WCET Analysis of Java Bytecode
Featuring Common Execution Environments

Software
Elite programme in Embedded Systems
2011
Title:  
WCET Analysis of Java Bytecode Featuring Common Execution Environments

Master’s Thesis in:  
Database and Programming Technologies part of the Elite Programme in Embedded Systems at Aalborg University

Project time frame:  
SW10, 1st February 2011 - 7th June, 2011

Group members:  
Christian Frost  
Casper Svenning Jensen  
Kasper Søe Luckow

Supervisor:  
Bent Thomsen

Abstract:  
We present a novel tool called Tool for Execution Time Analysis of Java bytecode (TetaJ) that is designed for statically determining Worst Case Execution Time (WCET) of Java Bytecode (JBC) programs. Similar tools have been devised for the same purpose, but they are either using an unsafe measurement-based technique or are relying on hardware capable of natively executing JBC. TetaJ differentiates itself by presenting a novel approach for WCET analysis, whose key characteristic is the incorporation of a software implementation of the Java Virtual Machine (JVM). The prime benefit is that it accommodates more common execution environments featuring common embedded processors, thereby, in our opinion, increasing the incentive for adopting Java for embedded hard real-time systems development.

The fundamental technique employed in TetaJ is to view the program analysis problem of determining WCET as a model-checking problem. Specifically, the executable of the JVM and the JBC program source are processed for yielding a reconstruction of their respective Control Flow Graph (CFG). Subsequently, a further transformation is responsible for constructing a network of timed automata that can be input to the state-of-the-art UPPAAL model checking tool that will determine the WCET.

To evaluate the applicability of TetaJ, a case study is conducted based on the classic mine pump real-time system using an execution environment featuring a concrete implementation of the JVM, called Hardware near Virtual Machine (HVM), that runs on an Atmel AVR ATmega2560 processor. From the case study we conclude that TetaJ is capable of estimating safe WCETs with reasonable precision. While TetaJ is only applied for one case study, it is stressed that it applies for arbitrary hardware and interpretation-based JVMs.

Copies: 7

Total pages: 158

Of this Appendices: 16

Paper finished: 7th of June 2011

The content of this report is freely available, but publication is only permitted with explicit permission from the authors.
This Master’s thesis is written by three software-engineering students participating in the elite programme in embedded systems at Aalborg University. The thesis was commenced on the 1st of February 2011, and finished on the 7th of June 2011.

During the project, help has been received from a number of people whom we would like to thank. First of all, we thank our supervisor Bent Thomsen, associate professor at Aalborg University, who has put great effort in giving advice regarding the project. Besides, Bent and his colleague, Anders P. Ravn, professor at Aalborg University, have engaged in valuable discussions concerning the problems of the thesis and the field in general. Related to this, we would like to thank René Rydhof Hansen, associate professor at Aalborg University, who censored our preliminary project and provided valuable advice on further directions. Furthermore, we would like to thank Alexandre David, associate professor at Aalborg University, and Benedikt Huber, research and teaching assistant at Vienna University of Technology, who helped with specific issues regarding the UPPAAL model checker and Mads Christian Olesen, Ph.D. student at Aalborg University, has provided help with issues regarding METAMOC. Finally, we thank Stephan Korsholm, lecturer at VIA University, for his commitment and enthusiasm in helping us modifying his Java Virtual Machine implementation called HVM.

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 software engineering Master’s 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.

A summary of this Master’s thesis is supplied in Appendix H.

The project has its own website, http://tetaj.dk, and the reader is encouraged to visit it for obtaining the source code for TetaJ.

Aalborg, June 2011
- Christian Frost, Casper Svenning Jensen, and Kasper Søe Luckow

__________________________
Christian Frost             Casper Svenning Jensen
__________________________
Kasper Søe Luckow
## Contents

1 Introduction 1

I Prerequisites 5

2 Real-Time Systems 6
   2.1 Real-Time Systems in General 6
   2.2 Categorisation of Real-Time Systems 8
   2.3 Tasks in a Real-Time System 9
   2.4 The Structure of a Real-Time System 10

3 Worst Case Execution Time Analysis 12
   3.1 Control Flow Graph 12
   3.2 WCET Analysis Overview 13
   3.3 High-Level WCET Analysis 15
   3.4 Low-Level Analysis 21

4 Scheduling Real-Time Systems 23
   4.1 Scheduling Policies 23
   4.2 Schedulability Analysis 25

5 Real-Time Systems Development in Java 28
   5.1 The Real-Time Specification for Java 28
   5.2 Java Real-Time Profiles 29

II Analysis 32

6 The Java Virtual Machine 33
   6.1 Java Bytecode 33
   6.2 Compilation Techniques 36
   6.3 Requirements for the JVM used by TetaJ 39

7 Real-Time Development Methods 42
   7.1 HRT-HOOD 42
   7.2 SARTS 43
7.3 The Context for TetaJ ...................................................... 44

8 Related WCET Tools .................................................... 45
  8.1 aiT ........................................................................... 45
  8.2 METAMOC ............................................................... 47
  8.3 WCET Analyzer .......................................................... 48
  8.4 eXtensible high-integrity Real-Time Java ......................... 48
  8.5 Ideas for TetaJ ............................................................ 49

9 UPPAAL ........................................................................ 52
  9.1 Overview .................................................................. 52
  9.2 Templates .................................................................. 52
  9.3 The Query Language ................................................... 54
  9.4 Applying UPPAAL for WCET Analysis ......................... 56
  9.5 Progres Measures ....................................................... 57

III The Tool ...................................................................... 58

10 Overview of TetaJ .......................................................... 59
  10.1 The Structure of TetaJ ............................................... 59
  10.2 The Usage of TetaJ .................................................... 60
  10.3 The Toolchain .......................................................... 61

11 CFG Generation ............................................................ 62
  11.1 TetaJ CFG Representation ......................................... 62
  11.2 Java Bytecode to TCFG ............................................ 63

12 Preparing the Model Checking ........................................ 65
  12.1 Loop Bound Analysis ................................................ 65
  12.2 Condition Optimisation ............................................. 70
  12.3 Progress Measures ................................................... 73
  12.4 Model Cleaner Optimisation ...................................... 74

13 UPPAAL Model Generation ............................................ 75
  13.1 Modelling a Basic Block .......................................... 75
  13.2 Modelling Loops ...................................................... 76
13.3 Method Call ......................................................... 76

14 Connecting the Model Layers for WCET Analysis .................. 79
  14.1 Initialisation Model ............................................. 79
  14.2 JVM Model ..................................................... 80
  14.3 Hardware Model ............................................... 80
  14.4 Estimating the WCET .......................................... 82

IV Case Study .......................................................... 83

15 The Mine Pump ...................................................... 84
  15.1 The Mine Pump Real-Time System ............................... 84
  15.2 Choice of Execution Environment ............................... 86

16 Hardware near Virtual Machine ..................................... 89
  16.1 General Properties of the HVM ................................ 89
  16.2 Concepts of the HVM .......................................... 90
  16.3 Making the HVM Analysable .................................. 91
  16.4 Constructing a JVM model for the HVM ...................... 96

17 AVR ATmega2560 ..................................................... 99
  17.1 Atmel AVR ATmega2560 ....................................... 99
  17.2 The Instruction Set of ATmega2560 ............................ 100
  17.3 Modelling the AVR ATmega2560 ............................... 100

18 Design and Implementation of the Mine Pump ....................... 101
  18.1 Design .......................................................... 101
  18.2 Implementation ............................................... 102

19 Evaluation .................................................................. 107
  19.1 Conducting WCET Analysis using TetaJ ....................... 107
  19.2 Evaluation of Optimisations .................................... 108
  19.3 Evaluating the Applicability of TetaJ ......................... 109
  19.4 WCET Analysis of the Mine Pump ............................. 110
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V Conclusion</td>
<td>112</td>
</tr>
<tr>
<td>20 Reflection</td>
<td>113</td>
</tr>
<tr>
<td>20.1 Unsupported Features</td>
<td>113</td>
</tr>
<tr>
<td>20.2 Over-simplifying the HVM</td>
<td>114</td>
</tr>
<tr>
<td>20.3 Foundation for Evaluation</td>
<td>115</td>
</tr>
<tr>
<td>20.4 Requirements of JVM</td>
<td>115</td>
</tr>
<tr>
<td>21 Conclusion</td>
<td>117</td>
</tr>
<tr>
<td>22 Future Work</td>
<td>119</td>
</tr>
<tr>
<td>VI Appendices</td>
<td>121</td>
</tr>
<tr>
<td>Appendix A System Architecture</td>
<td>122</td>
</tr>
<tr>
<td>A.1 Model Generator</td>
<td>122</td>
</tr>
<tr>
<td>A.2 Model Combiner</td>
<td>123</td>
</tr>
<tr>
<td>A.3 Model Processor</td>
<td>123</td>
</tr>
<tr>
<td>Appendix B TCFG Generators</td>
<td>125</td>
</tr>
<tr>
<td>Appendix C TCFG Analyser</td>
<td>127</td>
</tr>
<tr>
<td>Appendix D Model Generator</td>
<td>130</td>
</tr>
<tr>
<td>Appendix E Model Combiner</td>
<td>131</td>
</tr>
<tr>
<td>Appendix F Model Processor</td>
<td>132</td>
</tr>
<tr>
<td>Appendix G Java implemented Algorithms</td>
<td>133</td>
</tr>
<tr>
<td>Appendix H Resumé</td>
<td>136</td>
</tr>
<tr>
<td>Bibliography</td>
<td>138</td>
</tr>
<tr>
<td>Acronyms</td>
<td>146</td>
</tr>
</tbody>
</table>
Today, over 90 percent of all microprocessors are used in embedded systems (Corsaro and Schmidt, 2002). Their appliances range from daily household appliances, such as vacuum cleaners, to life-critical appliances, such as air-bags. This tendency is primarily attributed to a strong interest in the recent visions of pervasive computing (Qing Li, 2003), referring to systems that provide a single, dedicated functionality transparent and omnipresent to the user. This means that they affect our daily lives in many different areas while we do not have to explicitly reason about their presence. To ensure that pervasive computing eventually will become a reality, it naturally implies a tremendous demand for developing embedded systems. Many of these provide a safety-critical service which humans put total reliance in will be operating correctly and, due to this reason, there will often be a time aspect involved to e.g. avoid hazardous or harmful conditions to arise. Systems with such requirements are denoted embedded real-time systems. These are characterised by impose real-time requirements. Besides emphasising on functional correctness, these are required to respond in a timely fashion to events typically originating from the environment in which the system operates. Depending on the severity of exceeding deadlines, embedded real-time systems can be categorised as hard meaning that deadlines must never be exceeded. This naturally demands that one can prove that this never happens. Such systems are the focus of this project.

Historically, the programming languages used for embedded real-time systems development have not changed significantly. Initially, they were developed using assembly languages, but since the advent of the C programming language, there has also been a great tradition of using that as well. The C programming language brought many advancements compared with assembly language. Seen from the perspective of assembly, C is a high-level programming language while still achieving high performance due to optimising compilers. Other important aspects in the migration to C are: increased productivity, better portability of code, maintainability, readability, and, as product of these, avoidance of some common vulnerability that may accidentally be introduced using assembly.

In an attempt to develop a programming language specifically targeted at embedded real-time systems development, the Ada programming language was conceived. Compared to C, Ada emphasises on a much more secure programming model that includes strong typing, better modularity, and many features involved in parallel processing such as message passing and protected objects for avoiding concurrency issues. However, even though Ada contributes
with a much more applicable programming model for embedded real-time systems, newly graduated computer science programmers proficient in Ada are not assumed to be great in number. Regarding C, a recent tendency at educational institutions is also the shift towards higher-level languages such as Java and C#. All in all, programmers proficient in Ada and C are as of now, and eventually will be, costly resources that effectively conflicts with the demand for keeping the production cost as low as possible while the demand for real-time systems development increases.

To further mitigate development costs, one may naturally focus on further enhancing aspects such as productivity and maintainability. These are specifically the design criteria upon which the Java programming language was conceived, and Java is in addition the most popular programming language as of this writing (TIOBE-Software, 2011), which is a consequence of increased focus at educational institutions.

Another important aspect of Java is the secure environment it provides compared to C. Strong typing provides type safety thereby avoiding a great amount of the problematic issues with C. Furthermore, Java does not feature the notion of undefined behaviour as is the case with buffer overflows and division by zero in C. Besides, Java is known for achieving low development time, easy maintainability, and high reusability of developed components, thereby accommodating the demand for developing embedded systems.

Evidently, Java has high potential for embedded real-time systems development but is inappropriate for this purpose in its traditional sense. For instance, it lacks the notion of a deadline and high-resolution real-time clocks. Therefore, recent research has focused on introducing these in Java by a number of initiatives such as the Safety Critical Java (SCJ) specification (JSR302, 2010) and Predictable Java (PJ) (Bøgholm et al., 2009) specification targeted safety-critical systems development. The specification includes among others a new programming model that is more amenable to be proved temporally correct. Temporal correctness can be ensured by using the information from a Worst-Case Execution Time (WCET) analysis to perform schedulability analysis. This analysis concludes whether the real-time tasks of the system can be scheduled such that they will always meet their deadlines.

In respect to Java, the WCET analysis is complicated by the presence of the Java Virtual Machine (JVM), since the JVM introduces an additional layer between the application itself and underlying hardware platform. To mitigate this complexity, research has been focused on eliminating this middle layer by implementing the JVM in hardware thereby achieving native execution of Java Bytecode (JBC) (Schoeberl, 2005; aJile Systems, 2000). From this, it has been possible to experiment with Java to explore its applicability for real-time systems development. It is our opinion that this is indeed the case and the next step must be taken.

Concerning the employment of Java for real-time systems development in the industry, a substantial disadvantage of a JVM implemented in hardware is the necessity of special-purpose hardware. Conversely, a software implementation of the JVM allows for commonly used processors in the domain of embedded systems to be used such as those from the ARM and Atmel AVR families. We hypothesise that being capable of providing hard real-time guarantees for a Java execution environment featuring common embedded processors and
software implementations of the JVM, will greatly increase the incentive for adopting Java in
the domain of real-time systems development.

To achieve the temporal correctness in such execution environments, we present a novel
tool called Tool for Execution Time Analysis of Java bytecode (TetaJ). The purpose of TetaJ
is to conduct WCET analysis of Java tasks running on a JVM executed directly on common
embedded processors without an Operating System (OS). The key design characteristic is flex-
ibility to allow tailoring its usage to the concrete execution environment. Also it emphasises
on obtaining both safe and precise WCET estimates. Safety is naturally achieved by using
sound over-approximations which is at the expense of precision. To obtain high precision,
TetaJ is fundamentally build upon a model-based approach, since this approach more closely
captures the behaviour of the execution environment compared to other techniques (Huber
and Schoeberl, 2009; Dalsgaard et al., 2009; Bøgholm et al., 2008a; AbsInt, 2009; Metzner,
2004).

To ease the usage of TetaJ for the developers, capturing the behaviour of the real-time
tasks is automated after providing the source files and corresponding compiled files. Due to the
complexity of the JVM and underlying hardware, TetaJ expects that information regarding
these are provided by e.g. researchers internally in the given company or be the responsibility
of the JVM and hardware vendors.

To evaluate the applicability of TetaJ, we have conducted a case study featuring the classic
text-book example of a control system of a mine pump. The execution environment is based
on a software implementation of the JVM, called Hardware near Virtual Machine (HVM),
and a typical embedded processor specifically the Atmel AVR ATmega2560. The purpose of
the evaluation is twofold. First we want to show a hard real-time Java application running
on a software implementation of the JVM with common embedded hardware. Secondly, we
want to show how TetaJ can be used to determine the WCET of the tasks of this system.
CHAPTER 1. INTRODUCTION

The remaining report is structured into the following six parts:

Part 1: Prerequisites  This part provides an overview of the essential knowledge of embedded real-time systems used for understanding the context in which TetaJ will be used.

Part 2: Analysis   This part emphasises on the JVM and discusses different WCET analysis tools. Furthermore, an analysis is conducted to understand where TetaJ will fit in existing development methods for embedded real-time systems development.

Part 3: The Tool   Here TetaJ is presented in terms of its architectural design and important choices and considerations made during its construction.

Part 4: Case Study  To show the applicability of TetaJ, this part describes a case study containing the real-time control system of a mine pump.

Part 5: Conclusion  The current state of TetaJ and the experiences gained by the case study are reflected upon, and the project is concluded. Finally, future directions are presented.

Part 6: Appendices  This part contains the appendices referred to throughout the report.
Part I

Prerequisites
This chapter serves the purpose of describing real-time systems in general. The motivation for this is to establish fundamental knowledge that is required in the subsequent chapters.

### 2.1 Real-Time Systems in General

Real-time systems inherently rely on temporal correctness to either respond to external events or trigger an operation in a timely manner. This requires that the system is both capable of responding to events within given deadlines, and capable of postponing execution to avoid premature reactions (Qing Li, 2003). Therefore, it must be possible for the programmer to specify certain information regarding timing for the tasks to be performed. The timing information is described as being (Burns and Wellings, 2009):

- The release time of the task.
- The completion time of the task.
- How to accommodate the possibility of timing requirements not being upheld.
- How to accommodate that the timing requirements can dynamically change as a consequence of e.g. *mode changes* described later.

It should be remarked that performance is only needed to the extent where the deadline can be met.

Besides being characterised by emphasising on temporal properties of events and tasks, real-time systems often exhibit a number of other characteristics which are outlined in the following:

**High Complexity** In contrast to many other systems, real-time systems are often closely interacting with their respective environments. Specifically, they often receive input from the environment such as sensor readings which the real-time system will act upon. Due to the real world exhibiting complexity, it naturally implies that the real-time system is capable of reacting to these in the most appropriate way (Wellings, 2004). For
2.1. REAL-TIME SYSTEMS IN GENERAL

example, a real-time system responsible for monitoring the pressure in a tank cannot solely focus on this property in isolation of the environment. This is because pressure is dependent on a variety of other physical phenomena such as temperature and may in the specific case also be dependent on humidity and the materials of the tank. Hence, in such a situation, the real-time system should to some extent monitor such properties as well for providing a correct conclusion on pressure.

In addition, complexity is further increased considering that many real-time systems are used for creating generic components. Complexity in this case is a product of having to correctly operate even if the context and environment are notably different. An example of these is a Electronic Stability Control (ESC) system which is used for controlling the brakes individually in cars for optimal brake performance. If this system is generic, it has to function in different car brands which may comprise interacting with a variety of other systems.

**Dependable** Laprie et al. (1995) describe a dependable system as a system which emphasises on reliability, safety, and availability. Such systems are often situated in environments where correct behaviour is of utmost importance. This is most likely attributed that humans put total reliance in correct operation and are willing to assign them tasks which would otherwise need to be carried out by humans. To avoid failure of such systems, a variety of measures can be taken. One approach to prevent faults from occurring is to ensure that the systems operate correctly according to their requirements, that is, they are functionally and temporally correct. Ensuring this may entail conducting program analysis such as model checking or static analysis. To conduct these, it is evident that the system is predictable. Here the high complexity may contribute negatively since it makes such analyses more difficult to conduct. (Wellings, 2004)

**Memory Management** Many real-time systems are resource-constrained devices, that is, small embedded systems with low power-consumption and an absolute minimum of resources including processing capabilities and memory. Since embedded systems are often shipped in large quantities, mitigating the resources available on the system to the absolute minimum may yield significant cost savings.

Considering that real-time systems are often situated in a dynamic environment, it will often be necessary to dynamically allocate memory for accommodating environmental changes. However, dynamically allocating memory on the heap may introduce undesirable properties of the system such as timing unpredictability because the time taken for allocations and deallocations cannot in general be guaranteed (Qing Li, 2003). For example, the heap may at some point become fragmented in which case a defragmentation process may be performed to rearrange the heap for preserving memory.

Another contributor to timing unpredictability is the garbage collector. The problem with the garbage collector is that it is often difficult or even impossible to predict when it will run and for how long because its execution depends on runtime decisions such as when the garbage collector assumes the system is idle or when additional memory allocations cannot be accommodated as a consequence of insufficient memory (Bacon et al., 2003; Knuth, 1997; Schoeberl, 2005).
CHAPTER 2. REAL-TIME SYSTEMS

![Diagram of a closed-loop control system](image)

**Figure 2.1:** A closed-loop control, which, if restricted by a set of timing requirements, represents a real-time system. The illustration is inspired by Barr (2002).

2.1.1 Formally Describing Real-Time Systems

Formally, a real-time system can be described using closed-loop control known from control theory. A closed-loop control, as depicted in Figure 2.1, reacts to changes in its environment, denoted the *plant*, in a timely manner. To react to the changes, continuous feedback is obtained from the plant through its *output*, which can be obtained by sensors. Using this feedback, the *control* unit can decide how to change the state of the plant. When the decision is made, an *input* is given to the plant, such that its state is changed using actuators. As a consequence of the input, a new output is generated and the loop starts over again. Example 1 shows a practical example of applying the presented theory. (Barr, 2002)

**Example 1 (A closed-loop control real-time system)**

Consider a fluid-control real-time system in a pipe keeping the pressure of fluid constant such that the components placed after the pipe can operate safely. To control the pressure, a sensor, in the form of a flow meter, and an actuator, in the form of a valve, are used. A processing unit is then responsible for conducting continuous sensor readings, and from these, decide how to influence the fluid pressure by controlling the valve having temporal requirements. In this example, the pipe and fluid therefore represent the plant which, given an input from the valve, produces an output that can be obtained by the flow meter. Finally, the processing unit represents the control from the closed-loop control. (Burns and Wellings, 2009; Barr, 2002)

2.2 Categorisation of Real-Time Systems

Real-time systems can be categorised into three major groups: *hard*, *soft*, and *firm* real-time systems. These categories describe how the real-time systems tolerate timing requirements not being upheld. (Qing Li, 2003; Burns and Wellings, 2009)
2.3. TASKS IN A REAL-TIME SYSTEM

Hard Real-Time Systems In these, failing to meet a deadline results in a failure of the system and the result of the failed computation is useless.

Soft Real-Time Systems In soft real-time systems, an exceeded deadline does not necessarily result in a failure, and the result of the computation is still useful, even though its usefulness is decreased. A possible reaction to a missed deadline could be to adapt and downscale the expectations to the performance and functionality.

Firm Real-Time Systems Firm real-time systems merge properties from both the soft and hard categories. Similar to hard real-time systems, the usefulness of a computation exceeding its deadline is zero, but exceeding a deadline does not imply a failure of operation. On this point, firm real-time systems are more like soft real-time systems in which an exceeded deadline is acceptable.

Furthermore, real-time systems can be categorised safety-critical systems. This denotes a class of systems whose failure during operation can imply a catastrophic outcome such as endangerment or even loss of lives. Correct operation of such systems often fully depend on temporal correctness, hence they are hard real-time systems.

In the remaining chapters, we will use the term real-time systems to denote hard real-time embedded systems unless otherwise stated.

2.3 Tasks in a Real-Time System

Tasks are processes or threads that are released by the system to handle events in a timely manner. This is accomplished by having deadlines and release patterns associated with the tasks. In the following, the periodic, aperiodic, and sporadic release patterns are explained:(Burns and Wellings, 2009; Qing Li, 2003)

Periodic Periodic tasks are periodically released by the system with fixed time intervals called periods. Tasks having this release pattern are easily accounted for in the schedulability analysis.

Aperiodic Aperiodic tasks are released by the system with an unknown frequency. These have serious implications to schedulability analysis, since they can potentially be released faster than they can be handled, hence the system might get overloaded and unable to uphold deadlines. Therefore, systems containing aperiodic tasks cannot generally be concluded schedulable unless restrictions are made. For instance, an execution-time server can be employed which is a task with the highest possible priority. Whenever a task is triggered by an aperiodic event, the execution-time server will handle its execution. The execution of this task is bounded because the execution-time server is allocated a specific amount of execution time and when this is exceeded, the server, and any tasks associated with it, stops.(Burns and Wellings, 2009)
CHAPTER 2. REAL-TIME SYSTEMS

Sporadic  Sporadic tasks are released by the system with a specified minimum inter-arrival time, that defines an upper frequency of the release of the task during a given period. In some cases, knowing the minimum inter-arrival time of sporadic tasks can be problematic, since it may be difficult to guarantee this for external events. The use of sporadic tasks therefore highly depends on the specification being correct. (Bøgholm et al., 2009; JSR302, 2010)

2.4 The Structure of a Real-Time System

The life-cycle of a real-time system can be viewed using the concept of missions which consist of three phases, namely initialisation, execution, and termination. These are shown in Figure 2.2 together with the overall setup and teardown phases, and are described below: (Schoeberl et al., 2007; Bøgholm et al., 2009; Wellings, 2004; Kwon et al., 2002)

![Figure 2.2: The phases involved in a real-time system’s lifetime. The figure is inspired by (Bøgholm et al., 2011)](image)

**Setup**  The setup phase initialises the overall system making it ready for executing its first mission. This may entail setting up loggers and perform object allocation for later use.

**Initialisation**  This phase is not time-critical and is tasked with preparing the mission for further usage by e.g. initialising the necessary objects that will be used in the subsequent phase.

**Execution**  In this phase, the tasks are released in order to react to the real-time system’s events and represents the system in its fully operational state. This is the time-critical part of the mission.

**Termination**  The termination phase is used to clean-up the mission after it has completed. This could be performing deallocation of objects and releasing locks. The mission can be re-initialised if it is needed or another mission can be set for execution. JSR302 (2010) suggests using a mission sequencer, which is responsible for executing a sequence of independent or repeated missions. This is depicted in the figure as Next mission. Furthermore, it has been suggested that missions can be nested (Bøgholm et al., 2009).
2.4. THE STRUCTURE OF A REAL-TIME SYSTEM

**Teardown**  When the system terminates, the teardown phase is entered in which it releases resources and performs necessary clean-up routines.

The normal life-cycle can, however, by dynamically changed as a consequence of *mode changes* (Wellings, 2004; Søndergaard et al., 2008). Mode changes are sometimes necessary since real-time systems are typically required to act upon current environmental conditions. As an example, consider a fly-by-wire aircraft which has a take-off, a cruising, and a landing mode (Wellings, 2004). In those modes, different operations may be performed with different temporal requirements and hence the transition from one mode to another is critical to avoid temporal properties to be violated. This is especially the case when the mode changes cannot be planned beforehand and are therefore unpredictable.

