes at runtime. These configuration parameters are often based on some of the following strategies:

Lazy Loading: When using *lazy loading*, which is often the default class loading scheme used by non-real-time JVMs, the classes are loaded during runtime when they are first referenced. Any reference from the newly loaded class is only loaded once they are referenced and so forth. This ensures a fast startup time, but introduces unpredictable delays during execution.

Early Loading: Many JVMs support a configuration called *early loading* (or *eager loading*). When invoked, the JVM will do a recursive class loading of all classes referenced by the current class being loaded. Some implementations do this during startup, meaning a complete static class loading scheme, while others do it runtime upon the first invocation of the current class.

Adaptive Loading: The *adaptive loading* is a combination of lazy and early loading, and currently used in the Oracle HotSpot JVM. The idea is to use a *Hot Spot Detection* algorithm to detect code sections which are most likely to be executed. This is done by an interpreter which monitors the applications code behavior while trying to identify performance critical parts of the code.

The above schemes are important to consider when optimizing for real-time performance. Lazy and adaptive loading are unfit for real-time applications as they will potentially incur unpredictable delays. Early loading is an improvement of the characteristics of the traditional JIT challenges.
Several vendors provide Ahead-Of-Time (AOT) compilation to prevent byte code compilation from occurring at critical sections of program execution. AOT compilation does not benefit from the performance optimization techniques employed in many JVMs, as the target architecture must be determined a priori. The AOT compiler can afford to spend more time optimizing critical

sections of the code, compared to traditional JIT compilation. However, as the JVM Specification requires support for dynamic class loading, all compliant JVMs must include support for ^{T}*reflection* and dynamic byte code generation. This is for example illustrated when using the `Class.forname()` function as shown in listing 6.1.

```
public static void main(String[] args)
{
        Class classA = Class.forName(args[0]);
        ...
}
```

Listing 6.1: Example of reflection

From listing 6.1 it is seen how the instance named `classA` is runtime-dependent on the value of `args[0]`, and loaded through reflection using the `Class.forName` function. Such dynamic loading cannot be compiled AOT, and will require the JIT compiler to process the specific instance when invoked. Such situations must be considered by the developer when optimizing Java for real-time performance. Also a complete recursive AOT compilation may result in oversized footprint including many unused classes and libraries. A combination of AOT and early loading is supported by many vendors of real-time optimized JVMs, e.g. Atego provides this for their PERC Ultra JVM through their *ROMizer* [Atego12] and Aicas for their JamaicaVM through their *JamaicaBuilder* [Aicas12].

6.3.3 Runtime Environment

Options for optimization of the runtime environment are often specific to each JVM. This require a thorough understanding of their configuration, as most available JVMs allow for a highly configurable runtime environment including several parameters affecting the temporal behavior. Many JVMs implement "smart tuning" where the runtime environment is able to analyze and optimize the application at runtime. As mentioned in section 2.4, memory management has a significant impact on the temporal non-determinism in standard Java systems, which makes tuning the garbage collector algorithm an important factor. There is no "right way" of configuring the garbage collector as the best choice depends on the memory usage patterns of the specific application. The memory usage can be analyzed using various ^{T}*profilers*, which enables the memory usage of an application to be detailed. Profilers can also help detail CPU usage, thread scheduling, inefficient code segments, etc.

Figure 6.3 shows an example of how the runtime memory allocation behavior of the Car Controller example can be analyzed using a profiler. By investigating the memory profile, the garbage collector can be fine-tuned, where in this case the application is executed on the HotSpot JVM which uses a generational garbage collector (see section 2.4). Notice how the survivor space in figure 6.3 takes up a very small piece of the total allocated heap, because the application has a constant allocation rate with many short lived objects allocated and later reclaimed in the eden space. By optimizing the garbage collector profile (details included in appendix D) the maximum jitter can be reduced as shown in table 6.1.

Some vendors provide "real-time optimized" JVMs targeted standard Java applications without real-time language extensions as those mentioned in chapter 3. The *Atego PERC Ultra* is such a JVM and is further described in section 6.4. In order to compare the impact of optimizing the garbage collector, the Car Controller example has been used. The jitter of the `CruiseController` thread (similar to the test described in section 2.4.2.2) has been measured when

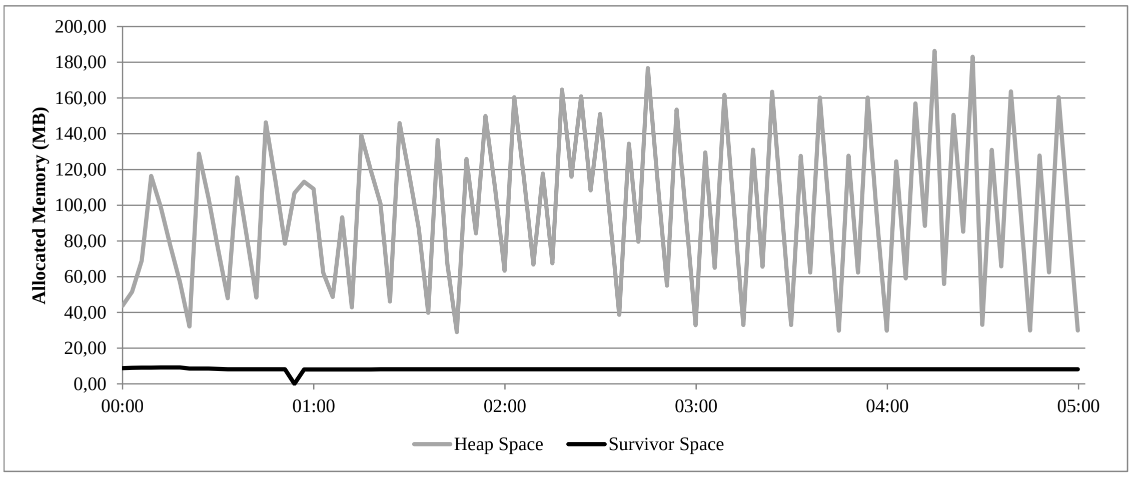


Figure 6.3: Car Controller Memory Profile

executing on the HotSpot with, and without, an optimized garbage collector. For comparison, the same test has been performed on the PERC Ultra JVM. The test results are shown in table 6.1

Thread Period (ms):	50	**Maximum**	**Average**	**Std. Deviation**
HotSpot Default Settings:	**Jitter (ms):**	73,26	0,76	2,81
	Jitter (%):	146,52%	1,52%	5,61%
HotSpot GC Tuned:	**Jitter (ms):**	62,11	0,52	1,54
	Jitter (%):	124,23%	1,05%	3,09%
PERC Ultra GC Tuned:	**Jitter (ms):**	6,98	0,78	0,48
	Jitter (%):	13,95%	1,56%	0,97%

Table 6.1: Comparison of jitter on HotSpot, with default and optimized settings and the PERC Ultra

Table 6.1 shows how the average and standard deviation jitter of the HotSpot JVM can be reduced by optimizing the garbage collector strategy, and how the PERC Ultra performs with the same application. The HotSpot optimization includes configuration of the generational ratios, forcing the garbage collector to do more frequent collections in the eden space and as a result the average jitter was reduced. It is important to notice that the used example has a constant memory allocation rate, which allows for a static configuration, where many real-life applications have a more dynamic memory allocation rate requiring a more general or adaptive configuration.

The runtime environment includes other important aspects, such as the underlying operating system and the available hardware resources. Some JVMs map threads directly to native threads as mentioned in section 2.2, which emphasizes the importance of choosing the correct operating system just as available memory, processing power etc. must be considered as well.

6.4. Analysis and Benchmark of Java Virtual Machines

The JVM is an essential component in any Java system, and for real-time systems the JVM must optimize application execution for temporal determinism. This section compares some of the most relevant JVMs targeting real-time performance, available on the market today[1].
Many real-time JVMs are discussed throughout the literature, however several are based on research projects or are no longer available. The following is a list of some of the most interesting and their current status:

Timesys RTSJ Reference Implementation: Timesys was the official RTSJ maintenance lead for specification 1.0.2 and responsible for the Reference Implementation (RI) [RTSJRI]. The JVM is no longer available and the RI download website has been closed.

Aicas JamaicaVM: The commercial JamaicaVM from Aicas provides RTSJ compliance along with deterministic garbage collection, AOT compilation, static linking and compatibility with the Java Standard Edition (JSE) 1.6 libraries [JamaicaVM]. The JamaicaVM runs on various platforms and provides a free evaluation edition.

IBM WebSphere Real-Time: The IBM WebSphere Real-Time is a commercial real-time optimized JVM from IBM, targeted enterprise applications [WebSphereRT]. The JVM is RTSJ compliant and provides a real-time garbage collector, AOT compilation and support for the JSE 1.7 libraries. WebSphere is certified to only a few IBM hardware profiles, and only supports two operating systems when using RTSJ applications (Red Hat Enterprise and SUSE Linux Enterprise Real-Time).

Java RTS: Oracle's Java Real-Time System (Java RTS) is a commercial JVM with RTSJ compliance [JavaRTS]. The JVM is no longer available, and is rumoured to be merged together with the Oracle JRockit JVM in near future.

Atego PERC: The commercial PERC product suite from Atego consists of four different virtual machines each targeting real-time Java systems of varying degrees of predictability [PERC]. The PERC Ultra is not RTSJ compliant but includes a deterministic garbage collector, AOT compiling and support for a wide variety of platforms and operating systems. The PERC Pico and PERC Raven JVMs are targeted platforms with hard real-time requirements and include some of the principles found in the SCJ specification and the RTSJ.

OVM: The OVM is a JVM based on a research project from Purdue University [OVM]. The OVM provides a real-time garbage collector and support for a large subset of the Java language, but still lacks functionality in key areas [Pizlo&09].

FijiVM: The FijiVM is a JVM from Fiji Systems and Purdue University which runs on an operating system targeted embedded systems [FijiVM]. The FijiVM provides real-time garbage collection, AOT compilation and SCJ support.

aJile: The aJile runtime environments from aJile Systems are a series of hardware coded real-time Java platforms [aJile].

Of the above JVMs several are discontinued or no longer maintained. Others are only targeted embedded systems which leave only a few relevant for the purpose of this thesis: the JamaicaVM, the IBM WebSphere and the Atego PERC Ultra. Unfortunately, it has not been possible to obtain

[1]As of December 2012

a license for the IBM WebSphere during the work on this thesis. This leaves the JamaicaVM and the PERC Ultra, which will be used for comparing real-time performance. The Oracle HotSpot JVM is included for comparison with a non-real-time JVM.
The JVMs are analyzed in the following sections, where section 6.4.1 describes the parameters used for comparison. These parameters are comprised of a set of qualitative and quantitative attributes. The latter is determined through use of benchmark tests which are introduced in section 6.4.2. Section 6.4.3 describes and evaluates the Oracle HotSpot, section 6.4.4 describes and evaluates the Atego PERC Ultra and finally section 6.4.5 describes and evaluates the Aicas JamaicaVM.

6.4.1 Basis for Comparison of Java Virtual Machines

In order to choose the correct JVM for a specific application's requirements, it is important to consider a number of attributes, relevant to real-time systems. This thesis divides these attributes in to two categories: quantitative and qualitative. The first category consists of three quantitative parameters which are determined through a series of benchmark tests:

Determinism: Determinism is an indication of how well the JVM is able to provide predictable temporal results. This is based on jitter measurements which is be separated into *maximum*, i.e. what is the latest occurrence of an event compared to the expected value, and in *average*, i.e. what is the general distribution of values. Lastly the *standard deviation* jitter gives an indication of dispersion from the average value, i.e. the nature of outliers.

Performance: The JVM performance describes how well the JVM optimizes the execution path, including its utilization of the available hardware resources and the efficiency of the JIT compiler. Performance is measured in operations per minute (ops/m), where the overall- (average) and peak-throughput are both important.

Initialization: The initialization parameter describes how the JVMs performs when starting, loading and executing the application. As mentioned in section 6.3.2 several parameters affect the initialization phase, such as JIT compilation and class loading. To provide a fair comparison, all JVMs are configured to use JIT compiling thus disregarding eager loading and AOT compilation.

While the above attributes are indeed important, the second category of qualitative attributes should also be considered before choosing a specific JVM implementation. This category consists of five parameters which are difficult to measure but can be obtained through analysis:

Maturity: The term maturity is an indication of how well proven the JVM is in terms of time on market, development support, the nature of the developers (enterprise, research project etc.) and the available community. A high degree of maturity includes a wide support for development tools such as Software Development Kit (SDK), JVM-specific compiler, runtime profilers etc.

Specification Support: The support for language specifications is an important aspect differencing several JVM products. This covers specifications ranging from the Java Language Specification to the RTSJ and the SCJ.

Scheduling: The scheduling parameter is an indication of the JVMs ability to handle real-time scheduling and thread preemption. This covers how well the JVM respects priorities, whether the scheduling is left to the Operating System (OS) or it is handled by the JVM internally.

Synchronization: The synchronization parameter describes the measures taken by the JVM in order to avoid synchronization challenges including unbounded priority inversion. This includes mechanisms such as the priority inheritance protocol and the priority ceiling protocol (see section 2.3.1).

Memory Management: The implementation of memory management, and especially the nature of the garbage collector, has a high influence on the real-time performance of the JVM. To gain a high degree of temporal determinism the JVM memory management should include a real-time garbage collector that can be preempted by application thread, and allow memory access unaffected by garbage collection delays.

The above eight attributes are used as a basis for comparing the Hotspot, the PERC Ultra and the JamaicaVM in the following sections. All are given grades of 1-3, where 3 is the best and 1 the worst. The qualitative attributes are graded based on an individual analysis with focus on real-time performance, i.e. a JVM providing a separate scheduler with real-time priorities will achieve a high score in scheduling. The analysis is based on available literature and the authors' experience obtained from working with each of the JVMs. The assigned grades are a result of the assessments done by the authors, and appendix D provides further details about the scale used for assigning grades.
The quantitative attributes are determined through a series benchmark tests, and are given scores based on their relative values, i.e. the JVM achieving the highest performance is given the score 3 and the one achieving the lowest is given the score 1.
The following sections 6.4.2 to 6.4.5 provide details about the JVM analysis. The results of the analysis are summarized, compared and discussed in section 6.6.

6.4.2 Benchmark Approach

The main purpose of benchmarking the available virtual machines is to determine the runtime attributes, but also to gain an impression of the required workload associated with optimizing and porting an existing application to the JVM. The benchmarks must mimic a realistic application behavior equal to that of a typical real-time system, such as scheduling periodic events together with non-real-time "background" processing (noise).
Two existing open-source benchmark applications were chosen for determining the metrics used to compare the quantitative attributes:

Collision Detector: The Collision Detector (CD) benchmark from Purdue University is designed for evaluating real-time capabilities on different JVMs, targeting both hard and soft real-time performance [CD]. CD simulates an air traffic control system, with a periodic hard real-time thread (detector) monitoring flight positions, and calculating potential collisions. Flight positions are extracted from radar information provided by a simulator thread (simulator) which generates radar frames based on, user defined, air traffic configurations. The benchmark monitors the time between the release of the periodic detector thread, as well as the time it takes to calculate potential collisions. The calculations are based on a full 3D collision detector algorithm, comparing the current position of all flights as well as their direction extracted from two consecutive frames. The CD benchmark provides support for running with RTSJ threads and memory as well as standard Java threads and memory (through a wrapper).

SPECjvm2008: The SPECjvm2008 (SPEC) benchmark suite from the Standard Performance Evaluation Corporation is designed for measuring performance of JVMs, using a range of different test categories each focusing on core Java functionality [SPEC]. SPEC was chosen due to its ability to precisely measure the performance of each JVM in terms of operations per minute (ops/m). The SPEC benchmark is used in isolation for a performance test, and together with CD to simulate background noise. The SPEC test *compress* were chosen as the benchmark test from the SPEC suite. The compress test executes a series of file compression iterations, and gives a realistic workload for evaluating the individual JVMs performance. The SPEC test memory and CPU-load profile are shown in appendix D.

Based on the SPEC benchmark and a modified version of the CD benchmark, three tests were composed, configured and applied to each of the JVMs investigated. Table 6.2 shows how the CD and SPEC benchmarks are used in the tests as well as which metrics are extracted.

		Test One	**Test Two**	**Test Three**
Benchmark				
	CD	-	✓	✓
	SPEC (Compress)	✓	-	✓
Metric				
	Determinism	-	✓	✓
	Performance	✓	-	✓
	Initilization	-	✓	✓

Table 6.2: Overview of applied benchmark tests

Test one: Test one uses the SPEC benchmark, which targets performance by running five consecutive iterations of the compress algorithm to determine the peak and average performance, measured in operations per minute (ops/m).

Test two: Test two consist of the CD benchmark targeting determinism, where the detector thread is configured to be scheduled every 50 ms. The actual timestamp of each period is collected and the jitter is calculated.

Test three: Test three is similar to test two, except the SPEC benchmark is loaded and executed by the CD benchmark to simulate background noise. The purpose of test three is to see how well the JVM handles a thread with a heavy CPU and memory load, while servicing a real-time thread with higher priority i.e. the real-time scheduling mechanism of the JVM. In test two and three, the processing time (total runtime for the detector thread) is subtracted from the total test time to indicate the initialization overhead. The CD benchmark is configured to use RTSJ threads and memory model when possible for test two and three.

For presentation of benchmark results the jitter for test three is plotted in a graph, to determine the layout of the overall profile. Furthermore the maximum, average and standard deviation jitter are collected and summarized in the corresponding table for both test two and three. However, these values ignore the initial 500 samples in order to reduce the impact of the initialization overhead introduced by factors such as JIT compilation and dynamic class loading.
Given the different settings and attributes of the individual JVM, finding a common (and fair) comparison configuration is a difficult task. However, a common denominator for each of the JVMs have been determined and their complete configuration together with a detailed description of each test are included in appendix D.

6.4.3 Oracle HotSpot

The HotSpot JVM is maintained by Oracle, which is the company behind the Java language specification. HotSpot is the default JVM included when acquiring the *Java Runtime Environment* (JRE) from Oracle. The HotSpot JVM focuses on achieving maximum performance and is not designed for real-time applications. This allows the HotSpot to utilize an "ergonomic" [Oracle12] runtime behavior which means it is able to optimize the performance of the executing application. This is done by dynamically selecting compiler, heap configuration and garbage collector for optimal performance at run-time. This dynamical adaption introduces a significant amount of temporal non-determinism and hence this approach cannot be used by JVMs providing real-time guarantees. The HotSpot JVM suffers from many of the challenges associated with achieving real-time performance in standard Java systems (see chapter 2).
The HotSpot JVM has been analyzed and graded within the five categories and the results are summarized in table 6.3.

Oracle HotSpot	Grade (1-3)
Maturity:	3
Language Specification Support:	1
Scheduling:	1
Synchronization:	1
Memory Management:	1

Table 6.3: Oracle HotSpot Grades

The basis for each grading is:

Maturity: Since its release in 1999, the HotSpot JVM has been deployed in millions of systems, including desktop computers and servers [Sun01]. As the HotSpot JVM is provided by the company behind the Java programming language, the JVM always has support for the latest libraries (currently JSE 1.7). Also multitudes of *Integrated Development Environments* (IDE) are available to developers. The HotSpot is regarded highly mature and has become the de facto standard among JVM implementations. Therefore Oracle HotSpot has been given a maximum grade of 3 in the maturity category.

Language Specification Support: The HotSpot JVM only supports standard Java and is therefore given a grade of 1.

Scheduling: The HotSpot JVM maps Java threads directly to native threads of the underlying operating system and therefore has no control over how these are scheduled. Application behavior can therefore differ depending on which underlying operating system is used. Furthermore, it is possible that different Java thread priorities are mapped to the same native priority of the underlying OS. Therefore the HotSpot JVM is given a grade of 1 in scheduling.

Synchronization: There is no mechanisms provided for avoiding unbounded priority inversion in the HotSpot JVM. Therefore a grade of 1 is given in this category.

Memory Management: The HotSpot memory model is separated into three generations, a young, an old and a permanent generation. This is why all supported garbage collectors are categorized as generational collectors. The HotSpot JVM supports several different garbage collector principles, many of which are covered in section 2.4. These include a *serial collector* with a traditional stop-the-world approach, a *parallel collector* using multiple CPUs and a *concurrent mark-sweep* collector splitting the steps of marking and collecting garbage into multiple incremental steps for faster response time. Recently (as of JSE 1.7) the HotSpot has been extended with a *Garbage-First* (G1) collector which uses a more intelligent memory analysis approach, where the different generations are further separated into regions. The G1 collector uses a parallel mark-sweep approach to collect and copy entire regions, thus allowing for more compacting. The HotSpot automatically selects a collector algorithm if not explicit instructed otherwise, based on simple criterias such as available memory and CPU cores. The garbage collected heap is the only available memory area in the HotSpot JVM and the provided algorithms are allowed to preempt application threads at any time. Therefore the HotSpot is given a grade of 1 in this category.

The conclusion of this analysis is, not surprisingly, that the HotSpot is not suitable for real-time systems, however this is not its purpose either. The HotSpot is built for performance and has been included in this analysis to represent a standard Java JVM.

Benchmark Results

The three benchmark test were carried out on the HotSpot JRE 1.6 with the configuration described in appendix D. The tradeoffs between temporal determinism and performance can be observed in table 6.4 where the results for all three test are listed. For test one the HotSpot JVM achieves a relative high average and peak performance, which is slightly lower for test three where the CD benchmark is combined with the SPEC benchmark. The ability of the HotSpot JVM to still achieve a relative high performance metric in test three is also reflected in the jitter results, with a high increase in average jitter compared to test two where the CD benchmarks runs in isolation. The high maximum jitter for test two is believed to be caused by a major garbage collection of all garbage generations.

	Test One	**Test Two**		**Test Three**	
Average Performance	30,08 ops/m	-			27,47 ops/m
Peak Performance	30,15 ops/m	-			27,47 ops/m
Maximum Jitter	-	149,82 ms	299,64%	51,54 ms	103,08%
Average Jitter	-	0,26 ms	0,52%	0,62 ms	1,24%
Standard Deviation	-	2,24 ms	4,49%	2,88 ms	5,76%
Initilization Time	-		9,9 s		72,70 s

Table 6.4: Test results for Oracle HotSpot based on samples 500-5000

The jitter profile of the detector thread, in test three, is illustrated in figure 6.4. The jitter profile for test two are omitted here but are included in appendix D. Figure 6.4 shows one outlier with a jitter value of 103% and periodic interrupts of the detector thread, assumed to be caused by the *stop-the-world* garbage collector. The frequent garbage collector pauses matches the memory profile of the SPEC benchmark (running together with CD in test three).
The benchmark results support the qualitative analysis of the HotSpot JVM. The HotSpot JVM achieves well for performance with a relative high ops/m count and a low initialization overhead. However, the JVM is shows temporal non-determinism with periodic values deviating more than 100% from the expected.

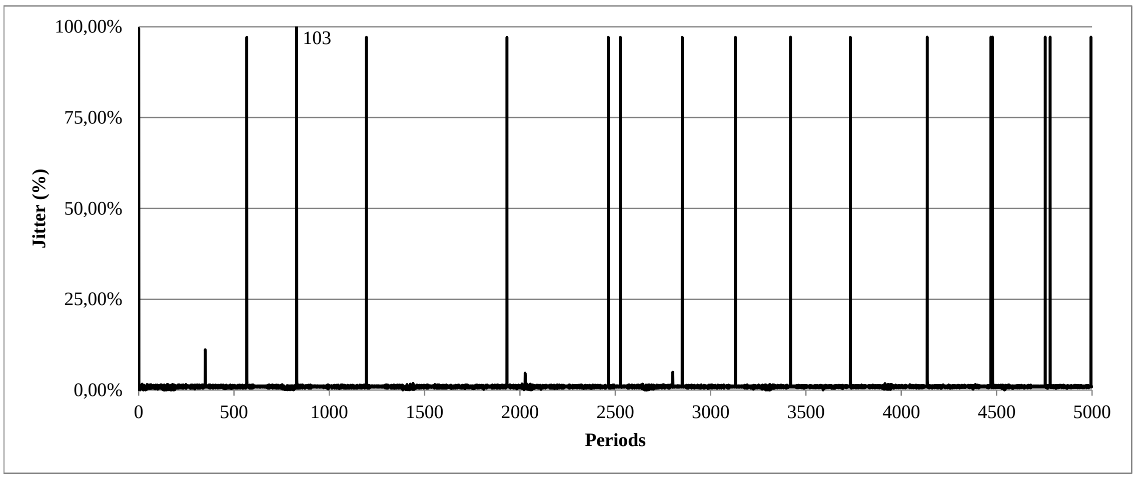


Figure 6.4: Jitter distribution for the HotSpot test three

6.4.4 Atego PERC Ultra

The PERC Ultra is one of several real-time JVMs, in the PERC family, developed by Atego. It is targeted soft real-time and embedded applications and supports the JSE libraries. The real-time performance of the PERC Ultra is achieved through predictable scheduling and synchronization of threads, accompanied by a proprietary real-time garbage collection algorithm.
The PERC Ultra has been analyzed and graded within the five categories and the results are summarized in table 6.5.

Atego PERC Ultra	Grade (1-3)
Maturity:	2
Language Specification Support:	1
Scheduling:	3
Synchronization:	2
Memory Management:	2

Table 6.5: Atego PERC Ultra Grades

The basis for each grading is:

Maturity: PERC Ultra has been on the market since 1997 [Nilsen09], and has been deployed in a large range of different real-time and critical systems. One of the most notable deployments, is in a US Navy weapon system called Aegis [Atego06]. PERC Ultra provides support for most JSE 1.6 libraries, either through own proprietary implementations or freely available open source implementations. The Eclipse IDE and its build-in debugger are supported for local or remote debugging of applications. Several tools are provided for monitoring and profiling applications, which can help tune the JVM for real-time performance. As a real-time JVM, the PERC Ultra is considered mature and is given a grade of 2 in this category.

Language Specification Support: Atego has chosen not to support the RTSJ in PERC Ultra and hence only supports standard Java. The reason for this, given by Atego, is that the PERC Ultra is built on technology which predates the RTSJ and that the use of the RTSJ API is difficult and prone to errors. Atego argues that soft real-time can be achieved with standard Java, by use of their JVM, the libraries provided with it and the included real-time garbage collector. If hard real-time is needed then Atego refers to another JVM product of theirs, called the PERC Pico [Nilsen09]. Therefore the PERC Ultra is given a grade of 1 in this category.

Scheduling: The scheduling mechanism of the PERC Ultra is a fixed priority preemptive scheduler with round robin scheduling for threads of equal priority (see section 2.2). The JVM maps Java threads directly to native threads of the underlying OS. However, the scheduler inside the PERC Ultra ensures that only one Java thread per processor appears to be eligible for execution to the scheduler, of the underlying operating system, at any time. The scheduler has a range of adjustable options for setting the duration of a timeslice, mapping of priorities between Java and the OS, etc. It is also possible to extend the range of available thread priorities to 32, instead of the 10 priorities defined by standard Java. Therefore the PERC Ultra has been given a grade of 3 in the scheduling category.

Synchronization: The PERC Ultra implements the priority inheritance protocol in order to avoid the challenges of unbounded priority inversion (see section 2.3). The priority ceiling protocol is not supported and hence a grade of 2 is given in this category.

