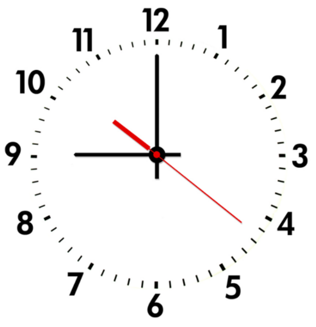
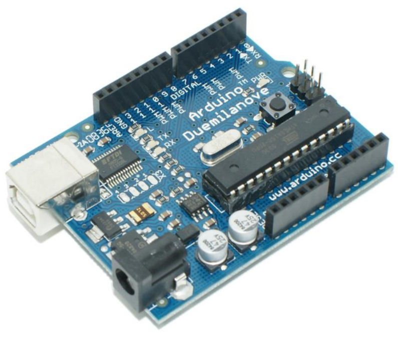
# Java for Real-Time Embedded Systems





Software 9 - 2010

AALBORG UNIVERSITET

*Studenterrapport*

**Aalborg University**
**Department of Computer Science**
Selma Lagerlöfs Vej 300
9220 Aalborg
Telephone:(45)96358080
http://www.cs.aau.dk

**Title:**
Java for Real-Time Embedded Systems

**Theme:**
Database and Programming Technologies

**Project time frame:**
SW9, 1st September 2010 - 10th January, 2011

**Project group:**
d507a

**Group members:**
Christian Frost
Casper Svenning Jensen
Kasper Søe Luckow

**Supervisors:**
Bent Thomsen
Anders Peter Ravn

**Abstract:**
This project is a study of embedded hard real-time systems with focus on using the Java programming language.

The introductory study of embedded hard real-time systems examines the characteristics these systems exhibit and how they differ from ordinary systems development in terms of platforms, operating systems, and programming languages. Additionally, inherent concepts forming the cornerstones in real-time systems development are highlighted with focus on determining Worst-Case Execution Time (WCET) of tasks and performing schedulability analysis.

The knowledge obtained in the area of embedded hard real-time systems forms the basis for understanding how the Java programming language can be applied to develop such systems, and the challenges encountered in the process.

Currently, there exists a number of Java profiles targeted at real-time embedded systems development. These will be evaluated according to their advantages and disadvantages. The evaluation is based on implementations of a mine pump which is a classic example of an embedded hard real-time system. One of the implementations is deployed on the Java Optimized Processor (JOP), which together with a LEGO construction of the mine pump represent a physical example. The timing correctness is evaluated using the WCET Analyzer (WCA) and TIMES tools. Finally, a schedulability analysis is conducted using utilisation-based schedulability analysis and Response Time Analysis (RTA) to apply the theoretical knowledge in this field.

**Copies:** 6

**Total pages:** 166

**Of this Appendices:** 24

**Paper finished:** 10th of January 2011

# Preface

This report is written by three software-engineering students attending the $9^{th}$ semester at Aalborg University as part of their semester project. The project was commenced on the $1^{st}$ of September 2010, and finished on the $10^{th}$ of January 2011.

During the project, help has been received from a number of people whom we would like to thank. First of all, our supervisors Bent Thomsen and Anders Peter Ravn, have been helpful with advice regarding the project and the content of the report. Besides, Bent has engaged in valuable discussions concerning the field in general. Finally, we would like to thank Thomas Bøgholm and Hans Søndergaard whose primary contributions have been in understanding subjects of Java in real-time embedded systems.

This report will concentrate on subjects related to computer science. Therefore, it is assumed that the reader has equivalent knowledge in the field of computer science, as that of a $9^{th}$ semester software engineering student.

Two types of source references are used throughout the report. One is a reference placed after a period which refers to the given section. The other type of reference is placed before a period which refers to the particular sentence or word. The sources of the references used throughout this report, can be found in the bibliography at the end of the report.

The development method used throughout the project is described in Appendix C.

The source code of the mine pump can be obtained at `http://www.cs.aau.dk/~luckow/minepump.tar.bz2`.

Aalborg, January 2011
- d507a

Christian Frost

Casper Svenning Jensen

Kasper Søe Luckow

# Contents

# 1 Introduction

It is evident that embedded systems form an integral part of our daily lives. In what seemed unimaginable some decades ago, many appliances that were manually operated are now being redefined in automatically operated embedded systems. The appliances range from daily household appliances, such as dishwashers and vacuum cleaners, to life-critical systems, such as defibrillators and airbags. Thus, these systems both bring value into our lives by easing many elementary tasks, and in some cases even impacting our health. In today's world, the absence of such systems may seem absurd, because we have become extremely dependent on them.

Besides embedded systems being an integral part of today's world, a subset called real-time embedded systems is also of evident importance. Such embedded systems have real-time requirements, that is, requirements describing a timely response, called a deadline, to an event that often occurs externally to the embedded system. In some cases, missing a deadline can have catastrophic consequences potentially putting human lives at risk or having paramount financial costs. Systems whose failure can be life-threatening are termed *safety-critical*. Due to the importance of upholding deadlines, great effort is put into ensuring both the functional and timing correctness of these systems.

Today, we frequently interact with computer systems without explicitly reasoning about them. For instance, the emergence of embedded systems has spawned systems such as the *vehicle tailgate detection system* used for automatically maintaining a safe distance to the car ahead. The tendency of incorporating embedded systems for such purposes is a product of the vision of *ubiquitous* or *pervasive computing* referring to computer systems that surround us and are designed for a specific purpose (Qing Li, 2003).

The main concept in ubiquitous systems is that they remain transparent to us. However, we may become aware of their presence in case they are not functioning correctly which possibly requires manually interacting with the domain in question. Malfunctioning systems can be a consequence of introduced errors in the application code and the frequency of these may be related with the choice of programming language. Hence, using the appropriate programming language is important, especially when considering, that many such systems potentially will become an integral part in safety-critical applications, where upholding real-time requirements, is an imperative for correct operation.

Programming real-time embedded systems is mainly done in the C programming language. There are many reasons for this choice such as higher expressiveness, control of the system, and facilities for easy interaction with hardware. All of these are desirable for embedded systems development. However, the C programming language poses significant responsibilities on the programmer to avoid buffer overflows, security-related issues etc. Evidently, such errors must be avoided considering that the application may serve a safety-critical function. Further evidence for the inapplicability of using relatively low-level programming languages is by considering that the need for real-time systems still increases. This causes the desire to improve portability and enhance productivity, among others. These are desirable factors because they form some of the cornerstones in the total production cost which naturally is kept as low as possible due to many competitors wanting to maintain their market positions. Conclusively, these factors conflict with the tradition of programming real-time embedded systems in C because these factors are not promoted in C. (Burns and Wellings, 2009)

Taking these deficiencies into account provides an incentive for using high-level programming languages such as Java. The incentive for using Java can partially be explained by its inherent goal of being secure, robust, and portable. In addition, Java contains many language constructs that promote abstraction thereby achieving higher productivity compared to relatively low-level languages. The primary reason why Java has not been considered for real-time systems development is that Java is missing a number of concepts which are necessary for real-time systems such as the notion of deadlines and high resolution real-time clocks. However, a significant effort has been put in the Java community of developing the Real-Time Specification for Java (RTSJ) and related real-time Java profiles to incorporate these concepts.

The incentive for using Java for real-time systems development is further enhanced by considering that it has gradually become the most popular programming language over the last 15 years (TIOBE-Software, 2010). Among others, this can be explained by the increased focus on high-level programming languages in teaching at educational institutions. Therefore, it is natural to expect that, in general, newly educated students are more proficient in using such languages. Programmers proficient in low-level programming languages will hence no longer be the norm effectively making them costly resources that conflicts with keeping production costs at a minimum. In total, Java for real-time systems development is an active research area that is interesting to take part of due to its evident potential for revolutionising current practices in the field of real-time systems development.

This project emphasises on two topics divided into two respective parts. The first part gives a survey of the field of real-time embedded systems and related concepts. The purpose here is to obtain a basic understanding of both the established knowledge and current research in the field. The second part emphasises on the Java programming language for real-time embedded systems, and covers topics of this active research field. The knowledge gained is used as foundation for constructing a LEGO model of the real-time mine pump text-book example. Some of the available real-time Java profiles are used for implementing the logic of the mine pump. These implementations are subsequently evaluated and discussed according to their usage and incorporation of real-time concepts.

## 1.1 Learning Goals

Due to being an academic project, it must fulfil the learning goals stated in the study regulation for a 9th semester software engineering project. These learning goals are vaguely formulated, hence, we have decided to define concrete learning goals. The reason for this is to concretise and give a concise description of the objectives of the project. The fulfilment of the concrete learning goals thereby implicitly fulfil the learning goals of the study regulation.

The overall learning goals dictated by the study regulation together with our concrete learning goals are described in the following. The learning goals written in *italics* are taken directly from the study regulation whereas the others are our concrete learning goals.

1. *Independently identify, formulate and analyse the chosen problem.*
    (a) Get a general understanding of real-time systems with emphasis on Worst-Case Execution Time (WCET) analysis and schedulability analysis.
    (b) Be able to account for the chosen problem of real-time Java.
    (c) Gain knowledge in the problem of real-time Java.
    (d) The analysis of the problems of real-time systems must enable the identification of directions for further research in the field.
2. *Demonstrate thorough knowledge of relevant theories and methods within the chosen problem.*
    (a) Analyse and compare different real-time Java profiles.
    (b) Obtain knowledge in practically applying WCET analysis and schedulability analysis on Java real-time programs.
3. *Relevantly relate the chosen problem to the field.*
    (a) Be able to relate the obtained knowledge of real-time Java with real-time systems in general.
4. *Be able to identify relevant scientific, theoretic and/or experimental methods for illustrating the chosen problem.*
    (a) Illustrate problems of real-time Java through a practical application.
    (b) Illustrate the problem of conducting WCET analysis and schedulability analysis for programs written in Java by applying them to a practical example.

## 1.2 Report Overview

The remaining report is structured into the following five parts:

**Part I: Real-Time Systems in General** This part describes concepts and terms of real-time systems in general. The descriptions serve to establish a fundamental knowledge required to understand the subsequent topics.

**Part II: Real-Time Java** This part emphasises on Java for real-time systems development.

**Part III: Mine Pump** The knowledge obtained in the previous parts are applied to a practical implementation of a real-time Java application.

**Part IV: Conclusion** The experience gained by the practical implementation is reflected upon, and the project is concluded. Finally, a set of future directions is listed.

**Part V: Appendices** This part contains the appendices referred to during the report.

# Part I

# Real-Time Systems in General

# 2
# Real-Time Embedded Systems

The purpose of this chapter is to highlight the essential concepts and characteristics of real-time embedded systems. The chapter serves to establish fundamental knowledge required to understand the subsequent analyses.

The following sections examine real-time embedded systems in general which involves examining the different classes of these, their characteristics, real-time platforms, and special characteristics to be aware of when developing such systems.

## 2.1 Definition of Real-Time Embedded Systems

As the name implies, a real-time embedded system is a combination of a real-time and an embedded system. To clearly understand this combination, the constituents are initially described in the following.

### 2.1.1 Embedded Systems

There exist various definitions of embedded systems and what they encompass. We will use the term *embedded systems* to denote the type of computing systems that are highly dependent on the underlying hardware to serve a dedicated functionality. An example of an embedded system could be a network router containing a special-purpose processor designed for executing routing algorithms and interface with the network ports. Many embedded systems are part of a larger embedded system which leads to the definition of the *embedding system* which is a system that can be comprised of a number of coexisting embedded systems. An embedding system could for instance be a digital set-top-box where the A/V decoder is an embedded system responsible for decoding the multimedia stream to produce sound and video frames. The A/V decoder operates in conjunction with another embedded system; the transport stream decoder that is responsible for providing the correct multimedia stream from the satellite to the A/V decoder.(Qing Li, 2003)

### 2.1.2 Real-Time Systems

A real-time system responds to one or more external events in a timely manner. Formally, a real-time system can be described using control theory. Specifically, real-time systems can be seen as an instance of the closed-loop control. A closed-loop control makes decisions based on logic and feedback of the particular environment under its command in a timely manner. This environment is termed the *plant* in control theory. The feedback from the plant is obtained from output information of the plant. The output can be obtained using sensors. Based on this output, the closed-control loop generates a decision which then is used as input to the plant, affecting its state e.g. by using actuators. This generates a new feedback which is input to the closed-loop control and the cycles starts over again.(Barr, 2002)

Overall the structure of a real-time system using the closed-loop control is illustrated in Figure 2.1.

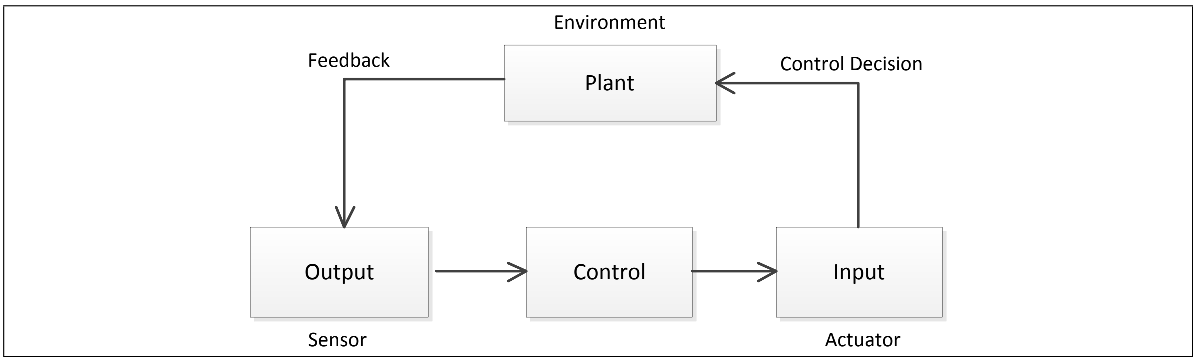


**Figure 2.1:** A real-time system represented by a closed-loop control.

An example of a real-time embedded system using the closed-loop control is given in Example 1.

**Example 1 (Real-Time System Structure)**
To understand the structure of a real-time system, consider a fluid-control system incorporated in a pipe. It is important that the flow of fluid through the pipe is constant, in order to ensure safe operation of components placed after the pipe. To regulate the flow to obtain a constant pressure of fluid, the pipe incorporates a sensor in the form of a flow meter and an actuator in the form of a valve. Both components interact with a processing unit which contains logic that regulates the valve according to the flow meter readings. In this case, the flow meter is the output and the valve is the input in the closed-control loop. The processing unit represents the control and the water pipe and water is the plant.

The example emphasises two important criteria: that a real-time system must produce correct computational results, referred to as functional correctness, and make the computational result or draw a conclusion at a predefined instant within a deadline, referred to as timing correctness.(Qing Li, 2003)

It is important to note that real-time in this context does not refer to fast response time or low latency, but instead it refers to complying with timing requirements. In this relation, it is also important to note that in some circumstances, the value of the computational result may be useless if it is obtained too early.(Qing Li, 2003)

### Hard, Soft, and Firm Real-Time Systems

Real-time systems can be classified into three different types depending on their application. Specifically, a real-time system can either be classified as being either *hard*, *soft*, or *firm*. The first two classifications differentiate in how they tolerate not complying with the timing requirements, the applicability of results obtained after a deadline, and severity of the penalty incurred for not upholding the timing requirements (Qing Li, 2003). The latter classification is a hybrid of the two others.

Hard Real-Time Systems Hard real-time systems have zero level of tolerance for missed deadlines. In this case, computed results generated after a deadline has been reached, are not useful and the penalty may be catastrophic. Hence, for hard real-time systems, a great effort is put into predicting beforehand that the real-time system can uphold its timing requirements. Systems whose failure may be life-threatening are also known as *safety-critical systems*. Examples of hard real-time systems could be the fluid-control system where safe operation of components placed after the pipe depend on a constant flow.(Qing Li, 2003)

Soft Real-Time Systems This classification represents real-time systems that strives to meet required deadlines however with a certain degree of tolerance. If tasks exceed their deadlines, the penalty does not yield a catastrophe and the result is still useful, although with a decreased value.

Another important factor is how to handle missed deadlines. For instance, it may be the case that a missed deadline results in initiating a recovery process to compensate for eventual misbehaviour. A concrete example of a soft real-time system is a DVD player decoding video and audio streams. This process may cause the decoder to miss deadlines and the result is video and audio being distorted because of delays.(Qing Li, 2003)

Firm Real-Time Systems Firm real-time systems are a combination of both hard and soft real-time systems. That is, it is acceptable to miss a deadline, as for soft real-time systems. However, if the deadline is missed, the result cannot be used as with hard real-time systems. Besides the timing requirement, firm real-time systems can also have an associated probabilistic requirement dictating a specified success rate. An example of a firm real-time system is a video decoder. In contrast to soft real-time systems, a firm real-time system will drop the video frames if they are not decoded within their deadlines.(Burns and Wellings, 2009)

### Types of Real-Time Tasks

In a real-time system, a task denotes a process or thread that is released by the system. Real-time systems consist of three different types of tasks: periodic, aperiodic, and sporadic.

Periodic A task is periodic if it is released within fixed time intervals, known as its period (Burns and Wellings, 2009). Considering the aforementioned example of maintaining a constant flow of fluid through a pipe, the processing unit could use a periodic real-time task to periodically request sensor readings from the flow meter.

Aperiodic Aperiodic tasks are released at unpredictable time intervals. Hence, this type of tasks is not desirable for hard real-time systems, since it cannot be guaranteed that theirs and other tasks' deadlines are met. However, servicing the aperiodic events can have a predictable deadline and WCET. To illustrate why deadlines of aperiodic tasks cannot be guaranteed, consider the example where a task is continuously released faster than its WCET. In this case, the deadline of the task cannot be upheld (Burns and Wellings, 2009). In the running example, an aperiodic task could be used to submit information from the flow meter whenever it detects a certain threshold of fluid pressure is exceeded.

Sporadic Finally, the tasks can be sporadic, which means that a known set of external events are separated by a minimum inter-arrival time. In the running example, this could be the case if the flow meter only submits information whenever a certain threshold is exceeded but consecutive readings can only be performed with a time interval such as $200ms$. Notice, that in case more events occur within the time period than defined, these can be ignored (Qing Li, 2003). Also note that in some cases, sporadic tasks can be problematic because situations may arise where it is uncertain whether external events in fact occur with a minimum inter-arrival time. Consider the example where the specification states that sensor values are retrieved at known time intervals from the sensor. If, however, the sensor aperiodically sends readings, it evidently causes problems due to the specification being incorrect on that matter. Hence, whenever sporadic tasks are used, one must ensure that they are correctly used (Bøgholm et al., 2009).

## 2.1.3 Real-Time Embedded Systems

A special kind of embedded systems is distinguished by incorporating the requirement of responding to events in real-time. This category of embedded systems is referred to as *real-time embedded systems*. The two systems are not equal but rather the class represents the intersection set of embedded systems and real-time systems as depicted in the Venn diagram in Figure 2.2.

In the following, an example of a system belonging to each of the sets are described by referring to their respective element of the figure.

A An electronic garage door system which does not necessarily have any real-time requirements belongs to the *embedded systems* set.

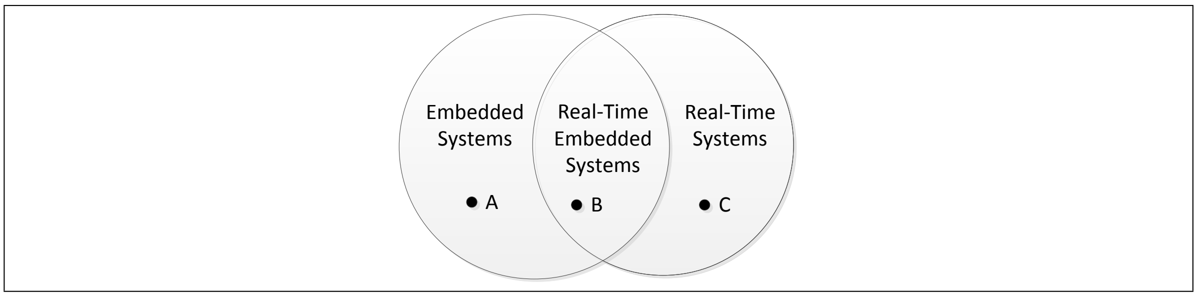


**Figure 2.2:** Real-time embedded systems are represented as the intersection set of the set of real-time systems and the set of embedded systems.

B The aforementioned example regarding the fluid-control system is a *real-time embedded system*, since it incorporates elements from both the definition of real-time and embedded systems. The fluid-control system operates under timing requirements and is an embedded system due to serving a single specific purpose and because it is part of a larger system.

C An electronic trading platform belongs to the *real-time systems* set because it consists of large platforms servin g a variety of purposes such as bank transfers and stock exchanges that must be handled in real-time.

In the remaining chapters of the report, the term *real-time systems* denotes hard real-time embedded systems, if not stated otherwise.

## 2.2 Characteristics

This section serves to highlight the essential characteristics of real-time systems. These are important areas one needs to be aware of when developing real-time systems, because they are significantly different from ordinary systems development. The following is primarily based on Burns and Wellings (2009) and Wellings (2004).

High Complexity Many real-time systems interact with the real world in terms of retrieving input and possibly generate a response to change the state of the environment in which the system is situated. Naturally, the real world is complex and hence the real-time system must incorporate these accordingly. A real-time system responsible for monitoring the climatic state of an environment needs for instance to incorporate many physical phenomena including humidity level, temperature, and air flow combined with their respective interconnections. Also, some real-time systems are distributed meaning that many systems must interact in a timely manner which again introduces high complexity (Wellings, 2004). Furthermore, complexity can be introduced in relation to creating generic components purposely targeted at component reuse in different contexts. The complexity lies in being able to interact with the environment consistently regardless of the given context. For instance, a real-time system responsible for controlling the airbag mechanism may be subject to reuse in other cars but it may be that the component needs to interact differently with the other embedded systems in the car.

Dependable A dependable system, as defined by Laprie et al. (1995), is a system which has a number of characteristics such as reliability, safety, and availability. Such systems control domains which are depending on the successful operation of the system and would need to run with a certain amount of reliability and availability to ensure completion of its tasks. A number of actions can be taken to avoid failures. One approach is to prevent faults by ensuring correct operation of the system through analysis. This requires the operation of the system to be predictable, such that the system's actions, both functional and timing, can be determined and analysed. One factor which limits predictability is the high complexity which real-time systems can exhibit.(Wellings, 2004)

Concurrent Control of Separate System Components Sometimes, real-time systems are responsible for monitoring multiple sensors. Since the measured environmental factors are expected to change in parallel, monitoring is supposed to be done likewise. Therefore, it is necessary to have support for concurrency on real-time systems. Different solutions for accommodating this exist. These are either to rely on the support for concurrency in a Real-Time Operating System (RTOS), e.g. using POSIX threads known from the C programming language or by using programming languages, such as Java, supporting concurrent programming.

Timing Facilities Real-time systems rely on timely interaction with their environment. Hence, they need facilities to respond to events before a deadline. Because of this need, it must be possible for the programmer to specify certain information regarding timing for the tasks to be performed. This timing information is described as being:(Burns and Wellings, 2009)

- The time at which a task is released.
- The time at which the task reaches completion.
- How to respond if not all timing requirements are met.
- How to respond when timing requirements are changed dynamically, e.g. as a consequence of mode changes.

Interaction with Hardware Interfaces Real-time embedded systems need to interact with the real world. This could be by having sensors monitoring the environment and actuators changing the state of it. This means that such systems must communicate with the connected hardware devices by e.g. using their input and output register interfaces. If this approach is used, traditionally either the RTOS handles this or the programmer is responsible for making assembly language inserts to access the I/O registers. However, due to interfacing with hardware being a cornerstone in many real-time systems, a more modern approach has been adopted. This approach concerns letting the programming languages provide functionality for accessing the given registers belonging to the specific hardware.

Efficient Implementation In some systems it might be an efficient implementation that makes it possible to uphold the deadlines. If this is the case, the programmer must e.g. be aware of which data structures and language features to use. Furthermore, some embedded systems are build in very large quantities, and thus a more efficient software system with lower hardware requirements could be desirable. Even small reductions in production costs per unit may turn out to yield a significant overall saving.

Memory Management Many real-time systems rely on keeping the memory footprint as low as possible to reduce the amount of memory needed thereby effectively reducing the production costs. Due to real-time systems operating in a dynamic environment, it is sometimes necessary to dynamically allocate memory to accommodate the environmental changes. The heap used for dynamically allocated memory is, hence, required to be timing predictable. In some cases, this can be difficult because of the conflicting desire of keeping memory fragmentation low to reduce memory footprint. Mitigating fragmentation requires time to rearrange the heap to compensate for dynamic allocations and deallocations. The time used for mitigating fragmentation can be difficult to analyse. Also, an essential issue is if the programming language used relies on the presence of a garbage collector. The execution of the garbage collector can be difficult or even impossible to predict because it depends on when it assumes the system is idle or when subsequent memory allocations cannot be accommodated as a result of insufficient memory (Bacon et al., 2003; Knuth, 1997; Schoeberl, 2005). A more detailed description of memory both in terms of stack- and heap-based memory and garbage collectors is given in Appendix A.

Development Method The development method in real-time systems development can contain some distinct characteristics. The main difference between real-time systems and ordinary systems development, is that the real-time systems development process contains a tight coupling between hardware and software engineers (Graaf et al., 2003). Also, often a significant effort is put in requirements analysis to ensure that the system, before producing specialised hardware, correctly reflects the desired needs. To accommodate ambiguity in the requirements, they are often expressed formally e.g. as timed automata often requiring training to efficiently use. Appendix B contains a more detailed description of the development methods suitable for real-time systems.

Program Structure In respect to Java real-time systems, several sources refer to the usage of phases in the program's lifetime (Schoeberl et al., 2007; Bøgholm et al., 2009; Wellings, 2004; Søndergaard et al., 2006). These are collectively known as a *mission* and constitute:

- **Initialisation** The objects and real-time threads needed throughout the mission's lifetime are created and initialised in this phase. The phase is not time-critical, meaning that none of the real-time threads are started.
- **Execution** In this phase, the tasks run concurrently, hence the phase is time-critical.

- **Termination** When all tasks have completed, and optionally made a local clean-up, this phase is reached and the application can make a global clean-up. After the clean-up, the initialisation phase can be reached again.

An illustration of the phases and how they are interconnected is shown in Figure 2.3.

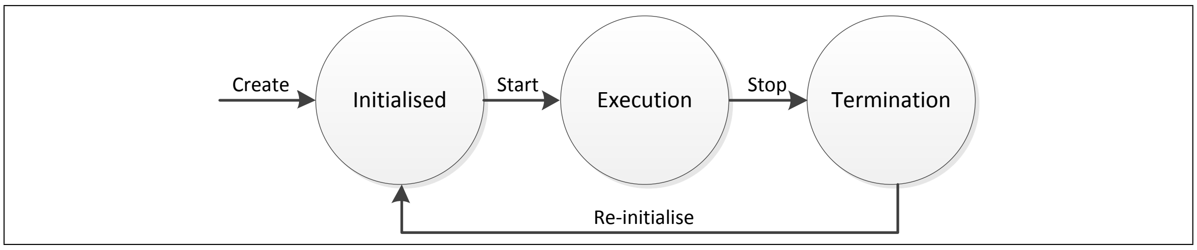


**Figure 2.3:** The phases involved in a real-time application's lifetime.

# 3 The Real-Time Embedded Structure

A real-time system consists of a structure of components such as hardware, RTOS, virtual machine, and programming language. The choice of structure is affected by e.g. RTOSs only supported on some hardware and programming languages requiring a virtual machine. In some cases the RTOS is skipped in favour of letting a running application handle the traditional RTOS facilities itself (Qing Li, 2003).

Figure 3.1 illustrates three different structures for real-time embedded systems. These structures include an application running directly on the hardware, running on top of an RTOS, and running on a virtual machine running on the RTOS. The choice of structure affects the resulting system, by increasing the complexity of timing predictability as the structure grows, while often providing additional functionality to the programmer. Thus, the choice of structure is a trade-off between facilities and complexity of timing predictability.(Schoeberl, 2003)

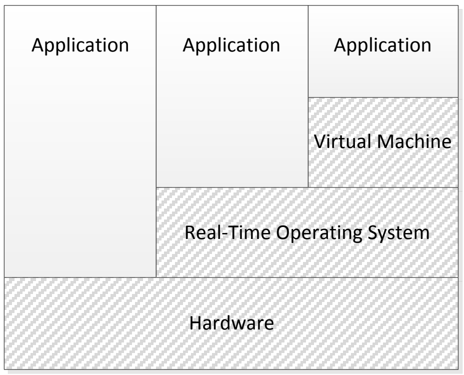


**Figure 3.1:** The three possible structure combinations with application, virtual machine, RTOS, and hardware.

The following describes the three layers: the hardware, RTOS, and programming language of which the application is written in. The virtual machine layer is not explicitly described since it is contained in the description of the programming languages depending on it.

## 3.1 Programming Languages

The analysis presented in this section is primarily based on Burns and Wellings (2009).

Many of the programming languages suitable for real-time embedded systems development do also apply as general-purpose programming languages. In the following, three different classes of programming languages suitable for real-time systems development are described.

### 3.1.1 Classes of Programming Languages

Using these language design criteria, a number of programming languages appropriately apply. Historically, three different classes of programming languages have been used which primarily reflect the present hardware and the evolution of programming paradigms and compiler technologies. We have chosen to classify programming languages into assembly languages, low-level programming languages, and high-level programming languages where *low-level* in this regard is relatively defined. That is, given programming languages such as Java featuring good support for object-oriented programming, C is regarded as low-level in this context. The language design criteria can be seen as being better fulfilled in some of the newer programming languages. For instance, assembly languages can be considered: hard to read and hard to transfer to other platforms without rewriting the entire program.

Assembly Languages Many real-time systems were initially programmed in assembly. This tendency reflects the fact that high-level languages were not supported on the micro-computers at that time to enable interacting with hardware. Assembly being a way of achieving efficient programs with a high level of control of underlying resources has the major drawback of being tightly bound to the particular hardware. Hence, assembly does not promote portability of programs which can turn out costly.

Low-Level Programming Languages Following the assembly language, compiler technology and programming language design became more mature. Some programming languages were specifically targeted at embedded systems programming languages such as Jovial. Also, in this regard the C programming language has been widely used. A problem with these languages, however, is that they generally lack facilities for real-time control and reliability.

High-Level Programming Languages The American Department of Defense identified that a substantial cost was associated with the use of 450 general-purpose languages for the development of embedded applications. Therefore, a single language was desirable to help reduce the cost of developing embedded applications. The result of the evaluation was the Ada programming language. Other languages have also been developed to suit the needs of embedded and real-time systems programming such as Modula-1. Finally, although initially not suitable for real-time programming, an extensive effort has been put in introducing real-time properties to the Java programming language.(Schoeberl, 2003; Bøgholm et al., 2009; RTSJ.org, 2010; Kwon et al., 2002; JSR302, 2010)

To better illustrate the differences between the programming language classes, we have made a simple assessment of their respective compliances with the design criteria listed in Appendix D based on our knowledge of the classes. Table 3.1 shows our assessment where we have given scores from one to three, where three is the highest.

| Criterion | Assembly | Low-Level | High-Level |
|---|---|---|---|
| Security | 1 | 2 | 3 |
| Readability | 1 | 2 | 3 |
| Flexibility | 1 | 3 | 3 |
| Simplicity | 1 | 2 | 3 |
| Portability | 1 | 2 | 3 |
| Efficiency | 3 | 2 | 1 |
| Total | 8 | 13 | 16 |

**Table 3.1:** Our assessment of assembly, low-level, and high-level programming languages in terms of their compliances with the design criteria listed in Appendix D.

As shown, the high-level languages score highest. This means that according to our assessment, high-level programming languages are best suited for real-time systems development. We think this is interesting since many real-time systems still are developed in programming languages such as C which is not part of this class and hence not as suitable according to our assessment. Also, due to an increased attention to the high-level languages drawn in the last 15 years, it may be beneficial to enhance them to include the necessary real-time properties (TIOBE-Software, 2010). This is mainly due to the tendency, that high-level languages gain more popularity because of increased focus in teaching at educational institutions. From this, it may be deduced, that newly educated students are more proficient in these languages. Furthermore, high-level languages evidently have higher level of abstraction than more traditional real-time systems development languages such as C, and therefore they may provide many benefits in real-time systems development such as potentially enhancing productivity, security, and portability.

Java and C# are some of the high-level programming languages that are popular today (TIOBE-Software, 2010). In the following, a description of these with a real-time perspective is given.

### 3.1.2 Java

Java, a language conceived in 1991, was originally intended for consumer electronics such as set-top boxes but was in 1994 applied in the Internet business due to the Internet receiving much attention at that time. Over time, Java has gradually become a general-purpose programming language. However, by default, it does not facilitate the necessary timing predictability and other properties being integral in the development of real-time systems

(RTSJ.org, 2010). E.g. the inclusion of a garbage collector is a problem that significantly influences the timing predictability of Java programs. Also, Java relying on a managed runtime environment in the form of the Java Virtual Machine (JVM), introduces issues because the compilation process of Java bytecode may be dependent on the particular execution pattern of the program. For example, if the JVM uses Just-In-Time (JIT) compilation, it is difficult to predict at runtime which optimisation strategies that may be applied to the program. Some JVMs even incorporate a hybrid approach using JIT compilation and interpretation which further complicates timing predictability.(Schoeberl, 2009)

Currently, a great effort is put in introducing real-time properties in Java for making it suitable for real-time systems development. Among others, the RTSJ has been introduced to accommodate this need. However, in respect to real-time embedded systems, the RTSJ is by some considered too broad, that is, it provides a set of unnecessary functionality in this context. Therefore, research has revolved around defining a real-time Java profile with the purpose of sub-setting the RTSJ. Ravenscar-Java (RJ), Safety Critical Java (SCJ), and Predictable Java (PJ) are all examples of such profiles.(Bøgholm et al., 2009; Kwon et al., 2002; JSR302, 2010)

### 3.1.3 C#

The .NET Platform, for which C# has been written, was introduced in 2000. The primary objective of the platform is to provide a development framework more capable of leveraging technologies than previous Microsoft platforms and more capable than other platforms such as Java with the JVM (Zerzelidis et al., 2004; Griffiths, 2010). C# is the object-oriented language introduced as part of .NET which has gained much attention since its release (TIOBE-Software, 2010).

C#, and more specifically its underlying .NET platform, was not designed for the purpose of being used in real-time systems development. For instance, .NET does not provide real-time facilities such as real-time scheduling, and the notions of periodic, aperiodic, and sporadic tasks. Furthermore, as with Java, C# also relies on automatic memory deallocation by introducing a garbage collector with unpredictable timings. The Common Language Runtime (CLR) of the .NET Platform also uses JIT compilation which further complicates the process of determining timings.(Zerzelidis et al., 2004)

It seems that, currently, no significant effort has been put in introducing real-time facilities to the .NET Platform. However, according to Zerzelidis et al. (2004), initial requirements have been set for extending the .NET Platform with real-time capabilities. These requirements take starting point in the requirements that were initially set for Java when it was introduced in a real-time context. The rationale behind drawing inspiration from Java is that .NET and Java have many similarities such as relying on a virtual machine, automatic memory management, and supporting dynamic class loading.

## 3.2 Real-Time Operating Systems

RTOSs are specialised Operating Systems (OSs) providing the necessary facilities for systems having timing requirements. In order to facilitate real-time applications, a number of characteristics can be specified which differentiate RTOSs from General Purpose Operating Systems (GPOSs). The following lists a number of characteristics for RTOSs that normally are not relevant to the same extent for GPOSs.(Qing Li, 2003)

Scaling Based on Application Needs The functionality of some RTOSs can be increased or decreased according to the needs for the applications. This is an advantage since reducing the functionality often simplifies the system, reducing system's program and memory footprint. Furthermore, it must be assumed that the lesser functionality potentially gives better timing predictability.

Note that not all RTOSs are scalable as described by Stankovic and Rajkumar (2004). Having special purpose RTOSs can be advantageous in some cases, e.g. if a larger company needs an RTOS for many embedded devices it might be easier for them simply to develop their own RTOS instead of relying on other implementations.

Reduced Memory Requirements For real-time systems, especially those which are manufactured in large quantities, it is desirable to minimise the resource usage, since this results in lower production costs. Therefore, RTOSs are typically very small compared to GPOSs. Notice, that tightening the memory budget too much might create the opposite effect of increasing costs due to development time and effort needed to modify and maintain the system.

Timing Predictability A real-time system often requires predictable behaviour in regard to time in order to react to events within its given deadlines. This is central for real-time systems, and thus RTOSs need to support this. As an example, the scheduling policy needs to be predictable in order to ensure that the deadlines of different tasks are not exceeded. Furthermore, system facilities such as locking mechanisms for shared memory and I/O access need to be predictable as well.

It is important to notice that this list of characteristics is not exhaustive, and not all systems require all of them. It all depends on which context the RTOS is to be used in. As an example, the two RTOSs VxWorks (River, 2010) and Xenomai (Xenomai, 2010) Linux are quite different. VxWorks is targeted at small embedded systems using few resources, and is deployed on systems such as the Mars Phoenix Lander (River, 2007). Xenomai Linux, on the other hand, is a full Linux OS with a real-time enabled co-kernel handling all interrupt and scheduling responsibilities. The Linux kernel is run as a low priority process. One could imagine that the Xenomai Linux is more complex to analyse in respect to timing predictability because of its execution of an independent Linux kernel.

## 3.3 Hardware

Real-time systems are typically based on special purpose hardware, which in relation to desktop computers have very limited resources and processing capabilities. Also, an essential characteristic of the hardware platform is that it must provide timing predictability. In the following, three interesting hardware platforms are presented. Specifically, these are: the family of ARM architectures and two platforms, Java Optimized Processor (JOP) and aJ-100, that implement a JVM in hardware.

### 3.3.1 ARM

The ARM processor is not a single processor core but instead a wide family of designs sharing similar design principles and a common instruction set (Andrew N. Sloss, 2004). In embedded systems, the ARM architecture is widely adopted due to its relative simplicity, which is a convenient trait in relation to low-powered applications. ARM processors are commonly found in handheld devices and in a multitude of other portable devices (Andrew N. Sloss, 2004). Also, the ARM architecture is supported by the majority of RTOS vendors (ARM, 2010).

A key design principle of the ARM architecture is not primarily to provide raw processor speed but instead a combination of system performance and low power consumption. The specifications of the ARM cores provide information about clock cycles needed for each instruction which is imperative for timing predictability. However, as stated in ARM (2006), the structure of ARM processors makes it virtually impossible to guarantee timing predictability. However, the research project by Dalsgaard et al. (2009) has succeeded in estimating the timings by using modelling techniques of the mechanisms that influence timing predictability on ARM. Some of these mechanisms are described in the following and are based on Andrew N. Sloss (2004).

Cache To mitigate the relatively slow memory performance of main memory, some of the ARM processor cores incorporate a hierarchy of memories. This is implemented in the form of caches of different size and of different locality to the core. However, the introduction of memory hierarchies complicates timing predictability of the executing application due to the memory access speeds depending on whether various caches or main memory is used. More specifically, the problem is, that during application execution, the cache quickly fills up which initiates the cache controller to frequently evict code or data from the cache to allow new ones to be cached. This eviction process is unsuitable for WCET analysis.

Pipeline Many of the ARM processors also incorporate a pipeline design which is a mechanism that essentially allows fetching next instructions while other instructions are being decoded and executed. This is an example of a three-staged pipeline. The incorporation of a pipeline is to increase performance in the form of increasing instruction throughput. As with the implementation of a memory hierarchy, pipelines may also complicate the determination of execution times. This is primarily due to the fact that some of the stages have interdependencies which may consequently lead to stalls and other situations that interrupt the sequential process of filling the pipeline. As an example, an instruction may require a result obtained from the previous instruction. However, this instruction may be delayed due to memory access.

### 3.3.2 aJ-100

aJ-100 is a real-time embedded single-chip Java micro-controller (Hardin, 2010). It builds on the JEM2 core, which is aJile Systems' hardware implementation of the JVM, implementing an architecture similar to RISC (aJile Systems, 2000). The micro-controller has a clock rate up to 100 MHz and due to its low-power design, the power consumption is 1 mW/MHz. Furthermore, through benchmarking, it has been shown that by implementing the JVM in silicon neutralises the myth that Java generally has lower performance compared to other languages such as C. Hardin (2010) has described that the performance of executing Java on the JEM2 core is comparable to executing C on a comparable architecture.(Hardin, 2010; aJile Systems, 2000)

The aJ-100 supports the entire instruction set of the JVM and thread model. Therefore, an interpreter or a JIT compiler as well as an actual RTOS are not needed. Furthermore, it is described that the time needed for executing *yield()* and resuming executing another thread is approximately 500 nano-seconds. Even though 500 nano-seconds is considered fast in many time budgets it is, as will be described later, not sufficient with approximate values in a hard real-time setting (Hardin, 2010). Additionally, aJ-100 implements a garbage collector, which is executed in an independent thread, and can be preempted in one nano-second.