In the fly-by-wire system, mode changes are generally planned, but should an unforeseen event occur such as heavy turbulence, it may initiate that the current mode is interrupted instantly to change to another mode for accommodating the specific operations that need to be carried out in this particular situation.
As mentioned, hard real-time systems are not allowed to exceed their deadlines. To ensure this, a schedulability analysis can be conducted. This determines if the set of tasks comprising the system can be scheduled such that no deadlines are exceeded. The topic of schedulability analysis will be further described in Chapter 4. In order to conduct schedulability analysis, an integral component is the WCET of specific tasks, which can be determined using WCET analysis and will therefore be described in this chapter.

3.1 Control Flow Graph

Before describing how WCET analyses can be conducted, the concept of a Control Flow Graph (CFG) is described. A CFG can be used as foundation for different program analyses such as for WCET analysis and for compiler optimisation to e.g. determine infeasible paths.(Allen, 1970)

Before introducing the actual CFG, it is convenient to establish the definition of a basic block, as given in Definition 1.

Definition 1 (Basic Block)

A basic block is a block enclosing a number of instructions 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.(Allen, 1970)

Call instructions are normally not considered as branches since the control flow continues immediately after the callee has processed. However, it is important to note that given that it cannot be guaranteed whether the control flow continues immediately after processing the callee, it must be considered as a basic block itself.(Allen, 1970)

A CFG of a program, diagrammatically represents all the paths that are traversable during execution, thereby capturing the control flow of the program. Diagrammatically the CFG is based on a directed graph and hence can be seen as an abstract representation of the control flow where each vertex in the graph represents a basic block. Connecting the vertices by
edges captures the jumps in the control flow. Formally, a CFG is defined as described in Definition 2. (Lokuciejewski and Marwedel, 2011)

**Definition 2 (CFG)**
Formally, we can describe a CFG as a directed graph $G = (V, E, i)$, where the vertices, $V$, correspond to basic blocks and edges $E \subseteq V \times V$ connect two basic blocks $v_i, v_j \in V$ iff $v_j$ is executed immediately after $v_i$. $i \in V$ represents the start vertex, denoted as the source, which has no incoming edges, that is: $\forall v \in V : (v, i) \notin E$. (Lokuciejewski and Marwedel, 2011)

An execution path is defined as a sequence of interconnected basic blocks. Formally, an execution path is defined as in Definition 3.

**Definition 3 (Execution Path)**
An execution path $\pi$ through a control flow graph $G = (V, E, i)$ is defined as a sequence of basic blocks $(v_1, \ldots, v_n) \in V^*$, with $v_1 = i$ and $\forall j \in 1, \ldots, n - 1 : (v_j, v_{j+1}) \in E$. (Lokuciejewski and Marwedel, 2011)

### 3.2 WCET Analysis Overview

Generally, WCET analyses can be categorised into two distinct approaches, namely: static methods and measurement-based methods. The first is conducted by statically analysing the tasks whereas the latter is conducted by running the tasks and then measuring their execution times. Determining the WCET of a task is in general an undecidable problem. Therefore, conducting a non-exhaustive attempt will only be able to yield an approximation. This can then be tightened by introducing restrictions in the task such as bounding loops. (Bartlett et al., 2008)

WCET analyses are evaluated according to two essential criteria, namely safety and precision. Safety refers to the metric that represents whether the analysis in question produces WCET estimates that are minimum as high as the actual WCET of the task. Precision is the evaluation criteria used for describing how closely the WCET estimate resembles the actual WCET of the task, that is, the extent of the pessimism present in the estimate.

Figure 3.1 illustrates how WCETs estimated using either measurement-based methods or static methods relates to the actual WCET of a task. As shown, the measured WCETs are equal or smaller than the actual WCET, since a measured WCET represents only one of the possible execution paths. Unfortunately, it is difficult to ensure that the measurement is conducted under the worst possible conditions, and thus often results in unsafe estimates. In contrast, the statically estimated WCETs are typically constructed such that they do not underestimate, and are thus safe, while providing pessimistic results.
CHAPTER 3. WORST CASE EXECUTION TIME ANALYSIS

Figure 3.1: Distribution of execution times where the actual WCET, measured WCET, and estimated WCET are depicted.

Since safe WCETs are preferred over unsafe, the following will only concern the static analyses of WCETs. This is mainly because the focus is on hard real-time systems where the WCET estimates must be guaranteed to be safe.

Static WCET analysis can be divided into two analyses, namely a high- and low-level WCET analysis (Engblom et al., 1999). These are defined as follows:

High-Level WCET Analysis The high-level WCET analysis concerns the task’s control flow. The control flow can be represented using a CFG which is a directed graph consisting of vertices, denoted as basic blocks, and edges. Each vertex represents a set of sequentially executed instructions. Connecting the basic blocks with edges then represents the control flow of the task capturing e.g. branches. In this context, the high-level analysis concerns the WCET of the task based on the control flow, and thus assumes that the cost of the basic blocks are known, e.g. through low-level analysis.

Generating the CFG from the compiled program is preferable due to accounting for optimisations introduced by the compiler. This is important since compiler optimisations significantly affect the WCETs of a program (Bøgholm et al., 2008a).

Low-Level WCET Analysis As described, the low-level WCET analysis concerns the execution time of individual basic blocks, and specifically the individual machine code instructions they represent. This involves estimating the execution time of individual instructions on the target execution environment, which raises a number of concerns. First of all, these are often not made available by the vendors. Besides, a variety of mechanisms used for reducing the average execution time, such as caching and pipelining, must be taken into account since their absence in the analyses yield very pessimistic results.

These analyses are further complicated by the presence of an OS and even a virtual machine, since they add additional layers of logic between the program itself and the hardware. As an example, system calls would require additional analysis of the OS.
Another property, which can greatly affect WCET analysis, is timing anomalies. If timing anomalies can occur in an execution environment, it is possible, that changes to the execution time of a single instruction, can counter-intuitively influence the global execution time. If this is possible, the analysis must take into account all possible paths and execution orders of a task, removing many possibilities of optimising the analysis such as discarding specific execution paths. An example of timing anomalies is depicted in Figure 3.2. (Lundqvist and Stenstrom, 2002; Gebhard, 2010)

**Figure 3.2:** Timing anomaly resulting in cache miss yielding the fastest execution time. The figure is inspired by Gebhard (2010).

As shown, the system in the example has two computing units, namely an ALU and an FPU. The combination of having these and having dependabilities on the instructions results in timing anomalies where the cache hit of A results in longest global execution time since C has not yet been dispatched, and B is therefore executed.

The following describes the high- and low-level analyses in more detail and further elaborates on a number of techniques used to conduct these.

### 3.3 High-Level WCET Analysis

As stated, high-level WCET analysis concerns the control flow of a task. For this to succeed, the control flow of the task must itself be analysable. Therefore, it must not contain constructs which make this impossible. Furthermore, constructs which results in very pessimistic WCET estimates should preferably be avoided as well. Some of these are described in the following (Puschner and Koza, 1989):

**Unbounded Loops** If loops are unbounded, the WCET cannot be determined since the loop potentially can be endless. The high-level analysis can try to establish the loop bound, e.g. by Data-Flow Analysis (DFA), or else require the user to provide annotations stating
the loop bound. Using a DFA allows for determining loop bounds for simple cases automatically, reducing the concern of loop bounds for the developer and reduces the chance of errors. This is preferable especially since it can be difficult for a programmer to know the exact loop bound at all times (Engblom et al., 1999).

Recursion  Recursion introduces similar problems as unbounded loops. Specifically, the maximal depth of the recursive call cannot be statically determined. Furthermore, recursion introduces memory issues since the stack space usage is dependent on the depth of the recursion.

Function Pointers  Using function pointers can make the WCET analysis difficult since it can be impossible using static code analysis to determine which method is actually called on runtime. Therefore, it can be necessary to estimate the WCET of a function pointer to be that of the function having the highest WCET in the program. Even though this will give safe WCET estimates, they are potentially very pessimistic.

Dynamic Method Dispatch  It can turn difficult to determine the exact method which is called when calling a virtual method of a superclass. A solution is similar to that of function pointers where the WCET of a virtual method is determined to be that of the implementation of the virtual method having the highest WCET.

In the following, three examples of high-level WCET analyses are given, namely path-based WCET analysis, Implicit Path Enumeration Technique (IPET), and model-based WCET analysis.

### 3.3.1 Path-Based WCET Analysis

One approach to high-level WCET analysis is to transform the CFG into a weighted graph, in which the execution cost of each individual basic block is used as weights. Standard graph algorithms can then be used to identify the longest path in the graph, which in turn results in the WCET of analysed task (Engblom et al., 2000; Wilhelm et al., 2008). One specific approach proposed by Engblom et al. (2000) consists of four steps where the final three are primarily used to increase precision.

1. Find the longest path in the graph using a standard graph algorithm.
2. Check if the path is feasible which can be done using e.g. DFA.
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.

Note that the complexity of analysing such a graph is exponential with the depth of conditional statements and thus poses some difficulty in analysing large systems (Huber and Puschner, 2010).
3.3. Implicit Path Enumeration Technique

IPET (Li and Malik, 1995) can be used to calculate the WCET of a task, using its CFG and
costs of its individual basic blocks. In principle, the WCET is calculated by summarising the
worst-case execution frequencies of the basic blocks multiplied with their costs, which can be
expressed as the following Integer Linear Programming (ILP) problem:

\[ WCET = \sum_{i=1}^{N} c_i x_i, \]

where \( N \) is the number of basic blocks, \( c_i \) is the cost, and \( x_i \) is the execution frequency
of basic block \( i \). To calculate the WCET, bounds for the execution frequencies of the basic
blocks must be determined and the resulting equation is maximised. Thus, bounds must be
determined for the execution frequencies, or else the result of maximising the equation would
be infinite. The bounds can be expressed using a set of constraints expressed for the control
flow of the program. These constraints are:

Structural Constraints  Constraints on the control flow of the program such as branches.
These can be extracted from the CFG.

Functional Constraints  Constraints on the behaviour of the control flow, such as loop bounds.
These can be extracted by annotations or by static analyses such as DFA.

The extraction process of constraints can be shown through a practical example. In Listing 3.1
a simple while loop containing an \textit{if-else} statement is shown. To determine the WCET
of this using IPET, assume that the prior low-level WCET analysis has been conducted.

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

Listing 3.1: Excerpt of a program for which WCET can be estimated using IPET.

First, the CFG which is shown in Figure 3.3 is constructed from the program. The
CFG consists of the basic blocks denoted \( B_0, B_1, \ldots, B_5 \). Furthermore, each basic block have
corresponding execution frequencies \( x_0, x_1, \ldots, x_5 \) and each transition between basic blocks
have corresponding transition frequencies \( d_0, d_1, \ldots, d_8 \).
Structural Constraints

Simple structural constraints can be expressed by equations stating the execution frequencies each basic block, \( x_0, x_1, \ldots, x_i \), as the sum of all its incoming transitions or the sum of all outgoing transitions. The frequency \( x_i \) for a given basic block \( i \) is thus given from the equation:

\[
x_i = \sum_{j \in I_i} d_j = \sum_{k \in O_i} d_k,
\]

in which \( I_i \) is the set of all incoming transitions to basic block \( i \), and \( O_i \) is the set of all outgoing edges from basic block \( i \). By repeating this process for each basic block, a set of constraints such as the following can be constructed for the CFG of the code example:
3.3. HIGH-LEVEL WCET ANALYSIS

\[
\begin{align*}
  x_0 &= d_0 + d_6 = d_1 + d_7 \\
  x_1 &= d_1 = d_2 + d_3 \\
  x_2 &= d_2 = d_4 \\
  x_3 &= d_3 = d_5 \\
  x_4 &= d_4 + d_5 = d_6 \\
  x_5 &= d_7 = d_8
\end{align*}
\]

Functional Constraints

A common functional constraint is loop bounds. A loop consists of a loop head, which is the entry basic block to the loop, and the loop body which is the basic blocks executed as part of the loop. A functional constraint is that the number of edges from the loop body to the loop head must be less than or equal to the number of incoming edges to the loop head, which are not originating from the loop body, multiplied with the loop bound. This is expressed in the equation (Schoeberl et al., 2010):

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

where \(C_h\) is the set of edges from loop body to loop head, and \(E_h\) is the set of incoming edges entering the loop head not originating from the loop body. Finally, \(n\) is the loop bound. From the ongoing example, the following functional constraint can be expressed:

\[
d_6 \leq n \cdot d_0.
\]

Calculating the WCET

The above equations are then solved using an ILP solver, which finds the execution frequencies resulting in the maximum value, in this case, the WCET. Solving these equations is an NP-complete problem in the general case, even though Li and Malik (1995) have shown that the nature of the problem often results in equations which can be solved in polynomial time.

3.3.3 Model-Based WCET Analysis

The CFG of a task can be transformed into a model that can be subject for model checking (Dalsgaard et al., 2009; Metzner, 2004). This can be achieved using timed automata. Depending on the approach, either basic blocks or instructions are modelled as vertices with
CHAPTER 3. WORST CASE EXECUTION TIME ANALYSIS

associated costs. For model checking, a number of different model checkers exist which can explore the model to estimate the WCET by determining which state has the highest observed total execution time. (Metzner, 2004; Schoeberrl et al., 2010; Bøgholm et al., 2008a)

Figure 3.4 illustrates an example of how a loop can be modelled by incrementing a counter each time the loop is iterated. The iteration count is ensured by only allowing to proceed to the loop body when \( i < maxBound \).

![Figure 3.4: Modelling a loop with loop bounds in a notation similar to UPPAAL.](image)

As shown, this captures the control flow of a loop in a model. However, to conduct the WCET analysis, it is necessary to add execution times to the modelled basic blocks. Following the example where basic blocks are modelled, Figure 3.5 depicts how the execution times of basic blocks can be included in the model.

![Figure 3.5: Modelling a control flow including execution times for each basic block.](image)

Using a global clock and a local clock, it is possible to simulate the execution times of each basic block while keeping track of the global execution time. Initially, both clocks are reset, and for each basic block, the combination of invariants and guards models the execution times. Each time an edge is followed, the local clock is reset such that it can be reused to model the execution time of the following basic block. When the exit block is reached, the highest observable value of the global clock is the WCET of the control flow, in this case 9.
Comparisons of IPET with model-based approaches have shown IPET to be a more time efficient solution, while model checking potentially yields preciser estimates depending on the level of detail included in the models (Schoeberl et al., 2010; Metzner, 2004). This is because the behaviour of the program can be more explicitly expressed in a model while in IPET, only static properties are considered. Regardless, a model-based approach may well be worth the effort since a preciser WCET estimate may deem that other and more cheap hardware resources may suffice. In such cases, the savings can be of significant importance especially considering if the system is part of a product that is produced in high quantities.

3.4 Low-Level Analysis

As stated, the purpose of the low-level analysis is to determine the WCET of a single basic block, which then can be supplied to the high-level WCET analysis. In its most simplistic form, this can be determined by using the stated execution times for individual instructions of the CPU and summing these for a basic block. Unfortunately, such information is not always provided since it may be confidential. Furthermore, modern processors include a number of mechanisms for increasing average performance. These should be included in the analysis to avoid overly pessimistic estimates as observed by Dalsgaard et al. (2009) who have shown that including caches in the estimation increases precision with up to 97%. More importantly, if timing anomalies can occur, taking these mechanisms into account can make the difference between safe and unsafe estimates. (Lokuciejewski and Marwedel, 2011; Wilhelm et al., 2008)

Two common approaches for low-level WCET analysis taking into account the processor behaviour are model-based WCET analysis (Dalsgaard et al., 2009) and abstract interpretation (Ferdinand, 2004; Theiling et al., 1999). Both of these approaches aim at describing the processor and important mechanisms, to determine the WCET of individual instructions, or a number of instructions executed sequentially. According to Metzner (2004), model checking potentially provides preciser results when compared to abstract interpretation. However, in respect to program analysis they have generally shown to have the same properties, meaning that using one over the other in some cases is a matter of preferences (Schmidt and Steffen, 1998). In the following, model-based WCET estimation will be described in further detail.

3.4.1 Model-Based WCET Analysis

The construction of a model for the low-level WCET analysis can be done in different ways. One approach is to construct a model for calculating the WCET of a single instruction, another is to calculate the WCET of a series of instructions. In practice this can be modelled using timed automata. (Dalsgaard et al., 2009; Metzner, 2004)

As an example of a model-based low-level WCET analysis, Dalsgaard et al. (2009) have shown how to model the cache and pipeline of an ARM processor. These models can then be used to calculate the execution time for a given set of instructions on the platform. Furthermore, they observe that for most embedded platforms, the caching can be generically
modelled given a set of parameters such as size and speed. This implies that the models can be reused for a wide range of different hardware in a processor family.

In respect to pipeline modelling, Dalsgaard et al. (2009) suggest to model each stage as individual timed automata. By synchronising those, it is possible to simulate the behaviour of the pipeline. Figure 3.6 depicts a five stage pipeline consisting of the stages fetch, decode, execute, memory, and writeback.

As shown, each stage is clear and ready to process new instructions as soon as a synchronisation with the next stage has been made. The program drives the pipeline by continually initiating the fetch stage.
Scheduling Real-Time Systems

The purpose of this chapter is to describe how schedulability analysis is conducted. Even though TetaJ is a WCET analysis tool, understanding its application domain is important since it may affect its design.

Showing that a task’s WCET is smaller than its deadline, only means that the task can meet its deadline when executed in isolation. However, since real-time systems often consist of multiple tasks executing concurrently, it must be shown that they meet their deadlines under circumstances where tasks are preempted, blocked, etc. Therefore, the scheduler of the system must be analysed to determine if there exists an ordering of the tasks yielding a schedulable system. This can be shown by conducting schedulability analysis. (Burns and Wellings, 2009; Wellings, 2004)

The following sections outline some well-known scheduling policies and schedulability analysis approaches.

4.1 Scheduling Policies

Generally, there are two groups of scheduling policies, namely: static policies and dynamic policies. The behaviour of the static policies can be predicted before executing the actual system, whereas the dynamic policies rely on decisions made at run-time. Three of the most typically described scheduling policies in the literature of real-time systems are: Cyclic Executive Scheduling (CES), Fixed-Priority Scheduling (FPS), and Earliest Deadline First Scheduling (EDF). Of these, only the latter is a dynamic scheduling policy. Since EDF is not supported by most Real-Time Operating Systems (RTOSs) and programming languages, we will only describe CES and FPS in the following. (Burns and Wellings, 2009)

4.1.1 Cyclic Executive Scheduling

This policy requires the schedule to be constructed for a known number of tasks before run-time. Decomposing each task into one or more procedures, the schedule is constructed such that the procedures, can be executed sequentially resulting in all tasks meeting their deadlines. The ordering consists of minor cycles which start periodically and runs in a fixed amount of
time. The set of minor cycles needed to execute all procedures constitute a major cycle. An example of a schedule is shown in Figure 4.1. (Burns and Wellings, 2009)

![Figure 4.1: Example of CES with minor and major cycles. As an example, consider task a. From the figure, it can be observed that a has a WCET of 2 and a period of 10.](image)

As shown, the system is not fully loaded in all minor cycles and is therefore idling sometimes, waiting for the period of the next minor cycle before executing the next procedure. This ensures that procedures are not executed too early which can be problematic if a procedure e.g. depends on periodic external events.

Due to its simplicity, CES has been used in many real-time systems. One of its advantages is that, since the order of all procedures is known before run-time, no locking mechanisms are needed for shared resources. Another advantage is that schedulability is proved by construction of the schedule. That is, no explicit schedulability analysis is conducted. However, one of the problems with CES is that the schedules become very complex for larger systems, where especially constructing the schedule itself and maintaining it during the lifetime of the system can be difficult. (Burns and Wellings, 2009)

### 4.1.2 Fixed Priority Scheduling

Using FPS, tasks are given fixed priorities before run-time and, afterwards, the scheduler always executes the task with the highest priority. (Burns and Wellings, 2009)

There exist two common priority assignment schemes, namely: rate-monotonic priority assignment and deadline-monotonic priority assignment. In rate-monotonic, the priorities are assigned according to the length of the tasks' periods. That is, the shorter period a task has, the higher priority it is assigned. Deadline-monotonic is a similar scheme. However, instead of assigning priorities according to periods, they are assigned according to deadlines. The advantage of using either of the two schemes is that they are optimal assignment schemes, meaning that if a priority ordering exists such that all tasks meet their deadlines, then the priority ordering given by the two schemes yields a schedulable system. (Liu and Layland, 1973; Leung and Whitehead, 1982)

In contrast to CES, tasks can be executed concurrently using FPS. As a consequence of this, care must be taken to avoid priority inversion when tasks share resources using locking mechanisms. To avoid such unwanted situations, protocols can be used. One of these is the
priority inheritance protocol, where the priority of a blocking task is set to that of any higher priority suspended task. This means that a task operates with two different priorities, namely base and active. By adhering to the protocol, the active priority can be larger than or equal to the base priority. (Wellings, 2004; Sha et al., 1990)

4.2 Schedulability Analysis

Schedulability analysis can roughly be grouped into traditional methods, such as utilisation-based schedulability analysis, and modern methods, such as model-based schedulability analysis. The advantage of the traditional methods is that they are relatively simple and for limited cases can be made by hand (SW9-Project, 2010). However, this is at the expense of precision of the analysis because they do not consider e.g. dynamic release patterns. To incorporate this, model-based schedulability analyses can be used (Bøgholm et al., 2008a). Since FPS is the most commonly used schedulability policy, this policy will be assumed in the following.

4.2.1 Utilisation-Based Schedulability Analysis

Utilisation-based schedulability analysis can be used in systems with a simplistic view of process representation. For instance, it is assumed that all tasks must be periodic with a deadline equal to their period, \( D = T \), and that they do not share resources. The analysis is given by: (Liu and Layland, 1973)

\[
\sum_{i=1}^{N} \left( \frac{C_i}{T_i} \right) \leq N \left( 2^{\frac{T}{N}} - 1 \right), \quad T = D, \tag{4.1}
\]

where \( C_i \) is the WCET for task \( i \), \( T_i \) is its period, and \( N \) denotes the total number of tasks. The \( C_i/T_i \) term describes the utilisation of task \( i \).

If it can be shown that a set of tasks fulfill the test, the system constituted by the tasks is schedulable. However, if the set of tasks does not fulfill the test, it is not guaranteed that they cannot be scheduled to meet their deadlines. In other terms, the utilisation-based schedulability analysis is said to be sufficient but not necessary. (Burns and Wellings, 2009)

4.2.2 Model-based Schedulability Analysis

The idea in this type of analysis is to model system behaviour and the given schedulability policy. Modelling allows for analysing many different systems containing periodic and sporadic tasks, concurrency, execution sequence dependencies, and synchronisation.

In some cases, traditional approaches may deem a system non-schedulable even if it is. This can be due to the inadequacy of modelling system behaviour and specifically, traditional methods are insensitive to many parts of program execution thereby resulting in significant
over-approximations. Often such behaviour can be modelled using the model-based approach, and hence it may be possible to conclude the system schedulable (Bøgholm et al., 2008a). As an example, consider the tasks with corresponding utilisations in Table 4.1.

<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>Period/Deadline</th>
<th>WCET</th>
<th>Priority</th>
<th>Utilisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Periodic</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0.50</td>
</tr>
<tr>
<td>b</td>
<td>Sporadic</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>Sporadic</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 4.1: A set of tasks with period/deadline, WCET, priority, and utilisation.

As shown, the system has one periodic task and two sporadic tasks. Since the total utilisation is 100, the system is not schedulable according to the utilisation-based schedulability analysis. The reason for this is that the sporadic tasks are expected to be released at each minimum inter-arrival time. However, suppose the sporadic tasks are released by the periodic task executing the code shown in Listing 4.1.

```java
boolean blocked = isSensorBlocked();
if(blocked == true){
    fire(b);
}
else{
    fire(c);
}
```

Listing 4.1: Release pattern of the two sporadic tasks $b$ and $c$.

As shown, it is impossible for both sporadic tasks to be released at the same time. The real utilisation is therefore 75, but this cannot be concluded by the utilisation-based schedulability analysis. However, in the model-based analysis it would be possible to model the release pattern and hence deem the system schedulable. An example of this is depicted in Figure 4.2, which is a model of the release pattern of the two sporadic tasks using the TIMES schedulability analysis tool (Amnell et al., 2004).

As shown, the release pattern is modelled such that either task B or C is fired according to the value of $\textit{blocked}$.
Figure 4.2: TIMES model of the release pattern of the two sporadic tasks presented in Listing 4.1.
Java in the traditional sense does not facilitate the development of real-time systems due to the absence of various real-time concepts, such as the notion of a deadline. Besides, traditional Java includes a number of mechanisms and concepts making real-time systems development difficult. The canonical example is the garbage collector for which timing guarantees generally cannot be established.

To make Java suitable for real-time systems development, great effort has been put in specifying a series of alterations to the traditional Java platform that enable real-time systems to be developed. The purpose of this chapter is to introduce some of these efforts namely the Real-Time Specification for Java (RTSJ) and the Java profiles: SCJ, and PJ. In respect to TetaJ these are important since they provide a restricted programming model that is amenable to WCET analysis. Thus they provide knowledge about the code that TetaJ should be capable of supporting.

5.1 The Real-Time Specification for Java

The RTSJ extends the JVM and Java class libraries in order to facilitate real-time development with Java. The initial version 1.0 of the specification was conceived in 2001 as JSR 1 (SUN, 2006), and version 1.1 is currently being developed as JSR 282 (Oracle, 2009). This description is based on the current stable version 1.02 of the specification (RTSJ.org, 2010).

The following gives an overview of some of the changes introduced by the RTSJ:

Memory Management    Java does not provide means for explicitly deallocating memory. Instead a garbage collector is normally used in various JVMs. Unfortunately, as noted by Wellings (2004), garbage collection affects the timing predictability of real-time systems, which in turn complicates WCET analysis. Some research has been conducted on the subject of real-time garbage collection (Schoeberl, 2009; Bacon et al., 2003), but the RTSJ does not rely solely on this yet. Instead, the RTSJ provides a memory model which avoids garbage collection, providing time predictable memory allocation and deallocation.
This new memory model consists of two new memory areas in addition to the heap which can be used when allocating new objects. The two new areas are *immortal memory*, which does not allow deallocation of the objects, and *scoped memory*, which explicitly allows for allocation and deallocation of its objects.

**Schedulable Objects and Scheduling** Java supports threads and priorities, but the Java specification does not guarantee that the threads are executed in accordance with their priorities if e.g. the underlying OS does not support them (Wellings, 2004). Furthermore, Java does not support expressing concepts such as release patterns in real-time systems. This is solved in the RTSJ through two changes. One is a more general notion of *schedulable objects* and a number of classes used to specify important properties of these, such as their priority and period. Furthermore, the JVM must be more strict in its notion of priorities such that real-time scheduling policies can be adhered to.

