A Monitoring Approach to Facilitate Run-time Verification of Software in Deeply Embedded Systems

Author
Conal Watterson
B. Eng. (1st Hons.), M. Eng.

Supervisor
Dr. Donal Heffernan

Submitted for the degree of Doctor of Philosophy
University of Limerick
Ireland

March 2010
A Monitoring Approach to Facilitate Run-time Verification of Software in Deeply Embedded Systems

Conal Watterson

Abstract

With embedded systems growing in hardware and software complexity, it is becoming increasingly difficult to avoid failures, despite costly design and testing efforts. Monitoring the behaviour of an embedded system's software is useful both during the product testing phase and in the field environment and can contribute to confidence in the overall system. Run-time verification combines monitoring with formal methods by checking specified rules against run-time observations. Embedded systems pose some problems for run-time verification, as their limited resources may preclude internal monitoring features and their self-contained design may inhibit sufficient observation of the software execution by an external monitor.

The main aim of the research outlined in this thesis is to investigate the feasibility of using run-time verification as a monitoring approach for software executing in an embedded system. An experimental evaluation framework is described that supports the study of this approach. The embedded system comprises a Java Optimised Processor (JOP) soft processor, instantiated in the fabric of an FPGA (field programmable gate array). The experimental system employs the Java-MaC (Java Monitoring and Checking) run-time verification method, arranged to indirectly monitor the run-time behaviour of a software application executing on JOP.

The experimental framework successfully allows run-time verification of a software application executing on the FPGA. A case study is presented that demonstrates that the monitoring approach is sufficient to observe a number of parameters used in an example ‘railroad crossing’ run-time verification scenario. A second case study aims to verify that a set of tasks scheduled under the rate-monotonic algorithm is scheduled at run time as predicted by static offline analysis. The results of the case studies provide detail on the potential and limitations of the monitoring arrangement. The gathered data is used to support recommendations for monitoring embedded systems, presented at the conclusion of this thesis.
Declaration

This thesis is presented in fulfilment of the requirements for a doctorate. I hereby declare that this thesis is entirely my own work and has not been submitted to any other University or higher educational institution, nor has it been submitted for any other award in this University. Where use has been made of the work of others, it has been fully acknowledged and fully referenced.

Signature:

__________________________________________________________________________

Conal Watterson

May 2010
Dedication

This work is dedicated to my parents,

Michael and Jaye Watterson
Acknowledgements

I would like first and foremost to thank my supervisor, Dr. Donal Heffernan, for all his support and guidance throughout this research.

I also thank Science Foundation Ireland and the Irish government for PhD scholarship funding and their support for this research and the inter-University research effort, the Centre for Telecommunications Value-Chain Research (CTVR). I wish to express my gratitude to all at the Circuits and Systems Research Centre for resources, time and a good work environment.

I wish to thank Dr. Martin Schoeberl for his helpful responses to enquiries and troubleshooting and also for all his work in developing and supporting the Java Optimised Processor. Also I thank Prof. Oleg Sokolsky for providing particular details concerning Java-MaC.

I also would like to thank the external examiner Dr. Michael Schukat and the internal examiner Dr. Richard Conway for their work in reviewing this thesis and for their recommendations, which were of great benefit in the improving this thesis.

Finally I thank my family and friends for all their support, which was particularly welcome towards the end of this work.
# Table of Contents

**LIST OF FIGURES** ........................................................................................................... XI

**LIST OF TABLES** ............................................................................................................ XIII

**GLOSSARY** ...................................................................................................................... XIV

**CHAPTER 1 – INTRODUCTION** ...................................................................................... 1

1.1 INTRODUCTION ............................................................................................................. 2
1.2 BACKGROUND ............................................................................................................... 3
1.3 RATIONALE FOR THE PROJECT ............................................................................... 5
1.4 OBJECTIVES ................................................................................................................ 6
1.5 SUBJECTS OF LITERATURE REVIEW .................................................................... 7
1.6 PROJECT BACKGROUND ............................................................................................ 9
1.7 PROJECT OVERVIEW ................................................................................................ 10
  1.7.1 Overview of literature review ............................................................................. 10
  1.7.2 Arranging a test platform .................................................................................... 11
  1.7.3 Test cases ............................................................................................................. 13
1.8 NOVELTY OF THE WORK .......................................................................................... 14
1.9 THESIS LAYOUT ......................................................................................................... 15
1.10 RESEARCH PUBLICATIONS ...................................................................................... 16
1.11 SUMMARY ................................................................................................................ 17

**CHAPTER 2 – MONITORING OF SOFTWARE EXECUTION** ............................................. 18

2.1 INTRODUCTION ............................................................................................................. 19
2.2 MONITORING OF SOFTWARE .................................................................................... 19
2.3 APPROACHES TO MONITORING ............................................................................. 20
  2.3.1 Hardware monitors ............................................................................................. 21
  2.3.2 Software monitors ................................................................................................ 22
  2.3.3 Hybrid monitors .................................................................................................. 24
2.4 ON-CHIP MONITORS ............................................................................................... 25
  2.4.2 On-chip monitoring research areas ..................................................................... 26
2.5 SUGGESTED MONITORING APPROACH ............................................................... 29
  2.5.1 Proposed monitoring arrangement ..................................................................... 31
2.6 SUMMARY .................................................................................................................. 32

**CHAPTER 3 – RUN-TIME VERIFICATION OF EMBEDDED SYSTEMS** ......................... 33

3.1 INTRODUCTION ............................................................................................................. 34
3.2 RUN-TIME VERIFICATION .......................................................................................... 34
  3.2.1 Expressing monitored properties using LTL ...................................................... 35
  3.2.2 Methods of observation to facilitate run-time verification .................................. 36
9.4.2  Results of railroad crossing case study............................................................... 155
9.4.3  Results of rate-monotonic scheduling analysis......................................................... 155
9.4.4  Overhead of monitoring arrangement...................................................................... 156
9.5  IMPROVED HOST SYSTEM USING JAVA-MAC/JOP ...................................................... 157
9.6  SUGGESTIONS FOR FURTHER WORK AND APPLICATIONS........................................... 158
  9.6.1  Monitoring-oriented system design........................................................................ 158
  9.6.2  Lifetime monitoring .............................................................................................. 159
  9.6.3  Using the monitoring approach for an automatic failsafe ........................................... 159
9.6.4  Potential application for enhanced product support .................................................. 160
9.7  SUMMARY................................................................................................................. 160
List of Figures

FIGURE 1.1 – FLOW CHART FOR A RUN-TIME VERIFICATION PROCEDURE ........................................ 2
FIGURE 1.2 – COMPARISON OF JAVA APPLICATION ON (A) JVM/PC VS. (B) JOP/FPGA ..................... 11
FIGURE 1.3 – ARRANGEMENT OF JAVA-MAC TO VERIFY AN APPLICATION .................................... 12
FIGURE 1.4 – CONCEPTUAL VIEW OF PLATFORM FOR EXPERIMENTAL WORK ............................. 13
FIGURE 2.1 – SIMPLIFIED VIEW OF A TYPICAL HARDWARE MONITOR ........................................ 21
FIGURE 2.2 – ABSTRACT VIEW SHOWING POSSIBLE LOCATIONS OF MONITORING SOFTWARE WITHIN A
TARGET SYSTEM ............................................................ 23
FIGURE 2.3 – ABSTRACT VIEW OF A HYBRID MONITORING ARRANGEMENT ................................. 25
FIGURE 2.4 – ASSERTION-BASED RUN-TIME DEBUGGER ............................................................. 28
FIGURE 2.5 – THEORETICAL ON-CHIP (HYBRID) MONITORING ARRANGEMENT ........................ 31
FIGURE 4.1 – TYPICAL JAVA-MAC ARRANGEMENT ........................................................................ 42
FIGURE 4.2 – SYNTAX FOR EVENTS AND CONDITIONS .............................................................. 43
FIGURE 4.3 – DENOTATION OF CONDITIONS .................................................................................. 44
FIGURE 4.4 – SEMANTICS OF EVENTS AND CONDITIONS ............................................................ 45
FIGURE 4.5 – PEDL SCRIPT EXAMPLE ............................................................................................ 46
FIGURE 4.6 – STRUCTURE OF MEDL SPECIFICATION ................................................................... 47
FIGURE 4.7 – PROPOSED JAVA-MAC DEPLOYMENT ....................................................................... 52
FIGURE 5.1 – JOP APPLICATION BUILD PROCESS .......................................................................... 59
FIGURE 5.2 – JOP DOWNLOAD PROCESS ....................................................................................... 60
FIGURE 6.1 – ELEMENTS OF A JAVA-MAC DEPLOYMENT WITH THE JOP PLATFORM ..................... 64
FIGURE 6.2 – MODIFIED BUILD CHAIN, OUTPUT REDIRECTED TO INTERMEDIATE APPLICATION .... 66
FIGURE 6.3 – COMPARISON OF ELEMENTS OF THE TARGET AND SKELETON APPLICATIONS ......... 67
FIGURE 6.4 – RUN-TIME VERIFICATION USING THE SKELETON APPLICATION ............................. 69
FIGURE 6.5 – DIAGRAM OF PRIME COUNTER MAIN EXECUTION LOOP ......................................... 71
FIGURE 6.6 – TARGET APPLICATION CODE SHOWING PLACEMENT OF ERROR ............................ 72
FIGURE 6.7 – MEDL SPECIFICATION FOR PRIME NUMBER COUNTER ........................................... 73
FIGURE 6.8 – PEDL SPECIFICATION FOR PRIME NUMBER COUNTER ............................................ 74
FIGURE 6.9 – CODE FOR PRIME NUMBER COUNTER INTERMEDIATE APPLICATION ..................... 75
FIGURE 7.1 – COMPONENTS OF THE RAILROAD CROSSING IMPLEMENTATION ............................ 81
FIGURE 7.2 – SEQUENCE OF SIGNALLING IN THE APPLICATION .................................................. 82
FIGURE 7.3 – MEDL SCRIPT FOR RAILROAD CROSSING .............................................................. 82
FIGURE 7.4 – PEDL SCRIPT FOR RAILROAD CROSSING .............................................................. 83
FIGURE 7.5 – COMPONENTS OF THE RAILROAD CROSSING MODEL (NOT TO SCALE) .................... 85
FIGURE 7.6 – CLASSES OF THE JOP RAILROAD CROSSING APPLICATION ..................................... 86
FIGURE 7.7 – STATE TRANSITION DIAGRAM FOR CONTROLLERMODEL .................................... 87
FIGURE 7.8 – SEQUENCE DIAGRAM OF THE JOP RAILROAD CROSSING APPLICATION ................ 88
FIGURE 7.9 – MEDL SPECIFICATION FOR THE JOP RAILROAD CROSSING IMPLEMENTATION ....... 89
FIGURE 7.10 – PEDL SPECIFICATION FOR THE JOP RAILROAD CROSSING IMPLEMENTATION ....... 90
List of Tables

Table 7.1 – Differences between railroad crossing implementations ........................................ 91
Table 7.2 – Execution time and cycles for additional bytecodes .............................................. 94
Table 8.1 – Task parameters for the scenario .............................................................................. 103
Table 8.2 – Execution times provided by WCET analysis ......................................................... 125
Table 8.3 – Calculations of required loop bounds ...................................................................... 125
Table 8.4 – Results of WCET analysis for all three tasks ............................................................ 130
Table 8.5 – Calculated execution times and periods based on timestamps .............................. 131
Table 8.6 – Measured execution times and differences compared to WCET values ............... 132
Table 8.7 – Summary of measurements over two seconds of execution time ......................... 133
Table 8.8 – Delay in sending/receiving one observation across the serial link ....................... 135
Table 9.1 – List of elements and attributes of proposed monitoring approach ....................... 153
Glossary

**API (Application-Programmer Interface):** Libraries of programming methods to allow programmers to incorporate specific functionality in applications.

**Assertion:** A programming statement placed in software code that should evaluate to true at the point in execution where it is placed. These assertions can be checked to ensure that they do indeed hold during program execution.

**Bytecode:** A pseudo-machine code to which Java application code is compiled such that it can be easily interpreted by a JVM.

**Condition:** A state that pertains to part of the target application, for example a variable remaining a certain value. Changes in conditions are precipitated by events.

**Embedded system:** A system where many features are incorporated in a compact device or onto a single chip.

**Event:** A change in the state of the target application or some aspect thereof, for example, the application entering/exiting execution of a method.

**FPGA (Field-Programmable Gate Array):** A device comprising an array of logic gates that can be programmed and reprogrammed with system-on-chip designs.

**HDL (Hardware Description Language):** A programming-like language used to specify circuit/processor designs to be instantiated to FPGA.

**Host:** The system upon which a monitor's evaluation of observations occurs or that hosts a run-time verification engine.

**HPET (High Performance Event Timer):** A high resolution hardware timer for PCs.

**Instantiate, Instantiation:** The process of programming a circuit or processor design to an FPGA.

**Instrumentation:** The adding of monitoring code to the target software in order to output information about the internal operation of the program. This code can be regarded a ‘software probe’ or an ‘instrument’. Application code to which monitoring code has been added is described as ‘instrumented’ code.

**Interference/Intrusion:** The effect introduced to the execution of the target application by the addition of the monitoring apparatus. For example, the execution of instrumented code can potentially delay the ordinary execution of the target application causing its behaviour to alter.

**IP block (Intellectual Property block), IP core:** A hardware design module forming intellectual property that can be licensed for use as a component in other hardware designs.

**Java:** A modern object-oriented programming language and execution platform developed by Sun Microsystems.

**Java API:** The set of programming libraries for the Java programming language and platform.

**Java-MaC (Java Monitoring and Checking):** A version of the MaC (Monitoring and Checking) verification engine developed for verifying Java applications.
JOP (Java Optimized Processor), JOP platform: A platform for executing Java applications on FPGAs. The platform comprises a soft-processor (the JOP core) and a set of API libraries and development utilities.

JOP API: A set of programming libraries for the JOP platform that replicate some of the functionality of the standard Java API, as well as providing extensions to utilise capabilities of the JOP core.

JOP core: A soft processor for Java that acts in place of a JVM, translating prepared Java byte code to native microcode and executing it.

JTAG (Joint Test Action Group): A widely-implemented standard for a common debug port for embedded systems.

JVM (Java Virtual Machine): The standard execution target for Java applications, interpreting Java byte code and executing corresponding native processor instructions.

LTL (Linear Temporal Logic): A formal logic with operators relating to time in addition to standard logic operators.

Monitor: The apparatus used for observing execution behaviour of the target, but also commonly including further apparatus used for analysis of execution behaviour.

Monitoring: The process of gathering observations and evaluating those observations. Also refers to online monitoring, run-time monitoring; the act of observing run-time execution behaviour and performing analysis at run time as events are received by the monitoring apparatus, as opposed to offline monitoring; the act of observing run-time execution behaviour and logging events, i.e. the trace, for later analysis.

Observations: Data gathered concerning the execution behaviour of software.

Overhead: Overhead refers specifically to the execution time required to execute any additional code introduced by instrumentation.

Probe: Part of the monitoring apparatus attached to the system (or placed within it) in order to provide information about the system's internal operation. Probes provide an intermediate output from the system in addition to its end output, allowing the system to be seen as more than a 'black box'.

PSL (Property Specification Language): A language used to specify assertions used for verification of hardware designs.

RTL (Register Transfer Language): A constrained form of hardware description language code that is ready for synthesis in hardware.

Run time: The time at which software is in execution.

Run-time checking: Monitoring a software program and checking it at run time against some specified parameters (usually expressed in terms of events and conditions, sometimes combined as logical and/or temporal expressions).

Run-time verification: The verification of execution behaviour captured by monitoring against specified desired behaviour.

Soft processor: A processor design that is instantiated to an FPGA rather than implemented as a dedicated chip.
**System-on-chip**: A hardware design where in addition to the processor, various hardware resources are incorporated onto a single chip rather than as discreet chip components on a board.

**Target**: The subject of monitoring, run-time verification or other scrutiny, i.e. the system to which a monitor is attached. Generally a software program, the *target application*, is monitored. The system in which the software executes is referred to as the *target system*.

**Trace**: A sequence of run-time events in the execution history of the target application. This sequence may be recorded in a log (also referred to as a trace) or received at run time by some part of the monitoring apparatus.

**WCET (Worst-Case Execution Time)**: The time taken for software to execute under a worst-case scenario, i.e. the maximum time expected for execution of the software.
CHAPTER 1 – INTRODUCTION
1.1 INTRODUCTION

This thesis is concerned with investigating key issues for monitoring embedded systems: providing assurances that software is operating within parameters, sufficiently observing the execution behaviour in order to verify such parameters, and minimising interference from arranging the monitoring, in order that crucial timing parameters of the target software are not affected. A central aim of this thesis was to establish if the concept of run-time verification could be applied to monitoring embedded software applications in order to provide a good monitoring solution for the issues highlighted above.

Figure 1.1 – Flow chart for a run-time verification procedure

Run-time verification involves run-time monitoring of software execution in order to check run-time behaviour against rules specified in formal logic. These generally form only a partial specification of the application, pertaining only to those parts of the application for which run-time monitoring is deemed more necessary. Figure 1.1 displays an abstract view of typical run-time verification elements. The application code and the partial specifications (specified rules) are used to tie specifications to elements of the target application and modify it on the basis of the necessary observation. The verification engine checks the output of the monitor against the specifications in order to provide output of any behaviour outside specifications.

In order to identify the most pertinent issues concerning the monitoring of software in embedded systems, a review was undertaken of existing research on monitoring solutions in general, as well as for embedded systems specifically. Further to the
identification of key concerns, experimental work was carried out to investigate the feasibility of applying run-time verification to software in an embedded system. This involved the specification and implementation of a test platform to allow run-time verification of an application executing in an embedded system. Test cases were also arranged in order to ascertain the feasibility of the approach, as well as to quantify the overhead of the monitoring arrangement.

This chapter summarises the background to the research in terms of the reasons for monitoring software execution, the issues in monitoring embedded systems and the use of the run-time verification as a monitoring approach. The rationale and objectives of the work are laid out in this chapter. A short review of literature in the areas of software execution monitoring, run-time verification, monitoring embedded systems and related work is presented. Details of the research project and its novelty are given and an outline of the thesis is included. Finally, publications based on the research are listed.

1.2 BACKGROUND

Today, embedded processing systems are ubiquitous as modern technology that interacts closely with the real-world environment becomes all-pervasive. Many embedded applications have real-time software requirements that must adhere to strict execution deadlines due to the responsiveness required of interactive systems or the need for strict scheduling to allow high utilisation of limited processing and system resources. Some embedded systems such as those in control systems may have safety-critical requirements. To ensure the correct functioning of applications as the complexity grows, developers have to be extremely cautious in designing such systems. Ensuring that software applications operate satisfactorily in their final environment involves extensive and costly efforts during the software development cycle. As a result of custom hardware designs or the lack of an operating system for example, embedded systems often have a close relationship between software and hardware and so ensuring proper software operation requires co-ordination with the efforts put into testing hardware designs. Despite the evolution of improved software and hardware development practices used to increase the correctness of the system design and implementation, in operational environments there can still remain an uncertainty as to the trustworthy operation of the system.
In the IEEE Glossary of Software Engineering Terminology [1], a *failure* is described as "the inability of a system or component to perform its required functions within specified performance requirements". Characteristics of software that result in failures are variously termed *faults*, *bugs* and *errors*. Referring to [1] once more, a definition of a fault as "an incorrect step, process or data definition" is given, with a note that the term error is often used with the same meaning (this usage for error will be avoided in this paper). A definition specific to the term error, also given in [1], is of an error as "the difference between a computed, observed or measured value or condition, and the true, specified, or theoretically correct value or condition". Bug is taken to be generally synonymous with "error".

Various approaches are used in testing in order to find errors in the software, with a catalogue of testing approaches published by Myers [2]. Broad strategies include ‘black box testing’ (the software is treated as a black box, with inputs chosen and outputs examined without reference to internal operation) and ‘white box testing’ (the internal behaviour of the software is evaluated). Testing of software can be decomposed to testing of individual parts of software using a strategy such as ‘unit testing’ (objects or methods are evaluated independently), which can then include ‘integration testing’ (evaluating the combined behaviour of previously individually tested components). This form of testing can be contrasted with ‘system testing’, where the software is tested in a complete state in the operational environment.

In order to address faults remaining in software, it is possible to persistently monitor software execution in its final environment, checking the observed behaviour and performance against specified rules. The approach of monitoring and checking, which can be termed ‘run-time checking’ if the monitoring outputs are checked at run time, can be used for various purposes, commonly some form of system testing. Potential applications of run-time checking include testing, debugging, verification, logging of errors, attempts at real-time fault recovery of the system to a safe state, and maintenance/diagnosis procedures in the field.

To address the extensive design and testing efforts required in modern software development, the software engineering research community offers formal theories, based in sound semantic models that lead to verifiable designs or correct-by-construction designs. Program verification determines whether a program satisfies a specification. However, Havelund and Goldberg [3] assert that program verification is
unsolvable in general and that theorem prover solutions do not in practice scale to large software systems. There is a growing interest in the use of more ‘lightweight’ formal method solutions, where some limited specifications can be checked automatically for analysis of properties. In safety-critical embedded system design, the verification task, including proof obligations, is most important. However, it is possible to monitor some proof obligations in the testing phase, or even in the actual operating phase of a system. Emerging research that is concerned with the monitoring of specific properties during program execution is referred to as run-time verification [4]. In such run-time verification schemes a monitor can react to property violations to provide a level of fault protection.

Embedded systems can present particular challenges for monitoring. System internals are not easily observable, as many device features are incorporated deep within complex chip packages. This hinders the observation of software execution in such devices. It is not necessarily feasible to allocate much system resources or processor time to the monitoring effort, due to the limited resources of embedded systems and the real-time software requirements that must adhere to strict execution deadlines. These traits result in a need to minimise the overhead of monitoring; high overhead could compromise core system resources or affect scheduling, causing interference to the target application. However, the need to provide insight into the internal operations of an embedded chip means that some monitoring effort has to be placed within the system in order to provide the necessary observations.

This thesis attempts to explore some of the issues above by reviewing relevant literature, suggesting a monitoring approach for embedded systems, proposing and implementing an experimental monitoring arrangement for an embedded system, and arriving at some conclusions on the basis of these experiences.

1.3 RATIONALE FOR THE PROJECT

Run-time verification, already in use for general software applications, can potentially be used to enhance the reliability of embedded software applications also. It can be used for testing of embedded systems, either during the testing phase of product development, or indeed after deployment for error diagnosis or to provide alarm in the event of unsafe operation. Run-time verification is lightweight compared to formal verification as it allows the verification task to be reduced in scale to one that is
feasible for verification at run time, i.e. the verification of particularly important rules. As a monitoring approach for embedded systems, this scalability is important due to the difficulties in monitoring software execution, in that the monitoring effort is constrained by the limited system resources available for in-system monitoring and by the limited observables of the system, requiring a certain level of in-system monitoring.

Another benefit of using run-time verification on software executing in embedded systems is that the software execution behaviour is verified in its native environment. Due to the close links between software and hardware design in an embedded system (indeed sometimes ‘co-designed’), this approach potentially offers greater benefits than a pure software analysis of the development system.

An experimental monitoring apparatus was envisaged as a means to examine the feasibility of run-time verification of software executing in an embedded system. The experimental platform provides a mechanism for relaying run-time monitoring observations of an application executing on an embedded system to a run-time verification tool. Such a platform, in facilitating such an arrangement, allows attempts at run-time verification in order to gather information on the potential and limitations of providing sufficient monitoring of an embedded system for the purposes of run-time verification.

1.4 OBJECTIVES

Ensuring the reliable operation of software in an embedded system by monitoring software execution is the goal that is investigated in this thesis. The main objectives of the project described in this thesis are outlined as follows:

- Investigate the suitability of the lightweight approach of run-time verification as a monitoring solution for safety-critical embedded systems.
- Propose an outline of a monitoring approach for deeply embedded systems allowing guarantees about correct execution of software.
- Evaluate the feasibility of the proposed working solution through the development of a working prototype and develop some case studies to explore the effectiveness of the approach.
• Analyse the effectiveness of the proposed solution in terms of system resource overheads, overhead in terms of execution time and preparations needed for monitoring.

• Derive a conclusion on the feasibility of the use of formal monitors for enhancing the reliability of embedded systems.

1.5 SUBJECTS OF LITERATURE REVIEW

In order to understand the background to the research area and also to base the work of this thesis upon existing principles, a review of relevant literature was undertaken in the following areas:

• Monitoring of software execution

• Types of execution monitor

• Monitoring and verifying embedded systems

• Run-time verification

• Related work

Literature and background to these areas is examined in more detail in chapters 2 and 3 of the thesis.

Monitoring of software execution

The concept of monitoring software execution at run time has been examined from the early years of computing. Some of the historic concepts involved in monitoring computers are set out in a tutorial on the subject by Nutt [5], providing a snapshot of the concerns at that time (such as resource utilisation) and identifying three broad approaches to monitoring systems: the use of hardware apparatus to monitor (hardware monitoring), additional software (software monitoring) or a combination of both (hybrid monitoring). Plattner and Nievergelt [6] present a survey of execution monitoring that presents a short history of execution monitoring up to that time, basic concepts and requirements for execution monitoring, design considerations for monitoring languages, and implementation considerations (for example, the extent to which monitoring affects the system being monitored). Delgado et al [7] provide a later survey of monitoring tools.
Types of execution monitor

The mechanisms for monitoring by using hardware apparatus or adding additional software to the system both have advantages and disadvantages. In presenting a hybrid monitoring approach for distributed systems, Haban and Wybranietz [8] identify the strengths and weaknesses of hardware monitors (particularly with the increased complexity of typical systems by that time) and also consider software monitors. In general there are problems with more complex systems in arranging sufficient observation of system features using hardware monitoring and in using software monitoring, the normal execution of the application being monitored may be affected by the overhead of executing the monitoring code. Hybrid monitoring can be used to allow the breadth of observation possible by using software monitoring, combined with placing some of the overhead of the monitoring task on additional monitoring hardware, in order to minimise interference. For this reason, hybrid monitoring is recommended in a presentation by Svobodova [9] of an early example of the approach.

Monitoring and verifying embedded systems

Several concepts and research areas are relevant to the monitoring of embedded systems. The concept of on-chip monitoring is implemented in a hybrid monitor presented by El Shobaki [10], with the monitoring apparatus co-located on the same chip as the target system. Hardware debug interfaces and their potential use in monitoring are examined by MacNamee and Heffernan [11]. Using assertion-based verification with on-chip support is described by Peterson and Savaria [12].

Run-time verification

Run-time verification, detailed by Havelund and Goldberg [3], is a method of checking the correctness of programs. Actual execution behaviour is checked at run time against formal specifications. A common approach, as used by Havelund and Roşu [13] with the run-time verification tool ‘PathExplorer’, is to specify rules for proper execution behaviour in a form of Linear Temporal Logic (LTL). The target application may be amended to emit events when relevant monitored entities are modified, an approach used by PathExplorer and also the Monitoring and Checking (MaC) framework of Lee et al [14].
Related work

Andersson et al [15] describe a framework intended to facilitate analysis of embedded systems. The approach relies on abstraction of the software application being analysed, with the creation of a model describing the system at an appropriate level of detail for the properties to be analysed. Similarly to run-time verification, the approach restricts itself to comparison of only certain properties of the model against execution behaviour. However, for the work described, a simulation of the software application is used, in the form of an executable derived from the model being investigated. This thesis although similarly concerned with verification of a restricted number of properties, describes verification of software in its native environment.

Run-time verification of software with no burden on the system processor due to monitoring is achieved by Lu and Forin [16]. P2V allows compilation of PSL assertions to a Verilog hardware description of a monitor. This monitor is instantiated onto the same FPGA as the processor with the software being monitored. The monitor verifies the PSL assertions at run time without interfering with the execution of the software. This approach does require allocation of system resources on the FPGA to the monitor.

1.6 PROJECT BACKGROUND

This project was undertaken at the Circuits and Systems Research Centre (CSRC) at the University of Limerick, a research centre including various electronics research groups working in the automotive, mixed signal integrated circuits, intelligent power management and control sectors. This research at CSRC formed part of the University of Limerick's involvement in an inter-University research effort, the Centre for Telecommunications Value-Chain Research (CTVR) funded by Science Foundation Ireland. As a contribution to CTVR research, a post-doctoral researcher and the author's supervisor undertook work in the related project of an on-chip monitor for a SoC (System on Chip). The work that is the subject matter of this thesis was the concern of the author and was a separate project that was also part of the CTVR Cost-Optimised Reliability Theme.
1.7 PROJECT OVERVIEW

There were three main phases to the research. First, a review of literature was undertaken in the areas of execution monitoring and run-time verification. Second, a mechanism was devised to allow run-time verification of applications executing on an embedded platform. Third, test cases were carried out to determine the potential of the monitoring approach.

1.7.1 Overview of literature review

Having set general objectives for the project, similar to those presented in section 1.2, some review of literature in areas relevant to achieving those objectives was carried out. The first area researched was that of execution monitors. Some initial observations were that the main issues with which modern execution monitors are concerned are fundamental issues in execution monitoring. As such, a review of early literature on monitoring was undertaken to appreciate the historical views on these issues.

As embedded systems are the target for the research problem in this thesis, existing monitoring tools used for embedded systems were examined. In particular the area of on-chip monitoring was investigated as it potentially overcomes some of the basic problems encountered in execution monitoring and specifically the problems of monitoring in an embedded environment. On-chip monitoring was however not implemented as a monitoring approach for the experimental work due to the development efforts that would be required for a new on-chip mechanism and the limited potential for using an existing mechanism with an existing run-time verification tool.

Run-time verification approaches were described in a number of papers relating to execution monitoring. The particulars of what run-time verification is and some of the relevant subject matter such as linear temporal logic were considered. Having decided on run-time verification as an approach to attempt with the monitoring of an embedded system, consideration was given as to how to use an existing run-time verification tool.
1.7.2 Arranging a test platform

There were three main components to the test platform arranged to further several objectives in section 1.4 above: a hardware platform, a software platform, and a monitoring tool.

Hardware and software platforms

To provide a programmable hardware experimental target, the Xilinx Spartan 3 FPGA was chosen. This allows a choice of target platforms as various hardware designs can be instantiated to the test board. In order to consider the real-time aspects of many embedded systems, the research target needed to provide real-time functionality in the form of a deterministic processor. Additionally a means to analyse the execution time of instructions was necessary. The Java Optimised Processor (JOP) presented by Schoeberl [18] was chosen as a suitable target as in addition to these aspects, it allows the use of a Java-based run-time verification solution while providing an embedded environment despite the uncommon use of Java for an embedded target. As shown in figure 1.2, the JOP soft-processor core allows Java applications to be executed within an FPGA, taking the place of a Java Virtual Machine (JVM) and conventional operating system on a PC. Ordinarily Java code is compiled to bytecode and interpreted by the JVM. The JOP core converts byte code to native microcode as part of the processor pipeline.

![Diagram](https://via.placeholder.com/150)

Figure 1.2 – Comparison of Java application on (a) JVM/PC vs. (b) JOP/FPGA
JOP has for each Java byte code a known WCET (worst case execution time) that has been minimised. The choice of the JOP platform also provides real-time programming APIs in the form of a real-time thread model and a real-time scheduler. The use of Java is also beneficial for research purposes as Java byte code is easier to instrument than the fully compiled machine code of conventional (non-interpreted) programming languages. As such Java has been a target language for run-time verification for non-embedded platforms. The JOP platform is also designed to avoid the use of garbage collection, which is not suitable for real-time systems.

**Monitoring tool**

Run-time verification was chosen as the monitoring approach to be evaluated for use with embedded systems, due to the formal basis provided for guarantees about the correct execution of elements of the target application, as well as the ability to only check specific conditions. This latter aspect provides a mechanism to scale the monitoring task to the abilities of the monitoring apparatus. Java-MaC (Monitoring and Checking), presented by Lee et al [14], was chosen as a suitable run-time verification tool due to the modular architecture, straightforward means of instrumentation and user-friendly specification languages.

![Diagram of Java-MaC](image)

**Figure 1.3 – Arrangement of Java-MaC to verify an application**

As depicted in figure 1.3, Java-MaC relies on a monitoring script to link a requirements specification to monitored entities in the target application. The target
has code added to gather observations, with a ‘filter’ component to communicate with Java-MaC on the monitoring host.

**Combined JOP/Java-MaC test platform**

Figure 1.4 provides a conceptual view of the platform as was envisaged for the experimental work. Instructions are added to the target Java application executing on the JOP platform on the FPGA, to provide observations in the form of an event trace to Java-MaC on a separate host PC.

![Diagram of JOP/Java-MaC test platform](image)

Figure 1.4 – Conceptual view of platform for experimental work

**1.7.3 Test cases**

In order to satisfy as to the feasibility of the Java-MaC/JOP combination, an initial test case was arranged. The scenario of a railroad crossing was chosen, reflecting one of the examples used with Java-MaC to illustrate its features. Translating the scenario to an application for the JOP platform involved some changes, as the implementation of the Java-API for the JOP platform is more limited. Changes were also made in order to make use of some custom JOP APIs that allow the use of features such as real-time threads. An implementation of the scenario was created where the various components were modelled as separate real-time threads. This implementation was tested and demonstrated a successful use of the experimental mechanism, with Java-MaC verifying the same conditions for the JOP implementation of the scenario as for the PC implementation. The extent of modifications to the target application
necessary for monitoring were considered, as was the process for arranging the experimental mechanism. It was during this test case that the relay application approach was developed.

An example of a potential run-time verification task for an embedded system was provided by the second test case. This test case considered use of Java-MaC to verify that a set of tasks scheduled under the rate-monotonic algorithm on the JOP platform would execute as predicted by prior analysis. Worst-case execution time analysis was performed on application code in order to facilitate the development of a test application with tasks that would execute for a specified duration. The verification task was performed and Java-MaC was used to determine if task execution durations remained within expected boundaries. The results were investigated and conclusions were drawn on the limitations and potential of the monitoring mechanism.

1.8 NOVELTY OF THE WORK

This project proposes a custom monitoring arrangement to allow run-time verification of software on an embedded system. Although monitoring of software on embedded systems has been presented elsewhere, this arrangement provides a monitoring approach to specifically facilitate run-time verification of software executing on an embedded system.

On-chip facilities to support debugging are available with commercial microprocessors and are used to observe details of the internal operation of system-on-chip designs. Commercial software such as Micrium's \( \mu \)C/probe [17] allows run-time monitoring of data from an embedded system. However, the proposed on-chip hybrid monitor in this thesis is intended to support run-time verification by monitoring elements of the software application, rather than monitoring hardware resources specifically.