Memory Management: The garbage collector of the PERC Ultra combines copying collection with mark-sweep collection (see section 2.4). The heap is divided in to many equally sized regions. A full garbage collection cycle consists of a mark-sweep collection followed by a copying collection. The mark-sweep collection leaves the heap fragmented. Therefore, the copying collection is applied afterwards in order to defragment the heap, by in turn using the small memory regions as *from* and *to* regions. The garbage collection algorithm performs the collection in small increments, which ensures that the application is able to preempt the garbage collector, and the garbage collection can be resumed from where it was left. The garbage collector is implemented as a periodic thread which must compete for the CPU alongside the application threads. Therefore the garbage collector thread is configurable with a priority, timeslice, period and the amount of heap space which must be in use before the thread is considered eligible for execution. The PERC Ultra is given a grade of 2 in this category.

The conclusion of this analysis is that the PERC Ultra provides features for achieving soft real-time performance. However, the JVM lacks features for providing hard real-time guarantees, hence it has been given a medium grade in the categories synchronization and memory management.

Benchmark Results

The three tests were carried out on the PERC Ultra JVM 6.1 SMP, with the configuration described in appendix D. The parameters were based on the knowledge obtained through a two day workshop which introduced the PERC Ultra, together with the general understanding of real-time theory (see chapter 2). The PERC Ultra is targeted real-time performance and allows for multiple configuration parameters, but with the limited resources in terms of scope and time, a baseline configuration was chosen to resemble the parameters available in the HotSpot and JamaicaVM.
Table 6.6 shows how the PERC Ultra JVM achieves significant lower ops/m for the SPEC benchmark compared to the HotSpot, in both test one and three. The maximum jitter is increased severely in test three where the CD and SPEC benchmark are executed together, compared to

test two with the CD executing in isolation. It can be seen how the PERC Ultra does not handle scheduling as expected, even though the detector thread (in CD) has a priority higher than the SPEC benchmark threads. The noise added in test three increase both the maximum and average jitter values.

	Test One	Test Two		Test Three	
Average Performance	7,07 ops/m	-			4,89 ops/m
Peak Performance	7,12 ops/m	-			4,89 ops/m
Maximum Jitter	-	45,87 ms	91,75%	148,48 ms	296,97%
Average Jitter	-	0,15 ms	0,31%	0,50 ms	1,01%
Standard Deviation	-	0,72 ms	1,44%	3,66 ms	7,32%
Initilization Time	-		23,10 s		79,15 s

Table 6.6: Test results for Atego PERC Ultra based on samples 500-5000

The jitter profiles of the detector thread, from test three, is illustrated in figure 6.5. The jitter profiles for test two is omitted here but are included in appendix D. Figure 6.5 shows several outliers with the maximum of 5099,98% located at the first period, and is ascribable to JIT compilation – notice that the test were completed without eager linking or eager JIT compilation. However, when comparing figure 6.5 with the profile from test two in appendix D it can be seen that the added noise from the SPEC benchmark introduces severe jitter. This is expected to be caused by the garbage collector not being able to catch up with the *out-of-memory* situations caused by the high allocation rate of the SPEC benchmark, thus forcing it to pause the detector thread.

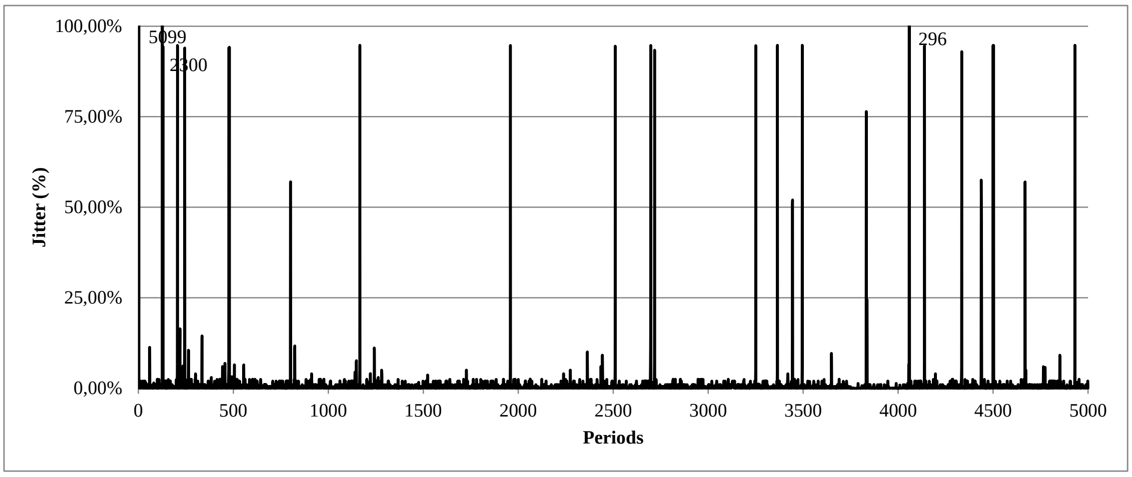


Figure 6.5: Jitter distribution for Atego PERC Ultra test three

The results from the three tests show, as expected, that the PERC Ultra JVM achieves less in both peak and average performance compared to the HotSpot. However, the jitter values, and especially the increase in jitter from test two to test three, does not match the expected behavior. The PERC Ultra was expected to achieve a more predictable runtime behavior for the detector thread, even when other threads (with lower priority) has a high memory consumption. It is important to notice that Atego provides an optimization tool with the PERC Product Suite, capable of optimizing applications for speed and performance. It is expected that such a tool might provide an increase in performance, while a more thorough investigation and optimization of the garbage collector parameters might decrease the jitter values.

6.4.5 Aicas JamaicaVM

The JamaicaVM is a JVM developed by Aicas targeted embedded, soft and hard real-time applications. This JVM achieves real-time performance by complying with the RTSJ and by use of a proprietary garbage collection algorithm. The JamaicaVM has been analyzed and graded within the five categories and the results are summarized in table 6.7.

Aicas JamaicaVM	Grade (1-3)
Maturity:	2
Language Specification Support:	2
Scheduling:	3
Synchronization:	3
Memory Management:	3

Table 6.7: Aicas JamaicaVM Grades

The basis for each grading is:

Maturity: The JamaicaVM has been available since 2001 and is deployed in numerous real-time systems mostly within avionics. An example is an unmanned aircraft called Barracuda [Aicas06]. JamaicaVM provides support for JSE 1.6 libraries which is mainly comprised of freely available open source implementations. A plugin for the Eclipse IDE is provided for generating build scripts and it is possible to debug local or remote applications through the IDE. The JamaicaVM can be configured to record profiling information about applications which can be used to tweak the JVM. It is also possible to record and inspect individual thread behavior through the *ThreadMonitor* tool. The JamaicaVM is considered a mature real-time JVM and is given a grade of 2 in this category.

Language Specification Support: The JamaicaVM supports the RTSJ 1.0.2. However, by default some of the strict rules imposed by the RTSJ are relaxed in order to ease development. The reason for this, given by Aicas, is that by using their real-time garbage collector there is no need for the stringent memory access rules imposed by the RTSJ. It is however possible to configure the JamaicaVM, to enable full RTSJ compliance. The JamaicaVM is given a grade of 2 for the support of the RTSJ.

Scheduling: The scheduler used in the JamaicaVM is a fixed priority preemptive scheduler as prescribed by the RTSJ. The scheduler supports round robin scheduling of threads with equal priorities. Threads are mapped directly to native threads and the JamaicaVM scheduler assists the underlying operating system in scheduling by choosing which Java threads appear eligible for execution. The JamaicaVM is given a grade of 3 for its real-time scheduling abilities.

Synchronization: The JamaicaVM supports both the priority inheritance protocol and the priority ceiling protocol in order to cope with the unbounded priority inversion problem (see section 2.3). Therefore the JamaicaVM receives a grade of 3 in this category.

Memory Management: The garbage collection algorithm used by the JamaicaVM is an incremental mark-sweep algorithm (see section 2.4). The heap is made up of small blocks of 32 bytes each. If an object does not fit inside one block then its memory usage can be spanned across several blocks. The memory blocks used by a single object do not need to be placed continuously, as the end of each block contains a pointer to the next block. This also helps avoid fragmentation of the heap, which is usually a problem with mark-sweep collectors. The JamaicaVM garbage collector works on these 32 byte memory blocks one at a time, and has no notion of Java objects. The worst case preemption time of the garbage collector is then the time it takes to mark and sweep a single memory block. The garbage collection algorithm is not executed within its own thread but inside application threads. The idea is that threads in need of memory must "pay" by doing an amount of garbage collection before it can allocate the memory it needs. In addition to the garbage collected heap the JamaicaVM also supports the scoped and immortal memory areas defines by the RTSJ. The JamaicaVM is given a grade of 3 in this category.

The conclusion of this analysis is that the JamaicaVM provides good features for achieving both soft and hard real-time performance. The additional support for the RTSJ ensures that the JamaicaVM is able to provide hard real-time guarantees.

Benchmark Results

The three benchmark tests were carried out on the Aicas JamaicaVM 6.1 with CD configured to use only RTSJ memory and threads. Table 6.8 shows the results of all three tests. The JamaicaVM achieves a very low rating in both peak and average performance, compared to both the PERC Ultra and the HotSpot. It is important to note that while both the PERC Ultra and the HotSpot are able to utilize both processing cores available in the test PC, the JamaicaVM (Personal Edition) can only use one[2]. This will explain some of the reduction in ops/m, but cannot account for the factor 30 that separates the PERC Ultra and the JamaicaVM. The lack of performance is also visible in test three where the (noise generating) SPEC benchmark is never scheduled for operations, as the CD benchmark (high priority) claims the CPU for the entire test.
The benchmark results support the qualitative analysis, stating that the JamaicaVM running RTSJ is able to achieve a high degree of timing predictability. This is illustrated by the low average jitter as well as the low standard deviation for both test two and three.

	Test One	Test Two		Test Three	
Average Performance	0,19 ops/m	-			0,00 ops/m
Peak Performance	0,19 ops/m	-			0,00 ops/m
Maximum Jitter	-	1,49 ms	2,98%	3,97 ms	7,94%
Average Jitter	-	0,01 ms	0,03%	0,01 ms	0,03%
Standard Deviation	-	0,04 ms	0,09%	0,20 ms	0,39%
Initilization Time	-		21,90 s		85,05 s

Table 6.8: Test results for Aicas JamaicaVM based on samples 500-5000

The jitter profile of test three on JamaicaVM is illustrated in figure 6.6. Small periodic deviations in can be observed, these may be ascribed to the lack of real-time scheduling between the operating system and the JVM, which is further described in appendix D. The JamaicaVM does however

[2]The available Personal Edition does not support parallel processing

allow for utilizing the special priorities of the underlying real-time scheduler in the operating system environment. For reference the test three were carried out using these real-time priorities, which resulted in even lower jitter values with an average of 0,01% as described in appendix D.

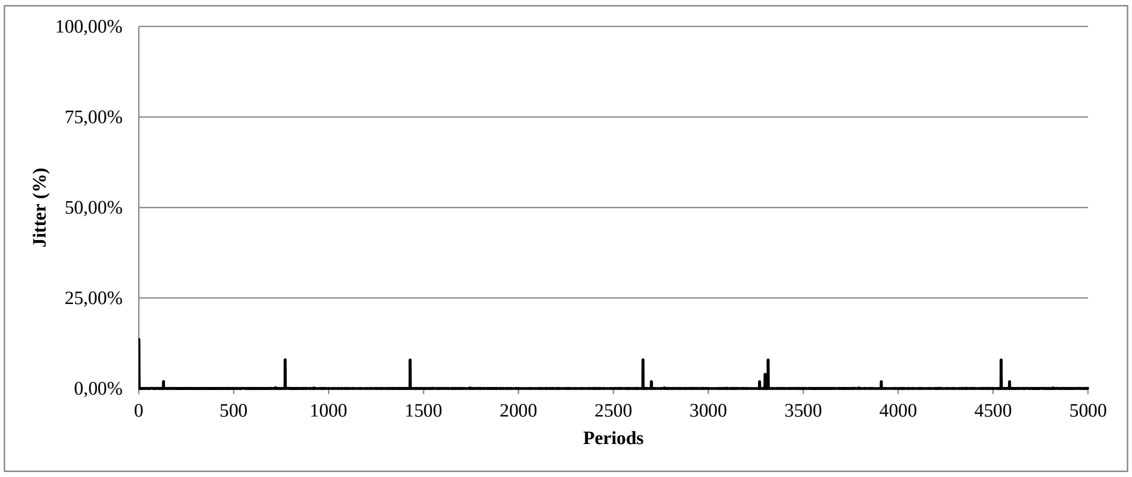


Figure 6.6: Jitter distribution for the JamaicaVM in test three with RTSJ memory and threads

The results for the JamaicaVM show, as expected, that the JVM performs worse in terms of throughput, but in return provides a highly predictable temporal behavior illustrated through the low jitter values. Aicas provides an optimization tool together with the JamaicaVM, called *JamaicaBuilder*, and it is expected that such a tool combined with a multi-processor edition will achieve a higher throughput in terms of ops/m.

6.5. Utilizing the Real-Time Specification for Java

Sub-step 3.3 of the TJARP method prescribes time critical parts of the application in question, to be rewritten for compliance with the RTSJ (see chapter 3. In order to evaluate the process of redesigning and rewriting an application for compliance with the specification, this section will take the Car Controller example through this process.
As the RTSJ is merely a specification, the JamaicaVM by Aicas (see section 6.4.5), which is RTSJ 1.0.2 compliant, has been used to test the application. Firstly the Car Controller example has been slightly redesigned in order to make use of the threads and memory areas defined by the RTSJ. The original design can be seen in appendix B (figure B.1, page 112) and the result of the RTSJ redesign can be seen here in figure 6.7.
The task of the `Navigation` thread is to update the in-car display. This is considered a soft real-time task (see chapter 4), hence it is implemented as a `RealTimeThread` (RT) which uses the heap memory area, as its operation requires a large amount of memory. The responsibility of the `CruiseController` thread is to monitor the speed of the car and alter it if needed. This is considered a hard real-time task and therefore it is implemented as a `NoHeapRealTimeThread` (NHRT), which is working within its own scoped memory area. This should protect the `Cruise-Controller` thread from delays caused by the garbage collector. The scoped memory area of the `CruiseController` thread is nested within another scoped memory area where the peripherals `Brakes` and `Engine` reside. These are used by the `BrakePedalEventHandler` and `GasPedalEventHandler` respectively. The two event handlers are implemented as instances

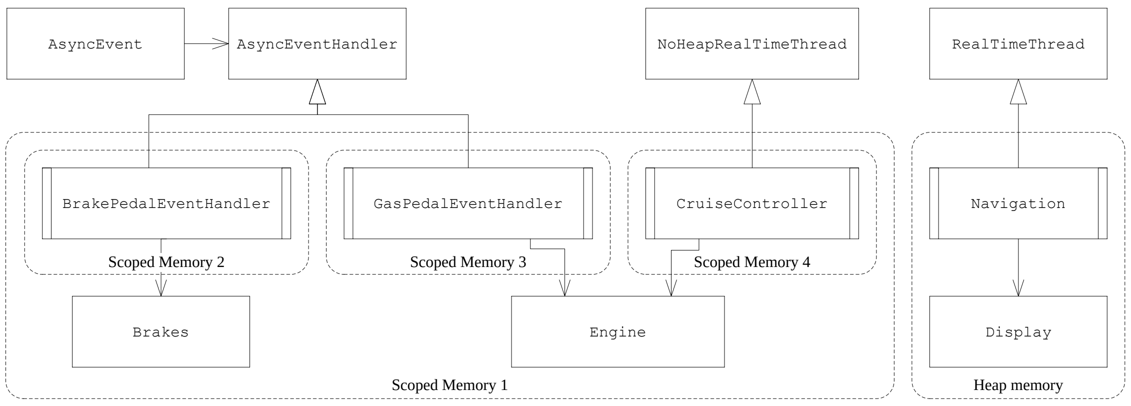


Figure 6.7: Car Controller example redesigned for RTSJ

of the `AsyncEventHandler` (AEH) class. Each executes within their own scoped memory areas, hence they should also be free from garbage collection delays. The two events `Brake-PedalEvent` and `GasPedalEvent` has been left out in figure 6.7 for the sake of simplicity, but they are implemented using the, RTSJ defined, `AsyncEvent` class which is used to trigger the AEH.
In section 6.5.1, 6.5.2 and 6.5.3 the RTSJ implementation of the Car Controller example will be tested within the three areas of scheduling, synchronization and memory management. These tests are similar to those used in chapter 2 for evaluating standard Java within the same three areas.

6.5.1 Scheduling

The result of the scheduling test (described in section 2.2.2) which examines the schedulers ability to respect thread priorities can be seen in figure 6.8. It is seen how the fixed priority preemptive

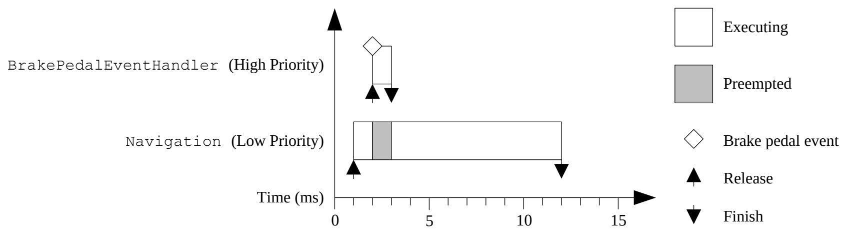


Figure 6.8: Scheduling in RTSJ

scheduler required by the RTSJ is working. It ensures that the low priority `Navigation` thread is immediately (within 0,1 ms) preempted, when an event from the brake pedal arrives and makes the `BrakePedalEventHandler` eligible for execution. For comparison, when the same test was run on standard Java (see section 2.2.2) a response time of 9 ms was measured, from the time of the event happening to the time when the handler was scheduled.

6.5.2 Synchronization

This test examines the priority inversion avoidance mechanism of the RTSJ. In section 2.3.2 it was illustrated how standard Java suffers from the problem of unbounded priority inversion. Figure 6.9 shows the same test performed using the RTSJ implementation. The priority of the

CruiseController has been lowered in this particular test in order to provoke the situation where priority inversion can arise, in all other tests the thread has a high priority. It is seen

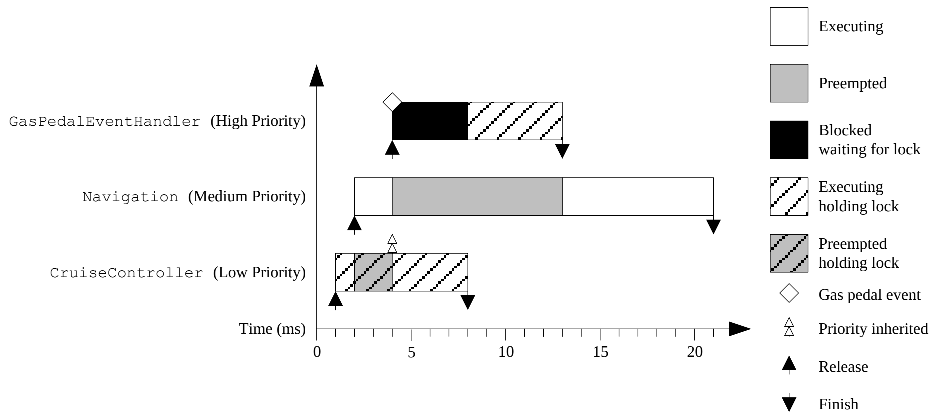


Figure 6.9: Synchronization in RTSJ

how the priority inheritance protocol required by the RTSJ solves the unbounded priority inversion problem. First the CruiseController thread gets scheduled and acquires the monitor lock on the shared resource (Engine). The scheduler then preempts the CruiseController thread before it can finish its job and release the lock, because the Navigation thread gets eligible for execution. Then an event from the gas pedal arrives, making the GasPedalEvent-Handler eligible for execution. However, the event handler cannot execute without acquiring the monitor lock on the Engine. Therefore, because of the priority inheritance protocol, the CruiseController thread inherits the high priority from the GasPedalEventHandler. This ensures that the CruiseController thread is able to run and hence releases the lock as fast as possible. As soon as the lock is released, the CruiseController loses the inherited priority and the GasPedalEventHandler starts executing. For comparison, when the test was executed on standard Java the GasPedalEventHandler could be delayed for the full 10 ms it takes for the Navigation thread to finish its operation.

6.5.3 Memory Management

The final test examines the delays incurred on the CruiseController thread when the Navigation thread is producing large amounts of garbage. Table 6.9 shows the results of a 10 minute test. For the sake of comparison, the results of the same test run on standard Java have been included. It is seen how the CruiseController thread is not affected by garbage collection as

Thread Period (ms):	50,00	**Maximum**	**Average**	**Std. Deviation**
RTSJ:	**Jitter (ms):**	0,10	0,01	0,01
	Jitter (%):	0,21%	0,02%	0,02%
Standard Java:	**Jitter (ms):**	73,26	0,76	2,81
	Jitter (%):	146,52%	1,52%	5,61%

Table 6.9: Jitter statistics for the CruiseController thread

it is implemented as a NHRT which cannot use the heap memory area. This significantly improves the threads predictability compared to the standard Java test, where the `CruiseController` thread experienced a delays of up to 73 ms.

6.6. Discussion

This chapter has introduced step 3 and 4 of the TJARP method, which are concerned with selecting and implementing the optimal strategy for achieving real-time performance. This chapter also presented important background knowledge, which should assist decision making when applying these steps. Among the presented results were an analysis of relevant JVMs, which provides an overview of currently available solutions.

Three different approaches towards achieving real-time performance has been described from section 6.3 to section 6.5. The results of each of these sections will be summarized and the approaches they describe will be discussed in the below sections 6.6.1 to 6.6.3.

6.6.1 Optimizing Standard Java

The first approach towards achieving real-time performance using Java, was presented in section 6.3 as sub-step 3.1 of the method. This approach is concerned with profiling the standard Java application and optimizing the code and the JVM accordingly. The result of using this approach on the Car Controller example revealed an improvement in performance and temporal predictability, however still no real-time guarantees could be given. This approach would be the first obvious step towards improving the predictability of an application, before system modeling. Hence, if predictable timing performance without real-time guarantees is sufficient according to the requirements, then step 2 of the TJARP would introduce a significant overhead and should be skipped.

An advantage of utilizing standard Java is that developers have access to a wide selection of different libraries and resources. When combining this with the superior performance of the Oracle HotSpot JVM or similar (see section 6.4), the developers benefit from the full power of the Java language. However as mentioned, this combination is unable to provide any real-time guarantees, as this is not the purpose of neither the language nor the JVM.

Therefore, this approach is only ideal for some types of applications with no hard real-time requirements. The authors believe that this approach should be used whenever the requirements allow for it.

6.6.2 Substituting JVM

For many systems the approach of optimizing standard Java is insufficient, because some degree of real-time guarantees is needed. Therefore, a second approach towards achieving real-time performance using Java, was presented in section 6.4 as sub-step 3.2 of the method.

This approach still uses the standard Java language, library API and memory model. However, real-time performance is achieved by rewriting the standard Java libraries with focus on temporal predictability and the JVM is equipped with a vendor specific real-time garbage collector. This approach is popular among commercial real-time JVM vendors, e.g. Aicas and Atego who support this approach through their products: JamaicaVM and PERC Ultra respectively.

The task of sub-step 3.2 is to choose a real-time JVM to substitute with the current JVM. In order to facilitate the choice of JVM, an analysis of quantitative and qualitative attributes were performed for each of them. The results of the analysis has been summarized and is illustrated in figure 6.10.

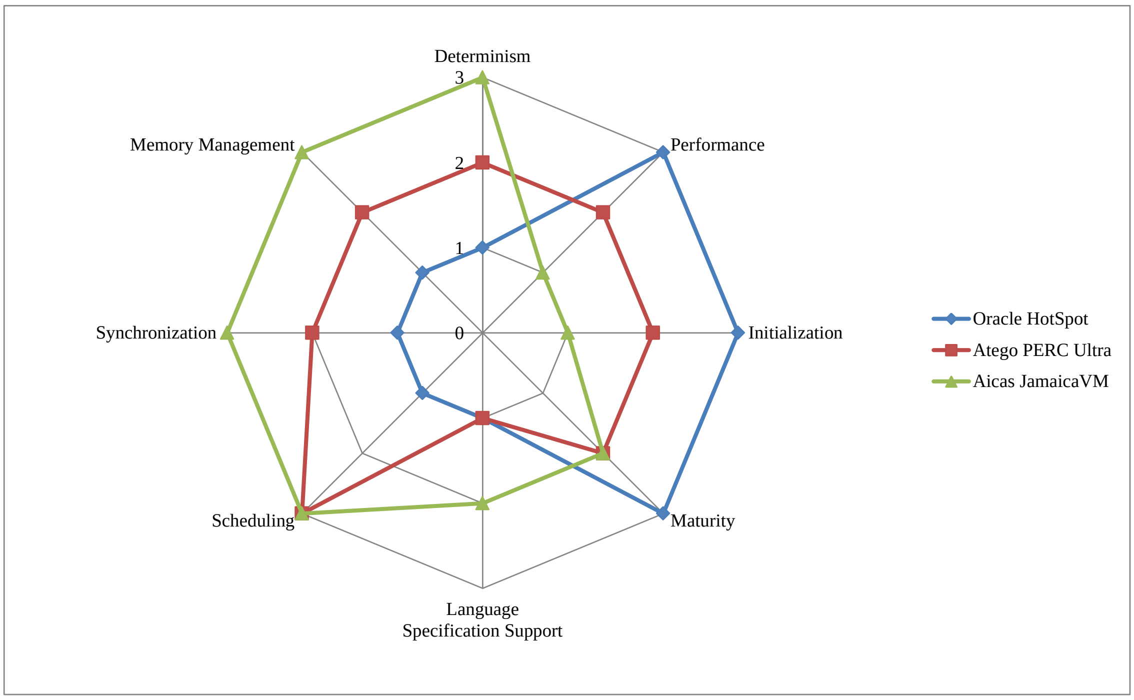


Figure 6.10: Comparison of the Oracle HotSpot, the Atego PERC Ultra and the Aicas JamaicaVM

The quantitative analysis was done by benchmarking the JVMs, and this is the basis for the parameters: *Determinism*, *Performance* and *Initialization*. From the benchmarking results the relation between these three parameters is seen. When the determinism of one of the JVMs increases, it has a negative effect on the performance and start up time of the JVM.

The qualitative analysis was done by analyzing features and characteristics of each JVM and grading them accordingly. From figure 6.10 it is seen how the JamaicaVM achieves the highest grades in *Memory Management*, *Synchronization* and *Language Specification Support* categories. The *Maturity* and the *Scheduling* are however estimated to be the same for the PERC Ultra and the JamaicaVM as they both support similar features for real-time within these areas.

To summarize, the HotSpot and the JamaicaVM represents two extremities, with performance at one end and real-time behavior at the other end. The PERC Ultra is an intermediate solution providing soft real-time guarantees, but at the price of reduced performance.

During the benchmarking of the JVMs it was hard to find a common configuration for all JVMs in order to do a fair comparison. As each JVM supports many individual parameters and has vendor specific features, the choice was to use the default settings and only adjust the parameters that the three JVMs have in common, e.g. total memory size etc. Therefore, it may be possible to achieve better benchmark results with all of the three JVMs by using vendor specific parameters and tools. For instance, Atego provides a tool called *PERC Accelerator* [Atego12], which can augment class files with native code or optimized byte code only compatible with the PERC Ultra. Similarly, Aicas provides a tool called *Jamaica Builder* [Aicas12], which is able to statically link all classes and compile them ahead-of-time. These tools can help improve startup and execution times significantly.