**Asynchronous Event Handling and Timers** In addition to the threading model, a model of asynchronous event handlers has been added which allows for executing schedulable objects on specific events. These events can be fired from the software itself, hardware interrupts, periodically by timers, etc. This model co-exists with the threading model and provides an alternative approach to executing tasks in a real-time system.

**Asynchronous Transfer of Control (ATC)** ATC is a mechanism which allows schedulable objects to interrupt each other in order to change their control flow. This mechanism is motivated by the need to implement forms of error recovery and mode change which needs to take effect quickly.

**Physical and Raw Memory Access** In real-time systems, and especially embedded real-time systems, it is often necessary to handle physical memory. This may be the case if one needs to interact with memory-mapped devices or differentiate between different types of memory connected to the system. To support this, the RTSJ provides facilities to specify the placement of memory areas and to access memory more directly than allowed in traditional Java.

**Time Values and Clocks** Time is inherently an important part of real-time systems, thus the RTSJ provides enhancements to Java in the form of a high resolution timer, with support for representing times down to one nano-second, given a hardware clock supporting this. The timer is supported by a concept of clocks which show the passage of time and concepts of absolute and relative time used in various parts of the API.

### 5.2 Java Real-Time Profiles

The changes introduced by the RTSJ result in much more tight semantics to Java with regards to scheduling and synchronisation, and avoids the usage of garbage collection (Kwon et al., 2002). It is very broad and allows for developing a wide range of real-time systems in Java, however this is at the cost of complexity and increased requirements to the implementation.
of the JVM (Henties et al., 2009). This has two effects: one is increased difficulties in implementing a resource-effective JVM for embedded systems and another is increased difficulties in conducting program analysis to determine WCETs and ensure both functional and timing correctness.

In order to address these problems, a number of profiles have been developed, targeting both embedded and safety-critical systems. Common for these is that they take into account the RTSJ, extending it with additional concepts and limiting the parts complicating analysis. In the following, two different profiles are described, namely the SCJ profile and the PJ profile.

### 5.2.1 Safety-Critical Java

The SCJ profile is targeted safety-critical systems and builds upon the RTSJ in order to provide real-time facilities to these systems. The profile is currently being standardised through JSR-302 (JSR302, 2010). SCJ limits the RTSJ through inheritance and certain parts of the RTSJ are disallowed. The significant changes consist of a changed programming model with the concept missions and compliance levels.

**Missions**

A mission consists of a number of periodic and aperiodic tasks logically grouped together. The missions have an orderly initialisation phase, in which all memory must be allocated, and a termination phase, in which schedulable objects are permitted to finish before terminating the mission like the general concept of missions introduced in Section 2.4. A mission sequencer is used to specify the order of the missions.

**Compliance Levels**

Certifying real-time systems is highly dependent on their complexity and therefore to make this process easier for small systems, compliance levels are introduced. The SCJ defines three compliance levels, 0, 1, and 2, which restricts the allowed elements in the SCJ. Compliance level 0 provides a very restricted set of facilities, with the advantage of a much more certifiable and analysable program. If more features are needed, level 1 and 2 are available, at the cost of being more complex to analyse.

Level 0 only allows the CES scheduling algorithm, hence the mission can be thought of as a set of computations that are executed periodically in a clock-driven timeline. Even though multiple periodic tasks are used in the system, only one thread is controlling the execution. Therefore, there is no synchronisation concerns when using this level. At level 1, the FPS scheduling policy is used. Furthermore, a single mission sequence with concurrently running threads is allowed. These can be in form of periodic and aperiodic event handlers. Finally, at level 2, the application starts with a single mission, but from this, it is possible to create more missions sequencers and make missions execute concurrently.

### 5.2.2 Predictable Java

Due to the ongoing process of standardising the SCJ profile, the PJ profile has been developed to present alternative design choices to SCJ (Bøgholm et al., 2009). The profile primarily con-
tributes by redefining the inheritance relationship to RTSJ and by redefining the programming model for missions. These are further described in the following:

**Approach for Extending the RTSJ**  The SCJ inheritance relationship to the RTSJ is based on inheritance for limitation which means that subclasses do not satisfy the specification of their parent. Therefore, this results in cluttered code. Furthermore, inheritance for limitation requires that annotations describe which parts of the RTSJ are allowed by SCJ. This has the undesirable property of relying on external conformance checkers that can examine the source code to determine whether the application conforms to the profile or not. The accommodate these problems, the PJ profile is a generalisation of the RTSJ.

The new inheritance relationship gives rise to problems since the RTSJ is already standardised, and hence cannot be defined as subclasses of the PJ profile. Bøgholm et al. (2009) suggest to solve this by using an adapter layer, that disallows some RTSJ methods and gives default values to some parameters.

**Missions as First Class Event Handlers**  The missions in PJ are similar to those in SCJ in the sense that they have three phases. Furthermore, the missions are a set of tasks, implemented as event handlers, which collectively provide the desired functionality of the real-time system. Generally, missions can be thought of as containers of tasks, however, due to their different phases, more logic is needed. This is recognised by the PJ profile, and it is suggested that missions, themselves, should be event handlers. Using this approach, the constructor of the event handler becomes the initialisation phase, and the `handleEvent()` method becomes the termination phase. Adding the mission to the scheduler then initiates its constituting tasks to enter their execution phase.
Part II

Analysis
Knowledge about JBC and the JVM is imperative for understanding the mechanisms that affect WCET of Java programs. The primary source of the following is The Java Virtual Machine Specification (Lindholm and Yellin, 1999).

The JVM is a stack machine and essentially prescribes a complete execution environment for programming languages targeted at the JBC instruction set such as Java, with the purpose of being independent of the underlying hardware and OS. Platform independence is achieved since the JVM can be implemented for the platforms of interest and may even be independent of an OS. This is achieved with the premise that the programs are not utilising platform-dependent functionality accessed by native methods or using non-standard runtime libraries. The general flow from source code to executable code is shown in Figure 6.1.

![Tombstone diagram showing the translation process of Java and subsequent execution on a JVM.](image)

As shown, the Java program is compiled into JBC, which can be uploaded to an arbitrary JVM running on an arbitrary processor.

### 6.1 Java Bytecode

Before introducing the actual JBC instructions, it is necessary to understand how the JVM operates on them. The JVM includes a component called the class loader whose responsibility is to load the class files containing the JBC, that is, both the class files of the code written by the programmer and the referenced class files from the Java API.
CHAPTER 6. THE JAVA VIRTUAL MACHINE

If dynamic class loading is supported, only those class files from the Java API that are actually referenced will be considered. How the actual dynamic class loading process is to be carried out is unspecified. Most implementations load entire classes whenever referenced whereas others try to limit this by only loading the actual JBC used in order to preserve memory. The prime benefit from dynamic class loading is reduced runtime overhead in terms of memory consumption however at the expense of execution time. Dynamic class loading, can be a problem for real-time systems if the class to be loaded is not known at compile time, since it complicates or even makes the WCET analysis impossible. For this reason, the RTSJ (RTSJ.org, 2010), disallows its usage.

The loaded class files are verified to ensure a variety of properties are upheld such as ensuring that the class file can actually be parsed to prevent that it will eventually crash the JVM. Next, the classes are linked and subject to be either interpreted or compiled to native machine code instructions. In respect to the WCET analysis, the verification of class files is problematic since it is conducted at runtime together with class loading. (Venners, 2000)

The central focus of the JBC instruction set is the operand stack. This means that when e.g. performing arithmetic calculations such as addition, the two operands are first pushed onto the stack. Next, the addition operator is applied which pops the two operands of the stack and stores the result of the addition operation back onto the stack.

The JVM does not contain registers for storing arbitrary values, but each method has a set of local variables which the instruction set, in effect, treats as a set of registers that are referred to using indices. If the result of the addition operation is to be stored in a local variable, the opcode referring to the specific index of the local variable or an opcode with the index as operand must be used.

To illustrate how JBC is represented, consider the Java code shown in Listing 6.1 which iteratively calculates the 7th Fibonacci number.

```java
public class JbcExample {
    public static void main() {
        int f = 0, g = 1, N = 7;
        for (int i = 1; i <= N; i++) {
            f = f + g;
            g = f - g;
            System.out.println(f);
        }
    }
}
```

Listing 6.1: Java code iteratively calculating the 7th Fibonacci.

Compiling this to a class file and afterwards disassembling it using `javap`, generates a representation of the JBC using the mnemonics for the opcodes. The disassembled class file is shown in Listing 6.2.
public class JbcExample extends java.lang.Object{
public JbcExample();
  Code:
  0:  aload_0
  1:  invokespecial #1; // Method java/lang/Object."<init>":()V
  4:  return

public static void main();
  Code:
  0:  iconst_0 // int f = 0, g = 1, N = 7;
  1:  istore_1
  2:  iconst_1
  3:  istore_2
  4:  bipush 7
  6:  istore_3
  7:  iconst_1 // for (int i = 1; i <= N; i++)
  8:  istore 4
  10: iload 4
  12: iload_3
  13: if_icmpgt 37
  16: iload_1 // f = f + g;
  17: iload_2
  19: iadd
  21: iload_1 // g = f - g;
  22: iload_2
  24: istore_1
  25: iload_1
  27: iload_2
  29: isub
  31: getstatic #2; // Field java/lang/System.out:Ljava/io/PrintStream;
  33: iload 1
  35: invokevirtual #3; // Method java/io/PrintStream.println : (I)V
  37: iinc 4, 1
  39: goto 10
  42: return
}

Listing 6.2: JBC of the Java code calculating the 7th Fibonacci number obtained by using the javap disassembler.

Much of the code is relatively self-explanatory so following will only focus on the essentials of invoking a method. To print out the calculated Fibonacci number, the invokevirtual opcode is used in line 31. The invokevirtual opcode assumes that an object reference is placed on top of the stack containing the method to be invoked. In the case of printing the Fibonacci number, a reference to PrintStream is obtained by using the getstatic opcode in line 29. Followed by the object reference on the stack, is a number of arguments that corresponds to the arguments used for invoking the method on the particular object. In this
example, a single argument is provided to the `println()` method; namely the value contained in the f local variable which is pushed onto the stack using the iload_1 opcode in line 30.

## 6.2 Compilation Techniques

Parts of the JVM specification are rather loosely defined. The specification describes some essential components every implementation of the JVM must include to be compliant but leaves out many details. For example, the JVM specification dictates that every JVM must be able to execute JBC contained in Java class files. In particular it specifies what the JVM must do when executing each JBC instruction. However, adherence to the specification does not rely on the technique used for compiling the JBC to the native platform for execution. (Venners, 2000)

The fact that the JVM specification does not enforce a compilation strategy allows for flexibility that enables JVM implementers to tailor the JVM to the domain in which it is used. For example, for embedded systems, a small memory footprint may be desirable and, thus, a compilation strategy that addresses low memory consumption is appropriate even if it is at the expense of reduced performance.

The JVM specification does not specify whether the JVM is implemented in software or in hardware. The latter of these two options allows for native execution of JBC as seen in a variety of hardware implementations such as the Java Optimized Processor (JOP) (JOP, 2008), and aJile Systems’ aJ-100 (aJile Systems, 2000). The focus in this project, however, is on software implementations of the JVM whose compilation process is generally conducted using either of the three approaches: interpretation, Just-In-Time (JIT) compilation, and Ahead-Of-Time (AOT) compilation. These are described in the following:(Venners, 2000)

### 6.2.1 Interpretation

In this technique, the JVM reads one JBC instruction at a time and emulates the execution of it on the target platform. Examples of JVMs incorporating this technique comprise HVM (Korsholm, 2011b), JamVM (Lougher, 2010), and early versions of the JVM from Sun (India, 2011) which was later changed to use JIT compilation.

Interpretation can take two forms: iterative interpretation and recursive interpretation. Iterative interpretation is suitable for source languages where the instructions are primitive and do not contain sub-components. In this scheme, instructions are continuously fetched, analysed, and executed one after another. Recursive interpretation is more sophisticated in that instructions may contain subcomponents that must be decomposed before execution.

JBC allows for iterative interpretation and can be implemented very simplistically using a single while loop, denoted the interpreter loop, in which JBC instructions are continuously fetched, analysed, and executed. In the fetch phase, the interpreter will retrieve the next JBC instruction to execute. The analysis phase involves decomposing the JBC instruction into its constituents, that is, opcode and operands, and to locate the implementation of the specific
6.2. COMPILATION TECHNIQUES

Figure 6.2: Graph showing the memory usage of a Java application run using either interpretation or AOT compilation, respectively.

opcode. The actual execution phase can be implemented using a switch statement where each case corresponds to a particular JBC opcode with corresponding implementation. (Watt and Brown, 2000)

Interpretation poses some serious considerations if performance is an important criteria, since it makes interpretation of JBC significantly slower than native execution (Venners, 2000). This is because, in general, interpreters do not make use of profiling to reveal frequently used execution patterns and many other optimisations. Moreover, the result of interpreting is not cached for benefiting reusability.

With regard to statically compiling JBC into machine code using AOT compilation, interpretation also poses the deficiency of relying on the interpreter itself being loaded into memory which increases the memory footprint. However, an important note is that JBC is more compact than the machine code generated from it. Hence, it follows, that although the interpreter adds overhead in terms of memory, there exists a point where AOT compiled programs will surpass the size of the corresponding program using an interpreter, as illustrated by Figure 6.2.

The advantage of using interpretation in favour of other compilation techniques is its simplistic nature. Many of the optimisation mechanisms used in JIT require a more complex implementation and further impose overhead; especially in the startup phase of the program where JBC is often interpreted while being profiled and monitored for potential optimisation. The memory footprint is therefore lower using an interpreter instead of JIT compilation which is also the primary reason for adopting this compilation technique for JVMs residing on resource-constrained devices such as mobile phones. Specifically, this is the case for DalvikVM until release of Android 2.2 (Android, 2010). Also, a very popular adopter of interpretation is the Java Micro Edition (Java ME) platform with its Connected Limited Device Configuration (CLDC) configuration that specifies, that the underlying K Virtual Machine (KVM) employs a bytecode interpreter (Microsystems, 2000).

In respect to WCET analysis, interpreted JBC can be analysed if their individual implementations have deterministic behaviour. Hence, if the relationship between JBC instructions and their implementation can be described through some abstractions, the predictability of interpreted JBC is to some extent similar to that of programs written in a native programming language like C. This means that the techniques used for predicting the behaviour of
JBC programs can draw inspiration from those for programs written in native programming languages which has underwent much research and has relatively well established techniques such as that conducted by Dalsgaard et al. (2009); AbsInt (2009). However, in respect to describing the JBC instruction implementations, this has not been subject to other research than using an unsafe measurement-based approach (Hu et al., 2002b).

6.2.2 Just-In-Time Compilation

JIT compilation refers to applying a hybrid approach of interpretation and statically compiling the code. Particularly, the actual compilation process is deferred until runtime similarly to the approach adopted in interpretation. The compilation process is typically conducted on a method level or class level instead of per instruction as in interpretation. In addition, the translated code is cached for benefiting that it may be used later which might be the case when executing loops or frequently executing the same method.

The goal of using JIT compilation is to merge the benefits of interpretation and AOT compilation. JIT compilation still promotes portability due to relying on JBC and not machine code while still having the benefit of fast execution. Fast execution is achieved by applying a series of runtime optimisations some of which cannot statically be determined. For example, some optimisation techniques rely on the actual execution patterns of the program. To show the performances achievable with JIT compilation, Sestoft (2005b) has shown that JIT compiled code potentially perform at least as good as AOT compiled code. (Venners, 2000)

Some JIT compilers suffer from a startup delay which is the consequence of conducting the compilation step when the program is loaded. The initial start-up delay is also affected by the amount of optimisations that are performed on the code. Hence, there exists a trade-off between efficient code generation and the time required to compile and execute it. To address this issue, some JIT compilers, notably the HotSpot JVM (Oracle, 2010), incorporate an adaptive optimisation strategy which dynamically performs recompilation of portions of the code based on the current execution profile. For instance, when the program is loaded, the JIT compiler may choose an interpretive scheme such that the startup delay is circumvented. During interpretation, the JIT compiler tries to establish an execution profile and based on this, the code is recompiled for optimal performance and cached for later usage.

JIT compilers are able to achieve higher runtime performances than solely using interpretation (Venners, 2000). This is, however, at the cost of higher memory demands which is attributed the size of the JIT compiler itself, its various optimisation mechanisms, and most notably due to caching the translated code. Due to the runtime performance, many JVMs targeting at high performance use JIT compilation; notably Java HotSpot which is the primary virtual machine for desktops and servers. Other common JVMs in this category comprise Apache Harmony (Apache, 2011), and CACAO (CACAOVM, 2009).

This approach results in timing unpredictability, since it can be difficult to predict when parts of the program are recompiled and how the runtime optimisations affect the execution time. Given the current research in JIT compilation techniques in relation with real-time systems, JIT compilation is not feasible for being used. It would, however, be very interesting
to examine further, because JIT compilation offers many benefits over interpretation techniques. The problems that need to be solved for adopting JIT compilation would encompass statically determining when and how the dynamic behaviour of compiling to native code will be conducted. (Bate et al., 2002a; Auerbach et al., 2007)

6.3. REQUIREMENTS FOR THE JVM USED BY TETAJ

6.2.3 Ahead-Of-Time Compilation

This type of compilation technique refers to compiling the JBC into a system-dependent executable prior to runtime. This can achieve better performance in some cases since no optimisation of code is done at runtime and no generated code needs to be cached for later usage. Furthermore, some optimisation techniques that are considered too costly to perform at runtime in JIT compilers can be conducted by AOT compilers. AOT compilation can be useful for resource-constrained embedded systems that need good performance, however at the expense of increased memory overhead. This is because the runtime system itself is not as demanding as using the JIT compilation technique while still achieving good performance since the code is statically compiled for the target platform.

The timing predictability of AOT compiled programs depends on the program and JVM elements which are combined in the final executable. Essentially this does not differ greatly from much of the existing work on predictability of programs written in programming languages like C or C++ that are compiled to native code (Hu et al., 2004; AbsInt, 2009; Dalsgaard et al., 2009).

The range of available AOT compilers for the JVM is limited. Some of the most notable include GCJ (GNU, 2009), Excelsior JET (LLC, 2011), and Ovm (Ovm-Project, 2009).

6.3 Requirements for the JVM used by TetaJ

The analysis presented in this chapter gives rise to a number of concerns regarding the JVM in relation to predictability of the hosted programs. These are described in the following sections.

6.3.1 Compilation Technique

A great concern regarding the JVM is the employed compilation technique. Only little research has been conducted in the area of JIT compilation of applications with real-time properties (Auerbach et al., 2007). Traditional JIT compilation evidently poses problems regarding WCET analysis. Hence, even if JIT compilation offers better performance than solely using interpretation, the WCET analysis may likely deem that higher WCETs will result using the JIT compilation scheme. Even if sophisticated WCET analysis is conducted to predict its dynamic nature, JIT compilation poses the deficiency of having higher memory consumption compared to interpretation and AOT compilation. Also, taking into account the limited resources available in embedded systems, JIT compilation may therefore be infeasible.
The choice of compilation strategy is hence limited to either AOT compilation or interpretation. AOT compilation has the benefit of offering better performance than interpretation but performance is not a prime objective of real-time systems. In these, predictability is the key criteria and performance should only be sufficient for upholding the temporal requirements. Since AOT compilation has the problem of higher memory demands than interpretation for larger programs, AOT compilation is not necessarily an advantage for embedded real-time systems. It is however not possible to conclude which of AOT compilation and interpretation is the most applicable for Java for real-time systems because of the trade-off between memory and processing power overheads, both of which are equally important aspects of an embedded system. However, we have chosen to focus on interpretation, since to our best knowledge the majority of JVMs available for resource-constrained devices employs this technique (Korsholm, 2011b; Harbaum, 2006; Lougher, 2010; Pizlo et al., 2009; Armbruster et al., 2007).

6.3.2 WCET Analysis of JBC

It is evident that it would be advantageous to base the WCET analysis of Java code on JBC. The primary reason is that the compiler optimisations performed by the Java compiler are incorporated in the subsequent analyses. Even though Java compilers such as javac and jikes do not perform many optimisations (Sestoft, 2005a), Bøgholm et al. (2008a) have shown that the model-based WCET techniques are sensitive, in terms of verification time, to the JBC produced by different compilers for the same source code. Additionally, another motivation is that other programming languages like Python and Ada for which compilers targeting JBC exist, would be analysable by TetaJ as well. Even though it is not the focus of our project to make TetaJ support multiple languages, using JBC as target implicitly and thereby effortless allows this.

6.3.3 Dynamic Class Loading

The JVM must not employ dynamic class loading features, since this introduces unpredictable behaviour. Instead, a JVM suitable for real-time systems should employ static linking or linking at program initialisation. This, however, introduces new concerns such as how to statically determine the classes needed prior to runtime. However, in various AOT compilation-based JVMs such as OVM and also in the interpretation-based HVM where dynamic class loading is absent, the problem has shown to be practically solvable. In the case of HVM, it has been solved using intelligent class linking where the application is analysed to determine which parts of its libraries may be accessed at runtime (Korsholm, 2011b).

6.3.4 Limiting the Size of the Executable

Due to this project emphasising on resource-constrained devices, another great concern is how to limit the size of the executable. Using the Java API significantly increases this, which in some cases may preclude using Java in this context. As an example, consider the `println()`
method often used for printing to standard output. In this case, the entire System class needs to be loaded. The problem in here is that the loaded class contains a great amount of unused code having a substantial effect on the final size of the application. Mechanisms for circumventing this problem have been proposed. Notably, ProGuard (ProGuard, 2011) employs shrinking and optimisation of Java class files while remaining functionally equivalent to the original source. This process is conducted by static code analysis where unused classes, fields, and methods are removed. Furthermore, an obfuscation process is used to shrink the Java class files further by renaming all identifier names to short ones.
When developing a tool aiding software development, one needs to consider how it suits in existing development practices. In the case of TetaJ, this concerns the development process for embedded real-time software. Hence, the practices used in these may influence the design of the tool.

### 7.1 HRT-HOOD

Hard Real-Time HOOD (HRT-HOOD) (Burns and Wellings, 1994) is a structured design method with emphasis on object-oriented design with concrete object types for representing common hard real-time abstractions. This includes the notion of a wide variety of concepts such as the possibility of associating an object with its specific release pattern. Also, it is possible to include the notion of scheduling policies, explicit timing requirements, and facilities and tools to allow schedulability analysis to be conducted. The prime purpose of HRT-HOOD is to allow issues such as timeliness and dependability to be considered much earlier in the development process. Burns and Wellings (1994) recognise that traditionally, timeliness, dependability and in general non-functional requirements are first considered late in the process, at a point where it is costly to make changes. Specifically timeliness is often postponed from consideration until the testing phase, in which case it may be associated with high costs if this requirement cannot be met by the application since a complete rewrite may be the only solution. To make the development process focus on these aspects sooner, it is based upon an iterative approach as outlined in Figure 7.1.

Initially, when using HRT-HOOD, one must state the systems requirements and from these, the logical architecture design can be made. This design is not dependent on the constraints of the execution environment. That is, the design is primarily concerned with the functional requirements. The logical architecture design is then used in the physical architecture design to map logical design to the physical resources provided by the target system. This e.g. requires knowledge regarding the behaviour of the given hardware. As depicted in Figure 7.1, the process of deducing the physical architecture design is typically an iterative process of refining the logical architecture design and physical architecture design. (Burns and Wellings, 1994)
7.2 SARTS

Having the architectural design activities completed, a detailed design is made from which the implementation can be done. As part of the implementation, timing estimations, such as WCET, are conducted. If the temporal requirements are not fulfilled, the architectural design activities can be repeated. However, if they are met, it is suggested that a measurement-based timing analysis is conducted to ensure that the system meets its temporal requirements in a practical deployment. (Burns and Wellings, 1994)

7.2 SARTS

The Schedulability Analyzer for Real-Time Systems (SARTS) project (Bøgholm et al., 2008a) has proposed a development process consisting of five steps in which models are used extensively for verifying both functional correctness and temporal correctness of a real-time system. The process is depicted in Figure 7.2. (Bøgholm et al., 2008c)

In the first step, the specification of requirements are provided as part of a request for a real-time system. In contrast to a specification of requirements in normal development this includes information of deadlines, periods, etc. With this, an initial model can be constructed to ensure properties such as schedulability are fulfilled. The advantage of creating the initial model is that it is a cost-effective way of determining early in the development whether or not the specific system can be developed such that it fulfils its requirements. If step two is passed, the actual system can be implemented, in step three. Afterwards, in step four, the implementation is analysed to obtain information such as memory consumption and WCET. Finally, in step five, a model of the implementation can be constructed. This model can then be used to verify both temporal and functional correctness. Temporal correctness can be determined by conducting schedulability analysis. Furthermore, it is possible to compare the model of the implementation with that of the specification derived in step two. This final step can be used to verify whether the implementation and the specification are consistent. (Bøgholm et al., 2008c)
7.3 The Context for TetaJ

Based on the above, we have identified that TetaJ can be used right after the implementation step. This especially applies for SARTS, where it forms the cornerstone to allow schedulability analysis to be conducted. With respect to HRT-HOOD, this also applies, since it does not distinguish between the implementation step and the WCET analysis. Furthermore, due to the iterative nature of HRT-HOOD, it is important that the WCET analysis can be conducted in-between iterations and within reasonable time. This is important to avoid having the development team wait unnecessarily long before proceeding with the development. Furthermore, it is also important that TetaJ will be usable for small program parts, that is, it must support that the implementation is only partly available as a consequence of the iterative process.

In the development process, TetaJ must be designed to require minimal intervention to reduce the effort required to use it. This is to support that the system in question is developed iteratively and hence require that TetaJ will often need to be used. This implies focusing on simplifying the usage of the tool, by limiting the number of concerns for the developers. For TetaJ, the execution environment rarely changes, and hence removing this concern in the general usage is desirable. This also supports the situation in which the development team is substituted by another one or that new developers are assigned a task in which TetaJ is also used. However, to make this usage possible, it is necessary that TetaJ is provided necessary information about the execution environment at the initial step in the development process. We envision it to be the responsibility of researchers, or hardware and JVM vendors to provide information of the components constituting the execution environment, since a general application developer is not expected to proficient in this.
The purpose of this chapter is to describe existing WCET tools. This is done in order to identify essential components and key choices that at least should be reasoned about in the design of TetaJ.