The aJ-100 seems in many aspects suitable for real-time embedded systems. However it suffers from a major drawback. Apparently, it is not possible to find any timing information of the processor (Schoeberl and Pedersen, 2006). This means that it is not possible to predict the timings of its microcode, hence excluding its usage in hard real-time systems.

### 3.3.3 JOP

An interesting processor in regard to real-time systems written in Java, is the JOP. JOP is a RISC architecture implementing the JVM directly in hardware, enabling it to execute Java bytecode natively. One of its goals is to allow Java programs to be executed on the processor in a timing predictable manner.(JOP, 2008)

One of the newest versions of the JOP is build on the Altera Cyclone (EP1C12) board, running with a clock frequency of 100MHz, and has 512KB of flash memory and 1MB of SRAM. The board is based on a Field-Programmable Gate Array (FPGA) which essentially is a configurable integrated circuit. The configuration of the JOP is specified using VHSIC Hardware Description Language (VHDL).(Schoeberl, 2009)

In order to make the JOP timing predictable, timing analyses have been conducted on the microcode that implements the Java bytecode instructions. These comprise not only basic bytecode instructions and memory accesses, but also complex bytecodes such as floating point instructions implemented in software and native methods. To allow native methods, these are implemented by means of special byte code instructions which again are implemented on the JOP in microcode.

Even though the JOP's main goal is to be timing predictable, it still needs to have an acceptable performance. Thus it implements pipelining and two different caches: a *method cache* and a *stack cache*. How these are implemented such that they are timing predictable is described in the following.(Schoeberl and Pedersen, 2006)

Pipeline The JOP implements a four-stage pipeline with the stages *bytecode fetch*, *microcode fetch*, *decode*, and *execute*. The first stage fetches the bytecode which is translated into one or more microcode instructions which then are fetched and executed. This rather simple pipeline is by Schoeberl (2005) described as being timing predictable.

Method Cache The method cache contains the instructions for the currently executing method. This cache is populated when the method is invoked and the population time is predictable since the needed memory amount can be statically determined and the transfer times are known. In contrast to the cache mechanism implemented in the ARM processors, this relatively simple caching mechanism is timing predictable and it means that the occurrence of cache misses is limited to invoke and return.(Schoeberl, 2005)

Even though the approach is timing predictable, it is not flawless. It is e.g. a problem if methods are larger than the available method cache. In this case they must be split into smaller methods. Another issue is that whole methods are loaded into cache, regardless of how they are used. Finally, it is required that methods loaded into the cache are loaded into consecutive cache blocks. To comply with this constraint, the JOP implements a FIFO-based replacement strategy. However, the behaviour of this strategy is known to be difficult to analyse.(Schoeberl and Pedersen, 2006)

Stack Cache The stack cache contains the current stack, or parts of it, for the running task. This speeds up memory access especially since the stack is heavily accessed. In order to make the cache timing predictable, it may only be exchanged with the main memory when a method is invoked, returns or is context switched. It is timing predictable since the stack height is known from static analysis, that is, the exact time needed to load or store the stack at invokes or returns is known. Furthermore, this time is used to predict the time needed for context switches where threads' local stacks must be loaded into the cache.(Schoeberl, 2005)

The scheduler implemented in JOP is a preemptive, priority-based scheduler (Schoeberl, 2009). The scheduler contains an unlimited amount of priority levels and a unique priority level is assigned for each schedulable object. The reason for assigning unique priority values is based on the fact that it eliminates the need for FIFO queues for each priority level, hence it is a simpler approach. Two tasks assigned the same priority, will be resolved by assigning a higher priority to an arbitrary chosen task.(Schoeberl, 2009)

The JOP implements a real-time garbage collector (Schoeberl, 2009). This is made possible by an interplay between hardware specific components and software. Generally, the garbage collector is implemented as a real-time thread running concurrently with the other processes. To ensure that it does not affect the other threads in a negative manner, it is given the lowest priority and furthermore its blocking time has been reduced to a maximum of 54 micro-seconds. Even though Schoeberl (2009) describes how WCET for the garbage collector can be made, it is also stated that the garbage collector requires more analysis before it should be used (Schoeberl, 2009). However, it is not stated how the garbage collector can be disabled. We have examined the implementation files of the garbage collector and conclude that it can be disabled by changing a constant in the `GC` class. The examination of the source files revealed that if the garbage collector is disabled, then it terminates the program if the JVM determines that depleted memory must be removed by using the garbage collector.

# 4
# Worst Case Execution Time Analysis

Hard real-time systems have zero tolerance on exceeding deadlines. Therefore, it is necessary to know that all deadlines can be upheld throughout the application's lifetime. To ensure the reliability of the application, the execution time of each task must be known. With this knowledge combined with whether the tasks are periodic, aperiodic, or sporadic, it is possible to conduct a schedulability analysis to unveil, whether it is possible to schedule them such that no deadlines are missed. To ensure that the result of the schedulability analysis is always valid, the execution times for each task in isolation must be worst-case. This is known as WCET.(Burns and Wellings, 2009)

The following describes the terminology used in relation to WCET. Afterwards, the two different overall approaches to WCET analysis are explored, namely: *static methods*, and *measurement-based methods*. During the exploration, a discussion of some of the issues and limitations of WCET analysis is provided.

## 4.1 Terminology

Initially, methods and tools used for WCET analysis are evaluated according to two different criteria being:(Wilhelm et al., 2008)

Safety This criterion is used to describe whether the method in question produces bounds or estimates. Bounds are usually said to provide safety because these guarantee that the WCET of the task is not exceeded. Estimates, on the other hand, are said to provide unsafe WCET due to the possibility of various omitted factors in the estimate consequently resulting in a higher WCET.

Precision This criterion is used to evaluate the degree to which the derived WCET deviates from the actual WCET of the task. Establishing an upper-bound which is safe may in some applications not be the only desirable characteristic because it can result in over-estimating the necessary hardware.

## 4.2 Static Methods

The static WCET methods do not require the program to be executed on real hardware or in simulation. Instead, the methods analyse abstract representations of the code and possibly combine them with an abstract representation of the system onto which it is supposed to execute to determine its WCET. It is important to note that the abstractions must be made such that the methods provide safe WCETs. If the program must be executed on different processors, new abstract representations must be made to estimate the WCET of the new hardware. Furthermore, since the representations of the processors are abstract, they are typically not representing the exact processor behaviour. Therefore, the WCET estimated by static methods is not always precise, potentially making over-estimates.(Wilhelm et al., 2008)

### 4.2.1 Preconditions

The prerequisite of statically determining the WCET of a task is that all its constituents, that is sequences, loops etc., need to be deterministic in regard to time. Hence, all information about the control flow and in addition the constraints of the control flow need to be known statically. This poses an essential problem in static timing analysis because the control flow of the program at run-time might depend on its input data, among others. This means that the WCET can be time consuming to determine if many different inputs need to be explored, resulting in an exponential number of execution paths. Furthermore, specific language constructs such as unbounded loops, can result in programs for which WCET cannot be derived.(Puschner and Koza, 1989)

In the following, a number of restrictions is presented to remove problems in some languages with regard to conducting WCET. The problems are based on (Puschner and Koza, 1989) with the exception of polymorphism which has been added by us.

**Unbounded Loops** The number of iterations of unbounded loops is either impossible or requires great effort to extract. However, due to being a fundamental part of many algorithms, eliminating these makes programming nearly impossible. Instead, the approach taken is to either limit the iterations or to specify a time limit for which the loop is allowed to iterate by means of programmer annotations. Loops with these restrictions are called bounded loops. Note that due to the manual nature of annotating the code, a premise is that these are stated correctly to get safe WCETs.

**Recursion** Introducing recursion in the program poses similar problems as in introducing unbounded loops. In this case, the maximal depth of the recursive calls is not statically determinable. Besides, recursive functions also introduce another problem, namely that the stack space demand cannot be determined statically. This means that the general approach in real-time systems of statically allocating the stack space required at run-time cannot be done.

Pointers to Functions These language constructs are able to reference different functions consequently leading to difficulties determining the WCET of a specific function call if the called function cannot be statically determined. E.g. if a function uses another function as a parameter, and it cannot be determined statically which function is given, then the WCET of the function pointer must be regarded as the largest WCET of all possible functions in the system, potentially resulting in over-estimation. Furthermore, function pointers can be used to implement recursion. The solution is simply to eliminate the use of function pointers and replace them by explicit function calls equivalents.

Polymorphism Using polymorphism can also be a problem for WCET analysis. The reason is that it cannot necessarily be inferred which method is actually called when calling a virtual method of a superclass. If possible, it can be inferred using Data-Flow Analysis (DFA) or alternatively the method whose implementation results in the greatest WCET can be used. The latter approach will, however, result in very pessimistic estimates. This is similar to problems encountered with function pointers and a possible solution is simply to omit polymorphism from the programming language.

### 4.2.2 Control Flow Graph

A fundamental technique used in program analysis is Control-Flow Analysis (CFA) and the Control-Flow Graph (CFG) produced by it. A CFG can be used as foundation for different analyses of programs, such as for compiler optimisation, e.g. by determining infeasible paths, constant propagation, and redundant assignment detection.(Allen, 1970)

Since CFGs are used in some of the subsequent WCET analysis techniques, the concepts and terminology needed to understand CFGs are introduced in the following.

A CFG consists of a number of basic block effectively representing the program or parts thereof. A basic block is defined as follows:

Basic Block A basic block is a block enclosing a number of program statements that do not contain any jumps or jump targets. The only jump target is when the basic block starts and the only jump is at the end of the basic block.

A CFG is a diagrammatic representation of all the paths that are traversable during execution, that is, it captures the control flow of the program. The representation is based on a directed graph and can be seen as an abstract representation of the control flow. Each node in the graph constitutes a basic block. All nodes in the CFG are interconnected via a set of edges representing jumps in the control flow. An execution path is defined as a sequence of interconnected basic blocks.(Lokuciejewski and Marwedel, 2011)

In Figure 4.1, an example of a CFG is shown.

By annotating the nodes in the graph, that is the basic blocks, with their execution costs, it should be clear that a WCET of the program represented by a CFG can be found by e.g. traversing the CFG to find the most costly path.

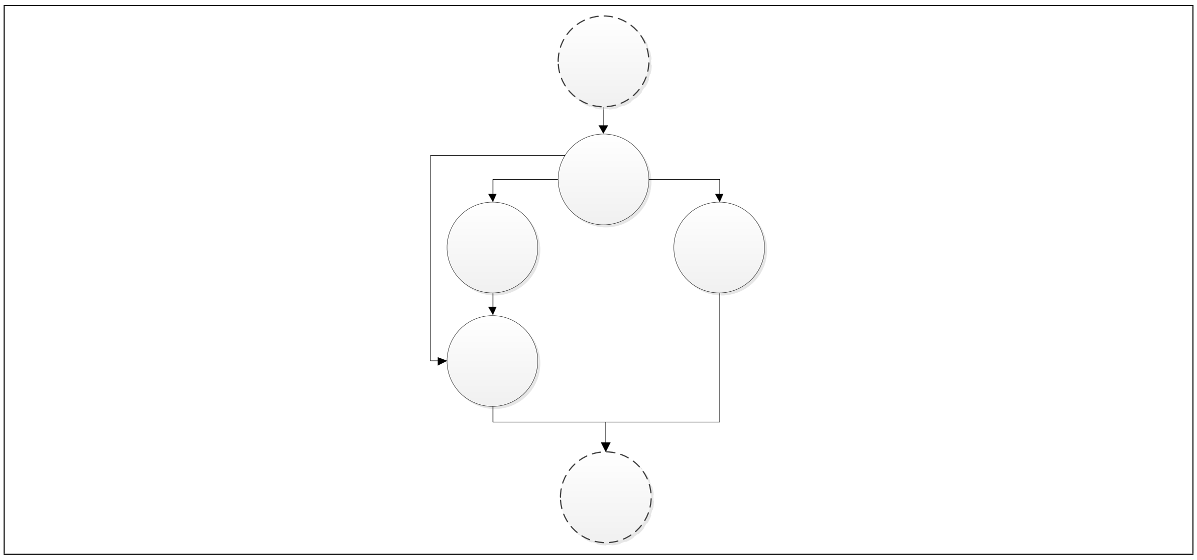


**Figure 4.1:** Example of a CFG. The representation illustrates both a loop structure and a conditional statement present in the program. The dashed basic are used to illustrate that the example is an excerpt of a CFG.

### 4.2.3 Data-Flow Analysis

DFA is the analysis technique that exercises a constructed CFG to collect information about the flow of data along the program execution paths. It is used in a variety of areas such as program optimisation (Blieberger, 2002). For instance, DFA can be used for analysing whether a variable assignment is ever used subsequently in the execution path or conduct dead code elimination.(Alfred V. Aho and Ullman, 2006)

Because WCET analysis often depends on programmer annotations of the code such as specifying loop bounds or other constraints for the control flow, it inevitably leads to inconsistencies due to the manual nature of this approach. Here, DFA can be supportive by examining the execution frequencies of paths or the relation between execution frequencies of different paths. For instance, DFA may try to establish the iteration count of unbounded loops. This may require a significant effort to conduct and, hence, programmer annotations or bounded loops may be preferred.(Wilhelm et al., 2008; Lokuciejewski and Marwedel, 2011)

### 4.2.4 Processor Behavioural Analysis

Modern processors introduce a number of facilities which improve the average execution time of instructions. Examples of such facilities are caches, pipelines, and branch prediction. This often leads to over-estimation of the execution time, since WCET analysis must take the worst-case into account even if the normal case is far more likely to occur. To reduce this over-estimation of the execution time, *processor behavioural analysis* can be used. The principle of this method is to model the behaviour of the underlying hardware facilities in order to create more precise WCETs (Lokuciejewski and Marwedel, 2011). The execution time of individual instructions, or even whole basic blocks, depend on the current state of the system which in turn is a product of the previously executed instructions(Wilhelm et al., 2008).

As an example of processor behavioural analysis, a model has been constructed for an ARM processor which allows for analysing its instructions and their execution times, taking the pipeline and caches into account (Dalsgaard et al., 2009). Also, a similar approach is taken by Sarkar et al. (2009), to model the x86-CC processor.

The modelling of these facilities can be difficult since most vendors do not specify in detail the behaviour of a given processor. Since these details are unknown, it is not possible to verify if the model correctly reflects the processor. Hence, to increase confidence in the correctness of the model, results obtained from the processor and the model can be compared. Furthermore, the presence of unknown input or unknown states in the model can result in state space explosion, since each possible state must be explored (Wilhelm et al., 2008). The reason for the necessity of exploring all states is due to the phenomenon called *timing anomalies*. Timing anomalies are defined by the fact that changes in local execution time of a single instruction can influence the global execution time in a counter-intuitive way. This means, that even though the execution time of a subset of a task is decreased, the execution time of the whole is not necessarily decreased. In fact, it can yield an increase. An example of a cause to a timing anomaly is given in Example 2.(Lundqvist and Stenstrom, 2002)

**Example 2 (Timing Anomaly)**
At a specific point in a program, a memory access can result in two outcomes: a cache miss or a cache hit. Intuitively, the cache hit should result in a shorter execution time than a cache miss, but this is not always the case because of timing anomalies.

If the processor supports out-of-order execution, then a cache hit or cache miss can result in a change in the order of scheduled instructions. This change in the execution order can affect the total running time by both increasing it and decreasing it.

In order to further improve the processor behavioural analysis, different methods can be used to determine the actual state of the processor at a given time. This will limit the state space explosion by reducing the number of possible inputs or states that must be exercised. One such method is value analysis, which can predict the values of the processor's registers at any time and execution context. Typically, the values of the registers cannot be exactly determined, and it is therefore necessary to give approximations for the values. To do this, safe lower- and upper-bounds can be found, hence guaranteeing the exact inclusion of the values obtained at runtime.(Heckmann and Ferdinand, 2009)

### 4.2.5 Calculating Worst Case Execution Time

The presented approaches for statically determining WCET utilise the concepts introduced previously about static methods. Specifically, *path-based calculation*, *Implicit Path Enumeration Technique (IPET)*, and *model-based determination* are introduced.

#### Path-Based Calculation

Path-Based Calculation is an approach to determine the WCET of a task represented as a CFG with annotated WCET for individual nodes. The CFG is examined to determine its longest path representing the upper-bound of the execution time of the task. This can be solved by transforming the CFG into a weighted graph, with weights representing the WCET of basic blocks, and afterwards using standard graph algorithms (Wilhelm et al., 2008). Engblom et al. (2000) describes the following procedure when determining the WCET:

1. Find the longest path in the graph using a standard graph algorithm.
2. Check if the path is feasible.
3. If the path is not feasible, exclude it from the graph and return to the first step.
4. When the longest feasible path is found, its length is the WCET of the task.

Step two to four are not required but increase the precision of the WCET. An example of path-based determination is provided in Example 3.

**Example 3 (Path-Based WCET Calculation)**
Figure 4.2 depicts an example containing a loop in which the longest path is identified.

As shown, the CFG consists of five basic blocks: $B_0, \dots, B_4$ with costs $C_0, \dots, C_4$ respectively. Also, notice that $B_1, B_2, B_3$, and $B_4$ constitute a loop body. The marked path indicates the longest path in the loop, giving WCET, of one loop iteration of $(3 + 4 + 7 + 2) = 16$ time units. Multiplying this with the loop bound gives the WCET for the entire loop.

The advantage of the method is that it is relatively simple. However, the complexity of the approach is exponential with the depth of conditional statements. Therefore, for larger systems with a structure containing many of these, the approach is not considered appropriate.(Huber and Puschner, 2010)

#### Implicit Path Enumeration Technique (IPET)

The general concept of IPET is to determine the worst and best case running times, but in contrast to the path-based calculation, IPET does not concern identifying the specific path

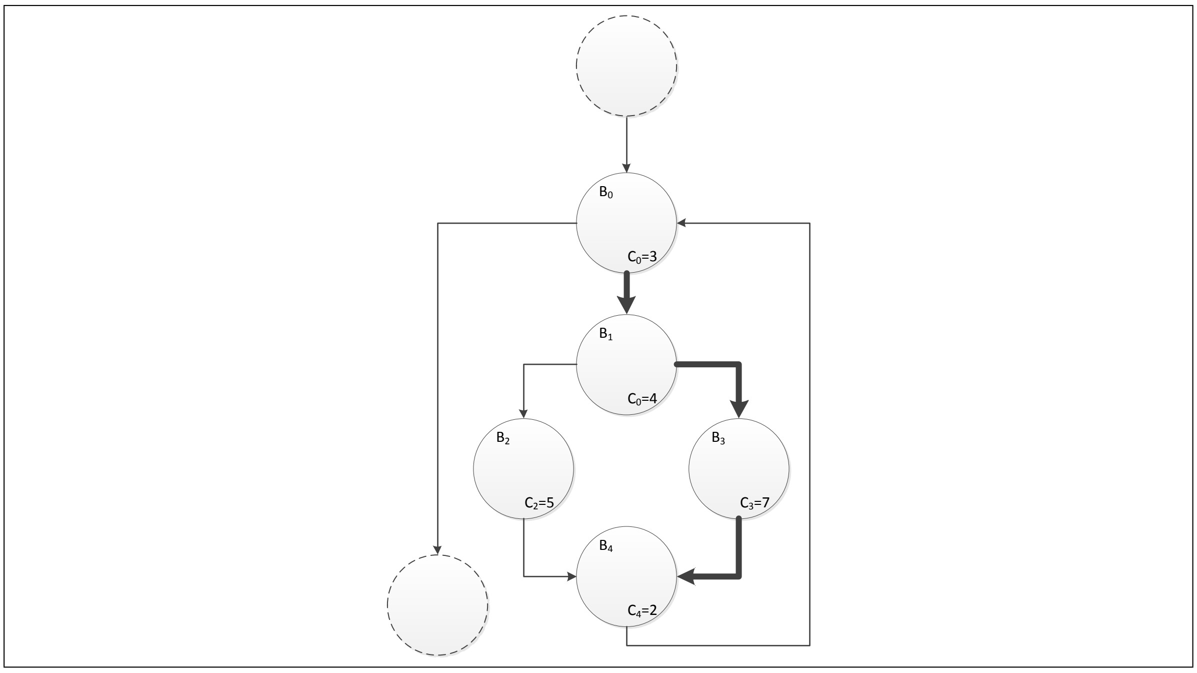


**Figure 4.2:** CFG illustrating how path-based calculations are performed. The marked path is the longest of the loop, hence the executions times of each basic block in it must be summed to determine the WCET for one loop iteration.

yielding either the best or worst-case execution time. WCET is estimated by summarising the number of times each basic block is executed, multiplied with its cost. This means that the method is dependent on knowing the cost of each basic block beforehand. The advantage of this approach is that the problem of estimating WCET then can be transformed into the following Integer Linear Programming (ILP) problem:(Li and Malik, 1995; Schoeberl et al., 2010)

$$\sum_{i=1}^{N} c_i f_i, \tag{4.1}$$

where $c_i$ is the cost of the $i$th basic block, $N$ denotes the total number of basic blocks, and $f_i$ is its execution frequency. Either by minimising or maximising the problem, the best and worst-case execution times can be determined, respectively. Since WCET concerns maximising the problem, only this will be considered in the following.(Li and Malik, 1995)

### Constraints

Solving an ILP problem requires two different types of constraints to be taken into account:

**Structural Constraints** These constraints are concerned with the structure, such as branches in the control flow, of the program. This information can be extracted from a CFG.

**Functional Constraints** These constraints are concerned with the behaviour of the control flow of the running program, such as knowing the bound of a loop. Some methods of extracting this information are through annotations or using DFA.

Even though ILP problems are NP-complete, Li and Malik (1995) were able to show that combining structural constraints with functionality constraints, obtained from the constructs in their information description language, allowed for collapsing the ILP problem to a problem which can be solved in polynomial time.

The ILP problem is an example of the necessity of bounding the loops in a program. Otherwise, the maximum value of Equation 4.1 is $\infty$ since the number of times to execute a basic block, $f_i$, can be infinite. For this reason, the problem is in general said to be undecidable, and a set of constraints is needed to make the problem decidable. In the following, these constraints and their interrelationships are described.(Li and Malik, 1995)

Listing 4.1 is used as a basis for the examples.

```
while(q<10){
    if(flag) {
        q++;
        flag = false;
    } else {
        flag = true;
    }
    q++;
}
r=q;
```

**Listing 4.1:** Code example illustrating how WCET depends on the program structure.

As shown, the code consists of a loop containing a condition in its body. It should be clear that the *if* and *else* statements are mutually exclusive in each iteration. This observation is an instance of a structural constraint. The WCET of the code is therefore dependent on the branch, and cannot be determined as e.g. the sum of the WCET of both branches. This is incorporated into the ILP problem.(Li and Malik, 1995)

The program structure can be extracted from the CFG of the program. Figure 4.3 depicts the CFG of the code from Listing 4.1. The number of times the edges are followed is denoted $d_0, \ldots, d_8$ and the execution frequency of a basic block is denoted $f_0, \ldots, f_8$.

The general constraint which can be derived from the CFG is that each basic block must have an execution frequency equal to the sum of incoming edges, which again must be equal to the sum of outgoing edges. This means that the following must hold:(Li and Malik, 1995; Schoeberl et al., 2010)

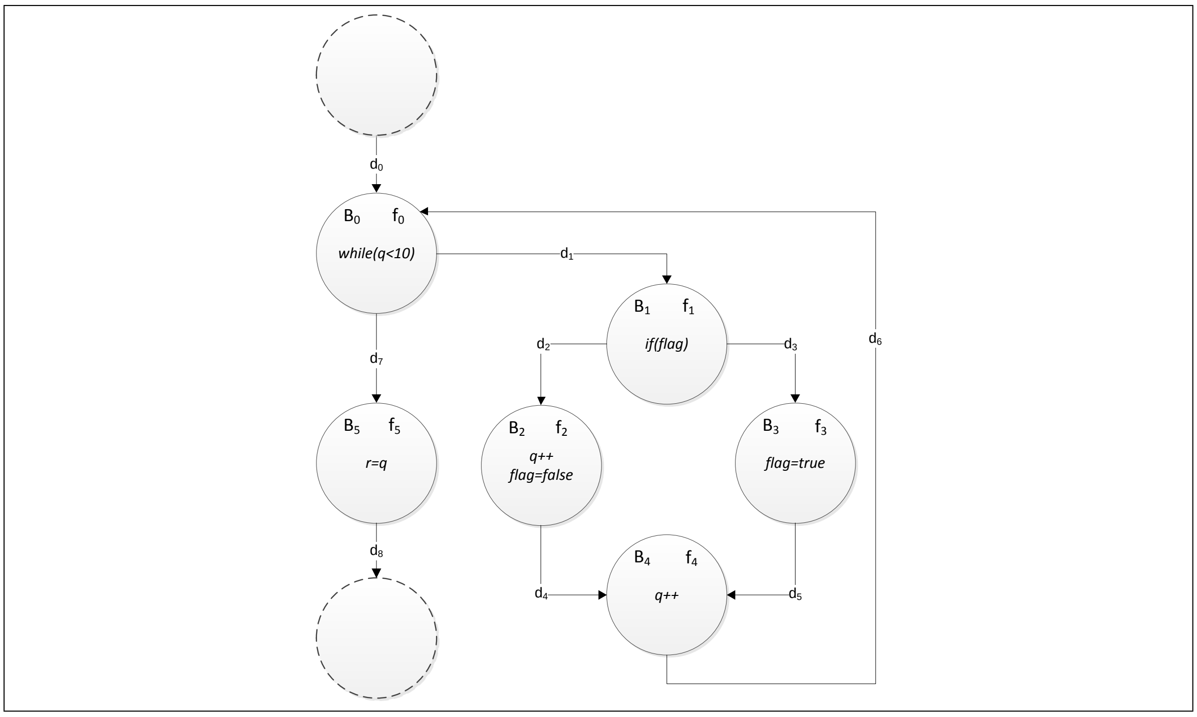


**Figure 4.3:** CFG of the code example shown in Listing 4.1.

$$f_i = \sum_{j \in I_i} d_j = \sum_{k \in O_i} d_k,$$

where $d$ denotes the execution frequency of the incoming and outgoing edges $j$ and $k$, and $I_i$ and $O_i$ denote two sets for the incoming and outgoing edges of a basic block, $B_i$, respectively. Returning to the example, consider only the condition inside the body of the loop, namely basic block $B_1, \ldots, B_4$. $f_1, \ldots, f_4$ denote the execution frequency of the respective basic blocks. Mapping the above constraint to this subset of the CFG the following constraints exist:

$$\begin{aligned} f_1 &= d_1 = d_2 + d_3 \\ f_2 &= d_2 = d_4 \\ f_3 &= d_3 = d_5 \\ f_4 &= d_4 + d_5 = d_6 \end{aligned}$$

A full example of applying IPET on a simple Java program using program structure constraints is given in Appendix E.

Similar constraints must be introduced to handle loops. In each loop, the number of back edges, that is edges returning to the head of the loop from the body of the loop, must be less than or equal to the number of incoming edges to the loop head multiplied with the loop bound. The constraint is shown in the following:(Schoeberl et al., 2010)

$$\sum_{j \in C_h} d_j \leq n \sum_{k \in E_h} d_k,$$

where $d$ is the execution frequency of the incoming and outgoing edges, $j$ and $k$, $C_h$ is the back edges that constitute the loop, and $E_h$ is the set of incoming edges entering the loop head. Finally, $n$ is the loop bound.

In respect to the example, this means that the following constraints must hold for the loop:

$$d_6 \leq n \cdot d_0.$$

Notice that $n$ is unknown in the running example.

### Model-Based WCET Determination

It is possible to use model-checking to determine WCET by creating a model of the desired program. The model can be build using either individual instructions or entire basic blocks as vertices and edges to define the control flow. This means that the model can be derived from the CFG, since it contains the needed information.

If compared to IPET, model-based analysis is generally more time consuming, since it requires exercising the entire state space. Regardless, if model-based analysis is used, it is easier to incorporate exact models of the underlying platform in order to create accurate estimates. It is therefore known that more precise WCET bounds can be achieved with models, than e.g. with IPET.(Schoeberl et al., 2010; Metzner, 2004)

Figure 4.4 depicts an example of how a loop can be modelled.

As shown, the loop is modelled using a traditional loop structure. Alternatively, it could have been unrolled. However, this approach is not recommended by Schoeberl et al. (2010) and Metzner (2004). Schoeberl et al. (2010) argue that in their implementation, it is more efficient to use the actual loop since the used model-checker, UPPAAL, can operate more efficiently on this. Before entering the *Loop Header*, an iterator is set to zero. On each loop, this iterator is incremented as long as it is less than *maxBound*. When the iterator becomes greater than or equal to *minBound* the *Next Operation* location can be reached.

As an example, the model-checker UPPAAL can be used to estimate the WCET of a given program. By exploring the state space, following all reachable locations, it is possible to determine path yielding the WCET of the program.(Metzner, 2004)

Schoeberl et al. (2010) and Metzner (2004) use the following approach for model-based WCET determination:

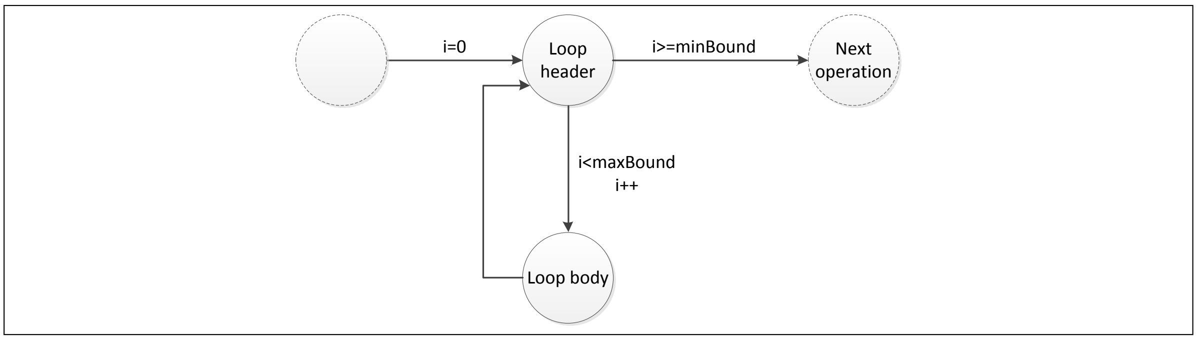


**Figure 4.4:** Modelling a loop with loop bounds. This model is similar to a UPPAAL model, whose annotations are described further in Section 10.2.

1. Model the cache.
2. Transfer the CFG of the application to a model in timed automata.
3. Find a safe WCET upper-bound. This can e.g. be done by using IPET.
4. Use a binary search to iteratively decrease the safe WCET to find a more precise and safe WCET.

Modelling the cache requires the specification of it on the chosen platform to be available. Figure 4.5 depicts how Schoeberl et al. (2010) has modelled the cache of the JOP.

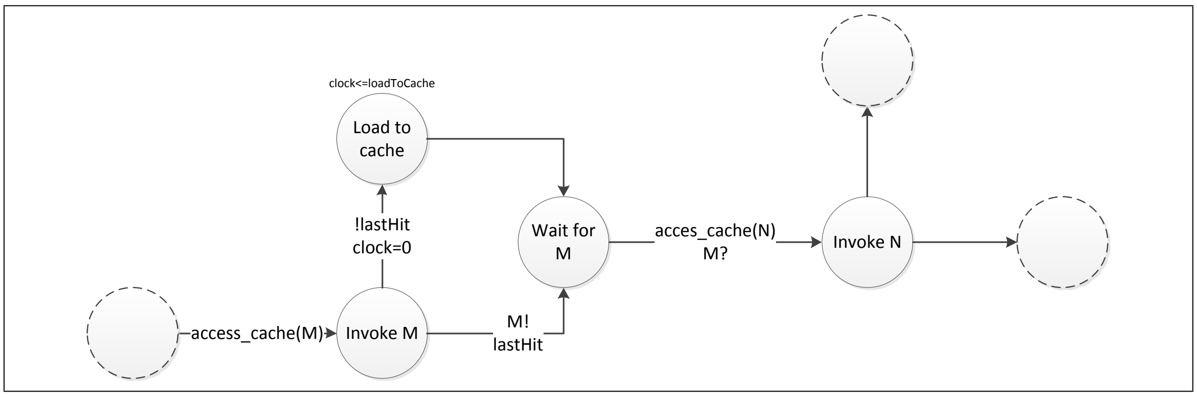


**Figure 4.5:** Modelling of the method cache used on JOP. This model is similar to a UPPAAL model, which annotations are described further in Section 10.2.

The function *acces_cache()* is called when invoking a method. This checks whether the method is already present in the cache and the *lastHit* variable is set accordingly. If it was in cache, the method is invoked by synchronising with its respective process. Otherwise, the *Load to cache* location is reached where the time needed to load the method into cache can be modelled by an invariant. When the process of M has finished executing, it synchronises with the invoking process and a new method invocation can be made.

To reduce the overall analysis time, Schoeberl et al. (2010) suggests combining IPET and model-based analysis such that model-based analysis is only used to important parts of the application. In our opinion, even long analysis times as a consequence of using model-based analysis may in many cases be the appropriate choice. Consider the situation where more precise WCET bounds are the result of model-based analysis which makes available other and cheaper platforms. In such case, conducting model-based analysis for determining WCET may be worth the efforts. Thus, there exists a trade-off between analysis time and precise bounds

A complete example of a WCET analysis using the above described technique is provided in Appendix F.

## 4.3 Measurement-Based Methods

An alternative to static methods of calculating the WCET is measurement-based methods. The essence of these is to measure the practical execution time of a program, or part of it, given a set of inputs. These methods have traditionally been used for determining WCET in the industry.(Sehlberg et al., 2006)

### 4.3.1 Measurements

The measurements can be conducted on either the actual hardware or in a simulator. For some categories of software, such as software for the avionic industry, a requirement is that the tested and production software must be identical. This can prove problematic if instrumentation of the source code, such as writing to standard output, has been used for measuring execution times (Lokuciejewski and Marwedel, 2011). Furthermore, because of the complexity of modern hardware, it can prove difficult to create a simulator which is timing predictable (Wilhelm et al., 2008).

In order to measure the WCET, the worst possible set of inputs must be identified which not surprisingly is difficult. Alternatively, one can measure the WCET using all possible inputs which similarly can yield difficult or even impossible. In practice, only a subset of inputs are tested and a safety margin is added in an attempt to avoid the WCET being under-estimated. However, if only a subset of inputs are used, the method is not safe, since an untested input could yield a longer execution time. If it is possible to fully test all possible inputs, or determine the worst-case input, then the resulting WCET is very precise.(Lokuciejewski and Marwedel, 2011)

Huber and Schoeberl (2009) compare measurement-based WCET with statically determined WCET, and it is evident, that the measurement-based WCET in most cases are more precise than those found through static analysis. However, in the comparisons, it is also stated that some of the measurement-based WCET are not safe. Even though the comparison with measurement-based WCET is not the main focus of their research, the unsafe measurements still indicate that safe measurement-based WCET values are difficult to make.

Since measurement-based methods rely on practical measurements on the given hardware or simulations of it, this method is easily applicable to new processors. The only requirement is that the program can run on the processor and one can gather precise measurements of the execution. With static methods, this would require a more in-depth knowledge of the timing properties of the new processor.(Lokuciejewski and Marwedel, 2011)

### 4.3.2 Hybrid of Static and Measurement-Based Methods

There exists a number of hybrid solutions which combine static and measurement-based approaches. As an example, static analysis is used to identify the basic blocks, and measurements are used to determine the execution time of these. This approach can e.g. be used on hardware where the timings of instructions are not specified.(Bernat et al., 2002)

Another approach is to use static analysis to derive information from the source code with the goal of determining which test data will ensure the measurement-based WCET to be safe. To keep the generated tests within a feasible quantity, Wilhelm et al. (2008) and Bernat et al. (2002) use an approach where smaller parts of the program are identified and their measured WCET are combined to get an overall WCET.

# 5 Schedulability Analysis

An aspect in real-time systems one needs to be aware of is how tasks are scheduled. It is the scheduler's responsibility to coordinate when tasks are allowed to run in accordance to a *scheduling policy*. This is much like a scheduler in a non-real-time system, but in a real-time system, it is important that tasks are scheduled such that they are able to meet their deadlines. An ordering of tasks that allows all deadlines to be met is called *schedulable*.

An important part of the schedulability analysis is WCET analysis described in Chapter 4. That is, tasks must be scheduled according to their period, deadline, and WCET.(Burns and Wellings, 2009; Wellings, 2004)

In the following, a task model is introduced and is used throughout this chapter. Afterwards, different *scheduling policies* and their properties are described. Furthermore, it is described how *schedulability analysis* is used to determine whether a real-time system is schedulable.

## 5.1 Simple Task Model

A simplified task model is used when describing the different scheduling policies to ease understanding the concepts, if not stated otherwise.(Burns and Wellings, 2009)

- All tasks are assumed to be periodic.
- The application is assumed to consist of a fixed set of tasks.
- The tasks are completely independent of each other.
- All system overheads, such as context-switching times, etc. are ignored.
- All tasks have deadlines, $D$, equal to their periods, $T$, that is $D = T$.
- All tasks have fixed WCET.
- No task contains any internal suspension points e.g. blocking I/O.
- All tasks execute on a single processor.

## 5.2 Scheduling Policies

Scheduling policies can be divided into two groups: *static* and *dynamic*. Static scheduling policies can predict whether the system is schedulable before runtime, while dynamic policies rely on run-time decisions (Burns and Wellings, 2009). In hard real-time systems, a desirable property of a given scheduling policy is the ability to predict that the deadlines of all critical tasks are kept.

In the following, three different scheduling policies are described. Specifically, we have chosen to describe Cyclic Executive Scheduling (CES), Fixed-Priority Scheduling (FPS), and Earliest Deadline First Scheduling (EDF) since these are commonly described in the literature.

The FPS and EDF scheduling policies allow the usage of preemption or non-preemption. According to the definitions, the preemptive schemes give basis for faster reaction to high-priority tasks than using a non-preemptive scheme. For this reason, the preemptive schemes are preferred. Furthermore, there exists a hybrid of the two schemes, called *deferred preemption*. In this scheme, the low-priority tasks are allowed to execute for a bounded period of time before they are preempted.(Burns and Wellings, 2009)

### 5.2.1 Cyclic Executive Scheduling

In CES, the schedule is constructed for a fixed set of periodic tasks before runtime. Therefore it is a static scheduling policy. Each task is implemented into one or more procedures, which are called in a specific order dictated by the schedule. Thus, this order must be constructed such that the deadlines of the tasks are upheld. The procedures are divided into *minor cycles*. The set of minor cycles needed to execute all tasks are referred to as the *major cycle*. Each minor cycle starts periodically and runs in a fixed amount of time.(Burns and Wellings, 2009)

As an example, consider three procedures, $a$, $b$, and $c$, each with a defined period and WCET shown in Table 5.1.

| Task | Type | Period/Deadline | WCET |
|---|---|---|---|
| a | Periodic | 10 | 2 |
| b | Periodic | 20 | 5 |
| c | Periodic | 40 | 3 |

**Table 5.1:** List of tasks and their periods/deadlines and WCETs.

The three procedures can be scheduled as shown in Figure 5.1.

As shown, when all procedures in a minor cycle are completed, the system waits until the period of the next minor cycle before executing the next procedure. This is done since executing a procedure too early can be as problematic as executing it to late, e.g. if the procedure depends on periodic external events.

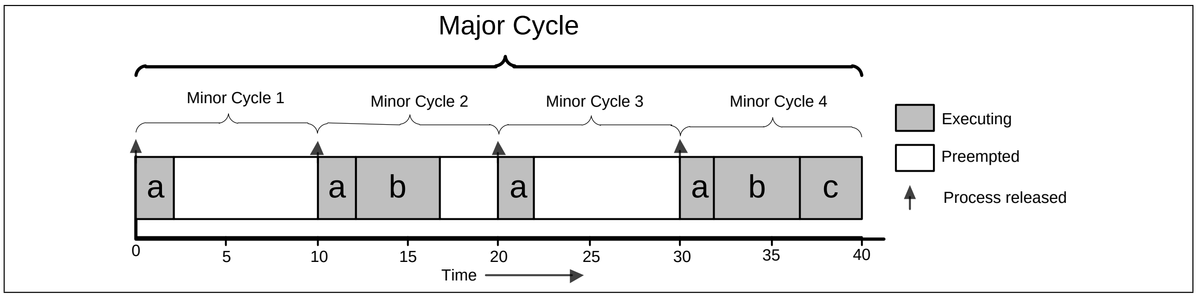


**Figure 5.1:** Example of CES with minor and major cycles.

Statically defining when different parts of the tasks are to be executed results in a number of positive effects. Since only known tasks are allowed and these are scheduled in the minor cycles while respecting the tasks deadlines and periods, it ensures that the tasks are schedulable, and no further analysis is needed. Furthermore, since there is no preemption and one has total control over execution order of the procedures, no locking mechanisms are needed for shared resources.(Burns and Wellings, 2009)

CES has been used for real-time systems, mainly because of its simplicity and low implementation requirements. However, it has shown to be difficult to use and especially maintain for larger systems, since a change in a procedure, which changes its WCET, requires a reevaluation of the schedule. Furthermore, it is known to be an NP-hard problem to construct the schedule, meaning that for larger systems, it might be associated with a substantial effort to do this (Burns and Wellings, 2009). As a result, some effort has been put into moving to task-based scheduling such as FPS (Audsley et al., 1995; Locke, 1992). However, research is still made in CES, e.g. Ravn and Schoeberl (2010) have presented an implementation of the policy for multiprocessors where the UPPAAL model-checker is used to generate schedules.