The qualitative analysis is based on a set of attributes and a grading scale set up by the authors. Some of the attributes were hard to measure e.g. maturity, therefore the grades was given based on judgment done by the authors. Other attributes were hard to compare e.g. the memory model and garbage collection strategy are very different, even between the two real-time JVMs (the JamaicaVM and the PERC Ultra). Therefore, in order to use the results of the analysis, it is important not only to inspect the grades given, but also the provided reasoning behind each grade.
An advantage of this approach, where the current JVM is substituted with a real-time JVM, is that existing Java code can be used directly, without rewriting it to make use of new thread types or memory models. However, a disadvantage is the real-time garbage collector algorithms, which need to be tailored towards the memory usage pattern of the specific application. This is done by analyzing and profiling the application and parameterizing the real-time garbage collector accordingly. Furthermore the complexity of the real-time garbage collectors makes it hard to analyze worst-case-execution-times and prove that deadlines will be met [Nilsen07]. The authors believe that this approach should be used when the requirements only prescribes soft real-time behavior and limited performance.

6.6.3 Applying the RTSJ

When soft real-time guarantees are insufficient, a third approach towards achieving real-time performance using Java can be used. This approach was presented in section 6.5 as sub-step 3.3 of the method.
Sub-step 3.3 relies on redesign and reimplementation of the application entirely or partly using the RTSJ. The real-time behavior obtained from doing this on the Car Controller example was highly predictable, and matched the theory of traditional real-time systems presented in chapter 2, as well as the theory of the RTSJ presented in chapter 3. The scheduling and synchronization mechanisms behaved predictably, and it was possible to create threads which were unaffected by garbage collection. However, the efforts spent, by the authors, to accomplish these results, using the RTSJ, was substantially larger than the effort associated with implementing sub-steps 3.1 and 3.2. This was due to the need for redesigning and rewriting the existing application entirely or partly. It is estimated that the required workload when using the RTSJ is larger than when working with standard Java or similar to that of implementing the same functionality using C or C++. The cause of the increased workload is mainly due to the introduction of additional memory areas by the RTSJ, and the memory access rules imposed on these.
The authors believe that this approach should be used when the requirements describe need for hard real-time performance and good reasons exist for not using traditional low-level programming languages.

Chapter 7

Case Study: Terma T-Core

This chapter applies the TJARP method to a complex industrial case study, the Terma T-Core framework, where each of the four steps of the method are followed as illustrated in figure 7.1. The case study defines the temporal requirements as described in chapter 4, then provide a model of the system as described in chapter 5. Finally the technical solutions discussed in chapter 6 are applied to the case study

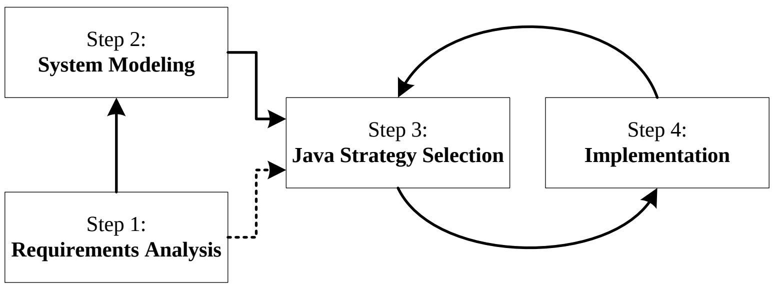


Figure 7.1: The four steps of the TJARP method

7.1. Introduction

This chapter evaluates the TJARP by applying it to the T-Core platform provided by Terma, both introduced in chapter 1. The T-Core platform consists of several distributed Java-based components, where the *Track Management* (TM) component is an essential part of the *threat evaluation* feature [Terma10]. The TM component is currently deployed in various mission critical systems, such as the *BMD-Flex International Air and Missile Defense Command and Control* system [Terma11] developed in cooperation with Lockheed Martin, and the *C-RAID Situational Awareness and C2 Control System* used for naval and coast guard monitoring [Terma12].
The TM component is able to provide a full situational picture of a specific geographical area based on inputs from available sensors and various data links. Detected physical objects are rep-

resented as radar *tracks* and are processed in order to correlate and fuse uniquely identify objects, detected by different sensors. Tracks added to the system, or tracks updated with new information, are published to subscribing entities, e.g. workstations in a *tactical command center* serviced by human operators.

Critical parts of the T-Core framework, including the TM component, are optimized for real-time performance by tweaking the runtime environment provided by the Oracle HotSpot JVM. However, the system suffers from the many temporal challenges applicable to Java systems as discussed in chapter 2. Terma wishes to reduce the workload associated with ensuring timing predictability of the TM component and investigate the possibilities for extending the system with sub-components constrained by hard real-time requirements. Currently the T-Core framework, and the TM component, does not negotiate track engagements by providing continuous track information to weapon systems. This is mainly due to strict timing requirements of the weapon interfaces. An example deployment of a system based on T-Core, with the TM component, an attached operator and a proof-of-concept weapon control system is illustrated in figure 7.2.

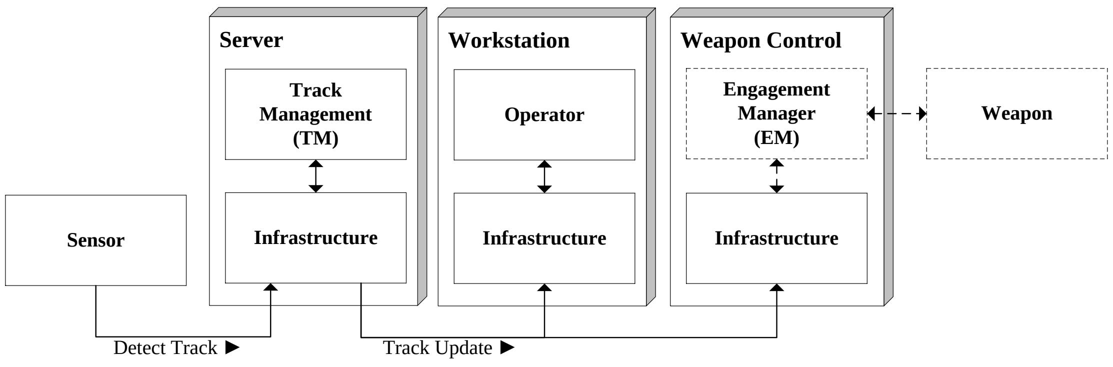


Figure 7.2: Components of the T-Core Case Study

The *Engagement Manager* (EM) illustrated in figure 7.2 is a fictional, but highly relevant, example of a hard real-time component attached to the non-real-time T-Core *Infrastructure* component. This allows the EM component to communicate with the network of distributed components within the T-Core system, including the TM component located on a central server. The EM component is marked with dotted lines to illustrate the additions to the existing system. The *Operator* component is included to show the concept of an extra non-real-time component. In order for the components to operate and utilize the features provided by the T-Core framework, additional components are included in the system such as the *Infrastructure* component.

The purpose of this case study is to apply the TJARP method to the T-Core framework, in order to introduce soft real-time performance to the TM component. Additionally the hard real-time proof-of-concept EM component is to be added, and should operate in cooperation with existing non-real-time components. The resulting system will include a mix of hard real-time (the EM component), soft real-time (the TM component) and non-real-time (Infrastructure and Operator). The first step of the method is described in section 7.2, the second in section 7.3 and the iterative steps 3 and 4 including the sub-steps are described in section 7.4. Finally the results of the case study is discussed in section 7.5.

7.2. Requirements Analysis

The first step of the TJARP method is to get a clear understanding of the requirements that the system is subject to (see chapter 4). The requirements for a system based on the T-Core framework vary depending on the type of system and customer. Furthermore, these requirements are usually company classified. Therefore, the requirements for this case study are fictional but still realistic. When following step 1, the first sub-step 1.1 is to determine the degree of real-time performance needed for the system. The T-Core framework is a large and complex platform, which would make the task of providing hard real-time guarantees across the entire system an immense task. Therefore, the TM component should at best be able to meet soft real-time requirements. The EM component however needs hard real-time performance to ensure consistent temporal behavior when evaluating threats and communicating with weapon systems. Hence, the T-Core platform needs mixed real-time performance. The functional requirements of the case study is described in section 7.2.1 and the non-functional requirements in section 7.2.1.

7.2.1 Functional Requirements

Sub-step 1.2 is concerned with clarifying the functional requirements for both the TM and the EM components. These have been elicited and listed below. The requirements have been simplified in order to keep focus on the real-time aspects of the system.

Track Manager

R1 The TM component shall be able to receive tracks from multiple sources.

R2 The TM component shall be able to correlate and fuse identical tracks received from multiple sources. (Soft real-time)

R3 The TM component shall be able to provide other components in the framework with track updates through the T-Core Infrastructure component.

R4 The TM component shall provide functionality for other components in the framework to subscribe and unsubscribe to track updates.

Engagement Manager:

R5 The EM component shall be able to subscribe to and receive track updates from the TM component through the Infrastructure component provided by T-Core.

R6 The EM component shall be able to evaluate track updates and determine if the track is a threat. (Hard real-time)

R7 The EM component shall be able to interface with up to ten weapons simultaneously. (Hard real-time)

R8 When the EM component identifies a track as a threat it must instruct a weapon, not currently in use, to engage the threat. (Hard real-time)

R9 The EM component shall be able to handle the situation when all weapons are currently in use by buffering and executing the threat engagement as soon as a weapon becomes available. (Hard real-time)

R10 The EM component must communicate with the weapon through a predefined protocol consisting of 10 consecutive messages sent at a fixed interval. (Hard real-time)

R11 The EM component shall not terminate communication with a weapon while negotiating a track engagement. (Hard real-time)

7.2.2 Non-Functional Requirements

The task of the final sub-step 1.3 is to define a set of non-functional requirements describing the real-time properties of the system. If the non-functional requirement is directly related to a functional requirement is has been given a similar identifier. The non-functional requirements for the TM and the EM components are listed here:

Track Manager

R3.1 The time from a track is received by the TM component and delivered to an operator situated on the same node must not exceed [redacted] ms, given a maximum of 5 operators.

R3.2 Requirement R3.1 must still hold while the TM component is handling 200 track updates per second.

Engagement Manager:

R6.1 The time from a track update is received until the threat evaluation is finished must not exceed [redacted] ms.

R8.1 The time from a track has been identified as a threat and until the weapon communication is initiated must not exceed [redacted] ms, if a weapon is available.

R10.1 The 10 consecutive message comprising the weapon communication protocol must be sent with a period of [redacted] ms.

R10.2 For requirement R10.1 the minimum tolerable time between to messages is [redacted] ms.

R10.3 For requirement R10.1 the maximum tolerable time between to messages is [redacted] ms.

R11 In accordance with requirements R6.1, R8.1 and R10.2 the time from a track update is received until the weapon communication finishes must not exceed [redacted] ms. (Hard real-time)

R12 Timing requirements R6.1, R8.1, R10.1, R10.2, R10.3 and R11 must still hold if the EM is deployed on the same node as the TM, while the TM is component is handling 200 track updates per second. (Hard real-time)

The real-time requirements leave out details about communication timing through the distributed infrastructure of the T-Core framework. This is done deliberately as it is beyond the scope of this thesis to consider real-time performance across distributed nodes (see section 1.3).
The timing requirements for the TM component are categorized as soft real-time. This is because the requirements are concerned with delivering information to human operators, where a track update delayed for a period of milliseconds would go unnoticed.
In contrast, the timing requirements for the EM component are considered hard real-time as the failure of communicating with a weapon and hence eliminating a threat could be catastrophic.

Requirements R3.2 and R12 are concerned with providing real-time guarantees on a single node, even though the system is under a considerable load. The ability to mix both real-time and non-real-time components on the same node while still maintaining real-time guarantees has been pointed out as a key feature by Terma. Hence, the system must be operational and comply with the requirements, while both the TM and EM deployed at the same processing node.

7.3. System Modeling

The T-Core platform is highly complex and consists of several thousand Java classes, where each component is dependent on several others, e.g. the TM component rely on Infrastructure component for communication. The task of gaining an overview of this large codebase, and especially understanding its temporal characteristics, can be overwhelming. This second step of the TJARP method prescribes an analysis of the application with focus on the timing requirements, supported by the formal modeling language VDM-RT.
The four sub-steps within this step, allows for gaining an understanding of the important parts and aspects of the system by raising the level of abstraction, and thus ignoring implementation details which do not affect the purpose of the model. Therefore, before proceeding to development of the model, a clear definition of the purpose must be defined. For the T-Core model the purpose states: *To provide an understanding of the temporal attributes of the Track Management component, and investigate possible design options for extending the system with an additional hard real-time component. The model shall help identify possible design pitfalls with respect to the timing requirements.*
This section applies the four sub-steps, of step 2, to the T-Core TM component and the EM component. Sub-step 2.1 models the overall system structure and is described in section 7.3.1. Sub-step 2.2 introduces concurrency to the model and is described in section 7.3.2. Sub-step 2.3 expands the model further by adding timing constraints as described in section 7.3.3. Finally the architectural design is explored in sub-step 2.4, described in section 7.3.4.

7.3.1 Modeling System Structure

To gain an understanding of the functional behavior of the TM component within the existing T-Core system, the overall structure is mapped directly to VDM-RT classes. However, as the purpose describes, the model must keep focus on temporal attributes, hence several implementation details can be ignored as the level of abstraction is raised.
The existing TM implementation acts as a central processing component, with several *publish-subscribe* mechanisms, i.e. the distribution of track information to subscribing listeners. The technical implementation of these mechanisms are not the focus of the model, however the functional behavior (the distribution of updates) is to be included to correctly asses the overhead of simultaneous processing in multiple listeners.
The TM component implements complex track-correlation and track-fusion algorithms which for the model are merged into one simplified evaluation algorithm, in order to emulate processing for each track-update.
The EM component must be able to isolate the weapon communication in a central class, in order for modeling a hard real-time sub-component. The EM component shall make use of existing T-Core functionality for subscribing to track updates, through the Infrastructure component. The component must also support multiple weapon interfaces simultaneously.
The structure of the model is illustrated in the simplified class diagram in figure 7.3. Notice that

active classes are marked in the diagram even though these are not identified until sub-step 2.2. The nodes (Sensor, Weapon, Workstation etc.) are included to provide a distribution overview, where the classes within the Server-node are all directly mapped from the existing T-Core Java classes. The top-level classes `World` and `TCoreEnvironment` are added to the model to control the configuration and execution of different scenarios.
Scenarios of the model are triggered by the `TrackSensor` reading a file with a predefined set of events which are processed and sent to the `TrackManager` through the `TrackReceiver` class. The `TrackManager` does a simplified track correlation and publishes the update to all subscribing instances of the `TrackListener` class. The sub-classes define operations to be invoked upon track updates. The `Operator` models a graphical presentation of track information emulating an operator workstation. The `TrackEvalutationHandler` (TEH) evaluates the track and if deemed hostile and positioned inside a pre-defined geographical area, commences the engagement.

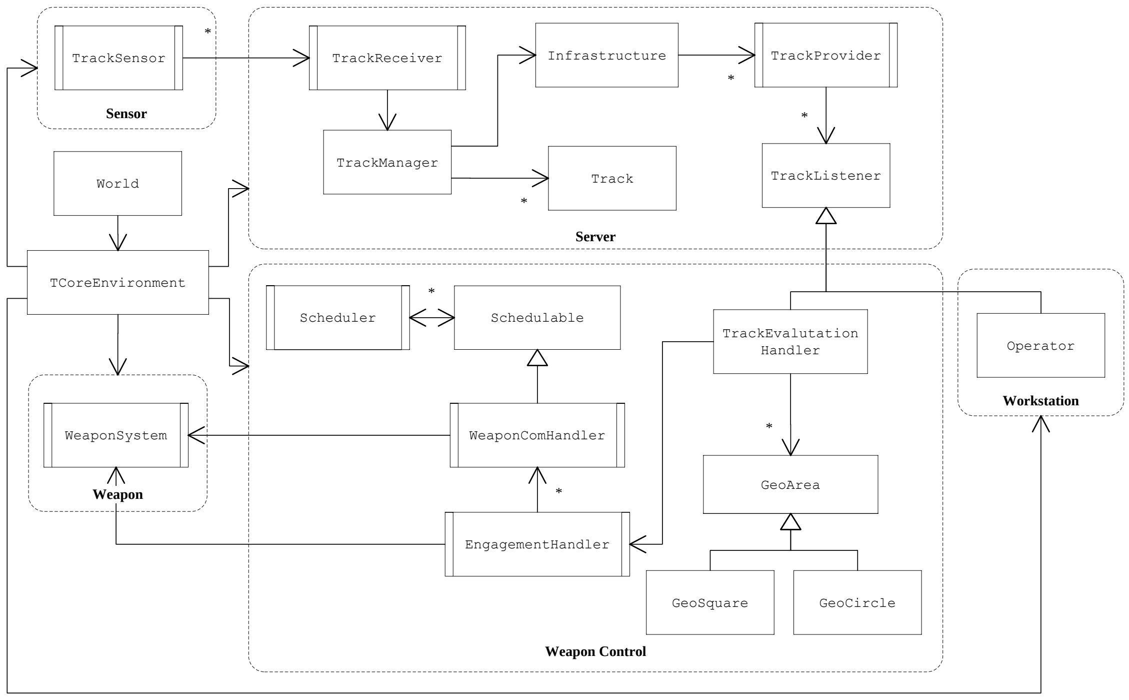


Figure 7.3: Simplified class diagram of the T-Core VDM-RT model

At this stage the model is implemented to provide the basic functional behavior, and provides a foundation for assessing possible design solutions. Especially the added functionality of the EM component is to be detailed further by imposing functional constraints directly within the model. This is possible by use of VDM *invariants* and *pre-conditions*. Listing 7.1 shows an example of such a pre-condition in the `EngagementHandler` class. The pre-condition maps directly to requirement R11 (see section 7.2) and provides a run-time check to ensure weapons systems are not removed from the system while they active.

```
public RemoveWeaponSystem : WeaponSystem ==> ()
  RemoveWeaponSystem(w) == weapons := {w} <-: weapons
pre w in set dom weapons and not weapons(w).IsActive()
```

Listing 7.1: Precondition within the `EngagementHandler` to ensure that no active weapons are removed

The *VDMUnit* unit-test framework is utilized to increase confidence within the functionality of the individual classes. These unit-tests together with the pre-conditions, invariants and runtime type-checks provide a high level of confidence in the functionality. From the model, at this stage, it became clear that the ability to handle multiple engagements concurrently would require extra care in order to meet the hard real-time requirements of the weapon communication. This is described by requirement R7 (see section 7.2) which states that simultaneous communication with multiple weapons must be supported while still complying with the timing constraints for the weapon communication.

7.3.2 Introducing Concurrency

Sub-step 2.2 motivates the introduction of concurrency, by identifying and creating active classes within the model. This increases confidence in the temporal behavior of the model as it moves closer to its defined purpose. Figure 7.3 shows how several classes are marked as active, which is achieved by explicit thread definitions or by the usage of the **async** keyword. The ability of the TM component to publish track updates has increased in complexity by the introduction of concurrency, with use of the `Infrastructure` class to concurrently notify subscribing instances of `TrackProvider`. These notifications creates new procedural threads for processing operations implemented within each subscribing `TrackListener` instance. By introducing concurrency it became clear that the process of receiving a track update in the TEH, and commence communication with the weapon, required extra attention as tracks may be published faster than they are communicated to the weapon. The weapon communication was subject to hard real-time requirements, and thus isolated within a single **periodic** thread implementation in the `WeaponComHandler` (WCH) class.

Synchronizing Engagement

The requirements R7 and R9 (see section 7.2) states that multiple engagements must be supported. The initial design suggested that the TEH should start the periodic WCH thread once a threat has been deemed hostile and detected within the protected geographical area. However, the model identified the engagement procedure as a bottleneck in this design. If all available weapons are in use when a track is to be engaged, the procedural behavior of the TEH thread would require some form of buffering mechanism to make sure that a track is not missed, and that the weapon is notified as soon as the previous operation is completed.

Therefore, an additional active procedural thread, the `EngagementHandler` was introduced with the responsibility of buffering tracks to engage and synchronize with available weapons in order to start the required instances of WCH. Figure 7.4 illustrates the scenario of receiving a hostile track and commencing the engagement communication with the weapon.

When a track is to be engaged, the `EngagementHandler` must check for available weapons and pass the weapon to the corresponding WCH before activating the communication. The process of checking for available weapons requires extra care, and is implemented by use of VDM-RT history

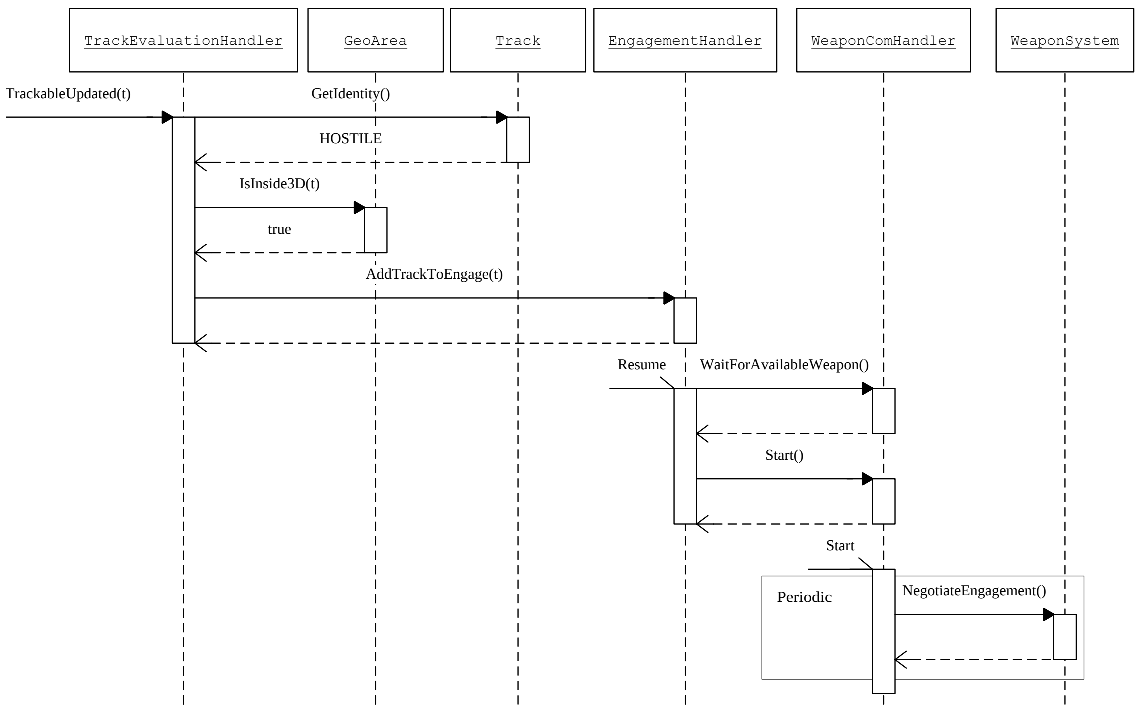


Figure 7.4: Sequence diagram of a track updated and engaged

counters as illustrated in listing 7.2. The instance variable `availableWeaponCount` is marked as static and decremented every time an instance of WCH is activated and incremented once it is finished. The permission predicate makes sure that the operation `WaitForAvailableWeapon` blocks until at least one weapon is available.

```
public WaitForAvailableWeapon : () ==> ()
WaitForAvailableWeapon() == skip;

sync
mutex(SetAsUnavailable,SetAsAvailable);
per WaitForAvailableWeapon =>  availableWeaponCount > 0;
```

Listing 7.2: Permission predicate for weapon availability

Scheduling Communication

In order for the WCH to guarantee the periodic timing requirements (see requirement R10.1 – R10.3 in section 7.2) it must support a fixed number of periodic iterations from it is started, and terminate upon completion. However, such behavior is not supported (by default) by the periodic or procedural threads in VDM-RT. The procedural thread does not support periodic invocations or pausing for specific time intervals, and the periodic thread does not allow for continuously starting and stopping.
The solution was to model a timer-based scheduling mechanism with the `Scheduler` class (see figure 7.3). The class implements a periodic thread, which increments an instance variable once

every millisecond. At each increment, the thread checks to see if any `Schedulable` instances are eligible for execution at that specific time, and if so unblocks the associated procedural thread. This scheduling mechanism is supported by an advanced use of permission predicates within the `Schedulable` class.
Classes inheriting from the `Schedulable` class, specifies a period within their constructor for which they are to be blocked when invoking the derived operation `WaitForNextPeriod`. The operation will add the caller to the `Scheduler` and wait for the remaining time interval since the last release. For example, an instance of the `Schedulable` class, with a period of 5 ms, which invokes the `WaitForNextPeriod` operation within this period, will be scheduled at time intervals 5, 10, 15 etc. The periodic check within `Scheduler` class is illustrated in listing 7.3 and is described in further details in appendix B.

```
private CheckSchedulables : () ==> ()
CheckSchedulables() ==
(
 if(timeUnit in set dom schedulables
 and schedulables(timeUnit) <> {}) then
  (
   for all s in set schedulables(timeUnit) do s.Release();
   schedulables := {startTime,...,timeUnit} <-: schedulables;
  );
);
```

Listing 7.3: Periodic check for waiting `Schedulable` instances

The implementation of the WCH through the `Scheduler` and `Schedulable` classes allow for detailed modeling of the periodic behavior of the RTSJ defined `RealTimeThread` (see section 3.2).
By the introduction of concurrency, the level of detail within the model is increased with respect to the purpose of the model. The model for identifying potential deadlocks or overlap of periodic threads through run-time errors printed directly within the Overture debugger.

7.3.3 Analyzing Timing Constraints

Sub-step 2.3 encourages the use of specific VDM-RT semantics to help identify timing pitfalls with respect to the purpose of the model. The `Operator` class is extended with the notion of time by using the **`duration`** keyword. This is used to model the computational heavy operation of updating the information on an operator workstation.
The EM component requires extra focus, where the hard real-time requirements are candidates for runtime checks with validation conjectures. The requirements R10.2 and R10.3 describe maximum and minimum separation between communication messages to the weapon system. In listing 7.4 the `deadlineMet` and `separate` conjectures are used to constrain the model with regards to these requirements. The conjectures prove a valuable tool, as the model at this in this sub-step does in fact violate all three conjectures.

```
/* timing invariants
deadlineMet(#act(WeaponComHandler`Run),
            #fin(WeaponComHandler`Run), 50 ms);
deadlineMet(#act(EngagementHandler`AddTrackToEngage),
            #act(WeaponComHandler`NegotiateEngagement), 1 ms);
separate(#act(WeaponComHandler`NegotiateEngagement),
         #act(WeaponComHandler`NegotiateEngagement), 4 ms);
*/
```

Listing 7.4: Validation conjecture for the `WeaponComHandler`

By using the RT Log Viewer (RTLV) of the Overture tool is it possible to identify the exact cause of the violation (see section 5.3). The RTLV allows for a detailed view of each CPU to show how processing and scheduling of threads are progressing through time. For the EM component the conjecture violation was identified to be caused by the scheduling of the `Operator` instance before the `TrackEvaluationHandler` thus delaying the engagement of tracks with the duration of the `Operator` operations.
The jitter values specified for the periodic `Scheduler` thread was also identified as a cause of conjecture violation. By specifying a jitter value of 0.5 ms, the theoretically maximum deviation is 5 ms (the WCH is scheduled 10 times), thus potentially violating all three conjectures as this affects the total execution time of the `WeaponComHandler`Run` operation. The periodic definition of the `Scheduler` is illustrated in listing 7.5.

```
thread periodic(1E6,5E5,0,0)
    (IncrementTime)
```

Listing 7.5: Periodic invocation of the `Scheduler` with jitter

The conjectures helped identify critical sections of the model as potential causes of temporal non-determinism, and proved how the model serves as a valuable input before proceeding to the actual implementation. It also illustrates how the temporal unpredictability within Java systems can be modeled.