We have chosen to focus on tools for WCET analysis of Java code to understand how the Java programming language influences design choices of a WCET tool. In this relation, we have chosen to analyse WCET Analyzer (WCA) (Schoebel et al., 2010), and the more framework-based tool XRTJ (Hu et al., 2004). Besides, more general WCET tools suitable for being adopted in a variety of settings have been chosen to identify general design characteristics. In this relation, we have chosen to analyse METAMOC (Dalsgaard et al., 2009) and aiT (AbsInt, 2009).

8.1 aiT

AbsInt has developed a commercial WCET analysis tool named aiT which uses static analysis techniques to create both safe and precise WCET estimates. Currently aiT supports a wide range of different processors, including ARM7, LEON2, LEON3, and various PowerPC processors. Additional processors are supported by supplying additional cache analysis and pipeline analysis components.

The WCET analysis is conducted through a combination of abstract interpretation for low-level WCET analysis and IPET for high-level WCET analysis. Figure 8.1 depicts the process of aiT’s WCET analysis.

As input, aiT takes an executable program created from a predefined set of supported compilers and programming languages. Currently, aiT offers support for subsets of C, C++, and Ada. The reason for only supporting subsets is that functionality such as suspending tasks using e.g. a call to \texttt{wait()} in C cannot be accounted for in the WCET analysis.

The executable program is disassembled and analysed by the \textit{CFG builder}, which build a CFG in their own language called Control flow Representation Language (CRL). The CRL is then passed to the \textit{value analysis}, \textit{cache analysis}, and \textit{pipeline analysis} components whose
analyses build upon abstract interpretation (Cousot and Cousot, 1977). The purposes of the three components are as follows:

**Value Analysis** The value analysis is responsible for analysing the program in order to determine the possible values of its variables. This is used both to determine loop bounds, memory addresses calculated through indirect memory access, and targets of branches. Determining the targets allows for pruning the CFG for branches not needed when estimating the WCET. Using value analysis for determining indirect memory access is necessary in order to reason about cache hits and cache misses in regard to e.g. pointers. In the case that some of these values cannot be determined, user annotations can be provided either in a separate file used by the CFG builder or directly in the source code.

**Cache Analysis** The cache analysis determines if a cache access results in a cache-hit or a cache-miss. The analysis supports the Least Recently Used (LRU) replacement strategy, and since common microprocessors use non-LRU replacement strategies, the analysis has to be modified to reflect some of these.

**Pipeline Analysis** The pipeline analysis is the final step in conducting the low-level WCET analysis. The result of this is an updated version of the CRL amenable to high-level WCET analysis.

Finally, the result from the pipeline analysis and the CRL is transformed into an ILP problem by the *ILP generator* which is then solved by the *LP solver* component returning the final WCET estimate.
METAMOC is a framework for determining the WCET of tasks from an executable. An essential design choice of METAMOC is flexibility. In particular, METAMOC allows for tailoring the WCET analysis to correctly reflect the target hardware and to take into account various mechanisms used in modern processors for enhancing performance. Including these, has shown to yield WCET estimates that are up to 97% preciser. This is realised by encapsulating the logic handling aspects such as the cache, pipeline, and main memory, in separate components and can therefore easily be substituted to accommodate other target hardware. Collectively, these form the METAMOC hardware model. This approach is similar to the one taken by aiT, differing mainly in the implementation techniques used. Figure 8.2 gives an overview of METAMOC.

![Figure 8.2: Overview of the constituents of METAMOC. The illustration is inspired by (Dalsgaard et al., 2009).](image)

In METAMOC, models are used to model the individual components of the architecture and the analysed task itself. The models of the analysed task are constructed for each of its functions using CFGs obtained from the disassembled executable containing the task.

The models created by METAMOC are combined such that they, in effect, are able to simulate an abstract execution of each instruction. This is done by letting each instruction of the program synchronise with the model of the pipeline. The model of the pipeline further synchronises with the model of the caches and finally the caches synchronise with the model of main memory. The WCET of the program is calculated using the UPPAAL model checker which maintains a global cycle counter clock and returns the highest value identified after exploring the full state space of the models. Currently, models have been created for ARM- and AVR-based processors.

To account for potential unbounded loops, METAMOC assumes the programmer bounds these explicitly in the source code using annotations. The annotations are comment-based and are of the form `@loop_bound` followed by an integer representing the loop bound.
8.3 WCET Analyzer

The WCA tool (Schoeberl et al., 2010) is used to conduct WCET analysis of real-time Java tasks running on the JOP (Schoeberl, 2005). Since the JOP contains a JVM implemented in hardware, the employed techniques resemble those used for analysing assembly on other platforms such as ARM and make the problem of WCET analysis of Java simpler. Furthermore, the JOP has specifically been designed to be timing predictable.

To limit the amount of user annotations for loop bounds, WCA uses a DFA to extract these (We have previously questioned the correctness of the current DFA implementation, see SW9-Project (2010) for more information). If however the implemented DFA is not capable of determining the loop bounds, user annotations must be provided in form of comments such as `// @WCA loop <= 10`. Besides deriving loop bounds, DFA is also used to perform receiver-type analysis, which is used to precisening the WCET on dynamic method dispatch.

WCA takes Java class files as input to its analysis and uses the BCEL library (Apache-Software-Foundation, 2006) to reconstruct the CFG of the method of interest. WCA allows the CFG to be used as input to either IPET or model-based WCET analysis. When using the IPET method, WCA transforms the CFG into an ILP problem for which the ILP solver `lp_solver` (Gourvest et al., 2010) is used to determine the WCET. Otherwise, if the model-based method is used, WCA transforms the CFG into a UPPAAL model where each location represents a basic block. WCA then makes an overestimated guess of the WCET for the provided code, and checks whether the global clock ever exceeds this. If not, a binary search is used to make the WCET preciser.

Implementing both IPET and model-based WCET analysis methods may seem unnecessary, but it is argued that they can be used for different purposes. The reason is that since the model-based approach generally yields preciser WCETs at the cost of the analysis time, this method should be used for smaller and important parts of the analysed code. The decisive factor in the precision of the WCETs is the ability to describe the hardware behaviour in the two approaches. In the model-based method, an exact model of the cache can be created, and as shown by Huber and Schoeberl (2009), this yields significantly preciser WCETs than with IPET.

8.4 eXtensible high-integrity Real-Time Java

eXtensible high-integrity Real-Time Java (XRTJ) (Hu, 2004) is not an actual WCET tool; instead it is a framework primarily intended for providing a predictable and portable programming environment to develop distributed and high-integrity real-time systems in Java (Hu et al., 2004). Specifically it introduces a predictable programming model, a static analysis environment, and a runtime environment appropriate for distributed systems and real-time systems. Also, its general design reflects the development process in which it is used. This is achieved by introducing tools and environments appropriate for different phases of the development.
The programming model of Java is changed in accordance with the Java predictability problems described in Chapter 5. The static analysis environment is based on multiple components. Specifically, it introduces a special annotation-format called eXtensible Annotations Class (XAC) which is used for specifying information that cannot be expressed directly in the Java class files. In addition, the XRTJ-Analyser is used for evaluating both functional and timing correctness. (Hu et al., 2002a)

Similar to some of the other tools, XRTJ uses comment-based annotations. However, XRTJ differentiates itself by extracting the annotations from the source code and stores them in XAC files afterwards. The motivation for this is that the annotation files are not bound to a particular programming language such as Java, but also allows annotations from other programming languages that are eventually compiled to JBC to be represented in this format.

For determining the WCET, XRTJ employs a static analysis approach which harnesses information specified in annotations and special timing models called Virtual Machine Timing Models (VMTMs) (Bate et al., 2002b). VMTMs were initially proposed by Bate et al. (2002b) and are essentially tables describing the timings of JBC instructions on the particular hardware including OS and JVM. The WCET analysis is therefore highly dependent on the VMTMs, meaning that their JBC instruction execution times must be safe and precise. Hu (2004) describes that deriving the VMTMs using static analysis is sophisticated. Therefore, it is suggested to use a measurement-based approach. Also, it is suggested that common sequences of JBC instructions can be provided in the VMTMs. This allows taking into account hardware optimisations such as pipelines. An excerpt of a VMTM table is shown in Table 8.1.

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>99.95th percentile</th>
<th>99.90th percentile</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>aload</td>
<td>48</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>bipush</td>
<td>40</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>iload</td>
<td>46</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>iastore</td>
<td>55</td>
<td>41</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 8.1: An excerpt of a VMTM derived by measurements. The excerpt is from (Hu, 2004).

8.5 Ideas for TetaJ

A number of interesting properties have been observed for the analysed WCET tools. In the following, these properties are compared and a number of choices are made to form the basic design of TetaJ.

8.5.1 Technique for WCET Estimation

The most essential part of TetaJ is the actual WCET estimation, and how this is conducted. The analysed tools present a number of different approaches to this problem, using combi-
nations of abstract interpretation, IPET, and models. These are either used separately for the high- and low-level analyses, or as proposed by Schoeberl et al. (2010) used to conduct estimations with different levels of precision and estimation time. We do not consider using multiple methods for this purpose, because we find it more important to provide safe and precise WCET estimates. The reasoning behind emphasising these criteria, at the expense of estimation time, is that preciser WCETs potentially reveal that lower resource requirements may suffice. This may in turn result in production costs savings that may very well be worth the additional estimation time.

The choice between models, abstract interpretation, and IPET is not straightforward. Each of the methods have been shown to be sufficient for WCET analysis. In TetaJ the same WCET technique for both high- and low-level WCET analyses is used in order to reduce the number of concepts the developers and maintainers of the tool must understand. As WCET technique, models and abstract interpretation are preferred over IPET mainly because of the potentially preciser WCET estimates. The choice between models and abstract interpretation is solely based on existing experience using models during past projects (SW8-Project, 2010; SW9-Project, 2010) since these techniques can have similar properties for program analysis (Schmidt and Steffen, 1998). In addition, taking into account the substantial efforts (Behrmann et al., 2002) that have been put in optimising the model checking process, this technique is indeed feasible even for relatively large analysis problems in terms of model checking time and memory consumption. Hence using model checking complies with the goal of using TetaJ in an iterative development process. Consequently, we find it natural to apply model checking for WCET estimation which is further supported by the observation that CFGs can easily be represented as models (Bøgholm et al., 2008a; Dalsgaard et al., 2009; Schoeberl et al., 2010).

8.5.2 Accounting for a Software Implementation of the JVM

We think that building a WCET tool for Java with the prime purpose of conducting WCET analysis of Java is essential. Existing tools for WCET analysis on Java programs such as WCA are tightly bound to the underlying hardware and JVM. Since one of the primary objectives of Java is portability, we think that the tools narrow the applicability of Java for real-time systems development. Hence, in order to preserve this design goal of Java and widen the possibilities for real-time systems development, we want to adopt similar ideas as those proposed in METAMOC and aiT. However, the design of aiT and METAMOC does not directly incorporate the notion of a JVM which may prove difficult to do. XRTJ partly does this by using VMTMs. However, we think that XRTJ is flawed because the VMTMs do not have the explicit notion of hardware and JVM individually, thus removing some flexibility. Furthermore, the VMTMs are based on measurements of executing each individual JBC instruction on the particular processor. This is problematic for hard real-time systems because it can yield unsafe WCETs e.g. if the execution of a JBC instruction changes the state of the JVM and thereby influences the execution times of subsequent JBC instructions. Instead, we envision, a JVM model describing its actual behaviour.
8.5.3 Accounting for Underlying Hardware

Multiple WCET tools have focused on including the hardware behaviour in the WCET analysis (Dalsgaard et al., 2009; AbsInt, 2009). We have assessed that describing the hardware behaviour ourselves is associated with great effort, and since this is not the focus of this project, we reuse the models for the hardware created by METAMOC. This is possible since the project is open sourced and hence the models are publicly available, which contrasts those of aiT. Furthermore, due to METAMOC being an ongoing project, more and more hardware is modelled thereby increasing the amount of hardware analysable with TetaJ.

8.5.4 Accounting for Loops

All loops must be bounded in order to conduct the WCET analysis. This can be solved using techniques such as DFA or manual annotations. The general approach has been to provide fixed loop bounds according to the specific functionality the system serves, however, should the system be used in another context it may require the loop bounds to be derived again. Therefore another approach suggested by Hunt et al. (2006) consists of annotating loop bounds using the Java Modelling Language (JML) (Leavens, 2011), and using the KeY tool (Beckert et al., 2007; of Karlsruhe, 2011) to determine the loop bounds symbolically. This means that the bounds depend on certain input, and when this input is available the correct bounds can be derived relatively easy. However, such an analyses, and even the DFA itself, requires much effort in order to be implemented and is therefore considered out of scope of this project.

Instead, we have chosen to explicitly declare loop bounds using annotations in the code. This approach is simpler without reducing the relevance of our tool. In this regard, we want to make use of the annotation strategy that yields the simplest implementation.

We assume an annotation strategy based on comments to be the simplest to implement and this approach is therefore taken. A disadvantage of this decision is that Java source files and class files are required for all the code to be analysed. This can be a problem if external libraries are used where only the class files are available. In such situations, a future solution could be to allow the library provider to specify annotations in separate files.
As stated in Chapter 8, TetaJ will use models, and specifically UPPAAL models, in order to conduct the WCET analysis. This chapter establishes the necessary knowledge for using UPPAAL for WCET analysis and is mainly based on Behrmann et al. (2004) and Bengtsson et al. (1996).

9.1 Overview

UPPAAL is a model checker tool that for a given system, modelled as an Network of Timed Automata (NTA), can verify properties of the model. The UPPAAL distribution consists of two components: a model checker backend, and a GUI that eases the task of modelling the system in a NTA and initiates the verification process. In addition, to harness the resources that may be available on application servers, UPPAAL allows the verification process to be conducted using the model checker backend.

The timed automata in UPPAAL are called templates and instances of these are called processes. Collectively, templates form the NTA system. Note that after this chapter, we do not explicitly distinguish between templates and UPPAAL NTA systems. They will collectively be referred to as models. It should be clear from the context which semantics model has.

The templates comprise a number of locations connected by edges which have a number of attributes e.g. describing constraints that must be fulfilled before firing an edge. Furthermore, each template has an associated code file, in which local variables and functions can be specified in a C-like language.

The following two sections describe the templates and the UPPAAL query language.

9.2 Templates

To ease the following description, the templates depicted in Figure 9.1 are used as reference. The figure depicts two templates, which, when instantiated, synchronise with each other in periods of 100 time units.
9.2. TEMPLATES

Locations in templates can be annotated with the following attributes:

**Name**   The *name* of a location is used as an identifier for the specific location, and can be used in the verification process for referring it.

**Initial Location** Each template has exactly one *initial location*. This type of location is marked with a double circle as is the case for *IdleMaster* and *IdleSlave* in the example.

**Invariant** *Invariants* are used to describe conditions that must be upheld when the process is in that location. In the example, the invariant on *IdleMaster* dictates that it can only be visited as long as the clock variable, *time*, is below or equal to 100.

**Committed** Locations can be marked as *committed*, represented by \( C \). This means that time is frozen when entering the location and an outgoing edge must be taken in the next transition immediately. Committed states can therefore be used to model atomic operations. In the example, *InSyncMaster* and *InSyncSlave* are committed locations.

**Urgent** Marking a location as *urgent*, symbolised with \( U \), is less strict than marking it committed. An urgent location is similar to adding an extra clock to a location, which on all incoming edges is set to zero. Furthermore, the location is associated with an invariant \( x \leq 0 \). The difference between this and a committed location is that the system is not forced to take one of the outgoing edges as the next transition, as long as time remains frozen.

Similar to the locations, the edges can be annotated with a set of attributes. These are described in the following:

**Selections** Annotating an edge with *selection* allows a value to be non-deterministically selected from a type. A type in UPPAAL is e.g. a scalar set or a bounded integer. In the example, a selection is made on the outgoing edge of *IdleMaster*, meaning that the *id* of the process being synchronised is selected. If, for instance, two processes of the *Slave* template were instantiated, the select statement would non-deterministically select one of them.

---

![Figure 9.1: Example of two templates, MASTER and SLAVE.](image)

---
CHAPTER 9. UPPAAL

Synchronisations Channels are used to synchronise processes. In the example, a synchronisation is made by issuing `synchronising!` in the `MASTER` template. This represents a signal being sent through a channel, to a receiver issuing `synchronising?`. After the synchronisation, the involved processes change locations at the same time. If the synchronisation is made between two processes, it is called a binary synchronisation. However, synchronisations can also be made broadcast, meaning that zero or multiple receivers synchronise on the signal. Finally, synchronisations can be urgent, to enforce that no delay occurs.

Guards This is a condition that must be upheld before firing the corresponding edge. For example, consider the outgoing edge of `IdleMaster`. This has a guard dictating that `time` must be equal to 100 before the edge leading to `InSyncMaster` can be fired.

Updates Update expressions are evaluated when their associated edge is fired. This is used in a wide range of scenarios such as assigning new values to system variables as with the `time` variable, which is updated when firing the outgoing edge of `InSyncMaster`.

9.3 The Query Language

One of the essential purposes of modelling the system is software verification, that is, ensuring that the system upholds its dictated requirements. After modelling the system, the requirements can be specified using the UPPAAL query language. The query language presents a variety of formulae that enables requirements to be explicitly represented and can broadly be categorised into two different classes: state formulae and path formulae.

State formulae are used to express the states of the timed automata and can consist of boolean expressions such as `foo == true` or expressions containing locations such as `ProcessA.locationX` denoting whether a particular process is in the particular location. Here, `ProcessA` denotes the particular process of interest, and `locationX` represents the particular location of that process. UPPAAL introduces the special state formula `deadlock` which can be used for testing whether the models have reached a deadlock state.

Path formulae are used to express the various paths the process may visit and are distinguished by three different classes. Particularly, path formulae can either concern reachability, safety, or liveness. The following gives a description of these and Figure 9.2 may serve as a convenient reference to understand how these are associated with the models in UPPAAL.

Reachability This class concerns whether a particular state formula, $\phi$, eventually can be satisfied by a reachable location. UPPAAL introduces the notion $E\Diamond\phi$ for describing these path formulae.

Safety State formulae that will possibly or never occur can be expressed using this class of path formulae. A prime example of this class is ensuring that a deadlock of the model will never occur. UPPAAL introduces two different path formulae for these purposes. $A\Box\phi$ is used to describe that in all reachable states, $\phi$ must be true. The other query
\( E \Box \phi \) describes that there should exist a maximal path where the state formula, \( \phi \), is true in all states.

**Liveness** This class is used to represent that something will eventually happen. For this, the UPPAAL path formula \( A \Diamond \phi \) describes that the state formula, \( \phi \), is eventually satisfied on every path. Besides, UPPAAL also introduces the path formula \( \phi \leadsto \psi \) which describes that whenever \( \phi \) is satisfied, eventually \( \psi \) is also satisfied.

In the development version 4.1.3 of UPPAAL, two new queries, namely the `sup` query and the `inf` query, were introduced to the query language. These are described below:

**sup query** The `sup` query, \( \text{sup}\{\phi\} : \text{var} \), is always satisfied but besides this, the query returns the suprema of the expression and variable provided to it. That is, it can be used to determine the maximum value of integers, clocks, etc.

**inf query** The `inf` query, \( \text{inf}\{\phi\} : \text{var} \), is similar to the sup but for infima. This means that the query can be used to determine the lowest observed value of the provided expression and variable.

![Figure 9.2: Representation of the path formulae supported in the UPPAAL query language. States that are marked represent states where the state formula \( \phi \) holds. The figure is inspired by Behrmann et al. (2004).](image_url)
9.4 Applying UPPAAL for WCET Analysis

Using UPPAAL models for WCET analyses implies that the models must be constructed such that queries can be used to determine the WCET of the modelled task. For this, there are several approaches. The general approach has been to use the reachability path formulae to conduct the analysis (Schoeberl et al., 2010). The problem with this is that it requires an initial guess of the WCET from which a tighter WCET can be approached by e.g. a binary search. The reachability approach is shown in Listing 9.1.

1 $\text{A} \parallel (\text{MainMethod.Terminate imply clock} \leq 3000)$

**Listing 9.1:** Reachability path formula used for determining whether a value of a clock is ever exceeded.

As shown, the formula is used to check whether $\text{clock}$ is always below or equal to 3000 when the MAINMETHOD process is in its Terminate location. If this is the case, the checked WCET can be decreased and the property can be verified again, meaning that the result converge to the WCET. This can be a problem for larger models where one check requires a significant amount of verification time. Therefore, since the introduction of the sup query, the approach for WCET analysis in UPPAAL has been to apply this (Dalsgaard et al., 2009). It can be used in either of the two ways shown in Listing 9.2.

1 sup{MainMethod.Terminate}:clock
2 sup:clock

**Listing 9.2:** Applying the sup query to determine the highest observed value of the clock.

The first of the queries can be used to determine the highest observed value of $\text{clock}$ when the MAINMETHOD process is in its Terminate location. The other query is more broad, and can be used to determine the highest observed value of $\text{clock}$ in the whole UPPAAL NTA system.

It should be remarked that the the Best Case Execution Time (BCET) also can be determined using the query language. In some cases this is important to know to see how big the interval between BCET and WCET is (Wilhelm, 2004). Listing 9.3 shows an example of how the BCET can be determined using the inf query.

1 inf{MainMethod.Terminate}:clock

**Listing 9.3:** Applying the inf query to determine the lowest observed value of the clock.
9.5 Progres Measures

UPPAAL models with large state space use considerable amounts of memory. To limit the memory consumption, UPPAAL features progress measures. The idea of these is to describe progress in the model such that UPPAAL can prune the states that are no longer necessary. In respect to program analysis, progress measures can be applied on the control flow since the flow naturally describes the progress of the program.

The progress measures are applied by monotonically incrementing them when progress is made in the model. That is, for the progress measure $pm$, $pm_{s1} < pm_{s2}$ must hold if there is an edge from location $s1$ to $s2$. Figure 9.3 depicts an example of applying the progress measures.

As shown, the progress measure, $pm$, is incremented at each edge since the flow never retreats. The vertical black lines indicate that progress has been made and that the states behind the line can be removed from memory.
Part III

The Tool
This chapter gives an overview of TetaJ, how it is to be used in practice, its capabilities, and its essential components. The most interesting of these are further elaborated in the subsequent chapters. For a more detailed description of the system architecture, see Appendix A.

10.1 The Structure of TetaJ

Recall, that the WCET of a program depends on the execution environment on which it is executed. Thus, both the JVM and the underlying hardware must be taken into account in the analysis. For this purpose, it was decided in Chapter 8 that TetaJ must employ a model-based approach in which timed automata, amenable to be model checked using UPPAAL, are used. Specifically, TetaJ uses three layers of models as illustrated in Figure 10.1.

![Diagram of the WCET analysis process]

**Figure 10.1:** The WCET analysis is conducted by combining three layers of UPPAAL models, a JBC layer, a JVM layer, and a hardware layer.

The JBC model layer represents the top level behaviour of the JBC task itself, and is effectively a model representing the control flow of the compiled Java task of the program. Hence, modelling techniques must be employed for representing basic blocks and their interconnection, loops, and method calls including native method calls. All these are further described in Chapter 13. As this model is explored, synchronisations are used to synchronise with the JVM model layer and thereby simulating the execution of JBC.

The JVM model layer must be constructed to reflect how the target JVM processes the individual JBC instructions. The models used in this layer represent all implementations
of the JBC instructions. The JVM model is expected to be provided to the developers before the development initiates. The JVM model synchronizes with the hardware model layer to simulate executing the implementation of the JBC instructions.

The hardware model layer models the actual execution on the concrete platform given the compiled native instructions of the JVM. Similar to the JVM model, the models in this layer are expected to be provided in the initial phase of the development.

Due to the dependability among the layers, the high- and low-level WCET analyses are not considered separately. This is a consequence of simulating the runtime behaviour, which entails both the control flow and the specific sequence of executed instructions. For example, to include caching, it is necessary to consider the entire sequence of instructions to determine whether a cache access results in a hit or a miss.

10.2 The Usage of TetaJ

To comply with making TetaJ usable for smaller parts of the real-time program the WCET analysis is conducted on a specific method in the analysed task. The following is therefore required as input to TetaJ:

- The source code of the analysed Java program, such that annotations can be extracted.
- The compiled Java program to construct the JBC model.
- The signature of the method to be analysed.
- A JVM model representing the JVM used in the execution environment.
- A hardware model representing the behaviour of the hardware.

The models in the three layers should be independent such that experiments can be made to determine the resulting WCET on different combinations of JVMs and hardware. We envision, this can be used to determine the best execution environment, in terms of e.g. execution time and price, for the specific program. Furthermore, this makes it possible for the JVM and hardware vendors to provide their models regardless of the given application using them. Also, if the hardware vendor does not want to publish the model of the hardware, it is possible for the application developers to send their model to the hardware vendor, who then is responsible for conducting the WCET analysis. Finally, having independent models, allows JVM developers to insert template stubs as the JBC model layer and as the hardware model layer, and thereby solely concentrate on their implementation of the JVM. This way, even though not verified for a specific real-time system, effects to the WCET after changing the JVM implementation can be found.

Another potential usage of TetaJ is profiling of tasks, that is, identifying parts of the code that are especially contributing to high WCET. This can help the developers to find potential bottlenecks.
10.3 The Toolchain

We have chosen to structure TetaJ as a toolchain consisting of three specialised tools. By making a toolchain, we envision that the individual tools, or parts of the toolchain, can be reused in other contexts. E.g. the generated UPPAAL models for the Java task could be used in other tools that examine functional correctness or schedulability analysis.

The three tools are: the Model Generator, the Model Combiner, and the Model Processor. Figure 10.2 shows how these tools form the toolchain.

The Model Generator is capable of transforming JBC into a CFG, analysing it, and finally converting it into a UPPAAL model. The Model Combiner is subsequently responsible for combining this with the models of the JVM and hardware into one UPPAAL model. Meanwhile, the models are processed for optimisation purposes. The final tool, the Model Processor, applies the UPPAAL model checker on the combined models with an appropriate query and formats the WCET to the user.
This chapter describes the TetaJ CFG representation (TCFG) and the transformation of JBC to this. The intermediate representation will make the basis for constructing the JBC model. For a more detailed description of the design, see Appendix B.