### 5.2.2 Fixed-Priority Scheduling (FPS)

In FPS, a task is given a fixed priority before the program is run, therefore it is a static scheduling policy. The scheduler always executes the task with the highest priority which, if care has been taken when selecting the priorities, should allow all deadlines to be upheld. This method is currently widely adopted for real-time systems (Burns and Wellings, 2009).

The policy can be applied directly to periodic and sporadic tasks. Sporadic tasks are scheduled as if they are periodic, defining their periods such that they represent their minimum inter-arrival time. However, for aperiodic tasks another approach must be taken since their releases cannot be predicted. Therefore, in case of aperiodic tasks, an execution-time server is used. This is a task with a fixed priority above the other tasks and with a specified period and WCET which still renders the system schedulable. When an aperiodic task enters the system, it is handled by this server, given the server has remaining resources. If the server exceeds the resources allocated to it, it stops handling aperiodic events. Effectively, this ensures that all other tasks that have been scheduled do not exceed their deadlines, while aperiodic tasks are handled in a best-effort manner. This technique can also be used for sporadic tasks in order to ensure that they are ignored if they exceed their minimum inter-arrival times.(Burns and Wellings, 2009)

The assigned priorities do not represent the importance of the tasks; they are derived from the timing requirements such as the deadlines of the tasks. In order to assign these priorities correctly, a number of methods exists. In the following the methods *rate-monotonic priority assignment* and *deadline-monotonic priority ordering* are described:(Liu and Layland, 1973; Leung and Whitehead, 1982)

**Rate-Monotonic Priority Assignment** The concept of rate-monotonic priority assignment is to assign the priorities, $P$, according to the periods, $T$, of a task such that the following holds for two tasks $i$ and $j$:

$$T_i < T_j \Rightarrow P_i > P_j \quad , \quad D = T. \tag{5.1}$$

This means that the shorter period a task has, the higher priority it is assigned.

It is known that if a set of tasks can be scheduled using fixed priorities, it can also be scheduled using the rate-monotonic scheme.(Liu and Layland, 1973)

**Deadline-Monotonic Priority Ordering** The principle of this priority assignment method is equal to the principle of rate-monotonic priority assignment, with the exception of the assumption $D = T$. This method assumes that the deadline is less than the period. Thus the following equation defines the priority assignment for two tasks $i$ and $j$ with deadline $D$:(Audsley et al., 1990)

$$D_i < D_j \Rightarrow P_i > P_j \quad , \quad D < T. \tag{5.2}$$

This results in tasks with a short deadline being prioritised higher than tasks with a long deadline.

One additional concern is *priority inversion.* This problem occurs in a special case where tasks share a common resource using locks. If a higher priority task is forced to wait for a lower priority task using the resource, the low priority task is not able to release its resource because it is not set for execution, effectively creating a deadlock. There exists a number of techniques to avoid this problem and to avoid other problems related to concurrent access to shared resources. These are described in following:

**Priority Inheritance** The concept of this protocol is to change the blocking task's priority to that of the suspended task, that is, inheriting the priority, if its current priority is lower. The priority scheme therefore becomes dynamic. As an example, consider a higher priority task $a$ which has been suspended by a lower priority task $b$. In this case, the priority of $b$ is set equal to $a$. Effectively, this means, that when using the FPS scheduling policy, the schedulable objects operate on two different priorities, namely *base* and *active* priorities. The base priority is provided when the object is instantiated. The active priority is the priority of the schedulable object when it is currently running and may be different from its base due to priority inheritance necessitating that the priorities are temporarily changed.(Wellings, 2004; Sha et al., 1990)

Priority Ceiling Protocols Generally, these protocols only allow the execution of a higher priority task to be blocked once by a lower priority tasks. One of them is the Immediate Ceiling Priority Protocol (ICPP). The concept of this is to raise the priority of a task as soon as it locks some resource regardless of the task blocking a higher priority task or not. Each resource is given a static ceiling value, which indicates the maximum priority of the tasks holding it. Thus, a task either has its own static priority or a priority corresponding to a resource that it holds.(English, 2004)

### 5.2.3 Earliest Deadline First Scheduling (EDF)

An alternative scheduling scheme to FPS is EDF which instead of using a priority derived from for instance the tasks' periods uses the absolute deadlines of the tasks at runtime. Hence, EDF is an instance of dynamic scheduling policies. A scheduler based on the EDF scheduling policy, will select the task with the closest deadline.(Insup Lee and Son, 2008)

With regard to handling aperiodic and sporadic tasks, EDF incorporates similar, however reinterpreted, ideas as those used in FPS. Specifically, the concept of execution-time servers can be incorporated but whereas the static system needs a priority to be assigned at run-time, the dynamic version needs to continuously determine its deadline each time it should be executed.(Burns and Wellings, 2009)

EDF exhibits similar problems with blocking and shared resources as FPS. However, whereas FPS can suffer from priority inversion, EDF can suffer from deadline inversion which occurs whenever a task is holding a resource that another task with a shorter deadline wants to acquire. Likewise, similar approaches as those found in FPS can be used for resolving the issue. The issue is relatively complex due to the dynamic nature of EDF which is apparent in the form of statically assigned priorities being easier to determine which task is currently blocking. In the case of EDF, one needs to resolve the issue by analysing which tasks with longer deadlines are active when the task is released. This can vary from one release to another.(Burns and Wellings, 2009)

As a final note, EDF is not supported by most RTOSs and programming languages. In Section 5.3, it is shown that EDF is superior to FPS with regard to utilisation so a reasonable question is why EDF is not being used as frequently as FPS. There are, however, a number of reasons for this. Initially, EDF requires a more complex run-time system to accommodate the dynamic nature. As a natural consequence, this leads to relatively high overhead for the process of scheduling. Secondly, it is easier to incorporate other factors, such as importance of criticality, into the notion of *priority* in FPS than it is into the notion of *deadline*. Finally, a significant disadvantage of EDF is how overload situations are handled. In the context of FPS, the behaviour is more predictable since it will always be the lower prioritised tasks that are missed. In EDF this situation exhibits a more unpredictable behaviour and is said to possibly experience a domino effect where many tasks miss their deadlines.(Burns and Wellings, 2009)

## 5.3 Schedulability Analysis

The prediction of worst-case behaviour gradually becomes more precise and models the actual execution more accurately as more details about the system are taken into account. Due to the extensive amount of details to add, the following descriptions are limited to only include a number of these for clarity. The approach taken in the following is to initially assume that the simple task model is used and then gradually refine the worst-case behaviour to include more details approximately reflecting a real system.

### 5.3.1 Utilisation-based Schedulability Analysis

Using rate-monotonic priority assignment for an FPS policy, it has been shown that a simple schedulability analysis can be made by only considering the utilisation of the set of tasks. If the utilisation-based schedulability analysis holds, the set of tasks is schedulable, that is, there exists a schedule for which the tasks will meet their deadlines. The utilisation-based schedulability analysis is given by:

$$\sum_{i=1}^{N} \left( \frac{C_i}{T_i} \right) \leq N \left( 2^{\frac{1}{N}} - 1 \right) \quad , \quad T = D, \tag{5.3}$$

where $C_i$ is the WCET, $T_i$ is the period for task $i$, and $N$ denotes the number of tasks. Note that $C_i/T_i$ describes the utilisation of task $i$.(Liu and Layland, 1973)

Increasing $N$, the bound asymptotically approaches 69.3 percent, meaning that if the utilisation of a set of tasks is less, it will always be schedulable regardless of how many tasks are running.

The schedulability analysis is sufficient and not necessary. Hence, if a set of tasks does not fulfil the test, it is not guaranteed that they cannot be scheduled to meet their deadlines. However, oppositely, if the test shows that the tasks can be scheduled satisfactorily, it is guaranteed that the system is schedulable.(Burns and Wellings, 2009)

To illustrate that the utilisation-based schedulability analysis is sufficient and not necessary, consider the set of tasks in Table 5.2.

As shown in the table the utilisation of both tasks is 50 percent, hence the overall utilisation is 100 percent. Since this is above, $2\left(2^{\frac{1}{2}} - 1\right) = 82.8$ percent, the test concludes that the

| Task | Type | Period/Deadline | WCET | Priority | Utilisation |
|---|---|---|---|---|---|
| a | Periodic | 6 | 3 | 1 | 0.50 |
| b | Periodic | 2 | 1 | 2 | 0.50 |

**Table 5.2:** A set of tasks with their corresponding period/deadline, WCET, priority, and utilisation.

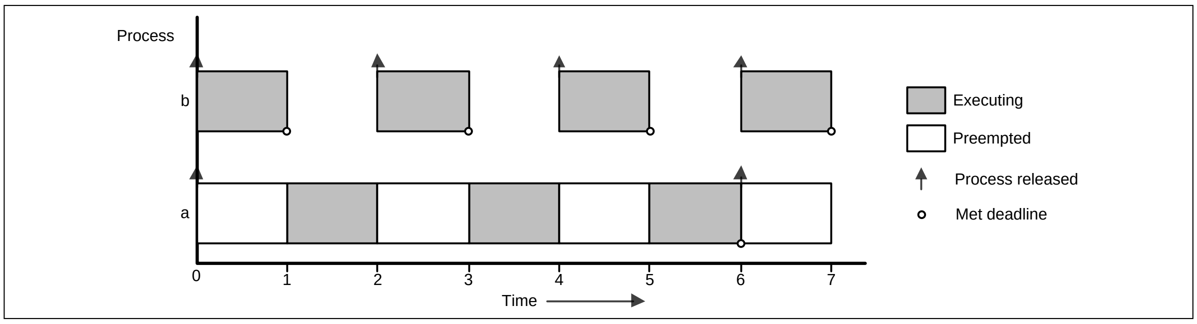


**Figure 5.2:** An illustration of a task set that according to the utilisation-based schedulability analysis cannot be scheduled successfully. However, as illustrated, this is not true.

system is not schedulable. However, as shown in Figure 5.2, the task set can actually be scheduled such that their deadlines are met.

Similar to the schedulability analysis for FPS, the following schedulability analysis holds for EDF:(Liu and Layland, 1973; Burns and Wellings, 2009)

$$\sum_{i=1}^{N} \left( \frac{C_i}{T_i} \right) \leq 1 \quad , \quad D = T. \tag{5.4}$$

As shown, the utilisation-based schedulability analysis for EDF is simpler than that for FPS. Instead of processor capacity being reduced according to the number of tasks being scheduled, the similar test for EDF only takes into account the full capacity of the processor. The conclusion is thus that higher utilisation can be achieved with EDF than FPS, under the assumption that the overhead in respect to the dynamic behaviour does not exceed this gain. Another conclusion is that the test is sufficient and necessary for schedulability.

### 5.3.2 Response Time Analysis

The utilisation-based schedulability analysis is not safe and should not be applied on more general models than the simple task model (Audsley et al., 2002). Another, and more precise approach, is to use Response Time Analysis (RTA) which takes into account the WCET of a task and interference from other tasks. In contrast to the utilisation-based schedulability analysis, it is considered sufficient and necessary for both FPS and EDF. In general RTA can gradually be refined by including additional details about the task model.

The given model for the Worst Case Response Time (WCRT) applies only to FPS. Baruah et al. (1990) has proposed a similar model for the EDF scheduling policy, but an analysis of it is considered out of scope for this project.

In its most general form, the WCRT of a task is given by:(Joseph and Pandya, 1986)

$$R_i = C_i + I_i, \tag{5.5}$$

where $R_i$ is the WCRT of task $i$, $C_i$ is the WCET of task $i$ and $I_i$ is the maximum interference task $i$ can experience in the time interval $(t, t + R_i)$. Note that a discrete time model is used in this relation, meaning that the completion of a task, can happen at the exact same time as a higher-priority task is released.

The maximum interference time, $I_i$, a task $i$ experiences depends on how often it is pre-empted in favour of executing a higher priority task. The maximum possible interference time task $i$ can experience will occur in what is referred to as the *critical instant*. A critical instant is when all higher-priority tasks are released at the same time as task $i$. Assuming that all tasks are released at time 0, then a higher-priority task, $j$, will in the time interval $[0, R_i[$ be released at maximum $\lceil R_i/T_j \rceil$ times. Note the use of the ceiling function is necessary to make safe calculations. Each of these releases of task $j$ will result in an interference of size $C_j$. Considering that this should be calculated for each higher-priority task than $i$, the maximum interference time of task $i$ can be expressed as:(Audsley et al., 2002)

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \tag{5.6}$$

where $hp(i)$ is the set of higher priority tasks in respect to task $i$. Substituting this into Equation 5.5 gives the following:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{5.7}$$

As shown, $R_i$ is present at both sides of the equation. To solve this, it is suggested to form the recurrence relationship:(Audsley et al., 2002)

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j, \tag{5.8}$$

where $R_i^0, R_i^1, \ldots, R_i^n, \ldots$ are monotonically non-decreasing. When the fixed-point is found, that is $R_i^{n+1} = R_i^n$, the solution has been found. To start the recurrence relationship, $R_i^0$ can be set equal to the WCET of the task.

Using the tasks set in Table 5.2, the WCRT for each task can be found. Since task $b$ has the highest priority, it does not suffer from interference from task $a$. The WCRT for $b$ is therefore $R_b = 1$.

For task $a$, the WCRT is given by forming the recurrence relationship:

$$\begin{aligned}
R_a^0 &= 3 && \Rightarrow \\
R_a^1 &= 3 + \left\lceil \frac{3}{2} \cdot 1 \right\rceil = 5 && \Rightarrow \\
R_a^2 &= 3 + \left\lceil \frac{5}{2} \cdot 1 \right\rceil = 6 && \Rightarrow \\
R_a^3 &= 3 + \left\lceil \frac{6}{2} \cdot 1 \right\rceil = 6
\end{aligned} \tag{5.9}$$

As shown, the WCRT for each task are below or equal to their respective deadlines. Hence, the task set is schedulable.

### Blocking

In the simple task model, the tasks are independent, that is, they are not allowed to block each other. This is a problem in many applications, e.g. if locks are used to synchronise tasks. Therefore, it is advantageous to take into account blocking when analysing the WCRT of a task.

The problem introduced by blocking can be illustrated by considering the set of tasks shown in Table 5.3. As shown, the total utilisation is 75 percent which according to the utilisation-based schedulability analysis, implies that the task set is schedulable. However, as shown in Figure 5.3, this is not the case when introducing blocking into the simple task model. As shown, having task $a$ executing with a lock, thereby blocking task $b$, results in task $b$ not meeting its second deadline.

| Task | Type | Period/Deadline | WCET | Priority | Utilisation |
|---|---|---|---|---|---|
| a | Periodic | 14 | 7 | 1 | 0.50 |
| b | Periodic | 4 | 1 | 2 | 0.25 |

**Table 5.3:** A set of tasks with their corresponding period/deadline, WCET, priority, and utilisation.

To compensate for blocking, the maximum blocking time for a task $i$, denoted $B_i$, is introduced in the RTA Equation 5.7:(Burns and Wellings, 2009):

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{5.10}$$

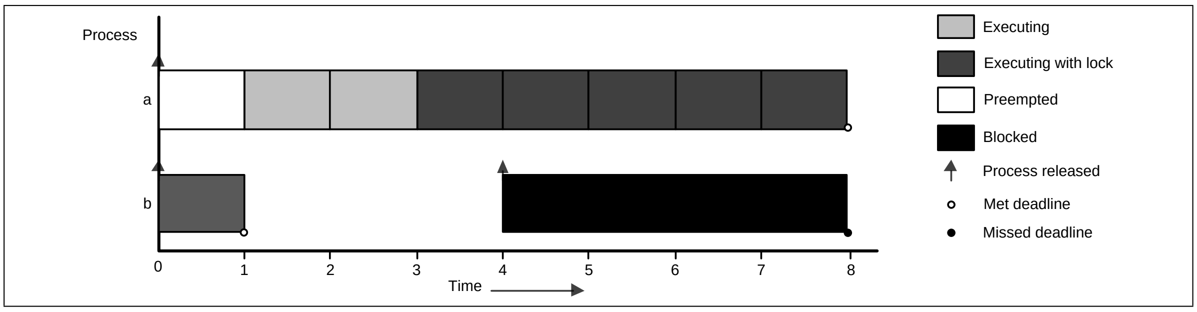


**Figure 5.3:** Example of two tasks using a shared resource.

The exact calculation of $B_i$ depends on which priority protocol is used. As previously mentioned, there exists a set of protocols called priority ceiling protocols. Furthermore, priority inheritance can be used.

Using the priority inheritance protocol, it has been shown that the number of times a task can be blocked by lower priority tasks is bound (Burns and Wellings, 2009). In brief, it is known that if $K$ critical sections are present, then a task can at maximum be blocked $K$ times. Formally, this can be written as:

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k), \tag{5.11}$$

where $usage(k,i)$ is a binary function returning true if the critical section $k$ with WCET $C(k)$, is used by at least one task, with a lower priority than that of $i$, and at least one task with a greater or equal priority to that of task $i$. If this is not the case, the function returns false.

In respect to the priority ceiling protocols, the maximum blocking time is simpler since the tasks are only allowed to be blocked once by any lower priority task. The maximum blocking time is therefore not the summation of the WCET for each critical section where a resource is shared with a lower priority task. Instead, it is given by the maximum WCET of these time periods:(Burns and Wellings, 2009)

$$B_i = \max_{k=1}^{K} \left(usage(k,i)C(k)\right). \tag{5.12}$$

Extending the RTA to take this into account is considered sufficient.

The scheduling equation presented in this section does not take into account context-switches, multiprocessor systems etc. that further complicates the equation. Also release jitter has been omitted. Release jitter describes the maximum variation in a task's release and can be used to correctly model how sporadic tasks interfere other tasks (Burns and Wellings, 2009). These are considered out of scope for this project.

### 5.3.3 Model-Based Schedulability Analysis

Another approach is to use *model-based schedulability analysis*. The concept is to model the application and the scheduling policy and use model-checking to verify the schedulability of the tasks. It must be noted that the reliability depends on a correct model of the application, a correct model-checker, and a correct specification.

Timed automata can be used for constructing the model. This approach has been shown to support schedulability analysis of real-time systems containing periodic and sporadic tasks, concurrency, execution sequence dependencies, and synchronisation. Furthermore, both preemptive and non-preemptive systems can be modelled (Fersman and Yi, 2004). Other research has provided a framework that allows model-based schedulability analysis to be conducted on multiprocessor systems.(Nicolescu and Mosterman, 2010)

The advantage of using the model-based approach is that it can result in more precise results compared with the more traditional approaches. Since traditional methods yield pessimistic estimates, they potentially deem systems non-schedulable even if they are in fact schedulable. This means that model-based schedulability analysis can verify the same systems to be schedulable since the approach more closely models the scheduler of a real system.(Bøgholm et al., 2008a)

#### Modelling the Tasks

There exist different approaches to modelling the system in timed automata. One approach is to model the system using a simple model of the scheduling policy and using a generic representation of the tasks, which states their properties such as type of task, WCET, and deadline (Fersman and Yi, 2004). Another approach is to model the scheduler in detail, that is, modelling the exact behaviour of context-switches etc., thereby potentially resulting in a more accurate schedulability analysis (Bøgholm et al., 2008a). Furthermore, it can be advantageous to model the tasks in detail in this approach as when determining the WCET, thereby achieving a synergy effect by combining the two modelling approaches. More specifically, the model of the cache from WCET analysis, can be combined with the model of the context-switches from the schedulability analysis.

Depending on the modelling approach, different properties can be checked in order to determine the schedulability of the task set. Fersman and Yi (2004) model a special error state that is reached if a deadline is exceeded. Schedulability is then checked by verifying the reachability of this state. In contrast, Bøgholm et al. (2008a) uses an approach where the system is proven schedulable if no deadlocks are found in the model.

In the following, an example showing the advantage of model-based schedulability analysis compared to utilisation-based schedulability analysis using FPS is given. Initially, consider the tasks in Table 5.4.

As shown, the system consists of one periodic task and two sporadic tasks. According to the utilisation-based schedulability analysis the system is not schedulable because the total

| Task | Type | Period/Deadline | WCET | Priority | Utilisation |
|---|---|---|---|---|---|
| a | Periodic | 2 | 1 | 1 | 0.50 |
| b | Sporadic | 4 | 1 | 2 | 0.25 |
| c | Sporadic | 4 | 1 | 3 | 0.25 |

**Table 5.4:** A set of tasks with period/deadline, WCET, priority, and utilisation. The example is inspired by Bøgholm et al. (2008b).

utilisation is 100 percent since sporadic tasks are expected to be released at each minimum inter-arrival time. However, the utilisation-based schedulability analysis is not capable of determining how sporadic tasks are released. Suppose they are released by the periodic task executing the code shown in Listing 5.1.

```
boolean releaseB = true;
if(releaseB){
    fire(b);
    releaseB = false;
}
else{
    fire(c);
    releaseB = true;
}
```

**Listing 5.1:** Release pattern of the two sporadic tasks *b* and *c*. The example is inspired by Bøgholm et al. (2008b).

As shown, the releases of the sporadic tasks are dependent on the *releaseB* variable, effectively making their releases mutually exclusive thereby in practice resulting in a maximum utilisation of 75 percent. Therefore, by modelling this mutually exclusive pattern, the system can be shown to be schedulable.

### Automatic Model Generation

Creating models for schedulability analysis can be time consuming and potentially error prone, especially if individual methods are modelled. One solution to this problem is to generate models from the program source code. An example of such a tool is Schedulability Analyzer for Real-Time Systems (SARTS) (Bøgholm et al., 2008a) which is capable of translating code written in a specific real-time Java profile called SCJ2 into UPPAAL models. Using the UPPAAL model-checker, it is possible to check if the model is deadlock-free. In case no deadlocks are present, the system is regarded as schedulable. SARTS models the scheduling policy FPS by default but in principle any other scheduling policy can be used.

# 6 Issues Identified During Analysis

From the previous analysis, a wide variety of topics have been analysed in the domain of real-time systems. One of these topics will be analysed further in the remaining part of the report.

Of interesting topics the authors found the topic of high-level languages for real-time embedded system development to be interesting. This topic was discussed in Section 3.1. The movement towards introducing new languages to the field and easing development poses some interesting problems especially with regard to ensuring both functional correctness and timing correctness.

We have assessed that having high-level programming languages for real-time embedded systems development holds great potential in many ways such as possibly increasing productivity and easing maintenance. Furthermore, if high-level languages become the norm for real-time systems development, it potentially reduces the development cost. In the discussions, we identified two programming languages that we think could be of particular interest in terms of real-time systems development. To recapitulate, these were Java and C#. Of these, we think that an analysis of Java is the most interesting since we have assessed that applying real-time concepts to a programming language like C# is a too cumbersome task because only minor prior research has been conducted in this field. Instead, Java for real-time systems development is an active research area. Hence, Java potentially creates a foundation for understanding how real-time concepts can be practically applied to high-level languages. Hence, gaining this knowledge can potentially be used subsequently to apply similar concepts to C#.

Analysing the problem of Java for real-time systems development is broad. Hence, we analyse a subset of it. Generally, the following analysis consists of three overall topics:

**Real-Time Specification for Java (RTSJ)** This initial analysis concentrates on the RTSJ which is a specification that describes facilities that make real-time systems development convenient in Java. Knowledge in the specification is therefore a good starting point to understand how to implement real-time Java systems.

**Real-Time Java profiles** The RTSJ meets the requirements of many different real-time systems. As a consequence of this, it is a relatively large specification, with a wide variety of details that target the needs of different real-time applications. Thus, the RTSJ may be unsuitable for a concrete target such as safety-critical applications. Therefore, research has been made to design and implement real-time Java profiles inspired by the RTSJ, with the purpose of subsetting it. The profiles are later evaluated according to a set of criteria.

**Practical Example** To illustrate the problems of real-time systems development using Java, we develop a practical example in each of the chosen real-time Java profiles to gain practical experiences. This includes conducting WCET analysis and schedulability analysis on the system.

# Part II

# Real-Time Java

# 7 Real-Time Specification for Java

The development of making Java facilitate real-time development started with the introduction of the RTSJ version 1.0 initiated as JSR 1 under the JCP in 2001. The RTSJ is an attempt to add a set of extensions to the JVM and to the class libraries of Java to facilitate real-time programming (Wellings, 2004). In the following, the newest version of the RTSJ will be used, that is, version 1.0.2 (RTSJ.org, 2010). Furthermore, the main extensions to Java introduced by the RTSJ are described and these are related with the concepts of real-time systems in general described in Part 1.

## 7.1 Memory Management

As described in Section 2.2, there are memory management concerns that must be accounted for in real-time embedded systems. These concerns constitute limiting the amount of needed memory and increasing the predictability of the program. To summarise, limiting the amount of needed memory is of concern since embedded devices typically have a very limited amount of memory at their disposal. Increasing the timing predictability is necessary to make WCET and schedulability analyses of the program possible. Regarding these two concerns, garbage collection comes into play. Garbage collection can help keeping the memory consumption low, by freeing memory, but it is known that garbage collection affects the timing predictability of real-time systems (Wellings, 2004). However, as mentioned in Appendix A, experiments with real-time garbage collectors for Java have shown good results in terms of timing predictability and memory overhead.

### 7.1.1 Memory Areas

To allow memory management which is not affected by garbage collection, the RTSJ introduces *immortal* and *scoped* memory, which both are logically placed outside the heap. The RTSJ does not exclude heap memory, since this might be useful in some applications. The memory class hierarchy of RTSJ is depicted in Figure 7.1.(RTSJ.org, 2010)

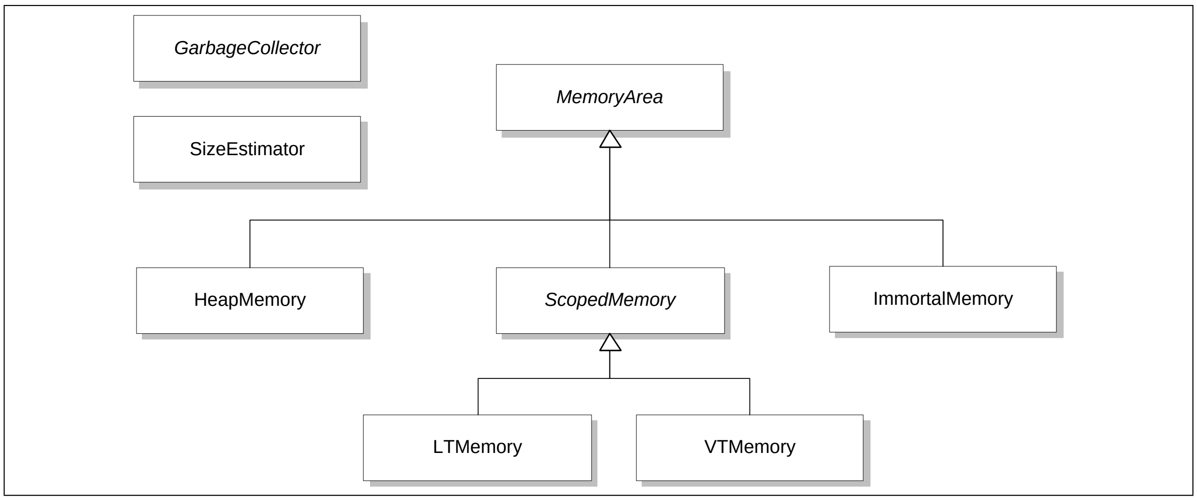


**Figure 7.1:** Class diagram depicting the structure of the memory hierarchy in RTSJ.

As shown, RTSJ uses the `MemoryArea` class as an abstract class from which all other memory areas are derived. The different `MemoryArea` specialisations are described in the following:(RTSJ.org, 2010)

`HeapMemory` This memory area enables objects to be allocated on the heap. This area is garbage collected at unpredictable time intervals, and should therefore not be used for time-critical tasks. As expected for memory allocated on the heap, it is guaranteed that objects are never freed until they no longer can be reached.

`ImmortalMemory` Immortal memory is shared between all threads and schedulable objects in the application. There is only one immortal memory area in a program, and a call to its *`instance()`* method returns a reference to this area, thereby yielding the *Singleton* design pattern.

The time used for allocating objects in immortal memory is not known and allocations should therefore be done in the initialisation phase of the system. This is a disadvantage since it requires the programmer to allocate all objects in advance. However, the advantage is that it is guaranteed to never be garbage collected since objects are never removed from immortal memory, thus accommodating timing predictability.

`ScopedMemory` Scoped memory has a well-defined lifetime and the areas can be created as needed. An object allocated from a memory scope is first freed when the scope is no longer active. To keep track of whether a scoped memory area is active or not, a reference counter can be used to keep track of how many times the area has been entered and exited. When the reference count eventually becomes zero, the allocated memory for the scope can be freed. In order to use the scoped memory areas more effectively, it is necessary to estimate the size of the objects required to be allocated in it. If this is under-estimated, the programmer will get an *OutOfMemoryError* exception, and if over-estimated, the overall memory consumption of the program might exceed the overall available memory. As a help for estimating the size of the required memory, the `SizeEstimator` class is provided.

`LTMemory` A subclass of `ScopedMemory` is `LTMemory`. The needed time for creating an object in this area is timing predictable. That is, the time needed for allocating an object is directly proportional to the size of the object. Even in situations where the memory area is exhausted, a new allocation using `LTMemory` will never provoke garbage collection to be initiated.

`VTMemory` This is also a subclass of `ScopedMemory`. It is similar to `LTMemory`, however with the difference that allocating objects in `VTMemory` is allowed to take variable amount of time. The advantage of this type of memory area is that allocations typically can be made faster than in `LTMemory` since its implementation can be optimised without taking into account timing predictability.

### 7.1.2 Assignment Restrictions

Since memory in different types of memory areas are collected differently, a set of restrictions on assignments between the different areas must be introduced. To illustrate the necessity for such restrictions, consider Example 4.(Wellings, 2004)

**Example 4**
Object A and B are created in scoped and heap memory, respectively. Object B is referencing object A. Now consider what happens when the reference count of the scoped memory gets zero. Object A is collected and object B now becomes a dangling pointer.

The assignment restrictions dictated by RTSJ are shown in Table 7.1. If a restriction is violated, an unchecked `IllegalAssignmentError` exception is thrown.

| Memory Area | Reference to Heap | Reference to Immortal | Reference to Scoped |
|---|---|---|---|
| Heap | × | × | |
| Immortal | × | × | |
| Scoped | × | × | ×[1] |
| Local variable | × | × | × |

[1]If referencing to same or a less deeply nested scope.

**Table 7.1:** Assignment restrictions dictating how references can be made between the different memory types. The × indicates that a reference is allowed.

Since RTSJ does not change the Java language, the restrictions must be enforced at runtime by the real-time JVM. Alternatively, one could imagine it to be the responsibility of the Java compiler to enforce the restrictions, e.g. by making static analysis. However, this approach requires the JVM to trust that such an analysis has been undertaken correctly.

## 7.2 Schedulable Objects and Scheduling

In standard Java, threads have priorities. Hence, one could mistakenly believe that Java guarantees the order in which threads are executed. Since the JVM may rely on the host OS which does not necessarily support priorities, the JVM can be forced to map Java threads to OS threads running with the same or less priorities than declared for the Java threads (Wellings, 2004). Thus, both the JVM implementation and OS must ensure correct mapping between Java threads and OS threads.

To make scheduling comply with a program's timing predictability, the RTSJ provides a number of classes. These are shown in Figure 7.2. Furthermore, the entities that can be scheduled are generalised from only concerning threads towards the notion of *schedulable objects*. Schedulable objects must implement the `Schedulable` interface.(RTSJ.org, 2010)

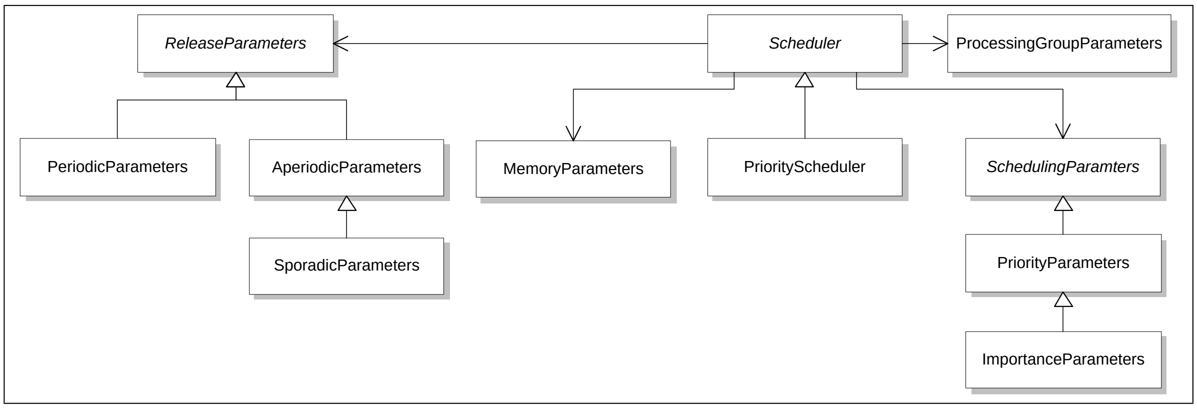


**Figure 7.2:** Class diagram depicting the `Scheduling` class and its parameter classes in RTSJ.

### 7.2.1 Parameter Classes

As shown in Figure 7.2, the `Scheduling` class can use the parameters: `ReleaseParameters`, `ProcessingGroupParameters`, `SchedulingParameters`, and `MemoryParameters`. These parameters are associated with each schedulable object and their meanings are described in the following:(RTSJ.org, 2010)

`ReleaseParameters` This is an abstract class whose purpose is to specify how often a schedulable object is released, its WCET, and its relative deadline for each release. Furthermore, it can be specified how to act if the WCET is overrun, and how to act if the deadline is missed. Furthermore, as shown in the class diagram, `PeriodicParameters` and `AperiodicParameters` classes are derived from the `ReleaseParameters` class. This way it is possible to specify further information required by these specialisations. In `PeriodicParameters`, the programmer is e.g. required to specify the start time and period of the schedulable object. In `AperiodicParameters` this cannot be specified due to the nature of aperiodic tasks. Since it is likely that aperiodic tasks are released before the previous has been completed, the RTSJ introduces an internal queue where the tasks can be queued for later processing. Notice that `AperiodicParameters` is further specialised into `SporadicParameters` which adds a minimum inter-arrival time to the schedulable object.

`SchedulingParameters` This is an abstract class whose subclasses provide the parameters to be used by the `Scheduler`. As shown, it has two subclasses, namely `PriorityParameters` and `ImportanceParameters`. Assigning the `PriorityParameters` class to each schedulable object, the `Scheduler` can determine the execution order, based on the priority defined in the parameter. This priority can be further specialised using the `ImportanceParameters` class. The two derived classes are used along with the FPS scheduling policy, and if another policy is to be implemented, new specialisations of the `SchedulingParameters` should be made.

`MemoryParameters` These parameters are used to specify information regarding the memory usage of the schedulable objects. One of the constructors of the class requires two arguments. Firstly, the maximum memory amount to be allocated by the schedulable object in its initial memory area and secondly, the maximum amount of memory to be allocated in the immortal memory.

`ProcessingGroupParameters` This class can be associated with one or more schedulable objects. Its purpose is to group schedulable objects, by specifying a common period, WCET, and deadline. An execution-time server is created for the group with the same parameters as those provided for the group. Grouping aperiodic tasks in an execution-time server ensures that released aperiodic tasks are handled as long as the server does not exceed its timing requirements, as described in Chapter 5. If the timing requirements are exceeded, it can be specified how to act upon it. This approach therefore solves the issue of handling an unspecified amount of aperiodic tasks without having the other periodic tasks starve and hence miss their deadlines.

The code example shown in Listing 7.1 illustrates how some of the parameters are used to create a schedulable object.

```
public class PeriodicExample {
  public static void main(String[] args) {
    RelativeTime S = new RelativeTime(0, 0);
    RelativeTime T = new RelativeTime(1000, 0);
    RelativeTime C = new RelativeTime(500, 0);
    RelativeTime D = new RelativeTime(1000, 0);

    SchedulingParameters schedParams = new PriorityParameters(PriorityScheduler.
        getMinPriority(null) + 10);
    ReleaseParameters relParams = new PeriodicParameters(S, T, C, D, null, null);

    RealtimeThread rt = new ThreadExample(schedParams, relParams);
    rt.start();
  }
}
```

**Listing 7.1:** Code example illustrating the usage of `SchedulingParameters` and `ReleaseParameters` when creating a real-time thread. The class `ThreadExample` implements the actual logic conducted in the thread and is omitted for clarity.

As shown, `SchedulingParameters` defines that the schedulable object gets a priority 10 higher than the minimum allowed priority. Furthermore, the `PeriodicParameters` defines the period of the periodic task to be 1000$ms$, its WCET to be 500 milliseconds, and its deadline to be 1000$ms$, and that it should be released immediately.

### 7.2.2 Scheduler

The RTSJ specifies a priority based scheduler, in the `PriorityScheduler` class, which is based on the FPS scheduling policy. This specification is made as a specialisation of the abstract `Scheduler`, such that, if wanted by the programmer, other scheduling policies can be implemented.

The `Scheduler` class opens for implementations of feasibility analysis of the program by defining the *`isFeasible()`* and *`setIfFeasible()`* methods. The first is used to query the current system for its feasibility. E.g. such a check could be made before executing the schedulable objects. The second method can be used if changes to the current parameters of the schedulable object are needed. *`setIfFeasible()`* changes the parameters in an atomic operation if the new configuration is feasible. Note that the behaviour of the two methods depends on the exact implementation, which in some cases might be relatively simple, and might not give a realistic estimate of the feasibility of the program.(RTSJ.org, 2010)

## 7.3 Real-Time Threads

Conventional Java provides the notion of threads through its `Thread` class. However, for real-time applications, this notion is too broad and restricts the expressiveness by excluding the necessary properties that are inherent in real-time tasks. Some of those properties comprise the notion of deadlines and scheduling properties of the particular thread. The RTSJ does not explicitly capture the properties of the three types of real-time tasks but only provides the notion of *real-time threads* which is realised by the provision of the `RealtimeThread` and `NoHeapRealtimeThread` classes. Both classes inherit from the traditional Java `Thread` class. However, note that to fully support real-time threads in Java, requires that certain parts of the underlying JVM are changed to accommodate the needs of real-time properties (Wellings, 2004). This could be the provision of high-resolution clocks that enable a fine granularity of scheduling.

### 7.3.1 NoHeapRealtimeThread

The `NoHeapRealtimeThread` class is similar to the `RealtimeThread` class, however with the important difference that the thread only accesses non-heap memory. This makes it suitable in cases where one does not want the garbage collector to interfere. `NoHeapRealtimeThread` inherits from the `RealtimeThread` class with the only difference that all constructors include a `MemoryArea` and a `MemoryParameters` in the parameter list.(RTSJ.org, 2010)

### 7.3.2 RealtimeThread

The `RealtimeThread` class contains a number of constructors corresponding to how many of the properties of scheduling the programmer wants to specify. By default, real-time threads inherit the scheduling parameters from their parent, that is, either the properties of the `Thread` class or from the schedulable object that created the real-time thread. Given that the parent has no scheduling parameters, default values of the scheduler are used instead.

