Run Time Detection of Timing Errors in Real-Time Systems

Osmar Marchi dos Santos

Submitted for the degree of Doctor of Philosophy

University of York
Department of Computer Science

October 2008
Abstract

Real-time systems comprise a class of computing systems whose functionality is interrelated to the timing constraints of the system, which are normally specified by deadlines. In order to guarantee that the deadlines of the system are going to be met during the runtime execution of the system, real-time systems often use feasibility analysis to test if the system meets all its deadlines. One crucial phase in the development of a real-time system, required for the feasibility analysis, is the derivation of the upper bounds of executions times for the threads in the system. This execution time upper bound is commonly known as the Worst-Case Execution Time (WCET).

Unfortunately, it can happen that, during the runtime execution of the system, a thread can execute more than the time initially estimated – an execution overrun occurs. The overrun of a thread may lead to a deadline being missed by the thread or even other threads in the system. From a fault tolerance perspective, both execution time overruns and deadline misses can be considered timing errors that could potentially cause a failure in the system’s functionality. In other words, a timing error can develop when the assumptions used in the feasibility analysis of the system are undermined during the system’s runtime execution. Indeed, in such context, the ideal is to detect the error in the system as soon as possible, so that the propagation of the error can be limited and error recovery strategies can take place with more accurate information.

This thesis argues that different runtime monitoring mechanisms for error detection could be used for the earlier detection of timing errors, before the occurrence of deadline misses. Moreover, where applicable, this approach could provide mechanisms to confine the temporal damage to the errant threads of the system, as well as facilitate error recovery strategies. The work focuses on real-time system executing in a uniprocessor architecture, using the Fixed-Priority Scheduling (FPS) approach and with the possibility of resources
being shared amongst threads (this can cause blocking time, where a higher priority thread is blocked whilst a lower priority thread executes). For such system, it is considered that the following timing errors could develop during run time: (i) deadline miss of a thread; (ii) overrun of a thread’s WCET; (iii) overrun of a thread’s WCET inside a shared resource; (iv) violation of the Minimum Inter-arrival Time (MIT) assumptions for a sporadic thread.

With these timing errors in mind, the research presents possible error detection and, where applicable, damage confinement mechanisms that could be used to help maintain the feasibility analysis of the system during run time. The mechanisms are addressed using two different technologies, the POSIX/C standard and the Real-Time Specification for Java (RTSJ). As well as providing a better understanding of timing error detection and temporal isolation mechanisms in both POSIX/C and RTSJ, this thesis provides three other major contributions. Firstly, a technique is developed to monitor the blocking time and, consequently, detect blocking time violations in POSIX/C. This provides a new perspective, with respect to the detection and possible actions upon blocking time violations, than previously found in the literature. Secondly, a formal model for the evaluation of the execution time monitoring mechanism of the RTSJ is presented. This resulted in the discovery of a bug in this mechanism of the RTSJ and the proposal of a new mechanism. The third contribution is the development of a blocking time monitoring mechanism for the RTSJ. This provides the ability to detect blocking time violations, something not previously possible in the RTSJ.

The blocking time monitoring mechanism for POSIX/C is implemented and evaluated in the open-source real-time operating system MaRTE OS. Moreover, an open-source runtime support platform for the RTSJ, which executes in a bare machine, is provided in this research. This platform is then used to implement and evaluate the proposed execution and blocking time monitoring mechanisms for the RTSJ. Performance figures for the proposed mechanisms are demonstrated.
Abstract

Acknowledgements

Declaration

1 Introduction

1.1 Fault Tolerance

1.2 Timing Errors

1.2.1 Run Time Error Detection

1.2.2 Run Time Damage Confinement

1.3 Thesis Hypothesis and Contributions

1.4 Thesis Outline

2 Timing Errors in Real-Time Systems

2.1 Response Time Analysis
2.1.1 Priority Inheritance Protocols ........................................ 38
2.1.2 Support for Aperiodic and Sporadic Threads using Servers ...... 40
2.2 Fault Tolerance ............................................................. 42
  2.2.1 Dependability ......................................................... 43
  2.2.2 Achieving Fault Tolerance .......................................... 46
    2.2.2.1 Error Detection .............................................. 47
    2.2.2.2 Damage Confinement and Assessment ....................... 48
    2.2.2.3 Error Recovery ............................................. 49
    2.2.2.4 Fault Treatment and Continued Service ..................... 49
2.3 Fault Tolerance for Timing Errors ...................................... 50
  2.3.1 Detection and Confinement of Execution Time Overruns .......... 51
    2.3.1.1 The Ada Programming Language ............................ 52
    2.3.1.2 The Chimera Real-Time Operating System .................. 53
    2.3.1.3 Time-Aware Fault-Tolerant Scheduling ...................... 54
    2.3.1.4 An Extension to the RTSJ ................................... 55
  2.3.2 Mode Changes for Assessment and Recovery Strategies .......... 57
2.4 Summary ................................................................. 58

3 Run Time Error Detection Models for Timing Errors ................. 59
  3.1 Computational Model Assumptions ..................................... 60
  3.2 Timing Errors .......................................................... 61
  3.3 The Portable Operating System Interface ............................. 64
    3.3.1 A Model for Error Detection ................................... 67
  3.4 The Real-Time Specification for Java ................................ 69
    3.4.1 A Model for Error Detection ................................... 75
  3.5 Summary ............................................................... 77
4 Timing Error Detection in the POSIX/C

4.1 Detection of Deadline Misses ........................................... 82
4.2 Detection of Execution Time Overruns ................................. 84
4.3 Detection of MIT Violations ............................................. 85
   4.3.1 Support for Aperiodic and Sporadic Computation Units ......... 87
4.4 Detection of Blocking Time Violations ................................. 88
   4.4.1 Identification of Blocking Time during Run Time ................. 90
   4.4.2 Detecting Violations ............................................. 98
4.5 Summary ............................................................... 99

5 A Formal Model of the RTSJ

5.1 Real-Time Threads ..................................................... 101
5.2 Modelling Architecture ................................................. 104
5.3 Formal Model of Periodic Real-Time Threads ........................ 106
5.4 Formal Model of Aperiodic and Sporadic Real-Time Threads ....... 110
5.5 Formal Model of Release Parameters Changes ........................ 115
5.6 Formal Evaluation ..................................................... 122
5.7 Summary ............................................................... 128

6 Timing Error Detection in the RTSJ

6.1 Deadline Monitoring .................................................... 132
6.2 Cost Monitoring and Enforcement ..................................... 133
   6.2.1 Modelling Architecture .......................................... 136
   6.2.2 Extended Formal Models of Real-Time Threads ................. 138
   6.2.3 Formal Model of the Cost Monitoring ............................ 142
   6.2.4 Formal Evaluation ............................................. 146
   6.2.5 A Revised Cost Monitoring and Enforcement Model .......... 158
List of Figures

2.1 The dependability tree (taken from [14]). . . . . . . . . . . . . . . . . . . . 46
2.2 The TAFT scheduling architecture (taken from [51]). . . . . . . . . . . . . 54
3.1 Example of a periodic thread in POSIX/C. . . . . . . . . . . . . . . . . . . 67
3.2 POSIX/C error detection model for timing errors. . . . . . . . . . . . . . 68
3.3 The run() method of a real-time thread. . . . . . . . . . . . . . . . . . . 74
3.4 An abridged RealtimeThread class specification. . . . . . . . . . . . . . 74
3.5 RTSJ error detection model for timing errors. . . . . . . . . . . . . . . . . 77
4.1 POSIX/C error detection model for timing errors. . . . . . . . . . . . . . 82
4.2 Detection of deadline miss for a periodic thread model. . . . . . . . . . . 83
4.3 Detection of execution time overrun for a periodic thread model. . . . . 85
4.4 Detection of MIT violations in POSIX/C. . . . . . . . . . . . . . . . . . . 86
4.5 Structure sched_param, used to set-up the POSIX/C sporadic server. . . . 88
4.6 Problem with thread self-suspending inside a shared resource. . . . . . 92
4.7 Detection of blocking time overrun for a periodic thread model. 99

5.1 Semantics for real-time threads (based on [37]). 103
5.2 Modelling architecture for real-time threads. 106
5.3 (a) RTT_SD and (b) RTT_wFNR automata. 107
5.4 (a) RTT_SDI and (b) RTT_Main automata. 108
5.5 RTT_PRD automaton. 110
5.6 TIME automaton. 112
5.7 (a) ASRTT_GA, (b) ASRTT_GAI and (c) ASRTT_ARD automata. 113
5.8 Extended automata RTT_SDI (a) and RTT_Main (b). 117
5.9 Extended automaton ASRTT_GA. 118
5.10 Extended automaton ASRTT_ARD. 119
5.11 Extended automaton ASRTT_GAI. 120
5.12 Extended automaton PRTT_PRD. 120
5.13 Extended automaton PRTT_PRD. 121
5.14 Extended automaton ASRTT_GAI. 122

6.1 RTSJ error detection model for timing errors. 132
6.2 Release parameter class and subclasses. 133
6.3 Requirements and definitions for cost monitoring (based on [18]). 135
6.4 Modelling architecture for the cost monitoring model. 138
6.5 Extended RTT_Main automaton. 139
6.6 Extended PRTT_RD automaton. 140
6.7 Extended (a) ASRTT_GAI and (b) ASRTT_ARD automata. 141
6.8 CM_Cost automaton. 142
6.9 (a) CM_CI, (b) CM_Main and (c) CM_Counter automata. 143
6.10 CM_MR automaton. 145
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.11</td>
<td>Counter-example generated for property e2.</td>
</tr>
<tr>
<td>6.12</td>
<td>Possible anomalous in the cost enforcement model of the RTSJ.</td>
</tr>
<tr>
<td>6.13</td>
<td>Problems with only considering a cost per period enforcement.</td>
</tr>
<tr>
<td>6.14</td>
<td>Example of the proposed solution.</td>
</tr>
<tr>
<td>6.15</td>
<td>Requirements and definitions for the revised cost monitoring (part 1).</td>
</tr>
<tr>
<td>6.16</td>
<td>Requirements and definitions for the revised cost monitoring (part 2).</td>
</tr>
<tr>
<td>6.17</td>
<td>Revised (a) CM_CI, (b) CM_Main and (c) CM_Counter automata.</td>
</tr>
<tr>
<td>6.18</td>
<td>Requirements and definitions for blocking time monitoring.</td>
</tr>
<tr>
<td>7.1</td>
<td>The jRate architecture (taken from [31]).</td>
</tr>
<tr>
<td>7.2</td>
<td>Original MaRTE architecture for C applications (taken from [59]).</td>
</tr>
<tr>
<td>7.3</td>
<td>MaRTE architecture for C applications under Linux (taken from [60]).</td>
</tr>
<tr>
<td>7.4</td>
<td>Overhead for different settings of the PI experiment in the POSIX/C.</td>
</tr>
<tr>
<td>7.5</td>
<td>Overhead for different settings of the PP experiment in the POSIX/C.</td>
</tr>
<tr>
<td>7.6</td>
<td>Abridged ReleaseParameters class with cost monitoring.</td>
</tr>
<tr>
<td>7.7</td>
<td>Abridged RealtimeThread class with cost monitoring.</td>
</tr>
<tr>
<td>7.8</td>
<td>Result for the experiments with cost monitoring mechanisms.</td>
</tr>
<tr>
<td>7.9</td>
<td>Abridged ReleaseParameters class with blocking time monitoring.</td>
</tr>
<tr>
<td>7.10</td>
<td>Abridged RealtimeThread class with blocking time monitoring.</td>
</tr>
<tr>
<td>7.11</td>
<td>Overhead for different settings of the PI experiment in the RTSJ.</td>
</tr>
<tr>
<td>7.12</td>
<td>Overhead for different settings of the PCE experiment in the RTSJ.</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Properties verified against the PRTT subsystem. . . . . . . . . . . . . . . . 126
5.2 Properties verified against the ASRTT subsystem. . . . . . . . . . . . . . . 127
5.3 Properties verified against changes to the release parameter. . . . . . . . . 128

6.1 Properties verified against the model with the PRTT subsystem. . . . . 157
6.2 Properties verified against the model with the ASRTT subsystem. . . . . 158
6.3 Properties verified against the model with the PRTT subsystem. . . . . 170
6.4 Properties verified against the model with the ASRTT subsystem. . . . . 171
List of Algorithms

4.1 Requirements for calculating the blocking time. . . . . . . . . . . . . . . . 94
4.2 Algorithm for calculating the blocking time (part 1). . . . . . . . . . . . . 95
4.3 Algorithm for calculating the blocking time (part 2). . . . . . . . . . . . . 97
Acknowledgements

I would like to express my gratitude to my supervisor, Professor Andy Wellings, whose guidance and support was paramount to this work. Also, I would like to thank my assessor, Professor Alan Burns, for his support during the development of this work.

I would also like to acknowledge CAPES, one of the funding agencies of the Brazilian government, which provided the financial support needed for this research work.

Many thanks to all the members of the Real-Time System Group, which provided a great working environment. I also would like to thank Yang Chang for helpful discussions about the implementation of jRate on top of MaRTe OS.

My deepest gratitude to all the great friends I met during my stay in York. In special, thanks to Ana Cavalcanti, Jim Woodcock, Marcel Oliveira, Lauana Oliveira, Fernando Luís Dotti, Leila Ribeiro, Leonardo Freitas, Angela Freitas, Richard Ribeiro and Rosa Ribeiro.

I would also like to thank, with all my love, my parents, Osmar Souza dos Santos and Luci Teresa Marchi dos Santos, and my sisters, Luciana Marchi dos Santos and Lucimar Marchi dos Santos. Their support was very important to me.

Finally, I would like to express my sincere gratitude to my wife, Simone Regina Ceolin. She was with me all the time during this journey. Simone, I love you.
Declaration

I declare that the research work described in this thesis is original work unless otherwise indicated in the text. The research was conducted by me, Osmar Marchi dos Santos, between 2004 and 2008 at the University of York, with the supervision of Professor Andy Wellings. Some parts of this thesis have been published in scientific publications.