7.3.4 Design Space Exploration

Through the previous three sub-steps, the EM component was shown to be under strict timing constraints and even unable to guarantee the timing requirements of the weapon communication. The final sub-step 2.4 prescribes a design space exploration. By utilizing the distributed modeling features of VDM-RT the entire EM component can be deployed on a separate `CPU` and thus be unaffected by scheduling of the `Operator` class. The declaration of the `CPU` for both the TM and EM components is illustrated in listing 7.6. The declaration of the `BUS` instance show how multiple CPUs are connected through the same bus.

```
static public CPUServer: CPU := new CPU (<FP>, 1E9);
static public CPUWeaponControl : CPU := new CPU(<FP>,1E9);

static public trackBus : BUS := new BUS(<CSMACD>, 72E3,
{CPUServer, CPUWeapon, CPUSensor, CPUWeaponControl})
```

Listing 7.6: Declaration of CPU and BUS for deployment

This distribution will cause the `TrackProvider` to invoke the update operations of the `Track-Listener` instances through the communication bus. Causing the processing of a track-update within `Operator` and the WCH to be done in parallel at two different CPUs. The deployment is illustrated in listing 7.7.

```

CPUServer.deploy(manager,"TrackManager");
CPUServer.deploy(infrastructure, "Infrastructure");
CPUServer.deploy(operator, "Operator");

CPUWeaponControl.deploy(evaluationHandler, "EvalutationHandler");
CPUWeaponControl.deploy(scheduler,"Scheduler");
```

Listing 7.7: Deployment of classes within the `CarEnvironment` constructor

This new architecture ensured that the EM component were able to meet the timing requirements elicited in step 1.

7.4. Java Strategy Selection and Implementation

Step 3 and 4 of the TJARP method is concerned with selecting and implementing the appropriate Java real-time strategy for meeting the real-time requirements. As described earlier, the T-Core case study requires mixed real-time performance as the TM component is subject to soft real-time requirements, while the EM component is subject to hard real-time requirements. Therefore the development of the two components will be split in two, according to step 3 of the TJARP method (see chapter 6). Hence, the TM component will start at sub-step 3.1, while the EM component will start at sub-step 3.3 due to its hard real-time requirements. The process of applying step 3 and 4 to the TM component will be described in section 7.4.1, while section 7.4.2 describes how the steps were applied to the EM component.

7.4.1 The Track Management Component

Sub-step 3.1 of the TJARP method recommends optimizing the existing application and JVM in order to improve real-time performance. This sub-step will not be part of this case study, as this approach is already applied by Terma before deploying a new configuration of the T-Core platform.
In order to investigate the possibilities for achieving further real-time performance, the TM component was subject to sub-step 3.2 of the TJARP method. This sub-step is concerned with substi-

tuting the JVM. The Atego PERC Ultra was chosen, to replace the Oracle HotSpot currently used, based on the analysis and comparison of JVMs presented in chapter 6. An advantage of using the PERC Ultra is its support for JSE 1.6, which in theory should allow the T-Core framework, including the TM component, to run on the PERC Ultra without modifications. In practice however, substituting the JVM was not that simple. Based on the choice of strategy in step 3, the case study moves to step 4 of the TJARP method.

7.4.1.1 Substituting the JVM

Sub-step 4.1 includes the task of porting relevant T-Core components to the PERC Ultra. Here a number of challenges were faced. For instance, the framework failed to execute because of an error during the initial class loading phase. The PERC Ultra applies a different class loading strategy compared to the Oracle HotSpot. This caused errors as some classes could not be found, and some libraries were incompatible with the PERC Ultra. However, the class libraries causing the problems were not used by the particular T-Core configuration used for this case study, hence they could easily be removed.
Eventually the port succeeded, and the T-Core components were executing on the PERC Ultra. The next task was to fine tune the parameters of the PERC Ultra such as adjusting the memory parameters and garbage collection settings.

7.4.1.2 Evaluating the Results

Sub-step 4.2 was concerned with evaluating the results of porting the T-Core components to the PERC Ultra with regards to the timing requirements outlined in section 7.2. In order to test requirements R3.1 and R3.2, a series of tests were set up where the system was stressed by forcing it to handle different amounts of simultaneous tracks. The amount of track updates per second was varied from 30 and up to 200. For each test a series of additional hostile track updates was sent into the TM at 4 updates per second. It was then measured how much time passed between a hostile track being updated with a new position, until this information was received by an operator. In between these two events the updated track information would pass through the TM component. The results of the tests can be seen from the chart in figure 7.5. The same tests performed on the HotSpot JVM have been included for comparison.

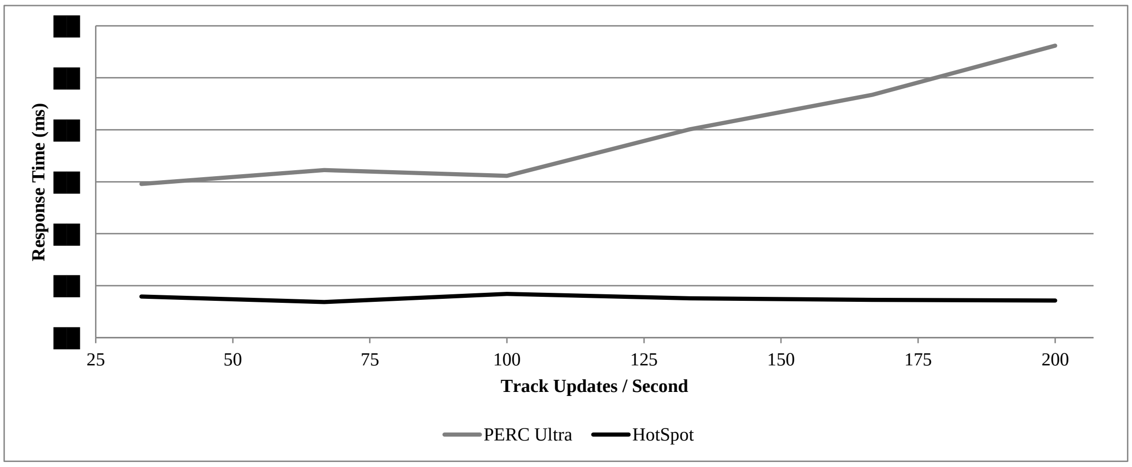


Figure 7.5: Track Load Tests - PERC Ultra vs. HotSpot

The horizontal axis shows the amount of track updates per second the system is handling, while the vertical axis indicates the average time for sending a hostile track update through the system. This average time is based on measurements of 2000 hostile track updates sent through the system. It is seen how the response time is relatively constant for both JVMs when receiving between 30 and 100 track updates per second. As expected from the benchmark results presented in chapter 6, the PERC Ultra performs slower than the HotSpot. This is not a problem for this case study as it is the predictability of the response time which is important. However, as the amount of track updates per second rises above 100, the response time starts increasing for the PERC Ultra, while the HotSpot is still maintaining a constant response time. It is seen how the PERC Ultra violates the maximum response time of [redacted] ms while receiving 200 track updates per second, as stated by requirements R3.1 and R3.2.
The performance of the PERC Ultra is therefore insufficient for the TM component executing on the particular configuration of the T-Core platform used for this case study. The same tests could also have been performed on the JamaicaVM, however the benchmarks result, presented in chapter 6, shows that the JamaicaVM has worse performance than the PERC Ultra. Therefore, the effort of porting the T-Core platform to the JamaicaVM was skipped.
Because of the limited performance of the PERC Ultra, the HotSpot was re-introduced for this particular deployment of the TM component. However, other systems based on the T-Core platform and with less strict requirements on performance, could be able to utilize the PERC Ultra. The authors believe that the performance requirements could have been met by further optimization of the PERC Ultra, e.g. by utilizing the *ROMizer* or *PERC Accelerator* provided by Atego.

7.4.2 The Engagement Manager Component

The EM component is subject to hard real-time constraints and therefore, the first two sub-steps of step 3 can be skipped. The following sub-step 3.3 prescribes that the hard real-time part must be rewritten for compliance with the RTSJ in order to achieve hard real-time performance. As the EM component is a newly added component in the system, there is no code to rewrite, hence the design and development of the component, begins here. The experience gained from modeling the EM component in step 2 serves as a good starting point for the design.

7.4.2.1 Applying the RTSJ

During sub-step 3.3, the EM component was designed to make use of the additional thread types and memory areas provided by the RTSJ. The result can be seen from the class diagram in figure 7.6.
When a track is updated with new information, e.g. position, the asynchronous event `Track-UpdateEvent` triggers the `TrackEvaluationHandler`. If the track is hostile and inside the protected geographical area then it is queued for the `EngagementHandler` to handle it. The `EngagementHandler` then locates a weapon and spawns a new `WeaponComHandler` thread, which performs the communication with the weapon.

7.4.2.2 Implementing the Engagement Manager Component

The next sub-step to apply was 4.1, where the RTSJ implementation was developed. The strict rules on memory areas, as described in section 3.2.3, proved to be a challenge when working with the RTSJ. For instance, the task of passing data between the `EngagementHandler` and the `WeaponComHandler`, which resides in different memory areas, were not as simple as just passing an object reference. Instead each piece of data within the object had to be explicitly copied

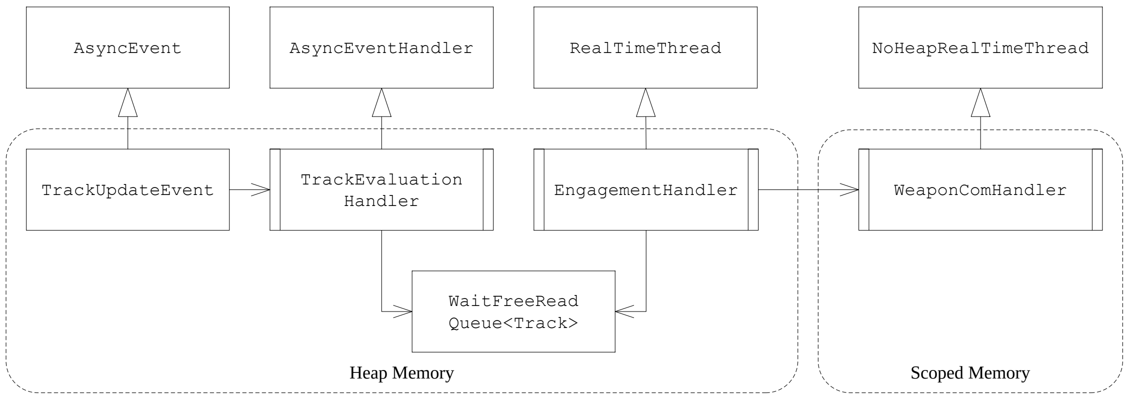


Figure 7.6: Engagement Manager Class Diagram

from one memory area to the other.

Listing 7.8 provides a code example illustrating how the `TrackEvaluationHandler` was implemented.

```
1  private class TrackEvaluationHandler extends AsyncEventHandler{
2   public TrackEvaluationHandler(){
3    super(new PriorityParameters(15),
4     null, null, null, null, false);
5   }
6
7   public void handleAsyncEvent(){
8    Track track = (Track)trackEvaluationQueue.read();
9    trackEvaluationStart[currentIteration] = System.nanoTime();
10   if( track != null &&
11       track.getIdentity().getValue() == TrackIdentity.HOSTILE &&
12       aCircle.isInside3D(track.getPosition()) ){
13    trackEvaluationEnd[currentIteration] = System.nanoTime();
14    currentIteration++;
15    engageTrack(track.getPosition());
16   }
17  }
18 }
```

Listing 7.8: Java code from the RTSJ implementation of TrackEvaluationHandler

Listing 7.8 shows how the `TrackEvaluationHandler` is given a higher priority than the 10 available in standard Java (line 3). The `handleAsyncEvent` is executed whenever a `Track-UpdateEvent` is triggered (line 7). The function retrieves the track which should be evaluated and checks if it is inside the protected geographical area, by utilizing functionality provided by the T-Core framework (line 10-12). If the hostile track is inside the geographical area it is queued for handling by the `EngagementHandler` by using the `engageTrack()` method (line 15). Further implementation details of the EM component is provided in appendix B.

7.4.2.3 Evaluating the Results

During sub-step 4.2 it was evaluated whether the EM component was able to meet the timing requirements set up in section 7.2. The JamaicaVM was chosen for testing the component, based on the real-time performance displayed through the analysis done in chapter 6. The test was performed while the TM component was deployed on the same node as the EM component, in accordance with requirement R12. The TM component was executing using the HotSpot JVM, while the EM component was executing using the JamaicaVM. The test would reveal if the RTSJ-based EM component would be able to meet its timing guarantees, even though the TM component was handling 200 track updates per second, on the same node. This should work in theory as the JamaicaVM is able to utilize an extended priority range reserved for real-time processing in the underlying operating system. These priorities are not available to the HotSpot JVM, hence the scheduler of the operating system should choose the EM component for execution in favor of the TM component.

The detailed timing results of the EM component executing on the JamaicaVM can be seen in table 7.1, for comparison the timing requirements from section 7.2 has been included in the table. The results are based on measurements of 2000 hostile tracks sent through the system, each separated by 0,25 seconds. To illustrate the different timing intervals during a track evaluation and engagement figure 7.7 is provided as an example scenario.

	Average	**Max**	**Std. Dev.**	**Required**
Total Handling Duration (ms):	[redacted]	[redacted]	[redacted]	≤ [redacted]
Evaluation Duration (ms):	[redacted]	[redacted]	[redacted]	≤ [redacted]
Weapon Communication Total Duration (ms):	[redacted]	[redacted]	[redacted]	[redacted]
Weapon Communication Period (ms):	[redacted]	[redacted]	[redacted]	[redacted]
Evaluation/Communication Delay (ms):	[redacted]	[redacted]	[redacted]	≤ [redacted]

Table 7.1: Engagement Manager Test Results, using the JamaicaVM

Total Handling Duration indicates the time spend from the `TrackUpdateEvent` is triggered until the `WeaponComHandler` has finished communicating with the weapon. The *Evaluation Duration* shows the amount of time the `TrackEvaluationHandler` spends doing the mathematical calculations, which checks if the hostile track is inside the protected geographical area. *Weapon Communication Total Duration* indicates the time used by the `WeaponComHandler` for sending the 10 consecutive messages, separated by [redacted] ms each (See requirement R10). The *Weapon Communication Period* shows the actual period used for sending the 10 consecutive messages. Finally, *Evaluation/Communication Delay* indicates the time it takes from a hostile track has been detected inside the protected area until the `WeaponComHandler`, which is spawned by the `EngagementHandler`, starts communicating with the weapon.

Figure 7.8 illustrates the Total Handling Duration for each of the 2000 tracks used for the test. Similar graphs for the remaining timing measurements can be found in appendix B.

It is seen how the total handling time for a track is mostly between [redacted] and [redacted] ms. However, outliers exist where the handling time reaches [redacted] and [redacted] ms. When the graph in figure 7.8 is compared to the graph showing the *Evaluation/Communication Delay* (figure B.5, page 120), it is clear that the spikes are caused in this particular time span. This is most likely due to overhead introduced when spawning a new thread. This problem could be avoided by having a pool of `WeaponComHandler` threads, which are ready to do communication instead of spawning a new thread every time communication is needed.

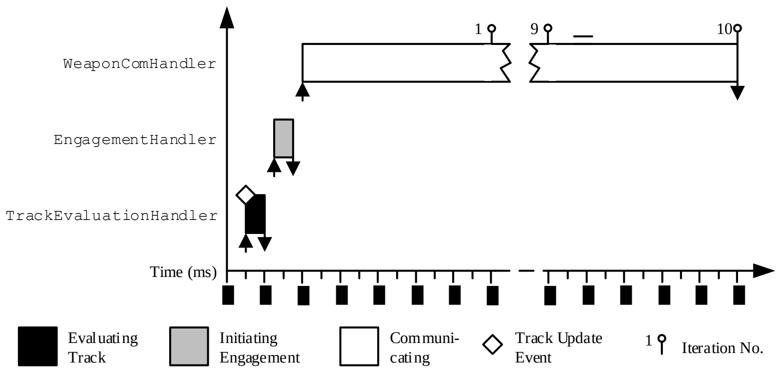


Figure 7.7: Ideal Timing for the Engagement Manager Component

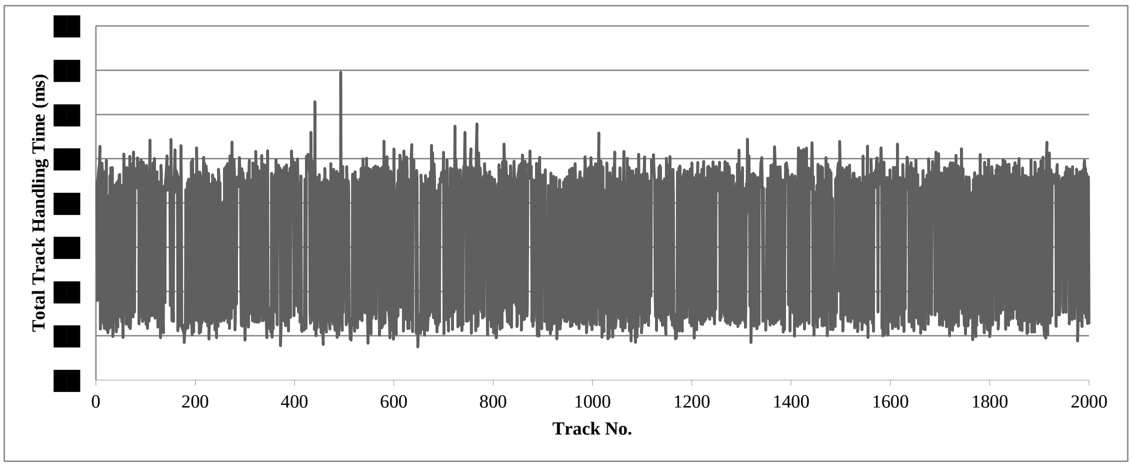


Figure 7.8: Total Handling Duration for the Engagement Manager Component

All of the results satisfy the requirements for the EM component stated in section 7.2, except requirement R8.1. This requirement is concerned with the interval *Evaluation/Communication Delay* in table 7.1, which must not exceed [redacted] ms. Hence this initial test was only partially successful.

7.5. Discussion

This chapter described how the TJARP method was applied to a complex industrial case study: The Terma T-Core framework. The case included real-time optimization of the existing TM component, and the addition of the new EM component with hard real-time requirements.

The first step of the method resulted in a clear definition of requirements and in particular the important timing requirements. These would later serve as guiding principles when making decisions in the followings steps. The timing requirements were also used to measure the success of the systems real-time behavior during evaluation. Furthermore, the characterization of timing requirements presented through chapter 4, served as a starting point for eliciting concrete timing requirements together with Terma. Hence, it was clarified for both parties, what was expected

from introducing real-time performance in the T-Core framework.
The second step of the method included modeling the system and resulted in the authors gaining an overview and understanding of the large code base, which comprises the T-Core framework. Also potential bottlenecks for the EM component were identified and mitigated through design space exploration leading to the initial system architecture. The model confirmed that the hard real-time EM component needed to be isolated from the rest of the system in order to meet its deadlines. Also the model helped introduce an additional handler thread into the component, which helped minimize delays when several weapons were controlled simultaneously.
The third step of the method resulted in a choice of real-time java strategy, for both the TM component and the EM component of the Terma T-Core framework. The choices were supported by the timing requirements elicited in step 1. The strategy was for the TM component to be ported to the PERC Ultra in order to introduce soft real-time performance. The strategy for the EM component was to implement it using the RTSJ, in order to achieve hard real-time performance.
During the fourth step, the TM component was successfully ported to the PERC Ultra JVM, and the EM component was implemented using RTSJ. By doing this, it was revealed that the performance of the PERC Ultra was unable to execute the TM component under the large workload as specified by the requirements. The EM component implemented using the RTSJ provided the desired real-time guarantees, except a single requirement (R8.1) which was violated by [redacted] ms. However, the authors believe that the proposed solution of using a dedicated thread pool, would allow the system to meet all timing requirements, including R8.1. The case study also showed that the hard real-time performance of the EM component was not compromised by executing non-real applications on the same node.
It might have been possible to achieve better performance with the PERC Ultra by using additional efforts e.g. by utilizing the optimization tools provided by Atego. The authors' qualifications for working with the PERC Ultra, were based on a two-day introduction workshop to the PERC Ultra JVM, held by Atego. Also, the obtained results must be seen in light of the limited timeframe available for the thesis work.
The EM component implemented using the RTSJ is somewhat limited in functionality as it does not interface with real hardware. This was chosen as the focus was on guaranteeing the timeliness of periodic invocations. Also the restrictions on accessing different memory areas imposed by the RTSJ, complicated the use of functionality and classes from the T-Core framework. In general, the most time consuming and challenging part of the case study was the design and configuration of the memory areas in the RTSJ implementation. The development process could however have benefitted from modeling the memory areas within step 2, thus utilizing VDM-RT for evaluating the usage of scoped memory, immortal memory and heap memory.

Chapter 8

Concluding Remarks and Future Work

This chapter briefly discusses the work presented in this thesis and concludes upon the achieved results. The thesis goals described in chapter 1 are related to the results, and especially the proposed TJARP method described in chapters 4 to 6. The results from the case study described in chapter 7 are used to evaluate the method.

8.1. Introduction

This thesis has investigated the area of real-time Java, where the Java community for several years has tried to bridge the gap between existing real-time techniques and the Java programming language. The widespread usage of the language with its high-level programming model and large amount of available libraries, together with the many benefits of automatic memory management and platform independence, have motivated this effort even further. However, as described throughout this thesis, the Java language suffers from non-deterministic temporal behavior and many challenges are faced when transforming a Java application towards real-time behavior. From the work supporting this thesis it is clear that real-time performance using Java is in fact possible, however as the degree of determinism increases so does the amount of compromises that have to be made.
Section 8.2 describes and discusses the achieved results. Future work is described in section 8.3, and finally the personal learning outcomes are described in section 8.4 together with final remarks in section 8.5.

8.2. Achieved Results

This thesis has analyzed available options for using Java in real-time systems, where an industrial case study was provided by Terma. The case study included a complex Java system with the need for obtaining real-time guarantees. The case study and the authors wide interest in Java and real-time systems motivated the goals of this thesis which, as presented in chapter 1, were:

1. **To provide an overview of available real-time Java technologies through evaluation and comparison, which will assist the choice of the optimum strategy towards achieving real-time performance.**

2. **To propose a methodology which will facilitate the process of introducing real-time performance in existing Java applications.**

In order to meet the thesis goals it was essential to study the available literature and relevant technologies, where especially the temporal challenges faced by standard Java applications were important to understand. These were investigated and described in chapter 2, where the Car Controller example was used to illustrate concrete examples e.g. automatic memory management was proven to have a negative effect on the temporal behavior.
Two official extensions to the Java language are proposed through the Java Community Process (JCP) to improve the temporal behavior of Java, the Real-Time Specification for Java (RTSJ) and the Safety Critical Java (SCJ) specification. These extensions are results of several years' effort towards obtaining real-time performance with Java. The extensions are described in detail in chapter 3, thus contributing to the first goal of this thesis.
The theory presented in chapter 2, the official extensions described in chapter 3 and the different strategies described in chapter 6 show the difficulty of selecting the optimum solutions for real-time Java. The available technologies all provide various kinds of advantages and disadvantages, where the best solution is highly dependent on the type of application and the applicable requirements. As a result the TJARP method is proposed with a structured set of guidelines for transforming Java applications towards real-time performance. The method is the primary product of this thesis as described by the second thesis goal. The method promotes a clear definition of real-time requirements followed by an analysis of the system by use of the modeling language, VDM-RT. Finally the method describes different implementation strategies, each with a specific purpose, goal and individual degree of real-time performance.
The following section highlights the achieved results for each step of the TJARP method. The first and second step of the method are discussed in section 8.2.1 and 8.2.2 respectively. The iterative process of step three and four is discussed in section 8.2.3. Finally the method was applied to an industrial case study, where the results are discussed in section 8.2.4.

8.2.1 Step 1: Requirements Analysis

Chapter 4 presented *Requirements Analysis* as the first step of the TJARP method. The chapter briefly discussed the role of requirements in real-time systems, where the process of extending functional requirements, with the non-functional real-time requirements was explained. The Car Controller example was used to illustrate key points e.g. how the minimum and maximum interval between two brake-events are equally important.
Different types of real-time requirements were introduced in the chapter, such as the separation of events, allowed deadlines, jitter etc. Furthermore, the importance of categorizing requirements into hard or soft was emphasized. The step is a crucial part of the TJARP method as the requirements serves as input for the model in step 2, the base for choosing a strategy in step 3 and as a basis for evaluation in step 4. The understanding and definition of real-time requirements contributes to both thesis goals.

8.2.2 Step 2: System Modelling

Chapter 5 presented *System Modeling* as the second step of the TJARP method, and motivated the development of a VDM-RT model with focus on real-time requirements and temporal attributes. This step utilizes the ability of the VDM-RT language to model both the functional and structural parts of the system, but also to constrain the model for temporal analysis. The Car Controller example was modeled using VDM-RT which identified temporal bottlenecks that later required

extra attention in the real-time implementation. But the model also showed how a new distributed architecture helped meet the specified requirements, proving how valuable time can be saved early in the design process before moving to the actual development step.
As part of this thesis the VDM development tool Overture, has been updated. The Real-Time Log Viewer (RTLV), in Overture, has been re-designed and re-implemented for faster processing and loading. The new design successfully contributed to the VDM community and together with the VDM-RT modeling language proved to be a valuable tool, both within chapter 5 but also for the Terma case study in chapter 7. The usage of formal modeling with VDM-RT is an important part of the TJARP method and contributes to meeting the second thesis goal.

8.2.3 Step 3 and 4: Selecting and Implementing Java Strategy

Chapter 6 presented *Java Strategy Selection* and *Implementation* as the third and fourth step of the TJARP method. Furthermore, an analysis uncovering the characteristics of relevant JVMs was performed and thus contributing to the first thesis goal. This analysis revealed that Aicas' JamaicaVM provides the best real-time characteristics but at the price of significantly reduced performance. The Oracle HotSpot provides the best performance but no real-time guarantees, while Atego's PERC Ultra is an intermediate solution providing soft real-time performance and medium performance.
The chapter also identified three different strategies for achieving real-time performance using Java, along with their strengths and weaknesses. These strategies are used as sub-steps in step 3 of the method and thus contribute to the second thesis goal. The strategies are:

Optimizing Standard Java: This approach relies on optimization of the existing application and the current JVM. Advantages of this solution are that it is a simple and fast way of improving the performance and timing predictability of the application. However, the major disadvantage is that no real-time guarantees can be provided by this approach.