The `RealtimeThread` class implements the methods dictated by the `Schedulable` interface which is described in Section 7.2. The `RealtimeThread` class also implements methods that are convenient, given the thread executes periodically. In this regard, the *`waitForNextPeriod()`* method can be used to suspend the thread until its next release. The thread will be blocked until a call is made to the *`scheduleperiodic()`* method. In case the thread is aperiodic or sporadic, no call is made to *`waitForNextPeriod()`* since these types of tasks do not have a period. As of RTSJ version 1.0.2, waiting for the next aperiodic release has to manually coded (RTSJ.org, 2010). However, in JSR 282, that is, the draft version of RTSJ 1.1, a request is made for a *`waitForNextRelease()`* method to support aperiodic releases (Oracle, 2009).(Wellings, 2004)

### 7.3.3 ReleaseParameters

It is important to note what happens to the real-time thread if a timing requirement is not upheld. Given that a cost overrun occurs, the real-time thread will immediately be descheduled and will not be considered for further rescheduling before its next release occurs or before the WCET is increased with a call to *`setCost()`* which is an instance method of the `ReleaseParameters` class. Also, when the cost overrun occurs, the overrun handler associated with the schedulable object is released. On the other hand, if a deadline miss occurs, the thread is not immediately descheduled and the real-time thread is allowed to continue. However, the handler associated with the deadline miss is immediately released. This design choice is primarily based on the fact that a cost overrun is more crucial for the system than missing a deadline. This is currently the behaviour of both periodic, aperiodic, and sporadic real-time threads.(RTSJ.org, 2010)

Note that the RTSJ as of version 1.0.2 does not specify how periodic, aperiodic, and sporadic threads are created. Instead, the RTSJ distinguishes between them in terms of their release parameters which can either be an instance of `PeriodicReleaseParameters`, `AperiodicReleaseParameters` or `SporadicReleaseParameters`. Due to this reason, the programmer has to make the explicit distinction.(RTSJ.org, 2010) In Listing 7.2, a code example of how a periodic real-time thread can be implemented is shown.

```
public class PeriodicRealtimeThread extends RealtimeThread {
    public PeriodicRealtimeThread(PriorityParameters priParams, PeriodicParameters
        perParams) {
        super(priParams, perParams);
    }

    public void run() {
        boolean continueRunning = true;
        while(continueRunning) {
            ...

            continueRunning = waitForNextPeriod();
        }
        ...
    }
}
```

**Listing 7.2:** An example of implementing a periodic real-time thread in RTSJ.(Wellings, 2004)

In this code example, it is worth noticing the use of the `PeriodicParameters` parameter of the constructor which is used for periodic real-time threads in RTSJ. Also, for making the thread behave as a periodic real-time thread, the overridden ***run()*** method includes a loop in which the logic of the thread is placed. At the end of the loop, a call to ***waitForNextPeriod()*** is made. In case a call to ***waitForNextPeriod()*** is made after exceeding the deadline, it may be that it returns false which consequently breaks the loop of the periodic thread.

## 7.4 Asynchronous Event Handling and Timers

Asynchronous event handling is used to support handling externally and internally fired events. External events are fired from the underlying RTOS or from interrupts, and internal events are fired from the running program itself. The architecture of this event system consists of three layers that are described in a top-down manner in the following:(Dibble, 2002)

Interface Layer This layer is responsible for listening for external events and associate these with corresponding events defined in the async event layer.

Async Event Layer This layer consists of defined events of the class `AsyncEvent` and its subclasses. Each instance of these classes represents an event which can be fired, resulting in all of the Asynchronous Event Handlers (AEHs) which are associated with the event being called asynchronously. The event can be bound to external events defined in the interface layer. Internal events are fired by explicitly firing one of the declared events in this layer.

Async Event Handler Layer This layer consists of the actual AEHs which are associated with one or more events in the async event layer.

AEHs function as schedulable objects similar to threads in RTSJ, and provide a similar API with regard to configuring scheduling parameters, release parameters, memory parameters, and processing group parameters. It is implementation-dependent how the JVM handles these events and provides a real-time thread that can execute the handler. Some implementations create a new thread for each fired event (Dibble, 2002). This can possibly introduce some latency between the event is fired and a thread is responsible for executing the handler (Wellings, 2004). In order to solve this problem, the `BoundAsyncEvent` subclass of `AsyncEvent` supports binding the handler to a specific thread, which is dedicated to executing the particular handler. Thus, only one bounded AEH can be associated with a thread. However, this approach results in an increased memory usage since each additional thread require some memory to be reserved to it.

AEHs can be used for a variety of purposes of which two are briefly described in the following:(Dibble, 2002)

Fault Handling AEHs can be used to handle faults in the running program instead of using exceptions or error codes in return values. This results in much more flexible error handling where it is not necessarily handled locally at the origin of the fault. Furthermore, it allows for handling errors without disturbing the flow of the originating thread. As an example, this feature could be used by a logging system to log errors without disturbing the execution time of the threads generating the errors.

Periodic, Aperiodic and Sporadic Events AEHs can also be used as a general notification system, allowing different parts of the program to listen and react to different events. This way, it is possible to create AEHs which react to periodic, aperiodic, and sporadic events in the system. This can potentially replace the real-time thread approach for handling such events.

## 7.5 Asynchronous Transfer of Control

Asynchronous Transfer of Control (ATC) is a mechanism in RTSJ that allows schedulable objects to interrupt each other. This is described as a necessity for certain problems of real-time systems such as for:(Wellings, 2004)

Error Recovery Consider the example where a set of schedulable objects operates in the same area and that one of them observes a fault. This needs to be reported to the other objects such that an error recovery can be started. In a real-time system, the error recovery process might be time critical and hence error reporting must not be deferred.

Mode Change Since real-time systems typically must act according to the current environment, they often have several modes of operation. An example given by Wellings (2004) is a fly-by-wire aircraft which has a take-off, cruising, and a landing mode. These modes must carefully be changed according to the environment. In general the mode changes can be planned, however, in some situations they cannot and the current mode needs to be interrupted where ATC can be used for this.

Generally, the problems can be handled by having each schedulable object poll for notifications. However, this is not considered to provide timely responses to the interrupts. Therefore, RTSJ specifies that ATC, allowing the point of execution in one schedulable object to be changed by another schedulable object, must be implemented. This gives rise to new problems such as how to write correct code. To accommodate some of the problems, ATC is based on the following principles:(Wellings, 2004)

- Schedulable objects must explicitly state that they allow ATC to be delivered. This is done by adding the `AsynchronouslyInterruptedException` to the throw clause of a method.
- When executing synchronised methods and statements, delivery of ATC is always deferred.
- When the termination of a delivered ATC is done, the execution point of the schedulable objects before being interrupted is not resumed. This means that ATC is closer to a goto statement than an actual procedure call.

## 7.6 Physical and Raw Memory Access

In Java and to some extent in real-time Java, programmers can completely ignore how the hardware physically handles memory in terms of translating virtual addresses to physical ones, and which memory is used together with the access policies. However, on some embedded devices and in some particular real-time applications, this fact cannot be ignored (Wellings, 2004). The areas which are of concern are described in the following:

- First of all, demand paging significantly affects the deterministic behaviour of memory access and hence in some critical parts of the application, this feature can advantageously be disabled.
- On many embedded devices, the physical memory available may not be comparable to desktop counterparts. For example, different types of memory may be available corresponding to whether fast read/write access is needed or whether slow but relatively

large amount of memory should be used. This could be represented by flash, ROM, and RAM. A programmer may want to gain a performance boost by taking this spectrum of memory into account when allocating objects by considering their access patterns and so on.(Dibble, 2002; Wellings, 2004)

- An important aspect of having greater control of physical memory relates to raw memory access. Traditionally, programs runnning on the JVM see memory solely as a stack of primitive data and objects which prove useful when just considering algorithms and computations that do not involve interacting with external resources that are not present in the JVM. However, due to real-time embedded systems often interact with the real world, the programs often need to interface with memory-mapped I/O devices. Thus, these types of applications need to conduct raw memory access for allowing this interaction. Traditionally, raw memory access is implemented by means of native methods normally written in C or assembly which can be realised by using Java Native Interface (JNI). However, this circumvents the secure environment in which the JVM runs because none of Java's security and protection arrangements function in native methods. Hence, the JVM trusts that the native methods do not pose dangerous pointer errors and the like. Such errors may be crucial in safety-critical systems.(Dibble, 2002)

To accommodate the aforementioned three needs, the RTSJ dictates the presence of the `PhysicalMemoryManager` class whose purpose is to manage memory allocations to physical memory objects by locating the specific memory requested by the programmer. These objects are instances of `VTPhysicalMemory`, `LTPhysicalMemory`, `ImmortalPhysicalMemory`, `RawMemoryAccess`, and `RawMemoryFloatAccess`. `ImmortalPhysicalMemory`, `VTPhysicalMemory`, and `LTPhysicalMemory` provide the physical counterparts to `ImmortalMemory` and the `ScopedMemory` derivatives, `VTMemory` and `LTMemory`, mentioned in Section 7.1. These counterparts enable the programmer to specify objects that should be created in a particular type of memory and other characteristics such as shared memory.(RTSJ.org, 2010)

The `RawMemoryAccess` and `RawMemoryFloatAccess` classes provide a convenient and secure way to access raw memory through the use of pointers. Among others, these allow programmers to access memory-mapped hardware thereby enabling that device drivers can be written, and flash memory can be programmed.

## 7.7 Time Values and Clocks

The RTSJ provides the concept of a clock, represented by the class `Clock`. An instance of `Clock`, or a subclass of it, is capable of returning the time of interest such as the current absolute time or execution time.

The specification defines that there should always be a default `Clock` instance available which should provide the current absolute calendar time. The clock should advance both monotonically and uniformly. This default `Clock` instance is available through the static method *`getRealtimeClock()`* of the class `Clock`.

### 7.7.1 High Resolution Time

In real-time systems it can be necessary to represent time with a high resolution. This is especially the case for real-time systems since these depend on many deadlines and requirements.(Wellings, 2004)

The RTSJ provides some additional facilities allowing the program to use *high resolution time*. In fact, the RTSJ supports representing time down to the resolution of one nanosecond. However, the necessary hardware needs to be available in order to support nanosecond resolution.

Note that the supported time is discrete, that is, the system cannot represent a time which is in-between two subsequent nanoseconds. This is in contrast to continuous time where a point in time will always be available between two arbitrary points.

In order to represent time, the abstract class `HighResolutionTime` is provided with two subclasses which represent two different aspects of time.

Absolute Time An instance of the class `AbsoluteTime` represents an absolute time in the system. It is internally represented as the time since some specific point in time, which by default is the Unix epoch (January 1, 1970, 00:00:00 GMT). Times before and after the given epoch is represented using a negative and positive value, respectively.

Relative Time Time which is relative to an absolute time, is represented by an instance of the class `RelativeTime`. These instances can be added or subtracted an absolute time, creating a new absolute time shifted by the relative value.

Listing 7.3 shows how clocks can be used to get the current time and how arithmetic can be used on times.

```
void foo() {
    AbsoluteTime startTime;
    AbsoluteTime stopTime;
    RelativeTime diff;
    Clock clock = Clock.getRealtimeClock();

    startTime = clock.getTime();

     ...

    endTime = clock.getTime();
    diff = endTime.subtract(startTime);
}
```

**Listing 7.3:** Example of the usage of clocks and subtraction of times.

Line 2-4 is used to define variables for `AbsoluteTime` and `RelativeTime` instances. In line 5 a reference to the default clock is obtained using the static method *`getRealtimeClock()`*, and is further used in line 7 to store the current time in the *`startTime`* instance. The dots in line 9 represent some unknown computations, in which the clock progresses. The new current time is then stored in *`endTime`* in line 11. Finally, the start time is subtracted from the stop time in line 12, resulting in a relative time representing the running time of the unknown computations.

## 7.8 Resource Sharing and Synchronisation

This section will assume that a priority-based scheduling policy is used in the system.

In traditional Java, communication and synchronisation is based on exclusive access to shared resources by an approximate implementation of the monitor construct. Specifically, Java contains *synchronized methods* and *synchronized blocks* which essentially implement the monitor by acquiring the object lock of either the current object, or in the case of synchronised blocks, the object lock of the object specified in the parameter. Conditional synchronisation is based on manually implementing wait-loops and corresponding notifications. Wait-loops are essentially loops in which threads wait for a certain condition to be satisfied by a call to *`wait()`* which effectively places the particular thread in the object's wait-set. Threads are manually awaken whenever a condition is changed by another thread by means of calls to either *`notify()`* or *`notifyAll()`*.(Gosling et al., 2005)

### 7.8.1 Priority Inversion

The mechanisms for synchronisation in the RTSJ have not been changed. However, all synchronisation forms that are based on mutual exclusion are vulnerable to priority inversion. The RTSJ specifically requires the following to be implemented in order to enforce priorities and avoid priority inversion:(RTSJ.org, 2010)

- All queues that are maintained by the underlying real-time JVM, such as the wait-set must be ordered according to priority. This policy is also applied for threads waiting for acquiring the object lock as a consequence of calling e.g. a synchronised method on an object for which another thread currently has the object lock of. Notice, that schedulable objects can have the same priority. In this relation, the RTSJ does not specify how to resolve the problem of schedulable objects having the same priority, but usally they are ordered as FIFO queues.

- The RTSJ allows the programmers to specify how priority inversion control algorithms are implemented. By default, the RTSJ specifies that the *priority inheritance protocol* must be implemented. The specification describes an implementation of the ICPP but does not require it.

### 7.8.2 Heap and No-Heap Resource Sharing

An important aspect of resource sharing is how to exchange or share resources among heap-based and no-heap-based schedulable objects. A set of requirements must be established to ensure that schedulable objects with no references to the heap are not indirectly delayed by the garbage collector as a result of the heap-based schedulable object working on the shared resource. These requirements are stated in the following:(Wellings, 2004)

- All the no-heap-based schedulable objects should always have greater priorities than the heap-based.
- All shared synchronised objects must be accessed according to the ICPP.
- All the shared synchronised objects should have all necessary memory preallocated as part of scoped or immortal memory.
- In relation to the previous requirement, it is also necessary that objects passed in any communication form should be based on the primitive types available in Java or be allocated in immortal or scoped memory.

# 8 Real-Time Java Profiles

Even though the RTSJ introduces real-time capabilities to Java, it is not without drawbacks. The RTSJ is broad and imposes few limitations on how the developer structures the application, such as by supporting multiple models of concurrency (Henties et al., 2009). Furthermore, the specification contains complex and computationally expensive features which complicate both its usage on resource constrained devices and subjectability to WCET analysis and schedulability analysis. However, the RTSJ is a good starting point for profiles targeting real-time embedded or safety-critical systems development because it, in contrast to traditional Java, contains sufficiently tight semantics for thread scheduling and synchronisation and because it allows removing the garbage collector from interfering (Kwon et al., 2002). Thus, in order to better support real-time embedded systems, a number of profiles exist that restricts the RTSJ and provides additional functionality.

The descriptions of the profiles ground the basis for the following implementations of a practical example in each profile. Through this, accompanied with the theoretical knowledge gained in this chapter, the profiles are evaluated upon in Chapter 13.

## 8.1 Ravenscar-Java

The RJ profile is an early example of a real-time Java profile. The profile defines an extended subset of RTSJ with the goal to decrease the number of complex and computationally expensive features such that it is better suited for small resource-constrained embedded systems. The following is based on the definition of the RJ profile provided by Kwon et al. (2002).

In general, the changes introduced to the RTSJ by RJ can be categorised into three areas of which the most notable changes are described below. The areas are: *predictability of timing*, *predictability of memory utilisation*, and *predictability of control and data flow.*

### 8.1.1 Predictability of Timing

In regard to timing, the different ways the RTSJ allows for handling periodic, aperiodic, and sporadic events have been reduced. First of all, only periodic and sporadic events are supported, since aperiodic events are problematic with regard to schedulability analysis. Furthermore, in the RTSJ, both threads and asynchronous event handling can be used to handle all types of events in the system. In RJ, this has been changed such that all periodic events are handled by periodic threads and all sporadic events are handled by AEHs. Specifically, this is done by instances of the class `PeriodicThread` and by instances of the class `SporadicEventHandler`, respectively. All other classes related to handling these events are prohibited.

In order to better support WCET analysis and schedulability analysis, all threads and AEHs must be defined in the initialisation phase, and their release parameters must not be changed during the execution phase.

The RTSJ defines FPS as the default scheduling policy, with the possibility of additional schedulers being provided by the JVM. This is restricted by RJ by defining that FPS is the only allowed scheduling policy. Furthermore, all APIs related to online feasibility checks and processing groups, which are encouraged by the RTSJ, have been prohibited in favour of offline schedulability analysis. This approach rectifies the problem of not knowing whether a correct feasibility test is implemented or not.

### 8.1.2 Predictability of Memory Utilisation

The RJ profile reduces the number of available memory areas to two: immortal memory, implemented by `ImmortalMemory`, and linear time scoped memory, implemented by `LTMemory`. A number of rules have been defined in regard to the usage of these with the goal of easing WCET analysis and schedulability analysis. E.g. it is required that all memory areas must be defined in the initialisation phase, scoped memory is not nested or shared between schedulable objects, and the size of scoped memory is not changed in the execution phase. Furthermore, all usage of heap memory and garbage collection is prohibited.

### 8.1.3 Predictability of Control and Data Flow

Some restrictions have been applied to the control flow of a program with the purpose of simplifying the implementation and easing the use of program analysis tools. One of the most notable changes is the prohibited use of ATC which both increases the complexity in the implementation and increases the possible control flows of a program substantially.

Other changes include the requirement of all constraints, such as those used in a loop, must be static. This eases the task of timing analyses. Finally, RJ prohibits the use of *continue* and *break* statements, due to these increasing the complexity in analysing the control flow.

## 8.2 Safety Critical Java

This section is primarily based on the works of Henties et al. (2009). The SCJ profile was primarily conceived with the purpose of sustaining and standardise safety-critical systems development in Java. The specification is still a draft and there is an ongoing effort to complete it (JSR302, 2010).

Safety-critical applications require that a rigorous validation and certification process is conducted as dictated by legal statute for instance. The SCJ profile is thus designed to include a safety-critical Java infrastructure and safety-critical libraries that are amenable to be certified under safe-critical standards.

The starting point for developing the SCJ is the RTSJ which is considered too broad. This is an undesirable property in terms of supporting safety-critical systems development due to the rigorous certification requirements that are mainly targeted at simpler programming models. Furthermore, the SCJ profile is also inspired by the RJ profile. Specifically, the SCJ level 1 which will described later in this section, draws inspiration from the RJ profile.

Due to the limitations and restrictions of the SCJ compared to the RTSJ, the SCJ takes the approach of using specialisation of the RTSJ by inheritance which is in contrast to the PJ profile described later in Section 8.3. This means that functionality from the RTSJ that is allowed in the SCJ must explicitly be annotated. Among others, the scoped memory model of the RTSJ is used to allow static analysis of the memory usage and the thread model is largely restricted to include periodic and asynchronous event handlers. Also, the SCJ defines a more simple programming model than the RTSJ, by defining concepts such as *missions*, *limited start-up procedures*, and *levels of compliance*. Also, the SCJ introduces special annotations of code to allow easier static analyses to verify that certain safety-critical properties are upheld by the application. In the following, the mission concept and compliance levels are introduced:

### 8.2.1 Missions

Applications that are SCJ compliant will consist of one or more missions. The mission concept defines a known set of periodic and aperiodic event handlers and might also contain the `NoHeapRealtimeThread` threads from RTSJ. In Section 2.2, the mission concept was described. Hence, this section will only describe the mission concept in relation to the SCJ.

During a mission, objects cannot be allocated in immortal memory or in mission memory. Instead, a dedicated block of memory is provided to the particular mission and objects created in this persist until the mission terminates.

The mission will start in initialisation mode during which all the necessary memory areas for the mission mode are allocated. Afterwards, the mission enters mission mode in which allocated mutable objects can be modified as part of the logic. All application processing is done in one or more schedulable objects.

The mission framework provides a means for orderly termination of schedulable objects. When termination is requested, the objects in the mission are notified to stop operating and once they have, the mission can safely stop and run possible clean-up code before terminating. A clean restart of the mission or a transition to another one is hence simple.

### 8.2.2 Compliance Levels

Safety-critical applications targets a wide variety of applications, from complex multi-threaded to single-thread applications. The certification process is highly dependent on the complexity. Hence, it is desirable to restrict the complexity of a program in order to enable an easier certification process of these. Due to these reasons, the SCJ defines three compliance levels which are described below:

Level 0 Applications written for this compliance level are based on a programming model similar to CES. Hence, the mission can be thought of as a set of computations that are executed periodically in a precise clock-driven timeline. Fundamentally, the schedulable objects used in applications of this compliance level are solely periodic event handlers having a period, priority, and start time relative to the beginning of a major cycle. The schedule is either constructed by the application programmer or by an offline tool. It is worth noting that all periodic event handlers execute under control of a single thread. Hence, there are no synchronisation concerns since each periodic event handler is processed sequentially.

Level 1 Applications written for this compliance level consist of a single mission with a set of concurrent computations. Each of the concurrent computations have a defined priority that is used in relation to the FPS scheduler used under this compliance level. The computations can either be performed in a mixture of periodic event handlers or aperiodic event handlers and the integrity of shared objects is preserved by means of synchronisation mechanisms.

Level 2 Applications written for this compliance level start with a single mission where they optionally can create more missions and execute them concurrently. In addition to level 1, this level allows the use of `NoHeapRealtimeThread`. Also, the Java methods ***`notify()`*** and *`wait()`* of the `Object` class may be used.

As a consequence of the focus of making certifications possible without too much effort, SCJ does not allow sporadic tasks to be used. The reason for this is that it requires implementing a monitor mechanism to ensure that the minimal inter-arrival times of sporadic tasks are upheld. According to JSR302 (2010) this monitor mechanism introduces significant complexity to the real-time system in terms of certification according to standards.

## 8.3 Predictable Java

As a concurrent process of standardising SCJ, another profile called PJ is being formulated. Its purpose is not to compete with the standardisation but instead to contribute to the main ideas of SCJ. As the authors of PJ writes: "*...if the main ideas are taken up by SCJ, PJ has served its purpose.*". The profile proposes two main differences, namely:(Bøgholm et al., 2009)

- PJ is a generalisation of RTSJ.
- Missions are first class event handlers.

### 8.3.1 Generalisation of Real-Time Specification for Java

RTSJ offers much functionality along with detailed theory, whereas, as previously described, what is needed for embedded safety-critical systems is an abstraction of it. This abstraction can be made in two ways: by letting the profile inherit for limiting the RTSJ or by letting RTSJ be a specialisation of the smaller profile. Not taking into account that RTSJ already exists and thereby not basing the decision on political reasons, the latter approach is suggested in PJ. This is in contrast to RJ and SCJ, that are described as being political regarding the existence of RTSJ. The authors of PJ argue that inheriting for specialisation, is similar to the analogy of deriving rational numbers from natural numbers and not the other way around (Bøgholm et al., 2009). However, this gives rise to problems since the RTSJ cannot be defined as subclasses of it. Therefore, it is suggested that an adapter layer disallowing some RTSJ methods and giving default values to some parameters, is used in an implementation of PJ.(Bøgholm et al., 2009)

### 8.3.2 Missions

Missions in PJ are similar to those known from SCJ in the sense that a mission generally has three phases. Also, missions are a set of tasks, event handlers, collaborating on providing a desired functionality. From this, it can be inferred, that missions are simple containers of tasks. However, as mentioned, they also constitute a set of phases, and hence more logic is needed. Therefore, it is suggested in PJ to let missions be handlers themselves. This way, the event handler constructor becomes the initialisation phase and the ***handleEvent()*** method becomes the termination phase. Listing 8.1 depicts the `Mission` class of PJ.(Bøgholm et al., 2009)

In the current state of PJ, the ***addToMission()*** method allows adding periodic, aperiodic, and mission event handlers to the list of event handlers associated with the given mission. Besides, this allows the programmer to nest missions such that they e.g. can be run sequentially.

```
public class Mission extends AperiodicEventHandler {
   Vector<ManagedEventHandler> eventHandlers;

   protected Mission(PriorityParameters priority,
                AperiodicParameters ap,
                Scheduler scheduler,
                MemoryArea memory) {
      super(priority, ap, scheduler, memory);
      eventHandlers = new Vector<ManagedEventHandler>();
   }
```

```

    public void addToMission(ManagedEventHandler eh){
        eventHandlers.add(eh);
    }

    public void handleEvent() {
        // the logic to be executed on termination
    }
}
```

**Listing 8.1:** An excerpt of the `Mission` class of PJ.

As shown in Listing 8.1, `Mission` is actually an aperiodic event handler which uses scoped memory. As a consequence, immortal memory is not necessary since objects that have to live throughout the lifetime of a mission can be declared in its constructor and thereby, exist until the mission is terminated. This means that the number of concepts used in
acPJ is reduced thereby being more concise without loss of functionality.

Note that the PJ further supports SCJ's decision of omitting sporadic tasks by questioning the tasks' statical checkability. Especially, external events are described as being a problem, since their occurrence is based on assumptions of the environment. In many cases, this cannot be completely assured since their arrival cannot be assured.(Bøgholm et al., 2009)

# Part III

# Mine Pump

# 9 Practical Example

In the following, a practical example of an embedded hard real-time system implemented in Java is presented. The purposes of this exercise are listed below and are all in accordance with our learning goals.

- To gain experience in developing real-time systems.
- To illustrate the problem of implementing real-time systems in Java.
- To illustrate the problem of WCET analysis and schedulability analyses.
- To evaluate the different real-time Java profiles according to criteria.

We do not use a real-time oriented development method for the implementation of the example. The primary reason is that using a new development method is not the focus of this project. Learning how to use a new development method entails a substantial effort in itself.

As a further limitation, the development process excludes ensuring functional correctness of the application. Only printouts have been used, where necessary, to provide subtle confidence that the application behaves as expected. In an implementation of an actual real-time system that is to be deployed in a production environment, ensuring functional correctness is an integral part that can never be omitted.

## 9.1 Mine Pump

A classic example of a hard real-time system, used in the literature, is the mine pump which is depicted in Figure 9.1 (Burns and Wellings, 2009; Liu and Joseph, 2005). The system's primary function is mine drainage where a pump is responsible for draining a shaft for water. This way, the mine will not get flooded, thereby not risking the lives of the miners.

The system consists of six different sensors monitoring the state of the mine, a number of alarms, and a submersible water pump. The functional requirements are:

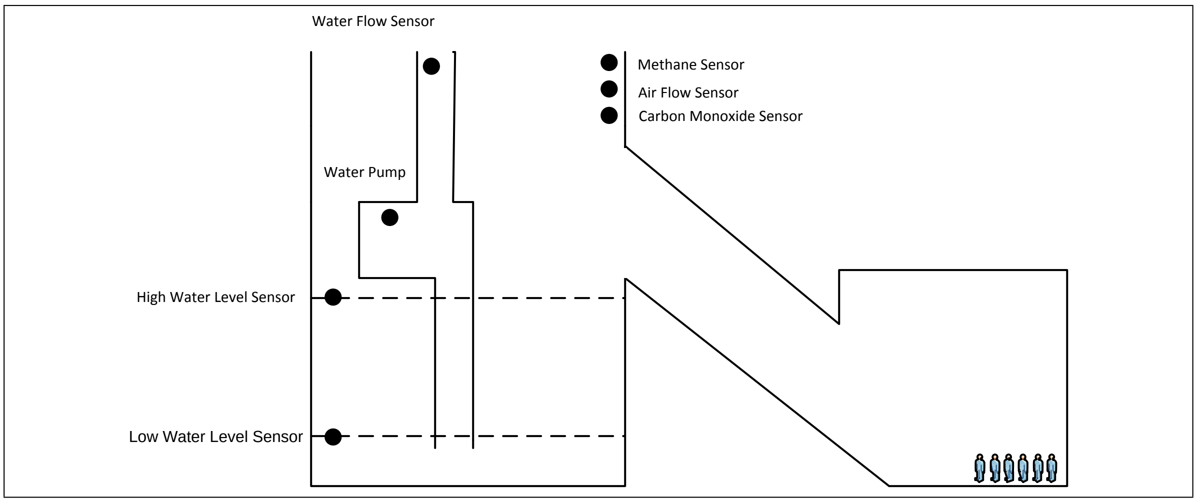


**Figure 9.1:** Illustration of the mine pump example.

Pump Operation The water pump must not be operating when the water level is below the low water level and it must be operating when the water level is high. To determine the two water levels sensors are used.

Operator Interaction Two types of operators can control and monitor the pump from the surface. Normal operators can manually start and stop the pump while the water level is between the high and low water levels. Supervisor operators ignore this limitation and can start and stop the pump regardless of the water level.

Environment Monitoring The mine is also equipped with carbon monoxide, methane, air-flow, and water-flow sensors.

To allow a safe work environment for the mine workers it is important that the concentrations of carbon monoxide and methane do not exceed a certain critical level. If they do, an alarm must be raised to initiate an evacuation process and the operator is informed about the incident. A similar alarm is raised if the air-flow sensor or the water pump are malfunctioning. A malfunctioning water pump can be detected using a water-flow sensor. Furthermore, operating the water pump under high methane levels poses a risk of explosion. Thus, the water pump is not permitted to run in this case, and must be stopped as soon as the methane level rises above critical.

System Monitoring All events in the system, such as alarms, sensor readings, and manual operations, are logged for later review.

Table 9.1 shows the timing requirements and have been extracted from Wellings (2004) and Liu and Joseph (2005). Notice, that the sources of the mine pump do not specify how the timing requirements are to be upheld, that is, they do not specify the particular tasks that are responsible for handling certain events in the mine pump. They only specify the periods and deadlines of the events occurring in the system.

| Event | Timing Requirement |
|---|---|
| Evacuation of mine | 1 hour |
| Notification of critical methane levels and carbon monoxide levels | 1s |
| Notification of critical air-flow | 2s |
| Notification of failure of water pump | 3s |
| Starting or stopping the water pump according to water levels | 200ms |
| Stopping the water pump when methane level is critical | 200ms |

**Table 9.1:** List of timing requirements for the operation of the mine pump.

### 9.1.1 Constrained Mine Pump Example

The main purpose of the practical example implementation is to gain practical knowledge in real-time systems. Therefore, some details of the original mine pump example are not needed to achieve this goal and would only add complexity to the construction and size of the logic. Therefore, we have omitted details to the extent that the omission does not reduce the knowledge obtained. As a consequence, the example is constrained to focus on the operation of the water pump and safety measures introduced by the methane sensor. In particular, to reduce the size and complexity of the mine pump, it is constrained in the three following ways:

- The operator roles of manually controlling the mine pump are left out.
- The flow of water being drained from the shaft is not monitored.
- The carbon and airflow sensors are not incorporated.
- The evacuation process, including alarms, is not taken into account.

### 9.1.2 Requirements

The following describes the specific requirements for the constrained mine pump example. The requirements are elicited from the description of the example.

1. The water pump must only be operating when there is no critical level of methane in the mine. This requirement is therefore a precondition that must be upheld in the remaining requirements.
2. When the water level is high, the water pump must be started.
3. When the water level is low, the water pump must be stopped. According to our interpretation, the shaft must not completely be emptied for water.
4. If there is no methane in the mine, the mine must never be flooded.

## 9.2 Choice of Hardware

LEGO and various LEGO RCX (Group, 2010) sensors and actuators are used to construct the example of the mine pump. This allows for both quick and flexible construction of the example.

A number of possible choices are available in regard to hardware controlling the sensors and actuators. Some are to use the LEGO RCX programmable brick, the JOP board, the aJ-100 board or an ARM based board. The LEGO brick, and the ARM based boards are not considered as candidates for usage. This is due to the need of timing predictable execution of Java code which may involve a significant effort or even turn out impossible to realise using these platforms. This means that a special-purpose Java processor is needed. This is one of the drawbacks of using Java for real-time systems development. The problem is that the Java processors are not as widespread as other processors. To allow Java to be executed in a timing predictable manner on processors such as ARM, it requires that the JVM combined with the RTOS facilitate timing predictability. As of this writing, this has not yet shown to be possible, and it is our opinion that a solution to the problem must be found in order to make Java attractive as a programming language for real-time systems development. As previously described, aJ-100 and JOP are hence possible choices. However, execution times of each instruction on the aJ-100 are not published, thereby excluding this platform as well. Therefore, the JOP is used.

The JOP supports RT-Java which is a real-time profile for Java specifically designed for the JOP. The JOP contains partially implemented versions of the SCJ and RJ. Therefore, these implementations do not follow their respective specifications completely. Furthermore, the newest SCJ version is not implemented. The RJ profile is almost fully implemented. However various elements such as the possibility of deadlines being different from periods is left out. This discovery was made during an examination of the implementation files of the RJ profile on the JOP. With regard to PJ, there does not exist an implementation for the JOP, as of this writing.

In respect to interacting with LEGO RCX sensors and actuators, the JOP project makes available a LEGO library that uses native calls for controlling them. An example of how the library conducts a sensor reading consider the code is shown in Listing 9.1.

```java
public static int readSensor(int index){
    return (Native.rd(IO_SENSORS) >> OFFSET_SENSOR[index]) & MASK_SENSOR;
}
```

**Listing 9.1:** Code example of a native call interacting with a LEGO sensor.

As shown, a native call is made for reading from a particular place in memory. The ***IO_SENSORS*** argument represents the beginning memory area where the I/O sensors are located. Offsetting this by the sensor id and masking the result, results in the exact memory area related to the specific sensor. The masking is needed because the ***Native.rd()*** method returns a 32-bit integer, whereas the I/O sensors values are located with 9-bit intervals.

## 9.3 Integrating the Mine Pump with LEGO and JOP

The LEGO construction of the mine pump is depicted in Figure 9.2.

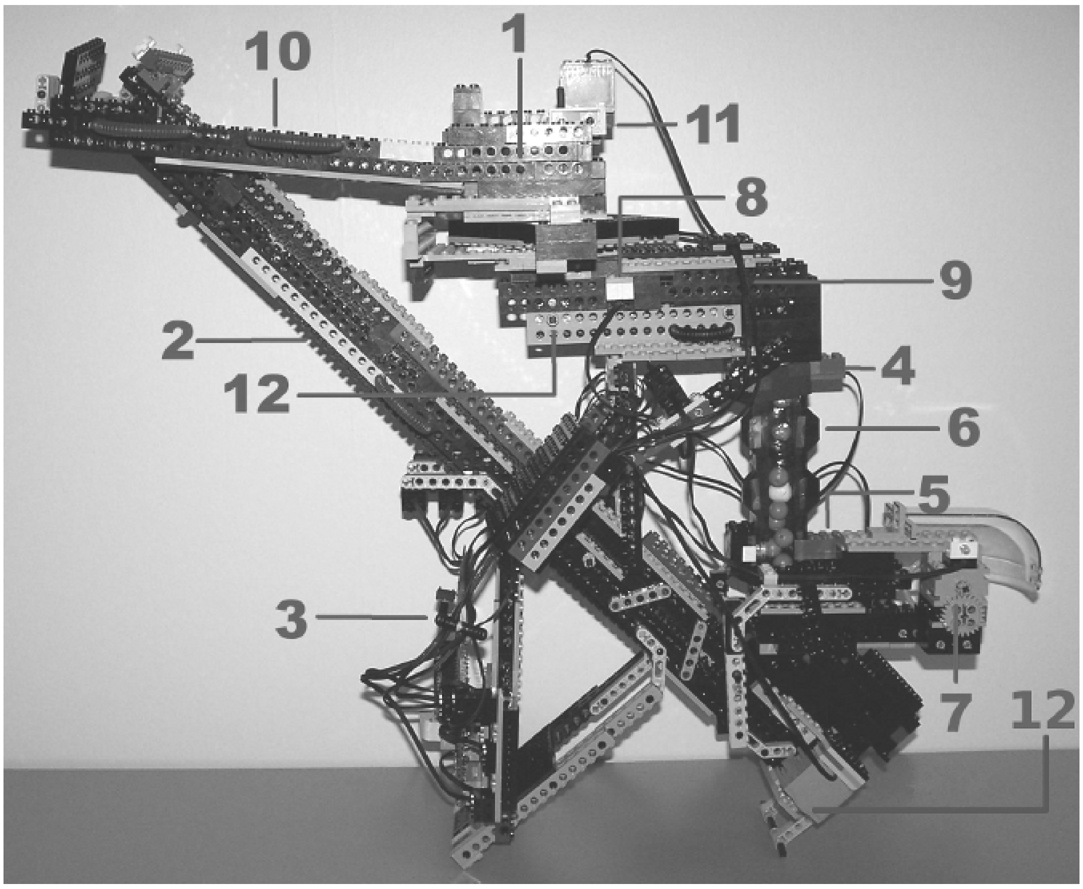


**Figure 9.2:** Picture of the LEGO construction simulating the behaviour of the mine pump example. The numbers on the picture represent: 1) Feeder. 2) Angled conveyor belt. 3) JOP board. 4) High water level sensor. 5) Low water level sensor. 6) Mine shaft. 7) Water pump actuator. 8) Methane sensor. 9) Conveyor belt from feeder to mine shaft. 10) Slideway from conveyor belt to feeder. 11) Feeder actuator. 12) Conveyor belt actuator. 13) Methane alarm.

Since methane and water are difficult to handle in a LEGO construction, they are represented by two differently coloured bricks. By colouring the bricks differently, different light intensity values are translated into a specific type of brick. Since the measured values fluctuate for the same colour, it is necessary to define light intensity ranges for each coloured brick. Determining these, requires a calibration process to be conducted prior to the actual deployment because the light intensity depends on the environment in which it is situated.

Initially, the decision was made to construct the mine pump to run in a continuous loop, that is, we should not continuously add methane and water to the system under operation. This is primarily due to easing testing the construction. Specifically, bricks which have successfully been removed by the pump, are added to the cycle again by adding them to the *feeder*, labelled 1 in Figure 9.2. The feeder is responsible for providing bricks to the system in small time intervals thereby prohibiting bricks from flooding the system immediately. From the feeder, the bricks enter a conveyor belt, labelled 9, which leads to the *mine shaft*, labelled 6. A light sensor is placed on the conveyor belt to determine whether the particular brick represents either methane or water, see label 8.

The mine shaft is represented by a hollow construction into which two light sensors are placed at the bottom and top, respectively. These serve the purpose of representing the high, labelled 4, and low, labelled 5, water level sensors. In addition, small light bulbs are placed in front of the sensors to make the light intensity more consistent such that variations in ambient light do not significantly distort sensor readings.

When the light sensor placed at the top determines that the water level is high, the *water pump*, labelled 7, is started at the bottom of the shaft to remove one brick at a time. In contrast, if the light sensor placed at the bottom of the shaft determines that the water level is low, the pump is immediately stopped. Bricks that are removed from the shaft by the pump, enter an angled conveyor belt, labelled 2, moving the bricks back to the top of the feeder by using the slideway, labelled 10, and the cycle starts over again.

All actuators, sensors, and light bulbs are based on LEGO RCX. Evidently, the light sensors and the pump actuator need to be connected to the JOP, labelled 3, in order to be controlled by the real-time application software. To make the design more consistent, the remaining light bulbs and conveyor belt actuators have been connected to the JOP as well to provide the means as power source.

## 9.4 Tasks and Timing Requirements

As suggested by Wellings (2004), the water pump can be controlled using three sporadic tasks, which are released when the water level is low or high, or if a critical methane level is detected. Optimally, the solution would be to release sporadic tasks based on hardware interrupts from the sensors. However, this would require the sensors to support interrupts being fired at certain values indicating that there is high or low water level for the respective sensors. This is not supported by the LEGO sensors and polling the values is therefore necessary. The water level sensor readings are therefore conducted periodically, and if critical water levels are detected, sporadic tasks are released corresponding to whether the pump should be started or stopped. The methane sensor is similarly periodically polled in order to determine the current methane level, however, subsequent actions are handled by the periodic task itself. Therefore, in total, four tasks are used to implement the mine pump. Table 9.2 shows the timing requirements and priorities of these.

The priorities are deduced using the rate-monotonic approach. The reason for using priorities in the first place, is that RJ only allows FPS as scheduling policy. This means that deadline- or rate-monotonic priority assignment can be used. However, the JOP implementation of RJ does not allow deadlines to be different from periods. Hence deadline-monotonic is not advantageous in this regard because they effectively yield the same result.

The argumentation for the individual timing requirements can be found in Appendix G, but here it suffices to say that they are selected in order to withstand the requirement of reacting to critical methane levels and changes in water levels within $200ms$. The deadlines are furthermore affected by the physical implementation of the mine pump, capabilities of the sensors, and the platform only supporting deadlines being equal to periods.

| Sensor | Event Type | Period/Deadline (ms) | Period/Deadline (Clock cycles) | Priority |
|---|---|---|---|---|
| Methane Level Check | Periodic | $56ms$ | 5,600,000 | 1 |
| High Water Level | Sporadic | $40ms$ | 4,000,000 | 2 |
| Low Water level | Sporadic | $40ms$ | 4,000,000 | 3 |
| Critical Water Level Check | Periodic | $40ms$ | 4,000,000 | 4 |

**Table 9.2:** Timing requirements for the periodic and sporadic tasks of the LEGO mine pump example. Additionally, the priority, where four is the highest priority, of each task is shown.