### 11.1 TetaJ CFG Representation

The control flow of the analysed Java method is captured in TCFG which can be traversed and manipulated by the *Model Generator* tool. Hence using TCFG is beneficial, since it forms an interface that can be used by the several components responsible for processing the CFG. The primary incentive is reusability and potentially the model generation can be used for arbitrary programs as long as TCFG exists for it. The idea of using TCFG is therefore similar to that of aiT's CRL, as described in Chapter 8. It may seem out of scope of TetaJ to introduce TCFG, however as will become apparent in Chapter 16, we have ourselves benefited from it by transforming another programming language to TCFG. Furthermore, TCFG is designed to provide functionality making the manipulation of the control flow easier. For example, it is possible to lookup source code lines for individual instructions.

TCFGs are created individually for each method in the source program, which makes it easier to distinguish them and conveniently maps onto the resulting UPPAAL models for providing a better overview of the models. Furthermore, each TCFG must contain all the details necessary for conducting the WCET analysis.

For developing the intermediate representation, we elicited four requirements:

- The TCFG must allow for extracting the individual instructions contained in its basic blocks.
- It must be possible to annotate the TCFG with information used for the CFG analyses.
- It must be possible to map the individual instructions to their placement in the source code.
- It must be possible to extract the fully qualified name of all invoked methods from the resulting TCFG to follow the execution across methods.
11.2 Java Bytecode to TCFG

In order to construct the TCFG, the control flow of the JBC must be captured. This can be done by constructing a CFG for a method using a number of existing solutions. As an alternative, designing and implementing a new solution is a complex task requiring a relative large effort (SOOT, 2010; FindBugs, 2009; Schoeberl, 2009; Volta, 2009; Bøgholm et al., 2008b). We assess that doing this ourselves will not add any significant advantages in contrast to reusing one of the existing solutions. Furthermore, we assess that the time needed for becoming acquainted with existing libraries, will not exceed the time required for making our own implementation. Thus, we will reuse an existing solution. Regardless of the choice of library, the resulting CFG must subsequently be transformed into TCFG outlined previously and thus must provide the necessary information.

In our selection of library, we focused on the provision of high-level functionality for CFG construction while still providing the information necessary in the later usage of the CFG. To determine which solution would suffice, we chose to conduct a number of experiments with the candidate libraries. These experiments helped us in understanding the solutions and helped in our assessment whether their usage is in accordance with the requirements for TCFG. Furthermore, the exploration of the libraries helped us making the requirements for TCFG.

In the following, four different solutions are considered. These were chosen because we observed their usage as part of existing WCET tools or because they are used as part of larger known analysis tools for Java. Many of them build upon the Byte Code Engineering Library (BCEL) which is a Java library used for accessing and manipulating Java class files. BCEL would have been a fundamental part if we had chosen to implement the CFG construction ourselves.

SOOT SOOT (SOOT, 2010) is a framework for conducting Java optimisations by representing Java programs in a number of formats and facilitating analyses of these. One of these steps consists of generating CFGs of the methods comprising the program. During the experimental analysis, we revealed that SOOT was overly complicated to use. In particular, SOOT immediately translates the JBC into an internal representation such as 3-address code or static single assignment form which are more amenable for applying optimisations. To our knowledge, it is not possible to reconstruct the JBC after the initial conversion to the internal representations, which is a problem, since we need to extract this information later when constructing the models. Furthermore, SOOT changes the control flow, which is a problem since TetaJ relies on a representation that has not been manipulated (Bøgholm et al., 2008b).

WCA WCA is a tool for analysing WCET for tasks executed on the JOP. In this process, the tool generates CFGs that are used internally. Examining the source of WCA revealed that the CFG construction is made as an independent component, hence making it amenable for reusability. In the WCA CFG representation both the bytecode instructions and the line numbers are represented.
CHAPTER 11. CFG GENERATION

Through experiments, we verified that WCA fulfils our requirements. However, a disadvantage is that the library is sparsely documented. This means that effort must be put in examining the API through the source code before using the library.

Volta Volta (Volta, 2009) is a suite of Java real-time development tools. One of the tools in the suite is a CFG analyser, called Cascade. Its main purpose is to generate CFGs for all methods of a Java class file, which then can be given as input for the other tools in the suite.

From the documentation, it can be deduced that Cascade is capable of fulfilling all our requirements. However, through experiments, it became apparent that this is not the case. Particularly, problems arise if class constructors are used in the methods of interest, since JBC of constructors cannot be looked up due to an error in their reflection utilities. Thus, this can not be used if objects are allowed to be allocated in the analysed code. The problem has been confirmed by communication with a Volta developer (Harmon, 2011).

WALA T. J. Watson Libraries for Analysis (WALA) (WALA, 2011) is an extensive library for static analysis of JBC providing a number of utilities and data structures. One of these is support for CFG construction.

WALA turned out to be complex while providing a huge amount of functionality. This is not necessarily a disadvantage if the needed functionality is supported. However, due to its complexity, we were not even able to use the library, since a long list of dependencies needed to be handled first. Since we assessed the time required for handling this issue to be more than the time for finding another solution, we decided not to proceed with the experiments.

Many of the libraries can be used for CFG construction. However, it is only WCA that fully supports our needs. Volta would have been an alternative, but we assessed that it would require too much effort to correct their reflection utility. If class constructors are not used this is of course not a problem, or at least a problem that can be circumvented. However, taking into account the various Java real-time profiles this should be supported even in the time critical phases since they contain the notion of timing predictable memory allocation. Therefore, since WCA supports constructors, it seem as a better choice to use that.
Preparing the Model Checking

Before conducting the WCET analysis, the models must be prepared. Therefore, a number of CFG analyses are conducted on the TCFGs before they are transformed into UPPAAL models. This encompasses determining loop bounds based on annotations, optimising the TCFGs, adding UPPAAL progress measures, and removing unnecessary models from the UPPAAL NTA system. These analyses are described in further detail throughout this chapter. Appendix C and E provide a more detailed description of the design and implementation of these.

12.1 Loop Bound Analysis

The purpose of the loop bound analysis, is to find and annotate all loops in the TCFGs to support the subsequent transformation of these into models. Specifically, a number of properties must be applied to specific edges in the models to correctly track the iteration count of loops and enforce loop bounds.

12.1.1 Annotating Loops

Loops are to be bounded using annotations in the source-code. This means that a unique identifier is required to distinguish annotations from ordinary comments. Similar to other WCET tools (Schoeberl et al., 2010; Hu et al., 2002b), we use //@ to identify annotations. This allows annotations to be extracted for a number of different languages including Java, Scala, etc. Since JBC-based languages can vary greatly in their syntax, care has been taken in the annotation extraction facilities such that they can easily be extended to support other languages and other annotation formats.

When annotating the source code, the annotation must be placed on the line immediately preceding the code of interest. Another approach would be to place the annotation on the same line as that of the source code that must be annotated, but it is our experience from using WCA that this degrades readability. Listing 12.1 shows an example of a loop bounded to 10 iterations.
12.1.2 The Structure of a Loop

The goal of the analysis is to support annotation of loops for a number of different loop constructs. Specifically, four different loop constructs are supported: for-, while-, foreach-, and do-while-loops. The structure of these are similar, with the exception of do-while-loops which reverses the order in which it executes its body and checks its loop condition.

To further accommodate the specific needs for Java developers, we do not assume a specific compiler is used, but instead, we want to support multiple compilers, specifically: the javac compiler provided by Sun’s JDK 1.6, the ECJ compiler part of the Eclipse IDE, Jikes from IBM (IBM, 2005), and GCJ from GCC.

The different structures observed for the loop constructs are shown in Figure 12.1. Specifically, Figure 12.1(a) shows the structure of the for-, while-, and foreach-loop when using the ECJ, Jikes and GCJ compilers, while Figure 12.1(b) shows the structure when using the javac compiler. The difference between the output from these lies in the sequence of the loop condition and loop body basic blocks. Finally, the structure of do-while-loops, shown in Figure 12.1(c), is identical across all of the analysed compilers.

Common for the loops are that they are reducible. A reducible loop contains a single loop header, that is, a single basic block which is always visited when the loop is executed. Furthermore, it has one or more back edges returning control from the loop itself to the loop header, that is, edges which point to ancestor basic blocks. To ease the understanding of the following we introduce two new edge types: loop entry edges and loop iteration edges. A loop entry edge is an edge which enters the loop header, and thus enters the loop construct. A loop iteration edge represents the edge taken on each iteration of the loop. All of these are depicted in Figure 12.1.

12.1.3 Detecting Loops

The general problem of identifying loops given a CFG has been extensively researched and various algorithmic solutions have been proposed (Sreedhar et al., 1995; Tarjan, 1973; Aho et al., 2006). For TetaJ, we chose to use the loop detection algorithm presented by Aho et al. (2006). This algorithm will not only detect the presence of a loop, but also determine all the basic blocks contained in the loop body as well as determine which basic block constitute the loop header. Furthermore, it also supports nesting of loops. The algorithm can be divided into three steps.

Listing 12.1: The annotation style used in TetaJ.

```java
//@loopbound=10
for (int i = 0; i < 10; i++) {
    // loop body
}
```
12.1. LOOP BOUND ANALYSIS

1. Conduct a depth-first ordering on the CFG, annotating it accordingly.

2. Find all back edges using the newly determined order. Each of these are part of a loop.

3. Determine the basic blocks contained in the loop body.

Each of these steps are described in the following.

Detecting Back Edges

The first step is to determine a depth-first ordering of the basic blocks in the CFG. In such a traversal, each node is visited, starting from the root node, following the rightmost and then the leftmost child. During this process, each node is assigned a value accordingly, denoted the Depth-First Number (DFN), when both of its children have been visited. An example of a CFG, in which each basic block is given a DFN, is shown in Figure 12.2.
The pseudocode of the depth-first ordering algorithm is shown in Listing 12.2.

```plaintext
depthFirstOrdering(G) {
    DFN[] = \emptyset
    c = number of nodes in G
    for each node n of G {
        mark n "unvisited"
    }
    search(n_0)
    return DFN
}

search(n) {
    mark n "visited"
    for each successor s of n {
        if(s is marked "unvisited") {
            mark s "visited"
            search(s)
        }
    }
    DFN[n] = c
    c = c - 1
}
```

**Pseudo-code 12.2:** Pseudocode of the algorithm used to make the depth-first ordering. The algorithm is inspired by Aho et al. (2006).

Using the newly found depth-first order, it is now trivial to detect the back edges. Specifically, since the DFN value increases as the graph is traversed, a back edge can be found as an edge \( a \rightarrow b \) for which depth-first-number[\( a \)] \( \geq \) depth-first-number[\( b \)].
12.1. LOOP BOUND ANALYSIS

Determining the Basic Blocks of Loops

The remaining problem is now how to construct the set of basic blocks comprising the particular loop given a CFG, $G$, and a back edge, $a \rightarrow b$. This can be done using a reverse depth-first search of the CFG using $a$ as the starting point and $b$ marked as \textit{visited} in order to stop the traversal beyond that point. The pseudocode for this algorithm is shown in Listing 12.3.

```
1 findLoopBasicBlocks(a, b, G) {
    Loop = \{a, b\}
    for each node n of G{
        mark n "unvisited"
    }
    mark b "visited"
    reverseSearch(a, Loop)
    return Loop
}
2 reverseSearch(a, Loop) {
    mark a "visited"
    for each ancestor s of a {
        if (s is marked "unvisited") {
            mark s "visited"
            add s to Loop
            reverseSearch(s, Loop)
        }
    }
}
```

\textbf{Pseudo-code 12.3:} Pseudocode of the algorithm used to make the reverse depth-first search and finds the basic blocks contained in the loop body with back edge $a \rightarrow b$ and CFG $G$. The algorithm is inspired by Aho et al. (2006).

After conducting this algorithm, the set $\text{Loop}$ contains all the basic blocks that reach $a$ from $b$. Using the CFG presented in Figure 12.2, the $\text{Loop}$ set will be $\text{Loop} = \{3, 4, 5, 6\}$.

12.1.4 Annotating the TCFG

Having found the basic blocks of the loop, the task that remains to be done is to extract the necessary information which is needed when generating the UPPAAL models. For this purpose, a counter is used to represent the loop count, and annotations are used on specific edges to either indicate that the counter must be updated or to indicate a loop bound. Intuitively, this means that two types of edges must be identified for a given loop: the \textit{loop entry edges} and the \textit{loop iteration edge}. 
The loop entry edges reset the loop iteration counter to compensate for previous executions of the loop. The edges can be determined as all incoming edges, except back edges, to the loop header. The loop iteration edge increments the iteration counter and ensures that the loop is bounded. This can be determined by finding a basic block in the loop with an edge pointing into the loop and an edge pointing out of the loop. This basic block must contain some condition selecting either to proceed into the loop or exit it. However, multiple instances of this type of basic block can exist in a loop, e.g. if the loop body contains an if-statement with a break-statement, resulting in a similar structure as the loop condition. An example of this is shown in Figure 12.3.

The correct basic block is identified by attempting to extract a loop bound annotation from each of the basic blocks contained in the loop body. Only one of these has an annotation, that is, the basic block representing the actual loop condition and this loop bound annotation is used in the model. This approach does pose a risk, since an if-statement with a loop bound inside a loop could be mistaken for the actual loop condition. Regardless, we assume that correct annotations have been applied by the programmer.

![Figure 12.3: Loop structure with an if-statement in the loop body containing a break-statement.](image)

### 12.2 Condition Optimisation

In model checking, the verification time is highly dependent on the size of the models’ state space. In this, branches are especially interesting because they exponentially increase the state space. Thus, a condition optimisation analysis is conducted to identify removable branches in conditions, such as if-statements, and thereby reduce the state space of the models. This is partially inspired from a similar optimisation conducted by Dalsgaard et al. (2009).
12.2. CONDITION OPTIMISATION

Typically, it is difficult to determine which branch results in the highest WCET, since it requires a WCET analysis to be conducted on each branch beforehand. However, in the case of an *if*-statement in isolation, that is, e.g. no *else*-statement, it always yield higher WCET to follow the edge leading to its body, than jumping past the body, assuming no timing anomalies are present. To illustrate the concept of this optimisation, consider Figure 12.4. In this, removing the dotted edge, will ensure that the body of the *if*-statement is always visited. Thus, the goal of this analysis is to detect these and remove them from the TCFGs.

![Figure 12.4: Illustration of removing an edge that is known not to be on the path resulting in the WCET. The dotted edge is removed by the optimisation.](image)

The removable edges are detected using a simple algorithm. First, the basic blocks are ordered using depth-first ordering. Then all occurrences of basic blocks with two outgoing edges are found, thereby all condition blocks are identified. One, denoted the *condition body edge*, points to the body of the *if*-statement, and the other, denoted the *candidate edge* points to the basic block right after the body. A path should exist through the condition body edge which reaches the candidate edge’s target basic block. If this is the case, the candidate edge can safely be removed.

The pseudo code for this algorithm is shown in Listing 12.4. Note that the candidate and condition body edges are detected by observing their target basic blocks’ DFN value. If the basic block reached by the candidate edge is visited later in the execution when following the condition body edge, the candidate edge is safely removed.
Pseudo-code 12.4: Pseudo code of the algorithm used to detect and remove if-condition edges from a TCFG $G$. 

```plaintext
condition-optimisation(G) {
    DFN[] = depth-first-ordering(G)

    for each basicblock bb of G {
        if (count outgoingEdges of bb == 2) {
            if (DFN[left edge of bb] > DFN[right edge of bb]) {
                candidateEdge = left edge of bb
                conditionBodyEdge = right edge of bb
            } else {
                candidateEdge = right edge of bb
                conditionBodyEdge = left edge of bb
            }

            for each basicblock markBb of G {
                mark markBb "unvisited";
            }

            if (edgeReaches(conditionBodyEdge, target basicblock of candidateEdge)) {
                remove candidateEdge
            }
        }
    }

    edgeReaches(candidateEdge, target) {
        candidate = target basicblock of candidateEdge

        if (candidate is marked "unvisited") {
            mark candidate "visited"

            if (candidate == target) {
                return True
            }

            for each edge edge of candidate {
                if (edgeReaches(edge, target)) {
                    return True
                }
            }
        }

        return False
    }
}
```
To reduce the memory consumption in UPPAAL progress measures are used. Since these should be incremented to indicate progress in the model, it is relatively simple to include as firing an edge represents progress. However, incrementing the progress measure for each edge potentially results in very large numbers potentially degrading performance (Bengtsson et al., 1996). Thus, to reduce its size it is only incremented for each basic block in the models, thereby reducing the incrementation considerably compared to incrementing it for each instruction. This is, however, at the cost of increased memory overhead, but it is nonetheless a good balance between performance and memory consumption.

Introducing progress measures for loop constructs requires some additional considerations. Specifically, the progress measure must be handled with care for the loop exit edge, since the value of the progress measure at this point depends on the size of the loop body and on the iteration count of the loop. In effect, the progress measure should be incremented at the loop exit such that the progress measure value at this point is always larger than the progress measure in the loop condition. An analysis is thus conducted on all loops in order to annotate the TCFG with the correct progress measure incrementation (Huber, 2011).

Figure 12.5: Illustration of progress measures in loops.

Figure 12.5 depicts an example of a loop with progress measures. The loop has a loop bound of 10, and increments the progress measure with 3 for each iteration because there are three basic block constituting the loop. Thus, the progress measure at the loop exit is further
incremented with $3 \cdot 10^3 - 3 \cdot \text{loop\_count}$, where \text{loop\_count} is the current iteration count, ensuring that the progress measure is in fact increased.

To annotate the TCFG with the required information, it is necessary to know the longest path in the loop body to find the maximum increase of the progress measure, that is, the number of basic blocks in the longest path. This is possible, but requires an NP-hard search through the loop.

### 12.4 Model Cleaner Optimisation

Recall, the JVM model was provided independently to TetaJ and of the analysed task. This is because this model should only be changed in case the JVM implementation is altered. However, not taking into account the analysed task is problematic because larger parts of the JVM model may never be needed. For instance, a small task that only increments a variable and then terminates only needs a few JBC instructions for being implemented hence leaving over 200 JBC instructions unnecessary. The problem is that the implementations of the JBC instructions are placed in individual templates in the JVM model. Unfortunately, these models add a significant overhead to the model checking process even when they are never synchronised, since they are instantiated. Hence, removing them significantly reduces the overhead in terms of the size of the state space. The model cleaner optimisation is responsible for doing this by determining unnecessary JBC instructions and remove their corresponding models from the combined model. Note that the original JVM model is preserved to support that the task during the development is altered thereby possibly requiring another set of JBC instructions.

The optimisation is conducted in two steps. First, the model of the analysed task is analysed to build a list of all used JBC instructions. Subsequently, the UPPAAL XML model file is consulted to remove all the unnecessary models containing implementations of JBC instructions.
In the following, the transformations from TCFGs to UPPAAL models are shown. Specifically, it is shown how basic blocks, program flow, loops, and method calls from TCFG are modelled in UPPAAL. All the models presented in this chapter are part of the JBC model layer forming the uppermost layer in Figure 10.1. Later, in Chapter 14, it will be described how these models interact with the underlying JVM model layer forming the middle layer in Figure 10.1. A more detailed description of the design of the implementation of generating UPPAAL models can be found in Appendix D.

### 13.1 Modelling a Basic Block

Modelling a basic block concerns modelling its instructions in the correct order. Simulating the execution of the instructions is done by synchronising with their respective implementation in the JVM model layer in the same order as they appear in the basic block in TCFG. As shown in Figure 13.1, this can be modelled using an urgent channel meaning that whenever the JVM model layer synchronises with the model of the program, the transition will be fired immediately. In the figure, this synchronisation channel is called \( jvm\_execute \) and prior to performing the synchronisation, the variable \( JVM\_instruction \) is set to reflect the particular instruction set for execution. Later in Section 14.2 it will become apparent how the synchronisation and this variable is used in the JVM model layer.

![Figure 13.1: UPPAAL model of a basic block with four instructions.](image)

Also, as shown in the figure, the model of a basic block is encapsulated by the `BasicBlockBegin_ID0` and `BasicBlockEnd_ID0` locations denoting the start and end of the basic block with index 0. The reason for only making the final of the two encapsulating locations committed is that the edge from the first basic block cannot be fired before the JVM model is ready to synchronise with it. Also, the location corresponding to executing the final instruction is committed such that the end of the basic block is reached in one step without taking additional time.
Connecting the basic blocks is a matter of directly transforming the edges between basic blocks in TCFG to edges in the model. That is, if a basic block branches to two other basic blocks, this must be reflected in the model. Figure 13.2 depicts connecting three basic blocks where the first branches to the others.

**Figure 13.2:** UPPAAL model of three connected basic block.

### 13.2 Modelling Loops

To bound loops, loop bounds are inserted into the model as guards on the loop iteration edge. This ensures that the loop body is at most iterated the number of times specified by the loop bound. Each time the loop iteration edge is taken, a variable is incremented to constrain the number of iterations of the loop body. Before entering the loop, the counter is reset to ensure that it can be iterated again should it be encountered later. Figure 13.3 depicts a loop with a loop bound of 3. The loop condition block corresponds to the basic block with index 2, and the loop body is represented by the basic block with index 1.

### 13.3 Method Call

To simulate method invocation, a synchronisation to the model of the invoked method is inserted. Furthermore, each model of a method implementation has a synchronisation on the edge leading from the initial location to the actual implementation of the method and one on
13.3. METHOD CALL

Figure 13.3: UPPAAL model excerpt depicting a loop structure. Basic block 2 is the loop condition, and basic block 1 is the loop body.

The edge returning to the initial location. These are used to model method invocation and return for the caller. An example of modelling method calls is depicted in Figure 13.4.

In the example shown in the figure, a method, called methodA shown in Figure 13.4(a), calls another method, methodB, shown in Figure 13.4(b). This is accomplished by letting methodA issue a synchronisation on the invoke_methodB channel. methodB likewise synchronises and start execution after the BasicBlockBegin_ID0 location. When it is done executing, methodB signals to its call site, methodA, by synchronising on the return_methodB channel. After this, methodA will continue executing from the point after the call was made.

13.3.1 Native Method Call

JBC allows for a special class of method calls, namely native method calls. These are special since it is the responsibility of the JVM to execute them. That is, in contrast to ordinary method calls, the synchronisation must be made to the corresponding models in the JVM model layer since the models of the implementations of the native methods are residing there. To model this, a special variable, NATIVE_CALL, is updated to hold the value of the called native method. It is then the responsibility of the JVM model to check this variable to synchronise with the correct native method. Figure 13.5 depicts an example of how the special variable is updated when a native call is made and how we envision JVM model to handle it.
CHAPTER 13. UPPAAL MODEL GENERATION

(a) Method A invoking method B.

(b) Method B.

Figure 13.4: UPPAAL models depicting how method calls are modelled through synchronisations.

Figure 13.5: UPPAAL models depicting how native method calls are modelled through synchronisations and guards.
Connecting the Model Layers for WCET Analysis

Being able to conduct WCET analysis requires the individual model layers to be connected into a single model. The purpose of this chapter is to describe how the different model layers are connected. Furthermore, it will be described how the resulting model is used for WCET estimation.

The tool responsible for combining these models into one is described in further detail in Appendix E, and the tool responsible for conducting the actual WCET estimation on the combined model is described in Appendix F.

### 14.1 Initialisation Model

To initiate the WCET analysis of the method of interest, an initialisation model is used which contains four locations, namely: *Idle*, *Execute*, *Main_Done*, and *Terminate* and is depicted in Figure 14.1.

![Figure 14.1: The UPPAAL initialisation model responsible for initiating the WCET analysis.](image)

The first synchronisation starts simulating the execution of the method for which the WCET is desired. When this has completed, the initialisation model reaches the *Main_Done* location, from which it synchronises with the METAMOC hardware model to indicate that the processing has completed. This makes the hardware model reach a deadlock which ensures that the hardware model has finished executing the remaining instructions.
CHAPTER 14. CONNECTING THE MODEL LAYERS FOR WCET ANALYSIS

14.2 JVM Model

The general behaviour of the JVM has been captured in a model representing the interpretation loop. This is modelled with an initial state from which the task can signal execution of the next JBC instruction as described in Section 13.1. Simulating the execution of the JBC instruction can be placed in individual models that are synchronised in a similar way as how method calls are dealt with. When the processing has completed, the process of the JVM model returns to its initial location allowing executing the next JBC instruction. Figure 14.2 depicts an excerpt of a simple JVM model supporting the execution of iload and istore.

![Figure 14.2: Illustration of a simple JVM modelled in UPPAAL.](image)

The fetch, analysis, and execution phases of interpreting JBC instructions are captured in the model. The fetch phase corresponds to firing the edge between the Idle and Analyse_JBC locations. After this, the analysis phase is conducted by determining which JBC instruction was fetched. Finally, the given implementation of the JBC instruction is simulated by synchronising with its implementation in the JVM model layer.

14.3 Hardware Model

The interaction between the JVM model layer and the hardware model layer is similar to the interaction between the JBC model layer and the JVM model layer. Consider the example where the program executes the istore JBC instruction. Suppose its implementation is realised by issuing two architecture-dependent instructions, namely instr_push and instr_store. These are processed in the model of the hardware by synchronising with the pipeline by issuing a synchronisation on the fetch channel. Whenever this synchronisation is issued, the identifier representing the particular instruction is provided. This information is used in the model of the hardware to account for the pipelining process.
In Figure 14.3, the model used in METAMOC of a simple pipeline is shown. The pipeline represents the outermost interface that the JVM model interacts with. Whenever, a synchronisation is made on the fetch channel, as done by the model in Figure 14.3(c), it will initiate the model shown in Figure 14.3(a), by placing the instruction in the pipeline. The invariant \( x \leq 1 \) together with the guard \( x = 1 \) represents, that for this particular pipeline, adding an instruction to it takes one clock cycle. Whenever adding an instruction, it initiates the execution stage of the pipeline afterwards. To model this, the fetch stage synchronises with the execution stage by synchronising on the execute channel as shown in Figure 14.3(b). In the execution stage, the function \texttt{set\_wait()}\footnote{Listing 14.1} is invoked which determines the execution time of the particular instruction. An excerpt of this function is shown in Listing 14.1.

```c
void set_wait()
{
    if (pipeline[EXECUTION\_STAGE] == ASSEMBLY\_ADD) {
        wait = CYCLES\_ADD;
        return;
    } else if (pipeline[EXECUTION\_STAGE] == ASSEMBLY\_ADC) {
        wait = CYCLES\_ADC;
        return;
    } else if (pipeline[EXECUTION\_STAGE] == ASSEMBLY\_ADIW) {
```
As shown in the listing, the `set_wait()` function is implemented by a comparison between the instruction placed in the execution stage of the pipeline and the instruction set of the given processor. `set_wait()` subsequently sets the `wait` variable according to how long the process should wait before the execution stage can proceed.