In Chapter 4, a preliminary version of the proposed mechanism for blocking time monitoring in the POSIX/C standard has been published as the conference paper “Run Time Detection of Blocking Time Violations in Real-Time Systems” [65], which appeared in the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.

Chapter 5 is based on the material previously published as a conference paper entitled “Formal Analysis of Aperiodic and Sporadic Real-Time Threads in the RTSJ” [62], which appeared in the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems.

An initial version of the formal analysis of the execution time monitoring mechanism shown in Chapter 6 was published as the conference paper “Cost Monitoring and Enforcement in the Real-Time Specification for Java - A Formal Evaluation” [61], which appeared in the 26th IEEE Real-Time Systems Symposium. This paper won the joint best paper award of the conference. A follow-on paper was published as a journal paper entitled “Cost Enforcement in the Real-Time Specification for Java” [63], which appeared in the Real-Time Systems Journal. A preliminary version of the blocking time monitoring mechanism for the RTSJ, also shown in Chapter 6, was published as the conference paper “Blocking Time Monitoring in the Real-Time Specification for Java” [64], which appeared in the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems.
Real-time systems comprise a class of computing systems whose functionality is interrelated to the timing constraints of the system. Therefore, in real-time systems, besides guaranteeing the correctness of functional requirements, it is necessary to guarantee the correctness of the timing requirements [24]. One common misconception regarding real-time systems is that these systems have to be extremely fast [69]. As with any other computing system, real-time system developers do want their systems to be very efficient, however, the most important attribute in the design of these systems is predictability. A predictable real-time system is one whose timing requirements are shown to be met, subject to any underlying assumptions made by the system (e.g., concerning workloads) [70].

In order to represent a system, whose temporal behaviour can be analysed, the real-time literature generally models real-time systems as a set of concurrent threads\(^1\) that have certain timing parameters associated to them. Amongst others [28], three parameters are

\(^1\)Other terms frequently used in the literature include processes and tasks.
commonly specified: (a) release time, the instant of time at which a thread becomes available for execution; (b) execution time, the time the thread needs to be executed by the system to finish its current computation; (c) deadline, the timing constraint that specifies when the thread should finish its current execution. Threads can be classified according to the configuration of their release times. A thread is said to be periodic if the time between two consecutive releases is defined by a constant value, called period. An aperiodic thread is one that is not periodic. A sporadic thread is an aperiodic thread that has a bound on the load it may impose on the system [66]. This is typically known as the Minimum Inter-arrival Time (MIT) and specifies that no two consecutive releases of the thread occur between an MIT.

Based on the value of the computation done by a thread, the deadline of the thread can be classified into two different categories [45]. A hard deadline is one where the failure of the thread to meet it can lead to a catastrophe. An example of a hard deadline is a late command to apply the brakes in a car, after the driver pressed the brake pedals, which can lead to a collision. Differently, a soft deadline is one where a deadline miss does not have a disastrous consequence, but the late completion of the thread can lead to a poorer performance of the system. One example of a soft deadline is a thread that generates a route in a Global Positioning System (GPS) Navigator. In principle, not generating the route by the deadline will not lead to a catastrophe, but the later the thread finishes, the poorer is the performance, and the more annoyed the user becomes.

In fact, real-time systems can be broadly classified according to the type of deadlines for their composing threads [45, 24]. A system that includes hard deadlines is said to be a hard real-time system. Although hard real-time systems are characterised by having hard deadlines, they may still contain threads with soft deadlines. However, it is imperative that the hard deadlines of the system are shown to be met. Examples of hard real-time systems include safety-critical systems, like flight, train, and missile control systems.
A soft real-time system is one composed of only soft deadlines. In these systems it is not required to show rigorously that deadlines are met. Instead, it is tolerated that, occasionally [24]: (a) deadlines are missed or aborted, maybe with an upper limit on the number of deadlines missed within a given interval; (b) threads complete late, after the deadline, but again with an upper limit specification. Examples of soft real-time systems include telephone switching network control systems, multimedia systems, and stock price quotation systems. The mechanisms discussed in this thesis can be used in both hard and soft real-time systems, but it is vital that the system’s timing constraints are analysed and shown to be met.

In order to guarantee that the deadlines of the system are going to be met during the run time execution of the system, real-time systems often use feasibility analysis\(^2\) to test if the system meets its deadlines. Amongst other parameters, the feasibility analysis can be characterized by the coverage of the feasibility test and the constraints of the computational model (e.g., type of computing architecture and scheduling approach). On one hand, a feasibility test can be sufficient (if the system passes the test, it will meet the deadlines) but not necessary (if the system fails the test, it may or may not meet its deadlines). Sufficient but not necessary tests are not ideal, but they are generally less complex [66]. An example of a sufficient but not necessary feasibility test is the utilization-based test [44]. On the other hand, a sufficient and necessary feasibility test always shows if a system will meet or not meet its deadlines. Unfortunately, sufficient and necessary feasibility tests can be very complex for certain computational models [66]. An example of a sufficient and necessary feasibility test is the response time analysis test [36, 13]. Moreover, with respect to the computational model, the feasibility test is very tied to the underlying scheduling approach that is used to arrange the resources, especially threads, in the system. The main scheduling approaches found in real-time literature include Fixed-

\(^2\) Also called schedulability analysis in the literature.
Priority Scheduling (FPS) and Earliest Deadline First (EDF) scheduling. Both FPS and EDF scheduling approaches have their advantages and disadvantages [45, 24, 66, 27]. This thesis focuses on the use of the FPS scheduling approach. The main reason for this choice is the widespread use, and therefore more matured technology, of FPS in both commercial and academic run time systems, when compared to EDF, which is mainly used in academia [27].

One crucial phase in the development of a real-time system, required for the feasibility analysis, is the derivation of upper bounds for the execution times of the threads in the system. This execution time upper bound is commonly known as the Worst-Case Execution Time (WCET). Basically, there are two methods that can be used for deriving the WCETs [78]: (a) static, where the code is analysed together with some (abstract) model of the hardware architecture; (b) measurement-based, where the thread or parts of it are executed in a given hardware or even simulated. Unfortunately, several factors found in modern computer architectures (e.g., prefetch queue and cache memory) can have an effect on the execution times that make it difficult to predict the WCET [28]. Therefore, deriving a precise estimation of the WCET is very difficult (if not impossible) to achieve [28]. Consequently, it is possible that, during the run time execution of the system, the thread can execute further to the computation time initially estimated – an execution overrun occurs. The overrun of a thread may lead to a deadline being missed by the thread or even others threads in the system. Both the execution time overrun and the deadline miss can be considered timing errors. Moreover, the overrunning thread should not force other threads in the system to miss their deadlines. Indeed, a temporal isolation mechanism for overrunning threads (erroneous threads, from a timing point of view) should be provided by the real-time programming language and supported by the run time system [23]. In this context, this research aims to provide a better understanding of timing error detection and temporal isolation mechanisms in both real-time programming
1.1 Fault Tolerance

A dependable computing system is one whose delivered service can be justifiably trusted [40, 42, 41, 14, 15]. The delivered service in this case is the behaviour of the system that is perceived by its user (another interacting system, either physical or human). A functional specification is used together with a non-functional specification to define the intended behaviour of the system [15]\(^3\), i.e., the system specification. Correct service is provided when the system complies to, or implements, the system specification. However, it is possible that system failures can occur to the system. A system failure occurs when the delivered service deviates from its correct service, leading to the delivery of an incorrect service. In other words, the dependability of a system can be defined as the ability of a system to avoid failures that are more frequent or severe than is acceptable by its user [14, 15].

The exposition to the concepts of dependability is usually described in three parts [41, 14, 15]: the threats to, the attributes of, and the means to achieve dependability. Reliability is one of the attributes of dependability and consists of the capability of a system to deliver a correct service. In particular, this research work is mainly concerned with fault tolerance for timing errors. Fault tolerance is considered one of the means to achieve dependability in a system and, therefore, improve the system’s reliability (one of the attributes of dependability). In fact, for providing a presentation of the steps usually followed in the design of fault tolerant systems, it is necessary to introduce the threats to dependability, i.e., the causes in the system that can lead to a system failure. According to standard terminology in the dependability area [42, 14, 15], a fault is a design (algorithmic) or component (physical) problem in the system that, when active, can cause an

\(^3\)In [15] the term meta-functional specification is used, instead of non-functional specification, to define the system’s specification.
error. An error is considered an invalid internal state of the system that can propagate, generating other errors, leading to a system failure. Indeed, the events that can lead to a system failure can be described with the fundamental chain [14, 15] presented below.

Concerning fault tolerance, the design of a fault tolerant system is usually composed of four different steps [42]. The starting point is (i) error detection, which detects the erroneous internal state of the system, as the fault itself, represented in the chain, cannot be detected directly. In terms of a real-time system, an example of error detection would be the detection of a deadline missed by a thread in the system. Next, (ii) damage confinement and assessment is used to assess the extent to which the system has been damaged and possibly confine the erroneous state, since the erroneous information could propagate through the system. From a real-time perspective, this phase could be exemplified as a temporal isolation support for the overrunning threads. Then, the third step is (iii) error recovery. Once the error has been detected and assessed, error recovery techniques are used to transform the system from an erroneous state to an error-free state. A technique often used for error recovery in real-time systems is to change the mode of the system by executing an alternative set of threads, with a possible correct but degraded service. In the last step, (iv) fault treatment and continued service takes place. The objective in this phase is to ensure that the fault, which caused the (detected) error, does not recur. In this case the fault is repaired or the system is reconfigured to avoid the fault. For real-time systems and non real-time system, this may lead to the system being stopped and corrective measures applied to solve the fault itself. Ideally, the faster the error is detected in the system, the better, as the next steps can be executed more effectively.
1.2 Timing Errors

In other words, once an error has been detected, the goal of the system is to limit the propagation of the error (damage confinement and assessment) so that error recovery can occur as soon as possible and with more accurate information.

1.2 Timing Errors

According to the fundamental chain presented earlier, a fault is an algorithmic or physical problem in the system that, when active, can cause an error. Unfortunately, due to its nature, it is not possible to detect the fault in the system, but only its consequence (the error). The dependability literature provides a vast classification of different types of faults that can happen in a system [14, 15]. Also, different failures modes that the system can exhibit upon a specific failure are classified [14, 15]. Indeed, errors are normally classified according to the type of failure mode they can lead the system to [14, 15]. The main concern of this research work is with timing errors that, if left unhandled in the system, can lead the system to a timing failure.

The principal timing error that can occur in a real-time system is a deadline miss. In fact, other timing errors are responsible for the occurrence of a deadline miss in the system. In order to describe the possible timing errors that can occur in a real-time system, it is important to inspect the timing aspects of the system that are used in the feasibility analysis test. Consider a real-time system executing in a uniprocessor architecture, using the FPS approach, and the possibility of sharing resources between threads. In this case, an ideal feasibility test, like response time analysis, takes the following major timing factors of the system into account for the analysis [12, 45, 24]:

(i) assumptions of the WCETs for the threads in the system. As described previously, the WCET describes the maximum computation time that a thread can execute in a given release. They are calculated with either static analysis methods or
measurement-based techniques. However, the determination of accurate execution time values is notoriously difficult to achieve. Therefore, imprecise values can be used as input to the feasibility analysis;

(ii) interference times of higher priority threads executing in preference to lower priority threads. Due to the priorities used in the scheduling process, lower priority threads have to wait for higher priority threads to finish their execution in order to start executing. The values used for calculating the time that lower priority threads have to wait are based on the WCETs of the higher priority threads [24];

(iii) assumptions of the potential blocking times that higher priority threads suffer when lower priority threads are accessing shared resources. In a real-time system with FPS, where threads can interact using lock-based shared resources, a problem called priority inversion [39] can occur. Priority inversion happens when a lower priority thread accesses a resource that is shared with a higher priority thread. In this case, the higher priority thread becomes blocked for an unbounded length of time, since it has to wait for the lower priority thread to leave the shared resource to resume execution. In order to deal with the priority inversion problem in the FPS approach, priority inheritance protocols have been proposed in the literature [67, 57]. These protocols allow for the derivation of the maximum time a higher priority thread has to wait, whilst a lower priority thread is executing the shared resource, in order to resume execution and enter the shared resource. The potential blocking times are calculated using the WCETs related to the shared resource segments that higher priority threads share with lower priority ones. As discussed previously, WCETs can be imprecise and so can the potential blocking time values being used;

(iv) guarantees about the release time of both periodic and sporadic threads. Providing this guarantee for periodic threads is quite straightforward, as the threads are
released in constant periods of time. For sporadic threads, worst-case assumptions regarding the MIT should be provided. However, it is always possible that, during the run time execution of the system, these assumptions may fail.

In this research, the underlying fault assumption is that the WCET for a thread used in the feasibility analysis can be imprecise and, therefore, undermined during the run time execution of the system. Indeed, it is also assumed that anything, from programming errors to the miscalculation of the WCET, could have lead the system to undermine the WCET of certain threads. This way, it is possible to identify four possible fault-error-failure chains, based on the four major timing factors used in the feasibility analysis test, which could be generated in a real-time system:

1. **fault** (in thread’s $T_i$ WCET assumptions executing in a non-shared resource) $\rightarrow$ error (overrun of thread’s $T_i$ WCET in a non-shared resource) $\rightarrow$ error propagation (deadline miss of thread $T_i$) $\rightarrow$ failure (to deliver service in a timely manner);

2. **fault** (in thread’s $T_i$ WCET assumptions executing in a non-shared resource) $\rightarrow$ error (overrun of thread’s $T_i$ WCET in a non-shared resource) $\rightarrow$ error propagation (greater interference on lower priority threads) $\rightarrow$ error propagation (deadline miss of lower priority threads) $\rightarrow$ failure (to deliver service in a timely manner);

3. **fault** (in thread’s $T_i$ WCET assumptions executing inside a shared resource) $\rightarrow$ error (overrun of thread’s $T_i$ WCET inside a shared resource) $\rightarrow$ error propagation (greater blocking time for higher priority threads sharing the resource) $\rightarrow$ error propagation (deadline miss of higher priority threads) $\rightarrow$ failure (to deliver service in a timely manner);

4. **fault** (in thread’s $T_i$ MIT assumptions) $\rightarrow$ error (overrun of thread’s $T_i$ MIT assumptions) $\rightarrow$ error propagation (greater interference on lower priority threads) $\rightarrow$ error
propagation (deadline miss of lower priority threads) → failure (to deliver service in a timely manner).