# 10 Tools

As part of the project, three different analyses are made of the mine pump implementation: modelling the mine pump example, WCET analysis, and schedulability analysis. Besides conducting the WCET analysis and the schedulability analysis, which as described, are essential analyses in respect to real-time systems, we have chosen to model the mine pump example. The reason for this is that we previously have experienced that modelling a problem helps in understanding it by clarifying potential misunderstandings of the specification (Sw801b, 2009). In respect to the mine pump example, we have assessed that the specification is complex and that the modelling therefore can help concretising it.

Conducting the modelling and WCET analysis manually would be a very time consuming and possibly error prone task, motivating the use of a tool to conduct these. However, conducting schedulability analysis is a matter of applying known information of the tasks and use relative simple calculations. Therefore, we have decided to conduct this manually.

As a general note, using tools poses the problem whether the conclusions they draw can be taken for granted. That is, one must rely on a correct implementation of the tools and that they uphold their respective specification. We rely on the assumption that they exhibit functional correctness but this property has neither been proven nor disproven for the tools. As an example, consider the problems arising when using a model-checker. The problems constitute the mapping between the verified properties of the model, the properties of the specification, and the properties of the actual implementation. That is, by verifying that the model is deadlock-free does not necessarily imply that the implementation is deadlock-free as well. A concern related to this topic is that of the reliability of the conclusions drawn by the model-checker.

## 10.1 Tool Needs

The following list introduces the chosen tool for each of the analyses. Alternative tools are also presented, however, not with an extensive description because this is not the focus of this project.

Modelling the Mine Pump Through an analysis, we know that the UPPAAL (Bengtsson et al., 1996) model-checker tool is sufficient for the needs required in this project (Sw801b, 2009). We are aware of alternatives such as TAPAs (Calzolai et al., 2008) and PRISM (Hinton et al., 2006) but we have assessed that they do not prove beneficial in respect to using UPPAAL for modelling the mine pump example.

WCET Analysis There exist a couple of WCET analysis tools supporting the JOP, namely WCET Analyzer (WCA) (Schoeberl et al., 2010) and Volta (Volta, 2009). Both tools analyse the Java bytecode in order to generate WCET estimates for parts of an application. However, Volta has previously been shown to be unable to analyse functions involving native calls to e.g. sensors (Bøgholm et al., 2008c). Whether this problem has been corrected in later releases is unknown, since no release notes or changelogs have been published. We have therefore chosen to use the WCA tool.

Schedulability Analysis For schedulability analysis, two different tools candidate: SARTS and TIMES exist. Both use the model-based schedulability analysis technique.(Bøgholm et al., 2008a; Amnell et al., 2004)

The advantage of using SARTS is that it is directly based on the Java program source code. Hence, there is no intermediate phase relying on the programmer to provide the correct program details. The tool translates Java bytecode into timed automata which are subject for model-based schedulability analysis by UPPAAL afterwards. Given the execution time for each bytecode instruction, the timed automata can be run along with a model of the scheduler and if the program is deadlock-free it is schedulable. The disadvantage of SARTS, in respect to this project, is that it only supports early versions of the SCJ profile. Since we do not use this version of SCJ, SARTS is not applicable in our project. Instead we use TIMES.(Amnell et al., 2004; DARTS, 2007)

## 10.2 Modelling the Mine Pump Using UPPAAL

This description of UPPAAL is mainly based on Behrmann et al. (2004) and UPPAAL (2010).

UPPAAL models a given system using timed automata for which properties, such as deadlock-freedom, can be verified. The tool consists of a model-checker back-end and a GUI front-end. The back-end can be used independently of the front-end, hence giving the opportunity of executing the verification process on an application server, while using a normal desktop computer to design the model using the GUI. This can be convenient due to the verification process potentially being time consuming.

A UPPAAL model consists of a number of timed automata that each represents a specific part of the modelled system. These are denoted templates, from which a number of processes can be instantiated. A template consists of locations connected by edges and a number of attributes. Furthermore, a template has a corresponding code file, in which C-like functions and local variables can be specified.

### 10.2.1 Annotations

Figure 10.1 depicts two UPPAAL templates that, when instantiated, synchronise with each other in periods of 100 time units. The templates are used as reference throughout the description of UPPAAL.

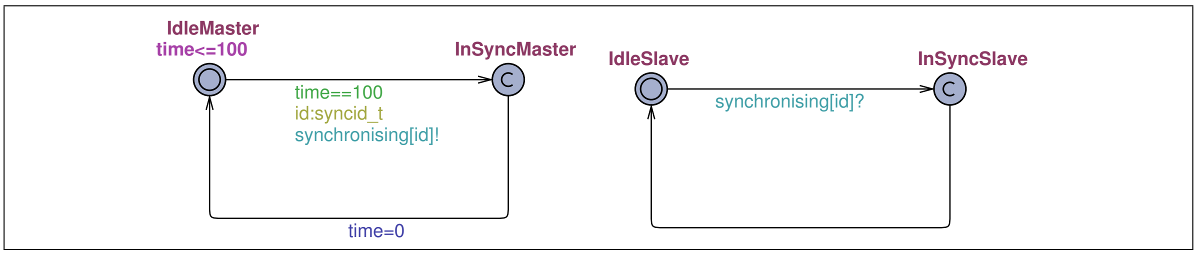


**Figure 10.1:** Example of two UPPAAL templates.

The following describes the annotations a location can have:

Name Locations can have an associated name that can be used as an identifier for the given location. The names can be used in the verification process to refer to the location when checking a given property.

Initial State Exactly one location in each template must be marked as initial. The location is marked by a double circle representing the starting point of the process. In the example, the *IdleMaster* and *IdleSlave* locations are initial locations.

Invariant An invariant is used as condition on a location that must be upheld as long as the system is in the given location. Consider *IdleMaster* which is annotated with the $time <= 100$ invariant. This means that the location only can be visited as long as the clock variable, *time*, is below or equal to 100.

Committed Marking a location as committed, represented by $C$, means that the time is frozen when entering the location and an outgoing edge must be taken in the next transition. This means that atomic operations can be modelled. The *InSyncMaster* and *InSyncSlave* locations are committed locations.

Urgent Urgent locations are less restrictive than committed locations. Marking a location urgent, represented by $U$, is similar to add an extra clock $x$ that is set to zero on all its incoming edges and having an invariant $x <= 0$ on the location. This means that the time is frozen in the location, but it does not force the next transition to be one of its outgoing edges.

The annotations that can be used for the edges are described in the following:

Selections These are used to non-deterministically select a value from a type and assign this to a variable. A type could e.g. be a scalar set or a bounded integer. In the example shown in Figure 10.1, a selection is made on the outgoing edge of *IdleMaster*. This selects an *id* of a process that is being synchronised. If there are e.g. two instances of the SLAVE template, the select statement chooses which of these to synchronise with.

Synchronisation Two processes can synchronise over channels. As an example, defining a channel called *synchronising*, issuing *synchronising!* represents a signal sent through the channel and *synchronising?* represents receiving the signal by another process. When the synchronisation is made, the involved processes change location at the same time. If two processes are synchronised, this is called binary synchronisation. Another variant is to make broadcast synchronisations which means that zero or multiple receivers synchronise with a single sender. Finally, synchronisation can also be made urgent, thereby enforcing synchronisations never to be delayed if the synchronisation is possible.

Guards A guard is a condition that must be upheld when firing an edge. If it is not, the given edge cannot be fired. The outgoing edge of *IdleMaster*, in the example, has a guard dictating that $time == 100$ must be true before the *InSyncMaster* can be reached. This, combined with the invariant on *IdleMaster*, ensure that the synchronisation is made in periods of 100 time units.

Updates When an edge is fired, its update expressions are evaluated. This can be used to make variable assignments. E.g. in the example, *time* is updated when firing the outgoing edge of *InSyncMaster*. Also, it is possible to make calls to C-like functions declared in the underlying code file.

### 10.2.2 Query Language

As previously described, a purpose of modelling a system is that properties, reflecting its requirement specification, can be verified. To do this, UPPAAL requires the specification to be expressed in a query language consisting of path formulae and state formulae. As the names imply, the former expresses paths of the model and the latter expresses the states of the model.

In UPPAAL, the state formulae are expressions that can be evaluated similarly as a boolean expression like $var == 5$. Also, state formulae can be used to test whether a process is in a specific location. This is expressed as $Proc.loc$ where $Proc$ is the name of the process and $loc$ is the name of the location of interest. Finally, a special state formulae, denoted *deadlock* is made available in UPPAAL. The *deadlock* state formula evaluates to true if the state is a deadlock.

The path formulae are classified into three classes, namely: *reachability*, *safety*, and *liveness*. Each of these are described in the following. Figure 10.2 can be used as reference when reading the descriptions.

Reachability This class of path formulae asks whether a given state formula, $\phi$, eventually can be satisfied by a reachable location. In UPPAAL this is written as $E\Diamond\phi$.

Safety This class is used to describe that some state formula will possibly or never occur. It is e.g. convenient to verify that a deadlock never occurs in a model. Generally, there are two path formulae that can be used on a state formula, $\phi$, namely $A\Box\phi$ and $E\Box\phi$. $A\Box\phi$ says that in all reachable states, $\phi$ must be true. $E\Box\phi$ says that there should exist a maximal path where the state formula, $\phi$, is true in all states.

Liveness Path formulae of this class describe that something eventually happens. In UPPAAL, $A\Diamond\phi$ describes that the state formula, $\phi$, is eventually satisfied on every path. Another convenient formula is $\phi \leadsto \psi$ meaning that whenever $\phi$ is satisfied, eventually $\psi$ is also satisfied.

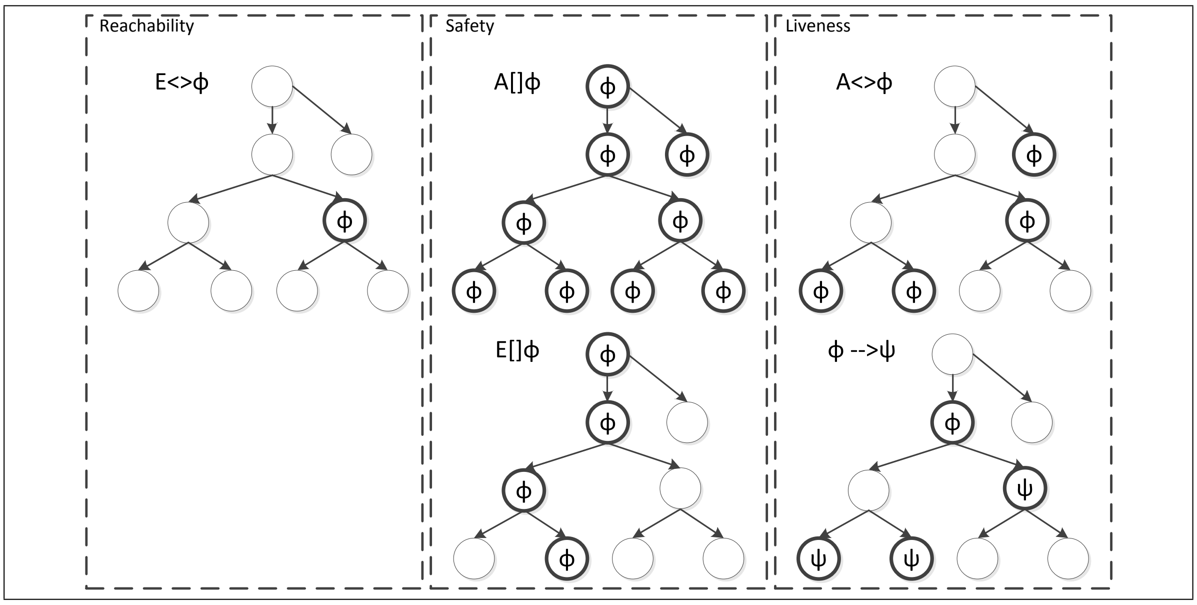


**Figure 10.2:** The supported path formulae in UPPAAL. The marked states are those where the state formula $\phi$ holds. The figure is inspired by Behrmann et al. (2004).

## 10.3 Worst Case Execution Time Analysis Using WCA

For WCET analysis, one needs to have some means of determining the WCET of a series of instructions. In respect to the JOP this is can be done by conducting its documentation where the execution times for each instruction is specified.

The primary source of the following description of WCA is based on Huber and Schoeberl (2009) and Schoeberl et al. (2010).

WCA is static program analysis tool that implements both IPET-based and model-based WCET analysis. In respect to the JOP the IPET-based analysis incorporates a static analysis of the method cache, while the model-based analysis, using the UPPAAL model-checker, exactly models the cache behaviour, potentially resulting in more precise WCET estimates. The two approaches to model the cache are as follows:

**IPET-Based Cache Model** This approach models the method cache behaviour by determining if an invoked method, and all methods invoked by it, can be stored in the cache. If this is the case, the cache is used. Otherwise, it is assumed that all cache accesses are missed.

**Model-Based Cache Model** This approach models the method cache behaviour precisely by modelling the contents of the cache and thus models cache hits and misses instead of the rather coarse-grained approach of determining cache hits and misses on method invocation level as in the IPET-based approach.

### 10.3.1 Data Flow Analysis

The tool optionally uses DFA in order to extract loop bounds and perform receiver-type analysis to make the WCET more precise on dynamic method dispatch. Specifically, receiver-type analysis is used to compute which types an object may actually have, thereby reducing the WCET when analysing virtual method calls. By using DFA to determine loop-bounds, some programmer annotations are avoided which could have introduced uncertainties due to the programmer being responsible for making these manually. However, when loop bounds cannot statically be resolved by these means, programmer annotations must be provided when e.g. the loop-bound is dependent on program arguments or I/O.

We were not able to get WCA to perform DFA for determining loop bounds. Even on simple loops, similar to those used by the JOP project to test the implemented DFA, the bounds could not be determined without programmer annotations. A further observation, was that DFA is not capable of determining loop bounds in the JOP's implementation of the JVM. By examining the source code of the JOP project, it is evident that the DFA apparently is implemented. However, due the complexity of the WCA tool implementation we assessed the effort required to identify the problem, compared to simply applying annotations, would not account for the possible gains. Note that not providing the annotations results in significantly over-estimating the WCETs because WCA uses a default loop bound of 1024 which in many cases is several orders of magnitudes higher than the actual value, and in some cases even too little, thus this results in unsafe estimates.

### 10.3.2 Worst Case Execution Time

WCA determines the WCET by accessing the class files using the Byte Code Engineering Library (BCEL). This library enables manipulation of the class files and the bytecodes of the methods. Based on this, WCA extracts the basic blocks making up the program and constructs the CFG. The CFG is then analysed to extract information about loop bounds and loop headers. Should programmer annotations be present at the loops, these are extracted as well for subsequent usage. If IPET is used as the WCET analysis approach, the WCA tool transforms the CFG to an ILP problem for which the ILP solver *lp_solver* (Gourvest et al., 2010) is used to determine the WCET.

If the model-based approach is used, WCA automatically constructs a UPPAAL model with a few extra states to denote the start and end of the task and a state denoting that the execution of the task has finished. The WCET analysis of a series of instructions is conducted by guessing a WCET and verifying if the instructions can be executed within this time. By gradually reducing the WCET guess, it is possible to converge towards a final WCET estimate. Since, establishing an appropriate guess can be difficult, the initial WCET is determined by using an IPET-based analysis.

In the development version of UPPAAL, specifically 4.1.1 and later, a *sup* query has been added to the query language. This allows for determining the maximum observed value of an integer or a clock variable. The feature greatly simplifies the above-mentioned procedure for estimating the WCET, since the model only needs to be verified once to extract the WCET and not multiple times to converge towards it. We have experimented with this new query by extracting the clock variables at the worst-case execution paths of the UPPAAL models generated by WCA. This resulted in the same WCET estimates as WCA while reducing the number of verification runs to one. The new verification query is written as in Listing 10.1.

As a consequence of using this rather new query, the model-based approach becomes less time consuming since only one verification run is needed because a binary search to converge the WCET to a safe and more precise estimate would otherwise be needed. This means that the difference in WCET determination times of the model-based approach and the IPET approach is reduced. However it should be remarked that, as documented by Huber and Schoeberl (2009), even one verification run in the model-based approach still is more time consuming than using the IPET approach.

```
sup{M0.E}:t
```

**Listing 10.1:** Sup verification query used for detecting the maximal value of the clock variable $t$ at location M0.E.

In WCA, using the IPET-based approach is usually faster than using the model-based approach, while the model-based approach provides more precise bounds of the WCET since it models the cache behaviour of the JOP precisely.

When WCA has successfully created a CFG and conducted the WCET analysis, a detailed report about the timings of the various methods including their cache misses, WCET of individual source code lines, execution times of methods and basic block constituents are formatted as HTML pages which provide a good overview.

## 10.4 Schedulability Analysis Using TIMES

The following is primarily based on Amnell et al. (2004) and DARTS (2007).

TIMES is a tool for model-based schedulability analysis. The principle of the analysis is to transform all provided information to timed automata on which UPPAAL is used to check whether a location where a task misses its deadline can be reached. The provided information consist of tasks, shared resources including when they are locked and unlocked, WCETs of constituting tasks, and scheduling policy. TIMES supports the rate-monotonic, deadline-monotonic, EDF, FIFO, and user defined priorities for the scheduling policies.

The declared tasks can be specified to be sporadic, periodic, or controlled. The release of sporadic and periodic tasks is handled by TIMES itself, while the release of controlled tasks are modelled by timed automata. This allows for tasks having releases dependent on specific states in the application, thereby increasing the accuracy of the schedulability analysis.

TIMES includes the possibility of code generation from models. However, this is omitted since the provided code generation is targeted at the Hitachi H8 processor of the LEGO Mindstorms RCX brick. If however, the code generation was targeting Java and specifically the used real-time profiles, it would be interesting to use. The advantage of doing this is that the coupling between the code and the model becomes tighter. This means that some of the concerns regarding how to show that an implementation is correct in respect to the model is minimised, provided that the code generation tool is proved to generate correct code from the model.

In the following, it is described how templates and tasks are modelled, and how schedulability analysis and verification are conducted in TIMES.

### 10.4.1 Templates and Processes

TIMES uses templates and processes similar to those described for UPPAAL. That is, they use uninstantiated timed automata containing locations, guards, invariants, and channels. The main difference between templates in TIMES and UPPAAL is that the templates in TIMES are allowed to release tasks. Figure 10.3 depicts a TIMES template.

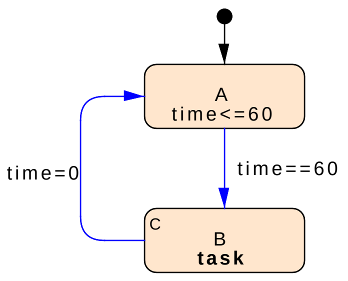


**Figure 10.3:** TIMES template illustrating how a task can be released periodically.

As shown, the template models a periodic release of `task`. The template initiates in location `A` and after 60 time units, it reaches location `B` at which the `task` is released in an atomic operation, denoted `c`, before resetting the clock variable and returning to the initial location.

### 10.4.2 Tasks

A task in TIMES consists of two parts: timing requirements and release model. The release model is defined by specifying the task to be periodic, sporadic, or controlled. For the periodic and sporadic tasks, TIMES automatically conducts the releases of the tasks according to the predefined timing requirements. Regarding the controlled tasks, a release model in terms of a template, modelling the release of the tasks, must be created.

Three different constraints exist for tasks. These are *timing*, *precedence*, and *resource* constraints, which are described below:

**Timing Requirements** Timing requirements consist of the task's relative deadline and WCET. Furthermore, if the task model is specified to be periodic or sporadic, a period or minimum inter-arrival time must be specified, respectively.

**Precedence Constraints** Tasks may need to execute according to a predefined precedence. Therefore, TIMES provides the possibility of specifying precedence for the set of tasks by using a precedence graph. Generally, the graph is an AND/OR precedence graph containing tasks, ordinary edges, and inter-iterative edges. The AND/OR nodes specify, as the names imply, boolean operators that must be fulfilled to follow an edge. The tasks are mapped to the set of specified tasks, that is, either controlled, periodic, or sporadic. Finally, the ordinary edges apply to all task iterations and inter-iterative edges apply to all except the first iteration.

Figure 10.4 depicts a precedence graph example. The dotted line models the inter-iterative edge. As shown, `taskC` can only be released if `taskA` or `taskB` have been released. Also, a release of `taskB` requires, besides the first time, that `taskC` has been released beforehand.

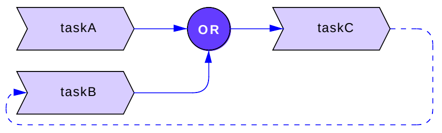


**Figure 10.4:** Example of a precedence graph in TIMES.

**Resource Constraints** A task can specify the shared resource it uses and when. The access pattern of shared resources can be specified as execution time of the task before it takes the resource, and the execution time for which it needs the resource before releasing it. The syntax for this specification is $S_i(P_i, V_i)$ where $S_i$ is the name of the shared resource, $P_i$ is the accumulated execution time needed for the task to reach the critical section, and $V_i$ is the accumulated execution time needed for the task to reach the end of the critical section.

The shared resource access is controlled by the priority ceiling protocol called ICPP. This is also the priority ceiling protocol used on the JOP.

Finally, it is possible to specify an interface for a task. An interface defines a change to either a clock or a variable which must be executed when the task finishes. This can be used to affect the state of the system using tasks, e.g. by releasing new tasks.

## 10.4.3 Schedulability Analysis with TIMES

When the set of tasks to be analysed have been modelled in TIMES with their correct constraints, the schedulability analysis can be conducted.

Figure 10.5 depicts an example of how TIMES allows the programmer to graphically follow the scheduling of tasks. This feature is especially convenient for debugging since it is possible to see precisely what goes wrong and where.

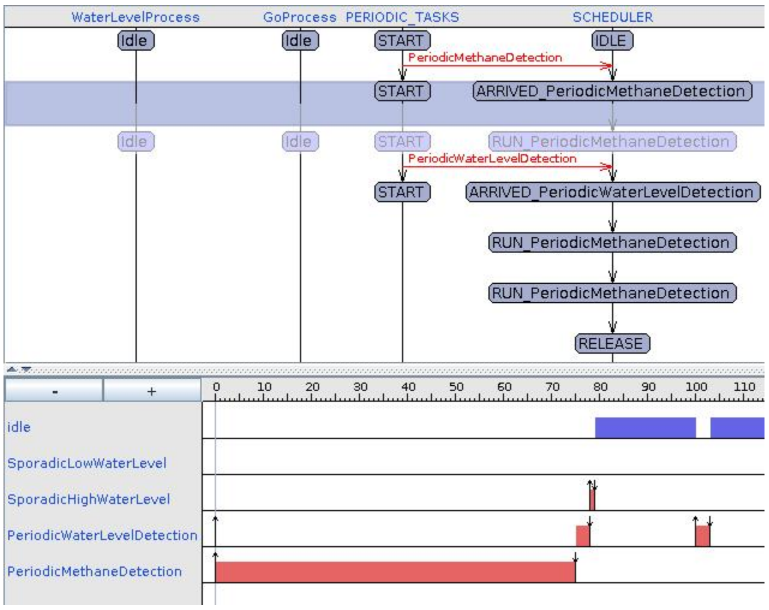


**Figure 10.5:** Example of a simulation run in TIMES.

As shown, the schedule is illustrated using a Gantt chart, where the marked rectangles represent execution of a given task. Also, the upward and downward arrows indicate releases and deadlines for tasks, respectively.

When a successful schedulability analysis has been conducted, it is possible to retrieve the WCRTs for each task. This can be used as an indication of the load of the system, since a WCRT close to the deadline would indicate that small additions to the system would result in the deadline being exceeded.

# 11 Design and Implementation

This chapter describes the design and implementation of the constrained LEGO mine pump example. Even though only the RJ implementation is described, the LEGO mine pump example is furthermore implemented in SCJ and PJ. The experience gained in this process will be used in Chapter 13 where the profiles are evaluated.

## 11.1 Design

The general design of the constrained LEGO mine pump example is shown in Figure 11.1. The mine pump is divided into three UML packages responsible for: sensors, actuators, and control. These are the parts of the general architecture of the system which resemble a layered architecture which becomes evident by considering that the `Control` package is the uppermost package using the underlying packages: `Sensors` and `Actuators` for applying its control logic. The functionality being dependent on the chosen real-time profile is therefore isolated in the control package thereby reducing the effort for implementing the mine pump using another real-time profile.

### Sensors

The sensors are responsible for interfacing with the actual sensor hardware, and creating a layer of abstraction hiding the functionality for interacting with these and the manipulation of sensor readings. The sensors only expose what the sensors are observing in terms of type of brick.

Brick A `Brick` represents an observed measurement of either methane or water. This means that the `MethaneSensor` must be able to differentiate methane bricks and water bricks.

Sensor This is the base class for all of the sensors that implements logic common for all sensors, such as the process of conducting sensor readings.

WaterSensor This is an abstract class, defining that its derived classes must implement functionality that determines whether their respective water level is reached.

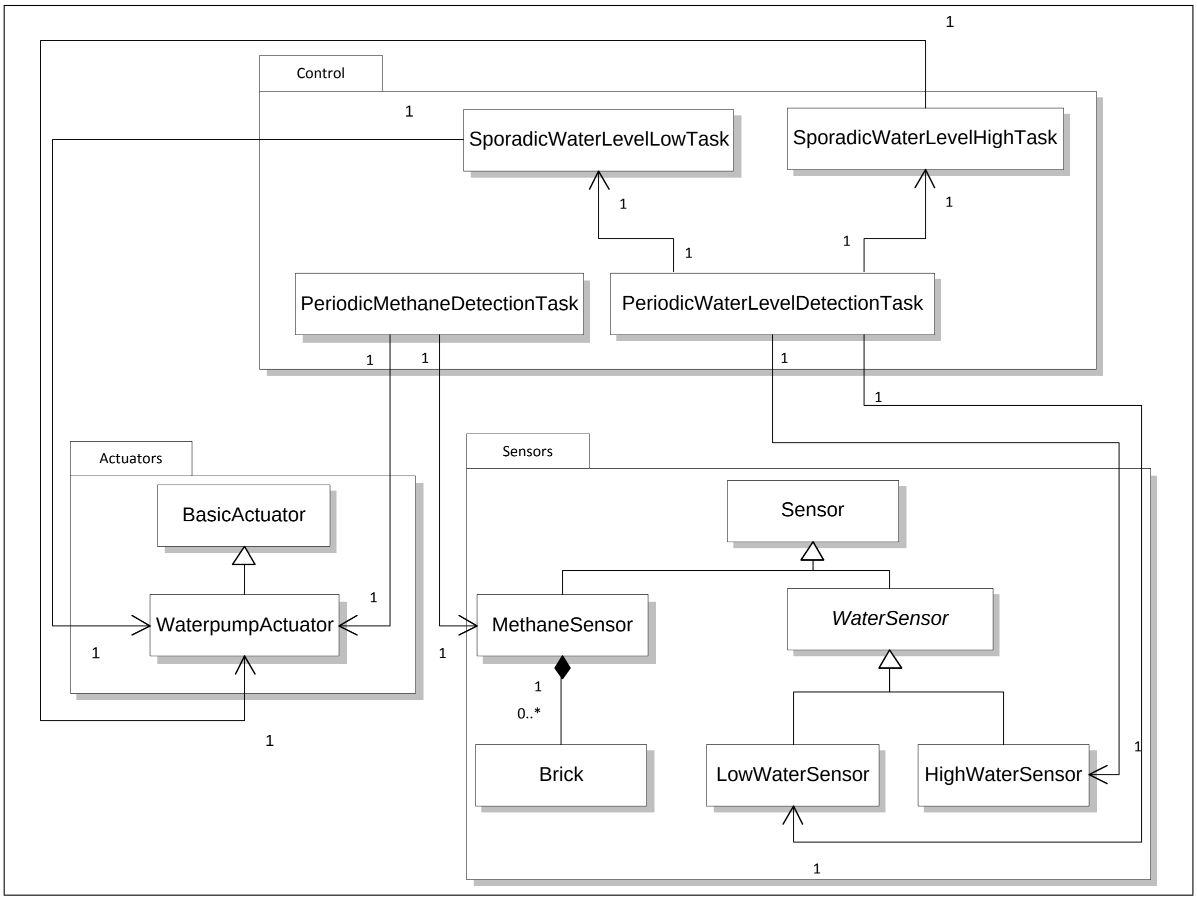


**Figure 11.1:** UML class diagram describing the design of the mine pump.

LowWaterSensor This sensor detects low water level. Since a false positive can potentially be observed when bricks are passing by the sensor, a number of consecutive readings is needed to provide evidence that the water level is actually low.

HighWaterSensor This class is almost similar to the `LowWaterSensor`, with the difference that it reacts to high water.

MethaneSensor The `MethaneSensor` class is responsible for detecting the concentration of methane observed over a history of readings.

### Actuators

These classes are responsible for controlling the actuators in the system. In general, there exists two types of actuators: actuators which drive the conveyor belts and feeder with constant speed and actuators capable of dynamically controlling the water pump.

BasicActuator This represents a basic actuator which can be started. This class is used for the conveyor belt actuators and the feeder.

WaterpumpActuator The class `WaterpumpActuator` inherits from the class `BasicActuator` and represents the actuator controlling the water pump. Besides the inherited functionality of starting an actuator, the class provides methods used to stop the actuator and control if the actuator is in an inoperable state such as when high methane concentration has been detected.

#### Control

These classes represent periodic and sporadic tasks present in the mine pump.

PeriodicWaterLevelDetectionTask This periodic task is responsible for polling the water level sensors to determine if either a high or a low water level is present in the mine shaft and for releasing a corresponding sporadic task if there is.

SporadicWaterLevelHighTask This task is released if the water level is high, and is thus responsible for handling this by starting the actuator if the methane level is below critical.

SporadicWaterLevelLowTask This task is released if the water level is low, and is thus responsible for handling this by stopping the actuator.

PeriodicMethaneDetectionTask This class represents a periodic task, checking if the current methane level in the mine pump is critical. Depending on the level of methane, this periodic task is responsible for ensuring that the water pump actuator is only running when the methane concentration is within the safe level.

### 11.1.1 Model of Design

To get confident in the descriptions of the mine pump example, the following is a description of how we model the problem. The model gives insight in our reasoning about the behaviour the mine pump. Furthermore, a verification is conducted in order to verify that the model yields properties similar to those expected from the requirements. As previously described, the used model-checker is UPPAAL.

Using model-checking, one must be aware of certain potential problems that may arise. The first is that the state space of the model should be limited due to the combinatorial explosion of the state space. If this is not accounted for, verifying certain properties of the model can potentially be a time consuming process or in worst-case, a problem that is not practically solvable.(Mathur, 2008)

The following describes the six templates: ENVIRONMENTFEEDER, PERIODICMETHANEDETECTION, PERIODICWATERLEVELDETECTION, SPORADICWATERLEVELHIGH, SPORADICWATERLEVELLOW, and WATERPUMP, used for modelling the constrained mine pump example. The templates having names starting with either periodic or sporadic, model the periodic and sporadic tasks described in the design of the mine pump. The EnvironmentFeeder models the `BasicActuator` used for controlling the environment. Furthermore, the WaterPump template models the `WaterPumpActuator`. We have decided not to model the three sensors, since these read a value when polled and are not responsible for implementing any other logic. The sensor readings are therefore incorporated in the PeriodicMethaneDetection and PeriodicWaterLevelDetection templates by checking for boolean values set by the EnvironmentFeeder.

Generally, the synchronisations use regular channels if not stated otherwise.

### Environment Feeder

The first template, shown in Figure 11.2, models the environment by periodically feeding a new brick, either water or methane, into the system. This is done by selecting a brick, and add it to the system through a call to the *updateEnvironment()* method. The purpose of *updateEnvironment()* is to update the values of the three sensors. This is done by setting the *isMethane* boolean variable representing if the methane concentration is critical and incrementing a variable representing the current amount of water in the system. The methane concentration is checked using an array of a given size where a history of previous elements are stored. The methane concentration is considered exceeded whenever the summation of methane elements in the history exceeds a predefined limit.

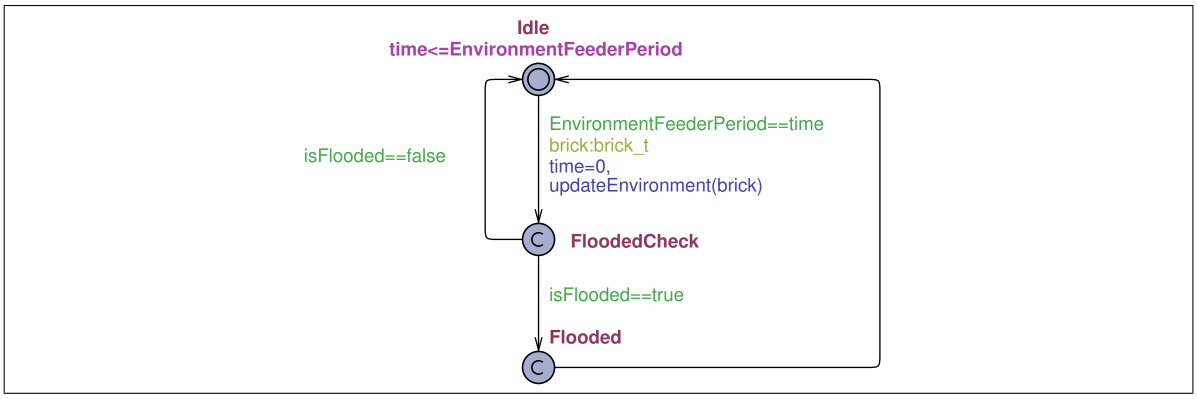


**Figure 11.2:** The EnvironmentFeeder UPPAAL template.

### Periodic Methane Detection

The periodic methane detection task is modelled as shown in Figure 11.3. Its purpose is to periodically detect if there is critical levels of methane in the mine. In case there is, a synchronisation is made with the WaterPump to bring this to an emergency state. Otherwise, a broadcast synchronisation on the *cancelEmergencyStopPump* channel is made with the WaterPump. The *cancelEmergencyStopPump* is made broadcast since the WaterPump not necessarily is in an emergency state when no methane is detected. Notice that when methane is detected the *isMethaneDetected* boolean is set to true, indicating that methane has currently been detected. This boolean is used to ease the verification of the WaterPump only being it its *Pumping* location when there is not detected methane.

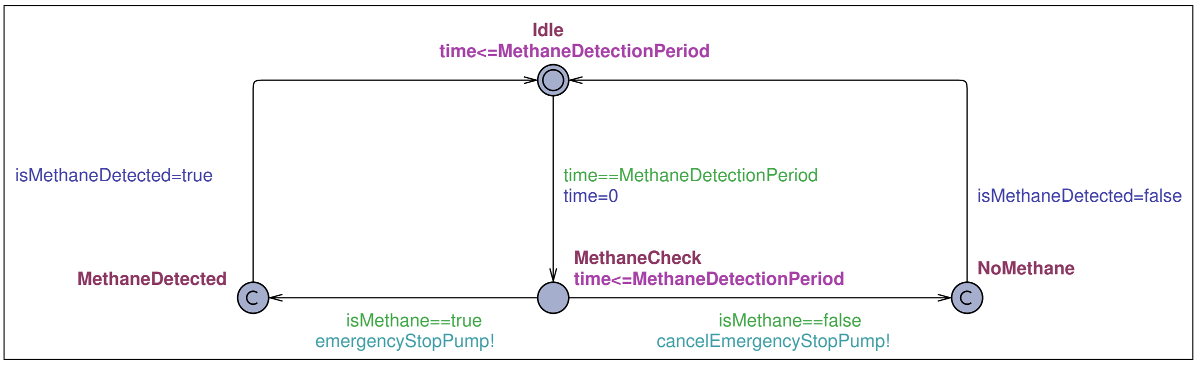


**Figure 11.3:** The PeriodicMethaneSensor UPPAAL template and its interaction with the WaterPump.

### Periodic Water Level Detection

The template responsible for detecting the water level present in the mine shaft is depicted in Figure 11.4.

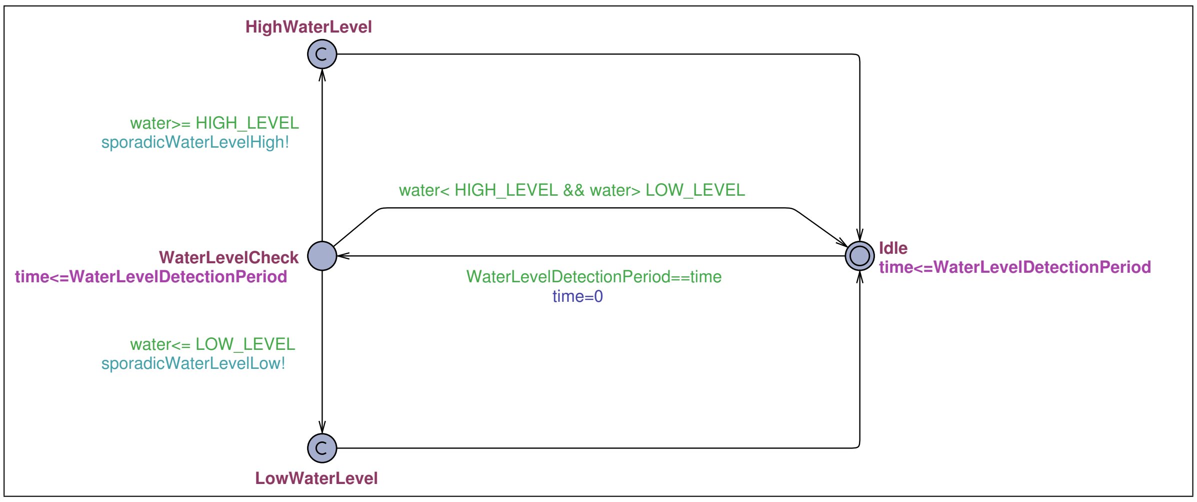


**Figure 11.4:** The PeriodicWaterLevelDetection UPPAAL template and its interaction with the other templates.

The template periodically determines whether one of the two critical water levels are reached. If one of them is, the *sporadicWaterLevelHigh* or *sporadicWaterLevelLow* channel is synchronised. In case none of the critical water levels have been reached, the *Idle* location is reached without further processing.

### Sporadic Water Level High and Low

To model the two tasks of sporadically starting and stopping the water pump, the SporadicWaterLevelHigh and SporadicWaterLevelLow templates are used. As shown in Figure 11.5, the two templates are rather simple. The leftmost, modelling sporadically starting the water pump, is initiated when synchronised on the *sporadicWaterLevelHigh* channel by the PeriodicWaterLevelDetection template. This models the idea of periodically checking the high water and release a sporadic task if the pump should be operating. After the synchronisation, the WaterPump is started by synchronising on the *startWaterPump* channel.

The SporadicWaterLevelLow template, shown to the right, is similar to SporadicWaterLevelHigh, with the only difference of simulating stopping the water pump instead of starting it.

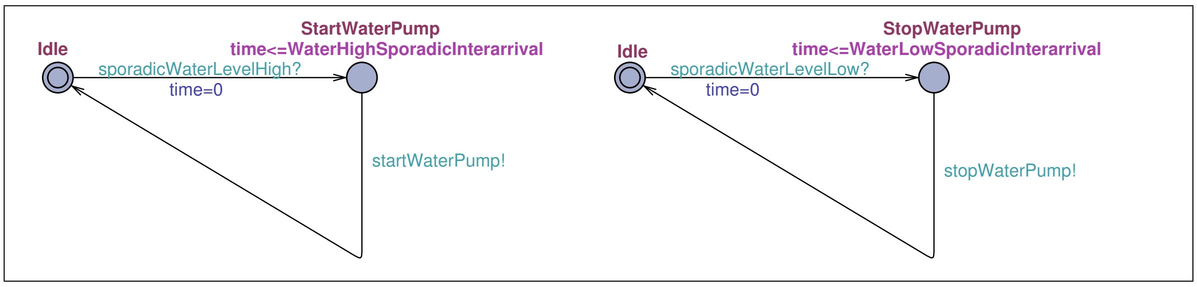


**Figure 11.5:** The SporadicWaterLevelHigh UPPAAL template, shown to the left, and for the SporadicWaterLevelHigh template, shown to the right.

### Water Pump

The template for the WaterPump is depicted in Figure 11.6.

Since critical methane levels can be detected at any given time all uncommitted locations have outgoing edges ending in the *EmergencyStop* location. From this location there is only one outgoing edge leading to another location, that is, only a synchronisation on the *cancelEmergencyStopPump* channel can cancel an emergency stop.

If there is no critical level of methane detected, the water pump should start removing the water if it is started by the sporadic task fired when high water level is detected. This is modelled by synchronising on the *startWaterPump* channel and then enter the *Pumping* location from which it can enter the loop that decrements the *water* variable in the mine shaft by means of a call to *pumpingWater()*. During this decrementation, the PeriodicWaterLevelDetection template can determine that the water level is low thereby stopping the water pump. If the WaterPump decrements the *water* variable faster than the PeriodicWaterLevelDetection can determine that the water level is too low, it is possible that the water level eventually becomes zero thereby drying up the mine shaft which is an undesirable property that should not happen.