A novel prototype monitoring arrangement was implemented, demonstrating run-time verification of an application executing in an embedded system. The proposed monitoring arrangement was implemented as a prototype without special on-chip support in order to allow experimental work to determine the feasibility of the proposed monitoring approach.

The prototype monitoring arrangement uses an existing run-time verification tool, Java-MaC by Lee et al [14], to verify application code executing on an embedded
system. The approach of examining the execution of a Java application in an embedded environment is possible due to use of the Java Optimised Processor (JOP) soft processor design of Schoebel [18], which allows Java application code to be executed on an embedded system (a field programmable gate array, or FPGA). This is useful for evaluating challenges in performing run-time verification of an application in an embedded system, as Java code is a common target for run-time verification in research.

The provision of a test platform allows guidelines for a good monitoring approach for embedded systems to be presented based not only upon a review of literature, but also based on experiences from attempting to perform run-time verification in the test environment.

1.9 THESIS LAYOUT
Chapter 2 of this thesis introduces some of the general concepts concerning the monitoring of software execution. Some definitions are provided and the main approaches to monitoring are introduced and their benefits and drawbacks considered. The area of on-chip monitoring is also introduced.

Chapter 3 briefly introduces the subject of run-time verification, as well as considering monitoring arrangements that will allow run-time verification of software executing in embedded systems. The choice of Java-MaC as a run-time verification tool for the experimental work is explained.

Chapter 4 describes the Java-MaC run-time verification tool used for the experimental work in this thesis. The components and specification languages used by Java-MaC are described and the operation phase of Java-MaC is detailed. A deployment of Java-MaC for use with embedded systems is proposed.

Chapter 5 gives a brief introduction to the issues surrounding the use of Java in embedded systems. A soft-processor architecture, JOP (Java Optimised Processor), to execute Java code on an embedded FPGA is introduced. JOP’s architecture and features are summarised and the use of JOP in practice is described. Reasons for choosing the JOP platform for the experimental work are given.

Chapter 6 introduces a deployment of Java-MaC to indirectly verify a software application executing on JOP. The implementation is described and an initial test of
the deployment is described. Two test cases are proposed to further explore the experimental arrangement.

Chapter 7 is concerned with a railroad crossing scenario used for an example run-time verification task with Java-MaC. The scenario’s original implementation, as included as an example with Java-MaC, is described. A new implementation of the scenario, suitable for execution on JOP, is introduced. The operation of the application implementing the scenario is described and the use of Java-MaC to indirectly verify the application executing on JOP is presented. The preparations necessary for the verification task and the results are discussed. A revised deployment of Java-MaC for JOP is described.

Chapter 8 presents a second case study, this time examining use of the experimental arrangement to allow Java-MaC to verify whether a set of tasks scheduled on JOP under the Rate-Monotonic Algorithm (RMA) are indeed scheduled as predicted. The use of a worst-case execution time analysis tool to help arrange the test application and predict its scheduling is described. The implementation of the verification task and results are discussed. Analysis of the capabilities and limitations of the prototype is also presented, with some conclusions drawn based on a consideration of the effects on the viability of the proposed verification scenario.

Chapter 9 concludes the thesis by collating the results from the various experimental work and providing conclusions drawn from these results. Some recommendations for a good monitoring solution for embedded systems are presented. Additionally, recommendations on improving the experimental arrangement are made along with suggestions for future work and applications of the research.

1.10 RESEARCH PUBLICATIONS

The following research papers were written in the course of this project, with the author as primary author or co-author:

Refereed journal papers


Paper in draft form:

• C. Watterson and D. Heffernan, “Using run-time verification to monitor rate monotonic tasks in embedded systems”.

Refereed conference papers


(Also presented as a poster presentation at the conference)

Other publications


1.11 SUMMARY

This chapter has introduced the work described in this thesis. The objectives of the project have been stated, with the background to the work and rationale for it given. A brief overview of the areas examined in the literature review was presented. The thesis layout was provided and a summary of the work presented in this thesis was given as a project overview. Novel aspects of the work were described. Finally, publications authored or co-authored by the author, arising during the course of the project, have been listed.
CHAPTER 2 – MONITORING

OF SOFTWARE EXECUTION
2.1 INTRODUCTION

One of the main concerns of this thesis is determining whether run-time verification of software executing in an embedded system is a feasible proposition. As well as the areas of run-time verification and embedded systems, the monitoring of software execution is of major relevance to this task. Such monitoring provides the observations against which run-time verification compares specified rules in order to determine if the software is behaving in a prescribed manner.

This chapter briefly introduces the area of monitoring software execution, provides some relevant definitions of terms used in the area generally and throughout this thesis and summarises the main approaches to the task. Various monitoring solutions throughout the history of the research area are referenced to provide examples where different monitoring approaches have been considered. The area of on-chip monitoring is specifically considered here due to its relevance to the monitoring of embedded systems. Finally, the author proposes a novel hybrid on-chip monitoring arrangement, chosen on the basis of points arising from consideration of the existing monitoring approaches. This proposed monitoring arrangement forms the basis for the experimental platform implemented and used for experimental work presented in this thesis.

2.2 MONITORING OF SOFTWARE

Monitoring the execution of a software application is part of the task of run-time verification with which this thesis is concerned. Monitoring involves observing the execution behaviour or performance of a target application in order to gain an insight into the operation of the software. Improved software development techniques have not lead to perfect software and so there remains a need to complement these techniques with tools that help diagnose the behaviour of real software implementations. The rationale for monitoring and the fundamental concepts in monitoring are detailed in introductions to the field by Nutt [5], Plattner and Nievergelt [6], Plattner [19], Delgado et al [7] and Thane [20].

Examples of run-time checking systems are detailed by Gates et al [21] and Mok and Liu [22]. Run-time verification has been presented by Havelund and Roşu [13], Drusinsky [23] and Lee et al [14] amongst others. General debugging and 'replay
debugging' schemes have been described by Tsai et al [24] and Thane [20] for example, while performance monitoring is described by Svobodova [9] and Haban and Wybranietz [8].

In general, software execution monitoring is used within an existing testing framework, with the form that execution monitoring takes being dependent on the purpose of the monitoring. For example, software execution monitoring is used to support the software testing concept of "code coverage". This is a metric used to quantify the extent to which the target application has been tested. Tools to provide this metric rely on execution monitoring, as described by Yang et al [25] in a survey of coverage-based testing tools. Another example of the use of software execution monitoring within existing software testing frameworks is that of using software execution monitoring to provide data for test case generation, as described by Dara et al [26]. Although software execution monitoring is a component of certain testing approaches used in industry such as those mentioned above, it is presented as a continuing research challenge in a survey of software testing research by Bertolino [27].

In its basic form, monitoring refers to the task of observing the execution behaviour of a software program and performing some action based on those observations. Jain [28] provides a reference on the subject of monitoring in order to provide the necessary background for the task of performance analysis. Some similar definitions to those used by Jain are provided in the glossary at the preface of this thesis.

### 2.3 APPROACHES TO MONITORING

Published reviews of monitoring approaches have been presented in the past by Plattner and Nievergelt [6] and Delgado et al [7]. This review considers various monitoring approaches, including: the use of internal system signals with hardware probes (hardware monitoring), the addition of code to the target system's software in order to perform operations related to monitoring at certain points in the application's execution (software monitoring), the combination of both approaches (hybrid monitoring) and the use of on-chip probes (on-chip monitoring), a form of hybrid monitoring. Selected references highlight the problems and advantages that have been considered in the past relating to these approaches.

The classification of monitoring systems according to the three approaches used for probes, hardware, software and hybrid, has been used both in early surveys on
monitoring such as that of Nutt [5] and later surveys such as that by Schroeder [29].
In the presentation of specific monitoring solutions, the categorisation continues to be
used, for example, by Thane et al [30]. These three approaches to monitoring, as well
as on-chip monitoring, satisfy to different degrees the two requirements important for
monitoring embedded systems; low overhead to avoid interference and sufficient
observation of complex hardware.

2.3.1 Hardware monitors

Pure hardware monitors have existed for some time, as they include simple monitors
that perhaps simply probe one or more internal signals of the target system hardware.
As far back as 1975 Nutt [5] makes reference, in his survey of monitors, to a
monitoring system that monitors the signal sent to a processor's activity light on an
IBM System/360 mainframe. More elaborate hardware monitors have been used in
the decades since. A later example is that of Tsai et al [24], where although the
monitoring system is essentially a bus monitor (monitoring the internal system buses
of the target), it consists of a substantial amount of additional hardware, including a
separate microprocessor. A typical hardware monitoring arrangement or ‘bus monitor’
is shown in figure 2.1. As depicted, dedicated monitoring hardware is attached to the
target system (i.e. to observe information sent over the internal system bus) and the
observed data is sent by the monitoring hardware for verification external to the target
system.

![Figure 2.1 – Simplified view of a typical hardware monitor](image)

The non-intrusion benefits gained by using additional hardware for the monitoring
function were recognised from quite early on in the area of performance evaluation of
mainframe computers, with the non-intrusion feature being highlighted as a key
advantage for hardware monitors by Calingaert [31], Karush [32], and Lucas [33]. Later work, such as that considered by Tsai et al [24] also focussed on the potential for minimising intrusion due to the execution of monitoring code by the target system, by using additional hardware for monitoring.

The difficulties of hardware monitoring have been emphasised for some time. In 1981 Gallo and Wilder [34] noted the problem of newer systems offering less physical probe points and suggested at the time that hardware monitors were becoming obsolete. Haban and Wybranietz [8] also note the inapplicability of hardware monitors with respect to more complex systems. This view of traditional hardware monitoring approaches is repeated in later work suggesting alternative monitoring approaches. Calvez and Pasquier [35] suggest a hybrid monitoring approach and an on-chip monitoring approach is suggested by El Shobaki and Lindh [36].

With the development of system-on-chip (SoC) devices, on-chip caching and more complicated processing and memory architectures have further reduced the visibility of program execution information system to external parts of the monitoring system that are separate from the target system (i.e. connected to the target only by hardware probes). Device-level monitoring remains a target area for hardware monitors, such as the bus monitor described by Pellizzoni et al [37], used to monitor peripheral devices and verify them against specifications. The monitor described, BusMOP, is automatically synthesised from the specifications to create a standalone FPGA-based monitor.

2.3.2 Software monitors

With software monitoring, there are different ways that the target system can be modified through the addition of software for monitoring. Figure 2.2 shows how code to perform observations on the target application execution can be added to the target application code itself (instrumentation), or can take the form of a modification to the operating system of the target system, or the monitor can be a separate process. The software monitoring approach has also long been used; in 1975 Nutt [5] even suggests that it is an older approach than that of hardware monitoring. Older surveys by Calingaert [31] and Karush [32] on the subject of performance evaluation of mainframe computers make reference to running extra monitoring processes to trace
execution steps, thus building a history of process execution behaviour for the system (job accounting).

Figure 2.2 – Abstract view showing possible locations of monitoring software within a target system

One of the main drawbacks of software monitoring, as noted by Nutt, Calingaert and Karush, is the potential side-effects from having the target system execute the additional software. Interference with the target system’s normal operation may arise if the execution of the target software is delayed due to the execution of monitoring code. However, it has been suggested by Lucas [33] even in examining older software monitoring tools, such as that of Stanley [38], that the effect of software monitoring can be minimised. Indeed software monitoring has been used for real-time and distributed systems as noted by Tokuda et al [39], where tolerance of intrusion is particularly low. Tokuda et al suggest an approach of building the monitoring apparatus permanently into the target system (also suggested by Svobodova [40]). The concept is to design a system with software probes permanently included, adjusting scheduling for the overhead of monitoring and allocating greater resources for the system as a whole. There are drawbacks to this approach in that more resources are required by the target system and overall system performance is impaired. In the area of embedded systems, where designs need to be as cost-effective as possible, the option of leaving software probes in the system is not particularly attractive, unless there are other cost/benefit considerations such as high-reliability requirements, where a cost premium can be justified. Leaving software probes in the system may be a
useful approach if the software is to be monitored in an ongoing basis in the normal operational environment.

The advantage to monitoring a system from a software monitor within the target is recognised by Deniston [41], in that such monitors have access to extensive information about the operation of a complex system, in contrast to the limited information available externally to hardware probes. Indeed, this flexibility is discussed by Rota and de Almeida [42] in more recent run-time monitoring research, where they explain their choice of the software monitoring approach, as compared to hardware or hybrid monitoring.

Applications of monitoring for run-time verification by Havelund and Roșu [13], Drusinsky [23] and Lee et al [14] have mostly used the software monitoring approach. Many of the monitoring tools catalogued by Delgado et al [7] use software monitoring.

2.3.3 Hybrid monitors

In general, hybrid monitoring refers to approaches that use a combination of additional software and hardware to monitor a target system, relying on the advantages of each approach and at the same time attempting to mitigate their disadvantages. Hybrid monitors can be mainly software-based (in which case they may even be termed software monitors) or mainly hardware-based (in which case they may be termed hardware monitors) but in general the term “hybrid monitor” is used to refer to monitoring approaches between the two extremes.

By relying on aspects of hardware monitoring, such as making observations using physical interfaces to the target system, hybrid monitoring is less likely to interfere with the behaviour of the target system. Since more complex systems do not facilitate sufficient observation of execution behaviour using such physical probes, hybrid monitoring also relies on additional monitoring code that is executed by the target system. Although the overhead of executing this code can potentially affect the behaviour and/or performance of the target system, the effects can be lessened compared to software monitoring due to the use of physical probes (i.e. less monitoring code is required to sufficiently observe the system).

Hybrid monitoring was identified as distinct from software or hardware monitoring by Svobodova [9], although earlier monitors (particularly those relying on monitoring
hardware) probably can be categorised as hybrid monitors rather than purely hardware monitors. An abstract view of a hybrid monitoring arrangement is depicted in figure 2.3. In the arrangement shown, the source code of the target process (the process being monitored) is instrumented, i.e. additional code is added to it to emit events when process features (e.g. variable values) being monitored are updated. These events are sent to the dedicated monitoring hardware, which analyses the events and checks them against the provided monitoring rules. The event analyser provides the dedicated monitor output.

![Figure 2.3 – Abstract view of a hybrid monitoring arrangement](image)

Various hybrid monitoring systems have been developed; for distributed systems by Haban and Wybranietz [8] and Hofmann et al [43], for real-time systems by Harelick and Stoyen [44] and for embedded systems by Calvez and Pasquier [35]. Hybrid monitoring has also been used for performance monitoring as discussed by El Shobaki and Lindh [36].

### 2.4 ON-CHIP MONITORS

The decreasing visibility of system operation to external observation in affecting hardware monitoring, also affects the hardware aspects of hybrid monitoring. On-chip monitoring, as described in the monitoring solution proposed by El Shobaki [10], provides a solution for continuing to facilitate hardware components of hybrid monitoring. On-chip monitoring increases the visibility of the target system operation for external parts of a monitoring system by adding hardware monitoring components into the target system-on-chip itself. Non-intrusive monitoring is facilitated through
this use of additional on-chip hardware; the overhead of communicating observations to external checking mechanisms can be placed upon a small amount of dedicated hardware within the target system. This can be preferable to adding to the main processing load in a tightly scheduled real-time environment. The integration of on-chip tools is the subject of work by Walters et al [45]. Arora et al [46] describe use of on-chip monitoring facilities to ensure the security integrity of an application executing on an embedded system.

2.4.2 On-chip monitoring research areas

Increasing hardware complexity has lead to the development of on-chip debug facilities to provide some of the functionality previously offered by in-circuit emulators. In-circuit emulators act in place of the normal processor in a circuit, offering extra probe outputs and debug capabilities compared to the standard processor. With system-on-chip designs, it is necessary to provide extra debug hardware incorporated onto the chip in order to provide access to the target processor for similar debugging. A debug port such as that of the IEEE1149.1 interface (JTAG) is used to access the on-chip debug components. Hopkins and McDonald-Maier [47] provide a review such on-chip debug facilities.

Various on-chip debug capabilities are incorporated into recent microprocessor and system-on-chip products. EmbeddedICE [48] developed by ARM, provides debug facilities for ARM’s RISC processor architecture, allowing the use of breakpoints and the examination of registers. The JTAG interface is used to communicate with the EmbeddedICE core. Embedded Trace Macrocell (ETM) [49] is another ARM technology providing more extensive debug features with a real-time trace. This trace can be output to a special trace port, or stored on-chip for slower read-out via JTAG. EJTAG [50] provides MIPS microprocessors and processor cores in system-on-chip designs with debug instructions and also uses the JTAG interface. The Background Debug Mode (BDM) [51] used with Freescale (previously Motorola) products provides debug features allowing breakpoints and the facility to read CPU state and memory.

In addition to built-in, on-chip debugging capabilities incorporated at design time to support observation of internal system behaviour, there is a need for hardware interfaces allowing the observations to be communicated to external monitors. Some
developments in this area are detailed by MacNamee and Heffernan in [52]; for example, the Nexus 5001 standard specifies a common debug interface for embedded systems. The widespread use of the IEEE1149.1 interface (JTAG) for various on-chip debuggers is also examined. MacNamee and Heffernan examine in [53] the possible uses of built-in on-chip debugging interfaces for requirements-based monitors.

The use of FPGA devices and customisable system-on-chip (SoC) designs offers new possibilities for hardware facilities supporting monitoring systems to be included at the design stage. The possibility of having an on-chip hardware-based monitor is examined in such a context by MacNamee and Heffernan [11]. Drechsler [54] describes a method of synthesising on-chip hardware to check specification properties. In the context of monitoring a target system for verification, an approach has been proposed by El Shobaki and Lindh [36] of including a monitoring IP block on SoC devices to support run-time monitoring.

Verification approaches at a circuit design level (this is a level below the consideration of execution behaviour) can also be regarded as relevant to the monitoring of software execution in embedded systems, as circuit level properties have even more stringent requirements for monitoring, such as interface bandwidth, probing, complexity, speed, etc. As complex hardware circuits, such as SoC devices, grow in complexity the need for automated design verification solutions is becoming more apparent.

Although assertion-based verification methods have been used in software engineering for some time, the concept has more recently been applied to hardware design verification. The emergence of assertion languages, such as PSL (Property Specification Language) [55], has accelerated the interest in assertion-based verification solutions for hardware-based systems. PSL is the IEEE Property Specification Language, which is defined in the IEEE 1850-2005 standard [55]. PSL provides a formal notation for the specification of electronic circuit behaviour, compatible with popular hardware design languages, such as VHDL, Verilog, SystemC and SystemVerilog. Thus, an integrated specification and verification flow can be realised for multi-language hardware-based circuit designs.

Currently, there are design tools available to support assertion-based verification during the verification and simulation stages of the SoC design process. Such
solutions use offline analysis of trace debug data, provided by assertion-based monitoring techniques. However, a run-time verification solution can be beneficial for online analysis. One of the first approaches to such a run-time verification solution at the SoC level is presented by Peterson and Savaria [12], where they describe a debug environment for complex hardware systems, providing real-time verification of prototype systems that are running at full clock speeds. Peterson et al. refer to their solution as an ‘Assertion-based Run-time Debugger’ (ABRD). A minimally invasive hardware on-chip probe interface collects relevant trace data and transfers this to an external board that incorporates an FPGA based assertion checker utilising a real-time assertion-based verification approach to validate behaviour. For a given product design, ABRD files are generated with the aid of a special compiler that includes the specified assertions expressed in an assertion language such as PSL. The properties, which are expressed in the RTL (Register Transfer Language) code at the product design time, can be used in the on-line debug process. Figure 2.4 shows the scheme.

Figure 2.4 – Assertion-based run-time debugger

The limited bandwidth of the probe interface and the loading effects at the SoC device interface are reported by the developers to highlight some limitations to the solution. However, the concept of extracting trace data in real time and applying assertion-based verification is a very important area of development for SoC hardware-level run-time verification. Emerging higher-density FPGA devices will more easily accommodate the on-chip interface along with the assertion checker on the SoC chip and thus reduce some of the reported limitations. These concepts are discussed by Fryer [56] in an embedded software environment where bandwidth and probing are less restricted than the SoC devices considered by Peterson and Savaria.
Although this thesis is concerned with verifying software in an embedded system rather than verifying hardware designs, the concept illustrated by Peterson and Savaria of using an on-chip probe with an external system performing verification could perhaps be applied to software verification also.

2.5 SUGGESTED MONITORING APPROACH

In this section, a hybrid on-chip monitoring approach suitable for use with embedded systems is proposed. Reasons for choosing this approach included:

- Disadvantages or infeasibility of other approaches
- Ability to sufficiently observe the internal operation of the target system
- Potential for minimising interference with the target system
- Facilitation of various monitoring tasks, such as run-time verification

These concerns are discussed in the following sections, followed by a presentation of the suggested approach.

Disadvantages and infeasibility of other approaches

From reviewing existing literature on the subject of monitoring software execution, it is apparent that a hardware monitoring approach, if it were feasible, would be most suitable for minimising interference. However such an approach may be rendered difficult for embedded systems as system internals are not easily observable, since many device features are incorporated deep within complex chip packages. An approach of solely using additional software for monitoring would be unsuitable for embedded systems due to the interference they may present to the target system due to the system resources or processor time required in executing monitoring code.

However, some degree of additional software for monitoring is indeed required to provide sufficient insight into internal operations of an embedded system for the purposes of observing run-time behaviour of software executing on it. Therefore the hybrid monitoring approach should be considered for monitoring of an embedded system, with emphasis on utilising hardware support for monitoring.
**Sufficiently observing the target system**

In order to facilitate the use of monitoring hardware by a hybrid monitoring approach, it is necessary that such hardware has access to internal components of the target system (e.g. in order to read register contents or memory locations). Due to the inclusion of many system features in a single chip package on an embedded system, any monitoring hardware may not be able to sufficiently interact with the target system software, even if it has been instrumented. Therefore the use of an on-chip monitoring approach is recommended.

**Minimising the potential interference with the target system**

The literature on the subject of on-chip monitoring offers some means for supplementing any monitoring software with hardware that performs part of the monitoring task. One of the aims of including on-chip hardware support for monitoring is to minimise the degree to which additional code has to be added to the target application for the purposes of monitoring, in order that the target application is less affected by the monitoring task.

However, the inclusion of on-chip monitoring hardware, although helping to avoid interference with the operation of the target application, adds a burden to the development of the target system hardware add to the cost of the system, as limited hardware resources must be allocated to the monitoring hardware. Therefore, it is proposed that any monitoring hardware included on the target system should be solely for the purposes of dispatching observations (made by the additional code added to the target application) to an external monitoring apparatus.

**Facilitating various monitoring tasks**

By arranging the monitoring elements such that observations are made by code added to the target application and then dispatched by an on-chip event dispatcher, one can facilitate various monitoring tasks. Any monitoring task that requires a trace, a series of events, can in theory rely on the output of the on-chip event dispatcher. Therefore it is proposed that the elements of the monitoring apparatus that are specific to the monitoring task (for example, run-time verification) remain external to the target system.
2.5.1 Proposed monitoring arrangement

A theoretical hybrid monitoring arrangement that uses an on-chip monitoring core in such as fashion is shown in figure 2.5. This hybrid monitoring arrangement was considered at the early stages of the work presented in this thesis.

Figure 2.5 – Theoretical on-chip (hybrid) monitoring arrangement

In this arrangement, the target process may be instrumented to emit events (for example assignments to variables). This monitoring code does not need to include complicated communication code, as a dedicated monitoring core on the chip can instead perform the task of dispatching events from the chip to the external observer. By limiting the monitoring code to simple output of events to on-chip monitoring hardware, the overhead of executing the extra code is limited, reducing the scope for interference with the target system. Similarly by limiting the scope of the on-chip monitoring hardware to the simple relay of this event information, the cost of accommodating the mechanism is reduced. External hardware observes the event information and can perform whatever tasks are required (e.g. logging, verification, debugging).

The actual monitoring arrangement described in chapter 6 of the thesis differs somewhat from this scheme in order to allow a basic prototype for testing. Although the code added to the target system does not contain any complicated communication code, it relies on the built in mechanisms for printed output to the serial port (an API
print line implementation using on-chip hardware for serial communication), rather than using a separate monitoring ‘event dispatcher’ element on-chip. The role of the observer shown in figure 2.5 is fulfilled by a host PC system connected to the target embedded system by the serial port. However, the experimental approach is intended to illustrate the potential for using the arrangement described in this section, perhaps with next-generation debug interfaces rather than the serial port used for the experimental deployment.

2.6 SUMMARY

A brief introduction to the monitoring of software execution has been provided. Three main monitoring approaches have been presented, with references to particular examples and arguments. The area of on-chip monitoring has been considered, with various areas of research and existing research projects referenced. Finally a proposed hybrid on-chip monitoring approach has been introduced. The reasons for choosing this approach have been given and explained in the context of existing monitoring approaches. The details of the monitoring arrangement and its relation to the approach implemented during the experimental work detailed later in this thesis have been described.
CHAPTER 3 – RUN-TIME VERIFICATION

OF EMBEDDED SYSTEMS
3.1 INTRODUCTION

The subject of this chapter is the concept of run-time verification, considered in this thesis as a potential monitoring approach for use with embedded systems. A review of literature provides an overview of the concept and comprises examples of work that various groups have undertaken in the area of run-time verification. Existing run-time verification tools that employ methods based on various temporal logics and specification languages are considered in terms of how they interface with the target system. Some relevant issues are highlighted for a practical application of run-time verification to an application in an embedded system. Finally, the choice of an existing run-time verification tool for incorporation into a test monitoring arrangement is explained.

3.2 RUN-TIME VERIFICATION

Run-time verification, detailed by Havelund and Goldberg [3], is a method of checking the correctness of software applications and is described by Havelund and Roşu [57] and Lee et al [14] as bridging the gap between formal verification and testing. Run-time verification is combined with test case generation by Artho et al [58], applied to NASA rover controller software written in C++. With run-time verification, detailed observations of the target system execution behaviour are checked at run time against properties that specify the intended system behaviour. These properties are often derived from the target application's specification documents, or in theory the properties to be monitored could form the entire specification. In either case, the properties are expressed formally. For example, Linear-time Temporal Logic (LTL) [59] is often used to formally state properties or models of target applications for monitoring and verification. LTL is a branch of modal logic that considers the notions of time and order. LTL is similar to propositional logic but includes temporal operators that can consider the values of propositions in the future or the past. These allow conditions such as ‘sometime in the future this must hold true’ and ‘always in the past this must hold true’.

A means must also be provided to automatically link low-level observations of program execution behaviour (which can be emitted as events from the suitably instrumented target system) to the relevant monitored properties. The Monitoring and Checking (MaC) framework described by Lee et al [14] and by Kim et al [60] uses
two languages: MEDL (Meta Event Definition Language) for specifications and PEDL (Primitive Event Definition Language) to map program level events to specification-level events.

Commercial use of run-time verification solutions is not as yet common, but solutions are offered by Time-Rover Inc. based on the Temporal Rover research of Drusinsky [23].

3.2.1 Expressing monitored properties using LTL

Havelund and Roșu [13] present a run-time verification tool named PathExplorer (PAX) that uses LTL. Rules expressing intended program behaviour based on program actions at run time (e.g. values assigned to variables) are monitored and verified by PathExplorer. A synthesis algorithm is used to generate a verification algorithm from finite trace linear past-time temporal logic (ptLTL) formulae. This verification algorithm checks that a finite sequence of events satisfies the formulae. The PAX architecture relies on instrumentation of the target application with monitoring code; with support for different implementations depending on the target software language (the need to support a combined C/Java environment is highlighted). In the case of a Java implementation, a tool named JTrek [61] is used to instrument byte code. The monitoring code emits events (changes in execution behaviour), which are dispatched to individual execution threads (that can execute on separate hardware) that check particular rules.

Java PathExplorer, the Java implantation of this run-time verification tool, makes use of Maude's rewriting logic (Clavel et al [62]) and provides future and past time linear temporal logics as detailed by Havelund and Roșu [63] [13] [64]. Specifications are used to instrument target applications with code to emit events and also to generate observers. These observers are used in monitoring software execution to allow error-pattern analysis (checking for programming errors independent of specifications), as well as checking against the specification formulae. The Eraser algorithm of Savage et al [65] has been implemented within the JPaX framework to provide data race error-pattern analysis.

A conclusion that LTL alone may not be the most appropriate formalism for logic-based monitoring in Java PathExplorer has been made by Havelund and Roșu [64]. LTL has however been extended for more specific purposes. Metric Temporal Logic
(MTL) as used in Temporal Rover by Drusinsky [23] allows the application of real-time constraints to the LTL operators, specifying duration bounds. Temporal Rover monitors program execution with respect to rules expressed using LTL and MTL and run-time verification is performed using executable alternating automata. An extended LTL/MTL logic combined with times series constraints (LTLD) is also described by Drusinsky [66], for use with Temporal Rover and a remote version, DB Rover. LTLD allows the representation of properties such as stability, monotonicity, temporal min/max and temporal average values over time. Monitoring and Checking (MaC) by Lee et al [14] uses a temporal logic that Havelund and Roșu [13] suggest can be classed as a form of interval logic to specify properties for monitoring. The languages used to express the logic have been expanded by Sammapun et al [67] to allow the use of regular expressions in event definitions, allowing simultaneous events to be more easily described.

The concept that different specific logic formalisms may be more appropriate than others, depending on the application, was a rationale for the development of EAGLE, described by Barringer et al [68]. In addition to allowing the use of various logics, EAGLE allows new logics to be defined for use in run-time verification. These logics can be used in executable rules written as Java code. HAWK is a temporal logic introduced by d'Amorim and Havelund [69] that extends the EAGLE framework. EAGLE requires the user to create a projection of the actual program state that is being monitored. By contrast, using HAWK one only need refer using specification formulae to low-level events emitted by the instrumented target application.

3.2.2 Methods of observation to facilitate run-time verification

In order to observe the execution behaviour in enough detail to facilitate the checking of (possibly complex) properties expressed in formal logic, most run-time verification tools rely primarily on the addition of extra monitoring software to the target application. The Temporal Rover system of Drusinsky [23] relies on assertions placed as comments in the target application code (Java, C, C++ or HDL). Code to perform run-time monitoring and verification is generated and incorporated as part of the target application at compile time. The PathExplorer architecture of Havelund and Roșu [13] also relies on instrumentation of the target application code. In the case of PathExplorer, the additional code is used to emit events when predicates across
program execution behaviour (e.g. values of variables) are updated. These events are received by a separate monitoring process that dispatches relevant events to separate verification processes.

Some run-time verification solutions make use of separate software components, some of which can be run on separate hardware (additional to the target system). Havelund and Roșu [57] mention that the ‘observer’ component of Java PathExplorer can be located on a separate computer from that of the target application and facilitating this, events are transmitted from the target system over a socket. The Monitoring and Checking (MaC) framework described by Lee et al [14] adopts a similar ‘event dispatch’ approach, also relying on instrumentation of the target application in order to dispatch event information to an observer. This observer recognises events and interfaces with a separate runtime checker component that performs the actual verification task.

3.3 FACILITATING RUN-TIME VERIFICATION OF SOFTWARE IN EMBEDDED SYSTEMS

There are certain run-time verification approaches that are more suitable for adaptation for use in an embedded environment. A more modular run-time verification approach is more suitable, as the verification engine and elements such as ‘event recognition’ (as in MaC) can be arranged on separate monitoring systems external to the target system. This means that the additional overhead incurred by the target system due to execution of monitoring code can be reduced (an important goal in minimising the possibility that normal behaviour of the target system will be affected by the monitoring). Run-time verification can be scaled as noted by Havelund and Roșu [13]; a particular critical set of properties is monitored rather than a set of properties forming a complete specification of intended system behaviour. Approaches such as those used by PathExplorer and MaC are suitable for this; selected rules rather than a comprehensive set can be monitored, thus reducing the monitoring overhead on the target system.

3.3.1 Observing the target system

Embedded systems can present particular challenges for monitoring. System internals are not easily observable, as many device features are incorporated deep within complex chip packages. This hinders the observation of software execution in such
devices. Many embedded systems have limited resources or have real-time software requirements that must adhere to strict execution deadlines. These traits result in a need to minimise the overhead of monitoring; high overhead could utilise an unaffordable proportion of limited core system resources or affect scheduling, causing interference to the target application.

The nature of embedded systems hinders simple probing of internal hardware mechanisms. Paradoxically, the constraints on resource usage hinder the use of software probes that have insight into the internal operation of the system. Such probes usually have high overhead that is likely to lead to interference with the normal operation of the target system. The importance of non-intrusive monitors and the problems with intrusive software monitoring systems are discussed by Harelick and Stoyen [44], who present a less intrusive software monitoring system. Fryer [70] examines non-intrusive and ‘low’ intrusive monitoring in the context of more complex systems and in particular embedded systems.

Nevertheless, sufficient probing of the target system is required to allow run-time verification. While the discussion of monitoring approaches in chapter 2 suggests that hardware or hybrid approaches to monitoring are preferable in terms of minimising overhead, even with non-embedded systems run-time verification approaches usually rely on some degree of software instrumentation. The quantity of code to be added to the target system can be reduced by a modular approach such as that described by Kim et al [71] for Java-MaC.

3.4 SUITABILITY OF JAVA-MAC FOR EXPERIMENTAL WORK

In choosing an existing run-time verification tool for implementing the experimental monitoring approach of chapter 2, some considerations were present. As creating or substantially altering a run-time verification scheme was not the main research focus, the overall consideration was the ease with which any run-time verification tool could be incorporated in an implementation of the monitoring approach outlined in chapter 2. Two specific considerations were a simple method of expressing verification rules and monitored properties and an adaptable architecture for the run-time verification scheme (in order that it could be incorporated into an embedded monitoring approach).
Java-MaC was chosen as a suitable run-time verification tool for implementing the experimental monitoring approach. The languages used for expressing specifications and monitored entities using Java-MaC have a formal basis, as described by Kim et al [60], but are nevertheless straightforward to use. Java-MaC is modular, not only allowing the verification engine to be run on a host system (in order not to place an execution overhead on the target system or use its resources) but also ensuring that only the front-end monitoring apparatus of the Java-MaC tool will need to be considering in arranging a deployment similar to that of the monitoring approach described in chapter 2.

Although Java-MaC adds a dedicated ‘filter’ thread to the target application that uses TCP sockets to communicate to the rest of the system, as well as function calls in the target application to the filter thread, an alternative approach relying more on hardware features of the target system and less on additional monitoring software, is described in chapter 6 of this thesis.