Moreover, the feasibility analysis is assumed to be correct in this research work. With respective to the fault tolerance for timing errors, real-time programming languages and run time support systems have traditionally focused on the deadline monitoring mechanism for the error detection. In this error detection mechanism, the system keeps track of the times where deadlines should be met. If a deadline for a given thread is missed (the timing error), the system detects it and executes specific application code to handle the error. Unfortunately, as depicted in the chains described earlier, the timing error (deadline miss) is the result of previous timing errors that propagated in the system. Therefore, other mechanisms for the earlier detection of timing errors could be provided, possibly improving fault tolerance for timing errors in real-time systems. The provision and evaluation of timing error detection mechanisms is the main focus of this thesis, along with the use of damage confinement mechanisms that, when applicable, can be used to temporally isolate errant threads.

1.2.1 Run Time Error Detection

Deadline monitoring has been the traditional approach used for the detection of timing errors in real-time systems. Nevertheless, another important approach for timing error detection that can be found in the literature is execution time monitoring. Indeed, early work by Burns and Wellings [23] discussed the need to provide temporal detection mechanisms for overrunning threads. Following their work, several others approaches in the literature were proposed for the run time detection of execution time overruns [71, 34, 58, 51, 55, 50, 21, 75]. From a run time error detection point of view, all these works share the same principle. The provision of an execution time budget (based on the WCET of the thread) for each release of the thread. Like in the deadline monitoring
approach, once the execution time budget is violated, i.e., the thread is overrunning its execution time; a handler capable of dealing with the overrun is generated. Recognising the need to equip real-time operating systems with the ability to monitor the execution time of threads, possibly taking actions upon overruns, the POSIX/C standard was extended with the notion of execution-time clocks and timers [53, 1]. This provided the basis for both monitoring the execution time (with the use of clocks) and detecting the overruns (with the use of timers).

The computational model assumed in this research work comprises a uniprocessor architecture, the FPS approach, and the possibility of threads sharing resources. With respect to this model, mechanisms based on execution time monitoring could also be used to monitor and detect overruns related to the blocking times of threads. For instance, using the execution-time clocks and timers of the POSIX/C standard, it is possible to build such mechanism by: (a) starting an execution-time timer (using the WCET of the particular shared resource segment as the execution time) at the entrance point of every shared resource; (b) stopping the execution-time timer at the exit point of the shared resource. Unfortunately, this approach has three drawbacks. Firstly, the granularity of the time provided in this approach is down to a certain piece of code, where additional overhead is placed every time a thread enters and leaves the shared resource. In this case, it would be ideal to have a mechanism implemented at the kernel level with a finer, therefore more precise, granularity of time. Secondly, upon the detection of the overrun, the system is only capable of determining the thread that is causing the blocking overrun. Although actions can be taken upon the overrunning thread, not much can be done for the (higher priority) threads that are being affected by the overrun (waiting to enter in the shared resource). Thirdly, in large real-time systems, where it is possible to have a large number of nested shared resources, it becomes a cumbersome and difficult task to set correctly the timers in the system. One of the contributions of this thesis is the extension of the
POSIX/C standard with clocks and timers that provides the basis for monitoring and detecting the overrun of blocking times. The proposed mechanism provides an alternative that resolves the discussed problems introduced when using execution-time clocks and timers to monitor the blocking times of threads.

1.2.2 Run Time Damage Confinement

Another aspect related to the detection of a timing error in a thread is the question of what actions to take. The most basic action, generally employed in deadline monitoring, is to signal the system that an error occurred, so the application can take further actions. With the introduction of execution time monitoring, a discussion of possible approaches to take upon execution overruns emerged [58]. In this sense, the common approach taken in the literature is to temporally isolate, enforce behaviour upon, the thread. That is, in terms of fault tolerance, confine the damage. Indeed, the earlier work by Burns and Wellings [23] already discussed the need to either stop the execution of a thread or reduce its priority, so that it could not impact on the execution of other threads (the system could still meet its deadlines).

Most mainstream real-time programming languages do not provide damage confinement mechanisms, but instead focus on delivering error detection mechanisms that can be used by the developer to implement an enforcement model, if required. Unfortunately, from the developers point of view, such task can become quite cumbersome and error prone. For instance, if using execution-time clocks and timers provided by the POSIX/C standard, the developer has to arrange correctly the timers for the threads and also deal with signal handling. Recently, an extension of the Java platform [11], called the Real-Time Specification for Java (RTSJ) [20, 18], was developed for the construction of real-time systems. Besides inheriting the good software engineering aspects of the Java programming language, like object orientation, the RTSJ tries to integrate timing error detection
facilities, like deadline and execution time monitoring, with the programming language and underlying runtime system.

More specifically, the RTSJ provides an execution time monitoring mechanism that enforces behaviour upon overrunning threads. From the developer’s point of view, this mechanism facilitates the deployment of damage confinement upon errant threads. Unfortunately, the execution time monitoring mechanism of the RTSJ is considered an optional facility for RTSJ implementations. This meant that, apart from the research work described in this thesis, no other work in the literature has undertaken a detailed study of the model of the mechanism. One research contribution of this thesis is the formal evaluation of the execution time monitoring model proposed in the RTSJ. It is shown that the model is incorrect, because it fails to confine errant threads in certain execution scenarios, and a new model is proposed and evaluated.

1.3 Thesis Hypothesis and Contributions

This research work is concerned with mechanisms that can be used for the detection of timing errors in real-time systems. At its core is the idea that, with respect to fault tolerance, the typical deadline monitoring mechanism does not provide enough accuracy for error detection in real-time systems. The hypothesis of the thesis can be summarized as follows:

Timing errors can develop when the assumptions used in the feasibility analysis of the system are undermined during the system’s runtime execution. In this context, different runtime monitoring mechanisms for error detection can be used for the earlier detection of timing errors, before the occurrence of deadline misses. Where applicable, this approach can provide mechanisms to confine the temporal damage to the errant threads of the system as well as
facilitate error recovery strategies.

The thesis contends that the implementation of different error detection mechanisms for timing errors can improve the reliability of real-time systems. In principle, with the detection of more timing errors, the system can tolerate (or at least become aware of) more faults. Moreover, whenever possible, this approach could introduce some form of confinement for the application, so that the temporal damage cannot interfere with the execution of other threads in the system. In order to demonstrate the hypothesis, the following contributions are included in the thesis:

(C1) development of a technique to monitor and detect violations for the blocking time of higher priority threads in the POSIX/C standard. This provides an additional error detection model to the use of execution-time clocks and timers;

(C2) development of a formal model for the evaluation of the execution time monitoring mechanism of the RTSJ. This resulted in the discovery of an error and the proposal of a new revised mechanism;

(C3) development of a mechanism to monitor and detect violations for the blocking time of threads in the RTSJ. The higher level constructs of the RTSJ provides the ability to detect a blocking time violation and quickly respond to it, using a mechanism that can improve the recovery for such type of timing error;

(C4) a better understanding of the relationship between timing errors and the possibility of using techniques for damage confinement in real-time systems.

As well as these contributions, used to demonstrate the thesis hypothesis, another contribution of the thesis includes:

(C5) the provision of an open-source run time support system for the RTSJ that executes on a bare machine under the x86 architecture.
1.4 Thesis Outline

The main topic addressed in this thesis is fault tolerant real-time systems. Chapter 2 provides a literature review of aspects related to this topic. More specifically, it starts by describing the response time analysis, which can be used for the timing analysis of real-time systems. Then, it presents in more detail the techniques involved in fault tolerance. It concludes with the presentation of different work aiming for the runtime detection of timing errors.

The computational model along with the description of the POSIX/C standard and the RTSJ are presented in Chapter 3. This chapter also summarizes different error detection models, for the considered timing errors, in both POSIX/C and RTSJ.

The model for detecting timing errors in terms of the POSIX/C standard is discussed in Chapter 4. Moreover, in this chapter, a mechanism for the monitoring and detection of blocking time violations is proposed.

The RTSJ computational model is a complex one, with many interactions between its features. In order to formally explore its execution time monitoring mechanism, some formal descriptions are required for the behaviour of real-time threads defined in the RTSJ. Chapter 5 focuses on the presentation and analysis of such a formal description.

The model for detecting timing errors in terms of the RTSJ is presented in Chapter 6. In this chapter, a formal analysis of the current execution time monitoring mechanism of the RTSJ is developed, and a problem is found and corrected. Also a mechanism for monitoring and detecting blocking time violations is described.

Chapter 7 presents a prototype implementation of the mechanisms proposed in this research work. An open-source run time environment, running in a bare machine, for the implementation of the mechanisms is provided. Using this, an evaluation of the overhead for the implementation of the proposed mechanisms is also presented.
The thesis concludes with Chapter 8, where a summary of the contributions of the research work is presented. Also, overall conclusions of the work, as well as, possible future work are discussed.
This chapter provides a literature review of the main topics, real-time systems and fault tolerance, addressed in this thesis. According to the assumptions of this thesis, timing errors occur due to the deviation of the system’s expected temporal execution during its run time execution. Section 2.1 describes a technique for feasibility analysis found in the literature. It focuses on the presentation of an ideal feasibility analysis, response time analysis, for a uniprocessor architecture, using the FPS (Fixed-Priority Scheduling) approach and the possibility of threads sharing resources using lock-based techniques. Since sharing resources is considered, different protocols targeting to bound the blocking time for higher priority threads in the system are shown. Also, the support for aperiodic and sporadic threads via servers already provided in the literature of real-time systems is discussed. In Section 2.2, a review of the fault tolerant computing area is presented. Then, in Section 2.3 related work found in the literature of real-time systems, tackling the detection and possible temporal isolation (damage confinement) for timing errors, are
discussed. Moreover, this section also presents a brief overview of mode changes, which provides some form of damage assessment and error recovery for overload conditions in real-time systems.

2.1 Response Time Analysis

Real-time systems are characterized not only by the correctness of its results, but also on the timing requirements of these results [24]. Therefore, one of the most important aspects of real-time is calculating if the system is feasible (schedulable). A feasible real-time system is one where all its threads meet their deadlines. In other words, the feasibility analysis (test) of the system is the validation of the temporal requirements of the system.

Amongst other parameters, the feasibility analysis can be characterized by the coverage of the feasibility test and the constraints of the computational model (e.g., type of computing architecture and scheduling approach). On one hand, a feasibility test can be sufficient (if the system passes the test, it will meet all deadlines) but not necessary (if the system fails the test, it may or may not meet its deadlines). Sufficient but not necessary tests are not ideal, but they are generally less complex [66]. An example of a sufficient but not necessary feasibility test is the utilization-based test [44]. On the other hand, a sufficient and necessary feasibility test always shows if a system will meet or not its deadlines. Unfortunately, sufficient and necessary feasibility tests can be very complex for certain computational models [66]. An example of a sufficient and necessary feasibility test is the response time analysis test [36, 13]. In this research, response time analysis is considered.

More specifically, with respect to the computational model, the feasibility test is very tied to the underlying scheduling approach that is used to arrange the resources, in particular threads, in the system. The main scheduling approaches found in real-time literature
include FPS and Earliest Deadline First (EDF) Scheduling. Both FPS and EDF scheduling approaches have their advantages and disadvantages [24, 47, 66, 27]. However, this thesis focuses on the use of the FPS approach. The main reason for this choice is the widespread use, and therefore more matured technology, of FPS in both commercial and academic run time systems, when compared to EDF, which is mainly used in academia [16].

Now, the response time analysis equation for a uniprocessor architecture with FPS, considering only independent periodic threads, is presented. As the threads in the system are scheduled using priorities, the response time (denoted by $r$) for the thread with highest priority in the system will be equal to its computation time (denoted by $C$): $r = C$. Differently, all the others threads in the system will suffer interference in their executions coming from higher priority threads. Therefore, the equation for the response time of a particular thread $i$ will be [24]:

$$r_i = C_i + I_i$$  \hspace{1cm} (2.1)

That is, the response time for thread $i$ equals its computation time added to the maximum interference (denoted by $I$) it can suffer from higher priority threads in the interval $[t, t+r_i]$ [24]. The maximum interference that a thread $i$ can suffer in the particular interval $[0, r_i)$, is calculated as follows [24]:

$$I_i = \sum_{\forall j \in hp(j)} \left\lfloor \frac{r_i}{T_j} \right\rfloor C_j$$  \hspace{1cm} (2.2)

In this equation, the ceiling function $\left\lfloor \frac{r_i}{T_j} \right\rfloor$ is used to determine the maximum number of releases (using the thread’s release period, denoted by $T$) that a higher priority thread $j$ can have in the interval $r$. Multiplying the maximum number of releases with the computation time of the higher priority thread gives the interference added by the particular thread. This way, the equation calculates the maximum interference for the set of higher priority
threads \( hp(j) \) in the system. In theory, combining Equations 2.1 and 2.2, would enable the response time for thread \( i \). Unfortunately, this is not possible, because the term \( r_i \) would appear in both sides of the equation. The solution is to use an iterative technique to solve the equation \([24]\):

\[
r_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{r_j^n}{T_j} \right\rceil C_j
\]

In order to calculate the response time, the iteration starts with \( r_i^0 \), and terminates when \( r_i^{n+1} = r_i^n \) (the response time for thread \( i \)). In the next section, the extension of the response time equation to consider threads that can share resources is considered, along with protocols that are used to bind the blocking time. Moreover, in Section 2.1.2 the use of servers to allow for the execution of aperiodic and sporadic threads is presented. It is also shown how the response time equation is extended to consider the use of servers.

### 2.1.1 Priority Inheritance Protocols

In a real-time system with FPS, where threads can interact using shared resources using lock-based techniques, a problem called priority inversion \([39]\) can occur. The priority inversion problem is defined as follows.