Substituting the JVM: This approach is concerned with substituting the current JVM with a real-time JVM, including a real-time garbage collector. A significant advantage of this approach is that there is no need to rewrite the application to make use of different thread types or memory models. Disadvantages are that only soft real-time guarantees are achievable and the performance of the real-time JVMs is limited.

Applying the RTSJ: This approach relies on re-design and re-implementation of the existing application using the RTSJ, entirely or partly. An advantage of this approach is that it is possible to achieve hard real-time performance. However, a large disadvantage is the effort associated with utilizing the RTSJ, caused by its memory model.

The three approaches were applied to the Car Controller example, in order to gain practical experience, which could contribute to the analysis. Applying the third approach to the Car Controller example was particular challenging, as the work with the RTSJ revealed to be cumbersome and error prone.
In addition to the three approaches described above a fourth approach exists. This approach targets the use of Java for certifiable safety critical systems e.g. through the SCJ specification (see chapter 3) or commercial alternatives. This approach has been left out of the TJARP method intentionally as it is still too immature for use in industry. This approach will be further described as part of future work (see section 8.3).

8.2.4 Applying the TJARP Method on the T-Core Case Study

Chapter 7 describes the application of the TJARP method on the Terma *T-Core* case study. The purpose of the case study was to evaluate the TJARP method by analyzing and introducing soft real-time performance to the existing Track Manager (TM) component. Furthermore, the design and Java strategy was determined, for a newly introduced Engagement Manager (EM) component, with hard real-time constraints.

The case study proved that the TJARP method was able to analyze an existing complex system. The existing functional requirements of the TM component were extended with non-functional timing requirements, and the design of the additional EM component was developed. Figure 8.1 provides an overview of the degree of real-time required for the different components of the T-Core case study. The system was modeled using VDM-RT which helped identify several areas of concern. Here especially the design of the EM component was refined as the model determined timing bottlenecks in the initial design.

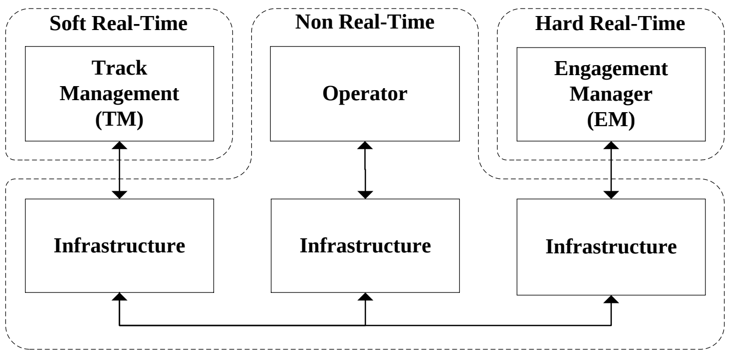


Figure 8.1: Degrees of real-time performance in the T-Core case study components

The experience gained through use of the model gave valuable input to the implementation of the EM component. The component was implemented using the RTSJ and executed on a separate JVM (the JamaicaVM), in order to isolate the hard real-time computations as identified by the model. The TM component was tested on the PERC Ultra JVM in order to improve its real-time guarantees. However, the test results revealed that the JVM was unable to meet the performance requirements of the system, and therefore the HotSpot had to be reintroduced. The final prototype of the EM component showed how the periodic communication with an imaginary weapon system was in fact able to meet hard real-time requirements. The periodic communication were guaranteed with maximum jitter value as low as 1,76%, even with an additional JVM doing intensive computation on the same node. This illustrates how the TJARP method can help determine a design which accommodates a temporal constrained component with achieving hard real-time guarantees. Furthermore, it is shown how the method supports development of systems with need for mixed real-time performance.

The case study allowed for evaluation of the TJARP method on a real-life case, and hence contributing to the second thesis goal. The method proved to be effective and helped select a strategy for the introduction of real-time guarantees. However, the case also provided the authors with important experience e.g. the sequential progress between step 2 and 3 actually turned out to be iterative. By continuously involving and updating the model in the implementation phase of step 3 and 4, the authors were able to do early testing and design updates on the model, saving valuable time and effort.

8.3. Future Work

Through the work on this thesis the authors has become aware of several interesting areas, which could be investigated further in the future. These areas are either concrete proposals for improving the TJARP method or simply related to the subjects described in this thesis. The following sections from 8.3.1 to 8.3.6 describe how future work within these areas could improve the results of this thesis.

8.3.1 Letting the TJARP Method Further Exploit the VDM Model

Practical experience was gained from applying the TJARP method to the T-Core case study. Here it turned out that the interaction between step 2 and step 3, were exhibiting a more iterative behavior than first described through chapters 5 and 6.
It became clear that the development process benefitted significantly from bringing the model from step 2, into step 3 and do continuously design space exploration using the executable model. Therefore, it is believed that the description of the TJARP method should be altered in order to encourage further usage of the VDM model in step 3. This would benefit the selection of real-time Java strategy, by allowing the strategy to be tested on the model before it is implemented during step 4.

8.3.2 Utilize VDM Modeling for Designing RTSJ Applications

Several times through this thesis it has been pointed out that the RTSJ is difficult to use in practice. This is mainly due to its complex memory model which introduces several types of memory areas with strict memory access rules.
However, the VDM language can potentially ease the task of working with the RTSJ, by augmenting the model with information about the RTSJ memory areas and their access rules. A concurrent model could be used for identifying potential violations of the memory access rules, which occurs when sharing resources between active objects. Hence, the active objects, their use of memory areas and interaction could be designed using the VDM model before any RTSJ code is written. This could be achieved by implementing a RTSJ memory library in VDM-RT which could be referenced by the model similar to the actual RTSJ implementation e.g. by adding class instances to a specific memory area implementation which is then subject to invariant checks for each operation.
This proposal could contribute to the RTSJ community in general, however the TJARP method would also benefit implicitly as it utilizes the RTSJ in sub-step 3.3. Additionally, this idea supports the proposal of utilizing the VDM model further during step 3 as described in section 8.3.1. It would then be possible to explore different designs of RTSJ components by using the model during step 3.

8.3.3 Extending the TJARP Method with Support for Mission Critical Systems

Support for certifiable safety critical applications has been left out of the TJARP method intentionally, as the available solutions are still too immature to be used in industry. Available solutions for using Java in certifiable safety critical systems include the SCJ specification (see section 3) and the commercial alternative by Atego, the PERC Pico [Nilsen07].
However, in the future, support for certifiable safety critical systems could be added to the TJARP method by expanding step 3 with another sub-step. This sub-step should then be concerned with redesigning and rewriting mission critical parts of the application for using SCJ or an alternative solution.

The introduction of support for these systems further motivates the TJARP method's use of a formal modeling technique like VDM during the development process. This potential expansion to the method would also benefit from incorporating the two proposals described in sections 8.3.1 and 8.3.2.

8.3.4 Extending the TJARP Method with Support for Distributed Nodes

Many Java based applications rely on a distributed architecture, where different Java applications communicate in order to achieve a common goal. This distributed communication is beyond the scope of this thesis. However, this is an interesting and highly relevant area for future work. Appendix E describes some of the most relevant specifications and available middleware solutions for optimizing real-time performance in a distributed environment. However, as described in the appendix, the only official language extension for distributed real-time Java (DRTSJ) is unfinished and currently marked as inactive.
The TJARP method can be extended to include the notion of distributed computation, by utilizing the advanced features of VDM-RT to do architectural prototyping on distributed nodes (see section 5.4.4). The method could further include distributed communication as a separate sub-step in the third step of the method. The introduction of real-time guarantees across nodes in systems, such as T-Core, is highly relevant, and a candidate for future work.

8.3.5 Further Exploiting the VDM Language and Tool Support

The VDM-RT modeling language together with the Overture tool, have proven to be valuable inputs to the TJARP method. Especially the ability to raise the level of abstraction while still maintaining the notion of time has proven to be useful. However, the many benefits described in chapter 5 does not fully cover all the possibilities offered by the language and tool support. Additional features such as combinatorial testing and model coverage could be interesting additions to the second step of the method. These features allow for further increasing the confidence within the behavior of the model, and thus contribute to the second thesis goal.
The Real-Time Log Viewer (RTLV) feature of the Overture tool has been re-implemented as part of this thesis. This new design has increased its usability by reducing the load time and improved the general performance. However, several additional features of the RTLV would make it even better. The user interface could be extended with a "live scroll bar", where the user is able to see a total overview of the log events and select special areas of interest which is then loaded by the main view. Similarly the ability to zoom in and zoom out could increase the usability even further. This is supported by the new architectural design of the RTLV.
The usage of VDM-RT in the TJARP method benefits greatly from validation conjectures, even though these are still an experimental feature of the Overture tool. Future work with conjectures should focus on improved tool support, e.g. by specifying conjectures through the user interface and not as comments within the model code as is the case in the current version. Similarly is the support for higher time resolution than the current milliseconds desirable, e.g. by specifying "ns" for nanoseconds instead of "ms" for milliseconds.

8.3.6 Improvements to the T-Core Case Study

The application of the TJARP method on the T-Core case study has provided the authors with valuable experiences as described in chapter 7. The limited timeframe of this thesis forced the authors to do a simplified proof-of-concept, with focus on the important aspects for evaluating the method. However, the case study, with both the VDM-RT model and the actual Java implementation could

benefit from future work.
The model could be extended to provide even more information about the temporal behavior of the existing T-Core system if a more detailed use of **`duration`** were utilized. This could be done by analyzing the real-life durations of specific operations and then use these values within the model. The evaluation of different JVMs and their influence on these **`duration`** statements could be implemented by adding a scalar, e.g. of one for the HotSpot and four for the PERC Ultra to model the difference in performance between the two.
The Java implementation of the EM component is simplified in order to extract only relevant attributes. Therefore, the current implementation could also benefit from future work. The component receives only track updates generated on the same JVM, as the communication between the TM component (on the HotSpot) and the EM component (on the JamaicaVM) caused additional difficulties.

8.4. Personal Learning Outcomes

Prior to this thesis both authors had very limited experience with academic research projects. Previous work was mostly concerned with technical implementations and the associated documentation. The work supporting this thesis required a comprehensive and thorough analysis of the available literature where the authors were required to extract only relevant information. This process has strengthened our ability to quickly obtain an overview of interesting and important parts of the available literature. This is an important ability as some research papers are highly relevant while others provide little or no useful information for the task at hand. The area of real-time Java is highly influenced by commercial interests and it has been an interesting challenge of providing a neutral view on the available solutions and technologies.
The authors have gained a much deeper knowledge of the possibilities for obtaining real-time performance with Java as desired by the two learning goals in section 1.3. Before this thesis the authors had a great interest in both Java and real-time systems together with a basic knowledge of both, but knew very little about real-time Java. The work of this thesis has promoted a better understanding of the temporal characteristics of standard Java including the challenges of using the essential high-level language features, such as garbage collection and platform independence, while providing deterministic temporal behavior.
This thesis has been completed in cooperation with Terma A/S who provided the T-Core case study, which at first, was too specific and product oriented to be the main subject for a master's thesis. Therefore, the authors learned to raise the levels of abstraction and generalize the given challenge, in order to propose a universal solution which could be put to use by others. This interaction with a company, which has a clear business goal of achieving real-time for its Java systems, served as a great motivation. This also allowed the authors to participate in a technical workshop held by Atego, which served as an introduction to the PERC Ultra JVM and the general challenges of obtaining real-time performance with Java. Therefore, to learn about the subject of real-time Java from a practitioner was a great learning experience. Here the authors were confirmed, that a substantial part of the material already produced for this thesis was indeed correct and highly relevant.
During this thesis, the main focus area has shifted several times. The original idea was to investigate if it was possible to achieve real-time performance using Java. As our knowledge of real-time Java was very limited beforehand, the initial approach was to research the area, where it quickly became clear that it was indeed possible. However, several solutions were available and it was difficult to assess which particular solution was the best choice. Hence the idea of a methodology,

for selecting the best approach for achieving real-time performance using Java, became the focus of the thesis.
Through the progress of this thesis, we have evolved as software engineers both in relation to the technical knowledge obtained, but also through the ability to communicate a highly detailed research area in a broad and consumable fashion. Through our education, to become software engineers, we have learned to divide large and overwhelming challenges in to smaller and more manageable parts. These skills were also put to use through this thesis where the task of analyzing the available strategies were broken down. First an investigation was carried out, uncovering why standard Java does not provide real-time guarantees. Afterwards existing solutions were researched and tried out. Finally, the theory and the practical experiences made were put together into this thesis.

8.5. Final Remarks

The goals of this thesis were to provide an overview of available real-time technologies for Java, and to propose a methodology to facilitate the process of selecting the optimum strategy. The authors are very pleased with the result of this thesis which successfully meets all specified goals. The thesis proves that the possibilities and options for achieving real-time guarantees using Java systems are many. The task of providing a complete and detailed overview of all available solutions is an unrealistic task for the course of a master´s thesis. Instead, a general method has been developed to provide a step-by-step approach supported by formal modeling to determine and analyze real-time requirements, as well as selecting the optimum path. This has been supported by a detailed analysis of some of the available language specifications and runtime environment implementations. A very realistic and suitable set of benchmark tests have been created, configured and applied to some of the most relevant JVM implementations on the market, including the Atego PERC Ultra and Aicas JamaicaVM. The proposed method has been applied successfully on an industrial case study, where hard real-time performance was obtained. The case study was a simplified proof-of-concept and allows for future work and optimizations, but the experience obtained were valuable input for evaluation of the method.
To conclude, obtaining real-time guarantees with Java is in fact possible. However, many tradeoffs and compromises must be made, as presented throughout this thesis.
The authors sincerely hope that others will adopt the TJARP method and use it for evaluating possible implementation strategies for reaching real-time performance with Java. The work supporting the method can optionally, be used in isolation for evaluating different approaches, such as the JVM benchmark results or the analysis of official language extensions for Java. It is believed that the shortcomings mentioned as future work, will strengthen the method and real-time Java even further. Especially the possibilities for using VDM-RT to model the complex memory approach of RTSJ before implementation, is an interesting area for future research.

References

[Aicas06] Aicas GmbH. The JamaicaVM brings Java Technology to Mission Software in an unmanned aircraft by EADS. Press Release, Jun 2006. http://www.aicas.com/press/pr_34_en_28-Jun-06.html [Accessed: 9. Dec, 2012]. [cited at p. 62]

[Aicas12] Aicas GmbH. *J*amaicaVM 6.1 - User Manual: Java Technology for Critical Embedded Systems. Technical Report, Aicas, 2012. [cited at p. 49, 51, 68]

[aJile] aJile Systems Inc. aJile Systems. Web, Dec 2012. http://www.ajile.com/ [Accessed: 9. Dec, 2012]. [cited at p. 53]

[Atego06] Atego Systems, Inc. Lockheed Martin Selects Aonix PERC Virtual Machine for Aegis Weapon System. Press Release, Oct 2006. http://www.atego.com/pressreleases/pressitem/lockheed-martin-selects-aonix-perc-virtual-machine-for-aegis-weapon-system [Accessed: 9. Dec, 2012]. [cited at p. 59]

[Atego12] Atego Systems, Inc. *P*ERC® Ultra SMP 6.1 - User Manual. Technical Report, Atego, 2012. [cited at p. 51, 68]

[Bacon&03] Bacon, David F. and Cheng, Perry and Rajan, V.T. The Metronome: A Simpler Approach to Garbage Collection in Real-time Systems. In *W*orkshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops, pages 466–478, 2003. [cited at p. 7, 28]

[Baker06] Baker, Theodore P. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *R*eal-Time Syst., 32(1-2):49–71, February 2006. [cited at p. 7, 13]

[Baker&88] Baker, T.P. and Shaw, A. The cyclic executive model and Ada. In *R*eal-Time Systems Symposium, 1988., Proceedings., pages 120 – 129, dec 1988. [cited at p. 12]

[Barbacci&08] Barbacci, Mario R. and Ellison, Robert and Lattanze, Anthony J. and Stafford, Judith A. and Weinstock, Charles B. and Wood, William G. Quality Attribute Workshops (QAWs). [cited at p. 33]

[Benowitz&03A] Benowitz, Edward G. and Niessner, Albert E. Experiences in adopting real-time java for flight-like software. In *O*TM 2003 Workshops, pages 490–496, Springer Verlag, 2003. [cited at p. 28]

[Benowitz&03B] Benowitz, Edward G. and Niessner, Albert E. A patterns catalog for RTSJ software designs. In *I*n Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops, pages 497–507, 2003. [cited at p. 28]

[Bloch01] Bloch, Joshua. *E*ffective Java programming language guide. Sun Microsystems, Inc., Mountain View, CA, USA, 2001. [cited at p. 21]

[Bøgholm&08] Bøgholm, Thomas and Kragh-Hansen, Henrik and Olsen, Petur. *M*odel-Based Schedulability Analysis of Real-Time Systems. Master's thesis, Department of Computer Science, Aalborg University, Jun 2008. 137 pages. [cited at p. 45]

[Bollella&00] Bollella, Gregory and Gosling, James. The Real-Time Specification for Java. *I*EEE Computer, 33(6):47–54, 2000. [cited at p. 23]

[CD] Purdue University. Collision Detector A Famility of Real-time Java Benchmarks. Web, Dec 2012. `http://sss.cs.purdue.edu/projects/cdx/` [Accessed: 10. Dec, 2012]. [cited at p. 55, 127]

[Chen&05] Chen, Yaofei and Dios, R. and Mili, A. and Wu, Lan and Wang, Kefei. An empirical study of programming language trends. *S*oftware, IEEE, 22(3):72 – 79, May-Jun 2005. [cited at p. 1]

[Chung&09] Chung, Lawrence and do Prado Leite, Julio. On Non-Functional Requirements in Software Engineering. In Borgida, Alexander and Chaudhri, Vinay and Giorgini, Paolo and Yu, Eric, editors, *C*onceptual Modeling: Foundations and Applications, pages 363–379, Springer Berlin / Heidelberg, 2009. [cited at p. 33]

[Clarke&86] Clarke, E. M. and Emerson, E. A. and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *A*CM Transactions on Programming Languages and Systems, 8:244–263, 1986. [cited at p. 35]

[Coffman&71] Coffman, E. G. and Elphick, M. and Shoshani, A. System Deadlocks. *A*CM Comput. Surv., 3(2):67–78, Jun 1971. [cited at p. 15]

[Crocker10] Crocker, David. Dynamic Memory Allocation in Critical Embedded Systems. Web, 2010. `http://critical.eschertech.com/2010/07/30/dynamic-memory-allocation-in-critical-embedded-systems/` [Accessed: 9. Dec, 2012]. [cited at p. 18]

[Davis&11] Davis, Robert I. and Burns, Alan. A survey of hard real-time scheduling for multiprocessor systems. *A*CM Comput. Surv., 43(4):35:1–35:44, October 2011. [cited at p. 14]

[Dawson07] Dawson, Michael. Real-time Java Part 6: Simplifying real-time Java development. jul 2007. [cited at p. 28]

[Dawson08] Dawson, Michael H. Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. In *P*roceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, pages 241–247, IEEE Computer Society, Washington, DC, USA, 2008. 7 pages. [cited at p. 19]

[Dijkstra65] Dijkstra, Edsger W. Solution of a problem in concurrent programming control. *C*ommun. ACM, 8(9):569–, September 1965. [cited at p. 15]

[Dijkstra68] Dijkstra, Edsger W. Cooperating sequential processes. In F. Genuys, editor, *P*rogramming Languages: NATO Advanced Study Institute, pages 43–112, Academic Press, 1968. [cited at p. 15]

[DO-178B] RTCA SC-167/EUROCAE WG-12. *S*oftware Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178B, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 1992. [cited at p. 27]

[Douglass01] Douglass, Bruce Powel. Capturing Real-Time Requirements. Nov 2001. `http://www.embedded.com/design/prototyping-and-development/4023862/Capturing-Real-Time-Requirements` [Accessed: 9. Dec, 2012]. [cited at p. 31]

[FijiVM] Fiji Systems Inc. Java. Anywhere. On time. Web, Dec 2012. `http://fiji-systems.com/` [Accessed: 9. Dec, 2012]. [cited at p. 53]

[Fitzgerald&07] Fitzgerald, John and Larsen, Peter Gorm and Tjell, Simon and Verhoef, Marcel. *Validation Support for Distributed Real-Time Embedded Systems in VDM++*. Technical Report CS-TR:1017, School of Computing Science, Newcastle University, April 2007. 18 pages.

[cited at p. 34, 43]

[Gamma&95] Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. *D*esign Patterns. Elements of Reusable Object-Oriented Software. Volume of *A*ddison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, edition, 1995. 395 pages. . [cited at p. 125]

[Gilb97] Gilb, Thomas. Towards the Engineering of Requirements. *R*equir. Eng., 2(3):165–169, 1997. [cited at p. 33]

[Glinz05] Glinz, Martin. Rethinking the Notion of Non-Functional Requirements. In *P*roceedings of the Third World Congress for Software Quality (3WCSQ'05, pages 55–64, 2005. [cited at p. 33]

[Gosling&96] Gosling, James and Joy, Bill and Steele, Guy L. *T*he Java Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. [cited at p. 14]

[Hertz&04] Hertz, Matthew and Berger, Emery D. Automatic vs. Explicit Memory Management: Settling the Performance Debate. *O*OPSLA '04, 2004. [cited at p. 19]

[Hertz&05] Hertz, Matthew and Berger, Emery D. Quantifying the performance of garbage collection vs. explicit memory management. In *P*roceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 313–326, ACM, New York, NY, USA, 2005. [cited at p. 18, 19]

[Higuera-Toledano&12] Higuera-Toledano, M. Teresa and Wellings, Andy, editors. *D*istributed, Embedded and Real-time Java Systems. Springer, 2012. 378 pages. [cited at p. 1, 14, 25, 137]

[Hoare74] Hoare, C. A. R. Monitors: an operating system structuring concept. *C*ommun. ACM, 17(10):549–557, October 1974. [cited at p. 15]

[IBM04] IBM developerWorks. Java theory and practice: Garbage collection and performance. Web, Jan. 2004. `http://www.ibm.com/developerworks/java/library/j-jtp01274/index.html` [Accessed: 9. Dec, 2012]. [cited at p. 49]

[IBM07] IBM developerWorks. Real-time Java, Part 1: Using Java code to program real-time systems. Web, Apr. 2007. `https://www.ibm.com/developerworks/java/library/j-rtj1/` [Accessed: 9. Dec, 2012]. [cited at p. 1]

[JamaicaVM] Aicas GmbH. JamaicaVM - Java Technology for Realtime. Web, Dec 2012. `http://www.aicas.com/jamaica.html` [Accessed: 9. Dec, 2012]. [cited at p. 53]

[JavaRTS] Oracle. Java Real-Time System. Web, Dec 2012. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-139921.html` [Accessed: 9. Dec, 2012]. [cited at p. 53]

[Jørgensen12] Jørgensen, Peter W. V. *Evaluation of Development Process and Methodology for Co-Models*. Master's thesis, Aarhus University School of Engineering, Dec 2012. 80 pages. [cited at p. 123]

[JSR001] Java Community Process. Java Specification Requests: JSR 1 - Real-Time Specification for Java. 1998. `http://jcp.org/en/jsr/detail?id=1` [Accessed: 9. Dec, 2012]. [cited at p. 2, 23]

[Karlsson97] Karlsson, Joachim and Ryan, Kevin. A Cost-Value Approach for Prioritizing Requirements. *I*EEE Softw., 14(5):67–74, Sep 1997. [cited at p. 36]

[Larsen&09] Peter Gorm Larsen and John Fitzgerald and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *I*ntl. Journal of Software and Informatics, 3(2-3), October 2009. [cited at p. 38]

[Larsen&10a] Larsen, Peter Gorm and Lausdahl, Kenneth and Battle, Nick. *The VDM-10 Language Manual*. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010. [cited at p. 37, 38, 42]

[Larsen&10b] Larsen, Peter Gorm and Lausdahl, Kenneth and Ribeiro, Augusto and Wolff, Sune and Battle, Nick. *Overture VDM-10 Tool Support: User Guide*. Technical Report TR-2010-02, The Overture Initiative, `www.overturetool.org`, May 2010. 103 pages. [cited at p. 38, 39]

[Larsen&10c] Larsen, Peter Gorm and Wolff, Sune and Battle, Nick and Fitzgerald, John and Pierce, Ken. *D*evelopment Process of Distributed Embedded Systems using VDM. Technical Report TR-2010-02, The Overture Open Source Initiative, April 2010. [cited at p. 38]

[Larsen11] Larsen, Peter Gorm. Introduction to the Modeling of Mission Critical Systems Course. Slideshow. [cited at p. 40]

[Lauesen02] Lauesen, Søren. *S*oftware Requirements: Styles and Techniques. Addison-Wesley, 2002. [cited at p. 32]

[Lehoczky&89] Lehoczky, J. and Sha, L. and Ding, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *R*eal Time Systems Symposium, 1989., Proceedings., pages 166 –171, dec 1989. [cited at p. 13]

[Levanoni&06] Levanoni, Yossi and Petrank, Erez. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, January 2006. [cited at p. 19]

[Lieberman&83] Lieberman, Henry and Hewitt, Carl. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983. [cited at p. 19]

[Lindstrom&05] Lindstrom, Gary and Mehlitz, Peter C. and Visser, Willem. Model checking real time java using java pathfinder. In *Proceedings of the Third international conference on Automated Technology for Verification and Analysis*, pages 444–456, Springer-Verlag, Berlin, Heidelberg, 2005. [cited at p. 45]

[Liu&73] Liu, C. L. and Layland, James W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973. [cited at p. 13]

[Locke&11] Locke, Doug and Andersen, B. Scott and Brosgol, Ben and Fulton, Mike and Henties, Thomas and Hunt, James J. and Nielsen, Johan Olmütz and Nilsen, Kelvin and Schoeberl, Martin and Tokar, Joyce and Vitek, Jan and Wellings, Andy. *Safety-Critical Java Technology Specification, Public draft.* 2011. [cited at p. 27]

[Lu&11] Lu, Yue and Kraft, Johan and Nolte, Thomas and Bate, Iain. A statistical approach to simulation model validation in response-time analysis of complex real-time embedded systems. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 711–716, ACM, New York, NY, USA, 2011. [cited at p. 45]

[Manna&92] Manna, Zohar and Pnueli, Amir. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992. [cited at p. 35]

[McCarthy60] McCarthy, John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, April 1960, 1960. [cited at p. 18]

[Mikhalenko06] Mikhalenko, Peter. Real-Time Java: An Introduction. May 2006. `http://onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html` [Accessed: 9. Dec, 2012]. [cited at p. 2]

[Mok83] Mok, A. K. *Fundamental design problems of distributed systems for the hard-real-time environment.* Technical Report, Cambridge, MA, USA, 1983. [cited at p. 13, 16]

[Nilsen07] Nilsen, Kelvin. Improving abstraction, encapsulation, and performance within mixed-mode real-time Java applications. In *P*roceedings of the 5th international workshop on Java technologies for real-time and embedded systems, pages 13–22, ACM, New York, NY, USA, 2007. [cited at p. 28, 69, 93]

[Nilsen09] Nilsen, Kelvin. *D*ifferentiating Features of the PERC Virtual Machine. Technical Report, Atego, 2009. [cited at p. 59, 60]