The monitoring approach suggested in this thesis is to leave ‘software probes’ in the target application at monitoring points, as described in section 2.3.2 of the preceding chapter. However the monitoring approach is not exclusively reliant on additional software, as the built-in hardware mechanisms of the system are used to communicate these observations to the host system on which Java-MaC executes. Allowing Java-MaC to indirectly monitor the target application by monitoring an application on the host system that receives these observations is the subject of the chapter 6 of this thesis.

3.5 SUMMARY

This chapter has given a brief introduction to the concept of run-time verification. Some existing tools and research projects have been referenced with the purpose of describing how run-time verification is arranged in general. The facilitation of run-time verification in an embedded system has been considered. Some of the concerns of monitoring embedded systems in general have been highlighted and the implications of the monitoring needed for run-time verification have been suggested. Some brief introduction has been provided as to how these concerns relate to the experimental approach described later in this thesis.
CHAPTER 4 – JAVA-MAC: A VERSATILE

RUN-TIME VERIFICATION SYSTEM
4.1 INTRODUCTION

The Java-MaC run-time verification system is the focus of this chapter. Java-MaC and its component elements are briefly introduced and details are provided of the specification languages used by Java-MaC. The run-time configuration for using Java-MaC to verify the execution behaviour of an application is explained. Criteria for the deployment of Java-MaC to verify an application executing on an embedded system are considered. Finally, the author proposes an arrangement for deploying Java-MaC to verify an application executing on an embedded system based on these considerations. The relationship is explained between this proposed arrangement and the monitoring arrangement proposed in chapter 2.

4.2 JAVA-MAC INTRODUCTION

Java-MaC was developed at the University of Pennsylvania by the Real-Time Systems Group and is a framework for testing that uses the technique of run-time verification in order to provide some assurances at run time about the correctness of a software application. Java-MaC comprises several components and uses two specification languages in its operation, as described by Kim et al [60]. As is the case for run-time verification in general, Java-MaC is concerned with checking that a current execution of a software application is correct at run time with respect to specified desired behaviour. Execution behaviour is monitored by instrumenting the byte code of the target application at points where updates to relevant application components occur. At run time, the observations produced by the additional code are used to check aspects of the current execution behaviour against properties expressing desired execution behaviour.

4.2.1 MaC components

There are three main components to Java-MaC: the filter, event recogniser and run-time checker. These interconnecting modules are shown in figure 4.1 in their typical Java-MaC configuration. The filter performs the role of generating observations within the target application and relaying the state of monitored entities to the event recogniser. The filter comprises two parts: a) a distinct thread of execution on the target system (label 2 in figure 4.1) that emits events from the target application via a TCP socket to the event recognizer (label 3); and b) function calls to the filter thread
added to the target application's own execution thread at points where monitored entities occur (label 1).

Figure 4.1 – Typical Java-MaC arrangement.

The event recogniser tracks the state of monitored entities and based on updates from the filter, determines if events/conditions have occurred/changed. The run-time checker, based on updates received from the event recogniser across a second TCP socket, notifies if safety properties have been broken or if alarms need to be raised.

4.3 MAC LANGUAGES

Java-MaC uses two languages, which have a formal basis as detailed by Kim et al [72], for specification of these properties. The following section is a summary of the overview of the MaC languages provided by Kim et al. The two languages are termed PEDL (primitive event definition language) and MEDL (meta event definition language). Conditions and events that occur at run time are linked to application execution behaviour using PEDL and these form the basis of more abstract events and conditions written in MEDL.

4.3.1 Events and conditions

Common to both PEDL and MEDL are the concepts of events and conditions. Events are instantaneous changes in state at run time, while conditions are particular states that hold for a particular duration. Observations are made by the filter component of Java-MaC at points in the target application execution where events concerning monitored entities occur. An example of an event would be the beginning or end of a method that is being monitored. Events are associated with a specific instant in time. Conditions apply between events, for example, the value of a monitored variable remaining at a certain value would constitute a condition, with any change to that
value constituting an event that might end the condition. Conditions apply over an interval of time.

### 4.3.2 Undefined variables

The logic used for Java-MaC allows reasoning about undefined variables. The monitoring of any variable is complicated by the fact that the variable may not have an associated value at certain points in time. For example, if at run time the variable has not yet been initialised, or at a later stage the variable is no longer in the current scope of the application’s execution, then that variable will be undefined. However, at these points in the application execution, the verification engine may nevertheless need to reason about these undefined variables. To allow reasoning about undefined variables, the logic supporting Java-MaC is three-valued, allowing the truth value of conditions to be true, false, or undefined.

### 4.3.3 Logic for events and conditions

Two specification languages are used by Java-MaC, a PEDL file and an MEDL file. Common to both scripts are the concepts of events and conditions. Conditions are particular logical states and events are instantaneous changes in state that can be used to update the Java-MaC verification task. Expressions using Boolean and temporal logic can be composed to define the events and conditions.

\[
\langle C \rangle ::= e | \text{defined}(\langle C \rangle) \upharpoonright \langle (E), (E) \rangle \upharpoonright ! (C) \\
\text{\quad} \upharpoonright \langle C \rangle \&\& \langle C \rangle \upharpoonright \langle C \rangle \| (C) \upharpoonright \langle C \rangle \Rightarrow (C) \\
\langle E \rangle ::= e | \text{start}(\langle C \rangle) \upharpoonright \text{end}(\langle C \rangle) \upharpoonright \langle (E), \&\& (E) \rangle \\
\text{\quad} \upharpoonright \langle (E) \| (E) \rangle \upharpoonright \langle (E) \rangle \text{when } \langle C \rangle
\]

Figure 4.2 – Syntax for events and conditions

The logic used for events and conditions is summarised in this section, based on the formal presentation by Kim et al [72]. There are two sorts to the logic used for Java-MaC: conditions and events. As shown in figure 4.2 (from Kim et al [72]), primitive conditions can be: a primitive condition, whether or not a condition is defined, an interval between two events, negation of a condition, one condition and another, one condition or another, and one condition implying another. Primitive events can be: a primitive event, the start of a condition, the end of a condition, one event and another, one event or another, and one event when a condition is true.
In order to define the logic for events and conditions, the first step is to define what is meant by a condition; its denotation ($D^t_M(c_k)$). This is defined in Figure 4.3, from Kim et al [72] (in which it is explained formally). $L_C(s_i,c_k)$ refers to the total function from states and conditions to true, false or undefined. Each of the primitive conditions listed in figure 4.2 are defined. First the denotation of a primitive condition itself is given. The other conditions can be explained as follows:

- **defined($c$)** is true if a condition $c$ is defined, or false otherwise.
- **[$e_1,e_2$]** is true if $e_1$ has occurred and $e_2$ has not yet occurred, otherwise it is false.
- **negation, disjunction, conjunction, and implication** are derived from the definition of a primitive condition and can be true or false according to classical interpretation, but can also take the value of undefined if certain primitive conditions on which they depend are themselves undefined.

**Figure 4.3 – Denotation of conditions**
This definition of the denotation of a condition, allows $M, t \models c$ to be defined (that is, condition $c$ being true in model $M$ at time $t$) as in figure 4.4.

<table>
<thead>
<tr>
<th>$M, t \vdash c$</th>
<th>iff</th>
<th>$D_M'(c) = true$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M, t \models e_k$ ($e_k$ primitive)</td>
<td>iff</td>
<td>there exists state $s_i$ such that $\tau(s_i) = t$ and $L_E(s_i, e_k)$ is defined.</td>
</tr>
<tr>
<td>$M, t \models \text{start}(c)$</td>
<td>iff</td>
<td>$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models c$ and if $i &gt; 0$ then $M, \tau(s_{i-1}) \not\models c$; i.e., start($c$) occurs when condition $c$ changes from false or undefined, to true; before state $s_0$ conditions are assumed to be undefined.</td>
</tr>
<tr>
<td>$M, t \models \text{end}(c)$</td>
<td>iff</td>
<td>$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models \neg c$ and if $i &gt; 0$ then $M, \tau(s_{i-1}) \not\models \neg c$; i.e., end($c$) occurs when condition $c$ changes from true or undefined, to false; before state $s_0$ conditions are assumed to be undefined.</td>
</tr>
<tr>
<td>$M, t \models e_1 \parallel e_2$</td>
<td>iff</td>
<td>$M, t \models e_1$ or $M, t \models e_2$.</td>
</tr>
<tr>
<td>$M, t \models e_1 &amp;&amp; e_2$</td>
<td>iff</td>
<td>$M, t \models e_1$ and $M, t \models e_2$.</td>
</tr>
<tr>
<td>$M, t \models e$ when $c$</td>
<td>iff</td>
<td>$M, t \models e$ and $M, t \models c$; i.e., event $e$ occurs when condition $c$ is true.</td>
</tr>
</tbody>
</table>

Figure 4.4 – Semantics of events and conditions

The semantics of events and conditions is specified in figure 4.4, from Kim et al [72]. As with the denotation of conditions, each event in figure 4.2 is specified. Events that occur at the start and end of a condition $c$ are start($c$) and end($c$) respectively.

### 4.3.4 PEDL and MEDL

Entities in the target application that need to be monitored are specified in the PEDL specification. These entities are either variables or methods. As such, the PEDL specification is language and implementation specific, as elements of the application code are used directly in expressions defining primitive events and conditions. Only the current state of the application execution is reasoned about using PEDL, in order to keep the process of event recognition simple.
The MEDL specification contains the safety requirements used during verification. Primitive events and conditions are imported from the PEDL specification, in order that the MEDL specification is not implementation-specific but rather comprises a more abstract verification task. Excerpts from the formal specifications of the target application might form the basis for what to include in the MEDL specification. The structure of the MEDL specification is summarised in section 4.3.6.

4.3.5 PEDL for Java

There are two types of entity in a target Java application that are monitored by Java-MaC: variables and methods. Observations of changes to these monitored entities form the basis for the events and conditions that Java-MaC uses to perform run-time verification. In the PEDL specification, primitive events and conditions are constructed as expressions using these monitored entities.

```plaintext
MonScr <spec_name>
    /* Export section */
    export event <e>;
    export condition <c>;

    /* Overhead reduction section */
    [timestamp;]
    [valueabstract;]
    [deltaabstract;]
    [multithread;]

    /* Monitored entity declaration section */
    monobj <var>;
    monmeth <meth>;

    /* Event and cond definition */
    event <e> = ...;
    condition <c> = ...;
End
```

Figure 4.5 – PEDL script example

The PEDL file is structured in an ordered fashion as shown in figure 4.5 (from Kim et al [72]). First the name of the script is declared, followed by a declaration of events and conditions to be exported from the event recognizer to the run-time checker. There follows an overhead reduction section, where techniques to minimise the overhead of the monitoring code can be switched on if required. Each monitored entity is then declared using `monobj` for variables and `monmeth` for methods. The type of the variable or return type of the method is specified, followed by the application code name for the entity with its full classpath.
The definition of events and conditions follows this, with each defined using the keyword *event* or *condition* with a user-specified identifier for that event or condition, followed by an equality sign and an expression for that event or condition. These expressions can be constructed around identifiers for monitored entities and/or events/conditions defined in the same PEDL script. Monitored methods can form part of expressions by including the start or end of a method as an event. Monitored variables can form part of conditions when logical expressions are used such as the variable being greater or less than a certain value.

### 4.3.6 MEDL structure

Two concepts exist for specifying the verification task in MEDL: alarms and safety properties. Alarms are used to specify events that should not occur (i.e. if verification task is successful, there will be no alarms raised). Safety properties are used to specify conditions that must remain true. A verification task might be expressed using either or both of these concepts, depending on what way it is easiest to reason about the correct operation of an application.

In order to allow more complex reasoning about application behaviour than the MaC logic would otherwise allow, one can define auxiliary variables. These can be used for example to count the number of occurrences of an event. Auxiliary variables can be used within event and condition definitions in the MEDL specification.

```plaintext
ReqSpec <spec_name>

/* Import Section */
import event <e>;
import condition <c>;

/* Auxiliary variable declaration*/
var int <aux_v>;

/* Event and condition definition*/
event <e> = ...;
condition <c> = ...;

/* Property and violation definition*/
property <c> = ...;
alarm <e> = ...;

/* Auxiliary variable update section*/
<e> -> { <aux_v'> := ...; }

End
```

Figure 4.6 – Structure of MEDL specification
The MEDL specification is structured somewhat similarly to the PEDL specification, as shown in figure 4.6 (from Kim et al [72]). The script is first declared, followed by a declaration of events and expressions to be imported from PEDL. Any auxiliary variables to be used in the MEDL specification are first declared. Events and conditions may be defined in the MEDL specification using the imported primitive events and conditions and/or auxiliary variables in logical expressions. Safety properties and/or alarms are expressed in terms of these events and conditions. Finally any updates to be made to auxiliary variables are defined, with an event triggering the update and an assignment made to the auxiliary variable.

4.4 MAC OPERATION

The Monitoring and Checking framework involves several different components and some setup is required prior to actual monitoring. The run-time stage is referred to as the ‘dynamic phase’ and the setup stage is referred to as the ‘static phase’. A graphical user interface can be used to manage the execution of the static phase tasks, as well as the execution of run-time components of MaC. During the static phase, commands are executed to parse the PEDL and MEDL files and instrument the target application. The dynamic phase consists of the execution of the target application as well as the run-time Java-MaC components. Apart from manual creation of the PEDL and MEDL specification files during the static phase, all other steps are automatic.

4.4.1 Static phase

After creating the PEDL and MEDL files, the user must parse the files to prepare for use at run-time. The Java-MaC PEDL and MEDL parsers construct abstract syntax trees suitable for use by the event recogniser and the run-time checker during the run-time verification. If the files have basic syntax errors, these will cause a failure in parsing. Once both scripts are parsed and the target application code has been compiled, the Java-MaC instrumentor is executed. This uses a tool called J Trek [61] to step through the application bytecode and to add the Java-MaC filter component by inserting extra bytecodes at points where updates to monitored entities (as defined in the PEDL script) are encountered (i.e. assignments to variables or beginning/end of methods).
4.4.2 Dynamic phase

The first steps in the dynamic phase of using Java-MaC are to initiate the event recogniser and run-time checker components. The run-time checker upon initiation waits for communication on a TCP socket from the event recogniser. The event recogniser upon initiation establishes a link with the run-time checker on that TCP socket and awaits communication from the filter component of Java-MaC on a separate socket. These sockets (and potentially hosts, localhost being the default) are defined during the static phase.

The next step is to execute the instrumented target application. This also starts the filter component of Java-MaC, which executes as a thread of execution within the target application with various method calls to this filter thread at the monitoring points in the target application. At each monitoring point a call is made to the filter thread relaying the new state of a monitored object (i.e. the new value of a variable, or the start/end of a method), as well as an integer reference used by Java-MaC to identify each monitored object. In order that the code is safe in a multi-threaded environment, a lock is obtained before the call to the filter thread and released thereafter. This can be disabled when not necessary (when the target application only has one thread with monitoring points). The filter thread communicates via a TCP socket to relay these observations to the external event recogniser.

With the target application executing alongside the filter component of Java-MaC and the external Java-MaC components already initialised, the event recogniser receives any observations sent from the filter across the TCP socket. The event recogniser matches observations to events and conditions described in the PEDL file. Any events or conditions that are exported from the PEDL script and imported by the MEDL script and that have been affected by the new observations have their new state relayed to the run-time checker. The run-time checker then checks that the new states are valid according to the MEDL specifications, i.e. all safety properties and alarms are evaluated with warnings or alarms raised if necessary.

4.5 CONSIDERATIONS FOR EMBEDDED JAVA-MAC DEPLOYMENT

Java-MaC is ordinarily designed for the verification of Java applications executing on a standard computing platform (e.g. PC). Facilitating its deployment for verification
of an application executing in an embedded environment requires some custom
arrangements. The requirements are:

- A suitable embedded platform
- Communication of observations from target application
- Minimisation of interference
- Preparation of the target application during the static phase

These requirements are considered below:

**Suitable embedded platform**

The first requirement is that the chosen embedded platform be capable of executing
Java application code (as is the target of Java-MaC). The chosen platform should not
merely provide the same capabilities as a standard computing platform, i.e. JVM with
full implementation of Java API, as this would not constitute a realistic embedded
platform for testing the feasibility of using Java-MaC to verify an application
executing in an embedded environment. The choice of a suitable embedded platform
is considered in chapter 5.

**Communication of events from target application**

An alternative means for relaying observations from the target application to Java-
MaC is required. Assuming an unaltered Java-MaC arrangement with the event
recogniser and run-time checker components on a host system and the filter on the
target system, TCP sockets would be used for communication of events from the filter
to the event recogniser. However, there may not be provision in an embedded system
for the implementation of the TCP stack (as Rodriguez-Andina et al [73] conclude: to
minimise cost, an FPGA may be gate-centric rather than provide as many I/O
options). The Spartan 3 FPGA platform used by the target embedded design described
in chapter 5 does not include Ethernet functionality to facilitate TCP. Even with TCP
available as a communication mechanism it would require target system processing
time to be used in preparing the TCP packets. Therefore a simpler communication
mechanism is recommended and for the purposes of the experimental work described
in this thesis, the serial port was used as a communication mechanism.
Minimisation of interference

With the limited system resources available on embedded systems and potential for affecting real-time schedules, there is a need to minimise interference to the target system. The execution of the Filter thread of Java-MaC on the target system would have an undesirable execution overhead and add another thread of execution that could potentially affect scheduling on the target system. Therefore, it is suggested that the calls to the filter thread added to the target application be replaced by statements that output observations to an external application on the host system, an application to which Java-MaC’s filter thread is appended.

Preparation of the target application

Rather than rely on the standard instrumentation performed by Java-MaC during the static phase, it is suggested that the code added by Java-MaC be replaced by code to simply output observations to the serial port. In order to achieve this, the Java-MaC static phase is first performed as usual. The instrumented bytecode is then examined and the locations of the instrumentation and their contents noted. The original target application code is then manually instrumented with printouts providing the same observations at the same monitoring points. Amendment of the Java-MaC Instrumentor could be arranged in order to perform this automatically, but this was not considered a priority in the test implementation.

4.5.1 Proposed deployment of Java-MaC

The proposed deployment of Java-MaC to verify an application executing on an embedded system is shown in figure 4.7. The main difference compared to conventional deployment is that the immediate target for Java-MaC is not the originator of the actual observations, i.e. the immediate target is the intermediate application. The target system has no monitoring features added apart from the code added during instrumentation.
The proposed monitoring approach of chapter 2 includes an on-chip monitoring core to relay observations to the external monitoring apparatus (in this case, Java-MaC). The role of the on-chip monitoring core is fulfilled in the proposed arrangement of figure 4.7 by relying on the built-in serial communication software and hardware components of the system. The results of experiments undertaken with the proposed solution can be extrapolated to derive conclusions on the feasibility of the monitoring approach described in chapter 2 for the purposes of run-time verification of software in an embedded system.

![Figure 4.7 – Proposed Java-MaC deployment](image)

The target platform for this deployment is introduced in chapter 5 and the specific implementation of this deployment on the target platform is presented in chapter 6.

### 4.6 SUMMARY

This chapter has provided a brief introduction to the particulars of the Java-MaC run-time verification tool. The components of Java-MaC have been described. The specification languages used to specify monitored entities and their relationship to events and conditions and also to specify safety conditions and alarms, have been detailed. The ordinary operation of Java-MaC has also been described. Finally, the deployment of Java-MaC to verify an application executing in an embedded system has been considered. A proposed method of deployment has been introduced based on these considerations.
CHAPTER 5 – JOP: A JAVA-BASED

EXPERIMENTAL PROCESSOR PLATFORM
5.1 INTRODUCTION

The focus of this chapter is provision of a suitable embedded platform for the Java-MaC deployment proposed in the preceding chapter. The monitoring arrangement described in chapter 2 has been proposed as a general approach to runtime verification in embedded systems. However, a Java-based implementation is featured in the experimental work due to the choice of Java-MaC as a runtime verification solution. The results of the experimental work are intended to apply to implementations of the monitoring approach in other programming languages that are more common in embedded systems.

In this chapter, features of the Java programming language relevant to consideration of its use in embedded systems are described and the particular obstacles to deployment of Java in embedded systems are considered. A solution for Java in embedded systems is described: the Java Optimised Processor (JOP) developed by Schoeberl [18], which is chosen here as a suitable experimental platform for the work in this thesis. Some detail is given of its architecture, features and typical operation. The chapter concludes with a presentation of the reasons the JOP platform was chosen as a suitable embedded platform for the experimental work described in this thesis, with reference to the JOP features already described.

5.2 JAVA

Applications written in the Java programming language are the target of a number of existing runtime verification tools, such as the Java PathExplorer tool of Havelund and Roșu [57] and the Java-MaC approach of Kim et al [71] (the runtime verification tool used in the experimental work described in this thesis). The Temporal Rover, described by Drusinsky [23], can also operate on Java applications. Due to the nature of the Java programming environment, such applications are a particularly flexible and useful target for such research.

Java, described in detail by Arnold et al [74], is a modern object-orientated programming language and application platform that attempts to provide a straightforward programming environment and also avoid some of the pitfalls of traditional languages (such as errors introduced due to explicit memory management in the application code).
Unlike most traditional programming languages, application code written in the Java language is translated to ‘bytecode’ and interpreted at run-time, rather than compiled directly to native machine code. This bytecode is interpreted by what is termed the Java Virtual Machine (JVM). The JVM is specific to each hardware platform and natively executes the instructions held in the Java bytecode.

In addition to the JVM and the programming language itself, the Java platform also includes an extensive set of programming libraries: the Java API (Application-Programmer Interface). Different versions of Java are available for various purposes. Java SE (Standard Edition) is commonly used for general purpose applications. Java EE (Enterprise Edition) provides a more extensive set of API functions. Java ME (Micro Edition) allows only a subset of basic API functions common to all Java versions.

5.2.1 Obstacles to using Java for embedded applications

The research focus in this thesis is to examine the feasibility of runtime verification of applications executing in an embedded environment. The existing runtime verification tool, Java-MaC, has been chosen for use in this research. In order to utilise Java-MaC, the issue of Java in the context of an embedded system must be addressed. There are certain difficulties in using Java in an embedded environment, even when considering Java Micro Edition (this and other restricted Java environments are considered by Schoeberl [75]). Two main concerns present an obstacle, with a third related concern to be addressed also. These are:

- **Real-time requirements**
  
  Many embedded applications have real-time requirements. Java is not a real-time programming environment and the JVM architecture results in unpredictable execution times.

- **Use of system resources**
  
  Typically embedded systems have limited system resources. Standard implementations of the Java platform require significant system resources.
• Performance

The JVM interprets bytecode rather than executing compiled code directly. The performance of this interpreter approach can require more systems resources and result in longer execution times.

Real-time issues with memory management

Memory management on the Java platform poses an obstacle to achieving a real-time implementation of Java. Rather than explicit memory management in the application code (e.g. explicit release of memory when objects are due to pass out of scope) Java relies instead on automatic memory management (in order to avoid problems such as the programmer forgetting to release memory used by an object that passes out of scope). This memory management is carried out by a component of the JVM termed the ‘garbage collector’. The garbage collector executes sporadically in order to release memory held by objects that are no longer in the scope of the current application(s).

Performance and use of system resources

In addition to the unpredictability this adds to the use of JVM execution time, garbage collection does not necessarily ensure frugal use of limited system memory for applications, as is often a priority for embedded applications. The JVM itself is an obstacle for embedded systems, as it requires a relatively large amount of system resources in the context of an embedded system and is not very efficient due to the overhead of interpretation. The implementation of an extensive general purpose API is one feature requiring more resources than may be feasible (though this is to some degree reduced by the use of Java ME). However, the JVM also requires resources for the purposes of performing byte-code interpretation.

5.3 FULFILLING EMBEDDED JAVA CONCERNS WITH JOP

Rather than interpret bytecode using a JVM, it is possible to create a hardware environment to execute Java bytecode natively, i.e. on a non-virtual Java Machine. This approach allows one to avoid the overhead of bytecode interpretation and can potentially be used for embedded systems. The JOP (Java Optimised Processor) platform developed by Schoeberl [18] provides a platform for executing Java applications in an embedded system. JOP has been used as a platform for some industrial applications, the experiences of which are described by Schoeberl [76].
A concept termed a ‘soft processor’, a stack computer processor design instantiated in an FPGA, is the core of the JOP platform. Java code is executed without run-time interpretation on the JOP core, i.e. rather than use a JVM, the JOP core acts as a hardware Java machine. The JOP core architecture, as described by Schoebelr [77], has been specifically designed and optimised with the behaviour of typical Java applications and bytecode taken into consideration. The performance of the JOP core has been evaluated by Schoebelr [78] and compared to various alternatives for executing Java in embedded systems.

5.3.1 Architecture of the JOP core

In theory a hardware Java machine offers a means of achieving superior performance to the JVM, by omitting the overhead of interpreting the bytecode. In practice, Java bytecode is cumbersome to execute natively. Although the JOP platform does not provide a virtual machine interpreting bytecode, neither does it execute Java bytecodes natively. Rather it executes its own native ‘microcode’ that has been designed from the ground up with a full consideration of the particulars of Java bytecodes and the usual code constructs and method arrangements used by Java applications; i.e. the microcode is optimised for Java.

In order to ensure a predictable WCET (worst-case execution time) for all Java bytecodes, the processor architecture is simple, without features such as branch prediction logic. There are simply four pipeline stages: microcode fetch, decode and execute, and an initial pipeline stage of bytecode fetch. This is possible due to bytecode translation to microcode taking just one processor cycle. The short pipeline diminishes the effects of omitting branch prediction, as conditional branch delays are relatively short.

The initial JOP pipeline stage of bytecode fetch performs translation from Java bytecodes to native microcode. Java bytecodes are mapped to microcode one to one for common simple instructions and one to many (or to multiple simpler bytecodes) for less common complex bytecodes. The main difference between the JOP core and a JVM is that the JOP core is a reduced instruction set computer (RISC) while a JVM is a complex instruction set computer (CISC).

Another example of the optimisation of the processor design to suit the average profile of Java applications is a ‘method cache’ to cache entire application methods,
as typically Java applications have many small methods. This also improves the performance of the JOP architecture.

5.3.2 Support for real-time applications on the JOP platform

Real-time applications can be executed on the JOP platform. Real-time libraries form part of the JOP API, allowing the creation of real-time applications for the JOP platform. The processor architecture of the JOP core and design of the microcode is such that each Java bytecode has a known WCET (Worst Case Execution Time) for execution on the JOP core. This facilitates real-time programming by allowing WCET analysis of Java applications written for the JOP platform. Scheduling and processor utilisation can be predicted prior to application execution.

In order to avoid the obstacles to real-time programming posed by garbage collection, the JOP platform by default does not perform garbage collection. Instead, memory is statically allocated for objects at the initialisation of the Java application executing on the JOP core. A drawback of this arrangement is that application programming must be arranged in such a fashion as to avoid spurious creation of new objects (as sufficient memory must be available for all such objects at initialisation of the application). The application size is also constrained by the requirement for sufficient memory.

Having a known WCET for each Java bytecode also enables WCET analysis of Java applications programmed for the JOP platform, facilitating design of the application and analysis of schedulability/utilisation. A tool for performing WCET analysis is available as part of the JOP distribution, presented by Schoeberl and Pedersen [79].

5.3.3 Programming for the JOP platform

Rather than use standard Java API libraries (e.g. Java SE or ME), the JOP platform includes its own implementation of commonly used parts of the Java SE library. Additionally, the JOP platform includes some custom API functions for JOP-specific features. A ‘native’ library is included to allow direct hardware I/O, although the library is included primarily for use by the JOP implementations of standard Java SE methods (e.g. System.out).

The standard Java print functions of System.out are available as part of the JOP API. On a PC with Java SE, such printouts are sent to the Java console, a debugging tool.
The JOP implementations of these functions route the output to the serial port of the FPGA board by default.

Real-time Java programs can be created for the JOP platform, due to the processor implementation facilitating a real-time scheduler and the JOP API libraries allowing creation of real-time applications. The RTThread class of JOP.Realtime allows the creation of real-time threads in a Java application for execution on the JOP platform. Instances of the class are given a period and priority, the latter usually being assigned by the programmer according to the period, allowing rate-monotonic scheduling.

5.4 BUILD PROCEDURE FOR THE JOP PLATFORM

The first step in the JOP build process is instantiation of the JOP core onto the FPGA via JTAG. Xilinx development tools are used to instantiate the JOP core to the Spartan 3 FPGA. The JOP core then awaits download of a Java application from the host PC to the FPGA board via the serial port.

The JOP application build process is demonstrated in figure 5.1. The Java application to be executed on the JOP core is written as normal Java code, albeit using the JOP API libraries. The Java code is compiled to bytecode as normal using the standard Java compiler, before being prepared as a JOP-ready format.

The JOP download process is demonstrated in figure 5.2. The compiled code is downloaded to the FPGA board using a JOP tool that communicates with the ready/waiting instantiated JOP core. A simple echo protocol is used, with each byte received by the JOP core echoed back to the host PC in order that the download application can exercise flow control to the serial connection.
Once download of the Java application is complete, the JOP core commences the execution phase. Any application output sent by the JOP core to the serial port of the FPGA board is received by a listening echo application on the host PC (executed directly after the download application is finished on the host). This echoes the JOP application output to the host PC standard output stream (the console screen by default).

5.5 CHOOSING THE JOP PLATFORM

The JOP platform was chosen as an embedded platform for the Java-MaC arrangement described in chapter 4 to facilitate run-time verification of an application executing on an embedded system. The concerns set out in section 5.2.1 are addressed by the JOP platform as follows:

- Performance

The hardware Java machine approach alleviates the performance problems of Java by dispensing with interpretation.

- Real-time features

The JOP architecture is designed to ensure a known WCET for all bytecodes. The JOP platform is also designed to avoid garbage collection (an obstacle to predicting execution time). A built-in scheduler and real-time thread API also facilitate development of real-time applications. The WCET analysis tool available with the JOP distribution allows static analysis of application code.
• System resources

The JOP platform implements a minimal subset of the standard Java API. Additionally, the lack of any interpretation features (only a single ‘translation’ pipeline stage) means that no additional resources are required to process bytecode.

Additionally the JOP project is open source and thus can be fully examined and code modified to suit the experimental work if necessary.

5.6 SUMMARY

This chapter introduces some of the basic features of the Java programming language and platform. Two main obstacles to the deployment of Java in embedded systems are detailed; system resources required and lack of real-time features. Also the third related area of performance is considered. The embedded Java solution of the JOP platform is described. The means by which its processor architecture is Java-optimised is summarised and the real-time features of the JOP platform are detailed. The usage of the JOP platform and its build process are described. Finally, the JOP platform features that overcome issues with executing Java in an embedded environment are presented as a rationale for choosing the JOP platform for the Java-MaC deployment proposed in chapter 4.
CHAPTER 6 – DEPLOYMENT

OF JAVA-MAC WITH JOP
6.1 INTRODUCTION

This chapter is concerned with implementing the Java-MaC deployment proposed in chapter 4, using the Java Optimised Processor described in chapter 5 as the target platform. The aims and objectives of undertaking this deployment are presented in this chapter. The proposed deployment approach is introduced along with details of its implementation. An initial test of the arrangement with a basic run-time verification scenario is presented. Some conclusions are drawn based on this initial test. Two test cases, detailed in chapter 7 and 8, are proposed to examine the effectiveness of the experimental embedded monitoring implementation in facilitating run-time verification.

6.2 AIMS AND OBJECTIVES

The specific aim of the work described in this chapter is to allow Java-MaC to verify an application that is executing on the JOP platform. This is intended to facilitate the general objective set out in chapter 1 of evaluating the feasibility of the proposed monitoring approach for deeply embedded systems (as proposed in chapter 2). A configuration for deploying Java-MaC to verify an application executing on an embedded system has been proposed in chapter 4 and this chapter describes the implementation of that proposal for the JOP platform.

The specific objectives of the work described in this chapter are:

- Propose a means of facilitating the proposed Java-MaC deployment
- Implement a working solution
- Test that the solution works in practice
- Develop conclusions from the initial working trials
- Develop test cases to further explore the effectiveness of the arrangement

The expectation prior to commencing this work was that a suitable arrangement could be provided to allow Java-MaC to indirectly verify an application executing on the JOP platform, thus demonstrating that run-time verification could be used as a monitoring approach for embedded systems and providing an experimental platform for case studies to further investigate the feasibility of the approach.
6.3 PROPOSED DEPLOYMENT

The elements involved in deploying Java-MaC with the JOP platform are shown in figure 6.1. The JOP core, instantiated to a Spartan 3 FPGA, forms the target platform (a labelled snapshot of the development board is included in appendix A). A general purpose PC forms the host platform on which Java-MaC is run. A serial port serves as the communication link between target and host. The target application is executed on the JOP core. An intermediary between the target application and Java-MaC is shown; this module of the test arrangement will now be explained.

![Figure 6.1 – Elements of a Java-MaC deployment with the JOP platform](image)

The Java-MaC deployment arrangement proposed in chapter 4 calls for use of an intermediary: first to receive observations from the target application on the JOP core and second to provide a local target for Java-MaC in order that it can use the observations. To fulfil this role, an intermediate Java application executing on the host PC is proposed. This application reads the serial port data to receive the monitoring output from JOP (provided by the instrumented code added to the target application) and acts as a local target application for Java-MaC on the host PC. This approach allows Java-MaC to indirectly verify the original target application using the observations received by the intermediate application. The intention is deployment of the intermediate application in such a way that Java-MaC interacts with it as if it were the original target application.

This avoids the need to adapt the Filter component of Java-MaC for use with the JOP platform. However, this precludes the normal arrangement used by Java-MaC, which is to automatically instrument the target application with both the Java-MaC filter thread and synchronised method calls to the Filter from within the target application code. In order to allow some use of this normal Java-MaC approach, the proposed
architecture of the intermediate application is to have a duplicate method and member structure to the original target application, in order that automatic Java-MaC instrumentation may be performed on the intermediate application as if it were the original target.

6.4 IMPLEMENTING THE INTERMEDIATE APPLICATION

The following must be achieved in order to implement the proposed intermediate application:

- Access to serial port data
  A Java application on the host PC cannot directly access the serial port (or other system hardware) as it executes on a Java Virtual Machine.

- Storing observations in a fashion ready for Java-MaC
  The Java-MaC verification task pertains to the original target application. As such, if the static phase is executed to verify the original target, the intermediate target must provide the same monitored entities if it is to act as a surrogate for the original target. These entities must be updated by the intermediate application as relevant observations are received from the target.

- Integration with Java-MaC
  The intermediate application must be prepared in such a way that little or no modification is required to Java-MaC.