### 14.4 Estimating the WCET

Using a global clock, denoted `clockcycles`, the maximum time used for executing the method can be determined by the following sup query:

```
sup:clockcycles
```

**Listing 14.2:** Sup query used for determining the maximum value of the clock variable `clockcycles`.

We assess this approach of determining the WCET using UPPAAL as significantly better than using the binary search approach to tighten the WCET, since the sup query only requires one check. Though it should be remarked that this requires the developers to use the development version of UPPAAL instead of the official release.
Part IV

Case Study
The chapters in this part describe a case study of using TetaJ on a real-time Java application. The purpose of this is twofold; first it is used to illustrate an example of a timing predictable Java real-time system running on common hardware for embedded systems and secondly it is used to evaluate the applicability of TetaJ. The case study used in this evaluation is a classic text-book example of a mine pump which is an embedded hard real-real time system (Burns and Wellings, 2009; Liu and Joseph, 2005). The system was originally designed and implemented as part of our preliminary research (Bøgholm et al., 2011) but has undergone adjustments to accommodate the specific execution environment used in this case study. This chapter will recapitulate the essentials of the mine pump, and outline the execution environment on which the mine pump system operates.

15.1 The Mine Pump Real-Time System

In short, the mine pump is operating inside a mine which is continuously filled with water and a system is tasked with controlling a water pump such that the mine is never flooded. A number of environmental parameters are further monitored to determine whether the water pump can safely be operated or that continuing operation may endanger the mine workers in which case different alarms must be raised.

15.1.1 The Constrained Mine Pump Example

Since the purpose of the case study is to illustrate an embedded real-time system and since the original description of the mine pump is quite elaborate, we have chosen to make a constrained example. This emphasises on essential aspects of the mine pump mainly to reduce the implementation time while still providing a good example of a real-time system for which the applicability of TetaJ can be evaluated. Figure 15.1 illustrates the constrained system.

The constrained mine pump example focuses on the safety control logic of the water pump, such that the water pump is only capable of operating under certain environmental conditions.
15.1. THE MINE PUMP REAL-TIME SYSTEM

Figure 15.1: Example of the mine pump consisting of the actual water pump and various sensors.

These are monitored using three different sensors; two water level sensors and one methane level sensor.

Three functional requirements can be elicited for the water pump:

1. The water pump must never operate if the amount of methane exceeds a critical level.
2. When the height of the water exceeds its high level, the water pump must be started.
3. When the height of the water falls below its low level, the water pump must be stopped.

From these, it can be deduced that the mine is never flooded as long as the methane level does not exceed its critical level, and the mine is never completely emptied for water since the water pump is stopped at the low water level. Besides the functional requirements, a set of timing requirements also exists:

1. Starting or stopping the water pump according to a change in water level must at most take 200ms.
2. Stopping the water pump when the level of methane reaches its critical level must at most take 200ms.

15.1.2 Tasks and their Timing Requirements

The mine pump can be realised using a number of different combinations of periodic and sporadic tasks. In our preliminary research, we implemented the example using both sporadic and periodic tasks (SW9-Project, 2010). However, the sporadic tasks are not in fact needed and were only included to examine how periodic and sporadic tasks can interact. Therefore,
a much simpler solution is to solely use periodic tasks without any loss of functionality. The
temporal requirements of the periodic tasks are listed in Table 15.1.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period/Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methane Level Detection</td>
<td>56ms</td>
</tr>
<tr>
<td>Water Level Detection</td>
<td>40ms</td>
</tr>
</tbody>
</table>

Table 15.1: Timing requirements for the periodic tasks of the constrained mine pump example.

The deadlines are derived such that the system can successfully respond to the event of critical methane level and changes in water levels within 200ms taking into account the physical implementation of the mine pump and capabilities of the sensors. Note that deadlines and periods are not distinguished in the case study. (SW9-Project, 2010)

15.1.3 The Physical Construction

We have constructed a physical model of the mine pump in LEGO to simulate the real-time system on a miniature model. The LEGO construction is depicted in Figure 15.2.

The LEGO mine pump is constructed to reflect a closed-loop control, where the control logic issues the methane and water level sensors to provide input to the environment, thereby changing its state. The system initiates by filling the feeder, labelled 1, with bricks representing water and methane. From here, the bricks are continuously fed into the mine shaft, labelled 6. Before entering the mine shaft, the bricks pass by the methane sensor, labelled 8, to keep track of the concentration of methane in the mine shaft.

The mine shaft is equipped with two water level sensors, namely the high water level sensor labelled 4 and the low water level sensor labelled 5. These are used to determine if the mine shaft is filled with water or not. If the shaft is filled with water and a critical amount of methane is not present, the mine pump, labelled 7, is started. This empties the the mine shaft, and places the bricks on the conveyor belt, labelled 2, from where the bricks are transported back to the feeder.

15.2 Choice of Execution Environment

The mine pump is supposed to be hosted on common embedded hardware to reflect its natural execution environment closely. Many processors exist for this purpose of which some of the more prominent include processors of the ARM and AVR processor families. The architecture of the AVR processors is generally simpler than that of many of the ARM processors due to the absence of caching among others and besides, we are in possession of an AVR ATmega2560 processor running on an AVR STK600 evaluation board. Hence, we have chosen to initially experiment with this simpler processor to determine the applicability of TetaJ. Furthermore,
15.2. CHOICE OF EXECUTION ENVIRONMENT

**Figure 15.2**: 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) ATmega2560 processor. 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.

It should be remarked that AVR with a market share of over 30 percent was the largest selling 8-bit microcontroller in 2004 and at the time had shipped a total of 500 million units (Atmel, 2004). The processor is therefore a good example of illustrating a real-time Java program running on common embedded hardware.

Evidently, a JVM must be chosen that supports the AVR architecture, is interpretation-based, and avoids problematic functionality such as dynamic class loading. To our knowledge, the only software implemented JVMs that provides real-time facilities, for resource constrained devices, are the Fiji VM (Pizlo et al., 2009) and JamaicaVM (Siebert, 2011). However, although Fiji VM has been identified to feature a interpreter in the future, it currently uses AOT compilation (Pizlo et al., 2009), and is thereby not applicable for TetaJ. JamaicaVM, however, supports both interpretation and AOT compilation but is closed source, meaning that we would not be able to modify it.
Of interpretation-based JVMs that are open source, we are aware of JamVM (Lougher, 2010), NanoVM (Harbaum, 2006), and HVM (Korsholm, 2011b). However, only NanoVM and HVM supports the AVR architecture. From these, we chose the HVM since we are personally acquainted with its author, Stephan Korsholm, from Via University College, who as of this writing is associated with Aalborg University. Since we quickly realised that modifying the JVM would inevitably be the case, personally knowing the JVM author was deemed a great benefit.
Hardware near Virtual Machine

The material available about the HVM is relatively sparse, so the following is partly based on examination of the source code of the HVM, Korsholm (2011b), and personal communication with the creator of the HVM (Korsholm, 2011a).

This chapter will describe the properties and concepts of the HVM, describe how it was changed in order to better support WCET analysis, and finally it will be described how the HVM is modelled.

16.1 General Properties of the HVM

The HVM is a simple implementation of the JVM, written in C, specifically targeted embedded systems. The simplicity of the HVM limits the amount of features which are often included in implementations of the JVM for desktop systems. Among others, this means that there is no bytecode verification and no support for dynamic class loading. However, seen from a real-time perspective, these components contribute to unpredictable behaviour, so their absence is beneficial when conducting static program analysis.

The HVM supports systems with 256 kB flash memory and 8 kB of RAM, thereby making it applicable for a wide variety of embedded systems. Furthermore, it incorporates a flexible design that allows it to be ported to new target platforms with relative ease. For portability, the HVM includes a C-like interface dictating approximately 20 functions that must be implemented to reflect the given target platform. Among others, this comprise interrupt handling where Interrupt Service Routines (ISRs) must be implemented for each of the available interrupts to be handled in the Java space. Interrupts can be handled in Java space because the HVM includes the notion of so-called hardware objects and first-level interrupt handling. The hardware objects included in Java are a result of the research conducted as part of Korsholm et al. (2008) that essentially describes an object-oriented abstraction of low-level hardware devices.

The HVM supports three different hardware architectures: Atmel’s AVR (specifically the 8-bit ATmega2560), National Semiconductor’s CompactRISC (specifically the 16-bit CR16C), and x86. It is important to note that the HVM does not include the necessary real-time facilities, that is, it does not feature real-time clocks and applicable scheduling policies, among
others. Hence a real-time Java profile must be implemented or a similar programming model must be employed.

16.2 Concepts of the HVM

The HVM is an iterative interpretation-based JVM built upon a number of concepts that makes it amenable for embedded systems. It focuses on the concept of intelligent class linking meaning that it will only link those Java libraries to the application that will be used at run-time. Due to being an undecidable problem, this cannot be exact and hence the mechanism links libraries conservatively. The prime purpose is to reduce the size of the resulting program hence being an important part to enhancing the potential of using Java for resource-constrained embedded systems.

The HVM also features both SDK and OS independence, that is, both the official SDK from Oracle as well as GNU Classpath and other vendor-specific SDKs can be used. With respect to OS independence, the HVM has been specifically designed for running without an OS. Finally, the HVM also places read-only data in read-only data segments to reduce the RAM memory footprint. The interpreter itself occupies approximately 30 kB of ROM and the heap size for the running Java application is set to 4 kB. This is within the bounds of the resources available on the ATmega2560.

All JBC instructions are implemented except those handling floating-point and double types. This is not seen as a great deficiency, since many embedded platforms do not incorporate FPUs thereby requiring costly software implementations. Another important feature of the HVM is the inclusion of a garbage collector which, unfortunately, is not designed for being used in real-time systems.

A variety of tools are shipped along the HVM distribution. Most importantly is the Eclipse-plugin, called icecap-tools, which is used for compiling Java programs for the HVM. This is the point where the HVM stands out from many other JVM implementations. When a Java program is compiled for the HVM, the resulting JBC instructions are effectively incorporated into the HVM itself. Hence, essentially the HVM and the program are not shipped individually but in a bundled package. The program itself is encapsulated in arrays that are made for each method and contain the JBC instructions. The interpreter will simply consult these for fetching the next JBC instruction of the currently executing method. However, note that, even though the program and the JVM are tightly coupled, it does not affect the analyses conducted on the HVM and specifically the interpreter. This is because the interpreter is independent of the program.
16.3 Making the HVM Analysable

Despite the simplicity of the HVM, it needs to be modified to be amenable for WCET analysis. The modifications that have been applied on the HVM are mainly due to the original inclusion of e.g. recursion and other undesirable solutions.

In the following, each change to the HVM is described in further detail.

16.3.1 Unbounded Loop for Instanceof

The implementation of the `instanceof` JBC instruction originally included an unbounded loop that tests whether the object reference is an instance of a given class by consulting its parent classes iteratively. Evidently, a tight loop bound cannot be established that applies for every class hierarchy in the application, since the depth of these may be different. It is only possible to determine a fixed and safe loop bound with the size corresponding to the maximum depth of the class hierarchy. Despite being a safe bound, this would yield overly pessimistic WCET estimates. Furthermore, the analysis of the HVM will in this case be dependent on the program which is not desirable for TetaJ. Due to these reasons, we wanted to resort to another solution. We solved the problem by implementing a matrix containing information about all the relationships among the classes. Using information of the object in question, the matrix can be looked-up for determining whether it is an instance of the class with time complexity $O(1)$. It should be noted that this is at the expense of a slight memory overhead. Specifically, each element in the matrix occupies a single byte. Hence, the total size of the matrix is quadratic with complexity $\Theta(n^2)$. This may not seem significant, but taking into account the memory available on embedded hardware, this solution may be infeasible. For example, if the Java application consists of 100 classes, the matrix will occupy 10 kB of program memory. An example of the matrix is shown in Table 16.1.

<table>
<thead>
<tr>
<th>Instance of</th>
<th>Class</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 16.1: Matrix used for looking up inheritance relationships as part of the `instanceof` JBC instruction.

In this example, a value of 1 indicates that a class is an instance of another class. Therefore, class A is an instance of class B for example. The total occupation of the matrix in memory will be 9 bytes.
16.3.2 Recursion in the Interpretation Loop

The interpretation loop in the HVM is built on a single function called methodInterpreter() which is responsible for continuously fetching, analysing, and executing the JBC instructions. Some of the JBC instructions, however, make recursive calls to the methodInterpreter() function. This is e.g. the case when invoking a method from another object which generates the invokevirtual JBC instruction. The call to the methodInterpreter() in this case will execute the logic that is part of the invoked method. This introduces a number of problems. First of all, there are no ways of bounding recursion in TetaJ therefore posing similar problems to unbounded loops. Secondly, and most importantly, recursion is difficult to model using timed automata. This is because, when a UPPAAL process is in a given location in a model, it cannot synchronise with itself because it is already in a state involving that particular model. The solution to the problem is to introduce the notion of stack frames and keep track of the call stack. This means that when a method is invoked, the context of the caller is saved which comprises the program counter, the top of the stack, and the method code. Afterwards, a new stack frame is created for the invoked method and the methodInterpreter() will continue operating on this. When the method returns, the point of the caller is restored by effectively popping the stack frame. It should be noted, however, that the original method of recursively calling the methodInterpreter() is still supported since this is still used by parts in the HVM which are not analysable.

An example of how the original recursive solution was implemented is shown in Listing 16.1 which is an excerpt of the implementation of the invokevirtual instruction.

```c
if(pgm_read_pointer(&methodInfo->code, unsigned char**) == 0) {
    int32 (*nativeFunc)(const void* methodInfo, int32 *sp);
    nativeFunc = (int32(*)(const void* methodInfo, int32 *sp))(pointer) pgm_read_pointer(&methodInfo->nativeFunc, int32(*)(const void* methodInfo, int32 *sp));
    exceptr = nativeFunc(methodInfo, &sp[top]);
} else {
    exceptr = methodInterpreter(methodInfo, &sp[top]);
}
```

Listing 16.1: The original recursive structure in which the methodInterpreter() method calls itself recursively whenever the method call was not to a native method. The variable methodInfo contains information about the particular method that is about to be invoked.

The code example represents the case-statement corresponding to the implementation of the invokevirtual JBC instruction. As shown, the methodInterpreter() will be called recursively whenever the method call was not to a native method. The variable methodInfo contains information about the particular method that is about to be invoked.

In Listing 16.2 the revised version of the implementation of the invokevirtual JBC instruction is shown.
16.3. MAKING THE HVM ANALYSABLE

if (pgm_read_pointer(&methodInfo->code, unsigned char**) == 0) {
    int32 (*nativeFunc)(const void* methodInfo, int32* sp);
    nativeFunc = (int32(*)(const void* methodInfo, int32* sp)) (pointer) pgm_read_pointer
    (&methodInfo->nativeFunc, int32(*)(const void* methodInfo, int32* sp));
    nativeFunc(methodInfo, &sp[top]);
    top += pgm_read_byte(&methodInfo->numReturnValues);
} else {
    pushStackFrame(&method, &sp, &pc, &top, &method_code, methodInfo, 0);
}
...

Listing 16.2: The modified implementation of invokevirtual using stack frames.

The notable differences between the two versions is that the recursive call to methodInterpreter() has been substituted by a call to pushStackFrame() which will save the context onto the stack and the methodInterpreter() will subsequently proceed executing the invoked method in an iterative way. When execution has completed, the methodInterpreter() will simply pop the stack frame from the stack and restore its context that was saved prior to the method invocation.

16.3.3 Hardware Specific Modifications

In the implementation of the HVM, multiplication is avoided. This is because some of the processors supported by the HVM do not include instruction set support for integer multiplication. In these cases, the compiler will produce a software implementation of multiplication which is generally costly.

Many of the workarounds to avoid the use of multiplication consist of using unbounded loops. For our case, the HVM is to be used on the ATmega2560 which is one of the few processors from AVR that has instruction set support for integer multiplication. Hence, in this specific scenario the loops in the HVM can simply be removed and substituted by equivalent solutions containing multiplication.

16.3.4 Unbounded Loops for Bit Shifting

Bit-shifting operators with variables as operands contain unbounded loops since these are implemented using loops bounded by operands whose values are unknown at compile time. In this case, we have manually examined the data types of the bit-shifting operand to determine an upper loop bound. In all cases, unsigned char data types are used, hence providing the upper bound of 255 iterations for the loops.
16.3.5 Extraction of JBC Implementations

Originally, the methodInterpreter() formed the interpretation loop and was realised by an endless while-loop containing a large switch-statement containing the implementations of all JBC instructions. This has been restructured in order to allow for easier analysis of the individual implementations. Specifically, by examining the assembly of the methodInterpreter(), it was found that identifying the implementations in the individual case statements is associated with a complex undertaking. In order to ease this task, a transformation is conducted on the switch-statement, extracting each JBC instruction implementation into a method, and inserting a call to this method into the switch-statement. It should be noted, however, that this adds overhead due to the need of calling a function every time a JBC instruction is executed.

Listing 16.3 shows the isub instruction in the case statement before the transformation, and Listing 16.4 shows how this is extracted into a new method as part of the transformation. Another transformation is that all local variables are modified to take into account that a reference to these are now provided as parameters. This approach is taken since it allows the values of the passed local variables to be changed.

```
... case ISUB_OPCODE: {
    int32 res = sp[--top];
    res = sp[top - 1] - res;
    sp[top - 1] = res;
    pc++; continue;
} ...
```

Listing 16.3: Excerpt of the interpretation loop before extracting the implementation of the isub JBC instruction.

```
int32 handleISUB_OPCODE(unsigned char* code, unsigned char** method_code, unsigned short* pc, int32** sp, unsigned short* top, MethodInfo** method) {
    int32 res = (*sp)[--(*top)];
    res = (*sp)[(*top) - 1] - res;
    (*sp)[(*top) - 1] = res;
    (*pc)++; return 0;
}
```

Listing 16.4: Excerpt of the interpretation loop after extracting the implementation of the isub JBC instruction.
16.3.6 Substituting the Interpretation Loop Switch-Statement

The original switch-statement located in the methodInterpreter() is difficult to precisely analyse because the time taken to decode a JBC instruction depends on the specific JBC instruction. This is attributed by the assembly implementation of the switch-statement which does not find the corresponding case statement in constant time. To solve this issue, a new solution was introduced which comprises an array of function pointers to each of the functions handling the JBC instructions. This can easily be done since all of these functions have the same signature and hence the problem is reduced to simply insert the function pointers in the array. Furthermore, for looking up the correct function pointer, the corresponding opcode is used as index, as shown in line 8 in Listing 16.5.

```c
int32 methodInterpreter(MethodInfo* method, int32 *sp) {

    while(1) {
        code = pgm_read_byte(&method_code[pc]);
        jbc_ret = jbc_func_array[code][&code, &method_code, &pc, &sp, &top, &method];
    }
}
```

Listing 16.5: Excerpt of the revised interpretation loop.

Using this structure achieves decoding with time complexity $O(1)$ and hence the time taken is a matter of determining the time of an array access.

16.3.7 Memory Usage

The HVM features a non-real-time garbage collector, therefore using this per default imposes serious implications to the schedulability analysis since it is both impossible to determine when it will run and impossible to determine for how long. Due to the unpredictable behaviour of this, only two options seem available: either to simply disable the garbage collector and possibly produce exceptions whenever it is attempted to run, or the other solution would be to substitute the garbage collector implementation with one having real-time properties as that proposed by Bacon et al. (2003). However, despite the deterministic behaviour of a real-time garbage collector, we have chosen not to focus on this area since it would involve a significant effort to realise. Instead, we envision that this could be subject for future work. However, omitting garbage collection imposes new issues since there are no means for deallocating memory. The problem can be circumvented by introducing new memory types similar to those proposed in the real-time Java profiles such as scoped memories and immortal memory but these are considered out of scope of this project as well.

By inspecting the implementation of the HVM reveals that malloc() and free() are used for memory allocation and deallocations for the Java objects. These are known to
exhibit timing unpredictability and should therefore be avoided during time critical parts of the program (Qing Li, 2003). However, the problem can be circumvented by statically allocating an array with size corresponding to the memory needs of the application and use this for dynamic memory allocation afterwards. Another possibility is simply to prohibit the programmer from allocating Java objects during time or mission critical parts of the application. The latter approach is one of the basic ideas in the memory models used in the real-time Java profiles previously introduced in Chapter 5.

16.4 Constructing a JVM model for the HVM

In order to use TetaJ for the HVM, it is necessary to construct the corresponding JVM model. We initially wanted to use METAMOC to support constructing the JVM model. That is, provided an executable of the JVM for the target processor and details about the target processor, a model of the JVM and corresponding processor should be made by METAMOC. The advantage of reusing METAMOC is that it is flexibly designed to allow that the underlying platform and the program which is executed on it can be interchanged with minimum effort. However, through further investigation, it was discovered, that METAMOC, in its current state, only partly supports the instruction-set needed for this project. Some of the problems encountered were dependencies that should be fixed, and hardcoded file paths both within METAMOC and its dependencies. We assessed that it would require a significant effort to solve these issues.

As an alternative, we chose to construct the JVM model ourselves, and then reuse the principles, hardware models, and general interfaces dictated by METAMOC. Recall, that the input to the model generation is TCFG. Hence, the JVM model can be constructed by first generating TCFGs of the executable of the HVM, and then reuse the model generation functionality already implemented in TetaJ to build the actual model.

In the following, it is described how the HVM is transformed into TCFG.

16.4.1 Constructing TCFGs from AVR

The targeted processor in our case study is Atmel’s AVR ATmega2560. Compared to the variety of solutions for reconstructing the control flow of JBC, not much effort has been put in providing similar tools for Atmel’s AVR. In fact, only the low-level Avrora (Avrora, 2011) tool suite allows this, thereby leaving us with the choice of doing it ourselves or use the tool suite. It has been assessed that analysing assembly for AVR and making the CFGs ourselves would require an effort that would not account for its benefits. Hence, we decided to use Avrora.

Avrora is a tool suite that can be used for analysing and simulating assembly programs written for the AVR family of processors. The tools are written in Java and offer a large API for analysing the CFG of the program in question. However, a challenging task is how Avrora’s notion of a CFG can be transformed into TCFG. Compared with Java, some information is
16.4. CONSTRUCTING A JVM MODEL FOR THE HVM

more difficult to determine in AVR, such as the corresponding line number for an arbitrary instruction. In the Java class files, a line number table is provided which is not the case for the AVR executables therefore requiring more elaborate methods for deriving this. For this we chose to use the external tool *avr-addr2line* provided by the GCC toolchain.

A problem however arises when using *avr-addr2line* whenever the compiler generates procedures which are not immediately apparent from the corresponding source code. This happens for instance when using arithmetic operations such as the modulo and division operators for which there are no counterparts in the AVR instruction set. Whenever these operators are used, the compiler generates procedures that implement them but during the examination process of the executable, *avr-addr2line* will report that the instructions of this procedure are part of the procedure using the operators. This is a problem if the generated procedure contains loops which need to be bounded. As it stands, such loops can be bounded using annotations just above the line conducting the arithmetic operation, which limits the support to only a single loop bound even though the implementations contain multiple loops.

16.4.2 Modelling the HVM

Generally, the HVM is modelled similarly to modelling the JBC task. However, it differs in two ways:

- The models generated from the TCFGs of the HVM are not compliant with the METAMOC hardware models. An interface model must therefore be constructed.
- Due to function pointers, the TCFG and model construction must be changed to support this.

Accounting for these differences is presented in the following.

METAMOC Interface Model

The JVM model must synchronise with the hardware models given by METAMOC. However, since we reuse the model generation functionality from TetaJ similar models for both JBC and JVM generation are produced. Therefore, to avoid treating them specially a METAMOC interface model is inserted, that inserts the AVR instruction into METAMOC’s pipeline. This interface model is depicted in Figure 16.1.

As shown, the interface model inserts the given assembly instruction into the METAMOC pipeline fetch stage after which it synchronises with the pipeline. Thus, using this approach, it is possible to use the METAMOC models with models generated from TCFG.

Handling Function Pointers

It is not possible to reconstruct the entire control flow for a method which contains function pointers, which is the case for the revised interpretation loop in the HVM, without providing
information that specify the possible call targets. The reason for this is that function pointers can potentially call all methods in the program. To retrieve the TCFGs for all potential call targets, these must be known such that their TCFGs can be obtained by explicitly requesting them. In respect to the interpretation loop, the call targets are known to be all functions implementing JBC instructions, and hence their respective TCFGs can be created. However, the control flow remains unlinked. This is done locating the function pointer and associating it with the TCFGs of the possible call targets.

To solve the problem of native calls the model of the JBC implementation handling these synchronises with the correct models of native calls based on a variable set in the JBC model. To incorporate this, a similar modification as with the interpretation loop, is made. That is, the TCFGs can explicitly be constructed for the known native methods.

Splitting the Interpretation Loop

The method containing the interpretation loop contains more logic than the actual loop. Therefore, since only the code within the loop is executed for each JBC instruction, the TCFG for the methodInterpreter should be minimised to reduce pessimism. This is done by removing all basic blocks that do not reside in the loop from the TCFG. Furthermore, inspired by Figure 14.2, the remaining TCFG is split into two TCFGs, namely a pre- and a post-processing TCFG representing the computations before and after the call to the method of the given JBC instruction. Figure 16.2 illustrates this.
This chapter describes Atmel’s AVR ATmega2560 in detail and is based on Atmel (2010). This analysis is important to understand which mechanisms are incorporated in the processor and how they might affect the WCET analysis. Furthermore, the hardware model of this processor is described.

17.1 Atmel AVR ATmega2560

A desirable property of the family of AVR microcontrollers is that they have code compatibility, meaning that the analysis conducted in this case study applies to the entire family of AVR microcontrollers with only minor adjustments such as clock speed and instruction timings.

Common to the AVR family is that they are built upon a weakly modified Harvard architecture, that is, program and data are stored in physically separate memories with respective address spaces. However, the AVR allows for reading data items that are residing in program memory by using special instructions.

As described, the AVR family is divided into various classifications where the ATmega series, of which the ATmega2560 is an instance, contains a relatively large flash memory for storing the program. The specifications of the ATmega2560 is shown in Table 17.1.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash memory</td>
<td>256kB for program memory</td>
</tr>
<tr>
<td>EEPROM</td>
<td>4kB non-volatile memory</td>
</tr>
<tr>
<td>SRAM</td>
<td>8kB for data memory</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>Up to 16MHz at 4.5V - 5.5V</td>
</tr>
</tbody>
</table>

Table 17.1: Specifications of the ATmega2560 microcontroller from Atmel.