[Oracle08] Oracle Technology Network. The Java HotSpot Performance Engine Architecture. Web, 2008. `http://www.oracle.com/technetwork/java/whitepaper-135217.html` [Accessed: 9. Dec, 2012]. [cited at p. 14, 49]

[Oracle12] Java SE HotSpot at a Glance. Web, Nov 2012. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html` [Accessed: 9. Dec, 2012]. [cited at p. 57]

[OVM] Purdue University. OVM Project. Web, Dec 2012. `http://www.cs.purdue.edu/homes/jv/soft/ovm/` [Accessed: 9. Dec, 2012]. [cited at p. 53]

[PERC] Atego. Aonix PERC. Web, Dec 2012. `http://www.atego.com/products/aonix-perc/` [Accessed: 9. Dec, 2012]. [cited at p. 53]

[Phipps99] Phipps, Geoffrey. Comparing observed bug and productivity rates for Java and C. *S*oftware — Practice and Experience, 29:345–358, 1999. [cited at p. 2]

[Pizlo&04] Pizlo, F. and Fox, J.M. and Holmes, D. and Vitek, J. Real-time Java scoped memory: design patterns and semantics. In *O*bject-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on, pages 101 –110, may 2004. [cited at p. 28]

[Pizlo&09] Pizlo, Filip and Ziarek, Lukasz and Vitek, Jan. Real time Java on resource-constrained platforms with Fiji VM. In *P*roceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, pages 110–119, ACM, New York, NY, USA, 2009. [cited at p. 53]

[Plsek09] Plsek, Ales. *S*OLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems. PhD thesis, Université des Sciences et Technologie de Lille - Lille I, Sep 2009. [cited at p. 2, 28]

[Rajkumar89] Rajkumar, R. *T*ask Synchronization in Real-time Systems. Carnegie-Mellon University, 1989. [cited at p. 13]

[Rajkumar91]	Rajkumar, R. *S*ynchronization in Real-time Systems: A Priority Inheritance Approach. *K*luwer international series in engineering and computer science: Real-time systems, Kluwer Academic Publishers, 1991. [cited at p. 13]
[Ramakrishna&96]	Ramakrishna, Y. S. and Melliar-Smith, P.M. and Moser, L.E. and Dillon, L. K. and Kutty, G. Interval Logics and Their Decision Procedures - Part I: An Interval Logic. *T*heoretical Computer Science, 170:166–1, 1996. [cited at p. 35]
[Ribeiro&11]	Ribeiro, Augusto and Lausdahl, Kenneth and Larsen, Peter Gorm. Run-Time Validation of Timing Constraints for VDM-RT Models. In *9*th Overture Workshop, June 2011, Limerick, Ireland, 2011. [cited at p. 39]
[RTSJRI]	Timesys. RTSJ Reference Implementation (RI) and Technology Compatibility Kit (TCK). Web, Dec 2012. `http://www.timesys.com/java/` [Accessed: 9. Dec, 2012]. [cited at p. 53]
[Sha&86]	Sha, Lui and Lehoczky, John P. and Rajkumar, Ragunathan. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *I*EEE Real-Time Systems Symposium, pages 181–191, IEEE Computer Society, 1986. [cited at p. 13]
[Sha&90]	Sha, L. and Rajkumar, R. and Lehoczky, J.P. Priority inheritance protocols: an approach to real-time synchronization. *C*omputers, IEEE Transactions on, 39(9):1175 –1185, sep 1990. [cited at p. 13, 15, 16]
[SPEC]	Standard Performance Evaluation Corporation. SPECjvm2008. Web, Dec 2012. `http://www.spec.org/jvm2008/` [Accessed: 10. Dec, 2012]. [cited at p. 56, 127]
[Stankovic88]	Stankovic, John A. Misconceptions About Real-Time Computing. *I*EEE Computer, 21(10):10–19, 1988. [cited at p. 33]
[Sun01]	Sun Microsystems. The Java HotSpot Virtual Machine. Technical White Paper. Web, May 2001. 23 pages. . [cited at p. 57]
[Sun03]	Sun Microsystems. Sun Bug Database Bug 4813310 - Map Thread priorities to system thread or process priorities. Web, 2003. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4813310` [Accessed: 9. Dec, 2012]. [cited at p. 21]
[Sun06]	Sun Microsystems. Memory Management in the Java HotSpot Virtual Machine. web, April 2006. 21 pages. A white paper from Sun Microsystems. [cited at p. 19, 20]
[Tene&05]	Tene, Gil and Posva, Ivan. Java Performance Myths Exposed. 2005. Azul Systems. [cited at p. 49]

[Terma10] Terma A/S. Track Management A component in the T-Core software complex. Web. `http://www.terma.com/defense/joint-and-land-systems/air-defense/` [Accessed: 9. Dec, 2012]. [cited at p. 71]

[Terma11] Terma A/S. BMD-Flex International Air and Missile Defense Command and Control. Web. `http://www.terma.com/defense/joint-and-land-systems/ballistic-missile-defense/` [Accessed: 9. Dec, 2012]. [cited at p. 5, 71]

[Terma12] Terma A/S. C-RAID Situational Awareness and C2 Capabilities for Tactical, Maritime Platforms. Web. `http://www.terma.com/defense/naval-tactical-solutions/c-raid-naval-c2-system-for-small-maritime-units/` [Accessed: 9. Dec, 2012]. [cited at p. 71]

[Thiele&00] Thiele, L. and Chakraborty, S. and Naedele, M. Real-time calculus for scheduling hard real-time systems. In *C*ircuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on, pages 101 –104 vol.4, 2000. [cited at p. 45]

[TIOBE12] TIOBE Software. TIOBE Programming Community Index for November 2012. Web. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` [Accessed: 9. Dec, 2012]. [cited at p. 1]

[VDMTools] CSK. VDMTools homepage. Web, Dec 2012. `http://www.vdmtools.jp/en/` [Accessed: 9. Dec, 2012]. [cited at p. 39]

[Venners99] Venners, Bill. *I*nside the Java Virtual Machine. McGraw-Hill Professional, 1st edition, 1999. [cited at p. 19]

[Walkup&94] Walkup, Elizabeth A. and Borriello, Gaetano. Interface timing verification with application to synthesis. In *P*roceedings of the 31st annual Design Automation Conference, pages 106–112, ACM, New York, NY, USA, 1994. [cited at p. 35]

[WebSphereRT] IBM. IBM® WebSphere® Real-Time. Web, Dec 2012. `http://www.ibm.com/software/webservers/realtime/` [Accessed: 9. Dec, 2012]. [cited at p. 53]

[Wegener&98] Wegener, Joachim and Grochtmann, Matthias. Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. *R*eal-Time Syst., 15(3):275–298, November 1998. [cited at p. 45]

[Wellings&02] Wellings, A. and Clark, R. and Jensen, D. and Wells, D. A. framework for integrating the real-time specification for Java and Java's remote method invocation. In *O*bject-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on, pages 13 –22, 2002. [cited at p. 137]

[Wieringa03] Wieringa, R. J. *D*esign Methods for Reactive Systems: Yourdon, Statemate and the UML. Morgan Kaufmann Publishers, 2003. [cited at p. 33]

Appendices

Appendix A

Terminology

$^{\mathcal{T}}$***FIFO*:** First In First Out. Describes the principle of a queue mechanism where the items leave the queue in same order as they arrive. For FIFO scheduling this means threads will be activated in the order they become ready.

$^{\mathcal{T}}$***JCP*:** Java Community Process. A formalized forum which allows interested parties to specify and develop technical specifications for extending the Java language and technology.

$^{\mathcal{T}}$***JSR*:** Java Specification Request. A formal document that describes a proposed specification within the Java Community Process. A final JSR must provide a reference implementation and a technology compatibility kit to verify the specification.

$^{\mathcal{T}}$***Monitor*:** An object or module often used in concurrent programming for signaling other threads that a certain condition has been met.

$^{\mathcal{T}}$***Mutex*:** An object or module, used in concurrent programming to ensure mutual exclusion when multiple tasks access a shared resource.

$^{\mathcal{T}}$***NP-Hard Problem* :** A problem is NP-hard if a given solution can be translated into one solving any non-deterministic polynomial time problems.

$^{\mathcal{T}}$***Preemption*:** Is the act of temporarily interrupting an active task with the intention of resuming the task at a later time.

$^{\mathcal{T}}$***Processor Utilization*:** The amount of CPU time a task or set of tasks are utilizing. Usually given in percentage.

$^{\mathcal{T}}$***Profiler*:** A tool capable of analyzing the runtime behavior of an application and providing an overview of specific attributes such as memory usage, CPU load etc.

$^{\mathcal{T}}$***Race Condition*:** An error situation which can arise in software systems when multiple tasks or processes depend on a shared state or event. Failure to synchronize access to shared resources may result in race conditions where the value of the resource is corrupted or invalid.

$^{\mathcal{T}}$***Reflection*:** The ability of a computer application to examine and modify the execution behavior at runtime. This often involves loading or changing specific objects depending on changing system states.

$^{\mathcal{T}}$***Schedulable*:** A given task is schedulable if it is ready for operation and the required deadlines can be met.

$^{\mathcal{T}}$***Semaphore*:** A object or variable that allows for controlling access to a shared resource. Semaphores can be binary, and thus acts as an mutex, or they can be counting semaphore which increments and decrements an interval value each time they are activated.

$^{\mathcal{T}}$***Starvation*:** Resource starvation in computer science occur when a task or process is denied access to a resource which is required for the task or process to meet its purpose. For instance, when continuous processing in high priority threads prevents lower priority threads from executing.

$^{\mathcal{T}}$***Transitive Closure*:** Transitive closure of a graph is the set of nodes that can be referenced from the initial node. Meaning that isolated subsets of the graph is not part of the transitive closure set.

$^{\mathcal{T}}$***WCET*:** Worst-Case-Execution-Time. Is the maximum length of time a given tasks could take to execute a specific operation.

Appendix B

Case Study Details

This appendix provides details about the two case studies used in this thesis. Details about the Car Controller example is found in section B.1, while section B.2 provides details about the T-Core case study. The source code created for each case study is included on the attached CD.

B.1. Car Controller

This section describes the details of the Car Controller case study. Section B.1.1 elaborates on the Java implementation and section B.1.2 on the VDM-RT implementation.

B.1.1 Java Application

Here the software design of the Java application in the Car Controller case study will be described. The Car Controller case study was first introduced in section 1.5.1. Figure B.1 provides an overview of the classes in the application.

The application has four active objects, where the `BrakePedalEventHandler` and the `GasPedalEventHandler` are asynchronous event handlers. They handle incoming events of the types `BrakePedalEvent` and `GasPedalEvent`, which are triggered when the driver alters the pressure on the brake pedal or gas pedal, respectively. The remaining two active objects are the `Navigation` and `CruiseController`, which are periodic threads. The `Navigation` thread updates the in-car navigation display with a period of 33 ms. The `CruiseController` thread monitors the speed of the car with a period of 50 ms and alters the speed if necessary, in order to keep a constant speed.

The Car Controller interfaces with three peripherals found in the car through instances of the classes `Brakes`, `Engine` and `Display`. The `Engine` class is shared among the `Cruise-Controller` thread and the `GasPedalEventHandler` thread, hence it needs synchronization such that both active objects does not alter the speed simultaneously.

The implementation of the Car Controller uses the API specified by the RTSJ (see chapter 3). Therefore, in order to test the application on a standard Java JVM, a wrapper library has been implemented. This library provides the same API as specified by the RTSJ, however the functionality is implemented using standard Java. This enables execution of the same application on both standard Java and RTSJ compliant JVM's without altering the application code.

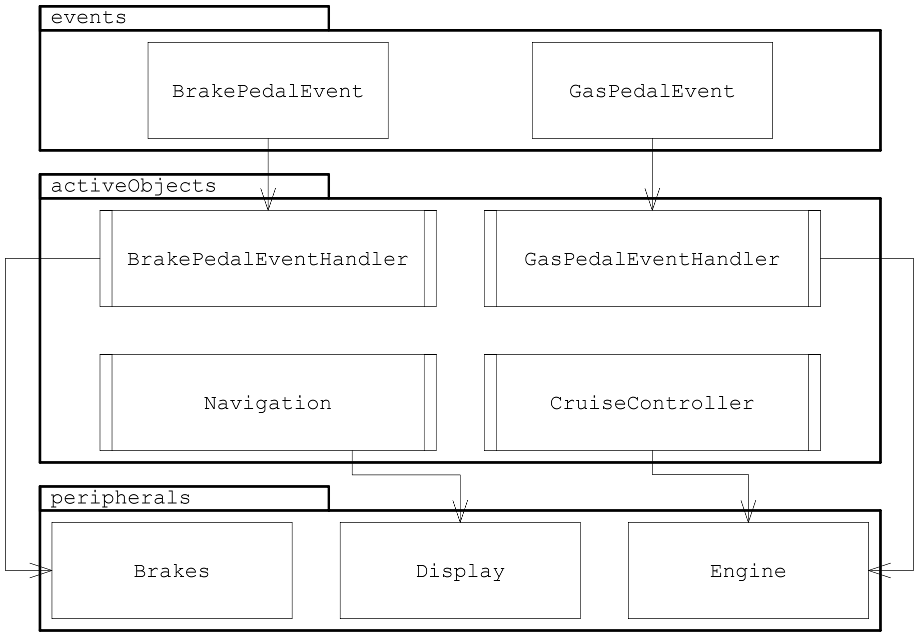


Figure B.1: Class diagram of the Car Controller software

B.1.2 VDM-RT Model

This section extends section 5.4 and describes the simplified VDM-RT model of the Car Controller used to illustrate important concepts. The model structure is similar to the Java equivalent, with a few additional classes. Figure B.2 shows the class diagram of the model, where the additional `World`, `Driver` and `CarEnvironment` are included. The `World` class is used for loading and starting the different scenarios of the model. The `Driver` class is used to stimulate the system with external events such as activating pedals. The `CarEnvironment` class is used to configure the architectural distribution of classes on different `CPU` instances. The `CarEnvironment` is identified by using the **`system`** keyword in the class definition.
The pedal classes identified as peripherals in section B.1.1 are simple classes with **`async`** operations which models asynchronous events. The `Engine` class is a simplified representation of an engine, where set- and get-operations are protected with permission predicates to model the behavior of a shared resource. The implementation of the `CruiseController` class is illustrated in listing B.1, where the simplified adjustment of speed is modeled in the operation `MonitorCruiseSpeed`. The operation is periodically invoked through the **`thread`** declaration.

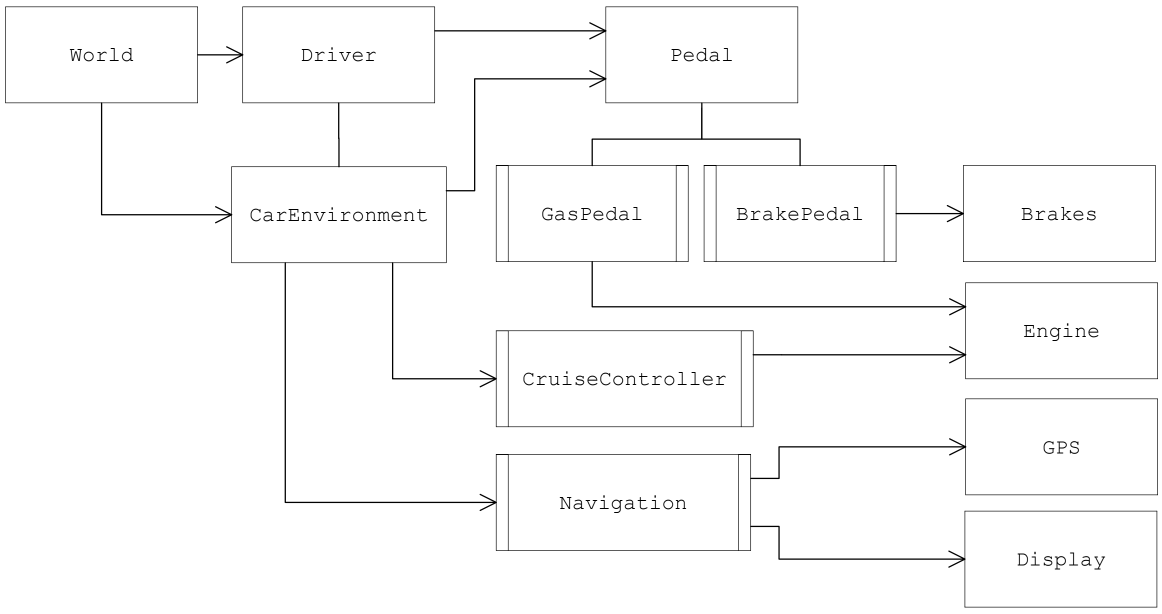


Figure B.2: Structural class diagram of the Car Controller model

```

public MonitorCruiseSpeed : () ==> ()
MonitorCruiseSpeed() ==
(
    dcl newRpm : nat :=
        CalculateNewMotorRpm(engine.GetRPM(), engine.GetRPM());
    if cruiseActive then engine.SetRPM(newRpm);
);

thread
    periodic(50E6,100,100,0)
        (MonitorCruiseSpeed)
```

Listing B.1: Processing and thread declaration within the `CruiseController` class

The `Navigation` class is implemented similarly with a periodic invocation of the processing operations.

B.2. Terma T-Core

This section describes the details of the T-Core case study. Section B.2.1 elaborates on the implementation of the VDM-RT model. Section describes the design and evaluation of the Java implementation of Engagement Manager (EM) component in the T-Core case study. Section B.2.3 provides further details of the obtained test results.

B.2.1 VDM-RT Model

This section elaborates on the T-Core model presented in section 7.3 of this thesis.
The structure of the model is illustrated in figure 7.3 (see page 76). The classes implemented on the *server* node are all related directly to existing classes within the T-Core framework. The model reflects their functional behavior with focus on temporal analysis. The classes marked within the *weapon control* node are the additional *Engagement Manager* component responsible for evaluating detected tracks, and if needed communicate with the weapon class.
The `World` class is the starting point-of-execution where different scenarios are specified. The scenarios are described in text files with track information specified as VDM tuples and loaded by the `TrackSensor` class. Listing B.2 shows the definition of the `SensorEvent` tuple which is read from text files.

```
--Type, Unique track id for this sensor, Category,
--Identity, Track Type, Latitude, Longitude, Altitude
SensorEvent = SensorEventType * nat * TrackCategory
* TrackIdentity * AirTrackType * nat * nat * nat;
```

Listing B.2: The operation for adding tracks within the `TrackManager`

Once an event is read and parsed by the `TrackSensor` class, the events are passed to the `TrackManager` class by use of the asynchronous operations within the `TrackReceiver` class. The sequence of detecting a new track, and passing the information through the system is illustrated in figure B.3. The figure shows how the `TrackManager` further publishes the new track information to subscribing instances of the `TrackListener` class by use of the `Infrastructure` class.

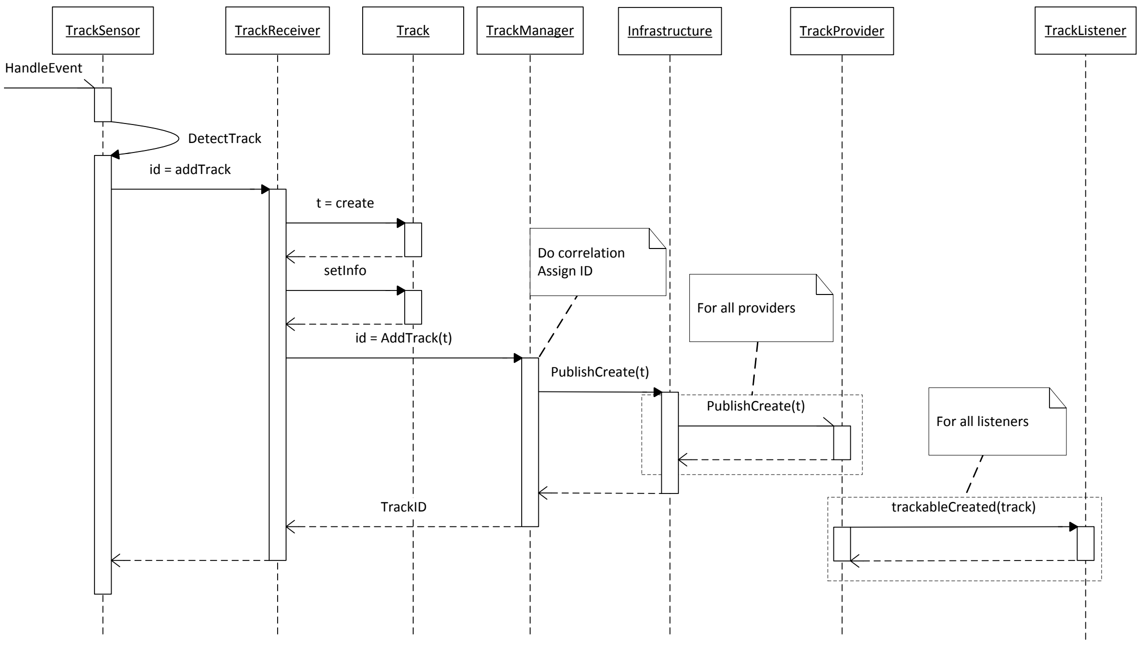


Figure B.3: Sequence diagram for track creation in the model

When a new track is added to the `TrackManager` class, the operation `AddTrack` is invoked.

The operation is shown in listing B.3. The operation uses the correlation algorithm implemented in `GetCorrelatedTrackID` to determine a new *Trackmanagement Item Reference Key* (TIRK) which is a unique key for all tracks in the system (line 4). When the key is determined the `TrackManager` class makes use of the `EvaluateTrack` operation to determine if a track is to be deemed hostile based on a simple set of rules. Once the track is processed it is published to subscribing listeners (line 13 and 14).

```
 1  public AddTrack : Track ==> Track`TrackID
 2  AddTrack(track) ==
 3  (
 4  dcl trackId : Track`TrackID := GetCorrelatedTrackID(track);
 5  if trackId not in set (dom tracks) then
 6  (
 7    --New Track
 8    track.SetTrackId(trackId);
 9
10    --Evaluate and publish track if we have a
11    --working infrastructure
12    track.SetIdentity(EvaluateTrack(track));
13    if infrastructure <> nil then
14      infrastructure.PublishCreate(track);
15
16    --Save track
17    tracks := tracks munion {trackId |-> track};
18  )
19  else
20  (
21    --Track is deemed equal to another track through correlation
22    --Return that track id to sensor for future references
23    skip;
24  );
25  return trackId;
26  );
```

Listing B.3: The operation for adding tracks within the `TrackManager`

The simplified track correlation algorithm is shown in listing B.4. The real algorithm from the T-Core system uses complex rules for merging tracks detected by different sensors. However, the model implementation raises the level of abstraction and implements a simplified version to model processing of tracks. The algorithm first checks if the track is located at the same position as an already known track, by defining an offset radius which creates a 3D-sphere with the current track as center (line 14 and 15). If the new track is within the sphere and of similar type then the tracks are correlated. The check for type is to avoid merging two different but close tracks e.g. an airplane at low altitude and a ground vehicle. If the track is merged then the TIRK of the original track is returned, if not a new TIRK is created and returned.

```
 1  public GetCorrelatedTrackID : Track ==> Track`TrackID
 2  GetCorrelatedTrackID(t) ==
(
 --Real life correlation is represented by
 --SystemTrackCorrelation interface containing
 --one SystemTrack TIRK and all corresponding LocalTrack TIRKs
 --This is simplified (abstraction) by this algorithm


  dcl id : Track`TrackID;
  dcl trackPos : GeoArea`Position := t.GetPosition().#2;

  --Create correlation area to merge similar tracks
  dcl correlationArea : GeoCircle :=
   new GeoCircle(trackPos,correlationRadius,correlationRadius);

  --The current algorithm does not support correlating
  --two existing (seperate) tracks, only one existing and one new
  if t.GetTrackId().TIRK <> 0 then
  (
    id := t.GetTrackId()
  )
  --If track position is within specified radius of an
  --already known track with similar category
  --(avoid merging a <GROUND> track with an <AIR>
  -- track at same position) then merge them into one
  else if exists localTrack in set (rng tracks) &
   correlationArea.IsInside3D(localTrack.GetPosition().#2)
  and (localTrack.GetTrackType().Category =
  t.GetTrackType().Category or
  t.GetTrackType().Category = <UNKNOWN>) then
  (
    let localTrack in set (rng tracks) be st
    (correlationArea.IsInside3D(trackPos) and
    (localTrack.GetTrackType().Category =
    t.GetTrackType().Category or
    t.GetTrackType().Category = <UNKNOWN>))
     in id := localTrack.GetTrackId();
  )
  --Else if track has no ID assume it to be unknown
  --and create new ID
  else
  (
    id := mk_Track`TrackID(tirkCounter,t.GetOwnerID());
    tirkCounter := tirkCounter+1;
  );

return id;
);
```

Listing B.4: The correlation algorithm within the `TrackManager`

When the `TrackManager` class, as explained above, publishes new track information, it is received by instances of the `TrackListener` class. The `TrackEvaluationHandler` inherits from the `TrackListener`, and evaluates the track to see if it is inside a predefined area and of the type *HOSTILE*. If so, the track is passed to the `TrackEngagementHandler` which creates and starts the periodic thread implemented in the `WeaponComHandler` (WCH). The WCH class inherits from the `Schedulable` class to model periodic time-based invocations by invoking the inherited `WaitForNextPeriod` operation as shown in listing B.5. When derived classes invoke the operation, their **`self`** reference is added to the `Scheduler` class, and then blocked by calling the `Block` operation (line 11 and 12). The `Scheduler` does periodic checks each millisecond, and when it reaches the required point in time it unblocks the `Schedulable` instance by invoking the `Release` operation (line 1 and 2).

```
 1 public Release : () ==> ()
 2 Release() == skip;
 3
 4 public WaitForNextPeriod : () ==> ()
 5 WaitForNextPeriod() ==
 6 (
 7   dcl now : nat := scheduler.GetTime();
 8   dcl offset : real := period - (now - lastSchedule);
 9
10   if offset > 0 then
11    scheduler.AddSchedulable(self,offset);
12   Block();
13 );
14
15 private Block : () ==> ()
16 Block() == lastSchedule := scheduler.GetTime();
17
18 protected Run : () ==> ()
19 Run() == is subclass responsibility;
20
21 thread
22   Run();
23
24 sync
25 per Block => #req(Block) <= #fin(Release)
```

Listing B.5: Synchronization operation for the `Schedulable` class

The implementation of the WCH, with both the `Scheduler` and the `Schedulable`, has allowed for modeling the jitter experienced within real JVM implementations and resembles the behavior of periodic treads in RTSJ.

B.2.2 RTSJ Implementation

Section 7.4.2 provided a code example of how the `TrackEvaluationHandler` class has been implemented using the RTSJ. In this section the implementations of the `EngagementHandler` and the `WeaponComHandler` threads will be presented.