6.4.1 Providing access to serial port

The monitoring output of the target system consists of observations sent by simple print statements included in the target application code at the locations where entities monitored by Java-MaC are updated. These observations are communicated using the simple mechanism of the serial port. This is used in the experimental work as a convenient communication mechanism with hardware support. In a more developed model, some other form of I/O port could be utilised, for example the IEEE 1149.1 JTAG interface [80] or a more advanced debug port.
Figure 6.2 – Modified build chain, output redirected to intermediate application

The echo program used by the JOP platform after download of an application to the FPGA automatically receives the JOP application output and echoes it to the console. In order to circumvent the issue of direct hardware access from inside the intermediate Java application, use is made of this console output by the echo program, as shown in figure 6.2. The echo program is executed with a command that simultaneously executes the intermediate application. The output of the echo program is piped to the standard input stream of an application on the host PC, allowing it to receive the monitoring observations. By this mechanism, the intermediate application is given access to data from the serial port.

6.4.2 Storing observations

In order to provide a new target for Java-MaC, the intermediate application must store the monitoring updates received from the input stream in entities corresponding to those in the original target application. The initial approach taken was that the intermediate application was given a class and member structure mimicking that of the target application, but only for those parts relevant to Java-MaC. An example of this approach is shown in figure 6.3, using a simplified version of a railroad crossing scenario.

The application components form a model of a railroad crossing with one train and one gate. The train progresses along a single track and the gate barriers are raised and lowered. This is a specific form of the generalised railroad crossing scenario described by Heitmeyer and Lynch [81] and is used as an example for a test case in chapter 7. Elements of the original target application are shown (a), along with their
counterparts in the intermediate application (b). Only the attributes that concern Java-MaC are duplicated by the intermediate. This type of intermediate application is thus described here by the author as a ‘skeleton application’.

![Diagram showing comparison of target and skeleton applications]

This strategy results in two RailroadCrossing implementations, one of which acts as a surrogate for the original. In the example of figure 6.3, the variables barrierPosition (a member of Gate, an instance of which is created by RailroadCrossing) and position (a member of Train, an instance of which is also created by RailroadCrossing) are entities of the target application that are to be monitored by Java-MaC. Various methods for implementing the application behaviour are present in the target application, but as these are not monitored by Java-MaC, there is no need for the skeleton application to duplicate them.

Instead, only one method, the main method, is present in the skeleton RailroadCrossing implementation. The input to the skeleton application is read and stored by this method. The input is stored by use of a switch statement that reads the
input and determines where to store the update. By this arrangement, variables in the skeleton application are updated to correspond with those in the target application.

### 6.4.3 Integration with Java-MaC

As described in chapter 4, ordinarily the target application is automatically instrumented with code to provide monitoring updates to Java-MaC. A Filter thread is added to the target application and statements to call the Filter and pass observations to it are added at each point at which a monitored entity is present in the target application code. The Java-MaC embedded deployment proposed in chapter 4 amends this by adding the filter thread to a new intermediate application (implemented here as a skeleton application) and replacing the calls to the Filter in the target application code with print statements.

By implementing the intermediate application as described in the preceding sections, not only can modification of the Java-MaC components be avoided, but also the specification files used as part of the verification task can apply exactly to the intermediate application as they would for the original target. In order to prepare the static phase of Java-MaC for use with the skeleton application rather than the target application, some additional steps are required to those described for the ordinary static phase described in chapter 4.

The amended procedure begins with the ordinary execution of the Java-MaC static phase, i.e. preparing the original target application and Java-MaC for use together. The modified target application is examined and the points at which statements have been added by Java-MaC are noted (as well as the references used by Java-MaC to refer to monitored entities). The original target application code is then manually modified with print statements added at each of the Java-MaC instrumentation points. These statements print the reference used by Java-MaC for the monitored entity and the updated value or state. At this stage, the original target application is ready for deployment on the JOP core and has been prepared for use with the modified Java-MaC deployment.

The skeleton application is then prepared as described in the preceding sections. The static phase of Java-MaC is then repeated with this skeleton application taking the place of the original target application. This adds the Java-MaC filter to the skeleton
application. At this stage, the skeleton application and Java-MaC have been prepared for use with the original target application. The execution phase can now commence.

The Java-MaC event recogniser and run-time checker are started. The JOP build process is executed as normal, except that the final command to download the target application is modified with the appending of a command to execute the skeleton application and pipe the output of the JOP download program to it. With the execution of this combined command, the run-time phase of the target application commences as soon as download to the JOP core is complete. As soon as the skeleton application receives any relevant input, it will update accordingly, allowing Java-MaC to indirectly verify the target application behaviour.

![Figure 6.4 – Run-time verification using the skeleton application](image)

The execution of run-time verification using the arrangement described in these sections is shown in figure 6.4. Updates are initiated by method calls to `System.out` within the target application to relay execution updates (1). These are sent across the serial port (2) and received by the JOP build tool, the console output of which is piped to the skeleton application (3). The instrumented skeleton application relays observations to the filter thread (4), which sends events via socket (5) to the event recognizer on the same host system. The event recognizer triggers the run-time checker where appropriate, via a second TCP socket (6).
6.5 TESTING THE JAVA-MAC DEPLOYMENT

In order to test that the Java-MaC deployment described in section 6.4 works in practice, it is necessary to determine if Java-MaC can indeed verify a rule pertaining to an application executing on the JOP platform. There are four steps in achieving this:

- Specification of a scenario to which a verification task can be applied
- Creation of a target application implementing this scenario
- Creation of Java-MaC specification files for the verification task
- Implementation of the intermediate application

After initialisation is confirmed, the Java-MaC event recogniser and run-time checker provide no indication to the user that run-time verification is continuing until an error is detected. There are two mechanisms by which such an error may be encountered and both pertain to verification rules specified in the MEDL script. First, if a safety property has been defined, any violation of that property will be reported to the user, i.e. if the property being monitored does not hold true. Second, if an alarm has been defined, it will be reported to the user if that alarm is raised, i.e. if Java-MaC detects that the event on which the alarm is based has occurred.

6.5.1 A ‘prime counter’ scenario

In order to provide successful trial of the Java-MaC deployment, a verification scenario was proposed such that only a single rule has to be defined and verified. In order to ensure feedback is provided to prove that run-time verification is taking place, the verification rule was designed in the form of an alarm. As such, it is only necessary to ensure that the target application provides the event that is the basis for the alarm. If Java-MaC raises the alarm, then it will have successfully detected that event, thus demonstrating that it has verified the original target application.

With the aim of providing a simple quick scenario for the verification task, a non-interactive autonomous application was envisaged, similar to applications used as programming examples. An application with a main execution loop that simply iterates through integer numbers provides this autonomy. In order that the application behaviour offers scope for verification, some ‘work’ was added to this execution loop.
Checking whether or not each integer is a prime number requires little implementation. The application was given the task of counting how many prime numbers are ‘found’. Errors can then be added to the implementation such that Java-MaC can be used to monitor for the discrepancy between a specified proper operation and an erroneous run-time execution.

6.5.2 Application implementing the ‘prime counter’ scenario

The application to implement the scenario was designed to iterate through integer numbers up to a specific limit (100 was used for testing). A function was implemented that checks whether a given integer is prime or not, by checking if the integer is divisible (i.e. an even number, or an odd number divisible by odd numbers up to the root of the number itself). The function accepts an integer as an argument passed to it and returns a boolean true or false depending on the result of the divisibility test.

Figure 6.5 – Diagram of prime counter main execution loop

Figure 6.5 shows a diagram of the main execution loop of the prime counter application. For each iteration of the loop, an integer variable is increased by one and passed to the function that checks for divisibility. A check is performed on the result

71
returned and if the number is found to be prime, the count of how many prime numbers have been found is incremented by one. This continues for a predetermined number of cycles (i.e. until the test integer reaches a certain value).

In order to add an error that can be detected by Java-MaC, the check on the result returned by the prime number checking function is artificially arranged. Rather than simply wrapping the function call in the if statement (so that the action taken depends on the result returned), the result returned is stored in a Boolean variable. A comparison is made between this Boolean value and the Boolean constant true (although again, normal practice would be simply to wrap the Boolean variable in the if statement).

```java
//CORRECT:
for(testvalue = 3; testvalue < CEILING; testvalue++) {
    isprime = isPrime(testvalue);
    if (isprime == true)
        primesfound++;
}

//ERROR:
for(testvalue = 3; testvalue < CEILING; testvalue++) {
    isprime = isPrime(testvalue);
    if (isprime = true)
        primesfound++;
}
```

Figure 6.6 – Target application code showing placement of error

This allows the incorporation of an error that might arise as a result of programmer inattention. Similarity between the Java language Boolean comparison operator ‘==’ and the assignment operator ‘=’ means the two can potentially be confused (and the application will compile normally). Incorporation of this programming error is shown in figure 6.6. The first code fragment is the error-free implementation of the target application. The second fragment contains the error, use of the assignment operator rather than the Boolean comparison operator.

The effect of the error is that the if statement will evaluate to true every time (isprime will be tested and found to be true, as the assignment occurs before the condition is checked). This means that primesfound will be incremented on each loop iteration, producing an erroneous result.
6.5.3 Verification task

The verification task, as stated in section 6.5, was designed such that an alarm can be raised by Java-MaC if an event occurs in the target application execution. This event should occur when the error described in the previous section is introduced into the application, but should not occur ordinarily. To fulfil these criteria, the verification task set for Java-MaC was to check that no more than the expected numbers of prime numbers are counted by the application (i.e. 25 prime numbers for the limit of 100 that the application tests up to). Only when the error is present in the target application will the count of prime numbers will be too high, leading it to exceed the expected result.

MEDL specification

The MEDL specification for the prime number counting scenario is detailed in figure 6.7. As explained in chapter 4, this specification does not make reference to implementation-specific features of the application being monitored. The verification task is expressed using an alarm, i.e. an event that should not occur (Java-MaC will provide notification if it does). The alarm toomanyprimes is raised when more than the expected numbers of prime numbers are found. This can be considered the start of a condition, primesabovelimit, i.e. a condition that holds true once more than 25 prime numbers are found (primesfound > 25).

```
ReqSpec testprime
    import event firstprimefound,newprimefound;
    var int primesfound;
    condition primesabovelimit = primesfound > 25;
    alarm toomanyprimes = start(primesabovelimit);
    firstprimefound -> {primesfound = 1; }
    newprimefound -> {primesfound = primesfound + 1; }
End
```

Figure 6.7 – MEDL specification for prime number counter

In order to count the number of prime numbers found, the number of times a new prime number is found (an event), is counted. The MEDL specification relies on a primitive event, newprimefound, imported from the event recogniser. These events are counted by means of an auxiliary variable, primesfound, that is incremented by one each time newprimefound occurs. As this incrementing relies on the previous
value of **primesfound**, the specification also has to define an initial value for **primesfound**. The number of prime numbers found can be initialised to one when the first prime number is found (a primitive event, **firstprimefound**).

**PEDL specification**

The MEDL specification relies upon certain primitive events (**firstprimefound** and **newprimefound**). These events must be linked to occurrences during the target application execution. Each time a new prime number is found by the target application, the **primesfound** integer variable (shown in the code fragment of figure 6.6) is incremented. Thus by monitoring this variable, the primitive events can be defined. The PEDL specification for this is shown in figure 6.8.

```plaintext
MonScr testprime
/* Primitive events to be exported to run-time checker */
export event firstprimefound, newprimefound;
/* Monitored entities */
monobj int testprime.TestPrime.primesfound;
/* Definition of primitive events */
event firstprimefound = update(testprime.TestPrime.primesfound)
when (testprime.TestPrime.primesfound == 1);
event newprimefound = update(testprime.TestPrime.primesfound)
when (testprime.TestPrime.primesfound > 1);
End
```

Figure 6.8 – PEDL specification for prime number counter

The PEDL specification begins with definition of the primitive events that the event recogniser should export to the run-time checker. The variable to be monitored is then defined. The two events are defined similarly. Each event is of the form ‘event when condition’, with the relied-upon event being an update of the variable. The event **firstprimefound** occurs when the **primesfound** variable is updated but also is equal to one. The **newprimefound** event occurs when the **primesfound** variable is updated but also is greater than one.

**6.5.4 Implementation of intermediate application**

The intermediate application for the prime number counting scenario is relatively trivial. It has to perform three functions: read input from the serial port, parse the input for monitoring observations and store any observations appropriately. The code implementing these functions is shown in figure 6.9. Note that the application is given the same class package/name as the reference made to the original application in the Java-MaC PEDL specification.
The input piped to the application at the console is read by opening the input reader in line 8. The input is read line by line by means of a loop (lines 11-21), with the attempt to read in a line (line 13) wrapped in an exception handler (lines 12-17). The if statement of line 18 checks for a valid line of input. A monitoring observation is of the form "01 xxx" where "01" is the Java-MaC reference to the monitored entity and "xxx" is its new value. Thus in line 19, an attempt is made to parse and match "00" at the beginning of the line (numbering of monitored entities begins at 0 and only one monitored entity is present in this example). In line 20, an attempt is made to parse the integer value – note that this is only possible due to knowing in advance what kind of value will be present for a particular monitored entity. In this example, the only monitored entity is of type integer. The new value of `primesfound` is stored in line 20 also.

### 6.6 CONCLUSIONS ON INITIAL TRIAL

The implementation of the prime number counting application and verification task provided an initial test of the Java-MaC deployment of section 6.4.3 that implements the ‘intermediate application’ arrangement proposed in section 6.3. The verification task described in section 6.5.3 was carried out on the target application and the alarm was raised by Java-MaC when an error was incorporated into the target application as described in section 6.5.2. This alarm was raised as soon as more than the expected amounts of prime numbers were counted by the target application, i.e. in advance of the target application providing an incorrect final count.
This initial test demonstrated a working deployment of a run-time verification tool (Java-MaC) to verify an application executing in an embedded system (the JOP platform). The chosen verification task demonstrated one of the capacities of run-time verification, namely, the capacity to highlight incorrect operation of the target application to the user before any indication of incorrect operation is provided by the application itself.

The difficulties in monitoring embedded systems have been detailed in chapter 2. The deployment detailed in this chapter provides some allowance for embedded systems, with less monitoring code executed by the target application than would be the case with an ordinary Java-MaC deployment. As well as avoiding inclusion of the filter thread within the target application, the overhead of executing the monitoring code is further reduced by replacing function calls at monitoring points with simple print statements. The platform detailed in this chapter and implemented with the initial trial is thus considered to be suitable for further investigation with the use of test cases to evaluate its performance.

6.7 DEVELOPING TEST CASES

Two test cases to further investigate the Java-MaC/JOP deployment described in this chapter were developed. The first test case seeks to implement an ordinary verification task for JOP, already implemented for PC using Java-MaC. This first test case, described in chapter 7, was designed to fulfil that purpose, as well as to:

- Monitor multiple entities
- Verify timeliness and ordering
- Demonstrate verification of a real-time application
- Evaluate effectiveness of the intermediate application arrangement

The second test case seeks to implement for JOP a verification task that is applicable to real-time embedded systems. This second test case, described in chapter 8, was also designed to:

- Monitor objects in multiple execution threads
- Verify timing and scheduling parameters
- Verify against static analysis
These test cases can be used to explore the effectiveness of the Java-MaC/JOP arrangement described in this chapter.

6.8 SUMMARY

This chapter described the implementation of the Java-MaC deployment proposed in chapter 4, on the JOP platform described in chapter 5. Details were provided concerning how an intermediate application was required to receive monitoring observations on the serial port and provide a new target for Java-MaC (allowing it to indirectly verify the original target application). An initial approach to this problem was described, along with the implementation details. A basic verification task for an application counting prime numbers was described as a test for the initial implementation. Conclusions were presented based on the execution of this initial test. The chapter concludes with a summary of the parameters used for developing two test cases to examine the effectiveness of the Java-MaC/JOP arrangement described in this chapter. The first test case will implement a standard verification task on the JOP platform and the second will implement a verification task with applicability to real-time and embedded systems.
CHAPTER 7 – RAILROAD CROSSING

TEST CASE
7.1 INTRODUCTION

The first test case proposed in the preceding chapter to explore the effectiveness of the JOP/Java-MaC arrangement is to implement an example run-time verification task for JOP. This chapter describes the implementation of a ‘railroad crossing’ scenario for JOP as well as an associated Java-MaC verification task. The railroad crossing scenario, which has already been used to demonstrate the features of Java-MaC, is introduced. Details are provided concerning the original implementation of the scenario as an example run-time verification task for Java-MaC. A new implementation of the scenario in a format suitable for deployment on JOP is then presented. Results from the execution of this test case are detailed. Conclusions are reached based on these results concerning the effectiveness of the monitoring approach for run-time verification of applications executing in embedded systems. Following on from one of the conclusions, an amended version of the monitoring arrangement is presented.

7.2 AIMS AND OBJECTIVES

The monitoring arrangement described in the preceding chapter was trialled with a basic initial example to test that the proposed arrangement could work in practice. In this chapter, the monitoring arrangement is explored further with the development of a test case study for the system, as proposed in the preceding chapter. The aim of this test case is to evaluate the monitoring approach and explore its effectiveness for run-time verification. Specifically, the test case was arranged to allow comparison between run-time verification using a traditional deployment, versus run-time verification using the embedded monitoring approach. There are some specific objectives to the work described in this chapter, outlined as follows:

- Definition of a suitable scenario for the test case, with run-time verification involving the monitoring of multiple entities.
- Implementation of a version of the scenario with typical application attributes for an embedded system, e.g. real-time functionality.
- Verification of timing and event ordering in the target application.
• Evaluation of the monitoring approach based on experiences from executing the case study and comparison with run-time verification on a non-embedded platform.

Following on from the initial aim, the author proposes amendments to the monitoring scheme, in order to provide an improved implementation for further evaluation of the monitoring approach for embedded systems.

It was anticipated that the objectives could be met by providing a new implementation of a scenario used for demonstrating the features of Java-MaC. This is the ‘railroad crossing’ scenario, which will be detailed in the course of this chapter.

7.3 RAILROAD CROSSING SCENARIO

The railroad crossing scenario used as an example by Kim et al [60] to illustrate the use of PEDL/MEDL is a model that can be considered as a specific form of the generalised railroad crossing scenario described by Heitmeyer and Lynch [81]. It is suggested by Heitmeyer and Lynch that this scenario can be used for comparing various formal methods. In the generalised scenario, the problem involves a railroad crossing (a railway level crossing) through which a number of trains may pass, on a number of tracks, in both directions. A gate is present at the crossing that can be raised or lowered. The problem has a safety property that the gate must be in the lowered position when any train is in the crossing.

The example of Kim et al [60] differs in that only one train at a time is present, passing in a single direction along the railway line. In this model the gate takes some time to raise or lower and a controller is used to raise the gate or lower it. The safety property remains that the train must not be in the crossing while the gate is not down.

7.3.1 JavaMaC implementation

In the Java-MaC implementation, the railroad crossing is modelled as a Java applet, providing a graphical animation of the scenario with a train crossing from left to right along a track and a gate raising/lowering. The applet also displays the time elapsed for the current train, the global time elapsed, and messages when gate is fully down.

The classes used to implement the railroad crossing scenario are shown in figure 7.1. The applet class RRC is defined to include separate classes for the controller, train and gate. Another class is used to implement a signal model, used by the controller...
and the gate. All the classes are public and the signal classes are monitored by their parent class, but modified by the other classes. Signals are set, cleared or waited upon using semaphores. Parameters for the scenario, such as the time taken for the train to reach the crossing, can be set in the webpage into which the applet is loaded.

![Diagram showing the components of the railroad crossing implementation](image)

**Figure 7.1 – Components of the railroad crossing implementation**

The four main classes are all ‘runnable’ threads and are executed concurrently at the commencement of the scenario (scheduling is left to the JVM although the thread RRC, performing the animation, is given higher priority). The classes Controller and Gate are both extensions of a TimeThread class that implements local clocks for any thread built upon it. As depicted in figure 7.1, the class RRC has one instance each of Controller, Train and Gate. Controller has two instances of class Signal, `ns` representing a signal that the train is near to the crossing and `ps` representing a signal that the train is past the crossing. Gate also has two instances of Signal, `lower` representing a signal that the gate should lower and `raise` representing a signal that the gate should raise.

The behaviour of the application is based around the sequence of communication between the various threads, as shown in figure 7.2. The thread `train` updates the train position as the execution progresses. As the train position approaches the crossing, `train` sets the signal `ns`, notifying `controller` that it is near the crossing. The signal `lower` is then set by `controller`, notifying `gate` to start lowering. Similarly when the train position passes the crossing and has left it, `train` sets the signal `ps`, notifying `controller` that it has passed the crossing. The signal `raise` is then set by `controller`, notifying `gate` to start raising.
As stated previously, the safety property is that the train must not be in the crossing, while the gate is not down. This safety property is expressed in MEDL as shown in figure 7.3. The property relies on two conditions, IC and GD, which correspond to ‘(train) In Crossing’ and ‘Gate Down’. The expression !IC || GD is a reworking using De Morgan's laws [82] [83] of the logical expression !(IC && !GD) that corresponds to ‘not (in the crossing while gate is not down)’.

The conditions in the MEDL specification are built from primitive events corresponding to the start and end of the conditions ‘In Crossing’ and ‘Gate Down’. These events, startIC, endIC, startGD, endGD are imported from the event recognizer. The condition IC is specified as the interval between startIC and endIC and the condition GD is specified as the interval between startGD and endGD.
The PEDL specification defines these four primitive events, exported by the event recogniser to the run-time checker, as shown in figure 7.4. In order to define a condition $IC$ (in crossing) a number of variables are monitored, although apart from $RRC.train_x$ (the train position) the variables are actually static in the target application and only monitored so that the PEDL specification can use their values in calculations. The train length is held in the $RRC.train_length$ variable, the crossing position by $RRC.cross_x$, and the crossing length by $RRC.cross_length$. The condition $IC$ is defined as the sum of the train length and position being greater than the crossing position and the train position being less than or equal to the sum of the crossing length and position. The primitive events $startIC$ and $endIC$ are specified as being the start and end respectively of the condition $IC$.

```plaintext
MonScr RailRoadCrossing

/* Primitive events exported to run-time checker*/
export event startIC, endIC, startGD, endGD;

/* Monitored entity declarations */
monobj float RRC.train_x;
monobj int RRC.train_length;
monobj int RRC.cross_x;
monobj int RRC.cross_length;

monmeth void Gate.gd(int);
monmeth void Gate.gu();

/* Event and condition definitions */
event startIC = start(IC);
event endIC = end(IC);
event startGD = endM(Gate.gd(int));
event endGD = startM(Gate.gu());

condition IC = RRC.train_x + RRC.train_length > RRC.cross_x &&
RRC.train_x <= RRC.cross_x + RRC.cross_length;

End
```

Figure 7.4 – PEDL script for railroad crossing

The method $gd(int)$ of the Gate class is monitored. This method lowers the gate and monitoring it allows the specification of $startGD$ (start of the gate being down) as being the end of the $gd(int)$ method. Similarly the method $gu()$ of the Gate class is monitored. This method raises the gate and $endGD$ (end of the gate being down) can be specified as the start of the $gu()$ method.

### 7.4 RRC IMPLEMENTATION FOR JOP

In order to fulfil the objectives set out in section 7.2, the railroad crossing scenario used as a Java-MaC example was implemented again for the JOP platform. For the purposes of exploring the effectiveness of the monitoring approach described in this
thesis for run-time verification in an embedded environment, the same safety property was verified in the JOP platform implementation as for Java-MaC, in order to allow comparison of the run-time verification. The new implementation followed the arrangement of the original scenario in some general respects (e.g. one train at a time passing in one direction along a single track), also to facilitate comparison between run-time verification of a PC-based application versus an application executing in the embedded environment.

7.4.1 Differences of the JOP implementation

Although the safety property being verified remained the same and the scenario similar, the actual implementation of the scenario for the JOP platform differed in three particular respects: the form of implementation, the class and thread model used and the concept used for signals.

The first difference is that the Java-MaC implementation is in the form of a Java applet that executes when viewed on a webpage in a web browser, whereas the implementation for the JOP platform is a full application. The second variation is that the concept of threads differs between the standard Java API and the JOP implementation. Rather than using ‘Runnable’ classes, the JOP platform allows (and indeed only offers) the concept of real-time threads, using a class ‘RTThread’ that other classes can be extended from. ‘Runnable’ classes implement an interface (i.e. provide the methods described in the interface definition, namely a “run” method) and can thus extend an arbitrary class while implementing the interface. Using the RTThread model, instances of this class must be used, or else user-defined classes must extend this class rather than extending other classes. As described in Chapter 5, the real-time threads are scheduled periodically by the JOP core’s built-in scheduler.

The third issue is the reliance on semaphores to implement signalling in the original application. The JOP API does not implement the wait/notify semaphore API methods used by the original implementation. Although user-defined semaphores could be programmed for the JOP implementation, this is avoided as a blocking thread of execution could cause priority inversion unless a user-defined scheduler model was also implemented. Instead the chosen approach was to rely on the periodic execution and scheduling to co-ordinate the order of communication.
7.4.2 Components of JOP RRC scenario

In order to accommodate the differences in implementation, the railroad crossing scenario differs from the original version used by Java-MaC. Specifically, signals are incorporated into the overall model as discreet components of the scenario. This facilitates the re-arranged communication used between threads in the implementation for the JOP platform.

![Components of the railroad crossing model](image)

Figure 7.5 – Components of the railroad crossing model (not to scale)

Components of the revised railroad crossing scenario are shown in figure 7.5. The train, controller and gate are present as before. Additionally, there are two signals: a pre-gate signal that is to be set when the end of the train passes it and a post-gate signal to be set when the end of the train passes it, with the pre-gate signal being cleared. In other words, the crossing area is implemented as a single track circuit. When the train has passed the post gate signal (i.e. post-gate signal is set) it is safe to raise the gate.

The model is designed to accommodate a particular maximum speed of train. The basic parameters, e.g. the distance between the pre-gate signal and the crossing gates, determine what speed of train can be accommodated. Thus, in the model described, using too high a value for the pre-determined speed of the train will cause the system to fail, i.e the train arrives at the crossing gates before the gates are fully lowered.

7.4.3 Classes in the JOP railroad crossing implementation

Each of the components of the revised railroad crossing scenario are implemented as classes in the application for the JOP platform, shown in figure 7.6. These are TrainModel (the train), GateModel (the gate) and ControllerModel (the controller). Additionally the overall scenario is encapsulated in a class RRCModel (the railroad crossing model). Each of these classes is implemented as an extension of the JOP API’s real-time thread class: RTThread.
To implement the two signals, two instances of a new SignalModel class were created within RRCModel. This SignalModel class has as members, a boolean variable that can be set or cleared (i.e. the status of the signal: danger or clear) as well as an integer position variable defined upon creation (the absolute position of the signal along the track section in question). It has methods for other classes to read its position or status.

TrainModel periodically updates a variable with the train’s position based on current system time, how long the train has been running, and its predefined speed. In each period, the position variable is checked to see if the train has passed a signal. If it has, the signal is set at danger.

GateModel has three possible states: raising the barriers, lowering the barriers, or idle. In addition, a variable can be set or cleared to denote whether a warning beacon for road users is active. In each period, if the barriers are being raised or lowered, a variable is updated appropriately with the angle of the barrier, as calculated by the time elapsed since raising/lowering began. If the barriers have reached the raised or lowered positions, the model is reset to an idle state once more.

ControllerModel tracks the overall state of the crossing model and operates accordingly each period depending on the state it is in. The state changes are dictated by current state and inputs such as the states of the SignalModel instances, current time, and state of the GateModel instance. These state changes are shown in the state transition diagram of figure 7.7.
7.4.4 Operation of railroad crossing model for JOP

As with the original application, the behaviour of the JOP application is based around the sequence of communication between the various threads. The sequence of interactions between the various threads is shown in figure 7.8. Note that preGateSignal and postGateSignal are not threads of execution and the other components, although executing in parallel, are scheduled periodically. For reference, the states of controller (described in figure 7.7) are also included on the diagram.

The complete railroad crossing model operates as follows:

- When the train passes the signal position, train sets preGateSignal.
- When isDanger is read from preGateSignal, controller leaves the state [start] and does nothing for a pre-set interval (state [wait]).
- When this interval has elapsed, controller calls gate method startWarning and goes to state [warn] where it does nothing for a pre-set interval.
- When this interval has elapsed, controller calls gate method startLowering and enters the state [lower].
- When an isLowering value of false is read from gate (isLowering is set to false when gate completes updating of the barriers to a lowered position), controller progresses to state [cross].
• In normal operation, the train passes through the crossing in this interval. When the train passes the post-gate signal position, \texttt{train} sets \texttt{postGateSignal} and clears \texttt{preGateSignal}.

• When an \texttt{isDanger} value of \texttt{false} is read from \texttt{preGateSignal}, \texttt{controller} progressed to state \texttt{[raise]} and calls \texttt{gate} method \texttt{startLowering}.

• When an \texttt{isRaising} value of \texttt{false} is read from \texttt{gate (isRaising} is set to \texttt{false} when \texttt{gate} completes updating of the barriers to a raised position), \texttt{controller} resets to state \texttt{[start]}.

![Sequence diagram of the JOP railroad crossing application](image)

Figure 7.8 – Sequence diagram of the JOP railroad crossing application

### 7.4.5 Verification of JOP RRC scenario using JavaMaC

Using the version of the railroad crossing scenario described in section 7.4.2 it was possible to use the original MEDL specification designed for the Java-MaC railroad crossing example. In order to facilitate readability, the abbreviations used in the original specifications were expanded for the JOP implementation's MEDL
specification (shown in figure 7.9). Thus IC is replaced by inCrossing, GD by gateDown, startIC by startInCrossing, endIC by endInCrossing, startGD by startGateDown, endGD by endGateDown.

```plaintext
RegSpec RailRoadCrossing

/* Primitive events imported from event recogniser*/
import event startInCrossing, endInCrossing,
       startGateDown, endGateDown;

/* Definition of conditions */
condition inCrossing = [startInCrossing, endInCrossing);
condition gateDown = [startGateDown, endGateDown];

/* Definition of safety property*/
property safeRRC = !inCrossing || gateDown;
End
```

Figure 7.9 – MEDL specification for the JOP railroad crossing implementation

As in the original Java-MaC example scenario, the purpose of the PEDL specification (shown in figure 7.10) was to specify the primitive events for the Java-MaC event recogniser to export to the run-time checker. These events, startInCrossing, endInCrossing, startGateDown, endGateDown are specified similarly to the original Java-MaC PEDL specification, with the difference that startGateDown and endGateDown are dependent not on the end of a gate lowering method and the start of a gate raising method respectively. Instead, both are now dependent on a GateModel variable degreeBarrierAngle (the angle of the barriers on the crossing). Both rely on the same condition, degreeBarrierAngle > 0, i.e. not fully lowered. The end of this condition (i.e. degreeBarrierAngle == 0) defines startGateDown (barrier fully lowered) and the start of the condition defines endGateDown.

As with the original Java-MaC railroad crossing implementation, four variables are monitored in the JOP implementation in order to define an inCrossing condition in the PEDL specification. Although the variable names and class references differ, the variables contain the same information:

- **rrcmodel.TrainModel.metresPosition**: position of the train in metres
- **rrcmodel.TrainModel.metresLength**: length of the train in metres
- **rrcmodel.GateModel.metresPosition**: position of the crossing in metres
- **rrcmodel.GateModel.metresWidth**: width of the crossing in metres

The condition inCrossing is defined as ‘train is past the start of the crossing and train position is before that position which would see the train clear of the crossing’.
7.5 RESULTS FROM IMPLEMENTING AND EXECUTING SCENARIO

In this section, the results gathered from implementing and executing the railroad crossing scenario on the JOP platform are presented under three categories:

- Ability to replicate original scenario
- Extent of preparations for verifying the application
- Overhead of the monitoring approach

These results allow the evaluation of the monitoring approach in terms of its effectiveness for run-time verification of an application executing in an embedded system.

7.5.1 Ability to replicate original scenario

It was possible to retain the same form of MEDL specification for the JOP implementation as for the original used as a Java-MaC example, thus verifying the same safety property in both versions of the railroad crossing scenario. Nevertheless, the application implementing the scenario had to be structured differently, due to the different programming paradigms in using the JOP API as opposed to the standard Java API. Specifically, as detailed in section 7.4.1, the application form, thread model and signalling methods all differed. These differences are summarised in table 7.1.

```plaintext
MonScr RailRoadCrossing

/* Primitive events exported to run-time checker*/
export event startInCrossing, endInCrossing,
       startGateDown, endGateDown;

/* Monitored entity declarations */
monobj int rrcmodel.GateModel.degreeBarrierAngle;
monobj int rrcmodel.TrainModel.metresPosition;
monobj int rrcmodel.TrainModel.metresLength;
monobj int rrcmodel.GateModel.metresPosition;
monobj int rrcmodel.GateModel.metresWidth;

/* Event and condition definitions */
event startInCrossing = start(inCrossing);
event endInCrossing = end(inCrossing);
event startGateDown = end(rrcmodel.GateModel.degreeBarrierAngle > 0);
event endGateDown = start(rrcmodel.GateModel.degreeBarrierAngle > 0);

condition inCrossing =
   rrcmodel.TrainModel.metresPosition > rrcmodel.GateModel.metresPosition
   && rrcmodel.TrainModel.metresPosition <
   rrcmodel.GateModel.metresPosition + rrcmodel.GateModel.metresWidth +
   rrcmodel.TrainModel.metresLength;
End
```
In section 7.4.5, it is noted that where the original Java-MaC example relies on monitoring two methods to define the gate down/up events in the PEDL specification, the JOP implementation relies instead on the value of a variable. This difference arose again due to issues in arranging the JOP implementation. The skeleton application approach described in the preceding chapter provides a surrogate target application for Java-MaC to verify. This approach involves copying the class structure in order that variables being monitored are accessible to Java-MaC as if they were in the target application. However, arranging the same for monitored methods, is not as straightforward, as the skeleton application must copy the control flow of the original application (e.g. when a ‘start of method’ observation is read by the skeleton application, it would have to enter a method of the same name in order that Java-MaC, in monitoring the target, would receive the same observation). Although this implementation was considered, it was abandoned due to the need to manually arrange both duplicate control flow and the function of reading observations and storing them. Thus the approach was taken of monitoring another variable rather than monitoring methods.