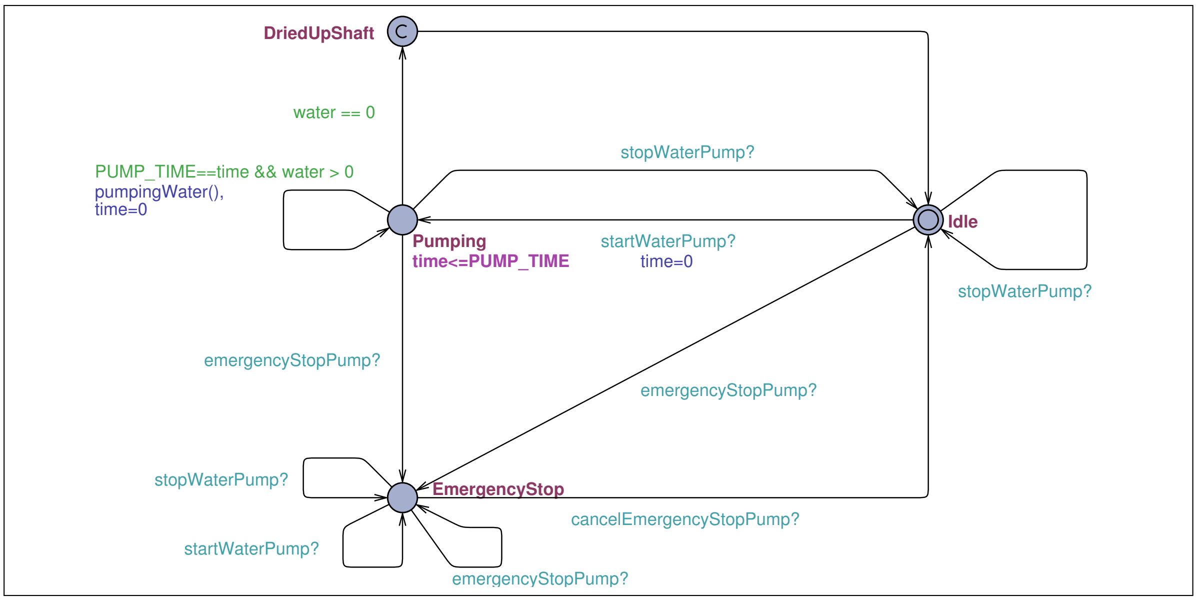


**Figure 11.6:** Model for the WaterPump UPPAAL template.

Notice that the reason for synchronising on the *stopWaterPump* channel in the *Idle* location is that the WaterLevelDetection template periodically tests for either high or low water level. Hence, each time a low water level is detected, the pump is requested to stop.

## Verification of Properties

During the verification process, a single instance of each of the templates has been used to mimic the structure of the constrained mine pump. The properties which are verified in the following have been elicited from the requirements specification, in Section 9.1.2, of the mine pump example to ensure that the model correctly simulates it.

## Sanity Checks

The properties listed in Listing 11.1, have been verified for sanity purposes of the models. Essentially, the models are verified to be deadlock-free and that key locations are reachable.

```
A[] !deadlock
E<> PeriodicMethaneDetection.MethaneDetected
E<> PeriodicWaterLevelDetection.HighWaterLevel
E<> PeriodicWaterLevelDetection.LowWaterLevel
E<> PeriodicMethaneDetection.NoMethane
E<> EnvironmentFeeder.Flooded
```

**Listing 11.1:** Verified properties checking basic properties of the mine pump.

### Requirement 1: The water pump must only be operating when there is no critical level of methane in the mine

This requirement is verified by the property listed in Listing 11.2. In general, this requirement is verified by ensuring that the WaterPump is not in its *Pumping* location whenever methane is present. If this is true, the requirement is satisfied since the water pump is not operating when methane is detected.

```
A[] WaterPump.Pumping imply isMethaneDetected==false
```

**Listing 11.2:** Verified property of the mine pump requirement 1.

### Requirement 2: When the water level is high, the water pump must be started

This requirement is verified through the single property in Listing 11.3, which simply verifies that the water pump is started if high water level is detected, given no methane has been detected.

```
PeriodicWaterLevelDetection.HighWaterLevel --> WaterPump.Pumping || WaterPump.
    EmergencyStop
```

**Listing 11.3:** Verified property of the mine pump requirement 2.

### Requirement 3: When the water level is low, the water pump must be stopped

This requirement is verified through the single property in Listing 11.4, which simply verifies that the water pump is stopped if low water level is detected, given no methane has been detected.

```
PeriodicWaterLevelDetection.LowWaterLevel --> WaterPump.Idle || WaterPump.
    EmergencyStop
```

**Listing 11.4:** Verified property of the mine pump requirement 2.

### Requirement 4: If there is no methane in the mine, the mine must never be flooded

This requirement is verified through the single property in Listing 11.5, which verifies that the the mine is never flooded if the water pump is not in its emergency state.

```
A[] EnvironmentFeeder.Flooded imply WaterPump.EmergencyStop
```

**Listing 11.5:** Verified property of the mine pump design.

The defined verification properties have all been verified by UPPAAL. Thus, it is concluded that the requirements defined for the mine pump are satisfied in the design.

The verification was made on an application server with two Xeon E5420 2.5Ghz CPUs and 31Gb of RAM. Since UPPAAL does not support multi-threading, a single CPU core was used. The overall verification time was dominated by the verification of the safety properties that each in average consumed 24 hours.

## 11.2 Implementation

The practical implementation is made in the three described profiles. Excerpts of these implementations are provided in Appendix I. However, the demonstration implementation on the JOP is solely made using the RJ profile. The reason for this is that it is the only profile which API is nearly fully implemented on the JOP as mentioned in the introduction.

The choice of implementing the mine pump in the RJ profile affects the design slightly, since the profile uses threads to implement periodic tasks and asynchronous event handling to implement sporadic tasks. The changes in the design are confined to the control part of the program, for which the class naming and inheritance is a bit different. Regardless, the concept and structure of the program is almost preserved.

In the following, implementation details for each of the major parts; control, sensors, and actuators, are presented.

### 11.2.1 Control

Periodic tasks are in RJ implemented using the `PeriodicThread` class which is given a `Runnable` instance implementing the periodic task. Sporadic tasks are implemented using the `SporadicEvent` class and the `SporadicEventHandler` class.

Listing 11.6 shows how periodic and sporadic tasks are initialised.

```
SporadicWaterLevelHigh waterHighHandler = new SporadicWaterLevelHigh(
    new PriorityParameters(SPORADIC_HIGH_PRIORITY),
    new SporadicParameters(new RelativeTime(SPORADIC_WATER_MIT, 0),1),
    waterpumpActuator);

SporadicWaterLevelLow waterLowHandler = new SporadicWaterLevelLow(
    new PriorityParameters(SPORADIC_LOW_PRIORITY),
    new SporadicParameters(new RelativeTime(SPORADIC_WATER_MIT, 0),1),
    waterpumpActuator);

SporadicEvent waterHighEvent = new SporadicEvent(waterHighHandler);
SporadicEvent waterLowEvent = new SporadicEvent(waterLowHandler);

new PeriodicThread(
    new PriorityParameters(PERIODIC_WATER_PRIORITY),
16      new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(
            PERIODIC_WATER_PERIOD, 0)),
17      new WaterLevelDetectionRunnable(waterHighEvent, waterLowEvent, highWaterSensor,
            lowWaterSensor));
```

**Listing 11.6:** An excerpt of the initialisation of periodic and sporadic tasks.

In lines 1-4 and lines 6-9, two sporadic tasks are created which handle the two tasks related to high and low water in the mine shaft. Each of the handlers are given a priority and a reference to the water pump. Furthermore, they are given an instance of `SporadicParameters` which represents the minimum inter-arrival time, defined by *`SPORADIC_WATER_MIT`*. Note that the `SporadicParameters` instance is given an instance of `RelativeTime` and an integer argument. Currently the meaning of the last argument is unknown, since it is inconsistently documented in the RJ. However, it is a buffer of some kind. The RJ implementation on the JOP ignores this parameter. The sporadic tasks are given to two different instances of `SporadicEvent` in lines 11-12.

In lines 14-17, a periodic task is created. This task is configured with the *WATER_ PRIORITY* priority, an offset start time which is zero, and a `RelativeTime` instance that dictates a period for the thread of *PERIODIC_ WATER_ PERIOD*. Furthermore, the task is initialised given a `WaterLevelDetectionRunnable` as argument that implements the actual logic. This runnable is provided references to the necessary sensors and sporadic events firing the sporadic tasks. When the periodic task detects either high or low water, it calls the *`fire()`* method on the associated asynchronous event, effectively releasing `SporadicWaterLevelHigh` or `SporadicWaterLevelLow` accordingly.

### 11.2.2 Sensors

The sensors are responsible for handling the sensor readings and transform them into corresponding bricks. As an example, Listing 11.7 shows the method *`isCriticalMethaneLevelReached()`* which is part of the `MethaneSensor` class. This method is responsible for reading the methane sensor values and either returning true or false depending on the current concentration of observed methane.

```
public boolean isCriticalMethaneLevelReached() {
    int sensorReading = conductMeasurement();

    if(super.isBrickMethane(sensorReading) && detectBrick == true)
    {
        mHistory.addMeasurement(Bricks.GAS);
        detectBrick = false;
    }
    else if(super.isBrickWater(sensorReading) && detectBrick == true)
    {
        mHistory.addMeasurement(Bricks.WATER);
        detectBrick = false;
13      }
14      else
15      {
16          detectBrick = true;
17      }
18      return mHistory.getMethaneLevel() >= this.criticalMethaneLevel;
19  }
```

**Listing 11.7:** An excerpt of the handling of the methane sensor readings.

In line 2, a reading from the sensor is conducted by issuing the inherited *conductMeasurement()* method. This reading is then analysed by the methods *isBrickMethane* and *isBrickWater* in lines 4 and 9. These methods are part of the `Sensor` class which the class `MethaneSensor` inherits. The two methods return true if the sensor reading is in the colour range denoting that the brick is either methane or water, respectively. In addition, note the boolean variable *detectBrick* which is used in the two conditions in line 4 and 9. This boolean is introduced to avoid that the same brick is observed twice. For a detailed explanation of this issue, see Appendix G. To overcome this problem, the implementation requires that between consecutive detections of bricks there must at least be one sensor reading where no brick is detected. This check is conducted in line 14, and sets the boolean *detectBrick* to true thereby allowing subsequent readings to add a brick to the measurement history.

If a brick is observed, it is added in a history object in line 6 or 11. This history object maintains a history of the last 10 observed bricks. Additional bricks added to the object replace old bricks. If no brick is detected, no changes are made to the observed methane level. By using this approach, it is possible to model the case where the level of methane is above some level relative to the number of bricks observed, emulating a real sensor.

In line 18, the methane level is determined and compared with the *criticalMethaneLevel* threshold. The methane level is determined as the number of methane bricks represented in the history. In the initial implementation, this check was slightly different. The methane level was represented as the percentage instead of the number of methane bricks. Suitably, a floating point number was used for representing this value. We afterwards observed that the operations connected with floating point numbers on the JOP are costly because floating point operations are implemented in software. This means that they significantly influence the WCET. Furthermore, a problem with calculating the percentage is that an unchecked exception can be thrown if a division by zero is made. The fact that Java has unchecked exceptions is in our opinion not a satisfying construct in respect to safety critical systems. This is because Java does not enforce nor give suggestions about their potential occurrence.

### 11.2.3 Actuators

In the mine pump, only one instance of the `WaterPump` class exists. All other actuators are part of the conveyor belts and feeder and are controlled by instances of the class `BasicActuator`, which is only capable of starting them. The water pump is controlled by an instance of

WaterpumpActuator inheriting from the BasicActuator. Besides being able to start the actuator, WaterpumpActuator is also capable of stopping it and handle the emergency state occurring when a critical methane level has been reached. Listing 11.8 shows the implementation of the methods *start()* and *emergencyStop()* implementing these functionalities.

```
1  public synchronized void start() {
2      if (this.emergencyState == false) {
3          super.start();
4      }
5  }
6
7  public synchronized void emergencyStop(boolean performEmergencyStop) {
8      this.emergencyState = performEmergencyStop;
9      if (performEmergencyStop == true) {
10         this.stop();
11     }
12 }
```

**Listing 11.8:** The *start()* and *emergencyStop()* methods of the WaterPumpActuator class.

Lines 1-5 override the *start()* method of the parent class BasicActuator to implement that the actuator cannot be started if it is in the emergency state. If the emergency state is active, the request for starting the actuator is ignored.

In lines 7-12, the emergency state can be set. If the emergency state is invoked, the actuator is stopped, ensuring that the actuator is not running while the emergency state is present. Note that both of the methods are synchronised in order to prevent race conditions which might occur because multiple handlers operate on the same WaterpumpActuator instance. The act of stopping the actuator could have been omitted, transferring the responsibility for stopping the actuator when entering the emergency mode to the callee. However, this approach is not chosen since it requires the callee to remember to call *stop()* and would require some locking to ensure no race conditions occur between transferring the actuator into an emergency state in one thread and starting the actuator in another.

# 12 Worst-Case Execution Time Analysis and Schedulability Analysis

The purpose of this chapter is to illustrate the problem of WCET analysis and schedulability analysis through practically applying their respective theories on the constrained mine pump example.

The WCET analysis and schedulability analysis have been conducted using the tools outlined in Chapter 10. Furthermore, in order to gain further practical knowledge in schedulability analysis, both utilisation-based schedulability analysis and RTA have been conducted.

## 12.1 Worst-Case Execution Time

The WCA tool has been used for conducting WCET analysis of the mine pump application. Specifically, the tool has been used to analyse the four periodic and sporadic tasks in the system such that a later schedulability analysis can be conducted. Due to a shared resource being present in the system, specifically the `WaterpumpActuator`, the WCETs before, after, and during the acquisition of this shared resource need to be extracted. This is because the schedulability tool TIMES needs this information in its model of the system.

The WCA tool allows for configuration by command-line options. These options need to reflect the specific processor type, hardware related settings, and options regarding the actual computation of the WCET such as IPET-based settings. If the WCA tool is misconfigured, it can result in determining unsafe WCETs. A review of the settings has been conducted to ensure that the settings are in correspondence with the specifications of the JOP implemented on the Altera Cyclone EP1C12 FPGA. However, some of the settings were complicated to review. As an example, the cache size and number of cache blocks were difficult since the JOP implementation is unspecific in this regard. To determine these two settings, the VHDL files describing the hardware design of JOP, have been examined from which it has been determined that the implemented method cache has a size of 1024 words where each word is 32 bit shared among 16 cache blocks (JOP, 2008). This discovery is further documented in Appendix H.

The WCA tool offers the possibility of using both IPET- and model-based WCET analyses. Hence to explore the possibilities of the tool both of these methods have been used. The resulting WCETs are slightly more precise using the model-based approach and the computation time was nearly identical to that of IPET. We assume that the almost identical computation times are due to the mine pump application being relatively small in terms of lines of code. Note that the WCETs derived from the model-based WCET analysis are used in the following.

### 12.1.1 Conducting WCET Analysis

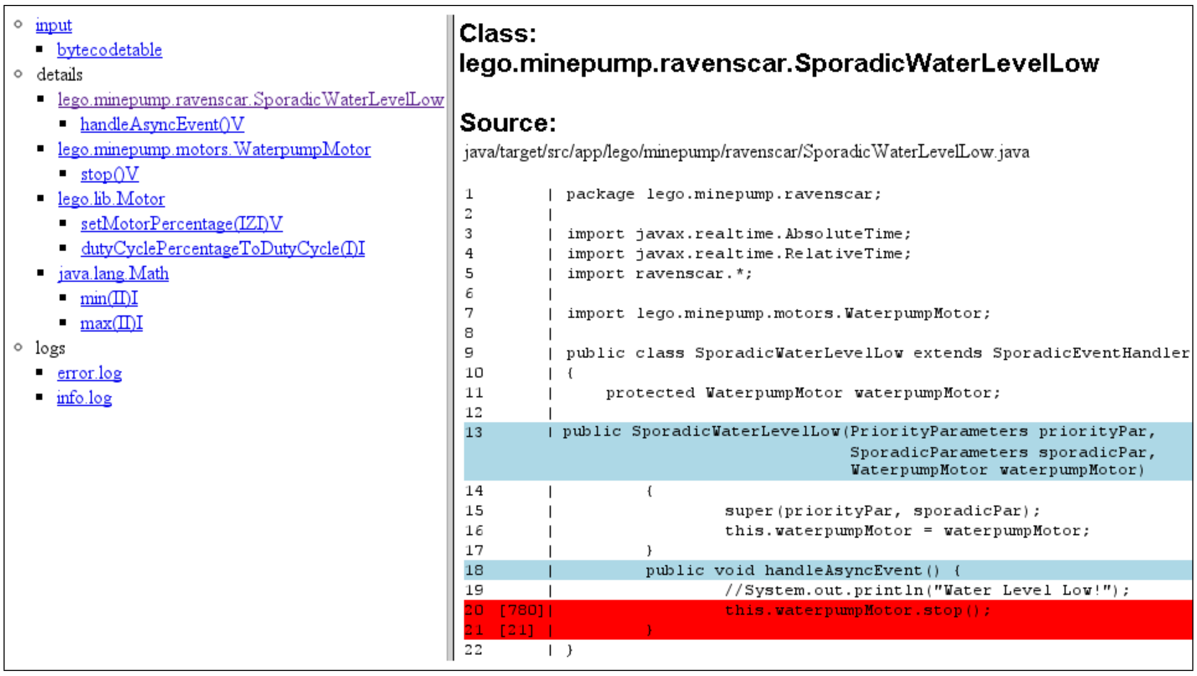
Using the above described settings, the WCA tool is able to extract the WCET values shown in Table 12.1.

| Task | WCET before | WCET during | WCET after | WCET total |
|---|---|---|---|---|
| Periodic Methane Detection | 13,883 | 1,008 | 733 | 15,624 |
| Sporadic High Water Level | 0 | 937 | 21 | 958 |
| Sporadic Low Water Level | 0 | 780 | 21 | 801 |
| Periodic Water Level | - | - | - | 2,134 |

**Table 12.1:** WCET extracted for the four tasks of the mine pump application. The WCETs are represented in number of clock cycles. Hence, for instance, the actual total time taken for periodic methane detection on the Altera Cyclone board with a 100 MHz processor is 0.15624ms. Note that for periodic water level, no WCETs are present before, during, and after taking the resource because this periodic task does not acquire it.

The WCA tool generates a HTML report showing the source code of the method for which the WCET analysis was conducted. The source code is annotated with the determined WCET and the worst-case paths in the program flow are highlighted. This report is used to extract the WCET estimates for further usage. Figure 12.1 shows an example of the generated HTML reports which in this case is the results of *Sporadic Low Water Level.*

It was initially observed that the periodic methane detection task had a very high WCET of $\sim 1.56 \cdot 10^6$ clock cycles, in contrast to the $\sim 1.50 \cdot 10^4$ clock cycles present in the table. The primary cause to this high WCET was identified to be due to a call to the Java method `equals()`, used to compare two objects. Refactoring this call by replacing it with a comparison between two integers reduced the computational time substantially and yielded the $\sim 1.50 \cdot 10^4$ clock cycles instead. From this, we conclude that one must be aware of which methods are used for implementing the same functionality.



- input
  - bytecodetable
- details
  - lego.minepump.ravenscar.SporadicWaterLevelLow
    - handleAsyncEvent()V
  - lego.minepump.motors.WaterpumpMotor
    - stop()V
  - lego.lib.Motor
    - setMotorPercentage(IZI)V
    - dutyCyclePercentageToDutyCycle(I)I
  - java.lang.Math
    - min(II)I
    - max(II)I
- logs
  - error.log
  - info.log

**Class:**
**lego.minepump.ravenscar.SporadicWaterLevelLow**

**Source:**
java/target/src/app/lego/minepump/ravenscar/SporadicWaterLevelLow.java

```
       | package lego.minepump.ravenscar;
       |
       | import javax.realtime.AbsoluteTime;
       | import javax.realtime.RelativeTime;
       | import ravenscar.*;
       |
       | import lego.minepump.motors.WaterpumpMotor;
       |
       | public class SporadicWaterLevelLow extends SporadicEventHandler
       | {
       |     protected WaterpumpMotor waterpumpMotor;
       |
       | public SporadicWaterLevelLow(PriorityParameters priorityPar,
                                      SporadicParameters sporadicPar,
                                      WaterpumpMotor waterpumpMotor)
       |         {
       |                 super(priorityPar, sporadicPar);
       |                 this.waterpumpMotor = waterpumpMotor;
       |         }
       |         public void handleAsyncEvent() {
       |                 //System.out.println("Water Level Low!");
 [780] |                 this.waterpumpMotor.stop();
 [21]  |         }
       | }
```

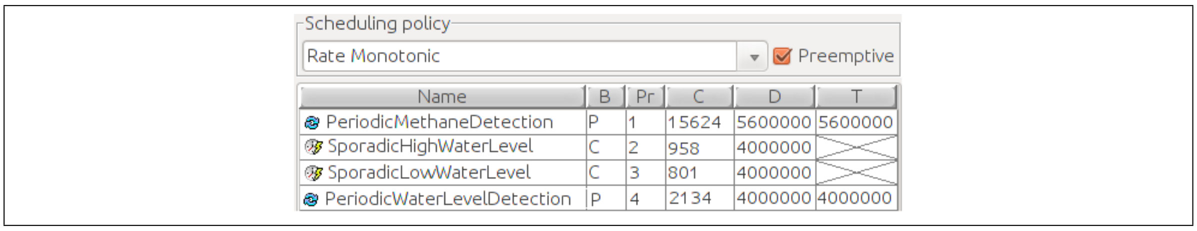
**Figure 12.1:** Example of the generated HTML reports the WCA tool is able to make. This specific example is the report for the *handleAsyncEvent()* method of the `SporadicWaterLevelLow` class.

## 12.2 Schedulability Analysis

In the following, the schedulability analysis is presented using the TIMES tool, utilisation-based schedulability analysis, and RTA.

### 12.2.1 Using TIMES

The four tasks of the mine pump are declared with their corresponding WCET found in the previous section using WCA. Figure 12.2 depicts how the tasks are specified in TIMES.



Scheduling policy: Rate Monotonic ☑ Preemptive

| Name | B | Pr | C | D | T |
|---|---|---|---|---|---|
| PeriodicMethaneDetection | P | 1 | 15624 | 5600000 | 5600000 |
| SporadicHighWaterLevel | C | 2 | 958 | 4000000 | |
| SporadicLowWaterLevel | C | 3 | 801 | 4000000 | |
| PeriodicWaterLevelDetection | P | 4 | 2134 | 4000000 | 4000000 |

**Figure 12.2:** Screenshot of TIMES illustrating how the tasks are defined with their corresponding type, $B$, priority, $Pr$, WCET, $C$, deadline, $D$, and period, $T$.

The firing of the two sporadic tasks is controlled by one of the periodic tasks. This means that the two sporadic tasks are considered *controlled tasks* in TIMES terminology. This means that their release must be controlled by a process modelled in the template. This process is denoted WATERLEVELDETECTIONPROCESS and is shown in Figure 12.3.

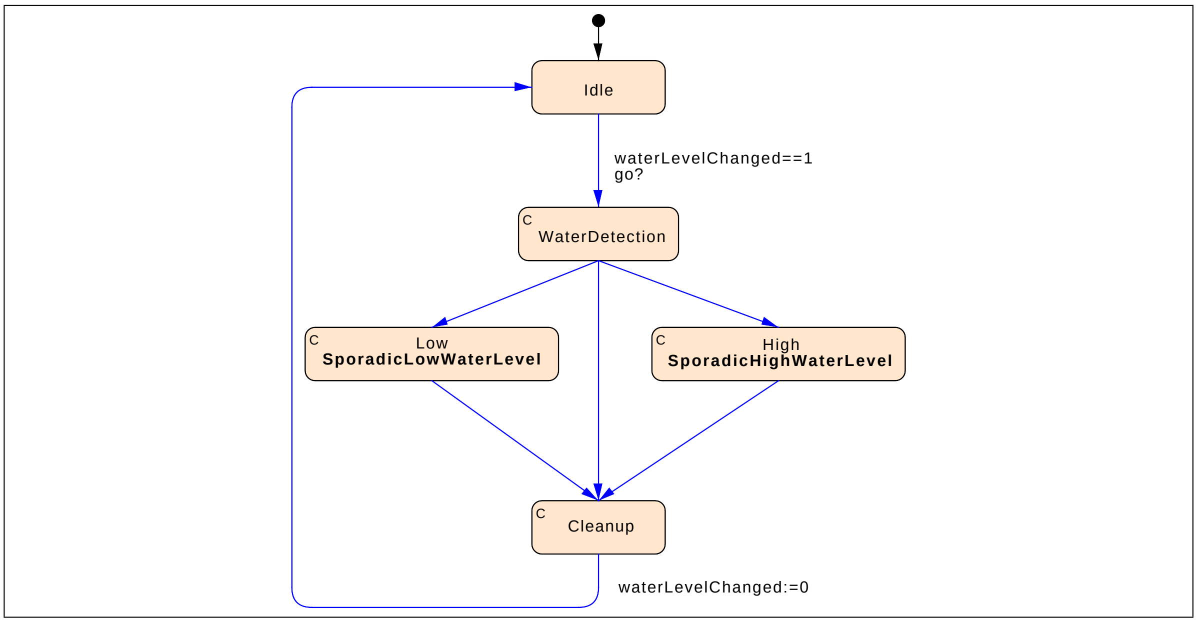


**Figure 12.3:** TIMES template of the WATERLEVELDETECTIONPROCESS whose purpose is to release either of the two sporadic tasks.

As shown, the template is initialised in the ***Idle*** location from where it can move to ***WaterDetection*** whenever the ***waterLevelChanged*** variable is set to one, and a synchronisation is made on the ***go*** urgent channel. The ***waterLevelChanged*** variable is set to one each time the ***PeriodicWaterLevelDetection*** is finished, meaning that it contains a $waterLevelChanged = 1$ expression in its *TIMES interface*. From the ***WaterDetection*** location, one of the three outgoing edges is taken in an atomic operation. Reaching ***Low*** or ***High*** releases the corresponding controlled task before ending in ***Cleanup***. The third outgoing edge reaches ***Cleanup*** without releasing any other tasks, modelling that the water level is between high and low. Before returning to ***Idle***, the ***waterLevelChanged*** is set to zero.

As described, the template synchronises on the urgent ***go*** channel. The sender of the signal is shown in Figure 12.4. The reason for this arrangement is to force the WATERLEVELDETECTIONPROCESS template to leave its ***Idle*** location whenever the guard evaluates to true. If the synchronisation was not there, it cannot be guaranteed that the outgoing edge is fired when the guard is true. It only makes the firing a legal operation. However, with the urgent synchronisation, the edge is taken the moment it is possible.

The WATERLEVELDETECTIONPROCESS template, combined with the interface of ***PeriodicWaterLevelDetection*** force the precedence of the system to be that a release of either of the two controlled tasks is dependent on a release of the ***PeriodicWaterLevelDetection***.

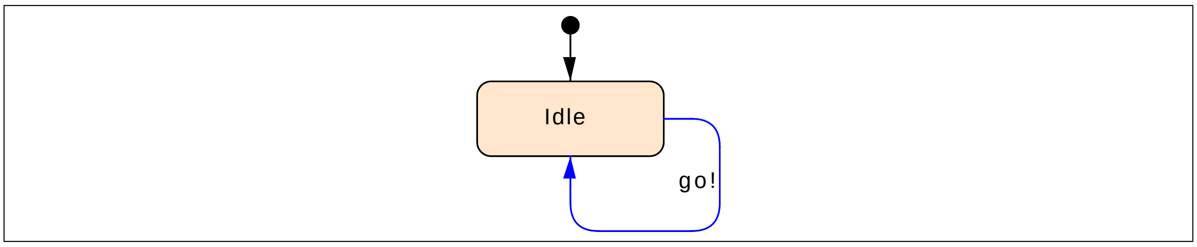


**Figure 12.4:** TIMES template depicting the GO template. Its purpose is to force the WATERLEVELDETECTIONPROCESS to execute when the *PeriodicWaterLevelDetection* is done.

Executing the schedulability analysis in TIMES using the described settings, verifies that the system is schedulable with the FPS scheduling policy. It is, however, important to note that TIMES draws its conclusion based on a task model that most likely is different from the one used in our mine pump example. TIMES does e.g. not allow the user to specify the context-switching times. Hence, we conclude that the system cannot safely be said to be schedulable.

### 12.2.2 Using Utilisation-Based Schedulability Analysis

Employing this method is fairly simple compared to other methods for schedulability analysis. The utilisation of the system containing four tasks is derived using Equation 5.3:

$$\begin{aligned} U &= \frac{15,624}{5,600,000} + \frac{2,134}{4,000,000} + \frac{958}{4,000,000} + \frac{801}{4,000,000} \\ &\approx 0.0038 \end{aligned} \tag{12.1}$$

The maximum allowed utilisation is determined to be:

$$\begin{aligned} U_{\max} &= 4\left(2^{\frac{1}{4}} - 1\right) \\ &\approx 0.76 \end{aligned} \tag{12.2}$$

The conclusion from this result is that since $0.0038 \leq 0.76$, the system is said to be schedulable, given the simple task model is used. However, the mine pump is not based on the simple task model, hence this conclusion is not safe because e.g. blocking is not considered.

### 12.2.3 Response Time Analysis

The RTA of the mine pump application is carried out by using the formula in Equation 5.10, thereby taking blocking into account.

In the following equation the WCRT of the methane detection task is determined:

$$\begin{aligned}
R^0_{methane} &= 15,624 + 0 + \left\lceil \frac{0}{4,000,000} \right\rceil \cdot 958 + \left\lceil \frac{0}{4,000,000} \right\rceil \cdot 801 + \left\lceil \frac{0}{4,000,000} \right\rceil \cdot 2,134 \\
&= 15,624 \quad \Rightarrow \\
R^1_{methane} &= 15,624 + 0 + \left\lceil \frac{15,624}{4,000,000} \right\rceil \cdot 958 + \left\lceil \frac{15,624}{4,000,000} \right\rceil \cdot 801 + \left\lceil \frac{15,624}{4,000,000} \right\rceil \cdot 2,134 \\
&= 19,517 \quad \Rightarrow \\
R^2_{methane} &= 15,624 + 0 + \left\lceil \frac{19,517}{4,000,000} \right\rceil \cdot 958 + \left\lceil \frac{19,517}{4,000,000} \right\rceil \cdot 801 + \left\lceil \frac{19,517}{4,000,000} \right\rceil \cdot 2,134 \\
&= 19,517.
\end{aligned}$$

Thus, the methane detection task has a WCRT of 19,517 clock cycles which is substantially lower than its deadline of 3,000,000 clock cycles. The same calculation is conducted for the high water sporadic task:

$$\begin{aligned}
R^0_{\text{high water}} &= 958 + 14,891 + \left\lceil \frac{0}{4,000,000} \right\rceil \cdot 801 + \left\lceil \frac{0}{4,000,000} \right\rceil \cdot 2134 \\
&= 15,849 \quad \Rightarrow \\
R^1_{\text{high water}} &= 958 + 14,891 + \left\lceil \frac{15,849}{4,000,000} \right\rceil \cdot 801 + \left\lceil \frac{15,849}{4,000,000} \right\rceil \cdot 2,134 \\
&= 18,784 \quad \Rightarrow \\
R^2_{\text{high water}} &= 958 + 14,891 + \left\lceil \frac{18,784}{4,000,000} \right\rceil \cdot 801 + \left\lceil \frac{18,784}{4,000,000} \right\rceil \cdot 2,134 \\
&= 18,784.
\end{aligned}$$

From this, it is concluded that the high water detected sporadic task does not exceed its deadline. A similar calculation is conducted for the low water sporadic task:

$$\begin{aligned}
R^0_{\text{low water}} &= 801 + 0 + \left\lceil \frac{0}{4,000,000} \right\rceil \cdot 2,134 \\
&= 801 \quad \Rightarrow \\
R^1_{\text{low water}} &= 801 + 0 + \left\lceil \frac{801}{4,000,000} \right\rceil \cdot 2,134 \\
&= 2,935 \quad \Rightarrow \\
R^2_{\text{low water}} &= 801 + 0 + \left\lceil \frac{2,935}{4,000,000} \right\rceil \cdot 2,134 \\
&= 2,935.
\end{aligned}$$

As shown, the WCRT is less than the task's deadline. Finally, the water level detection task can be in the following:

$$\begin{aligned} R^2_{water} &= 2,134 + 0 \\ &= 2,134. \end{aligned}$$

Similarly to the other tasks, the WCRT of this task is less than its deadline.

Since all tasks have a WCRT lower than their respective deadlines, the system can be concluded to be schedulable, given that only blocking is taken into account.

**Comparing RTA with TIMES**

As described, in Chapter 10, the TIMES tool allows WCRTs to be analysed for the tasks. Since we have made these manually, we want to compare them with the model-based approach. Table 12.2 shows the comparison.

| Task | Manual | TIMES tool |
|---|---|---|
| Periodic Methane Detection | 19,517 | 18,716 |
| Sporadic High Water Level | 18,784 | 16,582 |
| Sporadic Low Water Level | 2,935 | 801 |
| Periodic Water Level | 2,134 | 2,134 |

**Table 12.2:** Comparison of WCRT obtained manually and through TIMES.

As shown, all the WCRTs given by the TIMES tool are less than or equal to those given by the manual RTA. This was as expected since the model-based approach should provide more precise WCRTs than the traditional approach. Especially, the WCRT of the sporadic task fired when the water level is low is remarkably more precise using the model-based approach. The reason for this deviation is that the RTA is not aware of the release pattern of the sporadic tasks and thereby yields this pessimistic WCRT. That is, RTA expects that the sporadic task is blocked due to the shared resource being locked by the lower priority sporadic task. However, as previously described, the two sporadic tasks cannot be executing simultaneously due to their release pattern.

# Part IV

# Conclusion

# 13 Evaluation of Real-Time Java Profiles

Recent research has shown uncertainty in how a real-time Java profile should be designed (Bøgholm et al., 2009; Søndergaard et al., 2006; Schoeberl et al., 2007; Kwon et al., 2002). As previously described, in Chapter 8 and 9, we have implemented an example application in three of the profiles to gain experience with them. This has given us the ability to develop our own opinions of what constitutes a good profile.

One of the arguments for replacing C and Ada with Java is that it has higher level of abstraction. Our approach to the evaluation builds on the assumption that developers want to write real-time applications with a high level of abstraction. Thus, we generally prefer solutions that are simple and moves more responsibility to the language itself.

This chapter describes our experiences and evaluation of using the three different real-time Java profiles. First we introduce and argument for the criteria that forms the basis of the evaluation. Afterwards, each profile is evaluated against the criterion and finally, the profiles are compared. The reader is advised to consult the accompanying Appendix I during the following which shows an excerpt of the implementation of the mine pump using each of the profiles.

To our best effort, we have been objective in our evaluation of the profiles. It should be noted, however, that we do not have a neutral affiliation with the PJ since some of its authors are personally known.

## 13.1 Evaluation Criteria

To concretise the evaluation, a set of evaluation criteria are used. These are chosen based on areas of the profiles that are noticeable different. Furthermore, some of them are chosen since we find them important to be fulfilled by a safety-critical profile. The following lists the criteria:

Inheritance Relationship to the RTSJ This criterion is chosen to highlight the different approaches taken by the profiles with respect to their relation to the RTSJ.

Task Implementation The profiles implement the notion of tasks differently. Therefore, we want to discuss the solution we find most attractive.

Memory Implementation In the RTSJ, memory allocation must be handled specially. Therefore, this criterion is used to evaluate our experiences of handling memory, and furthermore how this differs in the profiles.

Mission Concept Two of the profiles explicitly introduce the mission concept, whereas the third only implicitly does. It is therefore a criterion worth discussing.

Expressiveness It is important that the profiles allow a sufficient amount of constructions, such that all necessary applications can be expressed intuitively. Otherwise the profile might be too restrictive, and hence limit the programmer's expressiveness.

Written Documentation To develop an application in a real-time Java profile, it is important that there exists documentation to describe how to apply the profile. Therefore, we find this criterion important.

Verbosity If an application gets too verbose as a consequence of using one of the profiles, it might decrease the read- and write-ability. It is a balance of having enough information in the code to make it understandable without bloating it with unnecessary information.

Scores from one to three, where three is best, are given to the profiles for each criterion to emphasise their differences.

## 13.2 Inheritance Relationship to the RTSJ

Both the SCJ and the RJ use *inheritance for limitation* to subset RTSJ. That is, they both inherit from the RTSJ classes and specify which portions of the RTSJ that should not be used, either by overwriting methods, removing constructors, or by using annotations.

In contrast to SCJ's and RJ's approach, the PJ profile uses *inheritance for specialisation* of the RTSJ. The result of this is a core of functionality from which the RTSJ hypothetically can inherit in order to provide the full RTSJ specification.

In our opinion, the approach taken by the PJ profile results in a cleaner class structure that avoids the need to annotate parts of the RTSJ to restrict it. Furthermore, *inheritance for specialisation* seems as a more intuitive inheritance approach than the *inheritance for limitations*. The approach taken by SCJ and RJ makes sense with respect to the existence of RTSJ, since this makes it difficult to let it be a specialisation of another profile. However, not taking this into account, it is our opinion that the PJ profile has the best relation to the RTSJ. One of the authors behind the SCJ profile actually says at the JTRES 2010 conference that the primary reason for using the *inheritance for specialisation* approach was due to political reasons, that is, that the RTSJ already exists (Wellings, 2010).

Based on the above, the PJ profile is given three points and the SCJ and RJ profiles are given one point each.

## 13.3 Task Implementation

As outlined previously, tasks are either implemented by means of using a thread or an event handler paradigm. To recapitulate, RJ uses the thread paradigm for periodic tasks and the event handler paradigm for sporadic tasks. On the other hand, PJ and SCJ solely base tasks on the event handler paradigm that in our opinion is an advantage because the number of concepts is reduced.

Furthermore, Java threads offer much functionality which is not used in relation to real-time tasks. Some of the functionality is even unwanted, such as conditional waits, since it might be difficult or even impossible to statically analyse. Thus, they place more responsibility on the programmer for correctly using the thread paradigm for implementing real-time tasks. On the other hand, basing the real-time tasks on the event handler paradigm is, in our opinion, a better model, because unwanted functionality present in the Java threads is not included.

In relation to the real-time task implementation, another important aspect is how the profiles incorporate the notion of periodic, sporadic, and aperiodic tasks. RJ supports periodic and sporadic tasks whereas PJ and SCJ support periodic and aperiodic tasks. These two approaches differ somewhat and both result in cases where some tasks cannot be expressed using these profiles. We think that this is a problem, since these types of tasks could often occur in such systems for which the profiles have no support of handling.

The absence of sporadic tasks in SCJ is based on the rationale that managing them, requires implementing a monitor mechanism which ensures that the minimum inter-arrival time of sporadic tasks is upheld. This monitor mechanism introduces significant complexity to the real-time system in terms of certification according to standards (JSR302, 2010). The PJ further backs up this decision of omitting sporadic tasks by questioning whether these tasks are statically checkable. The problem is that the possible external events firing the sporadic task must be assured to occur according to the minimum inter-arrival time which is described as a problem because of unpredictability of environmental events Bøgholm et al. (2009).

In some situations it should be possible to deduce whether the events happen with a minimum inter-arrival time. As a concrete example, the mine pump contains sporadic tasks whose minimum inter-arrival times can be guaranteed due to their release patterns. In this case, having sporadic tasks would have been convenient because these model the behaviour intuitively even though the solution of using aperiodic tasks result in the same behaviour due to the release patterns being similar. The non-intuitive way of using aperiodic tasks results in confusion since the literature is generally consistent in their descriptions of aperiodic tasks' timings being unpredictable.

The absence of aperiodic tasks in RJ builds on the rationale that constructing a schedule prior to execution of the system cannot guarantee successful completion of the aperiodic tasks (Søndergaard et al., 2006). However, we think that there exist events whose release patterns are aperiodic by nature. For instance, consider a naturally occurring phenomenon such as aperiodic oscillation of noise. In this case, if the system includes such phenomena, they should intuitively be modelled by the task type representing the release pattern of the event. However, in RJ, the engineers should seek another way of incorporating the phenomenon whose nature is not aperiodic. We think this reduces the degree to which the system can model real-world events. Hence, the real-time Java profile should not make restrictions in this regard.

On the basis of support for event types, all of the profiles are equal since they all have reduced notions of natural occurring events. Combined with the task paradigms where RJ uses both the thread and event paradigms, a score of two is given to PJ and SCJ, while a score of one is given to RJ.