As shown, the specification is in accordance with the requirements for the HVM, which requires approximately 30 kB of program memory and 4 kB of data memory.

An important feature of the ATmega2560, seen from the perspective of real-time systems, is the inclusion of a Real-Time Counter (RTC) with separate oscillator which prove relevant
for successfully supporting scheduling of real-time tasks. Also, the ATmega2560 includes six timers: two 8-bit timers and four 16-bit timers. These can be used for programming periodic tasks by defining ISRs that are executed whenever a particular type of timer interrupt is made.

17.2 The Instruction Set of ATmega2560

The instruction set of the AVR is classified as being 8-bit RISC with 32 general purpose registers. The total instruction set contains 130 instructions, all of which have the property of known execution times. This makes the AVR architecture deterministic and hence amenable to WCET analysis.

The deterministic behaviour of the instruction execution times is achieved due to the absence of many of the mechanisms that are predominant on architectures targeting the desktop computer domain such as the x86. These include among others caching and branch prediction. However, the AVR does include a simple two stage pipeline that separates the fetch and execution stages. Adding an instruction to the pipeline takes one clock cycle meaning that, while one instruction is being fetched from main memory, potentially another one is being executed.

The ATmega2560 requires floating point arithmetic to be implemented in software due to the absence of a FPU. This is not considered a problem since the HVM does not support floating-point arithmetic neither.

17.3 Modelling the AVR ATmega2560

Since the AVR is supported by METAMOC and TetaJ uses the UPPAAL model interfaces of METAMOC, modelling the Atmega2560 in UPPAAL is a matter of reusing the models already available. These models are identical to the ones presented in Section 14.3. To recapitulate, this models the simple two stage pipeline part of the AVR architecture. The only adjustment that needs to be performed is the specific details about the ATmega2560 processor, such as constants denoting the execution times of the individual instructions. Again, referring to the previous model shown in Figure 14.3(b), these constants need to be set as part of the set_wait() function which determines for how long the model must wait when simulating the execution of an instruction.
This chapter describes the design and implementation of the mine pump including considerations regarding the execution environment in which the mine pump is operating.

18.1 Design

The general design of the mine pump is shown in Figure 18.1. The design is divided into three UML packages representing: sensors, actuators, and the control. These resemble a layered architecture where the Control package is the uppermost layer that uses the Sensors and Actuators packages.

The layered architecture is constructed such that the control package handles the high-level logic in the application, while the sensors and actuators packages handles logic more closely related to the specific hardware. These packages are described in the following.
Control  The control package consists of two classes: the PeriodicMethaneDetection class and the PeriodicWaterLevelDetection class. These correspond to the two tasks controlling the mine pump. As a consequence of this, associations are made between these and the relevant sensors and actuators which they control.

Actuators  The actuators package encapsulates all logic related controlling the water pump. The BasicActuator class represents a simple motor while the WaterpumpActuator inherits from this and adds additional logic specific for the water pump, such as an emergency mode active when methane has been detected.

Sensors  The sensors package contains all sensors used in the mine pump. All of these inherit from the Sensor class which provide basic functionality common for all sensors. This functionality is then extended by the other classes with logic specifically related to the methane sensor, low water level sensor, and high water level sensor, respectively.

18.2 Implementation

The implementation of the mine pump is not made using any real-time Java profile since this is not supported by HVM. It would be possible to implement one of the profiles. For instance the oSCJ (Plsek et al., 2010) project provides an open implementation of the draft of the SCJ Level 0 profile and is specifically designed for being easily adaptable to new JVMs. The library features an interface containing approximately 20 native Java methods that need to be implemented by the underlying JVM. These concern mainly memory-related services and functionality used for obtaining real-time clock resolution. Hence, implementing the necessary functionality for supporting the oSCJ profile would be a significant step in writing real-time Java programs for a time predictable JVM. However, minutely examining the implementation of the oSCJ Level 0 library reveals that besides necessitating the implementation of aforementioned Java interface, it does, as of the current version, require that the underlying JVM has threading support. This is a major problem since the HVM does not have this and extending it with threading capabilities has been assessed to entail a major task in itself.

Because of our previous experience in implementing the case study in a number of real-time Java profiles, we have implicitly used one profile. This way of programming is actually suggested by Bøgholm et al. (2009). Ideally, the real-time system should after the implementation be checked by a compliance checker to verify whether the implementation conforms to the profile. Since we program in a style much similar to that of the PJ profile, we have manually checked that the implementation conforms to the style of writing predictable programs.

In the implementation we used the SDK provided by HVM, which among others makes available interrupt handling of hardware interrupts.

The following presents some of the essential implementation details for showing that the mine pump is not a trivial example.
18.2. IMPLEMENTATION

18.2.1 Mine Pump Initialisation

Listing 18.1 depicts the code used for initialising the mine pump, and thereby registering an interrupt handler on a timer interrupt.

```java
public void main(String[] args) {
    LegoAVRInterface.initialiseLego();
    LegoAVRInterface.enableInterrupts();

    WaterpumpActuator actuatorWaterpump = new WaterpumpActuator(
            ACTUATOR_ID_WATERPUMP);
    MethaneSensor methaneSensor = new MethaneSensor(SENSOR_ID_METHANE,
            CRITICAL_METHANE_LEVEL, BRICK_HISTORY_SIZE);
    HighWaterSensor highWaterSensor = new HighWaterSensor(SENSOR_ID_HIGH_WATER,
            CONSECUTIVE_HIGH_WATER_READINGS);
    LowWaterSensor lowWaterSensor = new LowWaterSensor(SENSOR_ID_LOW_WATER,
            CONSECUTIVE_NO_WATER_READINGS);

    PeriodicMethaneDetection periodicMethaneDetection = new
            PeriodicMethaneDetection(methaneSensor, actuatorWaterpump);
    PeriodicWaterLevelDetection periodicWaterLevelDetection = new
            PeriodicWaterLevelDetection(highWaterSensor, lowWaterSensor,
            actuatorWaterpump);

    ATMega2560InterruptHandler.init();

    InterruptHandler.registerHandler(periodicMethaneDetection,
            ATMega2560InterruptHandler.TIMER2_OVF);
    InterruptHandler.registerHandler(periodicWaterLevelDetection,
            ATMega2560InterruptHandler.TIMER0_OVF);

    Timer.initTimer2();
    Timer.initTimer0();

    //Loop forever. Interrupts will be handled
    for (;;) {
    }
}
```

**Listing 18.1:** The implementation of the mine pump’s `main()` method.

As shown, the first step in the initialisation is to initialise the ATmega2560 board such that it can interact with the LEGO hardware. The call to `LegoAVRInterface.initialiseLego()` executes a native method implemented by us, which among others sets up a number of timer interrupts and afterwards interrupts are enabled, since this is required for e.g. reading a sensor value.
The two tasks, periodicMethaneDetection and periodicWaterLevelDetection, are instantiated in line 8-9. These are the tasks to be released periodically, to determine whether a critical methane level is present, and to detect the current water level, respectively. Prior to those, the necessary sensors and actuators are instantiated as well and are given as argument to the respective tasks responsible for controlling them. This instantiation process can be seen as the initial step in the life cycle of an embedded system. Specifically, referring to Figure 2.2, this phase constitutes the setup phase and is hence not time critical. The primary motivation for doing this is to avoid allocations later in the execution phase where the operation of the system is time-critical.

The interrupt handler is initialised in line 13, by invoking the init() method of the ATMega2560InterruptHandler class implemented by the HVM SDK. Afterwards, interrupt handlers are registered for the TIMER2_OVF and TIMER0_OVF interrupts which are overflow interrupts driven by the 8-bit timers: timer2 and timer0, respectively. Both aforementioned tasks implement the Handler interface which dictates the implementation of a run() method which contains the logic to be executed whenever the ISR associated with the interrupt made by registerHandler() is triggered. After registering the handlers to their interrupts, the timers are initialised by initTimer2() and initTimer0(). Finally, the main() method enters an endless loop. This is because all the control logic is handled by ISRs and the mine pump should hence be kept alive doing nothing.

To control the release time of the two tasks, we control the ISRs of the TIMER2_OVF and TIMER0_OVF interrupts. These interrupts will be triggered when the clock cycle value of their respective timer overflows. This is an unsigned byte value and can hence maintain values from 0 to 255. Whenever 255 is exceeded, the overflow interrupt will be triggered thereby executing the logic part of the corresponding ISR. Knowing that the clock cycle counter of the timer will be incremented at each clock, and that the ATmega2560 runs with a clock frequency of 10MHz, the number of times the overflow interrupt is triggered before releasing the periodic task can be determined. Taking starting point in the periodic task of detecting critical amount of methane, Listing 18.2 shows the implementation of the ISR handling the TIMER2_OVF interrupt.

```c
ISR(TIMER2_OVF_vect) {
  if(has_time_for_timer_passed(++overflow_count_timer2, 56)) {
    if (DEVICES_INTERRUPTHANDLER_DISPATCH_I_V != -1) {
      isrMethodStack[0] = 16;
      methodInterpreter(&methods[DEVICES_INTERRUPTHANDLER_DISPATCH_I_V], &
                        isrMethodStack[0]);
    }
    overflow_count_timer2 = 0;
  }
}
```

Listing 18.2: The ISR responsible for handling the TIMER2_OVF.

The overflow_count variable is used to count the number of times the ISR has been executed. The function has_time_for_timer_passed() will return either true or false de-
pending on whether the value of this variable corresponds to when 56ms have passed. The
assigned value, 16, in line 4 symbolises the associated interrupt, that is, TIMER2_OVF and
will be given to the dispatch() method of the interrupt handler. This method is obtained by
using DEVICES_INTERRUPTHANDLER_DISPATCH_I_V as index in the methods array. In effect,
the dispatch() method will execute the run() method of the interrupt handler associated
with this interrupt, that is, the run() method of PeriodicMethaneDetection.

18.2.2 Methane Level

One of the essential parts of the mine pump, is to ensure safe operation of the water pump,
by disallowing it to run when methane is present in the mine shaft. To determine whether the
methane level is critical, a history of measurements is used. This history is maintained using
an array of measurements with a fixed size. Each time a new measurement is conducted,
it is added to this array overwriting the oldest measurement. Then, the methane level is
determined by counting the number of measurements in the history showing methane, and
if this count is above some threshold, the methane level is high. The getMethaneLevel() method,
as shown in Listing 18.3, is used to calculate the current methane level based on the
history.

```java
public int getMethaneLevel() {
    int methaneCount = 0;

    //@loopbound = 10
    for (int i = 0; i < this.history.length; i++) {
        if (this.history[i] == METHANE)
            methaneCount++;
    }

    return methaneCount;
}
```

Listing 18.3: The getMethaneLevel() method used to determine the amount of observed methane.

As shown, the getMethaneLevel() method is implemented using a loop counting the
number of observed methane measurements in the history. This loop is bounded to 10 iterations,
as annotated in line 4, which is equal to the size of the history. The loop bound is
deemed safe since it is equal to the length of the history.
18.2.3 Starting a Motor

The final implementation example is how to start a LEGO motor. The Java code of this is shown in Listing 18.4.

```java
public void start() {
    LegoAVRInterface.motorSpeed(this.legomotorId, SPEED);
}
```

Listing 18.4: An example of how to start a LEGO motor.

As shown, the code is rather simple since it only consists of a call to a native method, which is a wrapper made by us to interact with the motors. The corresponding implementation of the native method is shown in Listing 18.5.

```c
int32 com_tetaj_lego_legoavrinterface_motorspeed_ii_v(const void *methodInfo, int32 *sp) {
    unsigned short motor_no, speed;
    motor_no = sp[0];
    speed = sp[1];
    motor_speed((INT8U) motor_no, (INT8S) speed);
    return NATIVE_FUNCTIONS_SUCCESS;
}
```

Listing 18.5: The implementation of the native method responsible for controlling the motor speed.

The interesting part of this code is found in line 4 and 5, where the two uppermost values on the stack of the HVM are stored in the `motor_no` and `speed` variables, respectively. This is done to comply with the API set for HVM which specifies that a native method can read all its arguments from `sp[0]` and onwards. Afterwards, the LEGO method `motor_speed()` provided by HVM is issued with the arguments provided from the Java code.
The purpose of this chapter is to evaluate TetaJ, both in terms of safety and precision of the
WCET estimates and performance taking into account the effect of the various optimisations.
These evaluations are based on using TetaJ to estimate WCETs of the mine pump control logic
and a number of simpler algorithms. To conclude on safety and precision of the WCETs, these
are compared with time measurements using a stop clock approach using an ATmega2560
software simulator.

19.1 Conducting WCET Analysis using TetaJ

To use TetaJ, the necessary models for the execution environment must be present. In this
case, the hardware model for the ATmega2560 is reused from METAMOC and a JVM model
is generated based on the HVM. For this, we have reused components from TetaJ to construct
a tool capable of this. Listing 19.1 shows how this tool is used to generate the JVM model.

```
> java -jar JvmModelGenerator hvm.elf src/ jvmmmodel.xml
```

**Pseudo-code 19.1:** Execution of the tool used to generate the JVM model based on an executable
and source code of the HVM.

Conducting the actual WCET estimation using TetaJ requires three steps. First the JBC
model is generated using the method signature of the analysed method, the compiled class
files, and source files. Secondly, the generated model is combined with the JVM model and
hardware model. Finally, using this model, UPPAAL is used to conduct the actual WCET
estimation process.

Each of these steps are conducted using separate tools. Listing 19.2 shows the execution
of these steps on the command line which results in a WCET for the methane detection task.

```
> java -jar ModelGenerator bin/ javaClass#main()V src/ jbcmodel.xml
> java -jar ModelCombiner jbcmodel.xml jvmmmodel.xml staticjvmmmodel.xml hardwaremodel.
xml -o combined.xml
> java -jar ModelProcessor combined.xml
```

**Pseudo-code 19.2:** The three commands necessary to conduct a WCET analysis using TetaJ.
19.2 Evaluation of Optimisations

Throughout the development of TetaJ, a number of different optimisations have been applied to mitigate the size of the analysed tasks. Specifically, four optimisations are used in the evaluation of which progress measures, condition optimisation, and UPPAAL template reduction have already been described. The final optimisation is turning on State Space Reduction mode in UPPAAL. This mode is a feature which, when set to most, applies a number of techniques to reduce the memory consumption. The reduction is not exact since UPPAAL cannot know the possible states beforehand, hence in some cases, it is necessary to re-explore states previously removed from memory. Therefore, this is a trade-off between memory consumption and speed.(Bengtsson et al., 1996; David, 2011)

In the following, six different experiments are conducted to evaluate the impact of the optimisations. Specifically, one without optimisations, four with one individual optimisation enabled, and finally one with all optimisations enabled. The optimisations are evaluate according to two different parameters: memory usage, and analysis time.

Each of the experiments are conducted on an identical task written in Java consisting of a while-loop, bounded to 10, which increments a local variable, as illustrated in Listing 19.3.

```java
//@ loopbound = 10
while(i<10){
    i++;
}
```

Listing 19.3: Code example used to evaluate the optimisations.

All of the experiments result in the same WCET, specifically 17262 clock cycles, even though they consume vastly different amounts of memory and analysis time. Table 19.1 lists the memory usages, states explored, and maximum observed memory usages for the different experiments.

<table>
<thead>
<tr>
<th>Optimisation</th>
<th>Analysis time</th>
<th>States explored</th>
<th>Max memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Optimisations</td>
<td>19s</td>
<td>57553</td>
<td>144 MB</td>
</tr>
<tr>
<td>Only Progress Measures</td>
<td>13t 33m 21s</td>
<td>41854143</td>
<td>2,426 MB</td>
</tr>
<tr>
<td>Only State Space Reduction</td>
<td>1m 16s</td>
<td>53732</td>
<td>294 MB</td>
</tr>
<tr>
<td>Only Condition Optimisation</td>
<td>4t 46m 41s</td>
<td>41854143</td>
<td>3,851 MB</td>
</tr>
<tr>
<td>Only Template Reduction</td>
<td>14t 51m 17s</td>
<td>41854143</td>
<td>3,905 MB</td>
</tr>
</tbody>
</table>

Table 19.1: Analysis time, states explored and memory usage of a number of optimisations.

As shown, the experiment with the progress measures optimisation has not been able to draw conclusions within a reasonable time frame, that is, over 60 hours and for this reason,
we decided to terminate it. However, we observed that even after executing in more than 60 hours, memory consumption does not exceed 450 MB. Therefore, we conclude that progress measures do keep the memory consumption low, but as described, this is at the expense of a significantly higher analysis time.

As observed, comparing the results from all optimisations and no optimisations shows a significant difference in both execution time and maximum memory usage. Specifically, the analysis time decreases from almost 14 hours to 19 seconds, and the memory consumption decreases from 3.9 GB to 144 MB. Furthermore, it can be observed that especially the condition-optimisation significantly influences the analysis by safely removing branches. Also, the notable difference in analysis time between no optimisations and template reduction is remarkable. This indicates that having instantiated unnecessary templates in the model, degrades performance of the model checker substantially.

19.3 Evaluating the Applicability of TetaJ

Ensuring that the produced WCET estimates are indeed safe and precise requires a great effort to examine the entire executable and manually sum the execution times of the individual instructions. Since this approach is deemed too cumbersome, we have chosen to use a measurement-based approach. To evaluate the safety and precision of the WCETs, we compare measurement-based WCETs and WCETs obtained by TetaJ for four different algorithms.

19.3.1 Measurement-Based WCET

The measurement-based approach for WCET estimation can be done in a variety of ways. For example, Plsek et al. (2010) use a stopwatch approach, initiating the timer at the very beginning of program execution and stopping it at the very end prior to termination. However, using such an approach for this evaluation is not trivial. One solution would be to monitor the execution time in the software itself and display the elapsed time using the eight LEDs situated on the evaluation board. Unfortunately, this approach will result in interfering with the analysed program since the execution time must be tracked using interrupt handlers. Instead, we chose the solution of using the AVR Simulator part of the AVR Studio (Atmel-Corporation, 2011) IDE. The simulator is capable of simulating the execution of programs on the ATmega2560 used in the case study and during this, it keeps track of the total clock cycle count which can be read after successful execution.

19.3.2 Safety and Precision of WCET Estimates

The comparison of measurement-based WCETs and WCETs obtained by using TetaJ is based on four simple Java examples whose implementations can be found in Appendix G. Table 19.2 shows the resulting WCETs for the examples.
Algorithm | Measured WCET | TetaJ WCET
--- | --- | ---
Iterative Fibonacci | 20,226 cycles | 46,995 cycles
Factorial | 16,075 cycles | 40,939 cycles
Reverse Ordering | 29,732 | 95,429
Bubble Sort | 293,634 | 2,270,401 cycles

Table 19.2: Comparison of measurement-based WCETs and WCETs obtained by TetaJ for different algorithms.

Generally, the measurement-based approach produces under-estimated WCETs, meaning that the results should always be lower than the results obtained using TetaJ. However, the difference between the measurement-based WCETs and WCET estimates obtained by TetaJ is indicating that TetaJ produces overestimates of the WCET and hence provides WCET estimates that uphold the safety criterion. With respect to precision, the results indicate that the WCETs obtained by using TetaJ are not very precise. This is caused by e.g. branches for which TetaJ always follows the most expensive path, which does not necessarily reflect the runtime behaviour.

19.4 WCET Analysis of the Mine Pump

Due to the previous evaluation showing that TetaJ is applicable for WCET analysis of Java programs, we use it to analyse the mine pump from the case study.

The result of the WCET estimation is shown in Table 19.3. These results have been attained by running TetaJ with all available optimisations. As shown, both the estimation times and their respective memory usages are only marginally different. Furthermore, it is our opinion that these values are reasonable taking into account the iterative development methods used in the industry. The low memory usage also supports that the analysis may be conducted on ordinary desktop computers in contrast to using build servers which are often employed for such purposes.

<table>
<thead>
<tr>
<th>Task</th>
<th>Analysis time</th>
<th>States explored</th>
<th>Memory usage</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methane detection</td>
<td>4m 7s</td>
<td>1295779</td>
<td>267 MB</td>
<td>138,420 cycles</td>
</tr>
<tr>
<td>Water level detection</td>
<td>6m 40s</td>
<td>1898334</td>
<td>271 MB</td>
<td>70,712 cycles</td>
</tr>
</tbody>
</table>

Table 19.3: Results of conducting the WCET estimation on the mine pump.

The estimated WCETs are below the respective deadlines of the two tasks which are 560,000 clock cycles for methane detection and 400,000 for water level detection. This means that when running each task in isolation, they are capable of meeting their respective deadlines. However, this does not conclude whether the tasks can meet their deadlines in case
they are running simultaneously. Therefore, to show that this property is still upheld, we have conducted a schedulability analysis using TIMES (DARTS, 2007). Taking into account the WCETs of the tasks and the blocking times related to the shared resource, that is, the water pump actuator, TIMES deems the mine pump schedulable in the given execution environment.
Part V

Conclusion
This chapter presents aspects of the project which could have been done differently and presents aspects subject for further consideration. Most of the aspects are of concern if TetaJ should be employed in an industrial setting. Others are merely improvements which would have added to the applicability of TetaJ.

### 20.1 Unsupported Features

All the described modifications presented in Section 16.3 are necessary for a timing predictable HVM, but some features still remain unsupported. The following list presents some of these:

- Exceptions have generally been removed from the HVM. This is mainly because their inclusion would significantly add to the pessimism of the WCET results. Furthermore, the implementation of exception handling often include unbounded loops, used for propagating exceptions up the call stack, and memory allocation, used for allocating strings with error messages, which both are problematic for the WCET analysis. Thus, exceptions are currently not supported by TetaJ. Note that the depth of the call stack is in principle available when the TCFGs of the analysed task are built, and hence this might be useful if exceptions should be supported.

- Some JBC instructions still contain unbounded loops which cannot be rewritten. For instance, the `tablelookup` and `idiv` JBC instructions both contain loops which are depending on variables whose value can only be determined at runtime. In this case one may, however, be able to use the approach of determining loop bounds symbolically as proposed by Hunt et al. (2006).

- Resource allocations e.g. made by using the `new` operator or when creating a string constant, are not supported since these use `malloc()` for memory allocations. `malloc()` is generally known to exhibit timing unpredictability (Qing Li, 2003) and impedes WCET analysis. Hence, we make the assumption that resource allocations are not part of the time critical phase of the program and must be conducted prior to this. The problem may be circumvented by statically allocating a memory area and use this whenever objects are dynamically created. However, doing so is out the scope of this project and is
also related with introducing new memory types such as scoped and immortal memory known from the RTSJ.

- When generating the TCFGs of the JBC, we discovered that the WCA library is insufficient. During its CFG construction process, WCA will load the appropriate class files. In that process, it conducts a simple check to determine whether the particular class is residing in the Java class library using a regular expression testing if the fully qualified name starts with “java.” or “org.apache”. If a class is from the Java class library, it is skipped and will not be considered further in the process of constructing the CFG. This limits the applicability of WCA for constructing the TCFGs since it Java Class Library functionality cannot be used. The concrete reason for excluding this remains unknown and is not documented anywhere in the documentation for WCA. One of the reasons, however, may be that taking into account the classes from the Java class library significantly increases program size and TCFG size which in turn may have proved difficult when analysing the TCFGs.

We think that the problem may be solved using either of three solutions. First of all, the problem may be solvable either by modifying the WCA implementation or write a new library for constructing the CFGs from the bottom e.g. by using BCEL. The second solution may be to remove the check and employ a similar technique as HVM or ProGuard for pruning unused functionality in referenced libraries thereby possibly yielding that the problem sizes are reduced sufficiently to make WCET analysis possible. Finally, the last solution may be to use another Java Class Library such as Javolution (Javolution, 2011) which is an effort for implementing a new Java Class Library specifically targeted at real-time systems with emphasis on timing predictability.

20.2 Over-simplifying the HVM

Evidently, the HVM has undergone serious modifications in the case study conducted in this project. Some are merely necessary for making it analysable while others are used to make the analysis process easier. In both cases, solutions have been made which could have been done differently especially considering the serious implications they have had to the HVM and its original structure.

Initially, branches in the JVM models significantly affects model checking time and memory consumption. In an attempt to address this issue, many branches have been removed as a consequence of using the condition optimisation. This mitigated model checking time and memory consumption but proved to be inadequate when considered alone. To further mitigate the issue, the HVM underwent a great amount of modifications at the expense of yielding more pessimistic estimates when analysing programs. Specifically, some if-statements with accompanying if-else- and/or else-statements were rewritten to if-statements such that aforementioned optimisation will remove the branching edge. Of course this makes some parts of the analysis insensitive to actual execution paths, hence pessimistic estimates are inevitably a consequence. Instead, a better solution would be to employ DFA to e.g. prune infeasible paths.
The transformation of the cases in the switch-statement of the interpretation loop to individual functions could have been done differently. Upon encountering a new JBC instruction, this adds an overhead. To recapitulate, this modifications was made to easier extract the implementations of the individual JBC instructions but a better solution would be to develop a technique for extracting the implementations directly from the case-statements.

20.3 Foundation for Evaluation

One can question the actual approach for evaluating TetaJ using a single case study. To determine the extent to which TetaJ uphold the flexibility requirement, the evaluation could have been extended with other JVMs such as JamVM, which is also an appropriate JVM for resource-constrained devices, and other processors, e.g. ARM-based processors. The only reason why this was not done during the project, was the considerable effort this entails.

In addition, the evaluation does not make use of a Java real-time profile due to these remaining unsupported by the HVM. One of the prime purposes of these is to enforce a programming model that is applicable for real-time systems development and make the program amenable to static analyses. We made preliminary efforts in extending the HVM with support for SCJ Level 0 using the open-source version oSCJ (Plsek et al., 2010), but we realised that it would entail a significant effort. This is primarily attributed that the documentation is sparse and due to the observation that oSCJ uses threads for conducting CES. As of the current state of the HVM, threading is not supported.

Since a real-time Java profile is not supported, the mine pump was developed using the practices enforced by the real-time profiles manually. Specifically, to avoid serious consequences of dynamic object allocation during the time critical parts of program execution, these were avoided and only conducted during the initialisation phase of the mine pump. Furthermore, since periodic tasks are not explicitly supported, these were manually made by using a combination of hardware objects in the HVM and writing ISRs. This is in line with the ideas by the authors of the PJ profile, who suggest that the programming model should not be enforced by a profile. Instead they envision writing in a PJ-style which is afterwards checked for conformity to the specification of the profile (Bøgholm et al., 2009).