### 7.5.2 Extent of preparations for verifying the application

In order to perform the necessary monitoring for run-time verification, it was necessary, as described in section 6.4.3 of the preceding chapter, to add print statements at each location in the application code where a monitored entity had any assignment performed on it. Monitoring the five variables required the addition of code at three locations in TrainModel (two of those locations at the initialisation of the variables, one at the update of the train position) and five locations in GateModel (three of those being at the initialisation of the variables, one at the update of the barrier angle when raising barriers, one at the update of the barrier angle when
lowering barriers). An example of the additional code is shown in figure 7.11 where the TrainModel class has code added at the points where \texttt{metresPosition} is updated.

```java
//NO INSTRUMENTATION:
TrainModel {
  run() {
    while (metresPosition < METRES_TOTAL) {
      metresPosition = (mpersSpeed * msTimeRunning)/1000;
      waitForNextPeriod();
    }
  }
}

//WITH INSTRUMENTATION:
TrainModel {
  run() {
    while (metresPosition < METRES_TOTAL) {
      metresPosition = (mpersSpeed * msTimeRunning)/1000;
      System.out.print("01 ");
      System.out.println(metresPosition);
      waitForNextPeriod();
    }
  }
}
```

Figure 7.11 – Simplified code fragment showing instrumentation

It was also necessary to add two lines of code at each of these locations, rather than a single print statement at each location. The first statement prints the reference of the variable used by Java-MaC (0..n for n variables monitored) and the second prints the value of the variable. Initially a single print statement was used to send these two details to the serial port, but combining these statements resulted in unnecessary dynamic creation of temporary StringBuffer variables at run time. This quickly exhausted the memory available to JOP (that is, without garbage collection). Using the two statement example shown in figure 7.11, this problem was avoided. Using a StringBuffer object as a buffer for observations would allow a single print statement, however, the overhead of string manipulation would be greater than that of executing the second print statement.

As multiple threads have these print statements added (Gate and Train), there exists potential for output from the application to have errors if for example Train is pre-empted between print statements and Gate then executes its print statements. However, this is avoided by having periods defined for each class extending RTThread (including Train and Gate) such that pre-emption does not occur due to the short execution time of the threads and sufficiently generous periods.
7.5.3 Overhead of the monitoring approach

In adding to the target application code, the behaviour of the application may be affected if the overhead of executing the additional code is sufficiently large as to affect the timely execution of elements of the target application. Each line of Java code is translated into single or multiple bytecodes when the application is compiled. As described in chapter 5, the JOP core is implemented in such a way that each bytecode takes a set number of processor cycles to execute. By examining the bytecode added to the target application by each insertion of print lines (as described above) the overhead of the additional code can be quantified.

The bytecode resulting from the compilation of the code fragments of figure 7.11 is shown in figure 7.12. The assignment of a new value to $\text{metresPosition}$ by the statement $\text{metresPosition} = (\text{mpersSpeed} \times \text{msTimeRunning})/1000$; corresponds to bytes 11 to 24 of the bytecode in both columns. The $\text{waitForNextPeriod}$ statement corresponds to bytes 27 and 28 of the column on the left and bytes 45 and 46 on the right. The extra bytecode between bytes 24 and 45 in the righthand column corresponds to the extra code added for monitoring $\text{metresPosition}$, i.e. the two print statements.

```
//NO INSTRUMENTATION:
public class TrainModel extends RtThread {
    public void run() {
        // ...
    }
}

//WITH INSTRUMENTATION:
public class TrainModel extends RtThread {
    public void run() {
        // ...
    }
}
```

Figure 7.12 – Bytecode fragments from RRC application showing instrumentation

The execution times taken for each bytecode executed on the JOP core are listed in Appendix D of the JOP Handbook by Schoeberl [84]. This timing information is
presented in table 7.2 for each bytecode used in the additional bytecode of bytes 27 to 44 of the right column of figure 7.12.

In the reference execution times, r and w refer to the cycles taken for read and write wait states. These are implementation dependent. For the Spartan 3 FPGA used for the experimental work, both r and w are two cycles. The cycles taken for cache load are also referred to, as l. The JOP core is designed to cache entire methods, relying on the common feature of Java code to have many small methods. In the calculations of table 7.2, it is assumed that there is a cache hit. This corresponds to four processor cycles. The total processor cycles to execute all the bytecodes listed in table 7.2 are 287. With the 50 MHz clock of the Spartan 3, this corresponds to 5.74 µs of execution time. Additionally, the calculations for execution time must include the cycles taken to execute the bytecodes of the methods invoked (System.out.print(String), System.out.println(int)) and any nested invocations. Using WCET analysis of the code used to implement these methods, the invocations take 6312 cycles combined to execute (126.24 µs) with the parameters used in the railroad crossing scenario, resulting in a total execution time of 131.98 µs.

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Reference time</th>
<th>Actual cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>27:getstatic</td>
<td>12+2*r</td>
<td>12+2*2</td>
</tr>
<tr>
<td>30:ldc1</td>
<td>7+r</td>
<td>7+2</td>
</tr>
<tr>
<td>32:invokevirtual</td>
<td>100+4*r+1</td>
<td>100+4*2+4</td>
</tr>
<tr>
<td>35:getstatic</td>
<td>12+2*r</td>
<td>12+2*2</td>
</tr>
<tr>
<td>38:aload_0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>39:getfield</td>
<td>17+2*r</td>
<td>17+2*2</td>
</tr>
<tr>
<td>42:invokevirtual</td>
<td>100+4*r+1</td>
<td>100+4*2+4</td>
</tr>
</tbody>
</table>

Table 7.2 – Execution time and cycles for additional bytecodes

The print statements themselves take approx 0.1 ms to execute per character printed, as a blocking wait is used in the implementation of the function writing to the serial port. No handshaking is used in the serial communication. The length of the blocking wait depends on the serial port characteristics. In the case of JOP, the serial port operates at 115,200 baud (i.e. 11,520 chars/sec) resulting in the delay of approx 0.1 ms given that there is one start bit, one stop bit and an assumption of approx one bit spacing between messages. The print statements in figure 7.11 result in six to nine characters being printed to the serial port (depending on how many digits compose the
value of metresPosition). This results in an execution time of 0.6-0.9 ms approx, plus the bytecode execution time of 132 µs for a total execution time of 0.732-1.032 ms.

7.6 CONCLUSIONS FROM RAILROAD CROSSING SCENARIO

Implementing for the JOP platform a version of the railroad crossing scenario used as a Java-MaC example allowed the objectives set out in section 7.2 to be met. Particularly, using this scenario allowed evaluation of the effectiveness of the monitoring approach for run-time verification of applications executing in an embedded system.

By utilising the same scenario with verification of the same safety property, the monitoring approach could be compared with the normal deployment of Java-MaC. The differences in implementing the actual application used for the scenario served to fulfil the objective of implementing an application with some attributes typical of embedded applications. Specifically, the implementation for the JOP platform incorporated real-time functionality and a more basic programming environment.

7.6.1 Evaluation of approach based on results

The difficulty in arranging monitoring of a method, as used to facilitate the original run-time verification example, showed some limitations to the monitoring arrangement presented in the preceding chapter. However, the overall approach of an intermediate application did allow Java-MaC to indirectly verify the target application on the JOP core. Therefore a modified approach to implementing the intermediate application was considered.

The inclusion of monitoring code in the target application was fairly straightforward, following the sequence outlined in section 6.4.3 of the preceding chapter. In theory, the target application could be instrumented by this code in the same fashion as Java-MaC performs instrumentation of application bytecode. Fully automatic instrumentation would be preferable than the manual approach used, as there is the possibility of manual error.

As demonstrated by the results and analysis of section 7.5.3, the overhead of executing additional bytecode is not the main overhead in the monitoring approach. The blocking wait on sending output to the serial port does add significant overhead to the execution of the target application and being in the order of hundreds of
microseconds on each instance assignments are made to monitored variables, could potentially disrupt timing (the threads described in section 7.4.4 were scheduled as tasks with periods measured in milliseconds). However, it is intended that the monitoring approach would in practice not use the serial port, but rather a dedicated debug port. Assuming no communication delay (e.g. memory-mapped I/O) and arranging the existing API print methods to use this communications port rather than the serial port, the overhead of executing the monitoring statements as presented in section 7.5.3, would simply be of a similar order of execution time as the existing bytecodes that have execution time of 132.46 µs per monitoring statement.

7.7 REVISED EXPERIMENTAL CONFIGURATION

As mentioned in the conclusions above, the monitoring approach of using a ‘skeleton’ application offered limited functionality as an intermediate stage between the application on the JOP core and JavaMaC on the host, due to the need to manually duplicate method structure and control flow of the target application in order to monitor methods of the target application. In revising this approach, consideration was given to modifying JavaMaC. However, this was ruled out due to the scale of the development effort that would be required. Instead, attention was turned to the means in which JavaMaC instruments an application (as the intention was to duplicate the behaviour that would arise if JavaMaC instrumentation was present in the target application on the JOP core).

As described in chapter 4, the code added to the target application by Java-MaC consists of a filter thread that emits events from the target application via a TCP socket to the event recognizer and function calls to the filter thread added to the target application's own execution thread at points where monitored entities occur. In the revised approach for using Java-MaC to verify an application on JOP, the intermediate application is implemented such that it is no longer instrumented automatically by Java-MaC. Instead, code to create the Filter thread is manually coded into the intermediate application and calls to the Filter thread are also manually placed. This approach is termed the ‘relay application’ by the author.

7.7.1 Implementation of the relay application

Referring back to section 6.4 of the preceding chapter, the author considers the three implementation issues as relevant to the relay application:
• Reading from serial port

The approach outlined in section 6.4.1 remains unaltered

• Storing observations in a fashion ready for Java-MaC

Observations are not stored, but rather relayed directly to Java-MaC

• Integration with Java-MaC

The steps for integrating the relay application with Java-MaC are simplified from those used with the skeleton application

The relay application operates as shown in figure 7.13. At the start of execution, a Filter thread is initialised. The relay application then enters a loop to read input. For each line of input read, the input is parsed and a switch statement chooses, based on this input (denoted A-Z in the figure), what call to make to the Filter thread.

![Figure 7.13 – Execution behaviour of the relay application](image)

7.7.2 Implementing the revised configuration

The process for implementing the revised configuration for using Java-MaC to verify an application with JOP is similar to that described in section 7.3. The target application is coded normally and a copy archived. The static phase of Java-MaC is
performed on this application. The amended bytecode is examined to determine the points at which instrumentation has been added (i.e. where monitored entities are present in the application code). For the purposes of the relay application, it is noted specifically what identifying reference numbers Java-MaC has allocated to each monitored entity. As was the case with the skeleton application approach used in section 7.4, a version of the target application code is created with print statements (comprising the Java-MaC reference numbers for monitored entities and the new value/state of the monitored entity). These print statements are manually inserted at the monitoring points noted in the bytecode instrumented by Java-MaC. This version of the target application is then used for execution on JOP.

Unlike the previous approach, the static phase of Java-MaC is not rerun. Instead, the relay application takes the place of the target application on the host PC (i.e. it has the same class name and location as the target application code used during the static phase of JOP).

The complete revised deployment of Java-MaC for verifying an application executing on JOP is shown in figure 7.14. As with the previous approach, print statements are added to the target application (1), resulting in observations received on the serial port by the JOP download tool (2) and piped to the relay application (3). The observations are sent to the filter thread (4) using the mechanism described in the previous section.

Figure 7.14 – Revised arrangement for using Java-MaC with JOP

The complete revised deployment of Java-MaC for verifying an application executing on JOP is shown in figure 7.14. As with the previous approach, print statements are added to the target application (1), resulting in observations received on the serial port by the JOP download tool (2) and piped to the relay application (3). The observations are sent to the filter thread (4) using the mechanism described in the previous section.
The filter sends observations to the event recognizer using a TCP socket (5) and the event recogniser exports events to the run-time checker (6).

7.8 SUMMARY

This chapter described the implementation of a test case to evaluate the effectiveness of the monitoring arrangement for run-time verification of an application executing in an embedded system. The implementation for the JOP platform of a railroad crossing example used to illustrate Java-MaC’s MEDL and PEDL languages was described. The structure of the original example for Java-MaC was described, as was the verification performed by Java-MaC in the original example. An implementation of this scenario for JOP was introduced and the differences between it and the original example were explained. The particulars of how the target application was prepared for JOP and Java-MaC have also been given. Results were presented, detailing the extent to which the original scenario was implemented on JOP, the extent of preparations for run-time verification, and the overhead of the monitoring approach. The conclusion was reached that implementing the railroad crossing scenario for the JOP platform allowed the objectives to be met. An evaluation of the effectiveness of the monitoring approach is presented based on the results of implementing and executing the scenario. Finally an amended monitoring arrangement is presented, following on from one of the conclusions concerning the limitations of the previous monitoring arrangement. This new approach of a relay application is detailed.
CHAPTER 8 – RATE MONOTONIC

SCHEDULING TEST CASE
8.1 INTRODUCTION

This chapter concerns the second test case proposed in chapter 6: exploring the effectiveness of the Java-MaC/JOP arrangement with a verification task that could potentially be of use in a typical embedded system. Some introduction is provided concerning the rate monotonic algorithm and the proposed verification task. The particulars concerning implementation of the scenario are described. Use of a worst-case execution time (WCET) analysis tool to develop the target application is detailed. The implementation of the verification task is also described. Analysis of timing details for elements of the experimental framework is presented in order to examine the feasibility of the verification task. Results demonstrating these characteristics of the experimental apparatus are presented, comprising experiences from performing WCET analysis and executing the target application. Based on the analysis and results, conclusions are made concerning the general viability of the proposed monitoring approach for such a verification task. A revised monitoring strategy is proposed in order to achieve a viable approach for such a verification scenario.

8.2 AIM AND OBJECTIVES

The aim of this second test case is to further explore the effectiveness of the monitoring approach for run-time verification of applications in an embedded system. Specifically, the test case objectives are:

- Development of a real-time embedded system scenario to which run-time verification can be applied.
- Use of static analysis to develop the implementation of the scenario and provide a benchmark for verification.
- Investigate verification of timing parameters of multiple tasks.
- Analyse the experimental prototype and characterise the extent to which it can be used to perform such verification.

It was anticipated that these objectives would allow evaluation of the effectiveness of the proposed monitoring approach with specific reference to the usefulness of run-time verification for a real-time embedded application.
8.3 BACKGROUND

Software tasks in a multi-threaded embedded system are often scheduled according to the rate-monotonic scheduler (RMS) algorithm. Under RMS, each task is given a fixed priority according to its period, with shorter tasks having higher priority. It is the optimal scheduling algorithm for periodic tasks with hard deadlines and fixed priorities, as demonstrated by Liu and Layland [85], who proved that a set of periodic tasks with deadlines equal to period can be scheduled with fixed priorities under RMS if processor utilisation is below a set bound. The equation for this least upper bound on processor utilisation, also known as the schedulability equation, is shown in figure 8.1.

\[ U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left( \sqrt{2} - 1 \right) \]

Figure 8.1 – Schedulability equation for RMS

\( U \) refers to the processor utilisation, equal to the sum for all \( n \) tasks (\( i=1…n \)) of their computation time (\( C_i \)) divided by their period (\( T_i \)). Throughout the rest of this chapter, computation time will be referred to as execution time. The least upper bound on processor utilisation was proven by Liu and Layland [85] to be \( n \left( \sqrt{2} - 1 \right) \) for \( n \) tasks. As \( n \) tends to infinity, this tends to ln2 (approx. 0.69). As long as utilisation is below this boundary, the task set will be schedulable.

It is however possible to schedule a set of tasks with higher utilisation than this, as characterised by Lehoczky et al [86]. For such a set of tasks, whose utilisation is above that given by the equation of figure 8.1, further analysis is required to determine if the task set is schedulable.

8.4 REAL-TIME EMBEDDED SYSTEM SCENARIO

In order that analysis of scheduling is valid, such as that required in the case of tasks with total utilisation greater than 0.69 approx., it is necessary that the scheduler in the system behave as expected and with minimal overhead in pre-empting tasks. In order to examine the effectiveness of the monitoring approach for run-time verification of an application in an embedded system, in this chapter a real-time embedded scenario is proposed where a set of tasks has high total utilisation (i.e. > 0.69), but yet are
schedulable according to static analysis. It is proposed that run-time verification can be used to determine if the tasks are scheduled as anticipated on the JOP core.

A scenario of scheduling three tasks under the rate monotonic algorithm was considered. To simplify the scheduling problem, the two longer tasks of the scenario were given periods that were double and quadruple the period of the shortest task. It was envisaged that tasks would execute for multiples of this same time unit, again to simplify analysis of the scheduling. The tasks were arranged such that most of the available processor time would be allocated, in order that any overrun of the intended execution time at run time could be detected. The periods and execution times allocated to the tasks are shown in table 8.1. The three tasks T1, T2 and T3 were given periods of 100 ms, 200 ms and 400 ms, with intended execution times of 25 ms, 100 ms and 75 ms respectively.

<table>
<thead>
<tr>
<th>Task:</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period (ms):</td>
<td>100</td>
<td>200</td>
<td>400</td>
</tr>
<tr>
<td>Execution time (ms):</td>
<td>25</td>
<td>100</td>
<td>75</td>
</tr>
</tbody>
</table>

Table 8.1 – Task parameters for the scenario

Monitoring the actual execution time of the tasks at run time provides observations for verification of the predicted execution times. Verification that the run-time execution times correspond with the predicted execution times can confirm that the tasks remain schedulable. By contrast, if any task exceeds the predicted execution time, then even if the target application continues to execute normally in the short term, one can assume that the task set may not remain schedulable in the future.

In order to implement this scenario for the JOP platform, the following issues must be considered:

- Scheduling of tasks by the JOP core.
- Implementation of tasks using the JOP API.
- Arranging fixed execution times for tasks according to table 8.1, using static analysis to predict execution time of application code.
- Specification of parameters for run-time verification.
Additionally, the following aspects of the experimental prototype must be considered:

- The methods for timing events
- Delays due to the tool chain for the prototype

Investigation of an implementation of the scenario and results from executing the implementation can be used to demonstrate the characteristics of the experimental prototype and reason about the effectiveness of the monitoring approach.

8.4.1 Using the JOP scheduler

The JOP core includes a built-in scheduler. As described by Schoeberl [18], this scheduler is pre-emptive and priority-based, with unlimited priority levels and a unique priority level for each object that can be scheduled. A priority-based scheduler chooses which task to execute based on its priority, scheduling the highest priority task. A pre-emptive scheduler will pre-empt the currently executing task if a task of higher priority is released. Pre-emption involves suspending execution of the current task, with execution of the suspended task being completed later when task(s) of higher priority have been executed (i.e. when the suspended task once again has highest priority of tasks available for execution).

If priorities are assigned to tasks on the basis of their periods, then the JOP scheduler will behave as a rate monotonic scheduler. By assigning priorities on such a basis, the tasks described in the preceding section, with parameters as outlined in table 8.1, would be scheduled as depicted in the graph of figure 8.2. T1 with shortest period has
the highest priority, T2 the next priority, and T3 the lowest priority (as it has longest period). This means that tasks T2 and T3 would be pre-empted.

With an “ideal” scheduler that has no overhead from context switching, as shown in the graph of figure 8.2, the three tasks should be schedulable despite a high processor utilisation of 93.75%. However, for any real scheduler, the overhead of context switching needs to be taken into account. For the JOP platform, a context switch time of almost 3000 processor cycles has been observed from testing by Schoeberl [18]. This corresponds to 60 μs on the Spartan 3 platform used for this project. By analysing the schedule in figure 8.2 and adding context switch time to the task execution time, a processor utilisation reflecting context switch time is 93.89%.

Another consideration with regard to scheduling is the potential for “priority inversion”. This is a situation that arises when a low priority task acquires exclusive access to a shared resource. If pre-emption of the lower priority task occurs during this time, a higher priority task may be unable to execute without access to the shared resourced, forcing execution of the lower priority task (i.e. the priorities are essentially reversed). To avoid this situation, JOP implements the priority ceiling emulation protocol as detailed by Schoeberl [18], where a shared resource is assigned its own unique high priority.

8.4.2 Implementing tasks using JOP API

As described by Schoeberl [87], objects schedulable by the JOP core scheduler are implemented as classes in the JOP API. Two types of object are used to model concurrent threads of execution, periodic tasks and asynchronous sporadic tasks (represented by event handlers). These are implemented using the classes \texttt{RtThread} (for periodic tasks), \texttt{HWEvent} and \texttt{SWEvent} (event handlers for hardware and software events respectively), each of which acts as a thread of execution in the application. The \texttt{RtThread} class is sufficient to implement the proposed scenario that exclusively comprises periodic tasks.

Using the \texttt{RtThread} class, periodic tasks are modelled using an infinite execution loop, with invocations of a method \texttt{waitForNextPeriod} made to the scheduler to request rescheduling in the next time interval. The period of the task is set upon creation of an instance of a class extending \texttt{RtThread}. The typical structure of an application for the JOP platform is shown in figure 8.3. On the left is the class
implementing the application and on the right, an implementation of a periodic task for that application. An application on the JOP platform has two phases, an initialisation phase and a mission phase. All classes in the target application have to be loaded and initialised during the initialisation phase, as dynamic creation of objects or threads is not possible. The mission phase is started by an invocation of `RtThread.startMission()`. The work of periodic tasks is included in a method run, which is invoked when this mission phase of the application begins.

```java
/* Class implementing the application */
TaskSet {
    /* Declaration */
    WorkTask taskA;
    /* Application main method*/
    main() {
        /* Initialization phase */
        taskA = new WorkTask(PRIORITY, PERIOD);
        /* Start of mission phase */
        RtThread.startMission();
    }
}

/* Task implemented as RtThread */
WorkTask extends RtThread {
    /* Task work method */
    run() {
        /* Infinite execution loop */
        for (;;) {
            /* Some work */
            doWork();
            /* Reschedule for next period */
            waitForNextPeriod();
        }
    }
}
```

Figure 8.3 – Structure of an application using RtThreads

For the remainder of this chapter, the term `thread` refers to a thread of execution in an application executing on the JOP core. The term `task` is used to refer to the model of a task that is implemented as a class extending the `RtThread` class.

### 8.4.3 Planning task execution times

In order that the tasks in the scenario have execution times as outlined in table 8.1, it is necessary that the classes implementing the tasks have a mechanism to adhere to a pre-set execution time at run time. The code for the run method of the thread can be constructed in a way that parameters in the execution code (e.g. bounds on loops) can be used to determine the execution time.

In order to ensure that the execution time of the application code for the thread matches that used in the task model, static analysis is used. The worst-case execution time (WCET) tool described by Schoeberl and Pedersen [79] is used to analyse application code and determine the exact parameters required in the application code to provide a particular task execution time.
This WCET analysis tool analyses a given method of a Java application after it has been compiled to bytecode and readied for the JOP core and determines the worst-case execution time for the JOP platform. An example of WCET analysis is presented by Schoeberl [88], where a bubble sort algorithm is analysed. Performing this analysis on a section of Java code to be executed on the JOP platform is possible with certain restrictions such as no unbounded recursion and an upper bound on loops.

8.4.4 Verifying task execution time

Rather than monitor a clock in the target system to determine if timing is correct, Java-MaC is given independence in verifying the timing by utilising the host system time and having the run-time checker component reason about time through the use of auxiliary variables and a construction time(event) that provides the time that an event occurs in the filter thread (i.e. in the target application). Using this representation of time, for each task one can specify that there exists a violation if the task execution time is greater than the execution time laid out for the scenario (the values listed in table 8.1). This predicted execution time is termed the worst-case execution time (WCET) in the specification.
In order to verify the execution time of a task, the MEDL specification relies on two primitive events corresponding to the start and end of the task's execution. The MEDL specification for verifying the task T1 is shown in figure 8.5, where the primitive events $startT1$ and $endT1$ correspond to the start and end of the task execution. In order to reason about the actual times involved, two auxiliary variables are used: $startTimeT1$ to represent the time that task execution started and $execTimeT1$ to represent the execution time elapsed. When the $startT1$ event occurs, $startTimeT1$ is assigned the time of that event and $execTimeT1$ is initialised to zero. When the $endT1$ event occurs, $execTimeT1$ is assigned the time of that event less $startTimeT1$, the time that it started.

A condition $withinWCETT1$ is defined to specify if the execution time is less than or equal to the boundary set on the task's execution, i.e. less than or equal to 25 ms as specified in table 8.1. This condition is the basis for specifying a violation. The alarm $beyondWCETT1$ is fired as soon as the condition $withinWCETT1$ ends, i.e. as soon as the execution time is no longer within the predicted WCET.

**8.4.5 Verifying execution times with pre-emption**

Although the verification approach described in the preceding section allows the execution time of a single task to be monitored, the verification of multiple task execution times is more complex due to pre-emption. If no amendment is made to the approach used for monitoring a single task, then the execution times monitored would include any time that a task had been suspended. As stated in section 8.4.1, when a
task is pre-empted, its execution is suspended until that task is once again the task of highest priority that is ready to execute. In order to reason about execution times excluding the time interval for which a task is suspended, additional auxiliary variables have to be incorporated into the MEDL specification to allow the duration that a task is suspended for to be included in calculations.

Figure 8.6 – Task schedule for T1 and T2 with annotation of times and intervals

Part of the anticipated schedule for the scenario, including tasks T1 and T2, is shown in figure 8.6, with annotation of the auxiliary variables used by the MEDL specification.

In addition to startTimeT1 and execTimeT1 referred to in the previous section, the equivalent variables for task T2 (startTimeT2, execTimeT2) are noted. The complete specification also includes startTimeT3 and execTimeT3 for task T3. The additional auxiliary variables required to calculate the duration that task T2 is suspended for are startSuspTimeT2 (start of when T2 is suspended) and suspTimeT2 (length of time T2 is suspended for). For the complete specification the variables startSuspTimeT3 and suspTimeT3 are defined similarly. The complete MEDL specification for the taskset scenario is listed in figure 8.7.
The auxiliary variables `startSuspTimeT2`/`startSuspTimeT3` and `suspTimeT2`/`suspTimeT3` have to be updated at the instant a task is suspended, or returned to execution respectively. To this end, a number of conditions are defined to allow those instants (or events) to be defined. The conditions `startedT1`, `startedT2` and `startedT3` are each specified as the interval between when the task starts and finishes execution. This allows the condition that a task is suspended to be defined. The condition `suspendedT2` is true (i.e. task T2 is suspended) if task T2 is started but task T1 is also started (as T1 is higher priority, T2 will be or will have been suspended). Similarly for `suspendedT3`, if task T3 is started and task T2 or T1 are started, then T3 is suspended.
With these conditions defined, the events corresponding to the instant a task is
suspended or returned to execution can be defined. The event startSuspendedT2 is
defined as the event startT1 (i.e. start of task T1) occurring when T2 is already
started (startedT2). The event endSuspendedT2 is defined as the event endT1 (i.e.
end of task T1) occurring when T2 is already started. The events startSuspendedT3
and endSuspendedT3 are defined similarly.

These events are used to update the auxiliary variables. When startSuspendedT2
occurs, the variable startSuspTimeT2 is updated to be the time that the event
occurred. When endSuspendedT2 occurs, suspTimeT2 (i.e. length of time T2 has
been suspended) is assigned the previous value of suspTimeT2 (in case of previous
suspended intervals) plus the time of the endSuspendedT2 event, minus the time that
T2 was suspended at (startSuspTimeT2). The T3 auxiliary variables are similarly
updated when the T3 events occur.

With the updating of the suspTimeT2 and suspTimeT3 variables, the execution
times of the tasks can be calculated excluding any time that the tasks were suspended
for. As in the simpler single task example, the execution time is updated on
occurrence of the end task event (endT1, endT2, endT3). Execution time of T2 for
example is calculated by subtracting startTimeT2 (start time of task T2) minus the
time that the endT2 event occurred and minus the time that T2 was suspended
(suspTimeT2).

These calculations allow run-time verification to determine if the tasks exceed the
predicted worst-case execution time, irrespective of whether the tasks are pre-empted
or not.

8.4.6 Time in Java-MaC and host platform

The Java-MaC MEDL construct time(event) is intended to provide the time of an
event in milliseconds. An examination of the code implementation for the Java-MaC
run-time checker reveals that this relies on the Java API method
System.currentTimeMillis(). It is noted in the Java API documentation [89] that
although this method returns the current time in milliseconds, the actual measurable
interval using this method may be more than one millisecond. The reason for this is
that although Java code may be executed on Java Virtual Machines that are running
on various platforms, the implementation of methods depends on native features (in
the case of System.currentTimeMillis, the default system clock). Depending on the platform, the granularity of this may not necessarily be milliseconds. This issue is considered in the analysis of the experimental prototype and an execution of the implemented scenario also demonstrates the characteristics of this.

As regards the granularity of the underlying system clock, Kuperberg et al [90], in evaluating an algorithm ‘TimerMeter’ for quantifying accuracy and invocation cost of various software timing methods, report a resolution of 15 ms for System.currentTimeMillis under Microsoft Windows XP. Holmes [91] links this to the use of the Windows GetSystemTimeAsFileTime API method that is used to implement System.currentTimeMillis and suggests values of 10 ms or 15 ms for the resolution, depending on the platform. The GetSystemTimeAsFileTime returns the Windows System Time, which is stated by Microsoft [92] to be the current UTC time of day as reported by the real-time clock and updated at regular intervals. As noted by Russinovich and Solomon [93] in a discussion of thread scheduling by NT-based Windows, the clock interval is hardware-dependent and generally about 10 ms for single processor x86 machines but 15 ms for multiprocessor x86 machines (as was the case with the primary host machine used in the testing described in this thesis). Also described is the use of an application ClockRes by Russinovich [94] to determine current system clock resolution. As regards the experimental prototype described here, this application can be used to determine the clock resolution in use by the host system.

Another more recent Java API method providing higher precision timing can be used in any new Java code to be executed on the host system. Since Java version 1.5 (Java 2 Standard Edition 5.0), released in late 2004, a newer API method System.nanoTime has been available. As explained in the Java API documentation [89], System.nanoTime returns the value of the highest precision system timer available on the host platform for the JVM. In theory this means that sufficient resolution should be available for millisecond timing.

8.4.7 Time on the JOP platform

The time of events can also be considered on the JOP platform rather than utilising Java-MaC run-time checker for timing events. Unlike a Java virtual machine (JVM), applications executing on the JOP core have direct access to hardware resources. This
is accomplished using a JOP API library, com.jopdesign.sys.Native, which encapsulates ‘native’ operations. These operations are mostly provided for the use of other JOP API libraries, e.g. the JOPPrintStream class which provides the **print** and **println** methods by utilising the **wr** method (write a character to a given IO address) provided by com.jopdesign.sys.Native.

For the purposes of adding timestamps, an application executing on the JOP platform can invoke the **rdMem** method of class Native, a method which reads from a given memory address. The library com.jopdesign.sys.Const includes members that store integer references for memory addresses. The **IO_CNT** member references a memory-mapped IO address that is linked to a counter on the FPGA, storing the number of cycles executed. This provides a means for an application executing on the JOP platform to retrieve the current time in terms of cycles executed (50 MHz clock = 20 ns per cycle).

### 8.4.8 Communication delays

Another aspect of the experimental monitoring apparatus that needs to be considered in the context of verifying timing parameters is the delay in communicating events from the application on the target system to the verification engine on the host system. In order to fully consider the feasibility of the proposed verification task, the characteristics and effects of this communication delay need to be analysed. The communication chain for sending observations from the target application to the runtime checker on the host system is shown in figure 8.8.

![Diagram](image)

**Figure 8.8 – Communication chain from target to host**

After an event occurs (assignment to a monitored variable, start or end of a monitored method) a print statement is executed by the target application. The message is sent over a serial link to the host system. An application on the host system reads the serial port and echoes to standard output. This is piped as input to the Java relay application.
The observations are parsed from this input stream and calls are made to the filter thread attached to the relay application, passing the observations as arguments to a method in the filter thread. The filter thread timestamps and passes the observations over a TCP socket to the Java-MaC event recogniser. Any state change in primitive conditions, or occurrence of primitive events, is communicated from the event recogniser to the run-time checker. The run-time checker uses the timestamps from the filter thread in any calculations involving event times.

8.5 IMPLEMENTATION OF THE SCENARIO AND TESTING

Each of the tasks for the scenario described in this chapter was implemented as a class extending RtThread, with periods specified at initialisation as in table 8.1. The priorities were assigned according to the rate monotonic algorithm, i.e. highest priority for T1 and lowest for T3. In addition to accepting a period and priority at initialisation, the class implementations were constructed to accept an execution time, i.e. a duration for which the thread should execute.

In order to ensure the tasks execute for this specified duration, to match the scenario, a code construct was needed. Two potential methods of providing a pre-determined execution time are described: using a timer to set a bound on the execution of a loop, or using nested loops. These are explained in terms of the constructs used and how WCET analysis is performed. The impact of pre-emption of the executing thread is considered and is used as a basis for the chosen implementation.

8.5.1 Code structure for using timers

The class TaskSet represents the whole scenario described in section 8.4, forming the application that encapsulates the scenario. A main method in this class initialises other classes and invokes RtThread.startMission(). Three other classes, TaskShort, TaskMedium and TaskLong were created, extending the RtThread class in order to act as real-time threads. These classes correspond to tasks T1, T2 and T3 respectively.

The arrangement of an execution loop that iterates until a pre-set timeout expires is incorporated into the run method of each of the three task classes. In order to check for expiry of a timeout as the loop exit condition, a JOP API class is used. This class, Timer, implements some simple timer and watchdog handling. Two methods are used, Timer.getTimeoutMs(int) and Timer.timeout(int). The getTimeoutMs method takes as
an argument an integer representing an interval in milliseconds and returns an integer representing the future point in time (in microseconds) after which the interval will have elapsed. The timeout method takes an argument representing a time (in microseconds) and returns a boolean value of true or false depending on whether the system time is now greater than or less than that time.

Figure 8.9 – Using Timer class

The use of the Timer class is demonstrated in figure 8.9. At time \( \text{start} \), Task invokes Timer method \text{getTimeout} \ with a parameter \text{msOffset}, used to specify that the timeout will occur after an interval \( \text{offset} \). The value \text{usTimeout} is returned, representing the time \( \text{start} + \text{offset} \) in microseconds. If Task invokes the Timer method \text{timeout} \ with the parameter \text{usTimeout} before \( \text{offset} \) has elapsed, a value of \text{false} \ will be returned. When Task invokes the \text{timeout} \ method after \( \text{offset} \) has elapsed, a value of \text{true} \ will be returned, signifying that the time \text{timeout} has occurred.

The application code creating the three tasks and the code for TaskShort, one of the tasks using the Timer class, is shown in figure 8.10. Execution times in milliseconds are provided as one of the parameters for each task class initialisation, along with the priority and period. A method \text{doWork} \ implements the ‘work’ of the task in the model using a while loop and the Timer methods are used to set a timeout as the exit condition for the loop.