**Definition 2.1.** Priority inversion happens when a lower priority thread accesses a resource that is shared with a higher priority thread. In this case, the higher priority thread becomes blocked for a length of time, since it has to wait for the lower priority thread to leave the shared resource to resume execution.

In order to deal with the priority inversion problem in the FPS approach, priority inheritance protocols have been proposed in the literature \([67, 57]\). These protocols allow for the derivation of the maximum time a higher priority thread has to wait, whilst a lower priority thread is executing the shared resource, in order to resume execution and enter
the shared resource. The potential blocking times are calculated using the WCETs related to the shared resource segments that the higher priority threads share with lower priority ones. Two priority inheritance protocols, which are assumed in the computational model of this research work, are described in this section.

Firstly, the Basic Priority Inheritance (PI) protocol [67] is described. In general terms, PI is an appropriate synchronization protocol for large real-time systems, where it is often difficult to determine the pattern of indirect synchronization between threads. The idea of the protocol is that, a lower priority thread inside a shared resource inherits the highest priority of all threads attempting to enter the resource. The main advantages of this protocol are: (i) the wide support provided by Real-Time Operating Systems (RTOSs); (ii) priority changes only occur when needed (there is no cost in the common case when the shared resource is not in use). Its main disadvantages are that: (i) a thread may be blocked separately for each shared resource that it needs (and, therefore, deadlock can occur); (ii) “chained blocking” may occur when threads are waiting for shared resources that are held by other threads, which are waiting for shared resource held by different threads; (iii) implementation may be expensive because of nested (recursive) inheritance and the fact that a thread’s priority is changed by an action external to the thread. When using this PI protocol, the maximum blocking time that a thread \( i \) can suffer is defined as \( \min(n, m) \) shared resources, where \( n \) is the number of lower priority threads that could block thread \( i \) and \( m \) is the number of distinct shared resources that could be used to block \( i \) [67, 57, 26].

Differently from the PI protocol, the Priority Ceiling (PC)\(^1\) protocol allocates each shared resource a ceiling priority. This ceiling is set to the maximum priority that a thread trying to enter the shared resource can acquire. When a thread enters the resource, its

\(^1\)The Priority Ceiling protocol is called Priority Protection (PP) in the POSIX/C standard and Priority Ceiling Emulation (PCE) in the Real-Time Specification for Java (RTSJ).
priority is immediately raised to (if it is not already at) the ceiling. The protocol’s main advantages are: (i) if no thread can block whilst executing a shared resource, then a queue is not needed for that resource; (ii) “nested monitor” deadlock is prevented; (iii) a thread can be blocked at most once during each release by some lower priority thread executing the shared resource. The disadvantages include: (i) computation of ceilings needs careful analysis, especially if thread priorities and ceiling values can change dynamically; (ii) it requires a check and priority change at each call (used to prevent unbounded priority inversion). When using this PC protocol, the maximum blocking time that a thread $i$ can suffer is equal to the duration of one shared resource access [67, 57, 26].

In order to consider the maximum blocking time that a higher priority thread may suffer during its execution, the Equation 2.3 is modified as follows:

$$r_i^{n+1} = B_i + C_i + \sum_{\forall j \in hp(j)} \left\lceil \frac{r^n_j}{T_j} \right\rceil C_j$$

(2.4)

Note that, the only modification to the equation is the addition of the parameter $B_i$. This parameter represents the maximum blocking time that a thread $i$ can suffer from lower priority threads, and is dependent on the priority inheritance protocol being used. As presented previously, the calculation of these values for both PI and PC protocols need to follow an inspection of the application.

### 2.1.2 Support for Aperiodic and Sporadic Threads using Servers

In order to provide support for the execution of aperiodic and sporadic threads, so that good averages for their response time and, specially, no impact on the behaviour of other threads occurs in the system, the notion of servers was introduced for FPS [43]. A server is composed of a budget ($C_s$) and a replenishment period ($T_s$). When an aperiodic or sporadic thread wants to execute, it requests the server to execute using the available
2.1 Response Time Analysis

budget of the server. Therefore, whilst the thread executes, the server’s budget starts decreasing. If the server’s budget exhausts, the execution of the thread generally stops and no requests from aperiodic or sporadic threads are serviced, until the budget is replenished. Two servers are most relevant to this research: (a) Deferrable Server [43, 73], which can be implemented in the RTSJ [25]; (b) Sporadic Server [68], which is implemented in the POSIX/C standard. The difference between these two servers is related to the replenishment technique employed.

In the Deferrable Server (DS) technique, the replenishment occurs at every replenishment period $T_s$ of the server. One effect that can happen in the DS is the double hit [19], where the server (in certain scenarios) can impact twice on the execution of lower priority threads, i.e., just before the replenishment period occurs, the server executes until exhaustion, but because of the replenishment, it can continue executing for another budget. The RTSJ provides a mechanism, called $\text{ProcessingGroupParameters}$, which provides a server technique similar to the DS. A $\text{ProcessingGroupParameter}$ has a budget and a replenishment period – replenishing in the same way as the DS. However, the $\text{ProcessingGroupParameter}$ is used to provide a shared budget for several different threads (with periodic, sporadic and aperiodic timing requirements) that can have distinct priorities – they do not execute with the budget’s priority, but with their own thread priority. This results in a very difficult mechanism to be analysed, as described in [25]. Nevertheless, if a $\text{ProcessingGroupParameter}$ is used to provide budget for several threads executing with the same priority, its timing analysis can be calculated just like for the DS. The timing impact of a DS in the system can be calculated as [33, 19]:

$$
\left(1 + \left\lfloor \frac{r^n - C_s}{T_s} \right\rfloor \right) C_s
$$

Differently from the DS, the Sporadic Server (SS) only replenishes its budget when an
aperiodic or sporadic thread executes. This means that, once a thread stops its execution, the exact amount of capacity from the budget, which was used by the thread, will be replenished when the time $T_s$ passes in the system. The SS does not have the effect of double hit and its interference can be easily calculated as [19]:

$$\left\lceil \frac{r^n_i}{T_s} C_s \right\rceil$$ (2.6)

In order to also consider the interference of servers, response time analysis (Equation 2.4) has to be modified as follows:

$$r_i^{n+1} = B_i + C_i + \sum_{\forall j \in hp(j)} \left\lceil \frac{r^n_i}{T_j} \right\rceil C_j + S$$ (2.7)

Where $S$ is the interference of the server in use. If using the DS, the parameter $S$ has to be substituted using Equation 2.5. Otherwise, if using the SS, the parameter equals to Equation 2.6.

### 2.2 Fault Tolerance

This section reviews well-defined concepts in the fault tolerance area relevant to this research work. In a wider context, fault tolerance is one of the means used to achieve dependability in a computing system. Therefore, in the next section the concept of dependability is introduced. The threats of (notion of faults, errors and failures), means to (including fault tolerance) and attributes of (with a particular focus on reliability) dependability are overviewed. Section 2.2.2 presents in more details the steps usually followed to achieve fault tolerance in a computing system.
2.2.1 Dependability

The concept of dependability relates to [42] a property of a computing system which allows reliance to be justifiably placed on the service it delivers. Note that, this concept uses two key definitions. Firstly, the property of a computing system depends on the application intended for the system, i.e., different emphasis can be placed on different aspects of dependability [40]. An example of a dependability aspect for a system is [40] reliability, i.e., the property of a system to continue delivering a correct service. Secondly, the service a system delivers is the behaviour perceived by its user (another system either physical or human) via a system interface [14]. A system specification defines the correct behaviour of the system, i.e., what it is intended to do. Therefore, a system delivers the correct service when it implements this correct behaviour.

A system failure occurs when the system delivers an incorrect service, i.e., it does not implement the correct behaviour specified in the system specification. An error is considered to be an erroneous state of the system that can lead to a failure. The error itself is caused by a fault. A fault is said to be active when it causes an error, and dormant otherwise. Different classifications of faults can be found in the literature [40, 42, 14, 15]. Nevertheless, this research does not focus on a particular type of fault, but the error it causes. In particular, the focus of this thesis is on timing errors. That is, errors in the system that can lead the system to a timing failure [14, 15]. The mechanisms described so far compose the threats to dependability, and they can be described with the fundamental chain of dependability threats [14, 15] presented below.

```
... ➔ fault ➔ activation ➔ error ➔ propagation ➔ failure ➔ causation ➔ fault ➔ ...
```

From this thesis point of view, one example of a chain that can occur in the system is
a software bug (the fault) that affects the loop of a thread. This particular fault can lead the thread to execute longer than expected (the timing error). This can then potentially lead the system to miss the deadline of the thread (the timing failure), or other threads in the system. This failure can be perceived by the user of the system (another system in this case) as an error. Indeed, this follows the basics of the fundamental chain, where the error can propagate in the system.

In order to deal with these threats, systems usually make use of certain techniques to attain dependability. These are the means by which dependability is attained. More specifically, four steps are presented in the literature [40, 42, 14, 15]:

- fault prevention: this include techniques that can be used for preventing faults from being introduced into the operational system. For example, techniques for software include structured programming, information hiding and modularization [14]. An example for hardware includes rigorous design rules [14]. In terms of real-time, one example is the use of a real-time programming language that provides higher-level constructs for implementing the timing requirements of the system;

- fault tolerance: a consensus in the dependability area [40, 42] is that the design of a fault-free system is difficult to achieve and that mechanisms to deal with faults should be provided. The fault tolerance step aims to provide mechanisms to cope with possible faults that remain or develop in the system during its operational service. In the next section, the basic steps usually employed for building a fault tolerant system are presented. Examples in terms of real-time include the use of timing error detection mechanisms and temporal isolation policies for isolating the temporal damage of the errant threads;

- fault removal: this aims to remove faults from the system during both development and operational phases. An example of fault removal during the development phase
2.2 Fault Tolerance

is the use of model checking to uncover system faults. In terms of real-time, this includes the utilization of feasibility analysis techniques to the system;

- fault forecasting: this step includes techniques used to perform an evaluation of the system with respect to the current number and future occurrence of faults. Techniques used for fault forecasting include the use of quantitative analysis methods, like Markov Chains. In terms of real-time, the same methods could be used to predict the occurrence of deadline misses in real-time systems;

Dependability is a concept that integrates different aspects. As described previously, different emphasis (with respect to dependability of a system) can be put on different properties (or attributes). Attributes of dependability include [14, 15]:

- availability: readiness for correct service;
- reliability: continuity of correct service;
- safety: absence of catastrophic consequences on the user(s) and the environment;
- confidentiality: absence of unauthorized disclosure of information;
- integrity: absence of improper system state alterations;
- maintainability: ability to undergo repairs and modifications.

With particular interest to this work is the reliability attribute. According to [42], reliability can be characterized by a function $R(t)$ expressing the probability that the system conforms to its specification (provides correct service) throughout a duration $t$. In this definition, timing errors\textsuperscript{2} can always impact on the system’s reliability, since the system can potentially provide an incorrect service (timing failure being considered in this

\textsuperscript{2}Of course, functional errors have a similar impact on the system.
research). Therefore, the use of fault tolerance techniques to deal with such timing errors, during the operation of the system, can improve the system’s reliability by detecting erroneous conditions and possibly confining them, so that the fundamental chain (described previously) stops as soon as possible, with minimal damage to the overall system.

Figure 2.1 presents a summary of the threats to, means to, and the attributes of dependability found in the literature of dependable systems, presented previously.

![Dependability Tree](image)

Figure 2.1: The dependability tree (taken from [14]).

### 2.2.2 Achieving Fault Tolerance

As described in the previous section, a consensus in the dependability area [40, 42] is that the design a fault-free system is difficult to achieve (if not impossible). Therefore, mechanisms to deal with faults during the systems run time execution are necessary. Fault tolerance is the step that provides the basis to cope with possible faults in the system. Any fault tolerance technique depends on the use of **redundancy**. A redundant element is one that would not be needed if the system was fault-free [42]. Following this definition, all the timing error detection mechanisms described in this research are redundant, as they
would not be needed if the system was guaranteed to be correct. In order to tolerate a fault, certain steps are usually followed. According to [42], four steps should be followed:

- error detection: due to the nature of a fault, the direct detection of a fault is not possible. However, its first effects (the error) are. The first step in fault tolerance is to detect errors;
- damage confinement and assessment: the detection of an error means that another portion of the system state may be suspect of being erroneous as well. Due to the nature of errors, before dealing with the error, it may be necessary to assess the damage of the error in the system. A possibility is to also limit the damage of the error, via confinement;
- error recovery: this step includes well known techniques that can be used to recover the system from a fault;
- fault treatment and continued service: the last step of fault tolerance aims to effectively correct the fault (without the need to activate error recovery every time the fault becomes active and an error is detected).

In the next sections, each one of these steps is described in more detail.

### 2.2.2.1 Error Detection

At the starting point of any strategy for fault tolerance is the detection of the erroneous state of the system caused by a fault. In principle, the provision of more error detection mechanisms in a system will lead to a more reliable system (more errors are detected and dealt with, reducing the chance of system failures) [42]. Unfortunately, in practice there will be limitations imposed on the system with respect to the amount of error detection that can be provided. Some limitations include the cost of redundancy for error detection
and overheads incurred at run time due to extensive detection [42]. In [42], a broad classification of techniques for error detection is provided. In particular, this research focuses on timing error detection, which aims to detect if the timing constraints of the system are being met. A signal to the system is generated if the constraint is violated. A simple example, already described in this thesis, is deadline monitoring. When a deadline is missed, the system signals to the application that the constraint was violated.

### 2.2.2.2 Damage Confinement and Assessment

Once an erroneous state has been detected in the system, the first question arising is to what extent has the error damaged the system. In the damage confinement and assessment step, the system may try to enforce certain constraints upon the error detected in the system. Note that, in order to constraint behaviour upon the system, it is necessary to know the exact nature of the error, so that the system may not increase the damage that has already been done. An example of a damage confinement technique in terms of real-time is to stop a thread from executing once a deadline is missed. This can stop the particular thread from interfering with the execution of other threads in the system. Nevertheless, depending on the current state of the thread (e.g., if it is executing inside a shared resource or if the thread is critical to the system), stopping the thread may not be the best policy for damage confinement.