## 13.4 Memory Implementation

With regard to the memory management used in the different profiles, a number of concerns have been raised. Mainly, the need to explicitly declare memory area types with corresponding sizes, results in added difficulty for the programmer. A memory model where this would be unnecessary, such as if a real-time compatible garbage collector is used would have been preferred. It can be argued that the process of calculating the total memory size used by one part of the program and declaring it in another part, e.g. as with scoped memory, potentially introduces errors. A likely scenario could be that the programmer adds a new object or modifies the data stored in an object, without changing all related memory allocations thereby introducing inconsistency. Therefore, it would be preferable if the compiler could deduce the actual memory usage and maintain memory allocations.

The latest draft of SCJ mentions a new `StorageParameters` which requires the programmer to specify the memory usage with further details such as stack size (JSR302, 2010). Evidently, this adds more concerns to the programmer and, hence, is not a solution we prefer. Again, if possible, this could be omitted if memory usage could deduced statically.

The memory model used by the three profiles is similar, with the only major difference being the PJ profile which does not contain the notion of immortal memory since scoped memory combined with missions being first class event handlers function similarly. This reduces the number of memory related concepts in the profile and is hence more simple reasoning about.

Based on the above, the authors are generally not satisfied with the memory management of the profiles. The PJ profile improves it by reducing the types of memory areas. Therefore, the PJ profile is given a score of two, while the other two profiles are given a score of one.

## 13.5 Mission Concept

The two newer profiles, SCJ and PJ, directly incorporate the mission concept, whereas the older RJ does not. In our opinion, the choice of using the *mission* keyword is more intuitive since this is used extensively in the literature to describe a set of tasks working on the same overall task.

In respect to SCJ, the mission concept seems complicated by the introduction of the `MissionSequencer`. Its purpose is to allow multiple missions to be run in parallel or in sequence. However, in our opinion, this is more elegantly modelled in the PJ profile where missions can be nested, thereby e.g. having a master mission containing other missions. The `MissionSequencer` is especially inconvenient in applications where only one mission is needed, since an instance of it is still required.

In respect to the PJ profile, the usage of ***`handleAsyncEvent()`*** results in some ambiguity. In the periodic and aperiodic tasks, the execution phase code is placed inside the event handler, but in respect to the missions, the logic to be executed during the mission termination phase is placed in the event handler. It is of course a matter of knowing this difference, but we experienced that the design decision is confusing.

Based on these observations, we find PJ being closest to what we want from an introduction of the mission concept. Furthermore, we do not see the confusion regarding the ***`handleAsyncEvent()`*** as a problem compared to that of SCJ's `MissionSequencer`. The scores are therefore, three to PJ, two to SCJ, and one to RJ.

## 13.6 Expressiveness

In the constrained mine pump example, we found the expressiveness of all profiles sufficient. However, we observed two profile design decisions that could spur implications in the development.

First of all, we observed that the RJ profile only allows a preemptive priority-based scheduler (Kwon et al., 2002). In our example, and probably many others, this is sufficient. However, one could imagine situations where other scheduling policies advantageously could be used. Both PJ and SCJ allow the programmer to implement a scheduler of choice. In SCJ, the support for schedulers depends on the used compliance level. However, at level one and two there are basically no restrictions.

Secondly, only SCJ uses the notion of compliance levels. It is argued that the usage of levels is to accommodate the necessity of certifying safety-critical applications. Since this is a costly process, it is reasonable to reduce the costs by allowing the development of simpler applications to be made under more minimalistic conditions. Following this philosophy, compliance levels are considered as useful.

According to the above argumentation, we evaluate SCJ as being the best profile, followed by PJ, and finally RJ. Therefore they receive the scores of three, two, and one, respectively.

## 13.7 Written Documentation

We observed that the profiles lack documentation. That is, it is not possible to find full APIs for any of them. Only fragments published in various papers are available and it is our experience that these fragments only are sufficient for the simplest programming. As a concrete example, the RJ `SporadicParameters` class requires a parameter to specify a buffer, but it is not documented anywhere to our knowledge. Furthermore, due to the current specification process of SCJ, the profile exists in different versions in different publications. This is confusing since some of the versions allow certain properties that others do not. Furthermore, the publications do not make explicit which version of the SCJ they are documenting.

Due to the lack of documentation, we have leaned on smaller code examples for each profile found in the publications and in the code library of the JOP project and a code library of the PJ provided by one of its authors. The problem with this approach is that the code examples in the publications are small and do not explore the entire profiles. Furthermore, we have observed that the JOP project does not implement the actual profiles in their entirety, hence the reliability of the more extensive code examples of the JOP project is modest. Since we have not been able to check our implementation using the SCJ by running the code on a platform, we cannot assure that the implementation in fact is correct. In addition, a static code analysis tool capable of determining profile compliance, has not been available.

Based on these observations, our opinion is that a more detailed documentation for each profile is required before they can be used in mainstream safety-critical application development. Therefore, we evaluate all profiles to get one as score.

## 13.8 Verbosity

The three profiles contain a varying degree of verbosity in terms of amount of code that needs to be written and in terms of the details that need to be specified. Verbosity can be of concern if it introduces additional considerations for the programmer. Instead, one could imagine that the responsibility instead is put on the particular profile to reason about and derive the necessary details automatically.

In particular, the main amount of verbosity comes from the constructors used to create new real-time tasks. Common to the three profiles is the usage of the `PriorityParameters` class which is used to denote the particular priority of a task. This class originates from the RTSJ, and it is likely because of this that it is included in all the profiles. We argue that omitting the priority parameter would increase the abstraction level by moving the responsibility of determining the priorities if necessary and choose an appropriate scheduling policy to the profile. What may seem counter-intuitive is that the `PriorityParameters` class is essentially a wrapper class for a simple integer representing the priority. The only instance methods on the class are a getter and a setter. Hence, if one is interested in changing the priority dynamically, the `PriorityParameters` instance given to the task needs to be consulted. In our opinion, this seems counter-intuitive because changing the priority of a task does not involve consulting the instance of the task but instead the instance representing the priority. If priorities are needed, we think that a better approach is to omit the class and add priority-related getters and setters to the tasks and provide an enumerated type to the constructor representing the priority. However, the current usage of a class instead of an enumerated type might be excused due to the reason that when the RTSJ was defined in 2001, enumerated types had not yet been introduced in Java.

Another thing we have observed is that only the PJ profile can specify the scheduler used for each task. In PJ, a reference to the scheduler is provided as argument to the real-time tasks. This creates more verbosity for PJ but in contrast to the `PriorityParameters` class, this feature adds value to the PJ in terms of enhancing the degree to which the programmer can express the precise behaviour of the system. However, the slight verbosity introduced by the PJ could be circumvented by making a flexible solution where an overloaded constructor excludes the scheduler parameter indicating that it should rely on a default scheduler.

Regarding the constructors of real-time tasks, the PJ dictates that an `LTMemory` instance is provided. The `SCJ` instead dictates that an instance of `StorageParameters` is provided containing various arguments related to memory. As previously mentioned, one could imagine that the profile itself is capable of deducing the memory size.

The SCJ, being derived from the RTSJ, dictates the use of annotations of source code that specify whether certain functionality should be allowed to be used for various compliance levels or whether they are allowed at all. In our opinion, this adds significantly to verbosity since the programmer has to annotate the code with these. In cases where the compliance level is not necessary, one could imagine, the programmer not to be forced to write the annotations, thereby easing the implementation.

From this analysis, there is not, in our opinion, a profile that is significantly better than the other in terms of verbosity. The SCJ is different in its use of code annotations and is hence slightly more verbose. Due to this, the PJ and RJ get a score of two while the SCJ receives a score of one.

## 13.9 Comparison

The scores given in the previous sections are summarised in Table 13.1 and contains the total scores of the profiles.

In the comparison, the conclusion is that the PJ profile scores more than the other profiles in a number of criteria. Specifically, its relation to the RTSJ, where the RTSJ inherits from PJ is in our opinion one of its major improvements.

From the table, it can be deduced that one of the most critical evaluation criterion is memory implementation. None of the profiles receive three as score. The improvements we see in future releases revolve around automating the process to eliminate the manual nature. We think that it adds more uncertainty for the programmer when having to reason about the maximum memory consumption of individual tasks.

| Criterion | RJ | SCJ | PJ |
|---|---|---|---|
| Inheritance Relationship to the RTSJ | 1 | 1 | 3 |
| Task Implementation | 1 | 2 | 2 |
| Memory Implementation | 1 | 1 | 2 |
| Mission Concept | 1 | 2 | 3 |
| Expressiveness | 1 | 3 | 2 |
| Written Documentation | 1 | 1 | 1 |
| Verbosity | 2 | 1 | 2 |
| Total | 8 | 11 | 15 |

**Table 13.1:** Summary of scores given to each of the criteria.

Should the profiles enter industrial use, it is evident, that we see a significant improvement in documentation. As of this writing, none of the profiles contain any complete API reference and fragmented documentation obtained from independent sources may even be conflicting. We see it as important that documentation is provided especially in the domain of safety-critical applications development where a precise description of functionality is paramount to avoid critical errors to occur.

# 14 Conclusion

Real-time systems depend on timely reactions to events. This especially applies for hard real-time systems where missed deadlines can result in catastrophic failures. Therefore, much research has been conducted to ensure both the functional and timing correctness of such systems, spanning a wide variety of topics such as programming languages, WCET analysis and schedulability analysis.

Through the study of real-time systems in general, we encountered several interesting problems. Of these, we found the problem of developing real-time embedded systems in the Java programming language the most interesting. The incentive for using Java is that we identified that real-time embedded systems are difficult to develop, and that high-level programming languages hold great potential for easing this task. This problem was further studied through both analysing the RTSJ and various real-time Java profiles. However, using Java for real-time embedded system development is not yet standardised since the community is disagreeing on the approach. The RTSJ, being the initial effort in applying real-time concepts and language constructs for real-time systems development, is by many considered inappropriate. This has formed the emergence of a variety of Java real-time profiles aiming at this. Of these profiles we have analysed the RJ, SCJ, and PJ. Through this analysis, we gained theoretical knowledge in the profiles which prepared us to practically apply them in an example. From this we gained experience in both developing a real-time system and in understanding the profiles better. This allowed us to evaluate the different real-time Java profiles, and to provide our opinion on how a real-time Java profile should be designed. *This fulfils learning goals 1.a, 1.b, 1.c, 2.a, and 3.a described in Section 1.1*

In the comparison we found the PJ profile most appropriate. We specifically like its inheritance relationship to the RTSJ where the RTSJ inherits for specialisation. In our opinion, PJ is not the perfect real-time Java profile. Common for them all is a number of possible improvements. For instance, their memory models are very difficult to use and to safely maintain throughout the development process. Furthermore all of the profiles lack documentation, since only partial APIs are documented.

As the example implementation, we outlined a constrained version of the well-known textbook example of the mine pump real-time system. The physical model of the mine pump was constructed using LEGO and the JOP. Through the development we gained experience in understanding some of the challenges encountered in such systems. Implementing a specific functionality must be carefully considered for possible alternatives because relying on simple implementations may yield a substantial overhead that effectively increases the WCET with several orders of magnitude. In our case, we identified that using `equals()` for comparing two objects significantly impacted the WCET. The problem was solved by replacing it with a simple numeric comparison. *This fulfils learning goal 4.a.*

Both WCET analysis and schedulability analysis have been conducted on the mine pump implementation. WCET analysis was conducted using the WCA tool, revealing that the tool is not capable of determining loop bounds by using DFA. Instead we had to resort to manually specifying loop bounds for both our own code and specific elements of the JVM. Furthermore, we observed that the approach taken by WCA tool when conducting model-based WCET analysis can be improved. The reason for this is that it is based on the UPPAAL model-checker which in its latest development release supports a *sup query* that can be used to avoid using a binary search to converge the WCET to a safe and more precise estimate. Furthermore, we gained experience in schedulability analysis by manually conducting utilisation-based schedulability analysis and RTA, and by automatically conducting a model-based schedulability analysis using the TIMES tool. In this regard, we compared the WCRTs determined by the TIMES tool with those manually determined through RTA. The WCRTs were, as expected, more precise using the model-based schedulability analysis of TIMES. In respect to the manually conducted schedulability analyses made us explicitly reason about how the concepts are interrelated. *This fulfils learning goal 4.b.*

In respect to WCET analysis, it is our opinion that a major drawback for Java real-time systems development, is that only special-purpose processors, such as the JOP, allow Java programs to be executed in a timing predictable manner. Hence, hardware such as the more widespread ARM processors has not to our knowledge been shown able to execute Java in a timing predictable manner. Allowing Java to be executed in a timing predictable manner on these more common processors would substantially widen the potential of Java to possibly become the new preferred programming language for real-time systems development. *This fulfils learning goals 2.b and 4.b.*

On the basis of the above, we conclude that Java is still not sufficiently mature to become the standardised approach for real-time embedded systems development. Much discussion revolves around how the approach should be designed and identifying the key concepts and principles on which the approach must build.

## 14.1 Future Directions

The following presents and discusses a number of possible future directions which could form a continuation of this project. *Fulfilling learning goal 1.d.*

Java on Another Platform  Even though Java real-time systems can be executed in a timing predictable fashion on processors such as the JOP, it would be a major progress for real-time Java if the applications could be executed on more common processors such as those in the ARM family. The challenge of this, is to predict the behaviour of the interaction between the JVM, RTOS, and the hardware.

**Static Compilation** Another approach to allow real-time Java to be executed in a timing predictable fashion on other architectures is to make use of static compiling, that is, e.g. by compiling Java to C which again can be compiled to machine code. The advantage of this approach would be that WCET and schedulability analysis from C can be reused.

**New Real-Time Java Profile** The extensive analysis of the RTSJ and the RJ, SCJ, and PJ profiles provide a solid foundation for constructing a new profile that among others take into account solutions to the identified issues for the existing profiles. The profile could either be completely new or take starting point in one of the existing profiles.

**Map the Ideas of Real-time Java to the Common Language Infrastructure (CLI)** While we have focused on Java for real-time systems in this project, another interesting approach would be to introduce real-time capabilities to the CLI implemented by the .NET Framework and Mono, for instance. This would enable languages such as C# as an alternative to Java on such systems. Similar to Java, C# is a popular programming language, hence its usage in real-time systems holds great potential since many programmers are familiar with it.

**Implement One of the Real-Time Java Profiles on the JOP** During our implementation of the LEGO mine pump, we discovered that the JOP does not fully implement any of the real-time Java profiles. It would therefore be interesting to make a full implementation of one of these. More specifically, this implementation could be of the newest SCJ or the PJ profiles.

**Timing Predictable Just-in-time Compilation** Many of the current JVMs and the CLR incorporate JIT compilation for machine code generation on the particular platform. Using JIT compilation has some interesting benefits in contrast to static or ahead-of-time compilation. Among others, runtime optimisation techniques can be performed by e.g. monitoring execution patterns. In this relation, an issue is how to make JIT compilation timing predictable due to the dynamic nature of applying various optimisations at runtime.

**Developing a Compliance Tool for a Real-Time Java Profile** Certifying safety-critical applications is often a necessity for deployment in industrial settings. This certification process relies on ensuring that only a specific subset of provided functionality in a programming language, such as Java, is used. The SCJ relies on code annotations to specify which functionality is allowed at a particular compliance level. An alternative approach could be to develop a static analysis tool that exercises the application exhaustively to verify that the program indeed complies with a certain set of constraints.

**Make the WCA Tool Generic** Currently the WCA tool is tightly coupled with the JOP platform. It would be desirable to introduce a layer of indirection in the WCA such that it can ported to other platforms running real-time Java. This layer could e.g. take VHDL files as input and thereby adjust the tool accordingly.

Find an Approach to Determine the Worst Case Memory Consumption of a Task We observed that using programmer annotated memory consumption in the different real-time Java profiles that it would be desirable to remove these. An approach to remove these could be to determine the memory consumption at compile time, with possible warnings if the memory usage could result in run-time errors. This would involve some level of static program analysis to determine memory bounds. Some existing research in this field by Chin et al. (2005) creates memory bounds for methods depending on their input, and Albert et al. (2007); Cachera et al. (2005) have provided frameworks for detecting unbounded memory usage.

Modular WCET and Schedulability Analysis The approach used to conduct WCET analysis using the WCA tool, and approaches suggested by the theory presented in the report, assume that the program and all related tools are included in the actual analysis. In order to improve the speed of the analysis, a pre-made WCET estimate could be provided for used libraries, reducing the task of conducting WCET analysis to the program itself.

Unchecked Exceptions Not directly related to real-time systems, an interesting problem would be to look into unchecked exceptions and how they affect the functional correctness of the program. Specifically, we encountered unchecked exceptions as part of our implementation, in which we potentially could have introduced a fatal error into our program. Correctly informing a developer of possible errors introduced by unchecked exceptions would be a useful feature.

# Part V

# Appendices

# Appendix Contents

# A Memory Management

Most embedded systems have a relatively limited memory pool in contrast to other systems, hence, one should be aware of their memory consumption. This is especially of concern since memory depletion affects timing predictability which consequently may lead to missed deadlines. Furthermore, memory allocation in real-time systems needs to be deterministic with respect to time. Especially in hard real-time systems, it is necessary to know the allocation time such that it can be ensured that a task's deadline is not missed as a consequence of memory allocation. A concrete example of the difficulty of having timing predictable memory allocations is the problem of memory fragmentation since this can result in additional time spent for allocating memory at runtime (Qing Li, 2003). Different solutions exist which aims at providing memory management facilities. For instance, this can be done by removing the possibility of allocating memory in time-critical tasks (Dibble, 2002).

In the following, two different concerns are emphasised. These are memory allocation and memory deallocation using garbage collection.

## A.1 Memory Allocation

Memory can generally be allocated using either stack-based allocation or heap-based allocation. These are described in the following:

Stack-Based Memory Allocation In terms of real-time systems, it may be important to analyse the stack size during execution to avoid stack overflows occurring or that the usage of memory exceeds what is available. Burns and Wellings (2009) suggest techniques analogous to those used for WCET analysis to determine bounds for the stack usage. DFA which examines the allocations and deallocations can be used for this.

AbsInt's StackAnalyzer (AbsInt, 2010) is a tool that is capable of automatically exercising the program for stack usage. The tool can be useful in a real-time setting because the programmer is sometimes responsible for specifying the memory that is used by the stack (JSR302, 2010). The StackAnalyzer tool can in this regard prove helpful to verify that stack overflows do not occur and, in case the programmer has over-estimated the stack size needed, the tool will report this thereby enabling the programmer to tighten the stack size to reduce memory waste.

Heap-Based Memory Allocation In Java, memory is allocated on the heap whenever objects are created from class definitions. In the context of real-time systems, the heap-size and timeliness of heap-allocations are problematic. Similar to the stack-based memory allocations, statically determining the heap-size is needed in order to ensure that the available memory is sufficient. Furthermore, the allocation time on the heap can be unpredictable due to memory fragmentation.(Wellings, 2004)

The research project conducted by Craciunas et al. (2008) has shown that dynamic memory with predictable memory fragmentation and response times that are constant or linear in the size of the request is possible. Specifically, this is achieved by implementing a real-time memory management system used for explicit memory management. Finally, it has been shown that a static analysis method in the form of DFA that exercises the program source code for memory allocations, is capable of determining the Worst Case Heap Allocations (WCHA) of the program. In certain cases, memory allocations may be dependent on the execution of the system such as when the size of an array depends on program input. Under such circumstances, the tool conducting WCHA relies on programmer annotations specifying the maximum memory allocation size.(Puffitsch et al., 2010)

## A.2 Memory Deallocation using Garbage Collection

This section emphasises on the garbage collection ideas of Java. However, historically, Java is far from the pioneer in incorporating garbage collection ideas into the language. LISP being one of the earliest programming language also has garbage collection (Dibble, 2002).

Garbage collection can be described as the process of identifying unused objects and reclaim the memory they consume to allow new allocation requests to use that memory. With regard to the JVM, the garbage collection process will be instantiated whenever it assumes the system is idle, when the given program explicitly requests for garbage collection to be performed, and in the case there is not sufficient amount of free memory available to satisfy memory requests.(Dibble, 2002)

Care must be taken to ensure that the behaviour of the garbage collector does not affect the real-time system, especially with respect to introducing unpredictable timings for the running tasks. Specifically, it is difficult to predict when the garbage collector frees memory and how much time the process can delay an otherwise critical task. Therefore, efforts have been put in developing a garbage collector that upholds real-time properties, that is, it is timing predictable. Knuth (1997) was one of the first to describe the problem of combining real-time systems and real-time garbage collection, but an actual implementation was not made. This has instead been made by Bacon et al. (2003). Not only is it claimed to have real-time properties, but it does also feature modest memory overheads which is a desirable property of embedded systems.

The JOP project has implemented a real-time garbage collector for the JOP. It is implemented as a thread running concurrently with the application. To ensure that the thread does not negatively impact the real-time tasks, the garbage collector thread is given the lowest possible priority. Furthermore, its blocking time has been reduced to $54\mu s$. It must be remarked that Schoeberl (2009) does not encourage the garbage collector to be used on larger systems, since it requires more analysis to determine if it is timing predictable.(Schoeberl, 2009)

# B
# Real-Time Systems Development Method

A defined development method allows for structuring and controlling the development process and for avoiding certain common pitfalls encountered in software development (Sommerville, 1999). This applies to the development of many different types of systems. However, these systems introduce additional elements which need to be taken into account, such as task deadlines and high coupling with the production environment.(Selic, 1998)

There exist a number of development methods tailored for real-time embedded systems, which try to solve the specialised problems inherent in such systems. One example is HRT-HOOD (Burns and Wellings, 1994), that focuses on embedded hard real-time systems, and how these are to be developed in an environment with both software developers and hardware engineers. General for these systems is that there exists an added complexity if the development project covers the development of some hardware device and interaction with hardware engineers. This interaction has been found to complicate the development significantly, and in some cases the software engineers are entirely left out of the design phase potentially resulting in software being used incorrectly on such systems (Graaf et al., 2003).

A central part of developing real-time systems is the requirements specification. Informal requirements can be stated in natural language, and the benefit is that they are easily read by many of the people involved in the development process. However, due to the ambiguous nature of these, they may be insufficient for real-time systems, especially safety-critical systems where an unanticipated error or action may risk human lives.

The formal notations define more precisely and unambiguously the requirements of a system, given that the requirements were comprehended and defined correctly by their author. Because of the possibility to create software which, given a formal specification, can show different properties of the specification, effort has been put into increasing their use in the industry (Burns and Wellings, 2009). An example is to write the formal specification as a model in the model-checker tool UPPAAL (Bengtsson et al., 1996), which allows for defining a sequence of events in a system taking time into account. Afterwards, the model can be verified against properties that must be upheld. One problem with the formal specifications is their complexity that may be incomprehensible for people without the necessary training in the notation.(Graaf et al., 2003)

# C Development Method

The development method adopted, is internally known as the tailored Scrum method. The adjective *tailored* in this context refers to alterations of the original Scrum (Larman, 2003) method in terms of various practices and work products which during the semester project have proven useful and have had a significant improvement in our development process. Specifically, the tailored Scrum method draws experience from the incremental method (Sommerville, 1999), Extreme Programming (XP) (Larman, 2003), and Scrum. Thereby having an agile nature.

To accommodate the analytical nature of the project and relative small development task a number of previously used work products have been omitted. Specifically, the *Application and Problem Domain* work product known from Object-Oriented Design and Analysis (Munk-Madsen et al., 2000) did not make sense. This work product mainly targets at establishing a common understanding among team members and possible stakeholders about what main components are present in the application and establishing knowledge about the people that are affected by the application.

Following list gives an overview and accompanying description of the major components comprising the tailored Scrum method.

**Backlogs** This work product has been adopted from Scrum and constitutes one of the cornerstones in this method. In Scrum, there exist two backlogs, namely: product backlog, and sprint backlog. The product backlog contains all the features of the project such as user stories that must be conducted during the project. Hence, completing the features in the product backlog optimally completes the project. The sprint backlog, on the other hand, comprises all the features that must be completed during the particular sprint period. The features included in the sprint backlog are selected by the development team and potential stakeholders. In our case, features have been selected according to dependency to other features, importance, and priority.(Larman, 2003)

Using the backlogs allow us to be aware of the remaining tasks, both for the sprint and for the whole project. Furthermore, if working on a task generates new ideas to new tasks, these can be written in the product backlog such that they are remembered when planning the next sprint.

Iterative Sprints Sprints are one of the main components in Scrum. The sprints are essentially small periods with constant length. The entire project period is split into a number of these and before each beginning of a sprint, a subset of features are selected from the product backlog to the sprint backlog. The number of features that can be selected must in total make up the development time available for the team.(Larman, 2003)

We have experienced that sprints of two weeks fit well to our needs of being able to relatively quickly adapt to changes in the requirements, adapt to additional work load generated by meetings, and more quickly detect potential time slippages.

Burn Down Chart The burn down chart is a work product adopted from Scrum and is essentially a way to determine the progress of the current sprint. Whenever a feature of the sprint backlog is done or sufficient work has been conducted to remove the remaining effort estimate, the burn down chart is updated accordingly to reflect this change in the current status. Hence, the burn down chart showing the remaining effort in the current sprint provides a great overview of the overall status.(Larman, 2003)

We use the burn down chart to enhance the motivation, since completed effort is visualised thereby providing a good overview of progress. Furthermore, it is our experience that the teamwork is strengthened because it reminds us that we are working on a common goal that is achieved when the burn down chart reaches zero.

User Stories The user story work product is inspired from Extreme Programming. The Scrum backlogs do not dictate how features are represented making the decision dependent on what the team finds most useful. Since user stories have proven useful according to previous semester projects, it is used again in the current project. A user story is a simple sentence denoting the task, which could be a feature in the system or a report-related task for instance, that needs to be conducted (Sw701b, 2009; Sw801b, 2009). The sentence should be constructed in such a way that when consulting the user story at some later point, the sentence should remind the team of the conversation that led to the construction of the user story. Besides the sentence, a user story also contains an estimate that denotes the development effort that has been estimated according to the planning game activity in our case.(Cohn, 2004)

It is common practice that our user stories contain little information. In some cases it might even be reduced to a single word. However, we have not had any problems to refresh what the user stories covers. Also, the fact that the user stories are rather informal, and shortly written, gives motivation to practically use them because they are not connected with great effort to construct.

Planning Game This activity is adopted from Extreme Programming. As previously outlined, Scrum encourages estimating the features to provide a means for selecting the subset of features used in the sprints and for monitoring the progress of the project (Larman, 2003). The planning game in this case has proven useful for making the estimates in an agile way. The specific planning game that has been used during this project is based on the Poker Planning Game which is a time estimation technique that allows the entire team to not only estimate the task but in addition also provide a good insight to the definition of the task, that is, what it specifically comprises. The Poker Planning Game works by letting each developer initially come up with an estimate of the user story. Afterwards, the team members compare their estimate with each other. Given that the individual estimates differ, the team members with the lowest and highest estimate explain to the remaining team members how they derived it. In this way, a good foundation for a discussion about the user story is made, thereby allowing the team members to come up with different interpretations of the task that, hopefully, leads to a common understanding. After the discussion, each team member perform a new estimate. The process is repeated until the individual estimates are approximately equal.(Atira, 2009)

The prior experiences drawn from the poker planning game are positive. It has successfully led to important discussions that have uncovered essential details that could have been crucial if discovered at a late point in the development process. Also, being a rather informal an agile estimation technique, provides great motivation of using it considering that the outcome is very rewarding.

Comprehensive Documentation This activity is not based on any specific development method. In fact, Scrum, being the overall development method of this project, discourages the use of documentation that does not serve any significant importance to the project. However, it must be noted that Scrum does not dictate a concrete amount of documentation. Instead, according to its ceremony scale, it dictates that documentation should be made to the extent of which is sufficient. Due to this project having an academic nature, and due to this project mainly focusing on being analytical, comprehensive documentation must be made in order to fulfil the requirements of the study regulation.

# D Real-Time Programming Language Design Criteria

Generally, a real-time programming language desirably reflects six different language design criteria.

Security This criterion is the measure used to evaluate the extent to which the programming language automatically detects errors introduced by the programmer e.g. at compile time. Given a program written in such a language, similar checks need not be performed during run-time thereby reducing overhead. Another benefit is that errors are revealed earlier in the development process thereby reducing the overall cost. Errors that are introduced as part of logic that has been incorrectly implemented are hard to detect by these means. To accommodate this issue, another approach is that the programming language offers readability and is well-structured.

Readability The readability criterion is also to some extent related to better security and better maintainability. Additionally, readability may reduce the documentation efforts. Readability is difficult to define but revolves around elements such as appropriate use of keywords, possibility of defining types, and how easily code is modularised to separate concerns. Keywords should have meaningful names that avoid ambiguity and promote self-documenting code. In fact, a goal is to provide a notation that is sufficiently clear to let the programmer assimilate knowledge about the operation of the program without resorting to subsidiary documentation.

Flexibility This criterion describes the extent to which the programming language provides sufficient mechanisms and constructs to let the programmer express functionality in an intuitive and straightforward way. Hence, this criterion is to some extent related to readability because the language should not require the programmer to resort to complex solutions possibly requiring system commands or machine code inserts to obtain the desired result.

Simplicity This language design criterion is used to evaluate the programming language in terms of minimising the amount of language features that need to be used and also the precision of these. A programming language exhibiting high simplicity is likely to minimise the effort of compiler construction for that language and reduce the cost associated with training programmers to use the language. Also, as a consequence of being precise, it diminishes the likeliness of introducing programming errors as a consequence of misinterpreted language constructs and features.

Portability Portability is the language design criterion describing the extent to which programs written in the particular language are easily transferred to other platforms. Portability is a difficult criterion to obtain in real-time programming languages because real-time systems are often highly dependent on hardware resources. Nonetheless, portability can be improved by at least allowing for isolation of machine-dependent parts such that these are separated from the portable, or machine-independent parts, of the program. This allows for mainly focusing on modifying or rewriting the machine-dependent part in case the program is transferred to another platform.

Efficiency This criterion is used to describe the extent to which a program written in the particular language allows for efficient execution. It is important to note that efficiency must not introduce unpredictability in terms of both time and memory consumption. Furthermore, efficiency should not degrade the readability and security of the program.

E

# Example of the Implicit Path Enumeration Technique

This is an example of how a WCET analysis using the IPET technique is conducted. The example follows these steps:

1. Create a CFG of the source code.
2. Derive the summation containing execution cost and frequency for each basic block that must be maximised.
3. Derive linear expression constraints from the CFG.
4. Provide the linear expression and constraints to an ILP solver.

The code to be analysed is shown in Listing E.1.

```
public int ipetExample(int i){
        if(i==5)
            return i;
        else
            return i++;
}
```

**Listing E.1:** Java code of the *ipetExample()* method.

As shown, if the given input $i$ equals five, the method returns true and otherwise it returns false. Hence the code contains two branches.

The initial step of the WCET analysis is to create a CFG of the source code. A static code analysis tool can be used for this such as WCA. The result of this step is depicted in Figure E.1.

As shown, $B_0$, $B_1$, and $B_2$ are basic blocks annotated with their respective execution costs $C_0$, $C_1$, and $C_2$. The annotations $d_0, ..., d_4$ indicate the execution frequency of their respective edges.

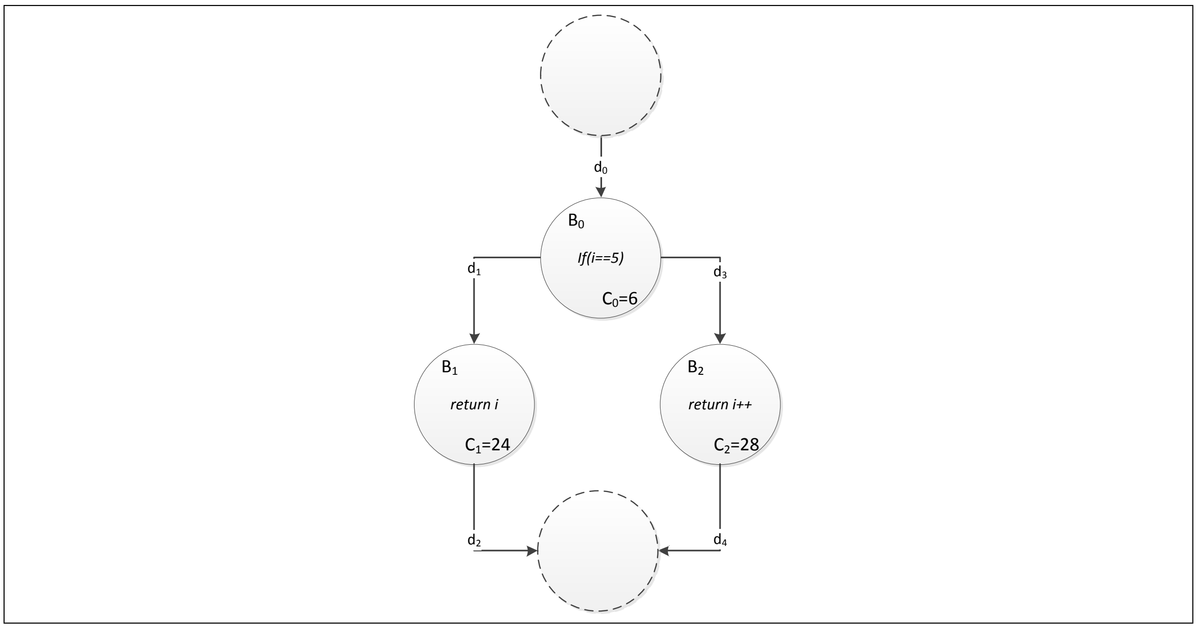


**Figure E.1:** CFG of the code shown in Listing E.1.

In Chapter 4, it is described that the execution frequency of a basic block is defined as being equal to the number of incoming edges, which again is equal to the number of outgoing edges. The linear expression to be maximised, to determine the WCET of the ***ipetExample()*** is therefore:

$$max(6 \cdot d_0 + 24 \cdot d_1 + 28 \cdot d_3).$$

Furthermore, a set of constraints can be derived using the theory of IPET constraints. The constraints for the ***ipetExample()*** method are:

$$\begin{aligned} d_1 &= d_2 \\ d_3 &= d_4 \\ d_0 &= d_1 + d_3 \\ d_0 &= 1 \\ d_2 + d_4 &= 1 \end{aligned}$$

The constraints should be self explanatory.

Providing the linear expression to be maximised and the constraints to an ILP solver application such as *lp_ solve* (Gourvest et al., 2010), the WCET analysis can be conducted. The result of the problem is:

$$WCET_{ipetExample()} = 6 + 28 = 34,$$

which is obtained by the following execution frequencies:

$$\begin{aligned} d_0 &= 1 \\ d_1 &= 0 \\ d_2 &= 0 \\ d_3 &= 1 \\ d_4 &= 1 \end{aligned}$$

This means that executing the basic blocks: $B_0$ and $B_3$, results in the WCET.

F

# Example of Model-Based WCET Analysis

Following is an example of how model-based WCET analysis can be conducted. The analysis uses the WCA tool to derive CFG, WCET, and UPPAAL models. The code of interest is shown in Listing F.1.

```
public int mainMethod(int a, int b){
        int y = f(a,b);
        return c;
}

public int f(int a, int b){
        return 2*a + b;
}
```

**Listing F.1:** Code example used throughout a model-based WCET analysis.

As shown the code consists of to two methods, namely: *mainMethod())* and *f()*. The flow of the code is that *mainMethod()* takes two integer variables as argument which it passes to *f()*. The purpose of *f()* is then to return a value calculated using the two inputs.

Figure F.1 depicts the CFG of the two methods.

The cost for each basic block in the CFG is then used in the model-based WCET analysis. The model for the two methods are therefore as shown in Figures F.2 and F.3.

As shown, the models reflect the CFGs, that is, there is roughly a one to one mapping of CFG basic blocks and UPPAAL locations. Furthermore, as shown, the interaction between the two methods is made by a synchronisation on the *invoke_1* channel.

Observer that the model of *f()*, shown in Figure F.3, contains a loop. This models the fact that the method potentially can be called multiple times whereas the *mainMethod()* only is executed once.

The two templates are instantiated as $M0$ and $M1$ for *mainMethod()* and *f()*, respectively. Issuing the $sup\{M0.E\} : t$ query in UPPAAL, returns the highest value of the clock $t$ at the $E$ location, that is the WCET of the *mainMethod()*. This gives 175, meaning that the WCET is 175 clock cycles.