As is demonstrated in the example of figure 8.9, another short interval may elapse between the time \text{timeout} and the time that the value \text{true} \ might be returned from an invocation of the Timer \text{timeout} method. In using this Timer class for the timeout that
ensures threads execute for a specified execution time, account must be taken of this interval, which will be present if the timeout occurs during the final loop iteration. The WCET analysis tool can be used with the application code to determine what this interval may be and thus how much to compensate when specifying a timeout.

```java
/* Class implementing the application*/ /* Task implemented as RtThread */
TaskSet {
    /* Declaration */
    public static TaskShort taskShort;
    public static TaskMedium taskMedium;
    public static TaskLong taskLong;

    /* Application main method*/
    main() {
        /* Initialization phase */
        taskShort = new TaskShort(3, 100 * 1000, 25);
        taskMedium = new TaskMedium(2, 200 * 1000, 100);
        taskLong = new TaskLong(1, 400 * 1000, 75);
        /* Start of mission phase */
        RtThread.startMission();
    }
    /* Task work method */
    run() {
        /* Infinite execution loop */
        for(;;){
            /* Some work */
            doWork();
            /* Reschedule for next period */
            waitForNextPeriod();
        }
    }
    private void doWork() {
        /* Get timeout */
        taskTimeout = Timer.getTimeoutMs(taskTime);
        /* Timeout Loop */
        while (!Timer.timeout(taskTimeout)) {} 
    }
}
```

Figure 8.10 – Application code for TaskSet and TaskShort

### 8.5.2 WCET analysis of timer implementation

In order to perform analysis of the code described in the previous section, a simple Java application can be created that includes the code of the doWork method. The method is first analysed without any timeout loop included, in order to have a baseline WCET for the method. As noted in section 8.4.3, any loops present in the code to be analysed must be annotated. To begin with, the timeout loop in the doWork method is annotated with a worst-case bound of one loop iteration. This is analysed and the result used with the baseline WCET to find the WCET for the first loop iteration. The loop is then annotated with a worst-case bound of two loop iterations in order to find the WCET for each additional loop iteration.
Using this value, it is possible to calculate the actual worst-case bound on the timeout loop, assuming the desired WCET for the task is used as the timeout. To calculate this, the WCET for each loop iteration is divided into the timeout value (minus the WCET for the first iteration). It can be assumed that the loop will iterate for the rounded-up result of this calculation (plus one for the first iteration).

Using this new worst-case bound on the number of loop iterations, the WCET for the method is calculated. The difference between the expected WCET of the method and the desired WCET is then calculated. This difference is used to determine the timeout to apply to the loop to ensure that the expected WCET is not only below the desired WCET, but also as close to it as possible.

8.5.3 Impact of pre-emption on use of timers

As the code constructs for arranging a duration bound on loops rely on timers that calculate whether a timeout comprising a start time plus and offset has elapsed, the basic form described above will not be sufficient for multiple threads of execution where pre-emption occurs. The timer timeout is intended to ensure the work method lasts for a specified duration, however, the construct used will include the time that a thread is suspended.

Similarly to the calculations for the MEDL specification of 8.4.5, the use of timers could be amended to include calculations for the time a thread is suspended based on the state of currently active threads. This would require such calculations to be undertaken outside the suspended thread. However, for the purposes of arranging a pre-determined execution time for tasks, it is more straightforward to have the code determining this to be wholly self-contained within that thread of execution.

8.5.4 Using nested loops to achieve specified execution time

Rather than rely on times provided by the Timer function, executing loops with a fixed number of iterations and some ‘dummy’ work can ensure a fixed amount of execution time for each task regardless of whether it is pre-empted and suspended. In order to provide an implementation suitable for all three tasks, the outer loop is arranged such that each iteration takes exactly one millisecond to execute.
The code for the `doWork` method is shown in figure 8.11. The outer loop executes for a number of iterations specified by `timelimit`, the number of milliseconds that the method execution is intended to take. Two inner loops are used to achieve an execution time of one millisecond per iteration of the outer loop. One of the two loops has a longer execution time per iteration than the other loop, as there is an extra instruction in the ‘dummy’ operation.

### 8.5.5 WCET analysis of loops

The `doWork` method is placed into a test application for WCET analysis. The method is first analysed with a WCET bound of zero loop iterations on the outer loop in order to determine the basic number of cycles for the method. The intention is that each iteration of the loop will add 50,000 cycles (i.e. one millisecond) to this initial number of cycles.

Further analysis is performed to determine how long each iteration of the two nested loops takes. The byte code for the method is shown in figure 8.12. Lines 8 and 17 (in bold) both refer to the loading of the loop bound integer, set to 128 for both inner loops in the application code for the purposes of the WCET analysis. If a smaller value is used, then rather than the instruction `sipush`, which pushes a 16 bit signed integer on the stack (taking three cycles to execute on the JOP core), the instruction `bipush` would be used, which pushes an 8 bit signed integer (>= -128, <= 127) on the stack (two cycles to execute) or for constants 0-5, the instructions `i_const0-i_const5` would be used to push 0-5 on the stack (taking one cycle to execute). As the final loop bounds are to be >127, the code being analysed uses the value of 128 in all circumstances (despite lower bounds being used in the annotation for the WCET analysis).
8.5.6 Verification

In order to perform the verification specified in section 8.4.5, the two primitive events exported by the event recogniser have to be defined, i.e. the start and end of the task execution. The PEDL specification for monitoring TaskShort, TaskMedium and TaskLong, i.e. task T1, T2 and T3 in the model of the scenario, is shown in figure 8.13. To monitor the task execution time, the method doWork() of each implementing class is monitored (this accounts for all of each task’s ‘work’ apart from the overhead of invoking the scheduler with waitForNextPeriod and iterating to the next loop iteration). For example, using the method taskset.TaskShort.doWork(), the event startT1 can be defined as the start of the method and the event endT1 can be defined as the end of the method.

The revised monitoring approach of the preceding chapter, the relay application, can be utilised to provide the necessary observations for Java-MaC. Unlike the earlier skeleton application monitoring approach used for the previous test case of the railroad crossing, the revised approach facilitates monitoring of a method without any complex arrangements in the intermediate application.

Figure 8.12 – Bytecode for doWork method

```java
public void doWork(int i) {
    // 0 0:iconst_0
    // 1 1:istore_3
    // 2 2:iload_3
    //START OUTER LOOP
    // 3 3:iload_1
    // 4 4:icmpge 55
    //OUTER LOOP COMPARE–JUMP
    // 5 7:iconst_0
    // 6 8:istore 4
    // 7 10:iload 4
    //START SHORT LOOP
    // 8 12:sipush 128
    // 9 15:icmpge 27
    //SHORT LOOP COMPARE–JUMP
    // 10 18:iload 4
    // 11 20:istore_2
    // 12 21:iinc 4 1
    //END SHORT LOOP
    // 13 24:goto 10
    // 14 27:iconst_0
    // 15 28:istore 4
    // 16 30:iload 4
    //START LONG LOOP
    // 17 32:sipush 128
    // 18 35:icmpge 49
    //LONG LOOP COMPARE–JUMP
    // 19 38:iload 4
    // 20 40:iconst_1
    // 21 41:iadd
    // 22 42:istore_2
    // 23 43:iinc 4 1
    //END LONG LOOP
    // 24 46:goto 30
    // 25 49:iinc 3 1
    // 26 52:goto 2
    //END OUTER LOOP
    // 27 55:return
}
```
In order to demonstrate the capability of the JOP platform to allow timing of events, as well as to demonstrate the utility of this, timestamps recorded by the application executing on JOP can be compared to expectations as provided by WCET analysis. This is achieved by using the target application to calculate a measure of execution times from these timestamps and print the result. The same code between the start and end of task timestamps was analysed by the JOP WCET analysis tool.

The instrumented code for the run method of each thread in the taskset application is shown in figure 8.14 (the doWork method is as in figure 8.11). In addition to the previous manual instrumentation denoting start/end of task execution, timestamps are printed, using the `Native.rdMem` method to read the current count of processor execution cycles.

```java
/* Task run (work) method */
run()
{
  /* Infinite execution loop */
  for(;;)
  {
    /* Read current count of execution cycles */
    startTime = Native.rdMem(Const.IO_CNT);
    /* Instrumentation to observe start of method */
    System.out.print("00S ");
    /* Print timestamp (cycles) */
    System.out.println(startTime);
    /* Execute loops for duration taskTime */
    doWork(taskTime);
    /* Instrumentation to observe end of method */
    System.out.print("00E ");
    /* Read and print current count of execution cycles */
    System.out.println(Native.rdMem(Const.IO_CNT));
  }
}
```

Figure 8.14 – Code for task run method with printing of timestamps
The JOP platform distribution includes example applications for comparing measured execution times of elements of application code (e.g. simple loops, calling a method, mathematical calculations) against WCET analysis, using the timestamping method described in the previous section. The code being examined is included in a single method `measure` with timestamps added at the start and end of the method. Results are provided for these examples, showing the match between WCET analysis and runtime measurements for various code constructs.

The same structure as these example applications was arranged for the taskset application described in section 8.5. The code for this arrangement is shown in figure 8.15 (the code for the `doWork` method is as in figure 8.11). The run method of the thread on the target application first calculates the overhead of measuring the execution times, in order that it can be subtracted from the measurement of execution time. The time for the start of execution and the time for the end of execution with the subtraction of the overhead are printed. The measurements before and after the invocation of doWork in the `measure` method are omitted from the WCET analysis, thus necessitating the modification to the second timestamp printed.

```java
/* Task run (work) method */
run() {
    int overhead;
    /* Calculate overhead */
    startTask = Native.rdMem(Const.IO_CNT);
    endTask = Native.rdMem(Const.IO_CNT);
    overhead = endTask - startTask;
    /* Infinite execution loop */
    for(;;) {
        /* Measure execution time of doWork */
        measure();
        /* Instrumentation to denote task */
        System.out.print("00 ");
        /* Print start/end execution time */
        System.out.print(startTask);
        System.out.print(' ');
        System.out.println(endTask - overhead);
        waitForNextPeriod();
    }
}

/* Method to measure execution time */
public void measure() {
    /* Record start time of execution */
    startTask = Native.rdMem(Const.IO_CNT);
    doWork(taskTime);
    /* Record end time of execution */
    endTask = Native.rdMem(Const.IO_CNT);
}
```

Figure 8.15 – Code to measure doWork method

Testing was undertaken to determine the execution time of the code under two scenarios. First the execution time of the measure method was measured in each of the three threads concurrently executing in the target application. Secondly, the measure method was incorporated into a standalone single-thread application to
provide data on the effect that the JOP scheduler has on execution time (this is not taken into account by the WCET analysis).

8.5.8 Demonstrating the timing characteristics of the host platform

In order to demonstrate the timing characteristics of the host platform, the relay application can be modified to print the current system time at the point where it reads an event from the target system. This demonstrates the effects of the timer granularity of the host system, noted in section 8.4.6. The relay application is modified such that statements invoking the Java-MaC Filter component are omitted, with calls to `System.println` and `System.currentTimeMillis` added instead, in order to print each line from which any monitor statements are parsed and the current system time. At run time the command to execute the relay application also directs the output to a text file log for off-line analysis.

After executing the arrangement, this log of monitoring events with timestamps can then be parsed into spreadsheet form. This allows calculation of execution times and periods from the timestamps added by the relay application. Execution times for each task can be calculated using the difference between the end of task events and start of task events. The periods were calculated using the difference between each subsequent start of task event. Similar calculations are used by the Java-MaC run-time checker for the MEDL specification described in section 8.4.5, i.e. it is based on the time that events are received on the host PC, as timestamps are added by the Filter component of Java-MaC. As such these calculations are affected by the granularity of the host PC system timer.

8.5.9 Analysing effects of the communication delay

Referring back to the tool chain shown in figure 8.8, two components of the communication delay can be considered separately. The first component is the delay in communicating from the target to the host and second is the delay between the reading of an observation from the serial port by the echo program and dispatching the observation to the filter thread.

The delay in communicating from the target to the host comprises the time to print to the serial port on the target and the time to read from the serial port at the host. In section 7.5.3 of the preceding chapter, where this print time was factored into analysis
of the execution time of the monitoring statements on the target, the delay in sending a character to the serial port was established.

In order to establish the delay in reading from the serial port at the host, the code for the JOP echo program can be modified to timestamp at the start and end of sequences of characters. Normally the echo program checks for a fixed string that denotes that the JOP application has terminated (in order to break from the character reading loop and exit). This can be modified to check for a different string, a sequence of monitoring observations, in order to synchronise for adding timestamps while avoiding any extra overhead from additional string comparison. One timestamp is made before the next character is read and another is placed after the five characters of the monitoring statement are read. These timestamps can then be printed for analysis.

In order to analyse the delay between the echo program and the relay program, the delay in printing across a pipe is considered. A fake echo program can be arranged in order to provide the expected monitoring observations for the relay program. This program uses the System.nanoTime() API method to send out the monitoring observations at approximately the expected intervals for Task T1 (i.e. 25 ms between start and end method events and 100 ms between successive start method events).

Timestamps can be added to the relay application, using the same System.nanoTime() method, at the points where monitoring observations are passed to the filter. Combining the timestamps from both applications provides a measure of the delay between sending observations from the echo program to the pipe and the time that observations are received by the filter.

As introduced in section 8.4.8, in order to evaluate the feasibility of the verification scenario for the experimental monitoring prototype, it is necessary to consider the effects of the communication delay between events being sent from the target application and when they are processed by the run-time checker.

With the analysis of these delays, the effects of the communication chain for observations can then be considered in the context of verifying the specifications described in section 8.4.5.
8.6 RESULTS AND ANALYSIS OF IMPLEMENTING SCENARIO

Results from the implementation and execution of the taskset scenario are presented in this section as follows:

- Results of WCET analysis of nested loop implementation
- Extent of preparations for verifying the application

Additionally, results are presented demonstrating the timing capabilities of the experimental prototype, in terms of the facilities on the JOP target platform and the host platform.

- Comparison of execution times calculated from timestamps in the JOP application against WCET analysis
- Demonstration of the effect of characteristics of the host system timer on calculations of execution time
- Analysis of the delays between dispatching observations from the target and receiving them in Java-MaC

These results allow conclusions on the ability of the prototype monitor and the proposed monitoring approach to verify this scenario that might arise in an embedded real-time system.

8.6.1 Results of WCET analysis

As described in section 8.5.4, the doWork method of the nested loop implementation is first analysed with a WCET bound of zero loop iterations on the outer loop in order to determine the basic number of cycles for the method, with further analysis performed to determine how long each iteration of the two nested loops takes.

To analyse the execution time of the two inner loops, the outer loop was set to a worst-case bound of one and bounds of 0 were set on both inner loops, to provide the basic overhead in cycles (i.e. set up of iterator variables and check of loop bounds).

As the value of 128 was used for the loop bounds in the code, the calculated overhead is only valid for loop bounds of >128. The analysis was repeated with one loop specified for the first ‘short’ inner loop and zero for the second ‘long’ inner loop, and again with the reverse. Subtracting the basic overhead for executing the inner loops,
this provided the execution time for each iteration of both the short and long inner loops (for each additional iteration above 127 iterations).

<table>
<thead>
<tr>
<th>WCET analysed:</th>
<th>Cycles:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero outer loop iterations</td>
<td>214</td>
</tr>
<tr>
<td>One outer loop, zero of either inner loop</td>
<td>252</td>
</tr>
<tr>
<td>One outer loop, one short inner loop, zero long inner loops</td>
<td>274</td>
</tr>
<tr>
<td>One outer loop, zero short inner loops, one long inner loop</td>
<td>276</td>
</tr>
<tr>
<td>Each short loop iteration (&gt; 127 total iterations)</td>
<td>22</td>
</tr>
<tr>
<td>Each long loop iteration (&gt; 127 total iterations)</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 8.2 – Execution times provided by WCET analysis

The execution times provided by WCET analysis are shown in table 8.2. With these execution times known, it was possible to calculate the required bounds on both the inner loops to ensure that each iteration of the outer loop takes exactly one millisecond to execute. The calculations are shown in table 8.3.

<table>
<thead>
<tr>
<th>Calculation:</th>
<th>Cycles:</th>
</tr>
</thead>
<tbody>
<tr>
<td>One millisecond of execution time</td>
<td>50,000</td>
</tr>
<tr>
<td>1000 iterations of both inner loops (minus overhead) = ((22+24) * 1000) - 252</td>
<td>46,038</td>
</tr>
<tr>
<td>Deficit in execution time</td>
<td>3,929</td>
</tr>
<tr>
<td>Extra iterations of long inner loop required = 3962 / 24 = 165.083</td>
<td></td>
</tr>
<tr>
<td>1000 short inner loop iterations and 1165 long inner loop iterations</td>
<td>49,998</td>
</tr>
<tr>
<td>999 short inner loop iterations and 1166 long inner loop iterations</td>
<td>50,000</td>
</tr>
</tbody>
</table>

Table 8.3 – Calculations of required loop bounds

Providing a bound of 1000 on both inner loops achieves an execution time in the region of that required (46038 cycles). The deficit in cycles is made up by extra long inner loop iterations, calculated by dividing the deficit by the execution time of the long inner loop. To the nearest iteration, this provides an execution time two cycles short. This is rectified by subtracting one short loop iteration and adding one long loop iteration. As shown in table 8.3, the final bounds required on the inner loops were 999 for the short inner loop and 1166 for the long inner loop.
8.6.2 Extent of preparations for run-time verification

Ordinarily, in order to monitor a method (i.e. detect the start and end of the execution of that method) the instrumentation added by the Java-MaC instrumentor comprises a statement added as the new first statement to be executed inside the monitored method and a statement added as the new last statement to be executed.

The instrumentation added by Java-MaC is shown in figure 8.16 for the method doWork(), as well as the manual instrumentation added instead for the custom monitoring arrangement. Java-MaC sends a string to the filter thread specifying the start or end of the method (with full classpath, class and method elaborated) and a second argument specifying which monitored entity the observation pertains to (the internal integer reference used by Java-MaC to identify monitored entities).

```java
/* Java-MaC instrumented */
private void doWork (int timelimit) {
    int value,ms,i,j;
    // Instrumentation to observe start of method */
    Filter.sendThrSocket("start of taskset.TaskShort.doWork()");
    // Outer loop */
    for (ms=0;ms<timelimit;ms++) {
        /* Shorter inner loop */
        for (i=0;i<SHORT_BOUND;i++)
            value = i;
        /* Longer inner loop */
        for (j=0;j<LONG_BOUND;j++)
            value = j + 1;
    }
    // Instrumentation to observe end of method */
    Filter.sendThrSocket("end of taskset.TaskShort.doWork()");
}

/* Manually instrumented */
private void doWork {
    int value,ms,i,j;
    // Instrumentation to observe start of method */
    System.out.println("00S");
    // Outer loop */
    for (ms=0;ms<timelimit;ms++) {
        /* Shorter inner loop */
        for (i=0;i<SHORT_BOUND;i++)
            value = i;
        /* Longer inner loop */
        for (j=0;j<LONG_BOUND;j++)
            value = j + 1;
    }
    // Instrumentation to observe end of method */
    System.out.println("00E");
}
```

Figure 8.16 – Instrumentation to monitor method doWork()

The replacement manual instrumentation specifies the monitored entity as the same integer reference as that used by Java-MaC (expressed as a fixed two digits for the
purposes of the experimental work) followed by a character S (for the start of a method) or E (for the end of a method).

The intermediate relay application was prepared as described in section 7.7 of the preceding chapter. The code for the relay application is shown in figure 8.17, with some simplification (e.g. omission of exception handling on input and parsing). A list of methods being monitored, listMonMethods, is added to the relay application (line 5), represented as an array of strings. The index of each string in the array corresponds to the reference number used by Java-MaC for the monitored method (i.e. monitored entity reference 0 in Java-MaC corresponds to taskset.TaskShort.doWork()). These references are determined during the initial automatic instrumentation by Java-MaC of the original target application to be deployed on the JOP core (though the run-time version of this original target application has manual instrumentation instead).

```java
package taskset;

public class TaskSet {
    /* List of methods being monitored */
    public static String[] listMonMethods = {
        "taskset.TaskShort.doWork()","taskset.TaskMedium.doWork()",
        "taskset.TaskLong.doWork()"};

    public static void main(String[] args) {
        /* definition of variables used for reading input */
        String thisLine = null;
        int thisMonitorRef;
        char action;
        /* open up standard input */
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        /* Initialise Filter */
        Filter.init(DEFAULT_HOST, DEFAULT_PORT);
        do {
            /* Read input */
            thisLine = br.readLine();
            /* Check for observations */
            if ((thisLine != null) && (thisLine.startsWith("0"))) {
                /* Parse monitor reference */
                thisMonitorRef = Integer.parseInt(thisLine.substring(0,2));
                /* Parse update */
                action = thisLine.charAt(2);
                if (action == 'S') {
                    /* Send observation to filter */
                    Filter.sendThrSocket("start of " + listMonMethods[thisMonitorRef], thisMonitorRef);
                } else if (action == 'E') {
                    /* Send observation to filter */
                    Filter.sendThrSocket("end of " + listMonMethods[thisMonitorRef], thisMonitorRef);
                }
            } while (thisLine != null);
        }
    }
}
```

Figure 8.17 – Code for the intermediate relay application
The relay application at run time follows the sequence of opening the input reader (line 13), starting a filter thread (line 15) and attempting to read lines of input (do-while loop of lines 16-32). Each line of input that is read is checked for being observation data (line 20) and if it is, the monitored entity reference is parsed as thisMonitorRef (line 22). The intended arrangement for the relay application is a switch statement based on this reference, in order that the appropriate call to the Filter is made. For simplicity, this is omitted in the code of figure 8.17 and the update is instead immediately parsed (line 24) as action. For the three monitored entities in the scenario, which are all methods, action will be a character ‘S’ (line 25) or ‘E’ (line 28) depending on whether the monitoring update pertains to the beginning or end of a method. The relevant observation is passed to the Filter thread by reconstructing the statement that Java-MaC would have included in the original target application during automatic instrumentation. The array of strings listMonMethods is used with thisMonitorRef to pass the appropriate method name to Java-MaC. The filter thread is also passed thisMonitorRef.

8.6.3 Comparison of measured execution time against WCET analysis

As described in section 8.5.6, the code for the doWork method (shown in figure 8.11) was analysed along with the measure method used to measure execution time at run time (code shown in figure 8.15). This was carried out for each of the three task implementations, TaskShort, TaskMedium and TaskLong, with the classes modified to have a main method to allow WCET analysis of each.

```java
0:    aload_0
1:    aload_0
2:    getfield  taskset.TaskShort.taskTime I (4)
5:    invokevirtual taskset.TaskShort.doWork (I)V (11)
8:    return

Attribute(s) =
LocalVariable(start_pc = 0, length = 9, index = 0:taskset.TaskShort this)
LineNumber(0, 39), LineNumber(8, 42)
```

Figure 8.18 – Bytecode for measure method and header for doWork

The bytecode for the implementation of the TaskShort (task T1) measure method is shown in figure 8.18. One consideration in the analysis is that the measured execution
time captures the entire execution of the measure method (as analysed by WCET analysis), with the exception of the final return bytecode. The three cycle counts provided by the WCET analysis also depend on the execution behaviour of the invokevirtual bytecode, used to invoke the doWork method, and the return bytecode of the doWork method. These two bytecodes take a variable number of cycles to execute depending on whether there is a cache miss or hit. The exact execution time is calculated using the formulae shown in figure 8.19, from the JOP handbook by Schoeberl [84].

\[
\text{return:} \quad 21 + \begin{cases} r - 3 & : r > 3 \\ 0 & : r \leq 3 \end{cases} + \begin{cases} l - 9 & : l > 9 \\ 0 & : l \leq 9 \end{cases}
\]

\[
\text{invokevirtual:} \quad 100 + 2r + \begin{cases} r - 3 & : r > 3 \\ 0 & : r \leq 3 \end{cases} + \begin{cases} r - 2 & : r > 2 \\ 0 & : r \leq 2 \end{cases} + \begin{cases} l - 37 & : l > 37 \\ 0 & : l \leq 37 \end{cases}
\]

\[
\text{load time } l = \begin{cases} 6 + (n + 1)(2 + c_{ws}) & : \text{cache miss} \\ 4 & : \text{cache hit} \end{cases}
\]

Figure 8.19 – Formulae for calculating execution time in cycles of return and invokevirtual bytecodes

In the above formulae, \( r \) refers to the memory read wait state (2 cycles for the Spartan 3 platform), \( c_{ws} \) refers to the cache wait state (\( r - 1 \) for \( r > 1 \), i.e. 1 cycle for Spartan 3) and \( n \) refers to the length of the method being loaded in 32-bit words. The two bytecode instructions return and invokevirtual are implemented by microcode for the JOP core (i.e. rather than corresponding to a single microcode) and so for short methods, the execution time of the microcode hides part or all of the load time of method on a cache miss.

For the code in figure 8.11 and 8.15, a miss on method invocation of doWork results in 20 extra cycles (doWork is 63 bytes/8 words long, resulting in \( 6 + (16 + 1)(2 + 1) - 37 = 20 \)). A miss on return from doWork is 9 extra cycles (measure is 9 bytes/3 words long, resulting in \( 6 + (3 + 1)(2 + 1) - 9 = 9 \)).

A tail extract from the output of the WCET analysis tool is shown in figure 8.20, for analysis of the measure method in TaskShort (i.e. with a loop bound of 25 on the outer loop of the doWork method). The WCET analysis tool provides three counts of worst-case processor cycles required to execute code, varying depending on usage of
the method cache at run time. A count is provided for always having a hit on the method cache when a method is invoked, always having a miss on the cache or assuming minimal cache usage (one incidence of a cache miss on method invocation).

The three computations for WCET are highlighted in bold in figure 8.20. The minimal caching scenario corresponds to a cache miss on the invocation of `doWork` from within the `measure` method (providing the difference of 20 cycles). This is denoted by the highlighted "cache: 20". The always miss scenario includes a cache miss on the return from `doWork` (load of `measure` method, another 9 cycles), denoted by the highlighted "cache: 29". The WCET analysis results are shown in table 8.4 with the three figures calculated for outer loop bounds of 100 (TaskMedium) and 75 (TaskLong) for the `doWork` method.

<table>
<thead>
<tr>
<th>Outer loop bound for <code>doWork</code></th>
<th>Always Hit</th>
<th>Always Miss</th>
<th>Minimal Caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 (TaskShort)</td>
<td>1,250,169</td>
<td>1,250,198</td>
<td>1,250,189</td>
</tr>
<tr>
<td>100 (TaskMedium)</td>
<td>5,000,169</td>
<td>5,000,198</td>
<td>5,000,189</td>
</tr>
<tr>
<td>75 (TaskLong)</td>
<td>3,750,169</td>
<td>3,750,198</td>
<td>3,750,189</td>
</tr>
</tbody>
</table>

Table 8.4 – Results of WCET analysis for all three tasks

The target application was executed on the JOP platform and the printouts for the start and end of tasks recorded (i.e. the measurements at the start/end of the method `measure` that was analysed for WCET). From this the execution times of the code in the three tasks was calculated, with periods calculated based on successive start of method events. The minimum, maximum and average values calculated from timestamps recorded over 5 seconds of execution are presented in table 8.5, along with the difference compared to the WCET analysis (minimal caching scenario) and specified period. The source dataset can be found in appendix B.
Task T1

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Difference</th>
<th>Period</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>1254768</td>
<td>4579</td>
<td>4989157</td>
<td>-10843</td>
</tr>
<tr>
<td>Maximum</td>
<td>1254908</td>
<td>4719</td>
<td>5010808</td>
<td>10808</td>
</tr>
<tr>
<td>Average</td>
<td>1254813</td>
<td>4624</td>
<td>4999981</td>
<td>-19.4</td>
</tr>
</tbody>
</table>

Task T2

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Difference</th>
<th>Period</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>5139637</td>
<td>139448</td>
<td>9988706</td>
<td>-11294</td>
</tr>
<tr>
<td>Maximum</td>
<td>5140662</td>
<td>140473</td>
<td>10011164</td>
<td>11164</td>
</tr>
<tr>
<td>Average</td>
<td>5140233</td>
<td>140044</td>
<td>9999971</td>
<td>-29.26</td>
</tr>
</tbody>
</table>

Task T3

<table>
<thead>
<tr>
<th></th>
<th>Execution Time</th>
<th>Difference</th>
<th>Period</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>3885630</td>
<td>135441</td>
<td>19993950</td>
<td>-6050</td>
</tr>
<tr>
<td>Maximum</td>
<td>3887097</td>
<td>136908</td>
<td>20005824</td>
<td>5824</td>
</tr>
<tr>
<td>Average</td>
<td>3886450</td>
<td>136260</td>
<td>20000010</td>
<td>10.04</td>
</tr>
</tbody>
</table>

Table 8.5 – Calculated execution times and periods based on timestamps

All three tasks have higher execution time than that predicted by WCET analysis. The period also varies from that expected, although on average matches more closely to that expected. As regards the execution time, the WCET analysis does not take into account the concurrent execution of multiple threads of execution and the overhead of the scheduler activity. The higher execution times calculated for T2 and T3 are also due to the method in which they are calculated. T1 task execution times were calculated by subtracting the start of execution timestamps from the end of execution timestamps. The nearest that T2 execution time can be calculated by the same approach is to subtract the calculated execution time of the pre-empting T1 execution from the T2 start/end of method timestamp calculation. This means that the T2 execution time calculation includes, in addition to execution time in the measure method of T2: the end of the pre-empting task T1 execution (outside of its measure method) and the time to switch back to T2 execution after T1 finishes execution after pre-empting T2. The calculations for T3 are similarly affected.
Table 8.6 – Measured execution times and differences compared to WCET values

<table>
<thead>
<tr>
<th>Outer loop bound for doWork</th>
<th>Measured</th>
<th>WCET Hit difference</th>
<th>WCET Miss difference</th>
<th>WCET Min difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 (task T1)</td>
<td>1,250,168</td>
<td>-1</td>
<td>-30</td>
<td>-21</td>
</tr>
<tr>
<td>100 (task T2)</td>
<td>3,750,168</td>
<td>-1</td>
<td>-30</td>
<td>-21</td>
</tr>
<tr>
<td>75 (task T3)</td>
<td>3,750,168</td>
<td>-1</td>
<td>-30</td>
<td>-21</td>
</tr>
</tbody>
</table>

In order to confirm that the excess execution time was due to the multi-threaded environment and that the code of the measure method itself executes within the WCET predicted by analysis, the code was inserted into a standalone application. This application was executed on the JOP core and the measured execution time from the single execution recorded. This was repeated for each value of parameter passed to the doWork method (i.e. 25, 100 and 75, corresponding to tasks T1, T2 and T3). The results are displayed in table 8.6 next to the difference compared to WCET analysis.

For these results, execution time can be matched exactly to the corresponding WCET analysis. The measured execution time does not include the return bytecode of the measure method, which accounts for 21 cycles in the case of a cache hit, or 30 cycles in the case of a cache miss, corresponding to the WCET analysis conditions of the right-hand columns in table (only cache misses, or minimal cache usage). There is a single cache miss on the invocation of the doWork method during the run time measurement, as evidenced by the difference of -1 cycle (-21 cycles for return, +20 cycles for cache miss on doWork invocation) from the always hit WCET condition.

8.6.4 Demonstration of the effects of the host system timer

In order to demonstrate the effects of the host system timer, two sets of timestamps were recorded for the start/end of task events on the target application. For each event a timestamp was added within the target application when events were printed and by the relay application when events were received. The characteristics of the host timing can then be compared to the characteristics of the target system timing.

Initially, the recorded timestamps were used to calculate the execution time and period of the task based on timestamps from the target and timestamps from the relay application. A summary of the resulting calculations over ten seconds of execution time is displayed in table 8.7 (the source dataset can be found in appendix C).
<table>
<thead>
<tr>
<th>Task</th>
<th>(Period: 100, $T_E$: 25)</th>
<th>Min (ms)</th>
<th>Max (ms)</th>
<th>Mean (ms)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time (JOP)</td>
<td>26.66</td>
<td>26.68</td>
<td>26.67</td>
<td>0.0042</td>
</tr>
<tr>
<td></td>
<td>Execution Time (Relay)</td>
<td>15.00</td>
<td>32.00</td>
<td>28.20</td>
<td>6.6151</td>
</tr>
<tr>
<td></td>
<td>Period (JOP)</td>
<td>99.78</td>
<td>100.22</td>
<td>100.00</td>
<td>0.1821</td>
</tr>
<tr>
<td></td>
<td>Period (Relay)</td>
<td>93.00</td>
<td>110.00</td>
<td>100.00</td>
<td>7.7717</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>(Period: 200, $T_E$: 100)</th>
<th>Min (ms)</th>
<th>Max (ms)</th>
<th>Mean (ms)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time (JOP)</td>
<td>103.75</td>
<td>103.78</td>
<td>103.76</td>
<td>0.0053</td>
</tr>
<tr>
<td></td>
<td>Execution Time (Relay)</td>
<td>94.00</td>
<td>110.00</td>
<td>103.20</td>
<td>7.5206</td>
</tr>
<tr>
<td></td>
<td>Period (JOP)</td>
<td>199.89</td>
<td>200.06</td>
<td>200.00</td>
<td>0.0380</td>
</tr>
<tr>
<td></td>
<td>Period (Relay)</td>
<td>187.00</td>
<td>204.00</td>
<td>200.00</td>
<td>6.5115</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>(Period: 400, $T_E$: 75)</th>
<th>Min (ms)</th>
<th>Max (ms)</th>
<th>Mean (ms)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time (JOP)</td>
<td>78.54</td>
<td>78.58</td>
<td>78.57</td>
<td>0.0087</td>
</tr>
<tr>
<td></td>
<td>Execution Time (Relay)</td>
<td>62.00</td>
<td>93.00</td>
<td>77.60</td>
<td>9.8102</td>
</tr>
<tr>
<td></td>
<td>Period (JOP)</td>
<td>399.94</td>
<td>400.07</td>
<td>400.00</td>
<td>0.0328</td>
</tr>
<tr>
<td></td>
<td>Period (Relay)</td>
<td>390.00</td>
<td>407.00</td>
<td>400.00</td>
<td>7.7717</td>
</tr>
</tbody>
</table>

Table 8.7 – Summary of measurements over two seconds of execution time

In table 8.7, the minimum and maximum values calculated for each parameter over the ten second interval are provided, along with the average value and the standard deviation for each parameter. Calculations based on the timestamps printed by the target application are labelled ‘JOP’ and calculations based on the timestamps added by the relay application on the host PC when events are received are labelled ‘Relay’. The target application has priorities and execution times ($T_E$) assigned as in table 8.1 at the start of this chapter, with the TaskShort, TaskMedium and TaskLong classes implementing T1, T2 and T3 respectively. The assigned periods are reflected in the matching average period calculation for both the JOP and relay timestamps. The JOP execution times are greater than the specified parameters due to the extra overhead of sending timestamps to the serial port.