The damage assessment part of this step is fundamentally interrelated to both error detection and error recovery steps. Indeed, from the error detection point of view, the assessment can be done automatically by analysing which error occurred and which thread was affected. Moreover, from the error recovery point of view, the assessment is fundamental by defining points in time that can be used, in case of an error, to rollback the system to a state without the error (checkpointing techniques). Due to this nature, damage assessment can be viewed as an uncertain and incomplete aspect of fault tolerance [42].
2.2 Fault Tolerance

In fact, some researchers [40, 14, 15] do not include this step as being used to build fault tolerance techniques.

2.2.2.3 Error Recovery

Error recovery techniques are central to the use of fault tolerance, as they provide the means to transform the system from an erroneous state into another state without errors, where the system can continue delivering correct service, albeit with a possible degradation on its quality. Basically, three different approaches to error recovery can take place [40, 14]: (i) rollback; (ii) compensation; and (iii) rollforward.

In the (i) rollback approach, the aim is to bring the system back from an erroneous state to a previous state prior to the occurrence of the error. In order to rollback from the current state, the system makes use of checkpointing, where the system establishes information necessary for restoring previous states. In the (ii) compensation approach, the system provides enough redundancy so that, even in an erroneous state, the system can still manage to deliver a correct service. For instance, in terms of real-time, the change of mode of a system to provide a degraded service, but still timing correct and functional, is an example of a compensation approach. Finally, in the (iii) rollforward approach, the aim is to find a new state (free from the error) where the correct service can still be provided.

2.2.2.4 Fault Treatment and Continued Service

All the techniques discussed so far can be used to deal with the error caused by a fault in the system. Therefore, once an error has been discovered, nothing stops it from happening again later, in the execution of the system, because its cause (the fault) has not been eliminated. The last step in fault tolerance aims to remove the fault from the system, so that a continued service (without that fault) can be provided. Fault treatment can be basically divided in two parts [42]: (i) fault location; and (ii) system repair.
The (i) fault location part aims to diagnose the system, via a number of checks, so that the specific type of the fault can be determined and located. Once a fault has been located, the (ii) system repair can take place. Techniques for system repair are based on reconfiguring some characteristic of the system, to remove the fault, possibly leading to a reinitialization of the system. For instance, if the fault in the system is a software bug, a possible system repair is to correct the bug (reconfigure) via a patch. This then, may lead to the reinitialization of the system.

2.3 Fault Tolerance for Timing Errors

Traditionally, real-time systems have focused on the detection of deadline misses as the mechanism for detecting timing errant behaviour in real-time systems. One of the key assumptions of this thesis is that a deadline miss is in fact a propagation of one or more timing errors in the system, for example, where a thread (or threads) in the system executes further to its computation time (WCET), used in the timing analysis of the system. The source that generates the timing error is assumed to be a (timing) fault relative to the timing requirements of the system, like the WCET calculation or the assumptions for the release times of sporadic threads. In fact, the WCET can be overrun during the run time execution of the system not just because it was miscalculated, but also due to functional faults that can occur in the system (e.g., an unbounded loop error or an erroneous variable attribution).

The real-time systems literature has recognized that the traditional approach of deadline miss detection may not be enough for guaranteeing certain timing constraints for the system. Therefore, several approaches have focused on providing mechanisms for detecting execution time overruns and, in certain cases, applying different policies for confining the behaviour of the errant thread. Work following this direction is considered in the next
Moreover, error detection mechanisms for timing errors provide the means to detect when the system is becoming overloaded, i.e., the computational demand requested by the set of threads composing the system exceeds the time available in the processor [26]. In the literature, a system under overload conditions has to guarantee that its critical threads will meet their deadlines. Therefore, following the fault tolerance steps, handling the overload condition is part of a damage assessment and also an error recovery mechanism for real-time systems. Although this research work does not focus on error recovery issues for timing errors, it does provide the basis for implementing them with more accuracy, since the timing errors can be detected earlier and, consequently, overload conditions can be categorized with more precision. Therefore, in Section 2.3.2, a brief overview of the mode changes technique, which can be used to provide measures for overload situations in real-time systems is given.

2.3.1 Detection and Confinement of Execution Time Overruns

Some work in the real-time community has targeted the detection of execution time overruns, followed by the possibility of confining the temporal damage using a pre-defined policy. In this section, different approaches are presented. Firstly, the error detection mechanism for execution time overruns in the Ada programming language is discussed. In Section 2.3.1.2, the approach implemented in the Chimera Real-Time Operating System (RTOS) is presented. The Time-Aware Fault-Tolerant (TAFT) scheduling architecture is covered in Section 2.3.1.3, and Section 2.3.1.4 describes an approach for extending the RTSJ in order to incorporate the detection of execution time overruns.
2.3.1.1 The Ada Programming Language

Burns and Wellings [23] presented one of the early works arguing for the need to monitor the execution time of threads. The discussed model was for the Ada 9X programming language and it used the CPU time accounting package for measuring the execution time. One of the motivations for monitoring the execution time was the support for fault tolerance for timing errors. Indeed, this motivation comes from the fact that a system is not capable of guaranteeing that all its deadlines will be met if one or more threads overrun their allocated execution time (the timing error). In their work, the detection of the execution overrun had a close relation to policies for damage confinement for errant threads.

Assuming an accurate measurement of the WCETs, the WCET was defined to be the budget time (the execution time allocated to the thread) plus an error handling time (the time needed by the system to take some action upon the detection of the overrun of the budget time). According to their view, upon the detection of an overrun, three different strategies could be followed by the system [23]: (a) allow the overrunning thread to continue executing, hoping for the system to stabilise – that may happen if other threads do not execute all their WCETs; (b) do not allow threads to overrun, by terminating the overrunning thread and performing error handling; (c) a combination of (a) and (b), where non-crucial threads (soft real-time threads) can overrun, but crucial threads cannot, ensuring that no overrun can result in a crucial thread missing its deadline. Note that, no implementation was actually provided in [23].

Later, Harbour et al [34] showed an implementation and use of execution time clocks and timers in Ada. Underlying the implementation was a POSIX/C execution-time clocks and timers model. The Ada implementation used a special architecture for arming the execution time timers and generating handlers. The architecture used the proposed package CPU_Time. The adoption of the following schemes were discussed upon the detection
2.3 Fault Tolerance for Timing Errors

of an overrun [34]: (a) handled, where the overrun is detected but the thread is allowed to complete its execution; (b) stopped, where the overrunning thread is stopped, so that lower priority threads can execute; (c) imprecise, in this scheme the imprecise computational model [46] can be built; (d) lowered, where the priority of the overrunning thread is lowered.

In the last version of the Ada programming language (Ada 2005), the support for monitoring the execution time of threads was fully adopted [16]. The monitoring is provided in a very similar manner to the approach proposed previously by Burns and Wellings [23] and Harbour et al [34], with the provision of CPU clocks and timers.

2.3.1.2 The Chimera Real-Time Operating System

In [71], Stewart and Khosla proposed a mechanism for detecting two types of timing errors: deadline misses and execution time overruns. The assumption was that the measurement of WCET may not be always correct due to [71]: (i) interrupts executing longer or occurring more frequently (stealing cycles from higher priority threads); (ii) variations in speed (due to caching, pipelining and bus arbitration); (iii) no easy way to accurately measure execution times for embedded code. The work provided a policy independent mechanism for monitoring and detecting the timing errors. The term policy independent means that the mechanism can be used to detect timing errors on top of different scheduling policies, like FPS or EDF. The mechanisms for error detection were implemented in the Chimera RTOS [72].

In order to detect deadline misses, the system sets a timer by which time the computation should have finished. If the timer expires and the thread has not finished its computation, a Timing Failure Handling (TFH) function associated to the thread is executed. With respect to the implementation of the detection of execution time overruns, the system associates the execution time to a software down-counter. Every execution time
cycle executed by the thread decrements this counter. If the counter reaches zero, this means the allocated time has been used up, and the associated TFH function is executed. After executing the TFH function, three options are provided for resuming processing of the thread [72]: (a) restart, causes the thread to return to the beginning of its code (using a mechanism similar to \texttt{setjmp()} and \texttt{longjmp()} found in the POSIX/C standard); (b) continue, the thread continues its execution from the point it was interrupted; (c) exit, used to kill the thread (it is implemented by calling the \texttt{return()} function in the POSIX/C standard).

### 2.3.1.3 Time-Aware Fault-Tolerant Scheduling

In [51], Nett \textit{et al} describes the Time-Aware Fault-Tolerant (TAFT) scheduling architecture. The TAFT architecture is used to tackle the issue of: preserving predictability by guaranteeing timely behaviour when threads, with estimated execution times, have to be scheduled online [51]. The proposed architecture is depicted in Figure 2.2.

![Figure 2.2: The TAFT scheduling architecture (taken from [51]).](image_url)

As presented in Figure 2.2, the TAFT architecture has two components: (i) monitoring; and (ii) FT scheduler. The monitoring component extracts timing information from the
application. The timing information is used to derive the ECET (Expected Case Execution Times) and pass it to the FT-Scheduling component. The ECET\(t,x\) equates to \([51]\): the execution time that has to be assigned to a release \(t\) in order to get a probability of \(x\) that \(t\) is completed within this time. In order to predict the ECET\(t,x\) for future instances of the thread, the ECET\(t,k,n\) is defined to be \([51]\): the minimal execution time needed to complete at least \(k\) out of \(n\) last executions of the thread before the release \(t\).

During execution, the prediction of the ECET may be wrong. The FT scheduler is responsible to ensure the predictability of the system also in this case. To do that, the thread has to be able to adapt to the possible change of timing parameters. Indeed, the timing error occurs in the architecture when the thread deviates from the estimated execution time. Therefore, every thread is designed as a TaskPair (TP). The TP is constituted by a MainPart (MP) and the ExceptPart (EP). When the EP is executed, it ensures that the TP leaves the application in a fail-safe state and the controlling system in a consistent state. This model is closely related to the model, presented previously, proposed by Burns and Wellings in \([23]\) for Ada.

The implementation of the TAFT scheduling architecture was done in a CORBA middleware. In order to implement the monitoring part, the application code is instrumented with code that generates some events. These events are interpreted by the monitoring component to derive information of the execution times. Note that this work on the TAFT scheduling architecture applies to real-time distributed systems.

### 2.3.1.4 An Extension to the RTSJ

Since an implementation of the execution time monitoring mechanism defined in the RTSJ is not found in the literature, the work by Masson and Midonnet \([50]\) proposed a way to implement an overrun detection mechanism on top of the RTSJ. Though, the computational model considered is only capable of handling periodic real-time threads. The basic
idea of the proposed overrun detection mechanism is to associate a periodic timer (used to
detect the overruns) to a periodic thread. The period value is the same for both periodic
timer and periodic thread, but an offset equal to the WCET of the periodic thread is set
to the periodic timer. This way, when the periodic timer starts, it checks if the periodic
thread has finished. If it has not, an overrun is detected. Although easy to implement, this
model is not capable of dealing with more complex computational models, for instance,
when shared resources are used in the system.

With respect to the handling of the overrun, the work describes three different poli-
cies [50]:

- stopping the erroneous thread: this approach is also present in the previous works
described in this chapter. Though, it is a pessimistic approach, in the sense that it
does not provide the thread with the opportunity to finish its current execution;

- stopping the erroneous thread after an allowance is granted to the thread: in this
case, when an overrun is detected, the system executes an online feasibility analysis
of the system (a response time analysis in this particular case) using the current
timing behaviour. This analysis is used to find out which allowance can be granted
equitably to the threads in the system and the system still be feasible. This allowance
is added to the cost of the thread, so that it can finish its execution;

- stopping the erroneous thread after an allowance is granted to the system: like in the
previous case, an online feasibility analysis is carried out, upon the overrun detection.
The difference is that the calculated allowance is given to the overrunning thread
with highest priority in the system.

These comprise the different policies proposed for damage confinement against the
overrunning thread. In one aspect, this work [50] goes further to the policies described
in the previous works, because it makes an online calculation of the possible allowances that can be granted to the system. The problem with this approach is that it has to call a response time analysis (an additional overhead) whenever an execution time overrun occurs in the system.

### 2.3.2 Mode Changes for Assessment and Recovery Strategies

The use of error detection mechanisms for timing errors provide the means to detect when a real-time system is becoming overloaded, i.e., the computational demand requested by the set of threads composing the system exceeds the time available in the processor [26]. During transient overload situations it is necessary to guarantee that critical threads of the system will continue executing. In order to provide this guarantee, the real-time system has to introduce some form of reconfiguration, changing its operational mode.

For instance, if the strategy used for confining the temporal damage has not worked, and the system is heading towards a failure, acceptable forms of mode changes include [56]:

- change to a safe state, and then stop the system;
- change to a safe state, and continue the execution but with a degraded functionality;
- change to another state, where different threads assume the execution of the system, and the system can still maintain its full functionality.

The complexity found in the implementation of mode changes is quite considerably. For instance, in the first approach, finding a safe state in a system where several threads are running concurrently is a difficult task (with respect to the synchronization of the whole system). The same is true for the other approaches. In the approach where a degraded functionality is provided, one possibility is to provide degraded modes of functionality, switching to them whenever a specific critical thread of the system fails. In the last approach, the redundancy required is very high at both hardware and software levels [56].
2.4 Summary

This chapter started with the presentation of a sufficient and necessary (therefore exact) feasibility analysis, called response time analysis, for fixed-priority real-time systems. The priority inheritance algorithms considered in this work, and how response time analysis has to be extended to consider them, has been described. Moreover, the concept of servers for preserving and bounding the execution of aperiodic and sporadic threads was presented, in conjunction with the changes to the response time analysis considering their use.

Then, an introduction to fault tolerance was shown. Specifically, the notion of dependability and its threats, means and attributes were presented. In a wider context, dependability is the research area that encompasses fault tolerance. In the description of the fault tolerance area, the four constituent parts were depicted: (i) error detection; (ii) damage confinement and assessment; (iii) error recovery; (iv) fault treatment and continued service. Error detection (in the context of timing faults) is the focus of this thesis.