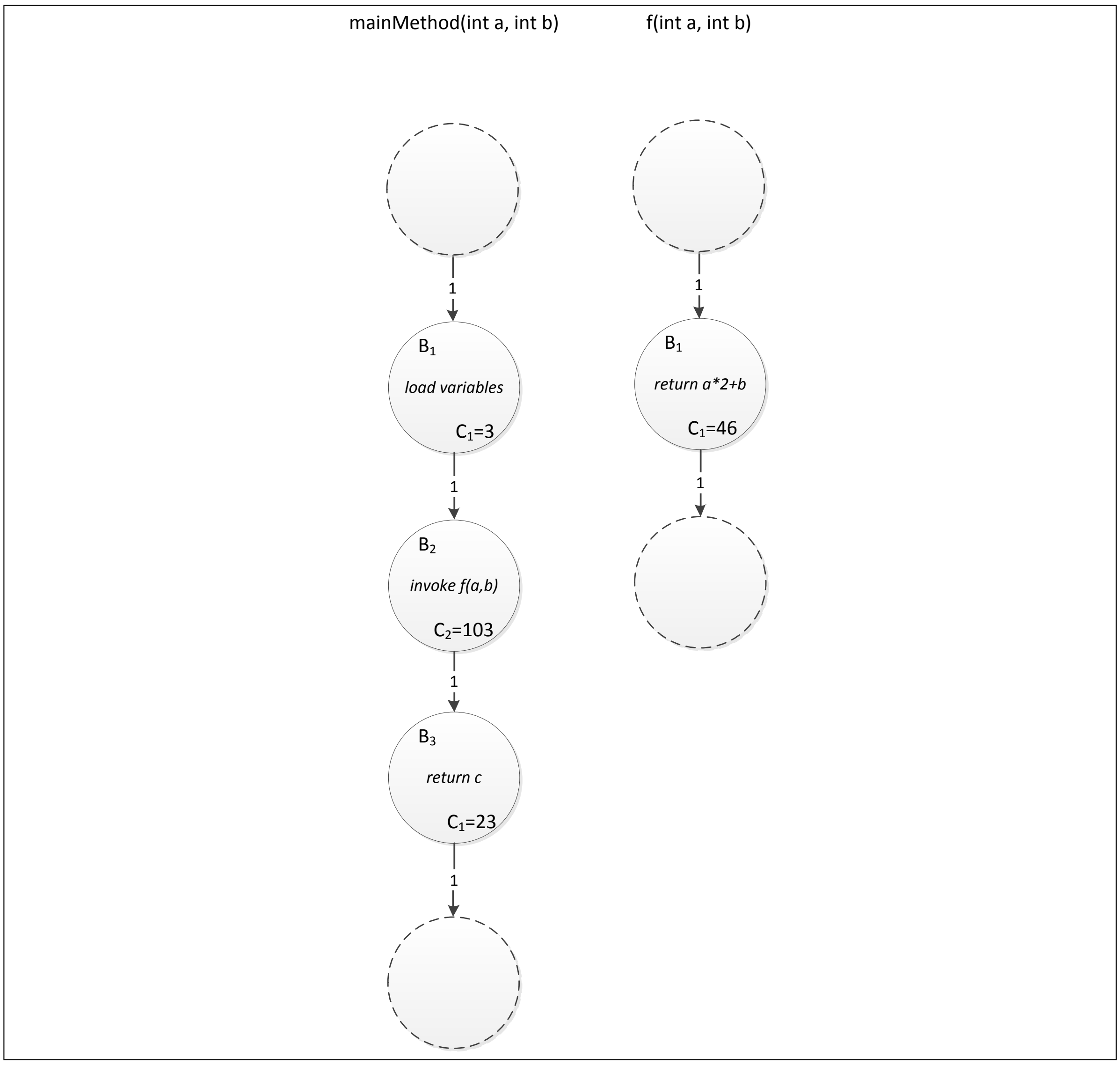


**Figure F.1:** CFGs of method *mainMethod()* and *f()*.

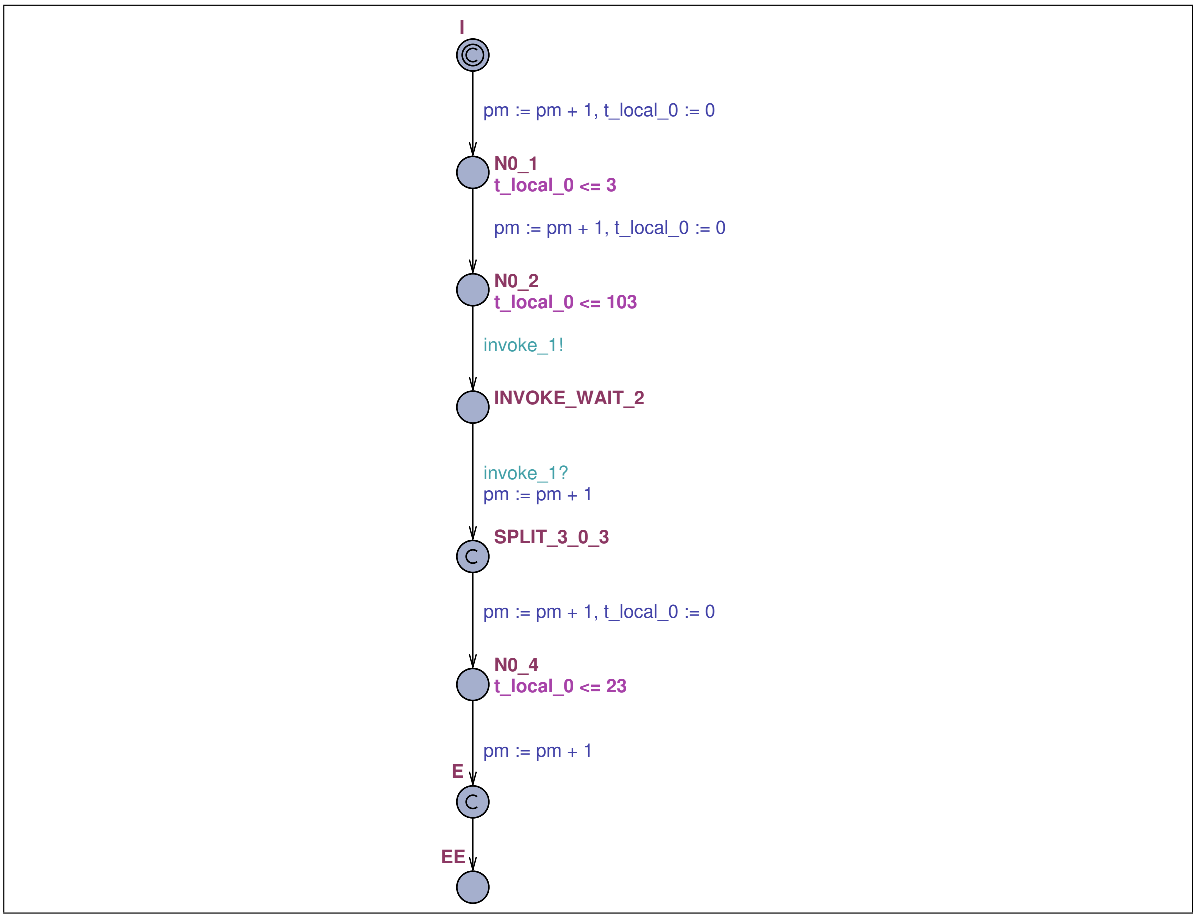


**Figure F.2:** UPPAAL template of the *mainMethod()* method.

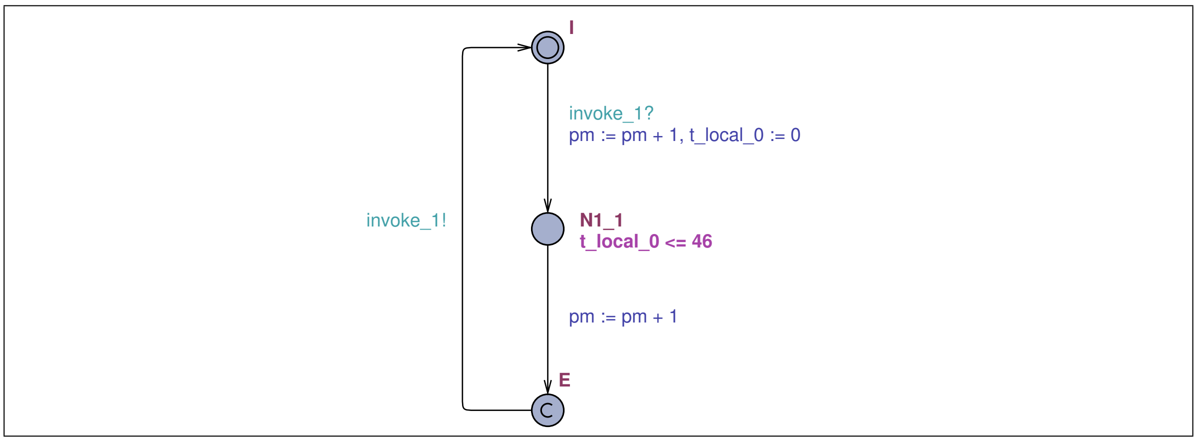


**Figure F.3:** UPPAAL template of the *f()* method.

# G
# Timing Requirements of Tasks

The following describes the chosen timing requirements for the four tasks implementing the mine pump. As described in Section 9.4, the goal is to react to changes in water level and critical methane level within $200ms$. The periods and deadlines must be chosen in accordance to a number of factors, such as the capabilities and physical layout of the sensors, capabilities of the JOP, and simulation of the environment. E.g. the JOP requires a tasks' deadlines to be equal to their periods, and the environment adds new bricks to the system in intervals of at most $3.5s$.

A restriction of the following calculations is the requirement of having deadlines being equal to periods on the JOP.

## G.1 Methane level check

Both water and methane bricks pass by the methane sensor on a conveyor belt. The sensor must register all bricks passing by, determine their type, and keep a history of observed bricks. It is not possible to predict the absolute time a brick passes by the sensor. Fortunately, the time frame for which a brick passing by the sensor is detectable can be calculated. This time frame is calculated using the speed of the conveyor belt and dimensions of the bricks. The speed of the conveyor belt is measured to be $88mm/s$, and the diameter of a brick is $14mm$. As a result of the physical properties of the bricks, at least $10mm$ of the brick is estimated to be detectable. This is caused by the brick being a sphere, for which light is reflected differently depending on the point on the curve observed by the light sensor. Using these details, the time frame which ensures the brick is detected is calculated to be:

$$\frac{10mm}{88mm/s} = 0.113s = 113ms.$$

The challenge is then to select a deadline and period for the methane detection task which ensures that a passing brick is always detected, but never registered twice. In order to ensure a passing brick is always detected, the worst-case time between two consecutive sensor readings must be shorter than the time frame for detecting the brick. Therefore, is the deadline set to be $56.5ms$, such that two consecutive sensor readings are at most $113ms$ apart. Unfortunately, the granularity of the implementation languages when defining periods are $1ms$, thus we must use a deadline of $56ms$ as an alternative.

Figure G.1 shows three consecutive periods and the detection time frame of a single brick. In the figure, the sensor readings in two periods are conducted with their worst-case distance to each other, showing why this distance should be shorter than the detection time frame. Since periods and deadlines are equal it is not possible to control when in a period the actual sensor reading is conducted. Thus, as shown in the figure, multiple readings can be conducted on the same brick. In the figure, both period 2 and period 3 detects the brick, and it is trivial to imagine the reading in period 1 being conducted later such that a third reading reads the same brick.

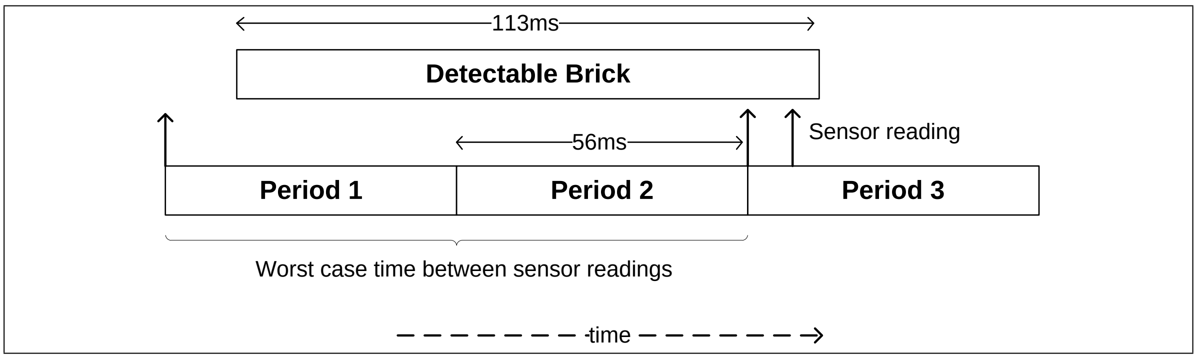


**Figure G.1:** Figure showing the time frame in which a brick can be detected in relation to the periods of the methane detection task.

In practice, filtering multiple detections of the same brick can be conducted by applying some simple logic. If multiple consecutive readings show the presence of a brick, only the first reading is seen as valid. Only when a reading indicates the absence of a brick are this behaviour reset, allowing the next detected brick to be observed as valid. Since it is known that bricks are added to the system with an inter-arrival time of at most $3.5s$ it is ensured that a reading with no bricks can be conducted between two bricks.

The requirement of reacting to critical methane levels within $200ms$ is kept, since methane is detected using at most two periods, where the second period must react to the event, resulting in a worst-case reaction time of two $56ms$ periods, equal to $112ms$.

## G.2 Critical Water Level Check

The specification states that the water pump must be stopped before $200ms$ after low water level is reached and started before $200ms$ of high water level is reached. In the example, either high or low water level is reached when the stack of bricks in the shaft reaches either the high or low water level sensors respectively. Thus, in order to detect the water level the sensors must detect the presence or absence of bricks in front of them, while ignoring false positives created by bricks falling through the shaft on their way to the top of the stack. In the following, it is described how the presence of bricks can be detected by a sensor, but in principle the same approach is used when detecting the absence and are thus omitted.

False positives are avoided for by requiring multiple consecutive readings to return the same result, such that measuring a single brick passing by in free fall can be ignored, under the assumption that the bricks are not feeded faster into the system than expected. A brick in free fall through the shaft passes by the sensor in $15ms$ calculated by

$$t = \sqrt{\frac{2d}{g}} \Rightarrow \tag{G.1}$$

$$\begin{aligned} t_{\text{detectable}} &= t_{\text{after sensor}} - t_{\text{before sensor}} \\ &= \sqrt{\frac{2d_{\text{after sensor}}}{g}} - \sqrt{\frac{2d_{\text{before sensor}}}{g}} \\ &= \sqrt{\frac{2 \cdot 0.054m}{9.82m/s^2}} - \sqrt{\frac{2 \cdot 0.04m}{9.82m/s^2}} \\ &= 15ms, \end{aligned} \tag{G.2}$$

where $t_{\text{after sensor}}$ and $t_{\text{before sensor}}$ represents the time the brick falls to the start of the sensor area in which it can be detected and the time where it has passed the sensor. Furthermore, $d_{\text{before sensor}}$ and $d_{\text{after sensor}}$ represents the distance to which the brick can be detected and the distance travelled before it cannot be detected any more. In our case we can detect the brick after $4mm$ and then the brick must travel $14mm$ before it has passed the sensor.

As described in the previous section, deadlines and periods are required to be equal and we are unable to define when in a period the sensor reading is conducted. Thus, a reading can be conducted late in one period and early in a following period such that a falling brick can be detected twice. This is avoided by requiring three readings to detect a brick.

In Figure G.2 the time-line of the detection of a brick is shown.

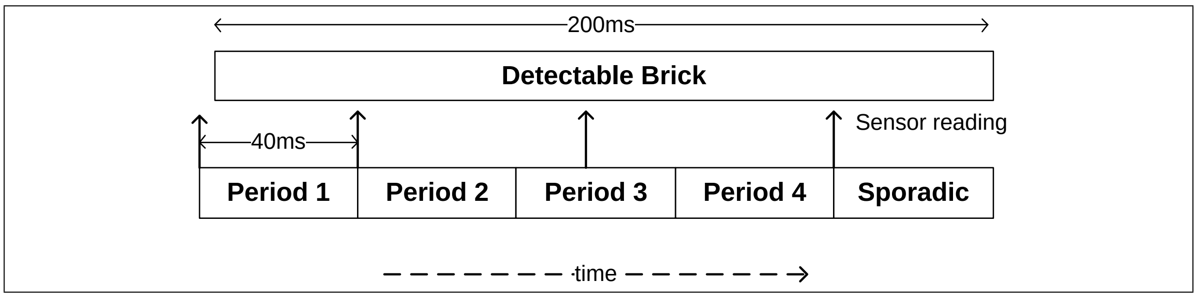


**Figure G.2:** Figure showing the time frame in which the water level has changed and the periods detecting the change.

Since the sensor reading conducted in period 1 can be conducted right before the water level changes a total of four periods can be necessary to detect the presence of bricks three times. Furthermore, when the water level change is detected, a sporadic task is fired in order to react to this change. Thus, a total of five tasks needs to be released within the $200ms$ time-frame. Thus by selecting a period of $40ms$ for the periodic tasks and inter-arrival time of $40ms$ for the sporadic task the $200ms$ reaction time requirement is upheld. Since the sporadic task are only released after at least three consecutive periods, and since it can be released in the beginning of the third period, the actual minimum inter-arrival time of the aperiodic task are $80ms$. Regardless, the $40ms$ deadline and period are selected since we must set deadlines equal to periods, and a deadline of $40ms$ are satisfactory in this case. This results in a more pessimistic estimate than necessary but this is not regarded as a problem as long as the system is schedulable.

# H
# VHDL Excerpt

Listing H.1 shows the excerpt of the VHDL file that defines the cache size and number of cache blocks used on the JOP.

```
entity jop is

generic (
    ram_cnt    : integer := 2;
    rom_cnt    : integer := 10;
    jpc_width  : integer := 12;
    block_bits : integer := 4;
    spm_width  : integer := 0;
);
```

**Listing H.1:** Excerpt of the VHDL file jopcyc12.vhd defining *jpc_width* and *block_bits* used later for initialising the cache of the JOP.

As shown, the JOP defines the *jpc_width* and the *block_bits* variables to be 12 and 4, respectively. Further examination of the VHDL files describing the JOP reveals that these constant values subsequently are used when initialising the cache. The cache is defined in the cache.vhd file and states that the cache size in bytes and number of blocks are defined by $cache\ size = 2^{jpc\_width}$ and $number\ of\ blocks = 2^{block\_bits}$. Hence, by substitution, these yield the values:

$$\begin{aligned} cache\ size &= 2^{12} = 4096 bytes = 1024 words \\ number\ of\ blocks &= 2^{4} = 16 \end{aligned} \tag{H.1}$$

# Code Examples of Real-Time Java Profiles

This appendix shows the essential class of the mine pump example used to initialise the system. I.1, I.2, and I.3 show the invidiual implementations of this initialisation of the RJ, PJ, and SCJ, respectively.

```java
public class Minepump extends Initializer {
    // Definition of constants are placed here
    public void run() {

        // Actuators
        WaterpumpActuator waterpumpActuator = new WaterpumpActuator(
            ACTUATOR_ID_WATERPUMP);
        GenericActuator environmentActuators = new GenericActuator(
            ACTUATOR_ID_ENVIRONMENT);

        // Sensors
        int criticalMethaneLevel = 7;
        int brickHistorySize = 10;
        MethaneSensor methaneSensor = new MethaneSensor(SENSOR_ID_METHANE,
            criticalMethaneLevel, brickHistorySize);

        int consecutiveNoWaterReadings = 3;
        int consecutiveHighWaterReadings = 3;
        HighWaterSensor highWaterSensor = new HighWaterSensor(SENSOR_ID_HIGH_WATER,
            consecutiveHighWaterReadings);
        LowWaterSensor lowWaterSensor = new LowWaterSensor(SENSOR_ID_LOW_WATER,
            consecutiveNoWaterReadings);

        // MethaneDetection
        new PeriodicThread(
            new PriorityParameters(PERIODIC_GAS_PRIORITY),
            new PeriodicParameters(new AbsoluteTime(0, 0), new RelativeTime(
                PERIODIC_GAS_PERIOD, 0)),
            new MethaneDetectionRunnable(methaneSensor, waterpumpActuator));

        // WaterLevelDetection
        SporadicWaterLevelHigh waterHighHandler = new SporadicWaterLevelHigh(
            new PriorityParameters(SPORADIC_HIGH_PRIORITY),
            new SporadicParameters(new RelativeTime(SPORADIC_WATER_PERIOD, 0),1),
            waterpumpActuator);

        SporadicWaterLevelLow waterLowHandler = new SporadicWaterLevelLow(
            new PriorityParameters(SPORADIC_LOW_PRIORITY),
            new SporadicParameters(new RelativeTime(SPORADIC_WATER_PERIOD, 0),1),
            waterpumpActuator);

        SporadicEvent waterHighEvent = new SporadicEvent(waterHighHandler);
        SporadicEvent waterLowEvent = new SporadicEvent(waterLowHandler);

        new PeriodicThread(
            new PriorityParameters(PERIODIC_WATER_PRIORITY),
            new PeriodicParameters(new AbsoluteTime(0, 0), new RelativeTime(
                PERIODIC_WATER_PERIOD, 0)),
            new WaterLevelDetectionRunnable(waterHighEvent, waterLowEvent,
                highWaterSensor, lowWaterSensor));

        // init system
        environmentActuators.start();
    }

    public static void main(String[] args) {
        new Minepump().start();
    }
}
```

**Listing I.1:** An excerpt of the `Minepump` class located in Minepump.java of RJ.

```
public class Minepump extends Mission {
    // Definition of constants are placed here
    public Minepump() {

        // Actuators
        WaterpumpActuator waterpumpActuator = new WaterpumpActuator(
            ACTUATOR_ID_WATERPUMP);
        GenericActuator environmentActuators = new GenericActuator(
            ACTUATOR_ID_ENVIRONMENT);

        // Sensors
        int criticalMethaneLevel = 2;
        int brickHistorySize = 10;
        MethaneSensor methaneSensor = new MethaneSensor(SENSOR_ID_METHANE,
            criticalMethaneLevel, brickHistorySize);

        int consecutiveNoWaterReadings = 3;
        int consecutiveHighWaterReadings = 3;
        HighWaterSensor highWaterSensor = new HighWaterSensor(SENSOR_ID_HIGH_WATER,
            consecutiveHighWaterReadings);
        LowWaterSensor lowWaterSensor = new LowWaterSensor(SENSOR_ID_LOW_WATER,
            consecutiveNoWaterReadings);

        // Methane mission
        addToMission(
            new PeriodicMethaneDetection(
                new PriorityParameters(GAS_PRIORITY),
                new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(
                    PERIODIC_GAS_PERIOD, 0),
                Scheduler.getDefaultScheduler(),
                new LTMemory(10*1024),
                methaneSensor,
                waterPumpActuator))
        );

        // Water Level detection mission
        AperiodicWaterLevelHigh aperiodicWaterLevelHigh = new AperiodicWaterLevelHigh
            (
            new PriorityParameters(WATER_PRIORITY-1),
            new AperiodicParameters(new RelativeTime(APERIODIC_WATER_COST, 0),
                RelativeTime(APERIODIC_WATER_DEADLINE, 0)),
            Scheduler.getDefaultScheduler(),
            new LTMemory(10*1024),
            waterpumpActuator);

        AperiodicWaterLevelLow aperiodicWaterLevelLow = new AperiodicWaterLevelLow(
            new PriorityParameters(WATER_PRIORITY-1),
            new AperiodicParameters(new RelativeTime(APERIODIC_WATER_COST, 0),
                RelativeTime(APERIODIC_WATER_DEADLINE, 0)),
            Scheduler.getDefaultScheduler(),
            new LTMemory(10*1024),
            waterpumpActuator);

        AperiodicEvent waterHighEvent = new AperiodicEvent(waterHighHandler);
        AperiodicEvent waterLowEvent = new AperiodicEvent(waterLowHandler);

        addToMission(
            new PeriodicWaterLevelDetection(
                new PriorityParameters(WATER_PRIORITY),
                new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(
                    PERIODIC_WATER_PERIOD, 0)),
                Scheduler.getDefaultScheduler(),
                new LTMemory(10*1024),
                waterHighEvent,
                waterLowEvent,
                HighWaterSensor,
                LowWaterSensor)
        );

        addToMission(aperiodicWaterLevelHigh);
        addToMission(aperiodicWaterLevelLow);
        add();

        // init system
        environmentActuators.start();
    }

    public static void main(String[] args) {
        new Minepump();
    }
}
```

**Listing I.2:** An excerpt of the Minepump class located in Minepump.java of PJ.

```
@SCJAllowed(members=true, value=LEVEL_2)
public class MainMission extends Mission {
    // Definition of constants are placed here
    protected void initialize() {

        // Actuators
        WaterpumpActuator waterpumpActuator = new WaterpumpActuator(
            ACTUATOR_ID_WATERPUMP);
        GenericActuator environmentActuators = new GenericActuator(
            ACTUATOR_ID_ENVIRONMENT);

        // Sensors
        int criticalMethaneLevel = 2;
        int brickHistorySize = 10;
        MethaneSensor methaneSensor = new MethaneSensor(SENSOR_ID_METHANE,
            criticalMethaneLevel, brickHistorySize);

        int consecutiveNoWaterReadings = 3;
        int consecutiveHighWaterReadings = 3;
        HighWaterSensor highWaterSensor = new HighWaterSensor(SENSOR_ID_HIGH_WATER,
            consecutiveHighWaterReadings);
        LowWaterSensor lowWaterSensor = new LowWaterSensor(SENSOR_ID_LOW_WATER,
            consecutiveNoWaterReadings);

        // Methane
        PeriodicMethaneDetectionEventHandler methane = new
            PeriodicMethaneDetectionEventHandler(
            new PriorityParameters(11),
            new PeriodicParameters(new RelativeTime(0,0), new RelativeTime(
                PERIODIC_GAS_PERIOD,0)),
            new StorageParameters(100000L, 1000L, 1000L, 1000, 1000),
            1000,
            methaneSensor);
        methane.register();

        // Water Level Mission
        AperiodicWaterLevelHighEventHander waterHighHandler = new
            AperiodicWaterLevelHighEventHander(
            new PriorityParameters(WATER_PRIORITY−1),
            new AperiodicParameters(new RelativeTime(SPORADIC_WATER_PERIOD, 0),1),
            new StorageParameters(100000L, 1000L, 1000L, 1000, 1000),
            1000,
            waterpumpActuator);

        AperiodicWaterLevelLowEventHander waterLowHandler = new
            AperiodicWaterLevelLowEventHander(
            new PriorityParameters(WATER_PRIORITY−1),
            new AperiodicParameters(new RelativeTime(SPORADIC_WATER_PERIOD, 0),1),
            new StorageParameters(100000L, 1000L, 1000L, 1000, 1000),
            1000,
            waterpumpActuator);

        AperiodicEvent waterHighEvent = new AperiodicEvent(waterHighHandler);
        AperiodicEvent waterLowEvent = new AperiodicEvent(waterLowHandler);

        PeriodicWaterLevelDetectionEventHandler water = new
            PeriodicWaterLevelDetectionEventHandler(
            new PriorityParameters(11),
            new PeriodicParameters(new RelativeTime(0,0), new RelativeTime(
                PERIODIC_WATER_PERIOD,0)),
            new StorageParameters(100000L, 1000L, 1000L, 1000, 1000),
            1000,
            waterHighEvent,
            waterLowEvent,
            highWaterSensor,
            lowWaterSensor);
        water.register();

        // init system
        environmentActuators.start();
    }
}
```
**Listing I.3:** An excerpt of the `MainMission` class located in MainMission.java of SCJ.

# Bibliography

AbsInt(2010). Stackanalyzer - stack usage analysis. `http://www.absint.com/stackanalyzer` (2 December 2010.).

aJile Systems(2000). aj-100tm real-time low power javatm processor. `www.ajile.com`.

Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D.(2007). Cost analysis of java bytecode. *Programming Languages and Systems*, 157–172.

Alfred V. Aho, Monica S. Lam, R.S. and Ullman, J.D.(2006). *Compilers: Principles, Techniques and Tools.* Pearson Education.

Allen, F.(1970). Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, 1–19. ACM.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W.(2004). Times - a tool for schedulability analysis and code generation of real-time systems. In K. Larsen and P. Niebert (eds.), *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, 60–72. Springer Berlin Heidelberg. URL `http://dx.doi.org/10.1007/978-3-540-40903-8_6`. 10.1007/978-3-540-40903-8_6.

Andrew N. Sloss, Dominic Symes, C.W.(2004). *ARM System Developer's Guide: Designing and Optimizing System Software.* Morgan Kaufmann.

ARM(2006). *Cortex-A8 revision r1p1 technical reference manual.* ARM.

ARM(2010). Real-time operating systems (rtos) - arm. `http://www.arm.com/community/software-enablement/rtos-real-time-operating-system.php` (9. September 2010).

Atira(2009). Scrum and agile methods: The real world. `https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2009/SOE/Slides/lecture14_atira.pdf` (15. October 2009).

Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A.(2002). Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5), 284–292.

Audsley, N., Burns, A., Richardson, M., and Wellings, A.(1990). *Deadline monotonic scheduling.* Citeseer.

Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W., and Wellings, A.J.(1995). Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems*, 8, 173–198. URL `http://dx.doi.org/10.1007/BF01094342`. 10.1007/BF01094342.

Bacon, D.F., Cheng, P., and Rajan, V.T.(2003). A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, 38, 285–298. doi:http://doi.acm.org/10.1145/640128.604155. URL `http://doi.acm.org/10.1145/640128.604155`.

Barr, M.(2002). Introduction to Closed-Loop Control. *Embedded Systems Programming*, 11, 55–56.

Baruah, S.K., Rosier, L.E., and Howell, R.R.(1990). Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2, 301–324. doi:http://dx.doi.org/10.1007/BF01995675. URL `http://dx.doi.org/10.1007/BF01995675`.

Behrmann, G., A, D., and Larsen, P.K.(2004). A tutorial on uppaal. In M. Bernardo and F. Corradini (eds.), *LNCS, Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185, 200–237. Springer Verlag. URL `http://doc.utwente.nl/51010/`.

Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W.(1996). UPPAAL — A Tool Suite for Automatic Verification of Real-time Systems. *Hybrid Systems III*, 232–243.

Bernat, G., Colin, A., and Petters, S.(2002). Wcet analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, 279 – 288. doi:10.1109/REAL.2002.1181582.

Blieberger, J.(2002). Data-flow Frameworks for Worst-case Execution Time Analysis. *Real-Time Systems*, 22(3), 183–227.

Bøgholm, T., Kragh-Hansen, H., Olsen, P., Thomsen, B., and Larsen, K.(2008a). Model-based Schedulability Analysis of Safety Critical Hard Real-time Java Programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, 106–114. ACM.

Bøgholm, T., Hansen, R.R., Ravn, A.P., Thomsen, B., and Søndergaard, H.(2009). A predictable java profile: Rationale and implementations. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, 150–159. ACM, New York, NY, USA. doi:http://doi.acm.org/10.1145/1620405.1620427.

Bøgholm, T., Kragh-Hansen, H., and Olsen, P.(2008b). Model-based schedulability analysis of real-time systems. `http://sarts.boegholm.dk/cd/Report/thesis.pdf` (2 December 2010.).

Bøgholm, T., Kragh-Hansen, H., and Olsen, P.(2008c). Real-time java. Technical report, Aalborg University, Department of Computer Science.

Burns, A. and Wellings, A.(1994). HRT-HOOD: A Structured Design Method for Hard Real-time Systems. *Real-Time Systems*, 6(1), 73–114.

Burns, A. and Wellings, A.(2009). *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Educational Publishers Inc., Boston, MA, USA, 4th edition.

Cachera, D., Jensen, T., Pichardie, D., and Schneider, G.(2005). Certified memory usage analysis. *FM 2005: Formal Methods*, 91–106.

Calzolai, F., De Nicola, R., Loreti, M., and Tiezzi, F.(2008). TAPAs: a Tool for the Analysis of Process Algebras. *Transactions on Petri Nets and Other Models of Concurrency I*, 54–70.

Chin, W., Nguyen, H., Qin, S., and Rinard, M.(2005). Memory Usage Verification for OO Programs. *Static Analysis*, 70–86.

Cohn, M.(2004). *User Stories Applied: For Agile Software Development.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Craciunas, S.S., Kirsch, C.M., Payer, H., Sokolova, A., Stadler, H., and Staudinger, R.(2008). A compacting real-time memory management system. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 349–362. USENIX Association, Berkeley, CA, USA. URL `http://portal.acm.org/citation.cfm?id=1404014.1404043`.

Dalsgaard, A., Olesen, M., Toft, M., Hansen, R., and Larsen, K.(2009). Wcet analysis of arm processors using real-time model checking. In *Doctoral Symposium on Systems Software Verification (DS SSV'09) Real Software, Real Problems, Real Solutions*, 4. Citeseer.

DARTS(2007). Timestool. `http://www.timestool.com/` (12. November 2010).

Dibble, P.C.(2002). *Real Time Java Platform Programming Platform.* Prentice Hall.

Engblom, J., Ermedahl, A., and Stappert, F.(2000). Comparing Different Worst-case Execution Time Analysis Methods. In *Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000).*

English, J.(2004). *Introduction to Operating Systems: Behind the Desktop.* Palgrave Macmillan.

Fersman, E. and Yi, W.(2004). A Generic Approach to Schedulability Analysis of Real-time Tasks. *Nordic Journal of Computing*, 11, 129–147.

Gosling, J., Joy, B., Steele, G., and Bracha, G.(2005). *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, 3 edition.

Gourvest, H., Pattton, W., Notebaert, P., Berkelaar, M., Eikland, K., and Dirks, J.(2010). lp_solve reference guide. `http://lpsolve.sourceforge.net/` (23 November 2010.).

Graaf, B., Lormans, M., , and Toetenel, H.(2003). Embedded software engineering: The state of the practice. *IEEE software.*

Griffiths, I.(2010). *Programming C# 4.0.* Pragma, 6th edition.

Group, T.L.(2010). Lego rcx. `http://www.lego.com/eng/education/mindstorms/home.asp?pagename=rcx` (7. December 2010).

Hardin, D.(2010). aJile Systems: Low-Power Direct-Execution Java Microprocessors for Real-Time and Networked Embedded Applications. White paper.

Heckmann, R. and Ferdinand, C.(2009). Worst-case Execution Time Prediction by Static Program Analysis. *White Paper, AbsInt Angewandte Informatik GmbH, 22nd May.*

Henties, T., Hunt, J.J., Locke, D., Nilsen, K., Schoeberl, M., and Vitek, J.(2009). Java for safety-critical applications. *Electronic Notes in Theoretical Computer Science.* URL `http://www.jopdesign.com/doc/safecert2009.pdf`.

Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D.(2006). PRISM: A tool for Automatic Verification of Probabilistic Systems. *Tools and Algorithms for the Construction and Analysis of Systems*, 441–444.

Huber, B. and Puschner, P.(2010). Techniques to calculate the worst-case execution-time. `http://ti.tuwien.ac.at/rts/teaching/courses/wcet/slides/wcet02_static_analysis.pdf` (9. December 2010).

Huber, B. and Schoeberl, M.(2009). Comparison of implicit path enumeration and model checking based wcet analysis. In N. Holsti (ed.), *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis.* Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. `http://drops.dagstuhl.de/opus/volltexte/2009/2281`.

Insup Lee, J.Y.T. and Son, S.H.(2008). *Handbook of Real-Time and Embedded Systems.* Chapman & Hall/CRC.

JOP(2008). Jop: A tiny java processor core for fpga. `http://www.jopdesign.com/` (9. September 2010).

Joseph, M. and Pandya, P.(1986). Finding response times in a real-time system. *The Computer Journal*, 29(5), 390.

JSR302, G.(2010). Safety critical specification for java. Special communication with JSR 302 Group.

Knuth, D.E.(1997). *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Kwon, J., Wellings, A., and King, S.(2002). Ravenscar-Java: A High Integrity Profile for Real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, 131–140. ACM.

Laprie, J., Arlat, J., Blanquart, J., Costes, A., Crouzet, Y., Deswarte, Y., Fabre, J., Guillermain, H., Kaâniche, M., Kanoun, K., et al.(1995). «Guide de la sûreté de fonctionnement», 324p. *Edition Cépaduès, Toulouse.*

Larman, C.(2003). *Agile and Iterative Development.* A Managers Guide.

Leung, J.Y.T. and Whitehead, J.(1982). On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4), 237 – 250. doi:DOI:10.1016/0166-5316(82)90024-4. URL `http://www.sciencedirect.com/science/article/B6V13-4903SW2-1D/2/471090643e85292d8b0c895c302f9bec`.

Li, Y. and Malik, S.(1995). Performance Analysis of Embedded Software using Implicit Path Enumeration. *ACM SIGPLAN Notices*, 30(11), 88–98.

Liu, C.L. and Layland, J.W.(1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 46–61. doi:http://doi.acm.org/10.1145/321738.321743.

Liu, Z. and Joseph, M.(2005). Real-Time and Fault-Tolerant Systems–Specification, Verification, Refinement and Scheduling. Technical report, Technical Report 323, UNU-IIST, PO Box 3058, Macau.

Locke, C.D.(1992). Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4, 37–53. URL `http://dx.doi.org/10.1007/BF00365463`. 10.1007/BF00365463.

Lokuciejewski, P. and Marwedel, P.(2011). WCET Analysis Techniques. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*, 13–22.

Lundqvist, T. and Stenstrom, P.(2002). Timing Anomalies in Dynamically Scheduled Microprocessors. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 12–21. IEEE.

Mathur, A.P.(2008). *Foundations of Software Testing.* Addison-Wesley Professional, 1st edition.

Metzner, A.(2004). Why model checking can improve wcet analysis. In R. Alur and D.A. Peled (eds.), *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, 298–301. Springer Berlin / Heidelberg. URL `http://dx.doi.org/10.1007/978-3-540-27813-9_26`. 10.1007/978-3-540-27813-9_26.

Munk-Madsen, A., Mathiassen, L., Nielsen, P.A., and Stage, J.(2000). *Object Oriented Analysis and Design.* Marko.

Nicolescu, G. and Mosterman, P.J.(2010). *Model-Based Design for Embedded Systems.* CRC Press.

Oracle(2009). RTSJ 1.1 Alpha 6, release notes. `http://www.jcp.org/en/jsr/detail?id=282`.

Puffitsch, W., Huber, B., and Schoeberl, M.(2010). Worst-case analysis of heap allocations. In *Proceedings of the 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2010).* URL `http://www.jopdesign.com/doc/wcmem.pdf`.

Puschner, P. and Koza, C.(1989). Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2), 159–176. doi:http://dx.doi.org/10.1007/BF00571421.

Qing Li, C.Y.(2003). *Real-Time Concepts for Embedded Systems, 1st edition.* CMP Books.

Ravn, A.P. and Schoeberl, M.(2010). Cyclic executive for safety-critical java on chip-multiprocessors. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, 63–69. ACM, New York, NY, USA. doi:http://doi.acm.org/10.1145/1850771.1850779. URL `http://doi.acm.org/10.1145/1850771.1850779`.

River, W.(2007). Wind river powers 'brain' for nasa's mars phoenix lander. `http://cdn.windriver.com/uk/press/pr.html?ID=4801` (13. December 2010).

River, W.(2010). Vxworks. `http://www.windriver.com/products/vxworks/` (13. December 2010).

RTSJ.org(2010). Rtsj 1.0.2. `http://www.rtsj.org/specjavadoc/book_index.html` (20. October 2010).

Sarkar, S., Sewell, P., Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M., and Alglave, J.(2009). The Semantics of x86-CC Multiprocessor Machine Code. *ACM SIGPLAN Notices*, 44(1), 379–391.

Schoeberl, M., Puffitsch, W., Pedersen, R., and Huber, B.(2010). Worst-case Execution Time Analysis for a Java Processor. *Software: Practice and Experience*, 40(6), 507–542.

Schoeberl, M., Thomsen, B., Ravn, A., and Søndergaard, H.(2007). A profile for safety critical java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07*, 94–101.

Schoeberl, M.(2003). JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, 346–359. Springer, Catania, Italy. doi:10.1007/b94345. URL `http://www.jopdesign.com/doc/jtres03.pdf`.

Schoeberl, M.(2005). *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Ph.D. thesis, Vienna University of Technology. URL `http://www.jopdesign.com/thesis/thesis.pdf`.

Schoeberl, M.(2009). *JOP Reference Handbook: Building Embedded Systems*. CreateSpace.

Schoeberl, M. and Pedersen, R.(2006). Wcet analysis for a java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, JTRES '06, 202–211. ACM, New York, NY, USA. doi:http://doi.acm.org/10.1145/1167999.1168033. URL `http://doi.acm.org/10.1145/1167999.1168033`.

Sehlberg, D., Ermedahl, A., Gustafsson, J., Lisper, B., and Wiegratz, S.(2006). Static wcet analysis of real-time task-oriented code in vehicle control systems. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, 212 –219. doi:10.1109/ISoLA.2006.63.

Selic, B.(1998). Using UML for Modelling Complex Real-Time Systems. In *Languages, Compilers, and Tools for Embedded Systems*, 250–260. Springer.

Sha, L., Rajkumar, R., and Lehoczky, J.(1990). Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on computers*, 1175–1185.

Sommerville, I.(1999). *Software Engineering.* Addison Wesley.

Søndergaard, H., Thomsen, B., and Ravn, A.P.(2006). A ravenscar-java profile implementation. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, 38–47. ACM, New York, NY, USA. doi: http://doi.acm.org/10.1145/1167999.1168008.

Stankovic, J.A. and Rajkumar, R.(2004). Real-time operating systems. *Real-Time Systems*, 28, 237–253. URL `http://dx.doi.org/10.1023/B:TIME.0000045319.20260.73`. 10.1023/B:TIME.0000045319.20260.73.

Sw701b(2009). Easy clocking - a system for automatically clocking in and out employees. Technical report, Aalborg University, Department of Computer Science. Christian Frost and Casper Svenning Jensen and Kasper Søe Luckow.

Sw801b(2009). Bluecaml - bluetooth, collaborative, and maintainable location system. Technical report, Aalborg University, Department of Computer Science. Christian Frost and Casper Svenning Jensen and Kasper Søe Luckow.

TIOBE-Software(2010). Tiobe programming community index for november 2010. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (23. November 2010).

UPPAAL(2010). Up4aal. `http://www.uppaal.com/` (8. December 2010).

Volta(2009). Wcet analyzer for jop. `http://volta.sourceforge.net` (19 November 2010.).

Wellings, A.(2004). *Concurrent and Real-time Programming in Java.* Wiley.

Wellings, A.(2010). Presentation of a locality model for the real-time specification for java. JTRES 2010 `http://d3s.mff.cuni.cz/conferences/jtres2010/video/p36-malik.html`.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.(2008). The Worst-case Execution-time Problem—overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 1–53.

Xenomai(2010). Xenomai. `http://www.xenomai.org/` (13. December 2010).

Zerzelidis, Wellings, Zerzelidis, A., and Wellings, A.J.(2004). Requirements for a real-time .net framework.

# Acronyms

| | | |
|---|---|---|
| RTOS | Real-Time Operating System | 11 |
| GPOS | General Purpose Operating System | 18 |
| WCET | Worst-Case Execution Time | 3 |
| CFA | Control-Flow Analysis | 25 |
| JOP | Java Optimized Processor | 19 |
| WCHA | Worst Case Heap Allocations | 124 |
| JVM | Java Virtual Machine | 17 |
| JIT | Just-In-Time | 17 |
| FPS | Fixed-Priority Scheduling | 37 |
| FPGA | Field-Programmable Gate Array | 20 |
| RJ | Ravenscar-Java | 17 |
| CFG | Control-Flow Graph | 25 |
| CLR | Common Language Runtime | 17 |
| CLI | Common Language Infrastructure | 119 |
| CES | Cyclic Executive Scheduling | 37 |
| EDF | Earliest Deadline First Scheduling | 37 |
| SARTS | Schedulability Analyzer for Real-Time Systems | 47 |
| PJ | Predictable Java | 17 |
| AEH | Asynchronous Event Handler | 59 |
| SCJ | Safety Critical Java | 17 |
| RTA | Response Time Analysis | 42 |
| IPET | Implicit Path Enumeration Technique | 28 |
| ILP | Integer Linear Programming | 29 |
| DFA | Data-Flow Analysis | 25 |
| RTSJ | Real-Time Specification for Java | 2 |
| JNI | Java Native Interface | 61 |
| ATC | Asynchronous Transfer of Control | 59 |
| PJ | Predictable Java | 17 |
| WCA | WCET Analyzer | 80 |
| SCJ | Safety Critical Java | 17 |
| BCEL | Byte Code Engineering Library | 84 |

| | | |
|---|---|---|
| WCRT | Worst Case Response Time | 43 |
| CLI | Common Language Infrastructure | 119 |
| OS | Operating System | 18 |
| VHDL | VHSIC Hardware Description Language | 20 |
| ICPP | Immediate Ceiling Priority Protocol | 40 |