The individual values calculated using the relay timestamps again vary greatly from the average (worst case 13ms for T1). The period (difference between timestamps for two successive start of task events) averages out correctly for the relay timestamps, so despite the variation in timestamps recorded by the relay application, the discrepancy
between the time recorded by the target application and the relay application does not grow worse over time.

In order to graphically illustrate the effects of the host system timer on the task schedule as calculated from the relay timestamps (i.e. the schedule as seen by JavaMaC), the start and end of task times as recorded by the target application and the relay application were used to graph a reconstructed task schedule, shown in figure 8.21. The lighter coloured lines represent the schedule as captured by the JOP timestamps and the darker coloured lines represent the schedule as captured by the relay timestamps.

ClockRes, the tool by Russinovich [94] mentioned in section 8.4.6, was downloaded and executed on the primary host machine used for this project (a Dell Dimension 5000 with 3 GHz Pentium 4 HT processor), reporting a maximum timer interval (default) of 15.625 milliseconds. Note that this corresponds to the value provided for multiprocessor x86 machines despite the single-core Pentium 4 processor. This is due to the HT (hyperthreading) feature providing two logical processors.

For the T1 and T3 task in particular, it can be seen that the granularity of the timer results in two different calculations for the task execution time, of longer or shorter duration than the task execution actually is. This is due to the measuring of times on the host system in 15.625 ms “chunks” (hence for example, times of 15 or 32 ms

Figure 8.21 – Comparison of task schedule reconstructions over two T3 periods
being calculated on the host system for T1 execution, the min/max values in table 8.7). The interval is approximately rather than exactly 15.625 milliseconds (Grobler and Kourie [95] detail results observing the slight variation in this interval in using a Windows Timer API).

### 8.6.5 Analysis of communication delay

As described in section 8.5.8, two components of the communication delay between the target application and Java-MaC were analysed, the delay across the target-host serial interface and the delay in sending observations across a pipe to the relay application.

**Serial link delay**

In section 7.5.3 the delay in sending observations across the serial link was established as 0.1 ms approx. per character, due to the 115,000 baud rate with no handshaking, one start bit and one stop bit, and assuming one bit spacing between messages. For the monitoring statements needed for the scenario outlined in this chapter, five characters are sent (two characters for monitoring reference, one for start/end of method and a newline/return), resulting in a delay of approx. 0.5 ms.

As described in section 8.5.8, the code for the JOP echo program was modified to timestamp before and after a sequence of characters for a monitoring observation was read. This data was gathered across 40 seconds of execution time to provide average delay as well as minimum, maximum and standard deviation across the observed interval. This data is combined with the delay in sending the message across the serial link in table 8.8.

<table>
<thead>
<tr>
<th></th>
<th>Delay (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sending from JOP</strong></td>
<td>0.5</td>
</tr>
<tr>
<td><strong>Receiving on Host</strong></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.373</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.320</td>
</tr>
<tr>
<td>Maximum</td>
<td>0.753</td>
</tr>
<tr>
<td>Std. Dev</td>
<td>0.068</td>
</tr>
<tr>
<td><strong>Combined Delay (average)</strong></td>
<td>0.873</td>
</tr>
</tbody>
</table>

Table 8.8 – Delay in sending/receiving one observation across the serial link
The delay in reading the observations from the serial port is not entirely consistent although below one millisecond in all cases. Examining the recorded values, it was noted that the delay tended towards the average after the first couple of seconds of execution time, with the maximum values occurring at the start of the test execution.

**Echo to relay/filter delay**

As described in section 8.5.8, a fake echo program was arranged to provide a replica of the observations that would normally be emitted from JOP and read from the serial port by the filter. This fake echo program was used to timestamp when observations were printed to standard output, with the relay application modified to timestamp the calls to the filter thread with the observations (using the same clock and System.nanoTime() Java API method). These timestamps were recorded across 20 seconds of execution time to provide an interval across which to analyse the average and min/max delay as well as standard deviation. These values are shown in table 8.9.

<table>
<thead>
<tr>
<th>Delay (ms)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.574</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.131</td>
</tr>
<tr>
<td>Maximum</td>
<td>29.242</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>2.203</td>
</tr>
</tbody>
</table>

Table 8.9 – Delay in sending one observation from echo to relay/filter

Unlike the delay in reading from the serial port, these timestamps demonstrated some occasional behaviour more significantly outside the average case. In fact, examining the timestamp calculations, most are closer to the minimum in table 8.9 rather than the average, which is skewed by some occasional anomalously high delays. These lengthier delays can be more easily visualised in the graph of figure 8.22.

Combined with the delay in sending across the serial link, the end to end delay between dispatching of observations from the target application on JOP and the receipt of those observations by Java-MaC can be calculated as just over 1 millisecond in the average case, although with extra delays from time to time in sending from the echo program to the relay application.
8.7 CONCLUSIONS

The aim of the test case, evaluating the monitoring approach with reference to the usefulness of run-time verification for an embedded environment, was achieved by fulfilment of the objectives set out in section 8.2. From fulfilling these objectives, it was possible to conclude that the experimental monitoring prototype (using Java-MaC to verify an application executing on the JOP platform) is not sufficient to monitor real-time timing properties of an application in an embedded system without modification to parts of the monitoring arrangement. More generally, the evaluation demonstrated how run-time verification might be arranged as a monitoring approach for embedded systems and with suitable host platform and target-host communication arrangements, it could prove a viable approach.

There are four main subjects of the conclusions presented in this section:

- Ability to implement the scenario using the experimental prototype.
The insufficiency of some potential remedies to the timing issues.

Some potential remedies for a more practical monitoring arrangement.

Conclusions on the general monitoring approach, based on testing.

8.7.1 Ability to implement the scenario

The implementation for the JOP platform of an application with tasks scheduled under the rate-monotonic algorithm was presented. The API methods and built-in scheduler allowed the implementation of such an application. Further to this, it was possible to perform static analysis of the application in order to determine that the tasks would execute for the specified execution time, such that the scheduler should be able to perform the scheduling as predicted in section 8.4.1.

The execution time of tasks in the revised target application (modified to avoid the deficiencies of the original implementation) was demonstrated to be close enough to implement the scheduling scenario, as measured execution time had a maximum deviation of 0.22 ms from the expected execution time. A comparison of timestamps printed by the target application against worst-case execution time expectations produced by the WCET analysis tool demonstrated this small deviation. Similar measurement of execution time of the same application code in a standalone application provided measurements matching the expected execution time as predicted by WCET analysis, thus showing that the deviation from the expected execution time in the original target application was due to the overhead of the scheduler. Specifically, as explained by Schoeberl [96], the magnitude of the overhead can be explained due to the scheduler being invoked by an idle tick that occurs if no new thread is released for 10 milliseconds.

As regards the proposed run-time verification, the MEDL language and Java-MaC run-time checker did provide the required mechanism to specify the timing conditions to be verified. It was possible to include the concept of pre-emption in this specification. However, the system timer facilities relied on by the API method used by Java-MaC inhibited the verification of the scenario for the parameters specified in section 8.4.1, as the timer granularity was not sufficient for timing intervals of that order.
Comparison of calculations from timestamps added to the implemented application against values from WCET analysis did demonstrate that the JOP platform was capable of executing the described scenario such that tasks were correctly scheduled. Performing calculations based on timestamps added by the relay application on the host system demonstrated the effects of the poor system timer granularity, as expected. A comparison against the same calculations for the JOP platform demonstrated the variation from the calculations that would need to be available to the Java-MaC run-time checker in order to provide run-time verification.

The experimental monitoring prototype, although allowing some aspects of the scenario to be implemented, is thus insufficient for performing the desired run-time verification due to the lack of host system timer resolution.

Additionally the delay in sending observations from the target application on JOP to Java-MaC on the host was considered. Testing and analysis provided data demonstrating that for the existing monitoring prototype, the link between the echo program and the relay application adds occasional delays far in excess of the average. Combining the echo program and the relay application (i.e. having the relay application read directly from the serial port) would eliminate these particular delays.

In general the verification scenario requires a consistent delay between the target and the host, even if the delay is in the order of milliseconds.

If it can be assumed that a relay application reading the serial port directly would not lead to any delays significantly outside the average case calculated (approx 1 ms), then in theory a verification scenario with task parameters of longer intervals (e.g. in the order of tens of milliseconds) and with lower utilisation than this scenario could potentially be verified using a modified form of the experimental apparatus described in this thesis.

8.7.2 Insufficiency of certain remedies

A higher resolution for System.currentTimeMillis is not possible under Windows as the API method uses the default clock resolution, rather than a user-defined higher resolution. As noted by Russinovich [97], the Windows system timer resolution can be set to a higher resolution, indeed ClockRes reported a current resolution of approx. 2 ms on the host machine used during testing. Indeed the JVM itself can request a higher clock resolution, albeit with some difficulties as noted by Holmes [91] (the
command line switch for the JVM to request 1 ms clock resolution does not take effect by default due to a bug [98]). However, Holmes also reports that System.currentTimeMillis uses the default resolution irrespective of any higher resolution currently applied. Testing on the host system used for the work presented in this chapter confirmed that setting the Windows System time to a higher resolution (1 ms) had no impact on the resolution of values returned by System.currentTimeMillis (in any event the effects of the default 15.625 ms resolution had already been demonstrated despite the host machine having previously had the resolution set to 2 ms).

In theory a higher resolution can be achieved under Linux by modifying the system configuration to update the system timer more frequently, but this tends to impair system performance due to a higher number of clock interrupts (hence why the default has been lowered to 250 Hz in recent Linux kernels, as opposed to a previous value of 1000 Hz).

As regards the higher precision System.nanoTime API method used during testing, there can be problems with the underlying timing methods. On Windows, the Win32 API QueryPerformanceCounter (QPC) is used, as noted by Holmes [91], who lists some of the system timers that the Windows operating system may use to implement the API. QueryPerformanceCounter and associated kernel mode KeQueryPerformanceCounter do not however, behave consistently across various versions/installations of Microsoft Windows. For example, with Windows XP or Server 2003, as described by Microsoft [99] a boot configuration file switch selects use of either the ‘PM timer’ (power management timer) or ‘TSC timer’ (time stamp counter timer). There are some underlying problems with use of a processor instruction, RDTSC, to quickly read the TSC timer, though use of QPC potentially avoids some pitfalls as described by Walbourn [100]. There can be problems with the TSC even with QPC and the PM timer may be selected to avoid these, as recommended by Microsoft [101] [102]. However, use of the PM timer requires safety mechanisms in application code, also recommended by Microsoft [103]. Various other clock sources may be used by the operating system, as described by Microsoft [104], including a HPET (High Precision Event Timer). This is only an option however with recent Microsoft operating systems (Vista, Windows 7) and
modern PC motherboards. With Linux, the operating system is also dependent on the hardware clock sources above as described by Bovet and Cesář [105].

As regards arranging a consistent delay in reading observations from the serial port and communicating them to Java-MaC, although reading the serial port directly in the relay application would bring delays more in line with the average, the host system has some underlying deficiencies in facilitating a consistent delay. Although an average figure for this delay can be provided from testing, there is no way to guarantee that individual delays will remain within a certain limit. This is due to the general purpose nature of the host system. As an x86-based PC, the execution of code is affected by pipelining (non-deterministic behaviour on branches) and caching (different execution time depending on cache hit or miss). The general purpose Windows XP operating system used also adds variable delays due to background operating system tasks, device drivers and disk/memory/power management arrangements. Finally the Java Virtual Machine adds variable delays due to intermittent garbage collection.

8.7.3 Potential remedies for a practical monitoring arrangement

The experimental approach could be improved by providing higher resolution timing to timestamp monitoring observations; however, the experimental approach would still be affected by the non-real time nature of the host PC and operating system. The higher resolution timestamps could be arranged by using a host PC and operating system that utilises the HPET as well as amending the Java-MaC filter component to timestamp monitoring observations using System.nanoTime. As regards the more general timing problems due to the characteristics of the host platform, operating system activity for example, could delay the reading of some monitoring observations from the serial port but not others, still resulting in some degree of jitter in the timestamps of observations.

The serial port is not suggested as a definitive communication mechanism for an actual monitoring approach, although with allowance made for any adverse effects it could be utilised. Specifically the delays in waiting for transmission of observations would need to be accounted for, as well as the need to avoid priority inversion with multiple threads writing to the serial port. An advantage to using the serial port is its
widespread implementation on embedded systems and higher bandwidth than other commonly used ports such as JTAG.

A preferable approach may be to use memory-mapped I/O, allowing high-bandwidth fast communication of monitoring observations with minimal overhead for executing the monitoring statements in the target application. The serial connection was used for the experimental work as a substitute for the expected use of a dedicated debug port for an actual monitoring approach, e.g. a next-generation successor to the IEEE 1149.1 interface (JTAG) [80]. Such a communication mechanism would allow transmission of target system timestamps along with monitoring observations, in order that the monitoring application could operate on the basis of actual timing of events in the target system.

Java-MaC in a normal configuration has the filter component executing on the target platform and thus monitoring observations are timestamped using the target system clock. With a communication mechanism allowing transmission of more data, the experimental arrangement could be modified to relay monitoring observations timestamped by the target system directly to the event recogniser component of Java-MaC. This would involve reworking the filter component entirely to consist of two components – the print outs from the target application (including reference to monitored entity, new state of entity, and timestamp) and the execution thread on the host to receive the communicated observations and relay them to the event recogniser.

Providing a real-time environment on the host PC could allow more deterministic timing on the handling of monitoring observations by the host monitoring application. A more time-oriented run-time verification approach could make use of real-time features to accurately respond to target application events. The real-time environment could be provided by installing a real-time operating system on suitable PC hardware, with a real-time programming language used to implement the monitoring application. Indeed an implementation of the Real-Time Specification for Java [106] could be used if a Java host monitoring application were still to be used. Sammapun et al [107], in presenting extensions to the MEDL and MaC to allow time-driven run-time verification for the purposes of verifying real-time applications, suggest that porting the approach to real time Java would allow more accurate monitoring and checking of real time properties.
By using embedded hardware suitable for real-time systems, e.g. an embedded PC, the host platform could be further customised for the purpose of hosting a run-time verification approach suitable for verifying real-time and embedded applications. This approach could be taken further by incorporating the host platform as a separate processing core on the actual target platform, for critical systems for which run-time verification would provide a useful safety assurance in production systems.

### 8.7.4 Conclusions on general monitoring approach based on testing

Although the experimental monitoring approach would require modification as suggested above to achieve the aim of verifying a real-time embedded application, it is considered that such an approach could be arranged. Elements of the test platform demonstrated features that could be utilised in an operation run-time verification approach for embedded systems. For example, the JOP platform demonstrated its suitability as a test platform by allowing static analysis of application bytecode in order to determine an exact WCET value for the code. The WCET analysis performed during testing allowed diagnosis of the operation of the target application and the issues with the host platform.

Despite being affected by the host system clock resolution, the use of Java-MaC with the experimental monitoring approach demonstrated that in principle, minimal instrumentation of the target system can provide the necessary monitoring observations to verify operation of a target application. However, to facilitate this approach in practice, a suitable communication mechanism between target and host would be required, one that does not delay execution of the target application (as the blocking wait on printing to the serial port did in the experimental approach).

### 8.8 SUMMARY

This chapter presented a case study of a scenario using the Java-MaC/JOP arrangement of chapter 6 to verify that a set of tasks scheduled on JOP, using the rate-monotonic algorithm and using a high percentage of processor time, are scheduled as predicted by off-line analysis. The JOP scheduler and JOP API methods for threads were described. Methods of implementing the scenario for JOP were presented, along with the use of a WCET tool to perform timing analysis of code to determine the necessary parameters to include in the application code in order that tasks execute for
a predetermined interval. The implementation of the verification task in MEDL and PEDL was detailed.

The timing facilities available to applications on the JOP platform and used by Java-MaC were described, along with the issues with the system timer used by the latter. Tests were presented to demonstrate the effects of the timer used by Java-MaC, as well as to demonstrate timing on the JOP platform. Tests were also described for analysing the delays between sending an observation from an application on the JOP platform and receiving it in Java-MaC.

Results were presented for the WCET analysis used to implement the scenario on JOP. The preparations for run-time verification are also discussed. Comparisons were presented of measured execution times using timestamps printed by the target against WCET analysis. Results of adding timestamps using the low granularity system timer demonstrated the effects of the use of this timer by Java-MaC. The communication delay between the target application and the host verification was analysed.

Conclusions are drawn concerning the ability to implement and verify the scenario using the prototype monitoring system. Reasons are provided for the inadequacy of certain simple modifications to the experimental arrangement, such as using a higher precision host clock. Viable remedies to provide a suitable host platform and run-time verification approach are also presented. A final conclusion is made that the suggested monitoring approach for embedded systems, of using run-time verification, could be a viable approach in general.
CHAPTER 9 – RESULTS,

CONCLUSIONS AND FUTURE WORK
9.1 INTRODUCTION

The work detailed in this thesis demonstrates that it is possible to verify the run-time behaviour of an application with real-time characteristics executing in a deeply embedded system against a subset of the application specifications. The overhead from this monitoring approach can be minimised depending on the specific arrangement used for the monitoring apparatus and the extent of the specifications verified.

The proposed monitoring approach in section 2.5.1, a hybrid on-chip monitoring scheme supporting run-time verification, has been presented as a workable monitoring approach for embedded systems that can offer guarantees about the reliability of an embedded application. The prototype implementation described in chapter six demonstrated a complete monitoring solution for embedded systems based on run-time verification, with the overhead of the monitoring quantified. It was demonstrated that tools designed for system level platforms can be scaled for use in embedded system monitoring.

Based on the research undertaken, requirements for a good monitoring solution using this approach are presented in this chapter. This chapter also summarises the results arising from a review of literature, experimental work, case studies and experiences in verifying the execution of embedded software by using an experimental monitoring apparatus. Recommendations for improving the experimental apparatus are made. Some suggestions for future work are given, along with a potential application of the monitoring approach.

9.2 CONCLUSIONS FROM LITERATURE REVIEW

The initial aims set out in chapter 1 and addressed by the literature review were: the investigation of the suitability of the lightweight approach of run-time verification as a monitoring solution for safety-critical embedded systems and the proposal of a workable monitoring approach for deeply embedded systems.

The lightweight approach of run-time verification as a monitoring solution for safety-critical embedded systems was examined in chapter 2, based on identification of the limitations of monitoring embedded systems in general and consideration of the attributes of the run-time verification approach. In general, run-time verification
offers guarantees about the operation of certain aspects of the software, while being a scalable approach suitable for the limited monitoring apparatus that can be supported by embedded systems. A workable solution for the development of a monitoring approach for deeply embedded systems was proposed in chapter 2, with its adaptation for the purposes of run-time verification described in chapter 3.

### 9.2.1 Specific conclusions from reviewing literature

Two conclusions regarding the monitoring of real-time embedded systems can be drawn from reviewing literature in the area. First, the literature suggests that in some regards, issues identified decades ago for monitoring software in general remain relevant today for the monitoring of software in an embedded device. Second, limited literature is available pertaining to traditional monitoring of software in an embedded environment and research concerning run-time verification of software is mostly confined to the research environment, or at best, a personal computer environment.

The long-standing issues for the monitoring of software in general are arranging sufficient visibility into the operation of the system and avoiding interference to the operation of the system by doing so. These issues are only more relevant than ever where embedded systems are concerned. Traditional monitoring of software relies on the addition of software to the target system to output observations (the overhead of which can interfere with normal operation) and/or attaching hardware probes (which can be difficult to arrange such that enough information is available).

For embedded systems, the overhead of additional software can potentially be an even more serious issue than with traditional systems, as embedded systems typically have limited computing resources and often use tightly organised software (e.g. real time scheduling). Attaching hardware probes or interfacing to the hardware is also more problematic with embedded systems, which may include much of the hardware on a single chip.

One of the later refinements of traditional monitoring of software has been the development of the concept of run-time verification. Rather than arbitrarily monitor software in an ad-hoc fashion, or attempt to completely verify the system, run-time verification relies on run-time checking of software against formal specifications that form a subset of the system's overall specifications. However, existing literature on the subject of run-time verification focuses on how to achieve the verification, with
fewer details on how software is monitored. For run-time verification tools such as Temporal Rover, Java Pathfinder and Java-MaC, the described target is software on a personal computer. Nevertheless, literature pertaining to the Java-MaC run-time verification tool describes separate modules to handle the observation of the target software, the relay of monitoring data, and the verification. This offered scope for a prototype monitoring arrangement suitable for software on embedded systems and sufficient for at least limited run-time verification, as the modules for observing the target software could be arranged for a different target-host arrangement without amending the run-time verification engine.

9.3 PROPOSED REQUIREMENTS FOR A MONITORING SOLUTION

Based on the experimental work detailed in this thesis, along with consideration of existing literature pertaining to traditional monitoring of software, monitoring embedded systems and run-time verification of software, some requirements for a good monitoring solution for embedded systems can now be proposed.

9.3.1 Run-time verification as a suitable monitoring approach

Formal guarantees about the correct operation of aspects of a target system can be provided through the use of run-time verification. It is partly for this reason that run-time verification is recommended as an approach for monitoring embedded systems, as indeed it is one of the main recommendations for using run-time verification in general. Based on the experimental work, a run-time verification tool should fulfil the following three requirements:

- Scalability
- Modularity
- Ease of use

Scalability of the monitoring approach is important for embedded systems as in general, the more complex the monitoring task, the more that such monitoring may interfere with the normal operation of the target system. Run-time verification is an inherently scalable monitoring approach as the approach of verifying the software itself, as opposed to a model (as used with traditional verification), precludes complete verification. As such, the scale of specifications against which the software is verified may be reduced or increased to allow for the scale of monitoring that is
feasible. This ensures that even with limited target resources to support the run-time verification approach, as might be the case with an embedded system, a few particularly important characteristics of the target application may be checked.

The modularity of the monitoring approach is important to the extent that part of the monitoring solution has to be incorporated on the target system (i.e. to observe software execution), while to reduce the overhead of accommodating monitoring apparatus on the target system itself, it is necessary that a separate module (or modules) of the monitoring apparatus be arranged on a host system, in order to perform the comparison of observations against specifications.

A simple form of expressing specifications is desirable, as unlike formal testing, run-time verification can be used to check the operation of software in its native environment. Such an approach may be deployed outside a controlled testing environment. The experimental work detailed in this thesis, utilised Java-MaC with its straightforward languages for expressing specifications and linking them to entities to be monitored, to demonstrate the scalability of run-time verification.

9.3.2 Host platform

In order to provide a suitable host platform for run-time verification of a real-time application, the host platform must have deterministic real-time functionality. The results of experimental work of chapter 9 confirm the particular difficulty of using a general purpose host platform to arrange verification of timing properties of real-time software executing in an embedded system. Although refinements to the arrangement used may offer the prospect of sufficient timing resolution to reason about time in the order of milliseconds, the use of a general purpose host platform precludes any guarantee that timing is accurate, due to non-deterministic behaviour of the host. A suitable host can be provided by providing both a software platform and a hardware platform supporting more deterministic behaviour.

Apart from the consideration of deterministic behaviour, a high level programming language is desirable to implement the run-time verification host application. Real-time Java is one possible choice therefore to support more deterministic behaviour and also provide a high-level language for run-time verification. Given the existing implementation of run-time verification applications in Java this could allow the possibility of porting an existing run-time verification tool. The choice of operating
system is linked to the choice of programming language. For example, Sun Java Real-Time System, a commercial implementation of the Real-Time Specification for Java (RTSJ) by Sun Microsystems, requires Solaris 10, SUSE Linux Enterprise Real Time, or Red Hat Enterprise MRG. The hardware platform must be capable of supporting the chosen operating system, although for the OS requirements of Sun's RTSJ, a general purpose x86 or Solaris PC is sufficient.

The experimental work detailed in chapter 8, though it used a general purpose operating system, highlighted the requirement for the host system to support accurate clock timing, in order to allow run-time verification of timing parameters. Although an RTOS can be installed on a general purpose PC that is lacking in such support, it is preferable that a PC with recent hardware features be used, for example the High Precision Event Timer. Dedicated hardware can also be utilised, e.g. a simple embedded PC for real-time computing. This can ensure that power management or multiple processor cores do not interfere with more basic timing methods such as using the time stamp counter (TSC) or power management timer (PMT).

9.3.3 Host-target interface

Based on the experiences with the experimental run-time verification arrangement for an application on the JOP platform, a general host-target interface mechanism can be specified for the proposed monitoring solution in this chapter. The proposed arrangement comprises minimal software instrumentation in the form of memory writes of observations, combined with a hardware communication interface on the target system to relay these to the host. The memory to which observations are written by the code added to the target application is mapped to I/O (the hardware communication interface).

The limited software instrumentation should form part of the host-target interface in order to overcome the difficulties of externally observing the execution of software in an embedded system. The additional monitoring code, by simply writing observations to a memory location, can avoid holding up execution of the target application. Only a known deterministic minimal memory write time need be considered in accounting for the effect of adding code to the target application. Even so, it may be necessary for the instrumentation of code to occur at design time, as any later addition of monitoring code may affect time-critical sections of the target application. However,
if sufficient specifications of the target system are known, it may be possible to compute the effects of monitoring code added at a later stage and account for these (i.e. compensate for interference) in any later verification task. Monitoring code incorporated at design time may have to form part of the final application as its removal would also potentially affect time-critical sections of the target application.

In order to allow the minimal interference caused by instrumentation consisting of simple memory writes, it is necessary that hardware on the target system takes the entire burden of communication between the target and the host. To achieve this, the hardware interface should include on-chip hardware used specifically for the purposes of communicating observations.

9.3.4 Overhead of proposed monitoring solution

The main overhead of the proposed monitoring solution in terms of burden on the target system is the execution time of the monitoring statements. Although it is intended that these be minimised by writing observations directly to a memory location (rather than undertaking any further processing or communication of the observations within the target application), the overhead must nevertheless be accounted for. The overhead is dependent on memory write time, the size of the observation, the processor cycles to execute the additional instructions in the target code, and the number of observations necessary.

Another factor is the overhead of hardware support for communication of the observations between the target and host. One needs to consider incorporating a buffer to hold observations. This means that observations can be printed from the target application without blocking execution. Also, potentially a couple of observations could occur close together, resulting in bytes of observation data being written to memory faster than they can be communicated to the host. With such a buffer, a fast enough communication interface between the target and host can be better supported.

Any run-time verification on the host would have to take account of potential delays between the time observations are recorded by the monitoring code within the target application and the time that observations are received on the host. Observations would however always be in order as long as there are no errors due to buffer overflow. As the host system may be reasoning about event times in a real-time
environment, worst-case delays would have to be considered. As such, it is necessary that as high speed a communication interface as possible is used between target and host (and that it is deterministic) in order not only to lessen the worst-case communication delay, but also allow as small an observation buffer as possible on the target, as the host would also have to incorporate a worst-case scenario of a buffer being filled (and thus maximum delay to an observation due to buffering).

9.3.5 Summary of proposed monitoring approach

The proposed monitoring approach attempts to serve the specific common attributes of embedded devices. Such devices often have limited hardware resources (compared to general purpose computing devices), real-time attributes, and strict operational parameters due to interactivity, real-time components, and potential use in real-life situations requiring fail safe operation.

Figure 9.1 – Attributes and reasons for the proposed monitoring approach
The attributes for each aspect of the proposed monitoring approach and their influence on other aspects of the monitoring approach are shown in figure 9.1 and listed in table 9.1. The two primary aspects of the monitoring approach are the use of run-time verification and hybrid monitoring. These both suit certain aspects of embedded systems. Specifically, hybrid monitoring allows the internal operation of an embedded system to be examined with the interference to the target system dependent on the specific arrangement of hardware vs. software in the monitoring apparatus (important in the context of real-time embedded systems). Run-time verification provides guarantees about the correct operation of the system, which could potentially be important for embedded systems with strict operational parameters. At the same time, run-time verification allows partial monitoring of specific conditions, suitable for the limited monitoring that may be possible due to the hardware of the target system.

<table>
<thead>
<tr>
<th>Element of proposal</th>
<th>Attribute (*not inherent)</th>
<th>Supporting reason(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid monitoring</td>
<td>Sufficient observation of internals</td>
<td>Overcome limited visibility into internal operation of embedded systems</td>
</tr>
<tr>
<td></td>
<td>Can minimise overhead</td>
<td>Avoid interference, support real-time attributes of typical embedded systems</td>
</tr>
<tr>
<td>Run-time verification</td>
<td>Scalable</td>
<td>Monitoring of embedded system may be limited by hardware visibility</td>
</tr>
<tr>
<td></td>
<td>Formal basis</td>
<td>Embedded systems have strict operational parameters</td>
</tr>
<tr>
<td></td>
<td>Modular design*</td>
<td>Modularity is needed to support the hybrid monitoring approach</td>
</tr>
<tr>
<td>Target-host interface</td>
<td>Buffer output*</td>
<td>Minimise interference to target</td>
</tr>
<tr>
<td></td>
<td>High speed*</td>
<td>Minimise interference to target</td>
</tr>
<tr>
<td></td>
<td>On-chip*</td>
<td>Minimise interference to target</td>
</tr>
<tr>
<td></td>
<td>Deterministic*</td>
<td>Provide visibility into internal operation of system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Support real-time attributes of typical embedded systems</td>
</tr>
<tr>
<td>Host</td>
<td>Real-time system*</td>
<td>Necessary to support verification of real-time embedded systems and/or strict operational parameters</td>
</tr>
<tr>
<td></td>
<td>(Real-time software environment, suitable hardware)</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1 – List of elements and attributes of proposed monitoring approach
The proposed host and host-target interface also have relationship to common attributes of embedded systems and of hybrid monitoring as shown in figure 9.1. In order to minimise the overhead of the monitoring apparatus, the interface needs to be on-chip (also to allow sufficient observation), it needs to be of sufficient speed, and it should have a buffer for observations to be placed in by the monitoring code. For monitoring real-time and strict parameters, the interface needs to be deterministic and the host for run-time verification also needs to operate as a real-time system.

9.4 RESULTS OF EXPERIMENTAL WORK

The literature review contributed to the initial aims of the thesis set out in chapter 1, as described in section 9.2, with the results of experimental work further contributing to these aims. Specifically, the following aims were addressed:

- The feasibility of the proposed working solution was examined through the development of a working prototype and the development of case studies to explore the effectiveness of the approach.

- The effectiveness of the proposed solution was quantified in terms of system resource overheads, timing overheads and development effort through examination of the results of the case studies.

These results also form the basis of the conclusions in this chapter on the feasibility of the use of formal monitors for enhancing the reliability of embedded systems, the final aim set out in chapter 1.

9.4.1 Results from developing a working prototype

The initial experimental arrangement described in chapter 6 was used to allow Java-MaC to verify a condition for a Java application executing on the JOP platform (a deployment described in section 6.3). When the target application was executed with a deliberate error causing it to operate incorrectly, an alarm was raised by Java-MaC as soon as the monitored variable went outside the parameters specified in the Java-MaC specification files.

By providing a ‘skeleton’ application to act as a target for Java-MaC and also act as a recipient of monitoring data from the target application on JOP, the specified rules on Java-MaC could be checked as if Java-MaC was monitoring the application on the JOP core directly.
The achievement of these results showed some feasibility of the suggested monitoring approach by successfully arranging a working prototype that allowed run-time verification of software executing in an embedded system.

### 9.4.2 Results of railroad crossing case study

The case study of the railroad crossing scenario described in chapter 7 demonstrated that a similar example to that used for run-time verification on the PC could be verified on the JOP platform using the experimental monitoring arrangement. In writing a new version of the target application suited for the JOP platform, the scenario acquired real-time characteristics, as the various objects in the scenario were modelled as real-time threads of execution.

The preparation of the skeleton application for the railroad crossing example demonstrated some difficulties with the experimental monitoring arrangement in its initial design, as the control flow of the original application needed to be duplicated. This required manual adaptation of the original application code and would be too cumbersome an approach for anything beyond trivial examples. The initial skeleton application approach was thus reconsidered and a modified monitoring arrangement relying on a ‘relay application’ to interface with JOP was designed.

Scrutiny of the bytecode of the target application, before and after instrumentation, showed that seven instructions, comprising 12 bytecodes, had to be added to the target application at each instance of a monitored variable. These bytecodes are executed each time the monitoring point is reached in the target application execution.

### 9.4.3 Results of rate-monotonic scheduling analysis

The case study described in chapter 8 demonstrated implementation of a scenario comprising verification that a set of tasks is scheduled at run time as predicted by static offline analysis. The worst-case execution time (WCET) analysis of the target application code provided the expected number of processor cycles for particular tasks to complete execution of their main execution loop. Calculation based on data from this WCET analysis provided the required iterations to apply to nested loops in the target application tasks in order that they would execute for a particular duration. The specification of the verification scenario in Java-MaC was also demonstrated. Analysis was provided of the timer methods used by Java-MaC (it uses a host system
timer that has poor granularity), as well as the delays in communicating across the prototype monitoring configuration.

Testing provided data characterising the performance of the aspects of the prototype platform. Execution time of the test application was measured at run time and compared against the expected worst-case execution time from static analysis. The results highlighted a discrepancy between the WCET from analysis and the measured execution time in a multithreaded application. However, an exact match to WCET was recorded for the measured execution time of the work method when executed in a single thread with no scheduler overhead. On the basis of this discrepancy, some investigation of the JOP scheduler behaviour was undertaken by Schoeberl [96], providing the explanation that the scheduler was invoked if no new thread became ready within 10 ms. This timeout was modified to be one second, resulting in the expected execution time being measured in a multithreaded application.

Analysis of the delays in sending observations from the target application to Java-MaC on the host system provided data demonstrating that the host system performance can sometimes vary greatly from the average, adding delays in the order of milliseconds for particular observations. A means of minimising this by modifying the relay application was suggested, where the relay application is modified to directly read from the serial port.

9.4.4 Overhead of monitoring arrangement

Analysis of the experimental approach provided details of the overhead of the monitoring apparatus. Additional code was placed at each point that a monitored entity's state changed within the target application. This resulted in extra execution time for the target application as well as additional code to be loaded into the target system during setup. The use of the serial communication interface for communicating observations was not exclusive and as such did not require extra hardware to facilitate the monitoring apparatus. However, the non-exclusive use of the serial port was ensured by the use of a blocking wait on printing to the serial port. This resulted in the execution time of print statements being lengthened, as the main execution had to wait for bytes to be sent before continuing to execute. Also this placed a burden on the host to parse monitoring observations from a normal communication stream; the first character is checked to determine if it matches the
start of a monitoring observation. Ordinary communications are printed to screen but sending monitoring observations to the run-time verification system.