In the final section of this chapter, different approaches aiming to either detect or confine timing errors in real-time systems were presented. Also, a brief overview of the mode changes approach for damage assessment and recovery was presented. The next chapter discusses general models in both the POSIX/C standard and the RTSJ for detecting and possibly confining timing errors.
The early detection of timing errors can improve the reliability of fault tolerant real-time systems by facilitating the next phases, damage confinement and assessment as well as error recovery, normally used in the construction of fault tolerant systems. Ideally, the earlier the timing error is detected, the better, as the goal of the system is to limit the propagation of the error, so that error recovery can occur as soon as possible and with more accurate information. The main idea of this chapter is the provision of two different models for the run time detection of timing errors, applicable to real-time systems, in terms of POSIX/C and the RTSJ. Initially, the assumptions surrounding the computational model utilized in the detection models are discussed. The possible timing errors, considered in this research, which could develop in the assumed computational model are described in Section 3.2. Section 3.3 describes the main components of the POSIX/C standard and a run time error detection model for it. Another error detection model, provided in terms of the RTSJ, is shown in Section 3.4. Differently from POSIX/C, the RTSJ includes higher-
level programming constructs that facilitates the implementation of damage confinement for timing errors.

3.1 Computational Model Assumptions

The two different models for the run time detection of timing errors, described in this chapter, are implemented in this thesis using a uniprocessor architecture and a FPS scheduler. Moreover, it is assumed in this work that the computing system is capable of multi-threading, i.e., having different threads in the system competing to gain access to the processor. Threads can communicate with each other via shared-memory and are allowed to access other (possibly blockable) resources from the system. A thread can suspend itself (e.g., by issuing a call to the sleep method for a certain amount of time or waiting on a condition variable) at any point of its execution, i.e., inside or outside a shared resource.

The scheduler is responsible for selecting the next thread in the system to execute on the processor. The execution eligibility assumed for the scheduler is based on priorities, hence FPS is assumed. Though, the way priorities are assigned, for instance using a monotonic order assignment \([44, 22]\), is not relevant. Shared resources are bounded against the priority inversion problem with the use of priority inheritance protocols for uniprocessor systems and the FPS approach \([67, 57]\), as described in the previous chapter.

In typical real-time systems, threads are normally classified according to release intervals. Therefore, a thread can be periodic (a fixed interval (period) of release), aperiodic (that is, not periodic) or sporadic (there is an MIT between two consecutive releases). In this work, all three types of threads are assumed to be used in the construction of a system.

Another restriction of the computational model is that threads have their deadline values less than or equal to their period values. The main reason is that the majority of real-time run time execution environments found in the literature do not support threads
that have their deadlines greater than their periods. In fact, if a deadline greater than period is adopted in FPS, the response time feasibility analysis has its complexity increased considerably [74].

It is important to note that the computational model assumed in this thesis is compatible with the computational models specified by both the POSIX/C standard and the RTSJ, used in this research. On the one hand, the POSIX/C standard does provide support for different computing architectures besides a uniprocessor architecture. On the other hand, the current specification of the RTSJ is essentially defined in terms of a uniprocessor architecture.

### 3.2 Timing Errors

As considered in this research, the possible timing errors that can occur in a real-time system can be described as deviations from the feasibility analysis, which is used to test the temporal behaviour of the system. Therefore, in an ideal feasibility analysis test, like response time analysis (considered in the last chapter), any deviation in the following timing factors of the system can lead to a timing error:

1. **Assumptions of the WCETs for the threads in the system.** The WCET describes the maximum computation time that a thread can execute in a given release. They are calculated with either static analysis methods or measurement-based techniques. However, the determination of accurate execution time values is notoriously difficult to achieve. Therefore, imprecise values can be used as input to the feasibility analysis;

2. **Interference times of higher priority threads executing in preference to lower priority threads.** Due to the priorities used in the scheduling process, lower priority threads have to wait for higher priority threads to finish their execution in order to start
executing. The values used for calculating the time that lower priority threads have to wait are based on the WCETs of the higher priority threads [24];

(iii) assumptions of the potential blocking times that higher priority threads suffer when lower priority threads are accessing shared resources. Priority inheritance protocols, as described in the last chapter, can be used to bound the blocking time of higher priority threads. The potential blocking times are calculated using the WCETs related to the shared resource segments that higher priority threads share with lower priority ones. As presented previously, WCETs can be imprecise and so can be the potential blocking time values being used;

(iv) guarantees about the release time of both periodic and sporadic threads. Providing this guarantee for periodic threads is quite straightforward, as the threads are released in constant periods of time. For sporadic threads, worst-case assumptions regarding the MIT should be provided. However, it is always possible that, during the run time execution of the system, these assumptions may fail.

It is possible to identify four possible fault-error-failure chains, based on the four major timing factors used in the feasibility analysis test, which could be generated in a real-time system:

1. \textit{fault} (in thread’s $T_i$ WCET assumptions) $\rightarrow$ \textit{error} (overrun of thread’s $T_i$ WCET) $\rightarrow$ \textit{error propagation} (deadline miss of thread $T_i$) $\rightarrow$ \textit{failure} (to deliver service in a timely manner);

2. \textit{fault} (in thread’s $T_i$ WCET assumptions) $\rightarrow$ \textit{error} (overrun of thread’s $T_i$ WCET) $\rightarrow$ \textit{error propagation} (greater interference on lower priority threads) $\rightarrow$ \textit{error propagation} (deadline miss of lower priority threads) $\rightarrow$ \textit{failure} (to deliver service in a timely manner);
3.2 Timing Errors

3. fault (in thread’s $T_i$ WCET assumptions executing inside a shared resource) $\rightarrow$ error (overrun of thread’s $T_i$ WCET inside a shared resource) $\rightarrow$ error propagation (greater blocking time for higher priority threads sharing the resource) $\rightarrow$ error propagation (deadline miss of higher priority threads) $\rightarrow$ failure (to deliver service in a timely manner);

4. fault (in thread’s $T_i$ MIT assumptions) $\rightarrow$ error (overrun of thread’s $T_i$ MIT assumptions) $\rightarrow$ error propagation (greater interference on lower priority threads) $\rightarrow$ error propagation (deadline miss of lower priority threads) $\rightarrow$ failure (to deliver service in a timely manner).

One of the assumptions in this research is that the feasibility analysis is correct. Hence, the violation of feasibility analysis parameters during run time is caused by a fault in the system. A closer look at the four possible chains of dependability threats shows that, essentially, four different types of timing errors can occur in the assumed computational model:

(a) deadline miss;

(b) execution overrun of the thread’s WCET;

(c) execution overrun of the thread’s WCET inside a shared resource;

(d) violation of the thread’s MIT assumptions.

A review of possible mechanisms, found in the literature, for detecting these timing errors, and possibly providing some form of damage confinement was presented in the previous chapter. In the next sections, two models for detecting timing errors are discussed. The first one is provided in terms of clocks and timers, as primitives, of the POSIX/C standard that can be used by developers to detect the timing errors. The second model
is provided in terms of the RTSJ and, due to the higher-level programming constructs available in the RTSJ, provides some facilities for damage confinement for the timing errors.

3.3 The Portable Operating System Interface

POSIX (Portable Operating System Interface) represents a family of standards developed by the PASC (Portable Applications Standards Committee) of the IEEE Computer Society. The current standard, found in [3], has been jointly developed by the IEEE Computer Society [4] and The Open Group [9]. It is simultaneously [3] the IEEE Std 1003.1 and the Open Group Base Specifications Issue 6. Moreover, it is also an ISO/IEC (International Organization for Standardization) [5] standard, number ISO/IEC 9945.

The standard describes a set of fundamental services needed for the construction of application programs. Access to these services is provided in the standard by the definition of an interface, using the C programming language\textsuperscript{1}, a command interpreter, and common utility programs that establish standard semantics and syntax [3]. The POSIX/C specification consists of four different parts: (a) base definitions; (b) system interfaces; (c) shell and utilities; (d) rationale.

Several principles guided the development of the POSIX/C standard [3], including:

- application-oriented: where the basic goal is to promote portability of applications across UNIX system environments by developing a clear, consistent, and unambiguous standard for the interface specification of a portable operating system based on the UNIX system documentation;

- interface, not implementation: by not providing a distinction between library func-

\textsuperscript{1}The C programming language is used in this research. Hence, the abbreviation POSIX/C is used throughout the text.
tions and system calls, referring to both as functions. No details of the implementation of any function are given;

- source, not object, portability: so that an application written and translated for execution on one conforming implementation may also be translated for execution on another conforming implementation;

- the C programming language: system interfaces and header definitions are written in terms of the standard C programming language as specified in the ISO C standard [35];

- no super user, no system administration: no intention to specify all aspects of an operating system;

- minimal interface, minimally defined: the mandatory core facilities of the standard are kept to as minimal as possible. Additional capabilities have been added as optional extensions;

- broadly implementable: where the developers of the standard endeavoured to make all specified functions implementable across a wide range of existing and potential systems.

Specifically for real-time system, the POSIX/C standard provides profiles (subsets) [53] addressing different requirements of different applications. Four different profiles are defined for: (i) small embedded systems; (ii) industrial controllers; (iii) large embedded systems; (iv) general-purpose computers with real-time requirements. In the implementation and evaluation of the mechanism proposed in the thesis, the open-source RTOS MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) [59, 7] is used. MaRTE OS implements the Minimal Real-Time POSIX.13 subset (for small em-
Therefore, the main capabilities of POSIX/C that are of interest include:

- threads: functions that enable the creation and control (e.g., setting priorities) the use of threads;

- signals: POSIX/C defines functions for setting and handling signals generated by the system. For instance, a signal generated by a previously specified timer;

- sporadic servers: POSIX/C provides support for bounding aperiodic and sporadic threads, by describing certain fields that, when passed to a thread, are used to implement the Sporadic Server (SS) protocol described in the last chapter;

- priority inheritance: two different priority inheritance protocols can be used according to the POSIX/C standard: (a) Basic Priority Inheritance (PI); and, (b) Priority Protection (PP);

- clocks and timers: clocks and timers with nanoseconds granularity, for real-time applications, are defined in the standard;

- execution-time clocks and timers: the standard also defines the use of execution-time clocks and timers for focusing on the time that a thread spends executing the processor.

In order to illustrate the use of POSIX/C, Figure 3.1 shows an example of a periodic thread implementation. The periodic thread is kept in a loop, where it does its periodic computation. After executing its computation, the thread calculates start time (in an absolute time value) for its next release (add_timespec function) and then sleeps (clock_nanosleep function) until the start of the next period.
3.3 The Portable Operating System Interface

periodic_thread() {
    /* periodic loop */
    while (1) {
        /* code executed by the thread */
        /* thread sleeps until its next release */
        add_timespec(&next_release, &next_release, &params->period);
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next_release, &remaining_time);
    }
}

Figure 3.1: Example of a periodic thread in POSIX/C.

3.3.1 A Model for Error Detection

The POSIX/C standard already provides the basis for detecting the timing errors discussed:

(a) deadline miss: using clocks and timers that are released at the time of a deadline miss;

(b) execution overrun of the thread’s WCET: as presented previously, one of the capabilities of the POSIX/C standard is the use of execution-time clocks and timers. This enables the measurement of the time a thread uses the processor. By setting an execution-time timer to the WCET of the thread, it is possible to detect when the thread overruns;

(c) execution overrun of the thread’s WCET inside a shared resource: using execution-time timers, it is also possible to detect when a thread overruns its maximum execution time inside a shared resource. Unfortunately, this approach has three drawbacks. Firstly, the granularity of the time provided in this approach is down to a certain piece of code, where additional overhead is placed every time a thread enters and leaves the shared resource. Secondly, upon the detection of the overrun, the system is only capable of determining the thread that is causing the blocking overrun. Thirdly, in large real-time systems, where it is possible to have a large number of
nested shared resources, it becomes a cumbersome and difficult task to set correctly the timers in the system;

(d) violation of the thread’s MIT assumptions: using normal real-time clocks and timer found in the standard, it is possible to detect when an MIT is being violated. Moreover, the standard provides the implementation of a sporadic server for providing support, enforcement in this case, for aperiodic and sporadic threads.

Because of the discussed problems, this research work contends that the error detection mechanism for item (c) is not ideal and another mechanism could be used. Therefore, an extension of the POSIX/C standard, considering the introduction of blocking-time clocks and timers, enables the detection of overruns inside a shared resource. In Figure 3.2, the error detection model proposed for the POSIX/C standard is presented. The figure also depicts the basic components of POSIX/C that are needed.

As described in Section 3.1, the computational model uses a uniprocessor architecture and FPS. The basic computation unit is provided in terms of threads. A thread can then be used to implement a periodic behaviour. Support for the execution of aperiodic and sporadic threads is provided with the use of sporadic servers, by defining certain
parameters in the creation of a thread. It is essential to have the support for priority inheritance protocols, as they provide the means to bound the maximum blocking time a higher priority thread may experience whilst a lower priority thread is executing inside a shared resource. The mechanisms that can be used for detecting the timing errors and already found in the POSIX/C standard are as follows: (a) normal real-time clocks and timers, which can be used to set the deadline by which time the thread should have finished its execution, and detect MIT violations; (b) execution-time clocks and timers, which accounts only for the time a thread spends using the processor.

The proposed mechanism for complementing the POSIX/C standard for the possible timing errors is described as blocking-time clocks and timers. The idea of blocking-time clocks and timers is similar to that of execution-time clocks and timers, but instead of accounting for the time a thread spends executing the processor, it focuses on the time a higher priority thread is blocked by lower priority threads when using the priority inheritance algorithms. The mechanisms described in this error detection model are discussed in more detail in Chapter 4.