20.4 Requirements of JVM

During the process of making the HVM analysable, we realised the significant effort this entails because alternative solutions must be devised while still preserving the overall program semantics. This was e.g. the case with transforming non-analysable loops. It turned out that performing the different transformations was associated with high complexity. Furthermore, we discovered that the number of branches in the JVM highly affects the WCET analysis time. These must therefore be minimised. Similar to that of bounding the loops, solving this can be difficult. It would be interesting to construct a new JVM specifically targeted at embedded hard real-time systems and WCET analysis. With the knowledge obtained during
the modifications of the HVM, we think that, should we do it again, it would be beneficial for us to build a new JVM with these characteristics instead of modifying an existing one.
Embedded hard real-time systems development is distinguished from traditional systems development since temporal requirements are of utmost importance. Inevitably, this entails conducting schedulability analysis to guarantee that the real-time tasks do not exceed their deadlines. An integral component in this analysis is WCET of the real-time tasks.

Traditionally, real-time systems have been developed in C. However, recent research has proposed that Java is a candidate for being its successor. Using Java has several advantages including safety, maintainability, and productivity. However, in respect to WCET analysis, Java gives rise to complications due to being dependent on a JVM. To overcome this problem, current research has, among others, used hardware implementations of the JVM thereby reducing the execution environment to solely a JVM without considerations about hardware behaviour. A significant disadvantage of this strategy is that it requires that the resulting JBC instructions are executed on special-purpose hardware. In this project, we have presented a novel tool called TetaJ that allows WCET analyses to be conducted for JBC in an execution environment featuring a software implementation of the JVM and processors commonly used in the domain of embedded systems. The fundamental approach for doing this is to view the program analysis problem of statically determining the WCET as model checking problem. In our opinion, TetaJ significantly enhances the potential of Java for real-time systems development in the industry.

With an industrial deployment in mind, TetaJ has been developed such that it can be adopted in a hard real-time development process. That is, TetaJ has been developed to be usable for smaller parts of real-time systems by allowing WCET analysis to be conducted on method level. The advantage is that the WCET analysis can be conducted on individual methods which besides being an integral component in schedulability analyses may prove relevant in a variety of other applications such as for profiling. Furthermore, as part of an industrial deployment, we identified that TetaJ should be usable in an iterative process. This means that the analysis time should be mitigated to address that further development may be dependent on the result.

Generally, model checking is known to be a time-consuming task for larger problems. However, applying a series of optimisations showed that model checking is indeed feasible for even non-trivial examples such as the mine pump control logic. The WCET analysis could in this example be conducted in approximately 7 minutes. From this, we conclude that TetaJ is
indeed an applicable tool for real-time systems development in Java and applies well for an iterative process where frequent use of the tool is key.

To examine the effects of applying the optimisations, we conducted a series of experiments which showed that analysis time was remarkably affected by these which is evident from the result showing the original analysis time being mitigated from 14 hours to 19 seconds for a simple Java program. From this, we conclude that optimisations have substantial impact on analysis time and should hence be applied for ensuring that a model-based approach is feasible.

Safety is of utmost importance when considering WCET estimates for embedded hard real-time systems. To provide indications that TetaJ does uphold this property, a comparison was conducted where WCETs from a measurement-based approach was compared with the results using TetaJ. These showed that TetaJ does uphold the safety property however at the expense of precision.

Through the case study, we used the JVM called HVM which was not originally designed for real-time systems and hence lacked properties such as timing predictability. Therefore, we put effort in applying a wide variety of modifications to the HVM to make this possible. These included multiple optimisations and alternative solutions to obtain a JVM that is timing predictable and is applicable for resource-constrained systems. Some of the modifications that were applied comprise accounting for unbounded loops and rewriting the entire interpretation-loop to enable that the analysis phase of bytecode interpretation can be performed in constant time.

On the basis of the above, we conclude that our presented tool can be used to conduct WCET analysis on hard real-time Java systems, taking into account a common embedded execution environment featuring interpretation-based JVMs and resource-constrained embedded devices. From this, we also conclude that one of the most important step towards a successful adoption of Java for embedded hard real-time systems development in the industry has been taken.
Even though this project has presented a tool capable of conducting WCET analysis of JBC on a wide range of hardware and software implementations of the JVM, provided models are constructed for these, there is still room for improvement in a number of different areas. Due to being a large project involving competencies in many different fields, these improvements have been left out from consideration due to prioritisation.

The case study which was used to evaluate the applicability of TetaJ could have been more elaborate and be directed towards the current practices in real-time systems development in Java. Specifically, the HVM does not support any of the real-time Java profiles such as SCJ and PJ. Most critically, this means that there is no notion of separate memory types including scoped and immortal memories and hence the programming model necessary when developing real-time applications for the HVM is largely restricted. An obvious extension to the HVM is hence the support of a real-time profile. One possibility would be to implement the necessary functionality for supporting oSCJ.

Some JBC instructions remain timing unpredictable. Therefore, another future direction could be to extend TetaJ to support more JBC instructions, such as the JBC instruction for `switch-statements`. Furthermore, it would be appropriate to focus on adding support for the Java Class Library to further extend its applicability.

Another possibility concerning modifying the HVM towards real-time capabilities would be to substitute the current implementation of the garbage collector with one having real-time properties, that is, deterministic behaviour in terms of time and amount of memory reclaimed when it is run. Research has been done in the area of providing deterministic garbage collection e.g. the research conducted by Bacon et al. (2003). Furthermore, it also features modest memory overheads thereby accommodating the needs for resource-constrained embedded systems. Another attempt in this area is the JOP project which has included a real-time garbage collector in their hardware implementation of the JVM (Schoeberl, 2010). This garbage collector is implemented using a concurrently running thread which is assured to not impact the other real-time tasks by assigning it the lowest possible priority.

A tempting extension of TetaJ is to introduce the possibility of conducting verification of functional correctness. In the domain of formal verification, model checking has already shown to be a prominent technique and since the entire control flow of the execution is available in
TetaJ, it may be possible to allow verification of functional properties to be conducted as well. For instance, considering an example where an alarm must be raised in a system, when a specific event occurs. For this, a query can be constructed that examines whether this is always the case using the control flow models. Making such extensions to TetaJ would significantly add to its applicability of embedded real-time systems since such systems can have very stringent requirements that must never be violated.

Usability issues in TetaJ have been of minor concern, but should be accounted for in case TetaJ should be adopted in industrial settings. One of these concerns extending TetaJ with a loop bound analysis. The benefits will be to reduce or in some circumstances completely remove manual loop bound annotations. Providing such loop bounds manually can be a cumbersome task that may concern analysing the behaviour of the program to ensure that the loop bound provides a safe bound. Concerning that this process may potentially need to be done for every loop, it inevitably leads to errors and thereby erroneous WCETs as a result. Employing a loop bound analysis, such as DFA or symbolic definition of loop bounds as suggested by Hunt et al. (2006), may also potentially yield that the bounds are preciser than those provided by the user. Furthermore, to make TetaJ usable in an iterative development process, effort should be put in optimising the UPPAAL models further, such that the WCET analysis can be conducted faster. An optimisation could e.g. be to conduct dead code elimination to reduce the size.

Even though it does not add functionality to TetaJ, it would have been interesting to extend the evaluation to include other execution featuring other common processors and JVMs. As previously noted, the HVM has support for other processors including National Semiconductor’s CR16C and various other interesting JVMs exist for resource-constrained devices such as JamVM which could have been subject for further examination. In terms of using other processors, the problem would entail finding a disassembler such that the executable can be examined for reconstructing its control flow. Using another JVM such as JamVM would most likely entail a greater effort since the JVM model must be constructed. As evidenced by the many modifications that have been applied to the HVM, JamVM would probably need to be rewritten as well.

The experiences gained in modifying the HVM has made us consider writing a new JVM purposely designed for being amenable to static analysis. Some of modifications made to the HVM have extensively changed its design and being associated with a great effort to make.
Part VI

Appendices
The operation of TetaJ entails a transformation from JBC to UPPAAL templates, combining the UPPAAL templates of the JBC with those of the JVM and hardware. Finally, the final UPPAAL system must be processed to estimate the WCET. Figure A.1 presents the system architecture of TetaJ containing this functionality through respective components.

Notice the optional components of using TetaJ that ease the task of modelling the JVM used in our case study. These have been constructed for easing the task of modifying the HVM since each modification requires a new JVM model to be constructed.

The three larger components and their constituents are described in the following.

A.1 Model Generator

TetaJ relies on three models, namely; a program model, a JVM model, and a hardware model. The JVM and hardware models are expected to remain static in contrast to the program model that rely on source code which is iteratively changed during the development. To ease the modelling for the developers, TetaJ therefore supports automatically modelling a JBC program. Furthermore, as previously described, TetaJ offers the optional feature of automating the JVM model generation for the HVM.

The Model Generator consists of six components:

- **JBC TCFG Builder**: This component transforms the given JBC into TCFG.
- **AVR TCFG Builder**: This component is used to help automating the process of transforming the JVM used in the case study into TCFG.
- **Loop Detector**: This component is responsible for conducting a loop bound analysis on the TCFG. The Annotation Extractor component is used in this context to extract user annotated loop bounds.
- **Annotation Extractor**: As part of analysing the TCFG, user annotations might be necessary. The Annotation Extractor Component is responsible for extracting annotations from the source code.
A.2. MODEL COMBINER

Condition Optimisation  This component is responsible for optimising branches in the TCFGs, by safely removing some of them.

Progress Measures  In respect to UPPAAL, progress measures are used to reduce the memory consumption. Since this must be handled carefully at loops, this component is responsible for annotating the TCFG with the needed information for the later model construction.

UPPAAL Model Builder  This component transforms the analysed TCFGs into UPPAAL templates expressed in XML.

A.2 Model Combiner

The purpose of this component is to combine the individual models of the system into one that can be input to the Model Processor which uses the UPPAAL model checker. The models needed by the Model Combiner are:

JBC Program Model  This is the UPPAAL model the Model Generator will produce and reflect the behaviour of the program regardless of underlying execution environment.

JVM Model  This model describes the behaviour of the used JVM. The model must comply with our intermediate METAMOC interface, such that METAMOC hardware models can be used in the WCET analysis.

Processor Model  In most cases, the JVM will be implemented in software, thereby relying on an underlying execution environment for actual execution. This model is used to describe the behaviour of the processor in terms of e.g. cache-model and pipeline-model. This model is based on the hardware models provided by METAMOC.

The provided models are in XML and hence the functionality of the tool is a matter of manipulating XML documents. This is considered trivial and this part is hence omitted for further explanation. In addition to generate the models, the Model Combiner needs to perform an additional step in which it cleans up parts of them for optimising performance. This is the responsiblity of the Model Cleaner which consults the XML files of the models and removes unnecessary templates for optimising performance.

A.3 Model Processor

The final tool in TetaJ is the Model Processor whose responsibility involves interacting with UPPAAL to determine the WCET of the system. Besides, this tool is also responsible for generating an appropriate output to the user based on the response from UPPAAL.
Figure A.1: Overview of the system architecture of the toolchain. The components marked by dashed rectangles are optional, and are only used for convenience in our case study.
This appendix describes the design of the *JBC TCFG Builder* and the *AVR TCFG Builder* components used to transform the JBC of a Java program to a set of TCFGs and the AVR machine code of the HVM into another set of TCFGs, respectively. Their actual placement in the overall system architecture can be seen in Figure A.1.

The class diagram depicting the classes involved in these components and how they are interconnected is shown in Figure B.1.

![UML class diagram showing the structure of the TCFG Generator.](image)

As described in Chapter 11, the *JBC TCFG Builder* component uses the WCA library for constructing the actual CFGs. The component uses the functionality provided to construct an instance of `ControlFlowGraph` representing the CFG of the method. The *AVR TCFG*
APPENDIX B. TCFG GENERATORS

Builder uses the Avrora library for constructing CFGs of assembly for the AVR processor. The TCFG Generator is easily extensible to allow that the JVM targets other processors.

Below is a description of each of the classes constituting the TCFG Generators of TetaJ:

ITCFGGenerator Since TetaJ introduces TCFG, it must be possible to transform other CFG formats into this. Implementing the ITCFGGenerator interface ensures that the underlying TCFG Generators provide the same API.

AVRToTCFG This class is used to transform an instance of the ControlFlowGraph class of the Avrora library to a corresponding ControlFlowGraph used in TCFG.

JBCToTCFG This class is responsible for transforming the ControlFlowGraph of the WCA library into the ControlFlowGraph used in TCFG. In this process, the WcaCfgHelper is used as a helper class.

ControlFlowGraph To represent a CFG, this class is used. The class contains a list of corresponding basic blocks represented by the BasicBlock class.

BasicBlock This class represents basic blocks in the CFG. Therefore, the class has a list of corresponding Instruction instances denoting the instructions of the basic block. Furthermore, if a method call is made in the basic block, the basic block holds a reference to the ControlFlowGraph representing the called method.

Instruction This class represents instructions of basic blocks. The class therefore contains information such as opcode and line number.

WcaCfgHelper This class is used by the JBCToTCFG to provide the functionality for using the WCA library to generate TCFGs for a given method and all its invoked methods. The TCFGs are stored in a HashMap with their fully qualified method names as lookup keys and the corresponding ControlFlowGraph as value.

HelperCfg By using the WCA library, this class generates and holds the CFG of a single method. Furthermore, the class provides the functionality for returning a list of invoked methods from its CFG.
This appendix describes the \textit{TetaJ CFG Analyser} and the \textit{Annotation Extractor} components used to analyse and manipulate TCFGs. The purpose of the \textit{TCFG Analyser} is to conduct a number of analyses on a given TCFG. The \textit{Annotation Extractor} component is used in relation to one of these analyses to obtain annotations given a file path and a line number.

The class diagram depicting the classes involved in these components and how they are interconnected is shown in Figure C.1.

Below is a description of each of the classes constituting the components:

\textbf{TCfgAnalysis} \quad The \texttt{TCfgAnalysis} class is responsible for conducting analyses on the TCFGs prior to conducting the transformation to UPPAAL templates. Currently three analyses are conducted to determine loop bounds, for removing branches associated with if-conditions, and annotate the TCFGs with information related to progress measures, respectively. The number of analyses is easily extended by using analyses that comply with the \texttt{IAnalysis} interface. The \texttt{TCfgAnalysis} will perform the analyses in sequence.

\textbf{LoopBoundAnalysis} \quad The \texttt{LoopBoundAnalysis} class iterates over all TCFGs and detects all occurrences of loops in these. The loop bound is then extracted using an annotation extractor conforming to the \texttt{IAnnotationExtractor} interface.

\textbf{ConditionAnalysis} \quad The \texttt{ConditionAnalysis} class iterates over all TCFGs and determines all occurrences of branch statements. Given that the particular branch branches to a condition body and a common condition exit, this analysis will remove the branching edge in the TCFG that jumps over the body of the condition.

\textbf{ProgressMeasureAnalysis} \quad The \texttt{ProgressMeasureAnalysis} class iterates over all TCFGs in order to annotate information necessary for inserting progress measures into the UPPAAL templates. This mainly relates to how progress measures are used in conjunction with loops.

\textbf{IAnnotationExtractor} \quad This interface defines the API used when extracting annotations. The interface dictates a single method that must be implemented whose signature includes the path of the source code of the class for which annotations must be extracted and
the line number indicating the source code line for which an accompanying annotation is requested. An instance of Annotation will be retrieved in case an accompanying annotation is found.

CStyleAnnotationExtractor This is a concrete class implementing the IAnnotationExtractor interface with the purpose of extracting annotations embedded in C-style comments hence working for Java and C programs.

FileFinder This class is used for retrieving the particular source code file corresponding to the generated binary file. The CStyleAnnotationExtractor will request this file by consulting the FileFinder with the path of the file. The FileFinder will locate, and
retrieve the file. To support that successive requests for annotations potentially will be targeted at the same source code file, the FileFinder will first look in the FileCache class for faster retrieval.

FileCache This class is responsible for caching files that have previously been accessed for retrieving annotations. This is to reduce the overhead involved in opening, reading, and closing the same file multiple times.

AnnotationParser The AnnotationParser receives an instance of File corresponding to the file for which annotations must be extracted. Furthermore, a line number is used to indicate the particular source code line from where an annotation is attempted to be found.

Annotation Instances of this class represent the particular annotation that will be returned as part of requesting an annotation to be retrieved for a particular Java class file and line number. The class contains information about which type of annotation it represents and the annotation value itself.
This appendix describes the design of the Model Generator tasked with transforming TCFGs into corresponding UPPAAL templates.

The class diagram depicting the classes involved in this component and how they are interconnected is shown in Figure D.1.

![Figure D.1: UML class diagram showing the structure of the Model Generator.](image)

The individual classes comprising the Model Generator are described in the following:

**UppaalModelGenerator**  
The UppaalModelGenerator class ensures that the NTA is initialised. To do this, a library provided by the JOP project for constructing UPPAAL models is used. The UppaalModelGenerator delegates the task of generating templates for each of the TCFGs to the MethodTemplate class. Furthermore, a number of channels are declared in the UPPAAL system.

**MethodTemplate**  
The MethodTemplate generates a UPPAAL template for a given TCFG. The locations and transitions of the individual basic blocks in the TCFG are generated by the BasicBlockModel class. This class further extracts loop bounds from the TCFG to insert the necessary guards and updates used to simulate loops in the model. The necessary locations for method invocation and return together with a name of the template are also inserted.

**BasicBlockModel**  
The BasicBlockModel class is responsible for generating the individual locations and transitions between these for a single basic block. The transitions between these groups of locations are inserted by the MethodTemplate class.
This appendix describes the design of the *Model Combiner*, capable of merging UPPAAL systems and conducting a UPPAAL template cleanup optimisation.

A class diagram depicting the classes implementing this tool is shown in Figure E.1.

![Class Diagram](image.png)

**Figure E.1**: UML class diagram showing the structure of the Model Combiner tool.

The individual classes comprising the *Model Combiner* are described in the following.

- **ModelCombiner** The *ModelCombiner* class is responsible for merging multiple UPPAAL systems into one. As input it takes a list of xml files containing UPPAAL systems for which it uses *UPPAALXmlProcessor* to merge them into one file.

- **ModelCleaner** The *ModelCleaner* class is responsible for optimising the final UPPAAL system by removing unused JBC implementation UPPAAL templates. As input it takes a *UPPAALXmlProcessor* instance which represents the combined model, which it conducts its changes directly on.

- **UPPAALXmlProcessor** The *UPPAALXmlProcessor* class encapsulates all logic related to manipulating UPPAAL xml files.
This appendix describes the design of the **Model Processor**, capable of using UPPAAL to determine the WCET of the provided models. Therefore, this component is responsible for conducting the actual WCET analysis.

The tool consists of one class:

**UppaalProcessor**  Similar to the purpose of the tool, this class is responsible for executing UPPAAL to conduct the WCET analysis on the provided model. Furthermore, the result from UPPAAL is formatted in an appropriate way to the user.
This appendix depicts the four algorithms used in the evaluation of TetaJ.

```
public void fibonacci()
{
    //loopbound=10
    for (int i = 0; i < n; i++)
    {
        z = x + y;
        x = y;
        y = z;
    }
    return;
}
```

**Listing G.1:** Implementation of iterative fibonacci.
public void factorial()
{
    int N = 10;
    int product = 1;

    //@loopbound=10
    for ( int j=1; j<=N; j++ )
        product *= j;

    return;
}

Listing G.2: Implementation of factorial.

public void bubble_sort(int n){
    int i, j, t=0;

    //@loopbound = 10
    for(i = 0; i < n; i++){
        //@loopbound = 9
        for(j = 1; j < (n-i); j++){

            if(a[j-1] > a[j]){  
                t = a[j-1];
                a[j-1]=a[j];
                a[j]=t;
            }
        }
    }
}

```java
public void reverse() {
    int left = 0;
    int right = a.length - 1;

    // @loopbound = 5
    while (left < right) {
        int temp = a[left];
        a[left] = a[right];
        a[right] = temp;

        left++;  // left++;
        right--;  // right--;
    }
}
```

Listing G.4: Implementation of binary search.
This project focuses on the construction of a novel tool called *Tool for Execution Time Analysis of Java Bytecode* (TetaJ) whose prime purpose is to allow Worst Case Execution Time (WCET) analysis to be conducted on Java Bytecode. During recent years, Java has drawn much interest in being adopted for real-time embedded systems development as seen by the great efforts of the Real-Time Specification for Java (RTSJ) project and various other initiatives. These initiatives introduce changes to the Java Virtual Machine (JVM) and provide additional facilities using libraries in order to provide a new programming model allowing Java to be used for real-time systems development.

An integral component in the adoption of Java in the domain of real-time development is tool support allowing temporal and functional requirements to be guaranteed such that critical errors can be avoided at runtime. The overall goal in relation to the temporal requirements is to show that the system is schedulable under all possible circumstances, that is, real-time tasks have deadlines and it must be ensured that these can be upheld under execution. An integral component in the schedulability analysis is the WCET of the real-time task in question.

Since the advent of introducing Java for real-time systems development, WCET analysis in this context has naturally been of much research. In its current state, most of the tools devised for such purposes, notably the WCET Analyzer (WCA), rely on special-purpose hardware capable of natively executing the Java Bytecode instructions. Effectively, this execution environment is a JVM implemented in hardware and is specifically designed for easing WCET analysis. However, relying on this execution environment of special purpose hardware has not shown great impact in the industry. In this relation, what is needed to ensure successful adoption of Java for real-time systems development is that Java can be used with common embedded processors such as those from the ARM family and Atmel AVR family while still allowing the aforementioned schedulability and WCET analyses to be conducted. We present TetaJ, which takes a novel approach to WCET analysis of JBC programs, incorporating both the notion of a software implementation of the JVM and common embedded hardware.

The fundamental technique employed in TetaJ for WCET estimation is to construct a model of the analysed task and combine this with models of the given execution environment, and to use the state-of-the-art UPPAAL model checker in order to determine the WCET of the task.
To evaluate the applicability of TetaJ, a case study is conducted using a construction of the classic text-book example of a real-time control software in a mine pump. A physical construction using LEGO with sensors and actuators is used to verify the control software that is written in the Java programming language. The control software is resident on an Atmel STK600 evaluation board featuring an Atmel AVR ATmega2560 processor which represent a good example of hardware that has many applications in embedded systems in general. The execution environment also features a software implementation of the JVM called Hardware near Virtual Machine (HVM) which is specifically designed for running on resource-constrained systems. The HVM is modified to make it more amenable to WCET analysis, before constructing a model of it. The hardware models are reused from the related research project called METAMOC who has specialised in modelling hardware for WCET analysis. While this evaluation features concrete details about the execution environment, it is stressed that TetaJ can potentially be applied for a wide range of hardware and implementations of the JVM.


Huber, B.(2011). Uppaal optimisations. Special communication with Benedikt Huber, research and teaching assistant, Vienna University of Technology.


Sestoft, P.(2005a). Java performance reducing time and space consumption. IT University of Copenhagen, Denmark.


<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOT</td>
<td>Ahead-Of-Time</td>
<td>36</td>
</tr>
<tr>
<td>ATC</td>
<td>Asynchronous Transfer of Control</td>
<td>29</td>
</tr>
<tr>
<td>BCEL</td>
<td>Byte Code Engineering Library</td>
<td>63</td>
</tr>
<tr>
<td>BCET</td>
<td>Best Case Execution Time</td>
<td>56</td>
</tr>
<tr>
<td>CES</td>
<td>Cyclic Executive Scheduling</td>
<td>23</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
<td>12</td>
</tr>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
<td>37</td>
</tr>
<tr>
<td>CRL</td>
<td>Control flow Representation Language</td>
<td>45</td>
</tr>
<tr>
<td>DFA</td>
<td>Data-Flow Analysis</td>
<td>15</td>
</tr>
<tr>
<td>DFN</td>
<td>Depth-First Number</td>
<td>67</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First Scheduling</td>
<td>23</td>
</tr>
<tr>
<td>ESC</td>
<td>Electronic Stability Control</td>
<td>7</td>
</tr>
<tr>
<td>FPS</td>
<td>Fixed-Priority Scheduling</td>
<td>23</td>
</tr>
<tr>
<td>HRT-HOOD</td>
<td>Hard Real-Time HOOD</td>
<td>42</td>
</tr>
<tr>
<td>HVM</td>
<td>Hardware near Virtual Machine</td>
<td>3</td>
</tr>
<tr>
<td>ILP</td>
<td>Integer Linear Programming</td>
<td>17</td>
</tr>
</tbody>
</table>
**BIBLIOGRAPHY**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPET</td>
<td>Implicit Path Enumeration Technique</td>
<td>16</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
<td>89</td>
</tr>
<tr>
<td>JBC</td>
<td>Java Bytecode</td>
<td>2</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
<td>36</td>
</tr>
<tr>
<td>JML</td>
<td>Java Modelling Language</td>
<td>51</td>
</tr>
<tr>
<td>JOP</td>
<td>Java Optimized Processor</td>
<td>36</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
<td>2</td>
</tr>
<tr>
<td>KVM</td>
<td>K Virtual Machine</td>
<td>37</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
<td>46</td>
</tr>
<tr>
<td>NTA</td>
<td>Network of Timed Automata</td>
<td>52</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
<td>3</td>
</tr>
<tr>
<td>PJ</td>
<td>Predictable Java</td>
<td>2</td>
</tr>
<tr>
<td>RTC</td>
<td>Real-Time Counter</td>
<td>99</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
<td>23</td>
</tr>
<tr>
<td>RTSJ</td>
<td>Real-Time Specification for Java</td>
<td>28</td>
</tr>
<tr>
<td>SARTS</td>
<td>Schedulability Analyzer for Real-Time Systems</td>
<td>43</td>
</tr>
<tr>
<td>SCJ</td>
<td>Safety Critical Java</td>
<td>2</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>TCFG</td>
<td>Tetaj CFG representation</td>
<td>62</td>
</tr>
<tr>
<td>TetaJ</td>
<td>Tool for Execution Time Analysis of Java bytecode</td>
<td>3</td>
</tr>
<tr>
<td>VMTM</td>
<td>Virtual Machine Timing Models</td>
<td>49</td>
</tr>
<tr>
<td>WALA</td>
<td>T. J. Watson Libraries for Analysis</td>
<td>64</td>
</tr>
<tr>
<td>WCA</td>
<td>WCET Analyzer</td>
<td>45</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
<td>2</td>
</tr>
<tr>
<td>XAC</td>
<td>eXtensible Annotations Class</td>
<td>49</td>
</tr>
<tr>
<td>XRTJ</td>
<td>eXtensible high-integrity Real-Time Java</td>
<td>48</td>
</tr>
</tbody>
</table>