As shown in the analysis of section 7.5 of chapter 7, the execution time required for each monitoring statement is dependent on the number of bytes being communicated to the host and the communication speed of the serial port (with some overhead). The measured time to communicate one byte was 0.1 ms, with WCET analysis predicting an overhead of 0.132 ms for executing each print statement.

The observations denoting the start or end of a method, used in the rate monotonic scheduling verification of chapter 8, are five characters (bytes) long. These observations comprise a reference to which monitored entity the observation pertains and what the state change is (start or end of method). As a result, the total execution time of the additional code added to each thread of the target application of the rate monotonic scheduling example is 1.264 ms per period. This is comprised of (0.132 * 2) µs for the two print statements and (0.1 * 10) µs for the ten characters printed.

The storage necessary on the target system to hold the target application code is also increased by the monitoring apparatus. An additional 18 bytes of code are included at each monitoring point. For the target application of the rate-monotonic scheduler example, this resulted in 108 bytes of additional code in total across the entire application.

### 9.5 IMPROVED HOST SYSTEM USING JAVA-MAC/JOP

Some improvements were suggested for the approach described in chapter 8, to facilitate more reliable use of Java-MaC to indirectly verify an application executing on a JOP core instantiated on a Spartan 3 FPGA. Specifically, an improved host platform is necessary to provide system timer functionality to reason about timing of events on the order of milliseconds. Additionally, a real-time operating system and programming implementation of Java-MaC is necessary to ensure that events are resolved in real-time by Java-MaC and that performance is unaffected by other system tasks (e.g. operating system tasks) or for example the JVM's garbage collection.

Recent PC hardware does provide higher resolution system timer functionality such as HPET (High Precision Event Timer). Java-MaC could be ported to the Sun Java Real-Time System. With installation of one of the recommended operating systems to
support Java Real-Time, this could render a general purpose host system such as a standard PC sufficiently suited for the purposes of real-time verification.

Although not comprehensive or providing great assurance about real-time performance, some improvement in the timing performance of the experimental approach would also result solely from using more recent PC hardware and a newer general purpose operating system that supports the HPET timer, with utilisation of the Java API System.nanoTime that uses the highest precision system timer available to the operating system. Combined with modification of the relay application to directly read observations from the serial port this would allow verification of scenarios such as that presented in chapter 8 if the timing parameters were of longer duration (thus not affected by small variations in the delay in communicating from the target application to the host).

9.6 SUGGESTIONS FOR FURTHER WORK AND APPLICATIONS

Further to the improvements suggested above for the existing experimental approach and the proposals for a good monitoring solution, there are some innovations to which the general monitoring approach of this thesis could be applied. Some examples are:

- Incorporation of monitoring support at design time
- Lifetime monitoring

Potential applications might include:

- Automatic failsafe in event of any failure to verify a condition
- Using the monitoring approach for enhanced product support

9.6.1 Monitoring-oriented system design

The incorporation of design features to support monitoring increases the scope for sufficient observation of system operation by the monitoring apparatus. For example, embedded systems usually incorporate some basic features to support debugging, such as the JTAG debug port. This allows debugging of the system without temporarily modifying it in a custom manner for the purposes of observing its internal operation (such modifications would potentially affect the system's normal operation). Further to the suggestions in section 9.3 for a good monitoring approach for embedded systems, hardware support can minimise the overhead from adding
additional software monitoring into a target application. A dedicated monitoring apparatus within the target system hardware could be arranged such that it can read monitoring observations held in system memory, independently of the target system processor. A special monitoring port could then be used to access this unit in order that a host system can be connected and receive observations for the purpose of runtime verification.

### 9.6.2 Lifetime monitoring

An extension of the concept of adding support for monitoring at design time is to not only support the gathering of observations from the target system, but also incorporate the monitoring task itself within or permanently attached to the target system (i.e. no separate host system). This would allow any monitoring task to be undertaken for the lifetime of the product, providing permanent assurances about its correct operation. Such an approach would potentially require a second processing unit for the purposes of evaluating observations. Utilising the target system processor would potentially affect the software being monitored due to the overhead for the processor of performing the monitoring task. However, if the target software were designed to account for a monitoring task, then this approach could also be viable.

### 9.6.3 Using the monitoring approach for an automatic failsafe

Although the monitoring approach as is provides an output in the form of reporting any violations of conditions being verified, this output could be used for applications at run time. For example, if a system performs a safety critical role, the system may be designed with redundant units. Rather than rely solely on hardware fail-safes (i.e. switching to back-ups only when hardware is outside accepted parameters), the output of the monitoring approach could be used to switch to redundant back-up units if there is a failure detected in the operation of the system software. This would provide an additional level of assurance, not only that correct operation of the system will be verified, but that if it is not, then corrective action will be taken. Such an approach would require the concept of lifetime monitoring described in the preceding section, as well as the consideration of the monitoring apparatus at design time.
9.6.4 Potential application for enhanced product support

Although embedded products are often designed to a high standard, allowing for any degradation of hardware components over the lifetime of a product, the lifetime monitoring approach suggested in section 9.6.2 could be used to provide improved product support. Despite design standards, it is possible for software to operate sub-optimally in the field due to changes in the performance of hardware due to a harsh operating environment, degradation in component performance over time, or misuse of the product.

The manufacturer of an embedded system could offer an enhanced product support whereby an additional monitoring unit is attached to the product to perform run-time verification to ensure the product remains at its peak intended operating performance. The monitoring unit can then ‘phone home’ if it remains Internet-connected and if a valid support contract is in place, the manufacturer can replace the device and/or service it (or remove it for analysis of the root cause of the degradation in performance).

9.7 SUMMARY

This chapter has presented the results arising from a review of literature and experimental work, as well as detailing how these results achieved the aims set out in chapter 1. Additionally, based on the experiences in arranging these experiments, a monitoring arrangement for run-time verification of software executing in embedded systems has been proposed. Some recommendations on improving the performance of the experimental arrangement were also presented. Finally, suggestions for further work have been given, as well a potential application of the monitoring approach.
References


[49] ARM Ltd.: “Embedded Trace Macrocell ETMv1.0 to ETMv3.4”, architecture specification.


De Morgan, A. (1847) *Formal Logic or the Calculus of Inference*.


[101] Microsoft Corporation: “Computers that are running Windows XP Service Pack 2 and that are equipped with multiple processors that support processor power management features may experience decreased performance”, *MSDN Knowledge Base*, Article ID: 896256, (Online, accessed Nov 2009), http://support.microsoft.com/kb/896256


APPENDIX A

SPARTAN 3 DEVELOPMENT BOARD
APPENDIX B

COMPARISON OF MEASURED EXECUTION TIMES AGAINST WCET ANALYSIS
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>End</td>
<td>Start</td>
<td>End</td>
<td>Start</td>
<td>End</td>
<td>Cycles</td>
<td>Diff WCET</td>
<td>Cycles</td>
<td>Diff Ref</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>100520781</td>
<td>101775549</td>
<td>1254768</td>
<td>4579</td>
<td>1254803</td>
<td>4614</td>
<td>5008642</td>
<td>8642</td>
<td>5000091</td>
<td>91</td>
</tr>
<tr>
<td>101891232</td>
<td>108285672</td>
<td>108404531</td>
<td>12005684</td>
<td>1254768</td>
<td>4579</td>
<td>4999910</td>
<td>-90</td>
<td>4998031</td>
<td>-9169</td>
</tr>
<tr>
<td>1254768</td>
<td>4579</td>
<td>1254803</td>
<td>4614</td>
<td>5000055</td>
<td>55</td>
<td>9900862</td>
<td>-9138</td>
<td>3885630</td>
<td>135441</td>
</tr>
<tr>
<td>101891232</td>
<td>108285672</td>
<td>1254768</td>
<td>4579</td>
<td>1254803</td>
<td>4614</td>
<td>5000055</td>
<td>55</td>
<td>1400352</td>
<td>140163</td>
</tr>
<tr>
<td>1254768</td>
<td>4579</td>
<td>1254803</td>
<td>4614</td>
<td>5000055</td>
<td>55</td>
<td>9900862</td>
<td>-9138</td>
<td>3885630</td>
<td>135441</td>
</tr>
<tr>
<td>101891232</td>
<td>108285672</td>
<td>1254768</td>
<td>4579</td>
<td>1254803</td>
<td>4614</td>
<td>5000055</td>
<td>55</td>
<td>1400352</td>
<td>140163</td>
</tr>
<tr>
<td>B-3</td>
<td>235529425</td>
<td>236784193</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>------------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23189915</td>
<td>238294790</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22805480</td>
<td>240057262</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254768</td>
<td>4579</td>
<td>4989561</td>
<td>-10439</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5140107</td>
<td>139918</td>
<td>9898555</td>
<td>-10445</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886507</td>
<td>136318</td>
<td>19994807</td>
<td>-5193</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240518986</td>
<td>241773789</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>245529441</td>
<td>246784314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>241899470</td>
<td>248284335</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999980</td>
<td>-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5139992</td>
<td>139803</td>
<td>9898564</td>
<td>-10436</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3885883</td>
<td>135694</td>
<td>20003649</td>
<td>3649</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250529478</td>
<td>251784316</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255529458</td>
<td>256784226</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>251900215</td>
<td>258295065</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>244002187</td>
<td>260057147</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999980</td>
<td>-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5139811</td>
<td>139862</td>
<td>10010306</td>
<td>10396</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886517</td>
<td>136328</td>
<td>20002026</td>
<td>2026</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>260518226</td>
<td>261774064</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>265529423</td>
<td>266784226</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>261889779</td>
<td>268284393</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999980</td>
<td>-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5139811</td>
<td>139862</td>
<td>10010306</td>
<td>10396</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886517</td>
<td>136328</td>
<td>20002026</td>
<td>2026</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>270529479</td>
<td>271784282</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>275529440</td>
<td>276784313</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>271900085</td>
<td>278295181</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>268402936</td>
<td>280056155</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999980</td>
<td>-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5139811</td>
<td>139862</td>
<td>10010306</td>
<td>10396</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886517</td>
<td>136328</td>
<td>20002026</td>
<td>2026</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>280520762</td>
<td>281775635</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>285529441</td>
<td>286784314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>281813739</td>
<td>288286811</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>290529481</td>
<td>291784249</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>295529441</td>
<td>296784314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>291299956</td>
<td>29824895</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>288405962</td>
<td>300057661</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999980</td>
<td>-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5140550</td>
<td>140370</td>
<td>10005877</td>
<td>8577</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886285</td>
<td>136096</td>
<td>19995877</td>
<td>-1423</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300519038</td>
<td>301773911</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>305529425</td>
<td>306784193</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30188958</td>
<td>308285270</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>310529479</td>
<td>311784282</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>315529422</td>
<td>316784225</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>311900268</td>
<td>318295450</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>308405339</td>
<td>320057116</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999943</td>
<td>-57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5140544</td>
<td>140355</td>
<td>10010310</td>
<td>10310</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886606</td>
<td>136417</td>
<td>19995665</td>
<td>-4335</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>320518713</td>
<td>321773566</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>325529441</td>
<td>326784314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32188403</td>
<td>328284863</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>330529478</td>
<td>331784316</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>335529423</td>
<td>336784225</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>331900128</td>
<td>338294689</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32840204</td>
<td>340051757</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254838</td>
<td>4649</td>
<td>4999945</td>
<td>-55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5139578</td>
<td>139569</td>
<td>989142</td>
<td>-10858</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3886342</td>
<td>136153</td>
<td>20000307</td>
<td>307</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>340519141</td>
<td>341773944</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>345529460</td>
<td>346784228</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>341889270</td>
<td>348284467</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>350529514</td>
<td>351784387</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>355529441</td>
<td>356784314</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>351900430</td>
<td>358295813</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>344000511</td>
<td>360053907</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1254803</td>
<td>4614</td>
<td>5010707</td>
<td>10707</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5140494</td>
<td>140305</td>
<td>1001164</td>
<td>11164</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3887907</td>
<td>136908</td>
<td>19999864</td>
<td>-136</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>370529478</td>
<td>371784316</td>
<td>1254838</td>
<td>4649</td>
<td>4999945</td>
<td>-55</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>375529423</td>
<td>376784226</td>
<td>1254803</td>
<td>4614</td>
<td>4991191</td>
<td>-8809</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>371900304</td>
<td>381295541</td>
<td>364800375</td>
<td>380052819</td>
<td>5140343</td>
<td>140245</td>
<td>9990867</td>
<td>-9133</td>
<td></td>
<td></td>
</tr>
<tr>
<td>380520614</td>
<td>381775487</td>
<td>1254873</td>
<td>4684</td>
<td>5008826</td>
<td>8826</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>385529440</td>
<td>386784313</td>
<td>1254873</td>
<td>4684</td>
<td>5000057</td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>381991171</td>
<td>388286428</td>
<td>1254768</td>
<td>4579</td>
<td>4999963</td>
<td>-37</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>390529497</td>
<td>391784265</td>
<td>1254768</td>
<td>4579</td>
<td>4990035</td>
<td>-9653</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>395529460</td>
<td>396784228</td>
<td>291900182</td>
<td>398295584</td>
<td>1400444</td>
<td>140255</td>
<td>9990007</td>
<td>-9993</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400519495</td>
<td>401774296</td>
<td>1254803</td>
<td>4614</td>
<td>5009928</td>
<td>9928</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>405529423</td>
<td>406784226</td>
<td>1254803</td>
<td>4614</td>
<td>5000092</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>401990189</td>
<td>408286526</td>
<td>1254803</td>
<td>4614</td>
<td>4999926</td>
<td>-74</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410529515</td>
<td>411784318</td>
<td>1254803</td>
<td>4614</td>
<td>4991911</td>
<td>-8089</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>415529441</td>
<td>416784314</td>
<td>1254803</td>
<td>4614</td>
<td>4999963</td>
<td>-37</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>411900056</td>
<td>418295584</td>
<td>408400376</td>
<td>420007067</td>
<td>5140186</td>
<td>139997</td>
<td>9991725</td>
<td>-8275</td>
<td></td>
<td></td>
</tr>
<tr>
<td>420521352</td>
<td>421776190</td>
<td>1254838</td>
<td>4649</td>
<td>5008089</td>
<td>8089</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>425529441</td>
<td>426784314</td>
<td>1254873</td>
<td>4684</td>
<td>5000056</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>421891781</td>
<td>428286751</td>
<td>1254768</td>
<td>4579</td>
<td>4999925</td>
<td>-75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>430529497</td>
<td>431784265</td>
<td>1254803</td>
<td>4614</td>
<td>4989823</td>
<td>-10177</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>435529422</td>
<td>436784225</td>
<td>1254803</td>
<td>4614</td>
<td>4989823</td>
<td>-10177</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>431899929</td>
<td>438294477</td>
<td>428400596</td>
<td>440057584</td>
<td>5140043</td>
<td>139854</td>
<td>9990001</td>
<td>-9999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>440519425</td>
<td>441774048</td>
<td>1254803</td>
<td>4614</td>
<td>5010179</td>
<td>10179</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>445529424</td>
<td>446784192</td>
<td>1254873</td>
<td>4684</td>
<td>5000092</td>
<td>92</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44189930</td>
<td>448285180</td>
<td>1254803</td>
<td>4614</td>
<td>4999906</td>
<td>-94</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>450529516</td>
<td>451784389</td>
<td>1254803</td>
<td>4614</td>
<td>4989680</td>
<td>-10320</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>455529422</td>
<td>456784225</td>
<td>1254803</td>
<td>4614</td>
<td>4989680</td>
<td>-10320</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451900061</td>
<td>458295818</td>
<td>448400508</td>
<td>460057272</td>
<td>5140354</td>
<td>140165</td>
<td>9989144</td>
<td>-10856</td>
<td></td>
<td></td>
</tr>
<tr>
<td>460519102</td>
<td>461773940</td>
<td>1254838</td>
<td>4649</td>
<td>5010339</td>
<td>10339</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>465529441</td>
<td>466784314</td>
<td>1254873</td>
<td>4684</td>
<td>5000056</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>461889805</td>
<td>468284949</td>
<td>1254803</td>
<td>4614</td>
<td>4999925</td>
<td>-75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>470529497</td>
<td>471784265</td>
<td>1254803</td>
<td>4614</td>
<td>4991179</td>
<td>-8821</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>475529422</td>
<td>476784225</td>
<td>471900105</td>
<td>47895131</td>
<td>140223</td>
<td>140034</td>
<td>9991290</td>
<td>-8710</td>
<td></td>
<td></td>
</tr>
<tr>
<td>468020881</td>
<td>48177509</td>
<td>1254908</td>
<td>4719</td>
<td>5008821</td>
<td>8821</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>485529422</td>
<td>486784225</td>
<td>1254803</td>
<td>4614</td>
<td>5000058</td>
<td>58</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>481901395</td>
<td>488286460</td>
<td>1254803</td>
<td>4614</td>
<td>4999942</td>
<td>-58</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>490529480</td>
<td>491784318</td>
<td>1254803</td>
<td>4614</td>
<td>4991596</td>
<td>-8404</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>495529422</td>
<td>496784225</td>
<td>491900409</td>
<td>498295816</td>
<td>1405604</td>
<td>140415</td>
<td>9991282</td>
<td>-8708</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500521018</td>
<td>501775856</td>
<td>1254838</td>
<td>4649</td>
<td>5008404</td>
<td>8404</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>505529422</td>
<td>506784225</td>
<td>1254803</td>
<td>4614</td>
<td>5000094</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C

COMPARISON OF EXECUTION TIMES

MEASURED BY JOP/RELAY
<table>
<thead>
<tr>
<th>Start T1</th>
<th>End T1</th>
<th>Start T2</th>
<th>End T2</th>
<th>Start T3</th>
<th>End T3</th>
<th>T1 Exec. Time</th>
<th>T1 Period</th>
<th>T2 Exec. Time</th>
<th>T2 Period</th>
<th>T3 Exec. Time</th>
<th>T3 Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>120519751</td>
<td>1255973462171</td>
<td>121853497</td>
<td>1255973462203</td>
<td>121923093</td>
<td>1255973462203</td>
<td>26.67492</td>
<td>32</td>
<td>100.1826</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125528881</td>
<td>1255973462265</td>
<td>126862539</td>
<td>1255973462296</td>
<td>126845829</td>
<td>1255973462328</td>
<td>26.67316</td>
<td>31</td>
<td>99.84568</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130521165</td>
<td>1255973462375</td>
<td>131854729</td>
<td>1255973462390</td>
<td>131904729</td>
<td>1255973462390</td>
<td>26.67128</td>
<td>15</td>
<td>100.15396</td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>135528883</td>
<td>1255973462468</td>
<td>138862479</td>
<td>1255973462500</td>
<td>138446199</td>
<td>1255973462531</td>
<td>26.67232</td>
<td>32</td>
<td>99.79399</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140518558</td>
<td>1255973462562</td>
<td>141852088</td>
<td>1255973462593</td>
<td>141922515</td>
<td>1255973462593</td>
<td>26.6706</td>
<td>31</td>
<td>100.20611</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>145528883</td>
<td>1255973462671</td>
<td>148862412</td>
<td>1255973462703</td>
<td>148443999</td>
<td>1255973462734</td>
<td>26.67088</td>
<td>32</td>
<td>99.81736</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150519731</td>
<td>1255973462765</td>
<td>151853314</td>
<td>1255973462796</td>
<td>151923741</td>
<td>1255973462796</td>
<td>26.67166</td>
<td>31</td>
<td>100.183</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>155528883</td>
<td>1255973462875</td>
<td>156862358</td>
<td>1255973462900</td>
<td>158445205</td>
<td>1255973462921</td>
<td>26.66954</td>
<td>15</td>
<td>99.79118</td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160519440</td>
<td>1255973462968</td>
<td>161851517</td>
<td>1255973463000</td>
<td>161921949</td>
<td>1255973463000</td>
<td>26.66154</td>
<td>32</td>
<td>100.20864</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>165528882</td>
<td>1255973463082</td>
<td>168862296</td>
<td>1255973463035</td>
<td>168443046</td>
<td>1255973463125</td>
<td>26.66828</td>
<td>31</td>
<td>99.78868</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>170518316</td>
<td>1255973463171</td>
<td>171851890</td>
<td>1255973463203</td>
<td>171922325</td>
<td>1255973463203</td>
<td>26.67148</td>
<td>32</td>
<td>100.21166</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>175528889</td>
<td>1255973463265</td>
<td>176862649</td>
<td>1255973463296</td>
<td>178444211</td>
<td>1255973463328</td>
<td>26.675</td>
<td>31</td>
<td>99.82754</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180267480</td>
<td>1255973463359</td>
<td>182367480</td>
<td>1255973463359</td>
<td>182367480</td>
<td>1255973463359</td>
<td>26.675</td>
<td>31</td>
<td>99.82754</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18052076 1255973463375</td>
<td>181853982 1255973463390</td>
<td>26.67412 15 100.17174 93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>185528883 1255973463468</td>
<td>188862935 1255973463500</td>
<td>26.67464 32 99.85506 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>190521616 1255973463562</td>
<td>191855220 1255973463592</td>
<td>26.67286 31 100.14496 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195528884 1255973463671</td>
<td>198862527 1255973463703</td>
<td>26.67326 32 99.79436 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200518582 1255973463765</td>
<td>201852131 1255973463796</td>
<td>26.67098 31 100.20638 110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>205528901 1255973463875</td>
<td>208862466 1255973463890</td>
<td>26.6713 15 99.80296 93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210510449 1255973463968</td>
<td>211852063 1255973464000</td>
<td>26.66028 32 100.19632 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215528885 1255973464082</td>
<td>218862402 1255973464093</td>
<td>26.6704 31 99.82776 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>220520253 1255973464171</td>
<td>221853729 1255973464203</td>
<td>26.66952 32 100.17222 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>225528884 1255973464265</td>
<td>228863231 1255973464296</td>
<td>26.66934 31 99.84104 110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>230520916 1255973464375</td>
<td>231854534 1255973464393</td>
<td>26.6736 15 100.15966 93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>235528899 1255973464468</td>
<td>238862285 1255973464500</td>
<td>26.66732 32 99.81906 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240518582 1255973464562</td>
<td>241853601 1255973464592</td>
<td>26.67486 31 100.18088 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>245528901 1255973464671</td>
<td>248863072 1255973464703</td>
<td>26.68342 32 99.8553 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18052076 1255973463375</td>
<td>181853982 1255973463390</td>
<td>103.76295 109 200.03328 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>185528883 1255973463468</td>
<td>188862935 1255973463500</td>
<td>103.76436 109 199.92958 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>190521616 1255973463562</td>
<td>191855220 1255973463592</td>
<td>200371659 1255973463595</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195528884 1255973463671</td>
<td>198862527 1255973463703</td>
<td>103.74928 110 199.9988 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200518582 1255973463765</td>
<td>201852131 1255973463796</td>
<td>208514149 1255973463921</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>205528901 1255973463875</td>
<td>208862466 1255973463890</td>
<td>103.75822 94 200.03328 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210510449 1255973463968</td>
<td>211852063 1255973464000</td>
<td>78.56866 62 309.94558 390</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>215528885 1255973464082</td>
<td>218862402 1255973464093</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>220520253 1255973464171</td>
<td>221853729 1255973464203</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>225528884 1255973464265</td>
<td>228863231 1255973464296</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>230520916 1255973464375</td>
<td>231854534 1255973464393</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>235528899 1255973464468</td>
<td>238862285 1255973464500</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240519982 1255973464562</td>
<td>241853601 1255973464592</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>245528901 1255973464671</td>
<td>248863072 1255973464703</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256521066</td>
<td>1255973464765</td>
<td>248516921</td>
<td>1255973464734</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>251885281</td>
<td>1255973464736</td>
<td>251256897</td>
<td>1255973464736</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>255528899</td>
<td>1255973464875</td>
<td>268462024</td>
<td>1255973464921</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>260518183</td>
<td>1255973464968</td>
<td>260371287</td>
<td>1255973464968</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>265528885</td>
<td>1255973465062</td>
<td>269444621</td>
<td>1255973465125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>270519399</td>
<td>1255973465171</td>
<td>269514622</td>
<td>1255973465125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>275528881</td>
<td>1255973465265</td>
<td>270519399</td>
<td>1255973465328</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>280519973</td>
<td>1255973465375</td>
<td>280398992</td>
<td>1255973465359</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>285528881</td>
<td>1255973465468</td>
<td>281922914</td>
<td>1255973465390</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>290519715</td>
<td>1255973465562</td>
<td>291923708</td>
<td>1255973465593</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>295528881</td>
<td>1255973465671</td>
<td>296822917</td>
<td>1255973465703</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300518594</td>
<td>1255973465765</td>
<td>301922786</td>
<td>1255973465796</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>305528889</td>
<td>1255973465875</td>
<td>308826893</td>
<td>1255973465892</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>310520481</td>
<td>1255973465968</td>
<td>311924881</td>
<td>1255973466000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>315528885</td>
<td>1255973466162</td>
<td>316826817</td>
<td>1255973466020</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
<td>E1</td>
<td>F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>318446777</td>
<td>1255973466125</td>
<td>320370002</td>
<td>1255973466171</td>
<td>103.76288</td>
<td>94</td>
<td>199.92958</td>
<td>203</td>
<td>78.57664</td>
<td>93</td>
<td>399.96288</td>
<td>407</td>
</tr>
<tr>
<td>320517802</td>
<td>1255973466171</td>
<td>321850932</td>
<td>1255973466171</td>
<td>26.6626</td>
<td>32</td>
<td>100.22126</td>
<td>94</td>
<td>320513818</td>
<td>1255973466328</td>
<td>26.67392</td>
<td>32</td>
</tr>
<tr>
<td>332037002</td>
<td>1255973466400</td>
<td>3321850932</td>
<td>1255973466468</td>
<td>26.6694</td>
<td>31</td>
<td>100.17676</td>
<td>109</td>
<td>3321850932</td>
<td>1255973466468</td>
<td>26.6712</td>
<td>31</td>
</tr>
<tr>
<td>339520497</td>
<td>1255973467375</td>
<td>3381859999</td>
<td>1255973467390</td>
<td>26.67004</td>
<td>15</td>
<td>100.16734</td>
<td>93</td>
<td>3381859999</td>
<td>1255973467390</td>
<td>26.67004</td>
<td>15</td>
</tr>
<tr>
<td>Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38882804 125597346750</td>
<td>38848324 125597346751</td>
<td>26.6748 32 99.8570 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>388516882 125597346751</td>
<td></td>
<td>103.7604 109 200.0314 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>390521716 1255973467562</td>
<td>391853241 1255973467593</td>
<td>26.6705 31 100.1437 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>391928089 1255973467593</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>395289901 1255973467671</td>
<td>396825245 1255973467720</td>
<td>26.6726 32 99.7937 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>398447981 1255973467794</td>
<td>400371680 1255973467795</td>
<td>103.7646 109 199.9268 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400516590 1255973467765</td>
<td>401852138 1255973467796</td>
<td>78.5706 62 399.9027 390</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40528881 1255973467890</td>
<td>40862475 1255973467930</td>
<td>26.6718 15 99.8129 31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40844031 1255973467921</td>
<td>408515021 1255973467921</td>
<td>403.7528 110 200.0245 204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410519791 1255973467968</td>
<td>411923799 1255973468000</td>
<td>26.6718 32 100.1823 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41528900 1255973468062</td>
<td>41982442 1255973468090</td>
<td>26.6704 31 99.8112 129</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41844563 1255973468125</td>
<td>420389388 1255973468171</td>
<td>103.7548 94 199.9878 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>420519962 1255973468171</td>
<td>421923312 1255973468203</td>
<td>78.5777 93 400.0148 407</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42652863 1255973468265</td>
<td>42962355 1255973468296</td>
<td>26.675 32 100.1865 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42944525 1255973468328</td>
<td>430518761 1255973468328</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>430520153 1255973468375</td>
<td>431941155 1255973468390</td>
<td>26.6704 15 100.167 93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43528883 1255973468468</td>
<td>43882292 1255973468500</td>
<td>26.6705 31 100.1842 109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>438446007 1255973468531</td>
<td>440393700 1255973468562</td>
<td>103.7606 109 199.9815 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>440519678 1255973468562</td>
<td>44183131 1255973468593</td>
<td>78.5689 78 400.0059 406</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>441923621 1255973468593</td>
<td>44445507 1255973468734</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>445288899 1255973468771</td>
<td>44682659 1255973468720</td>
<td>26.6752 32 99.8302 94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>446445507 1255973468734</td>
<td>448516060 125597346874</td>
<td>103.76252 109 200.0332 203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>450220815 1255973468765</td>
<td>451854847 1255973468796</td>
<td>26.6806 31 100.1610 110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>451925284 1255973468796</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
45528886 1255973468875
468825937 1255973468921
458447169 1255973468921
450370860 1255973468921
450370860 1255973468921
26.67462 15 99.78676 93
103.76308 110 199.94664 204
78.56892 78 399.99042 391
460518204 1255973468921
461852139 1255973469913
461922616 1255973469913
461922616 1255973469913
26.67146 32 100.21354 94
103.76498 94 200.0422 203
465528881 1255973469925
468825253 1255973469925
468444517 1255973469925
468444517 1255973469925
26.67304 31 99.81964 109
103.76435 94 200.0422 203
470519989 1255973469931
471853856 1255973469931
471924726 1255973469931
471924726 1255973469931
26.67462 15 99.78676 93
103.76308 110 199.94664 204
78.56892 78 399.99042 391
475528889 1255973469947
476852467 1255973469947
479466111 1255973469947
479466111 1255973469947
26.67136 31 99.82284 110
103.76034 94 199.99006 187
78.56032 77 400.03198 408
480520041 1255973469954
481855797 1255973469954
481924229 1255973469954
481924229 1255973469954
26.67512 15 100.17682 93
103.76748 109 200.03304 203
485528882 1255973469961
486852396 1255973469961
486446171 1255973469961
486446171 1255973469961
26.67462 15 100.21354 94
103.76498 94 200.0422 203
490521413 1255973469971
491855409 1255973469971
491925881 1255973469971
491925881 1255973469971
26.68094 31 100.14972 109
103.76032 109 199.98142 203
495528899 1255973469987
496852345 1255973469987
496447343 1255973469987
496447343 1255973469987
26.68802 32 99.83388 94
103.76032 109 199.98142 203
500520099 1255973469994
501854277 1255973469994
501924552 1255973469994
501924552 1255973469994
26.67868 31 100.18654 110
103.76032 110 200.04222 204
505528864 1255973470000
506852269 1255973470000
506446847 1255973470000
506446847 1255973470000
26.68844 32 100.134 94
103.76032 110 200.04222 204
510522165 1255973470016
511851617 1255973470016
511927063 1255973470016
511927063 1255973470016
26.68802 32 100.14972 109
103.76032 109 199.98142 203
515528865 1255973470023
516852236 1255973470023
518448001 1255973470023
518448001 1255973470023
26.67542 31 99.77842 109
103.77034 94 199.94942 203
520517786 1255973470039
521851381 1255973470039
520373938 1255973470039
520373938 1255973470039
26.6719 32 100.2219 94
78.55736 93 399.93698 407
<table>
<thead>
<tr>
<th>Account Number</th>
<th>Date</th>
<th>Amount 1</th>
<th>Amount 2</th>
<th>Balance 1</th>
<th>Balance 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>525529888</td>
<td>1255973470265</td>
<td>52882575</td>
<td>1255973470296</td>
<td>528443702</td>
<td>1255973470328</td>
</tr>
<tr>
<td>521921809</td>
<td>1255973470203</td>
<td>26.67388</td>
<td>31  99.8037</td>
<td>110  103.76398 94  200.02466 187</td>
<td></td>
</tr>
<tr>
<td>530519086</td>
<td>1255973470375</td>
<td>531852617</td>
<td>1255973470390</td>
<td>531923042</td>
<td>1255973470390</td>
</tr>
<tr>
<td>535288855</td>
<td>1255973470468</td>
<td>53662526</td>
<td>1255973470500</td>
<td>536444935</td>
<td>1255973470531</td>
</tr>
<tr>
<td>540518349</td>
<td>1255973470562</td>
<td>54185256</td>
<td>1255973470593</td>
<td>541922976</td>
<td>1255973470593</td>
</tr>
<tr>
<td>545288865</td>
<td>1255973470671</td>
<td>548682453</td>
<td>1255973470703</td>
<td>548444863</td>
<td>1255973470734</td>
</tr>
<tr>
<td>550520181</td>
<td>1255973470765</td>
<td>551853785</td>
<td>1255973470796</td>
<td>551924204</td>
<td>1255973470796</td>
</tr>
<tr>
<td>555288864</td>
<td>1255973470875</td>
<td>55662388</td>
<td>1255973470890</td>
<td>558445866</td>
<td>1255973470921</td>
</tr>
<tr>
<td>560519341</td>
<td>1255973470968</td>
<td>561852856</td>
<td>1255973471000</td>
<td>561923375</td>
<td>1255973471000</td>
</tr>
<tr>
<td>555288866</td>
<td>1255973471062</td>
<td>568682326</td>
<td>1255973471093</td>
<td>568444741</td>
<td>1255973471125</td>
</tr>
<tr>
<td>565230063</td>
<td>1255973471171</td>
<td>571853654</td>
<td>1255973471203</td>
<td>571924083</td>
<td>1255973471203</td>
</tr>
<tr>
<td>575528882</td>
<td>1255973471265</td>
<td>576882262</td>
<td>1255973471296</td>
<td>578445539</td>
<td>1255973471328</td>
</tr>
<tr>
<td>580519306</td>
<td>1255973471375</td>
<td>581852718</td>
<td>1255973471390</td>
<td>581923576</td>
<td>1255973471390</td>
</tr>
<tr>
<td>585528899</td>
<td>1255973471468</td>
<td>588853057</td>
<td>1255973471500</td>
<td>588446332</td>
<td>1255973471531</td>
</tr>
<tr>
<td>590522149</td>
<td>1255973471562</td>
<td>588517326</td>
<td>1255973471531</td>
<td>588517326</td>
<td>1255973471531</td>
</tr>
<tr>
<td>Code</td>
<td>Name</td>
<td>Phone</td>
<td>Email</td>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>C-9</td>
<td>Name1</td>
<td>Phone1</td>
<td>Email1</td>
<td>Address1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name2</td>
<td>Phone2</td>
<td>Email2</td>
<td>Address2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name3</td>
<td>Phone3</td>
<td>Email3</td>
<td>Address3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name4</td>
<td>Phone4</td>
<td>Email4</td>
<td>Address4</td>
<td></td>
</tr>
</tbody>
</table>