3.4 The Real-Time Specification for Java

The first proposal towards a real-time extension to Java was defined under the auspices of the NIST (National Institute of Standards and Technology) [29]. Basically, some guiding principles and requirements for developing a Real-Time Java were defined. Following this work, the Real-Time for Java Experts (RTJE) produced an initial version of the Real-Time Specification for Java (RTSJ) [20]. The main objective of the RTSJ is to extend the Java programming language and Java Virtual Machine (JVM) specifications providing an Application Programming Interface (API), where Java threads with timeliness constraints can be correctly created, analysed, executed and managed [20].
Although the RTSJ mentions the NIST guiding principles and requirements, it defines its own guiding principles. The principles are used to delimit the scope of work and also introduce some compatibility requirements for the RTSJ. In particular, the guiding principles states that the RTSJ shall [18]:

- not include specifications that restrict the applicability to particular Java environments;
- be backward compatible with the execution of non-real-time applications;
- support the principle of “Write Once, Run Anywhere” but not at the expense of predictability;
- address current real-time system practice and allow future implementations to include advanced features;
- give the highest priority to predictable execution in all trade-offs;
- not require syntactic extension to the language;
- allow variation in implementation decisions.

Based on these principles, the design of the RTSJ enhances the following areas of the Java programming language and virtual machine [20, 76]:

(i) time and clocks: the model for time and clocks supported by the Java language is based on calendar time and UTC (Coordinated Universal Time). Unfortunately, this model is not sufficient to express some timing requirements found in real-time systems. Therefore, the RTSJ defines a hierarchy of time classes that are used to express time with nanosecond granularity;
(ii) scheduling model: in the JVM’s scheduling model it is not possible to determine in a predictable way the next thread that will execute the processor. Also, the range of priorities given by the JVM for threads is very limited for use in real-time systems. To deal with these problems, the RTSJ firstly introduces the notion of a schedulable object, which can have specific requirements for: release (timeliness requirements for the execution of the thread), memory (allocation of memory), and scheduling (execution eligibility requirements for the thread). Secondly, the RTSJ specifies a FPS scheduler, called the base scheduler. In particular, the RTSJ recognizes the importance of execution time monitoring and also specifies an execution time monitoring model for real-time threads. Though, this facility is considered optional in the RTSJ (further details about this model are discussed in Chapter 6);

(iii) synchronization and resource sharing: the mechanism used in the Java programming language for communication and synchronization is based on mutual exclusion. Unfortunately, such mechanism suffers from the priority inversion problem. Therefore, the RTSJ supports the Basic Priority Inheritance (PI) and Priority Ceiling Emulation (PCE) algorithms. Moreover, it also specifies a new priority inheritance algorithm. This algorithm allows for a mixture of both simple priority inheritance and priority ceiling protocols in the same application. Though, this research work does not tackle this particular protocol;

(iv) asynchronous transfer of control: in Java, the interrupt mechanism provides an asynchronous notification mechanism for certain activities. However, the model is synchronous and requires a running thread to keep polling for the reception of the notification [76]. Due to the delay required by this method, the RTSJ introduces an alternative approach, consisting of the use of Asynchronous Transfer of Control (ATC). The ATC model is integrated with the Java exception handling mecha-
nism [76]. It requires the definition of an ATC event and methods that allow the delivery of ATC events;

(v) memory management: in the memory management model of the Java programming language, all objects are allocated in the heap. In order to free memory areas that are no longer used, the JVM has to implement a garbage collector. Unfortunately, the use of a garbage collector impacts on the ability to analyze the timing properties of the application. To deal with this, the RTSJ extends the memory management model of Java introducing the notion of memory areas. Memory areas are allocated outside the heap and are not subject to garbage collection;

(vi) physical and raw memory access: the RTSJ provides one mechanism to place objects into areas of memory that have particular properties or access requirements (e.g., Direct Memory Access (DMA)) [76]. It also defines a mechanism to access raw memory locations that are being used to interface the outside world (e.g., memory-mapped input and output device registers) [76]. In order to provide this mechanism, the RTSJ introduces the possibility of accessing raw memory by reading and writing Java variables.

For the purposes of this research, the most important areas of improvement of the RTSJ to the Java programming platform are the scheduling model and the synchronization and resource sharing. With respect to the scheduling model, the RTSJ introduces the notion of a Schedulable Object (SO) [20] for implementing real-time threads and asynchronous event handlers. As the name implies, a schedulable object is an object that has attributes that allow a run time scheduler to perform admission control and dispatching. These attributes are captured by a Java interface called the Schedulable interface. An SO is any object that implements this interface. Two types of SOs are defined by the specification: RealtimeThreads and AsynchronousEventHandlers. This research focus is on
3.4 The Real-Time Specification for Java

real-time threads.

One of the attributes that can be associated with a real-time thread\textsuperscript{2} is the \textit{ReleaseParameters}. Each thread executes in response to release events. Each \textit{release} can be identified by a natural number. Hence, the \textit{0}\textsuperscript{th} release is the thread’s execution in response to its initial release event; the \textit{i}\textsuperscript{th} release is the thread’s execution in response to its \textit{i}\textsuperscript{th} release event. The term \textit{complete} is used to indicate that the thread has finished the execution associated with a particular release. Consequently, the \textit{i}\textsuperscript{th} completion event is the event that occurs when the thread indicates that it has completed the execution associated with the \textit{i}\textsuperscript{th} release event.

The \textit{ReleaseParameters} attribute provides information about the estimated processor consumption cost per release (budget) and any deadline associated with when a release must be completed. It is also possible to specify an interval for the thread if it is released periodically (the period of the thread) or sporadically (the Minimum Inter-arrival Time (MIT) of the thread). The \textit{RealtimeThread} class both encapsulates the operations that can be applied on a thread and provides the infrastructure the thread needs to execute.

The current model of real-time threads in the RTSJ only fully supports periodic release parameters. Recently, the Expert Group of the RTSJ has extended this model to also consider sporadic and aperiodic release parameters \cite{37}. This new model is the one considered in this work. The semantics is described fully in Chapter 5, where the model of real-time threads is analysed. One important method supported by real-time threads is the \textit{waitForNextRelease()} method (wFNR). A thread calls this method when it wishes to complete its current release and wait for the next release event. The code to be executed in response to each release event is usually encapsulated in the \textit{run()} method of the real-time thread. It typically takes the form illustrated in Figure 3.3.

\textsuperscript{2}From now on, the term thread is used to mean a real-time thread. Java threads are supported by the RTSJ but they have regular Java semantics and, consequently, are not covered by the model that is discussed now.
A thread can be descheduled by the application by calling the `deschedule()` method in its associated thread object. When this happens, the thread continues its execution until it completes its current release. It receives no further release events (although they are still generated) until the `schedule()` method is invoked on its thread object. After this has occurred, the thread is able to receive its release events again. It, however, will not be released until the next release event is generated. Figure 3.4 summarises the important characteristics of the real-time thread class, with respect to scheduling. The `start()` method is called to start the first release of the thread.

```
public class RealtimeThread implements Schedulable {
    public void deschedule();
    public void run();
    public void schedule();
    public void start();
}
```

Figure 3.4: An abridged `RealtimeThread` class specification.

From a synchronization and resource sharing point of view, each schedulable object has a `base` and an `active` priority. The base priority is the priority allocated by the developer. The active priority is the priority that the scheduler uses to order the run queue. According to the RTSJ, the real-time JVM must support priority-ordered queues and perform priority inheritance whenever high priority threads are blocked by low priority ones. The active priority of a thread becomes the maximum of its base priority and the priority it has inherited.

In order to allow the developer to specify an appropriate priority inheritance algorithm
for its application-defined shared resources (those associated with any objects that have synchronized methods or that are used in a synchronized statement), three classes are defined. The abstract class \textit{MonitorControl} defines a static method that allows the default priority inheritance algorithm to be set along with a static method that allows a particular object to have the default overridden. Two subclasses of \textit{MonitorControl} are provided: \textit{PriorityInheritance} and \textit{PriorityCeilingEmulation}, which allow the developer to specify the priority inheritance and the priority ceiling emulation algorithm respectively. As presented previously, the RTSJ introduces a new algorithm which enables these two algorithms to work together in the same application.

\subsection*{3.4.1 A Model for Error Detection}

Differently from the POSIX/C standard, the RTSJ does not rely on the provision of clocks and timers, so that developers can implement the desired temporal constraints for its threads. Instead, the RTSJ focuses on the provision of a higher-level monitoring model that is integrated to the scheduling model. Currently, the discussed timing errors are addressed in the RTSJ as follows:

(a) deadline miss: a mechanism integrated to the scheduling model whereby the user sets the relative deadline for a given thread. Once a deadline is missed, an Asynchronous Event Handler (AEH) is released to deal with the deadline miss;

(b) execution overrun of the thread’s WCET: the RTSJ provides a model, called the cost monitoring and enforcement, which aims to provide damage confinement against the overrun of the pre-defined WCET (called cost) for a given thread. Unfortunately, under certain execution scenarios, the current model of the RTSJ may not correctly confine the behaviour of errant threads. This problem is shown in detail in Chapter 6, where a new revised model is proposed;
(c) execution overrun of the thread’s WCET inside a shared resource: because the RTSJ does not provide clocks and timers, specially execution-time timers, it is not possible for the developer to implement the monitoring of a particular shared resource. Therefore, a monitoring model, also integrated to the scheduling model of the RTSJ, for blocking time violations is discussed and defined in Chapter 6;

(d) violation of the thread’s MIT assumptions: the RTSJ provides a mechanism for the detection of MIT violations that is integrated to the behaviour of sporadic real-time threads (see Chapter 5). Moreover, it also specifies support for the execution of aperiodic and sporadic threads within a temporal isolation model by providing a form of Deferrable Server (DS). This takes the form of ProcessingGroupParameters.

This research contends that item (b) currently available in the RTSJ is not ideal, as it is flawed. Moreover, there is no mechanism in the RTSJ for handling item (c). Therefore, an extension to the RTSJ is proposed in order to consider the definition of a new cost monitoring mechanism and also the introduction of a blocking monitoring mechanism. In Figure 3.5, the error detection model for the RTSJ for the considered timing errors is presented. The diagram also depicts basic components of the RTSJ that are needed.

As described in Section 3.1, the computational model uses a uniprocessor architecture and FPS. The basic computation unit is provided in terms of real-time threads, defined by the RTSJ. Besides implementing threads with periodic release parameters, real-time threads can also be used to implement threads with aperiodic and sporadic release parameters. This is a new model proposed for the RTSJ, not previously implemented, which has its formal analysis presented in Chapter 5. Special support for aperiodic and sporadic computation units can also be provided with the use of processing group parameters. With respect to the support for priority inheritance protocols, the RTSJ allows the use of basic priority inheritance or priority ceiling protocol, and also defines a new protocol that
One key assumption behind this thesis is that timing errors can occur, during the run

time execution of the system, when the system’s temporal behaviour deviates from the

behaviour assumed in the feasibility analysis test. Initially, this chapter described the

enables the mixed execution of both protocols. This last protocol is not tackled in this

research work.

The mechanisms that can be used for detecting the timing errors already found in the

RTSJ are: (a) deadline monitoring mechanism, which can be used to detect a deadline

and release an AEH to handle it; (b) cost monitoring model, which enables the monitoring

of the execution time of a thread and the possibility of stopping its execution once an

execution overrun is detected. Unfortunately, the cost monitoring model of the RTSJ is

flawed, and a new model is proposed. Furthermore, a blocking time monitoring model is

also provided in order to deal with the problem of a higher priority thread being blocked

for more time than expected. This error detection model is discussed in more details in

Chapter 6.

3.5 Summary

One key assumption behind this thesis is that timing errors can occur, during the run

computational model assumptions used throughout this thesis, which are mainly focused on a uniprocessor architecture, the Fixed-Priority Scheduling (FPS) approach and the use of priority inheritance protocols (to bound the blocking time for threads in the system). Then, the possible timing errors that can occur during the run time execution of the system were presented. Indeed, four different types of timing errors were characterized: (a) deadline miss; (b) execution overrun of a thread’s WCET; (c) execution overrun of a thread’s WCET inside a shared resource; (d) violation of a thread’s MIT assumptions.

The hypothesis of this thesis states that, with respect to timing errors, different mechanisms for the earlier detection of timing errors could be used, before the occurrence of a deadline miss. Therefore, this chapter has described the POSIX/C standard and how the current model could be used to detect the specified timing errors. Based on this model, a possible error detection model for the POSIX/C standard was provided. This new detection model extends the POSIX/C standard with the introduction of blocking-time clocks and timers, which provides an alternative for the detection of blocking time overruns during runtime. The next chapter will focus on the detailed presentation of this detection model.

The Real-Time Specification for Java (RTSJ) was also presented in this chapter. Different from POSIX/C, the RTSJ introduces high-level computational units that facilitate the implementation of threads with possible timing constraints (i.e., periodic, sporadic and aperiodic threads). Moreover, the RTSJ integrates its scheduling mechanisms with the detection of deadline misses and, with respect to execution overruns, provides a model for enforcing the temporal execution time for a particular real-time thread in the system. This chapter has described how the RTSJ could be used to detect timing errors assumed in this thesis. Based on this description, a new error detection model for the RTSJ was proposed. This new model includes the modification of the current RTSJ model for enforcing the temporal execution time and introduces a blocking time monitoring model.
3.5 Summary

The proposed detection model for the RTSJ is presented in details in Chapter 6.

Moreover, with respect to the RTSJ, this chapter included notes about the behaviour of real-time threads and the priority inheritance algorithm specified in the RTSJ. The behaviour of real-time threads has been recently proposed and, so far in the literature, has not been implemented in mainstream run time environments. Since this aspect of the RTSJ is important to the correct execution of the error detection mechanisms, it is formally specified and analysed (using model checking techniques) in Chapter 5.
This chapter discusses implementations for the proposed timing error detection models, in terms of the POSIX/C standard. The contributions of the chapter are twofold. Firstly, it describes in more details the mechanisms provided by the standard that can be used for implementing error detection for:

- deadline miss: using real-time clocks and timers that are released periodically at the time of a deadline miss;

- execution overrun of the thread’s WCET: one of the capabilities of the POSIX/C standard is the definition of execution-time clocks and timers, which enables to account only for the time a thread uses the processor. By setting an execution-time timer to the WCET of a thread, it is possible to detect when the thread overruns its assumed WCET;

- violation of the thread’s MIT assumptions: with the use of real-time clocks, it is
possible to detect when MIT’s are being violated. Moreover, the standard provides the implementation of a Sporadic Server (SS) for providing support (enforcement in this case) for aperiodic and sporadic threads.