Listing B.6 shows how the `EngagementHandler` thread has been implemented using the RTSJ.

```
1  private class EngagementHandler extends RealtimeThread {
2   public EngagementHandler() {
3    super( new PriorityParameters(17));
4           this.setDaemon(true);
5   }
6
7   public void run() {
8    while(isRunning) {
9     //Block until data is available
10    try {
11     trackEngagementQueue.waitForData();
12    }
13    catch(UnsupportedOperationException e){ aLog.error("", e); }
14    catch(InterruptedException e){ aLog.error("", e); }
15
16    //Read the data for the WeaponComHandler
17    final LatLongAltitude position =
18     (LatLongAltitude)trackEngagementQueue.read();
19
20    //For now we assume a single weapon which is always ready
21    //Spawn and start the NHRT WeaponComHandler in a parent scope.
22    nhrtMemory.enter(new Runnable() {
23     public void run() {
24      (new WeaponComHandler(position)).start();
25     }
26    });
27   }
28  }
29 }
```

Listing B.6: Java code from the RTSJ implementation of EngagementHandler

The thread extends the `RealTimeThread` class as defined by the RTSJ (line 1). This, among other things, allows the thread to have a priority of 17 (line 3), whereas standard Java threads are only allowed priorities of 1-10. The `EngagementHandler` thread runs inside a loop, until the EM component is removed from the T-Core framework (line 8). The thread is blocked until it receives a position from the `TrackEvaluationHandler` through the `trackEngagement-Queue` (line 11). When this happens it means that a hostile track has been detected inside a protected geographical area. The thread then extracts the position of the hostile track from the queue (line 17) and it should then localize an available weapon. However, this functionality has been left out of this initial implementation, instead it is simply assumed that a single weapon is always available. Therefore, after obtaining the position, the `EngagementHandler` thread spawns a `WeaponComHandler` thread which communicates with the weapon (line 24). The `WeaponComHandler` thread is a `NoHeapRealTimeThread` (NHRT) as defined by the RTSJ. Such threads are only able to execute within a scoped memory area, which does not have the heap memory area as its direct parent. As the `EngagementHandler` thread operates in

the heap memory area, the scoped memory area called nhrtMemory (line 22) is used as parent to spawn the WeaponComHandler thread and its own scoped memory area inside. The WeaponComHandler thread receives the position of the track to engage through its constructor (line 24).
Listing B.7 shows how the WeaponComHandler thread has been implemented using the RTSJ.

```
1  private class WeaponComHandler extends NoHeapRealtimeThread{
2   LatLongAltitude trackPosition;
3
4   public WeaponComHandler(LatLongAltitude position){
5    super(
6     new PriorityParameters(20),
7     new PeriodicParameters(new RelativeTime(5, 0)),
8     new LTMemory(1*1024*1024));
9    this.trackPosition = position;
10  }
11
12  public void run() {
13   long startTime = System.nanoTime();
14   long endTime = 0L;
15   weaponComStart[currentIteration] = startTime;
16   //Do a series of periodic timestamps
17   //to simulate real-time communication
18   for(int i = 0; i < 10; i++)
19   {
20    waitForNextPeriod();
21    //Do communication
22    endTime = System.nanoTime();
23    weaponComJitter[currentIteration][i] = endTime-startTime;
24    startTime = endTime;
25   }
26   weaponComEnd[currentIteration] = endTime;
27   currentIteration++;
28  }
29 }
```

Listing B.7: Java code from the RTSJ implementation of WeaponComHandler

As mentioned the WeaponComHandler thread is implemented as a NHRT (line 1). The thread has a variable for holding the position of the track which is to be engaged (line 2). The variable is set through the constructor (line 9) but is not currently used as this implementation does not communicate with real hardware. Instead the communication of 10 messages is simulated by measuring the timeliness of this periodic thread. The WeaponComHandler thread is made periodic by specifying its period of 5 ms in the NHRT's constructor (line 7). It is then possible to use the RTSJ defined method waitForNextPeriod (line 20) which blocks the thread until the next period begins. The advantage of this compared to the standard way of making periodic threads in Java using the Sleep method, is that there is no need for calculating how long the processing of the current iteration has taken. A series of timestamps is done through the execution in order to measure the timeliness of the WeaponComHandler thread.

B.2.3 Test Results

This section provides charts detailing the test results of the EM component in addition to those presented in section 7.4.2. The results should be viewed in connection with figure 7.7 (page 86) and table 7.1 (page 85), which describe the timings intervals covered by each of the chart figures B.4 to B.7.

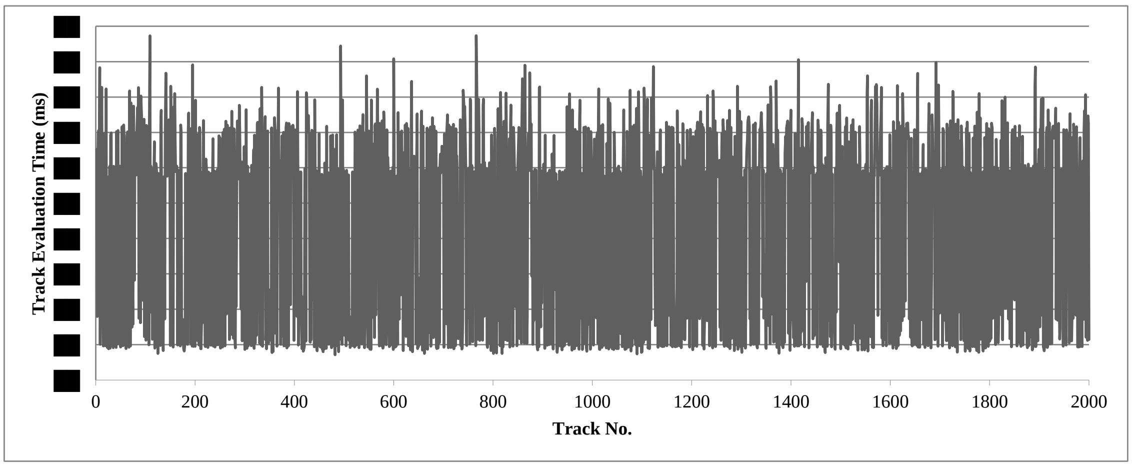


Figure B.4: Evaluation Duration for the `TrackEvaluationHandler`

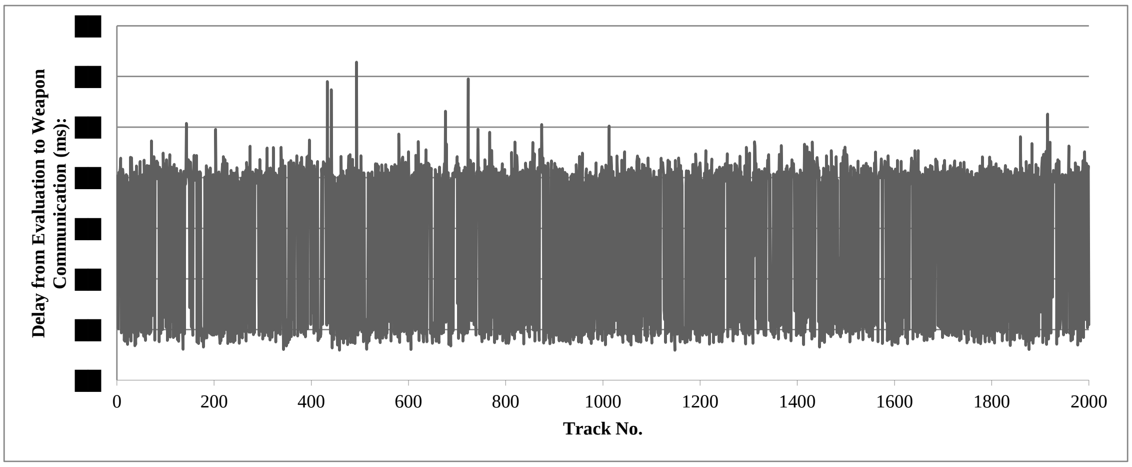


Figure B.5: Evaluation/Communication Delay Between the `TrackEvaluationHandler` and the `WeaponComHandler`

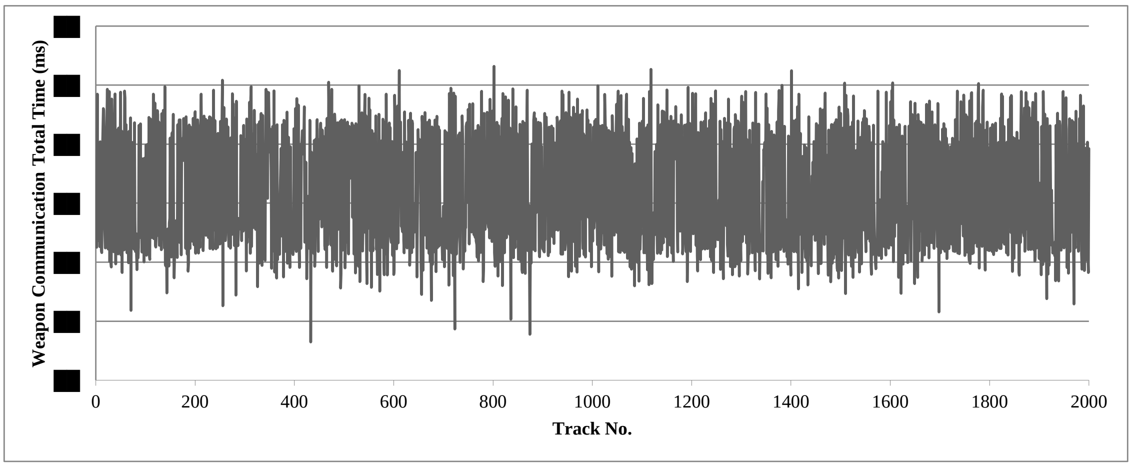


Figure B.6: Weapon Communication Total Duration for the `WeaponComHandler`

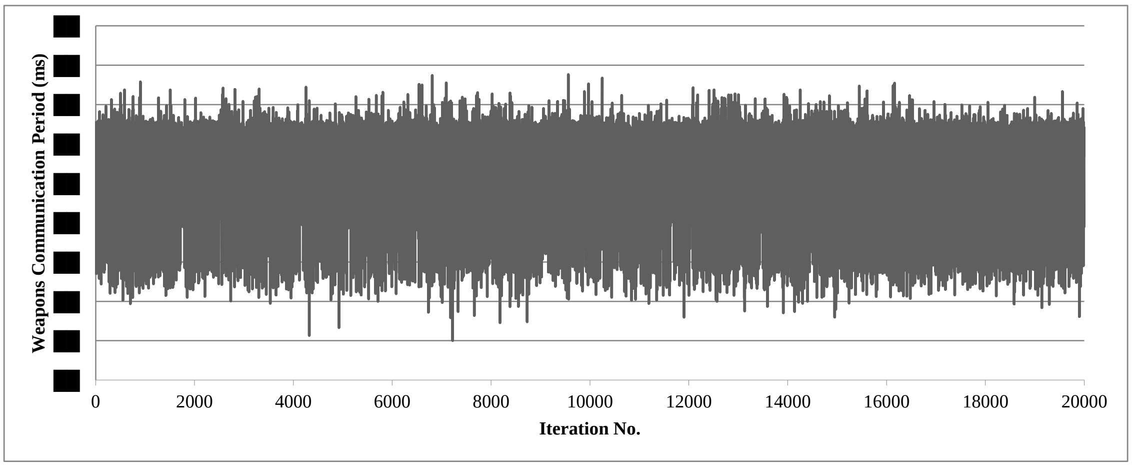


Figure B.7: Weapon Communication Period for the `WeaponComHandler`

Appendix C

Overture Real-Time Log Viewer

This appendix provides additional details about the re-development of the *Real-Time Log Viewer* (RTLV) plugin for the Overture tool (see section 5.3). This plugin has been developed in collaboration with Peter W. V. Jørgensen, which provide a similar version of this appendix in his Master's thesis [Jørgensen12].
The new design of the RTLV plugin, and the performance gain resulting from it, are described in sections C.1 and C.2 .

C.1. Design

When executing a VDM-RT model, a series of events are logged and timestamped by a component of the Overture tool, called the *RT Logger*. During execution, these events are triggered by various actions such as function calls, object creations and thread activations. So far these events have been written to a text file (the trace file), which make them readable to humans. However, these files can grow to immense sizes and contain up to thousands of lines. As a consequence, this makes it difficult for a human to use them for getting an overview of the entire execution. The RTLV plugin has facilitated this problem, by enabling graphical visualization of these events, as exemplified in figure 5.3 on page 44. The RTLV plugin offers three different view types for inspecting the logged events:

Architecture Overview: This is a simple overview of the CPUs and buses comprising the modeled system. From this view, it is possible to see how CPUs are connected via different communication buses.

Execution Overview: This is a detailed overview of the CPUs and buses illustrating thread swaps, operation calls and communication across the buses.

CPU Overview: This is a detailed view, provided for each CPU of the system. From this view, it is possible to inspect how threads execute code contained within different objects, and how they block on synchronous bus communication etc.

Within these views, it is possible for the user to scroll through all the logged events, and inspect the execution details at a particular point in time. This makes the trace file analysis much more manageable.

The following sections elaborate on the RTLV design. Section C.1.1 describes the differences between the old and the new design. Section C.1.2 describes the new design in details.

C.1.1 Old vs. New Design

The old approach of writing and reading the text files, containing the event data, is illustrated in figure C.1.

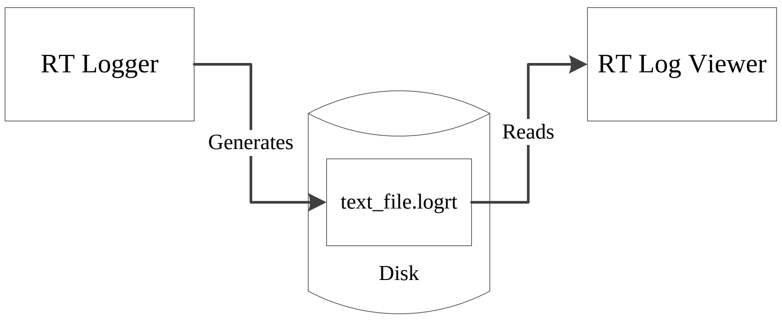


Figure C.1: Old RT Log Viewer

The problem with this approach is its lack of performance, which also motivated the RTLV plugin re-design. Using the old approach, the time it takes to read the large text files from the disk is not only slow, but also time-consuming to process the data structure representing the events and their relations. In addition to this, the code doing the processing of all the events was very inefficient. These issues resulted in a bad user experience, as the entire Overture tool would stall for minutes, before showing the overviews. In worst case, it would even result in the Overture tool crashing.
Therefore, the RT Logger and the RTLV plugin needed a re-design, in order to become efficient. Before the work of this thesis commenced, the RT Logger was re-designed, and re-implemented. This new implementation produces an object oriented data structure, used for drawing the different overviews efficiently. However, the RTLV plugin was not yet able to parse this data structure, and display the data graphically. Therefore, this functionality was implemented as part of this thesis. The new and more efficient approach to writing and reading events, is illustrated in figure C.2. The new data structure can be binary serialized and deserialized. This makes it possible to save

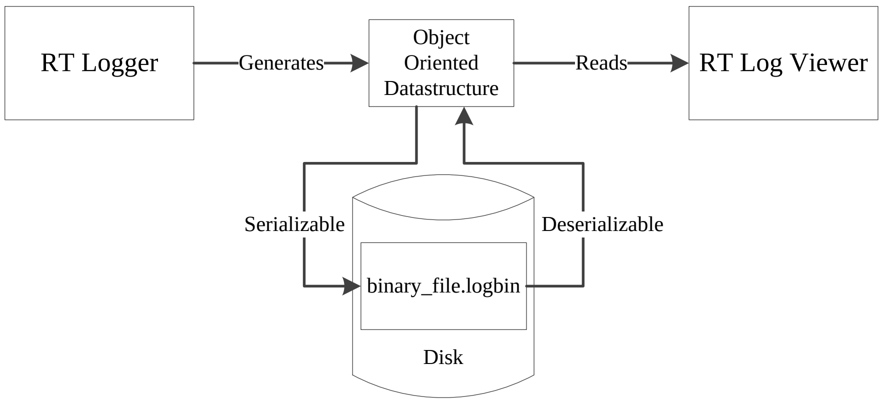


Figure C.2: New RT Log Viewer

this data structure to a file, or keep it in memory, in order to visualize it using the RTLV plugin immediately.

C.1.2 Detailed Design of the New RT Log Viewer

The new RTLV plugin design promotes a clear separation of concerns, and loose coupling between classes. This is done by defining three layers (as Java packages): *data*, *draw* and *view*. Each layer has its own area of responsibility, e.g. the classes within the *draw* layer, are responsible for drawing information specific to the user interface. The design is illustrated in figure C.3. Classes inheriting from the `TraceViewer` class, are part of the *draw* layer. The *data* layer is composed of classes, inheriting from `EventHandler` and `TraceData`. Finally, all top-level classes, such as the `VdmRtLogEditor` class, are part of the *view* layer. This architecture makes it easier to extend the RTLV plugin with additional functionality, as well as changing the data representations, drawing functionality etc.

The approach taken by the old RTLV plugin design, was to iterate through all events, when the trace file was loaded. This allowed the RTLV plugin to save the state of each data item (CPU, bus, threads etc.) at each point in time, but introduced severe performance overhead when loading the trace file. Instead the new design loads the binary file, and parses only the visible amount of events. However, when the user moves the inspection to another point in time, the RTLV plugin must determine the current state of each data item, at that specific time. This requires processing of all events, for that specific data item, up until the given time. For example, a thread may be active or inactive, based on events which occurred prior to the current time. To accommodate this, the new design saves all state information in a series of classes, managed by the `TraceData` class (`TraceThread`, `TraceCPU` etc.). The classes inheriting from the `EventHandler` class, are part of a *strategy-pattern* [Gamma&95], where the active event-handler is changed, based on the specific event being processed. The iteration and processing of events, are controlled by the `TraceFileRunner` class.

The event-handlers process the current event, updates the corresponding data item class (`Trace-CPU`, `TraceBus` etc.), and invokes the required drawing functions through instances of `Trace-Viewer`. These are also based on the strategy-pattern, where the active strategy is changed based on which view is selected (CPU, architecture or execution).

C.2. Results

The impact on the load time of the RTLV plugin overview, resulting from the new design, is illustrated in figure C.4.

It can be seen how the load time of the old RTLV plugin grows non-linearly, as the number of events to display increase. The line representing the load time of the new RTLV plugin, can be hard to see, as its load time is constantly low. This is due to the new design, which only parses and draws a small amount of events, when the RTLV plugin is loaded. The old approach was to load all events, when initiating the RTLV plugin. A disadvantage of the new approach, compared to the old one, is that scrolling and moving inspection between different events, can exhibit longer load times, as the new RTLV plugin loads the events as needed. However, practical experience has shown that this relatively small increase in load time does not inhibit the user experience of the plugin.

Therefore, the new version of this plugin heightens the usability by an increase in performance and load times. The design of the plugin has been changed to support a clear separation of logic concerns to promote reuse and additional optimizations in future releases.

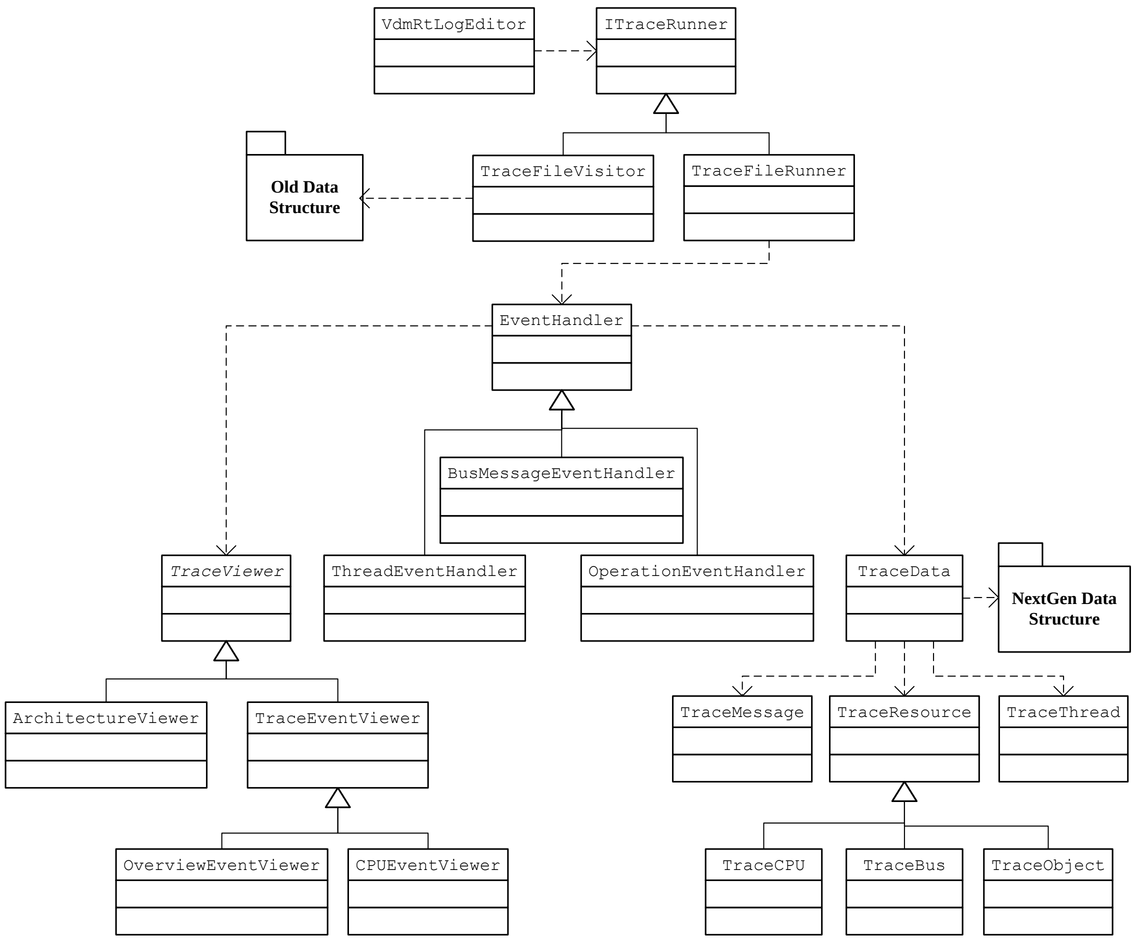


Figure C.3: Class diagram of the new RTLV design

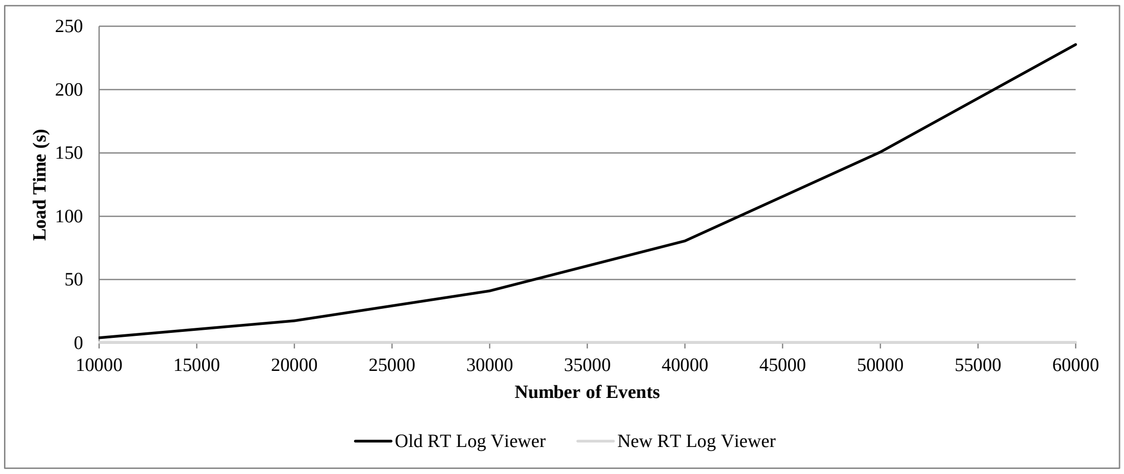


Figure C.4: New vs. old RTLV load times

Appendix D

Java Virtual Machine Analysis

This appendix provides a brief technical overview of the parameters used for comparing Java Virtual Machines (JVM) in chapter 6 including the two benchmark applications which are applied in three different tests as described in section 6.4.2. The first test consists of the *SPECjvm2008* (SPEC) benchmark [SPEC], the second of the *Collision Detector* (CD) benchmark [CD] and the third of both combined.
The basis for optimizing garbage collectors with standard Java is described in section D.1. The qualitative and quantitative parameters used for comparing JVMs is explained in D.2. Details of the SPEC benchmark is described in section D.3.1 and the CD benchmark is described in section D.3.2.

D.1. Optimizing Standard Java

This section explains the parameters used for optimizing the runtime behavior of the Car Controller on the two JVMs: The Oracle HotSpot and the Atego PERC Ultra. Section 6.3 describes the difference in jitter values for the Car Controller running with default settings on the HotSpot, compared to running with an optimized garbage collector. The settings used for the jitter results (see table 6.1 on page 52) are shown in table D.1 for the HotSpot and table D.2 for the PERC Ultra.

Parameter	**Description**
Xmx512m	Sets the maximum heap size to 512MB
Xms512m	Sets the initial heap size to 512MB
XX:NewRatio=10	Defines the ratio between young and tenured generations to 1:10

Table D.1: Configuration parameters for GC optimizing the HotSpot JVM

Parameter	**Description**
eager-jit	JIT compile methods as soon as classes are loaded
eager-link	Recursively resolve all referenced classes
region-size 1m	Sets the memory regions to 1MB each
num-regions 512	Defines a heap with 512 regions
gcprio 7	Sets the garbage collector priority to 7
gc 50	Activates garbage collector when the available heap is less than 50 percent
gcperiod 2500	Sets the garbage collector period in ticks
gcslice 0	Instructs the garbage collector to operate in FIFO scheduling mode allowing it to complete a full cycle each time (unless preempted)
timeslice 30	Sets the maximum number of ticks for each thread before they are preempted
tickperiod 500	Sets the number of microseconds per tick

Table D.2: Configuration parameters for GC optimizing the PERC Ultra JVM

D.2. Comparison

Table D.3 describes the grading scale which the analysis in chapter 6 is based upon. The JVMs are given grades from 1 to 3, where the minimum requirements are described in the table. The higher grade includes the requirements of the grades below.

Attribute	**Type**	**Grades**		
		1	2	3
Maturity	Qualiative	Proof-of-concept or research project and compliant with JVM Specification	Real-life deployments, regular updates and SDK support	Widespread usage and dedicated libraries
Specification Support	Qualiative	Java Language Specification support	RTSJ support	SCJ support
Scheduling	Qualiative	Relies on the OS scheduler	Own RT scheduler	Own RT scheduler with support for priorities exceeding standard Java priorities
Synchronization	Qualiative	No support for priority inversion avoidance	Support for the priority inheritance protocol	Support for the priority ceiling protocol
Memory Management	Qualiative	Heap memory	Real-time garbage collector	Support for memory areas unaffected by garbage collection
Determinism	Quantitative	Assigned relatively based on benchmark results e.g. the JVM achieving highest performance value is assigned 3 etc.		
Performance	Quantitative			
Initialization	Quantitative			

Table D.3: Scale for rating the attributes of the individual JVMs

D.3. Benchmark

This section describes the two benchmark applications used for extracting the quantitative attributes for each JVM. Section D.3.1 describes the SPEC benchmark and section D.3.2 describes the CD benchmark. Finally the results are illustrated in section D.3.3.

D.3.1 SPECjvm2008

The main purpose of the SPEC benchmark is to measure the performance of the JVM, the underlying hardware and operating system. The SPEC benchmark was chosen because it provides a good comparison of throughput across the three JVM's tested: Oracle HotSpot, Atego PERC Ultra and Aicas JamaicaVM.
The SPEC benchmark consists of several individual benchmark tests where the *compress* bench-

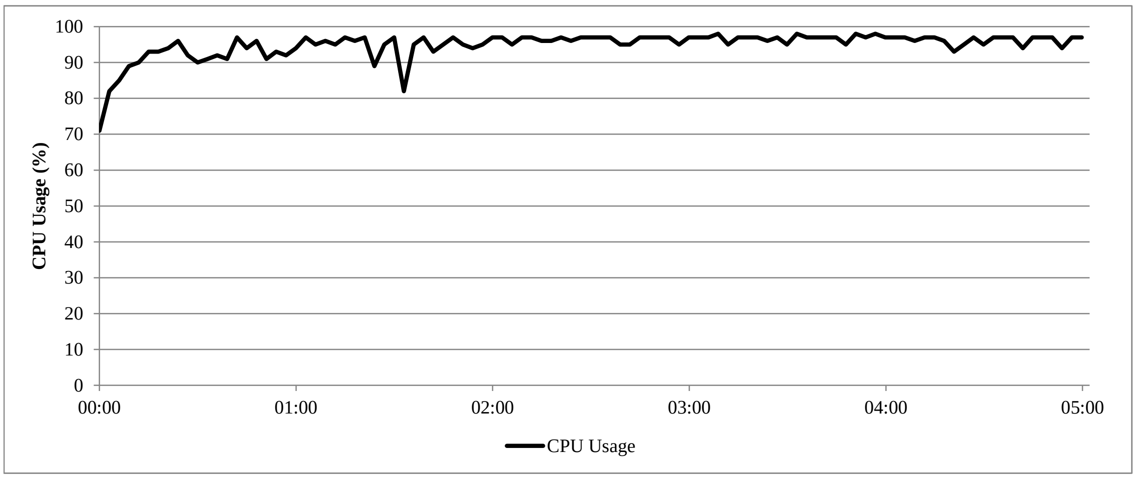


Figure D.1: CPU profile of the SPECjvm2008 Compress benchmark

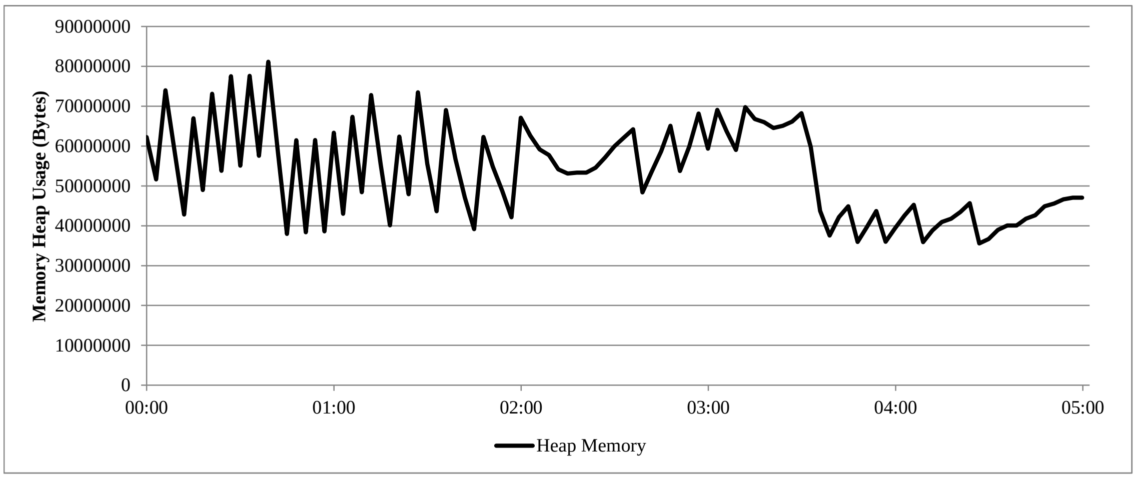


Figure D.2: Memory profile of the SPECjvm2008 Compress benchmark

mark was selected. The compress algorithm compresses data by searching for common substrings and replaces them with a predefined variable. The algorithm is deterministic, which mean that the exact same test is executed on all three JVMs.
The compress algorithm provides a high CPU load and memory consumption. The CPU profile is illustrated in figure D.1 and the memory profile in figure D.2..
The benchmark has been configured through a series of command-line parameters, where the most interesting are shown in table D.4. The column *test* describes in which of the three tests the corresponding parameter is used.

Parameter	Description	Test
compress	Selects and runs the compress benchmark	one and three
wt 0s	Warmup time. Specified to zero to begin test once loading is completed	one and three
ikv	Ignore Kit Validation. Specified to zero to ignore checksum validation	one and three
it 240s	Iteration Time. Specified to four minutes for each iteration	one
mi 5	Minimum Number of Iterations. Specified to five iterations	one and three
mi 100	Minimum Number of Iterations. Specified to 100 to make sure SPEC does not finish before CD iterations	three

Table D.4: Configuration parameters for the SPECjvm2008 tests

D.3.2 Collision Detector

The CD benchmark is an open source application that simulates an air traffic control system which must determine if monitored aircrafts are on collision course. The application targets both hard and soft real-time applications, and includes RTSJ support. CD consists of a *simulator thread* which generates a series of simulated radar frames. The frames are received periodically by a *detector thread* which computes a full 3D collision detection algorithm to detect potential airplane collisions. During execution the benchmark timestamps each periodically scheduling of the detector thread, which can later be compared with expected values to calculate the deviation (jitter).
The CD benchmark supports both Java and RTSJ which makes it ideal comparing the three JVMs. However, a few minor alterations were required to execute it on the three JVMs and extract the desired results:

JVM Support: The CD benchmark allows for building and executing the application on different JVMs. The initial version has been extended with support for the PERC Ultra and the Aicas JamaicaVM.

SPECjvm2008 Support: The initial version of the CD benchmark includes support for loading third party JAR files in order to simulating background noise. However, it was prepared for the SPECjvm98 benchmark which is now obsolete. The application has instead been updated with support for the new SPECjvm2008.

Fair Initialization: The benchmark has been updated with additional configuration parameters, e.g. one parameter to configure a delay for the detector thread in order to allow the SPEC thread to begin processing in test three.

Output Data: Additional configuration parameters have been added to allow the CD benchmark to be part of an automatic test environment. Here the CD benchmark is loaded with an output path, thus allowing it to run several consecutive tests and not override the results of the previous tests.

The altered version of the CD benchmark has been configured with a series of command-line parameters which are listed in table D.5. The table shows for which test the corresponding parameters have been used. Notice that the benchmark is configured to “presimulate”, thus generating all frames before the detector thread starts its periodic invocation. This is chosen as test two must determine the jitter without noise, and test three uses the SPEC benchmark to generate noise. The noise generated by the simulator is therefore ignored.

Table D.5: Configuration parameters for the Collision Detector benchmark

Parameter	**Description**	**Test**
col.bin	Select the col profile for input data. Contains 40 aircrafts and includes collisions	two and three
MAX_FRAMES 5000	Maximum number of frames generated by simulator thread	two and three
BUFFER_FRAMES 5001	Size of frame buffer for detector thread	two and three
DETECTOR_PERIOD 50	Period in milliseconds for detector thread	two and three
SIMULATOR_PRIORITY 5	Priority for simulator thread	two and three
DETECTOR_PRIORITY 10 *	Priority for detector thread	two and three
PRESIMULATE	Generate all frames before starting detector thread	two and three
USE_SPEC_NOISE	Activate the SPECjvm2008 benchmark in background	three
SPEC_METHOD "main"	Specifies the SPECjvm2008 function to invoke	three
DETECTOR_STARTUP _OFFSET_MILLIS 0	Detector offset in time calculations	two and three
DETECTOR_STARTUP _WAIT_MILLIS 60000	Detector wait time before starting CD benchmark	three

* For RTSJ execution with the JamaicaVM this priority is raised to 20 to exceed priorities of normal Java.

In addition to command-line parameters, the three JVMs were configured for each test. A fair configuration for comparison was chosen. Test one was defined with a heap size of 1024 megabytes, and both test two and three with 512 megabyte heap. The PERC Ultra has further been configured to allow it to execute the SPEC benchmark which uses multiple recursive invocations for its compress algorithm. This required raising the max stack size to 1024 kilobytes.

D.3.3 Benchmark Results

The three tests were applied to all three JVMs. The following figures show the jitter distribution for the detector thread in test two and three. Notice that test two is with the CD benchmark and no noise generation, where test three is with the CD but with the SPEC benchmark generating noise. Figure D.9 shows the result of test three with the JamaicaVM utilizing the real-time scheduling policy of the OS with real-time kernel patch. This is achieved by specifying the scheduling policy to FIFO through the JamaicaVM configuration and allows the JVM to preempt OS threads by increasing the priority of its own threads.

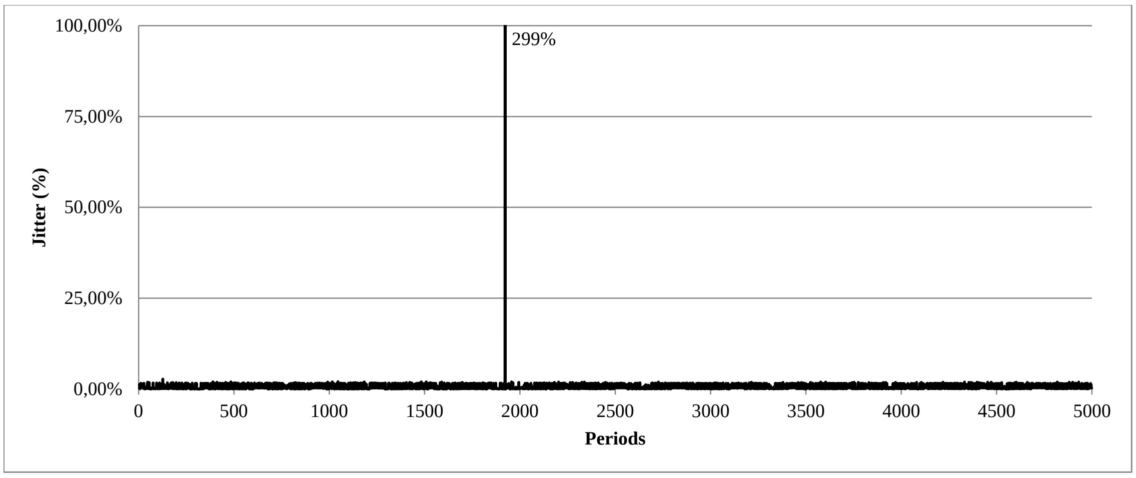


Figure D.3: Jitter distribution for the HotSpot test two

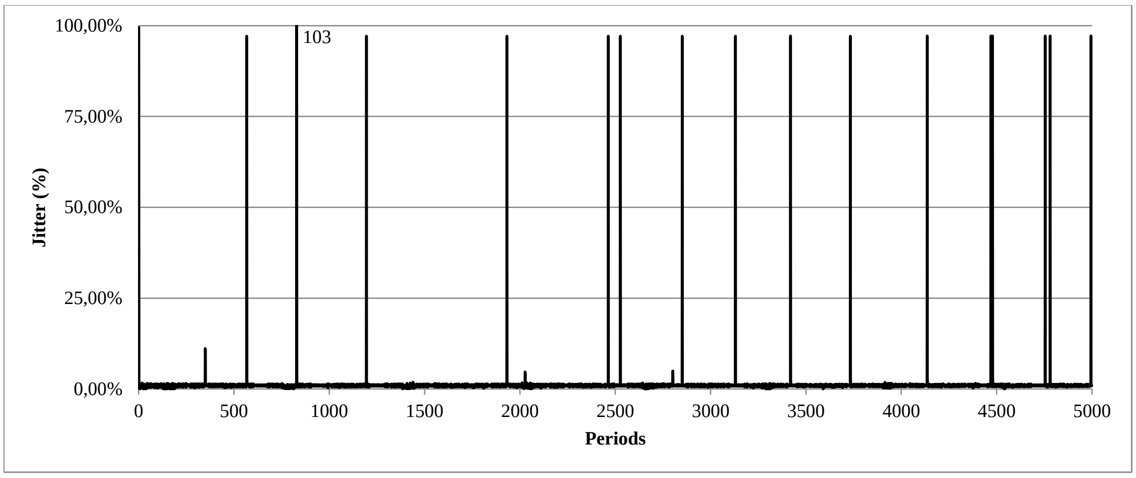


Figure D.4: Jitter distribution for the HotSpot test three

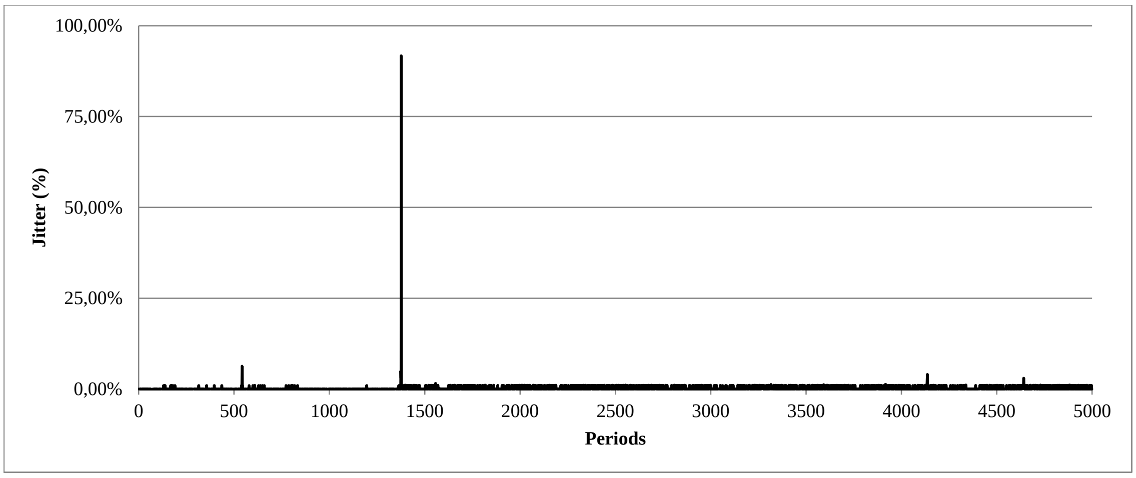


Figure D.5: Jitter distribution for the PERC Ultra test two

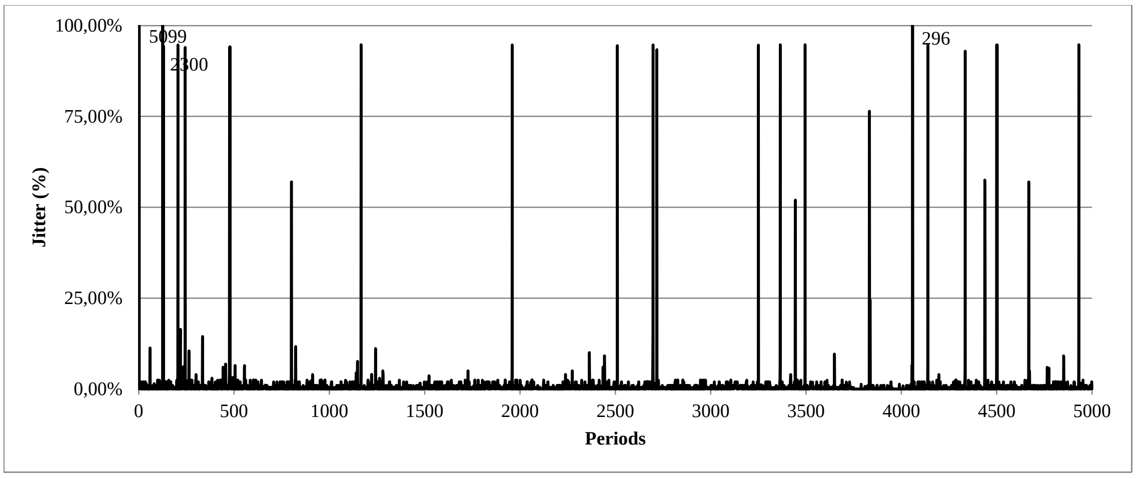


Figure D.6: Jitter distribution for the PERC Ultra test three

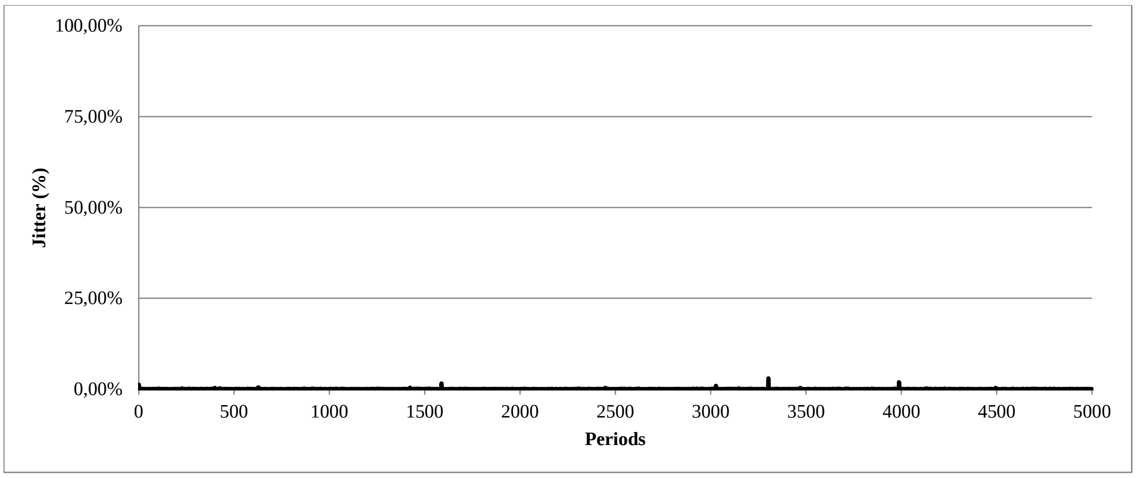


Figure D.7: Jitter distribution for the JamaicaVM test two with RTSJ

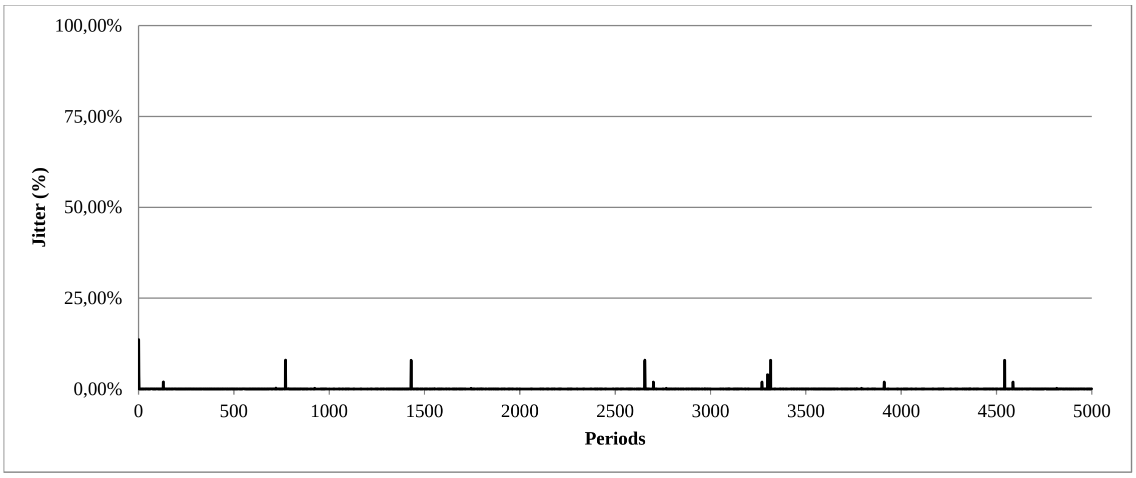


Figure D.8: Jitter distribution for the JamaicaVM test three with RTSJ

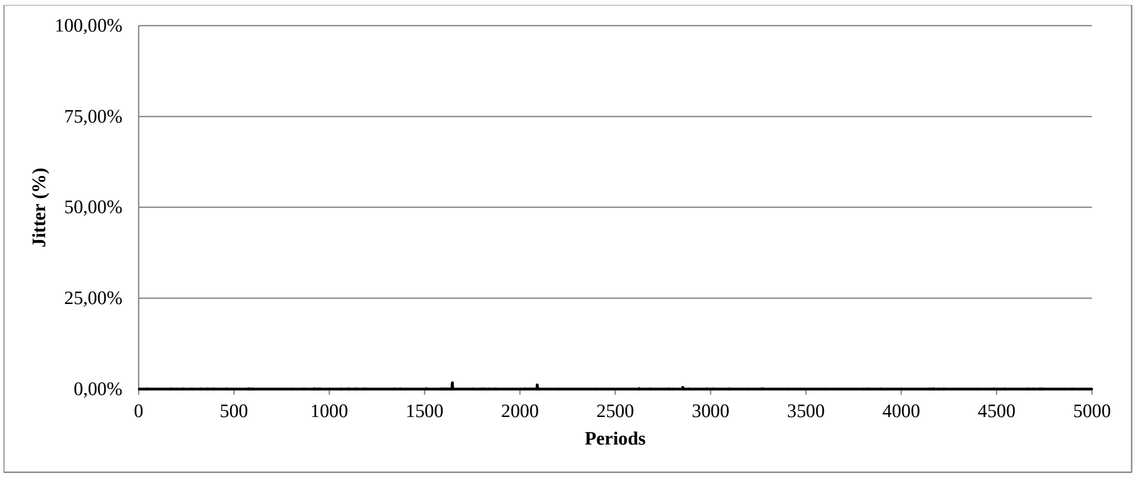


Figure D.9: Jitter distribution for the JamaicaVM test three with RTSJ and real-time RT Linux priorities

Appendix E

Distributed Computing

This appendix provides an overview of some of the available solutions for introducing real-time performance across distributed nodes. The notion of distributed communication is beyond the scope of this thesis. However, this appendix serves as an input for further investigation and analysis as distributed computing is an essential feature of many modern Java applications i.e. they use a network of interconnected nodes to achieve a common goal. Distributed computing is widely supported by standard Java, where developers have several options to implement inter-process communication from a high abstraction layer. This is typically achieved through the integrated Java *Remote Method Invocation* (RMI) or third party middleware's such as *Common Object Request Broker Architecture* (CORBA) or *Data Distribution Service* (DDS) [Wellings&02]. In contrast to typical distribution services, where a high abstraction layer is often the main focus, real-time applications require a more stringent control of Quality of Service (QoS) parameters such as high performance and dependability.

The Java community and industry, offers several commercial and open-source platforms for distributed real-time communication. These can be described by the following categories each with different purpose and characteristics [Higuera-Toledano&12]:

Control-Flow: Is the concept of distributing both application data and the point of execution between nodes in a simple request/response model. Java RMI and other solutions based on *Remote Procedure Call* (RPC) belong in this category.

Data-Flow: This covers distribution of data with no point of execution among entities. This include systems based on the publish/subscribe paradigm such as DDS.

Networked: This is a category of entities which have no clear execution point and data can be exchanged both synchronous and asynchronous.

Available solutions for standard Java cover all of the above categories, but do not provide the necessary features for real-time implementations. Such features may include distributed real-time threads, scheduling of remote invocations, handling of remote memory etc.

The following section describes an official language extension in section E.1, relevant middleware solutions in section E.2 and real-time transport solutions in section E.3.

E.1. Distributed RTSJ

The RTSJ (see chapter 3) does not address the subject of introducing distributed computing in real-time systems, and does not describe how to guarantee end-to-end predictability. The main effort towards a formal specification for distributed real-time computing in Java, is in progress under the ^T^*Java Community Process* with the title JSR-50. The specification named *Distributed Real-Time Specification for Java* (DRTSJ) is led by an expert group with members of industry, and was initially formed in April 2000. The Expert Group has formed a draft of the DRTSJ, where some of the essential features are summarized in the following:

Distributable Threads: Includes a new thread named `DistributableThread`, of which instances can span across multiple nodes. A special feature is the definition of an *active head* which is the initial execution point, and receiver of potential failure or exceptions.

Scheduling: Supports the `Schedulable` interface from RTSJ but provides a user-defined scheduling algorithm, instead of a JVM-defined scheduling algorithm.

Serializable Classes: Several important classes specified by RTSJ has been marked `Serializable` by the DRTSJ in order to support RMI on objects.

Unfortunately the current status of the DRTSJ is unfinished and marked as inactive. However several third-party middleware solutions are available each with specific purpose and characteristics.

E.2. Middleware

Some middleware implementations try to add features from RTSJ to existing distributed solutions, leaving a non-RTSJ-compliant platform. Others try to merge their features, for distributed communication, into the RTSJ. As a result there are two overall strategies for distributed real-time Java; Middleware communication with an RTSJ flavor, or an RTSJ implementation with additional features for distributed communication. Choosing the correct strategy is a difficult task since several different solutions and platforms exist, and it is beyond the scope of this thesis to describe them all. However a few of the most promising efforts is described in brief:

RT-CORBA: An enhanced version of CORBA from the Object Management Group (OMG) intended for real-time applications, which provides the traditional features of cross-language communication. Mapping RT-CORBA to RTSJ is not an easy effort, and requires some balancing between the two worlds, i.e. RT-CORBA encourages the use of `mutex` for synchronization, and only provides one real-time thread implementation where RTSJ uses the **`Synchronized`** keyword and provides two real-time thread implementations. This solution is within the control-flow category.

RTZen: CORBA and RT-CORBA have received criticism for introducing significant overhead, which has limited its deployment in the industry. However the RTZen project, from the University of California, tries to provide an *Object Request Broker* (ORB) implementation to RT-CORBA. RTZen is designed to comply with the RTSJ while keeping the footprint and processing overhead at a minimum. As with RT-CORBA, RTZen also falls within the control-flow category.

DREQUIEMI: Another approach, also within the control-flow category, is DREQUIEMI which tries to optimize the existing RMI framework towards real-time. DREQUIEMI, from the Universidad Carlos III de Madrid, is RTSJ compliant with additional extension to support distributed computing. Some of these features include a new type of memory object called *No-heap remote objects* to suppress garbage collection on certain remote objects. Additionally they included a *time-triggered communication* abstraction to allow the developers to use a periodic communication protocol like CAN or FlexRay.

Open Splice: This is a DDS model and falls within the data-flow category. Open Splice is an open source implementation of the OMG DDS specification and currently maintained by PrimTech. It allows for a distributed publish/subscribe model and provides a rich set of QoS attributes as well as data filtering.

E.3. Real-Time Transport

All relevant real-time communication implementations for Java and RTSJ are unable to guarantee any timing requirements, if the underlying network protocol does not provide the necessary predictability. One widely used protocol in modern network is TCP/IP which, per default, does not guarantee any timing boundaries on message delivery, nor does it guarantee end-to-end response times. Many alternatives have been proposed in the literature including low level protocols such as TTA, CAN or FlexRay. Another approach motivated by reuse of existing Ethernet setups, have been developed using specialized routing equipment. This approach resulted in a specification for *Real Time Ethernet*, a solution well suited for integrating real-time components, into an existing non-real-time distributed environment.

Mads von Qualen and Martin Askov Andersen, A Methodology for Transforming Java Applications Towards Real-Time Performance, 2013

Department of Engineering
Aarhus University
Edison, Finlandsgade 22
8200 Aarhus N
Denmark

Tel.: +45 4189 3000