Secondly, it proposes the abstraction of blocking-time clocks and timers, which can be used more effectively for detecting the following timing error:

• execution overrun of a thread’s WCET inside a shared resource (also called blocking time violation for a thread): a mechanism, called blocking-time clocks and timers, is proposed to complement the POSIX/C standard for the detection of timing error. Blocking-time clocks and timers are used to account for the time a higher priority thread is blocked by lower priority threads, when using the priority inheritance algorithms found in POSIX/C.

Figure 4.1, previously presented in Section 3.3.1, summarises the POSIX/C mechanisms that can be used to detect timing errors. This figure also depicts other components of the system that are assumed for implementing the correct functionality of the model. Basically, this includes the use of a uniprocessor architecture, FPS and priority inheritance algorithms.

The structure of this chapter is as follows. Section 4.1 describes an implementation of deadline miss detection in POSIX/C. In Section 4.2 the use of execution-time clocks and timers for detecting execution time overruns is discussed. The detection of MIT violations and the support for sporadic and aperiodic computation units in POSIX/C are presented in Section 4.3. The research contribution for complementing POSIX/C with a detection model for the execution overrun inside a shared resource is proposed in Section 4.4.
4.1 Detection of Deadline Misses

Some features of the POSIX/C standard were presented in Section 3.3, including an example code that illustrated the implementation of a periodic thread. This section builds on that code, showing how to implement the detection of deadline misses. An abridged version of the application code, which focuses on the detection of deadline miss for a periodic thread is presented in Figure 4.2.

Basically, the deadline miss detection model for a periodic thread is implemented using a real-time timer (created using the `timer_create` function). The timer is set to the value of the deadline time, using the `timer_settime` function, before the thread starts its current release. Just before starting its execution, the volatile variable `pr->executing_state` is set to true (value 1), meaning that the thread is currently executing. This variable is set to false (value 0) when the thread finishes its current computation and is about to sleep until its next release, which means the thread is not executing. If the timer expires during the execution of the periodic thread a pre-defined signal is generated. This particular signal is defined inside the variable `pr->event` that also allows for data to be passed within the signal. When handling the signal, the system can be configured to identify which thread...
4.1 Detection of Deadline Misses

```c
void *periodic_thread (void *args) {
    struct timespec next_release, remaining_time;
    struct itimerspec deadline_time;
    timer_t deadline_timer;
    struct release_parameters *pr;
    pr = (struct release_parameters *)args;
    /* Create deadline detection timer */
    timer_create (CLOCK_REALTIME, &pr->event, &deadline_timer);
    next_release = pr->start;
    /* Set deadline timer */
    add_timespec(&deadline_time.it_value, &next_release, &pr->deadline);
    timer_settime (deadline_timer, TIMER_ABSTIME, &deadline_time, NULL);
    /* Wait until start time */
    clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME, &next_release, &remaining_time);
    pr->executing_state = 1;
    int i = 0;
    while(1) {
        /* Thread does computation */
        /* Set executing state to false */
        pr->executing_state = 0;
        add_timespec(&next_release, &next_release, &pr->period);
        /* Set deadline timer */
        add_timespec(&deadline_time.it_value, &next_release, &pr->deadline);
        timer_settime (deadline_timer, TIMER_ABSTIME, &deadline_time, NULL);
        /* Wait until next period */
        clock_nanosleep (CLOCK_REALTIME, TIMER_ABSTIME, &next_release, &remaining_time);
        pr->executing_state = 1;
    }
}
```

Figure 4.2: Detection of deadline miss for a periodic thread model.

The signal was related to.

The particular code for handling this signal is not shown here, but it can either use a specific thread that waits for incoming signals (using the `sigwait` function) or establish a specific signal handler to be executed upon the generated signal (using the `sigaction` function). Therefore, when handling this pre-defined signal, the first step is to check if the thread is running or not, which is specified in the `pr->executing_state` variable. It could happen that after waking up, due to a very overloaded system state, the thread is not able to set the `pr->executing_state` variable to 1 (a race condition occurs). In
this scenario, the race condition can be avoided if the system ignores the state of `pr->executing_state` when a deadline miss occurs. With the possibility of passing data within the signal event structure, the variable `pr->executing_state` can be directly associated to the thread, generating a handling code for this particular situation and thread. Moreover, because the variable `pr->executing_state` is changed by the thread and read by the handling routine, it is important to declare it as a volatile type. This means that the compiler will make sure that the system reads the contents of `pr->executing_state` set by the thread. Therefore, if the thread is running, a deadline miss has occurred and further action can be taken. Otherwise, the signal generated by the timer is ignored – if the deadline timer for the new release has not been set in time.

4.2 Detection of Execution Time Overruns

In order to present a possible detection mechanism for the execution overrun of a particular thread, the example of a periodic thread (shown in Section 3.3) is also extended. POSIX/C provides execution-time clocks and timers, which facilitates the implementation of a detection model for execution time overruns. In Figure 4.3 an abridged version of the application code, which only displays the scope of the periodic thread, is presented.

The execution time overrun detection for a periodic thread is implemented by initially creating an execution-time timer (using the `timer_create` function, with the `CLOCK_THREAD_CPUTIME_ID` option). This timer is then immediately set to the value of the allowed execution time for the thread, using the `timer_settime` function. Differently from the deadline miss detection model presented in the previous section, when detecting an execution overrun there is no need to have a variable to identify if the thread is currently executing – because the clock only increases when the thread is executing the processor.

\begin{footnote}
\textsuperscript{1}This constant specifies that the execution-time clock of the current executing thread is to be used.
\end{footnote}
4.3 Detection of MIT Violations

The MIT specifies the minimum time taken between two consecutive release events for a specific sporadic thread. Even though system designers tend to provide a worst-case bound for the MIT of a sporadic thread, it may be possible that, during run time, the system might get more release events than expected. In this case, violations of the MIT start occurring. In order to detect these violations, the system has to monitor the time taken between two consecutive release events. Figure 4.4 presents an abridged application

Figure 4.3: Detection of execution time overrun for a periodic thread model.

If the timer expires during the execution of the periodic thread, a pre-defined signal is generated. The signal can be handled using either the `sigwait` or the `sigaction` functions. Moreover, by passing relevant information about the thread to which the signal is generated to (variable `pr->event`), default actions upon the execution time overrun for the particular thread can be taken.

### 4.3 Detection of MIT Violations

The MIT specifies the minimum time taken between two consecutive release events for a specific sporadic thread. Even though system designers tend to provide a worst-case bound for the MIT of a sporadic thread, it may be possible that, during run time, the system might get more release events than expected. In this case, violations of the MIT start occurring. In order to detect these violations, the system has to monitor the time taken between two consecutive release events. Figure 4.4 presents an abridged application
code that demonstrates how the system can monitor and detect MIT violations.

```c
void *sporadic_aperiodic_release_handler (void *args) {
    struct release_parameters *pr;
    struct timespec next_release, now, remaining_time;
    pr = (struct release_parameters *)args;
    sigset_t set, set_wait;
    int received_sig;

    /* Block all signals */
    sigfillset (&set);
    pthread_sigmask (SIG_BLOCK, &set, NULL);
    sigemptyset (&set_wait);
    sigaddset (&set_wait, SPORADIC_APERIODIC_RELEASE);

    /* Wait until start time */
    next_release = pr->start;
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next_release, &remaining_time);

    /* Wait for signals to come */
    while (1) {
        sigwait (&set_wait, &received_sig);
        /* Handle the occurrence of a sporadic or aperiodic release */
        if (received_sig == SPORADIC_APERIODIC_RELEASE) {
            /* Get the current time */
            clock_gettime(CLOCK_REALTIME, &now);
            if (smaller_or_equal_timespec(&next_release,&now) == 1) {
                /* Calculate the next MIT */
                add_timespec(&next_release, &now, &pr->mit);
                /* Release the sporadic/aperiodic thread */
            } else {
                /* There has been an MIT violation, take further actions here */
            }
        } else {
            /* Other signals */
        }
    }
}
```

Figure 4.4: Detection of MIT violations in POSIX/C.

In the code of Figure 4.4, the release events are generated by the system through signals. The generated signals are then obtained by a thread responsible for handling release events, using the `sigwait` function in this case. In order to make sure that this code is not preempted by the system, the thread should execute at a very high priority. For detecting the violation of the MIT, the release event handling thread adopts the following
4.3 Detection of MIT Violations

behaviour. The variable next_release (a structure of type timespec) holds information about the next acceptable release by the system, which guarantees the MIT. Once a new release event arrives, it gets the current time in the system (using the clock_gettime function) and checks if the current time is greater or equal to the next acceptable release – this is done using the smaller_or_equal_timespec function. If this condition is true, the release event is accepted and the sporadic or aperiodic thread is released. More information about the support of POSIX/C to execute sporadic and aperiodic threads is presented in the next section. Otherwise, an MIT violation occurs and the system can adopt further actions upon it.

4.3.1 Support for Aperiodic and Sporadic Computation Units

Complementary to the detection of violations to the MIT assumptions, POSIX/C provides support for executing sporadic and aperiodic threads using a sporadic server. As already described in Chapter 2, when using a server the developer can limit the amount of time (budget) dedicated to a sporadic or aperiodic thread in the system in a given period of time. The support for sporadic servers in POSIX/C is provided by threads. Indeed, a sporadic server thread is configured using different scheduling parameters, which are defined in the sched_param structure and set with the pthread_setschedparam function. The attributes that compose the sched_param structure are shown in Figure 4.5. Relevant attributes in the sched_param structure for setting a sporadic server include: policy, which specifies the use of the appropriate scheduling policy, the SCHED_SPORADIC constant in this case; sched_priority, specifies the normal priority of the server when it is running within its budget; sched_ss_low_priority, defines the lower priority of the sporadic server when the budget exhausts (instead of stopping, the server can still execute, but with a background priority); ss_repl, used to specify the replenishment period; ss_budget, states the budget of the sporadic server; sched_ss_max_repl, which is used to define the maximum
number of pending replenishment operations – this aims to bound the amount of system overhead that is needed to implement the sporadic server in POSIX/C. Because several replenishment operations may be pending at the same time, each of which will be serviced at its respective scheduled time. However, they must not exceed the maximum allowed number of pending replenishments set by `sched_max_repl`. Moreover, if the scheduled time for replenishment is before the current time, the replenishment operation occurs straight away.

```c
#include <sched.h>

struct sched_param {
    int policy;
    int sched_priority;
    int sched_ss_low_priority;
    struct timespec sched_ss_repl_period;
    struct timespec sched_ss_init_budget;
    int sched_ss_max_repl;
};
```

Figure 4.5: Structure `sched_param`, used to set-up the POSIX/C sporadic server.

### 4.4 Detection of Blocking Time Violations

The main contribution of this chapter to the overall thesis is described in this section. This section defines a monitoring mechanism that can be used to detect the violation of the blocking time for threads that are waiting to gain access to a specific shared resource. In order to define such a mechanism, an algorithm for identifying the blocking time during run time is provided in the next section. Moreover, an example of using blocking-time clocks and timers is presented.

**Notation for the computational model:** Before describing the method used to identify the blocking time for a thread and the monitoring algorithm based on it, a notation for the computational model is needed. Let \( T \) be a thread in the system. \( T \) is composed of six different elements: \( Ap \) (the thread’s current active priority), \( Bp \) (the thread’s current base
priority), $Blk\_Time\_Max$ (the worst-case blocking time that the thread can suffer, according to its feasibility analysis), $Blk\_Time\_Start$ (the last time at which the thread started being blocked by a lower priority thread), $Blk\_Time\_Total$ (the total time the thread has been blocked), and $Blk\_Timer$ (provides a timer\(^2\) that signals when the $Blk\_Time\_Max$ has been reached). The current thread in the system assigned to execute the processor is denoted by $T_c$, whilst the previous thread that was executing the processor, when a context switch occurred, is denoted by $T_p$. Moreover, two system variables $Ap_{previous}$ and $Bp_{previous}$ are defined. They denote, respectively, the active and base priorities for the previous thread that was assigned to the processor. These variables provide a snapshot for the previous state of the thread’s priorities, which could have changed.

Most importantly, it is assumed that the system’s run-queue holds information about both threads that have been released (currently ready to execute) as well as threads that are currently blocked (waiting for a specific shared resource to resume execution). With this assumption, it is easier to provide the Theorems used to identify the blocking time of threads during run time. Though, the work in this thesis recognises that most operating systems do not provide such run-queue. In fact, in Chapter 7, this assumption is removed in order to implement the algorithm. Threads that have finished their previous execution but have not been released yet are not considered to be in the system’s run-queue. Moreover, another queue containing all the threads that have self-suspended in the system is available. This is needed for the implementation of the monitoring algorithm in the special case where threads can self-suspend whilst holding a shared resource.

\(^2\)The discussion regarding the implementation of timers in the system is postponed to Chapter 7. At this point it is only important to know that the timers are set to a specific moment in time and they signal the system when this moment has been reached.
4.4.1 Identification of Blocking Time during Run Time

For the purpose of calculating the blocking time for a thread, during the run-time execution of the system, two basic steps have to be followed. Firstly, there is the need to identify the potential condition that may cause higher priority threads to become blocked by lower priority threads. Secondly, it is necessary to provide a criterion to identify the threads that are being blocked by the current thread. These two steps are captured in the following Theorem 4.1.

**Theorem 4.1.** If the current thread assigned to the processor \( T_c \) has its active priority greater than its base priority, i.e., \( T_c.Ap > T_c.Bp \), a priority range \([ T_c.Bp + 1 , T_c.Ap] \) (with respect to the system’s run-queue) is formed – note that the range of priorities forms a closed set that starts with the threads in the priority run-queue \( T_c.Bp + 1 \) (as the threads in the priority run-queue \( T_c.Bp \) are not blocked) up to, and including, \( T_c.Ap \). A thread \( T_b \) is blocked by thread \( T_c \) iff the base priority of thread \( T_b \) is in the priority range \([ T_c.Bp + 1 , T_c.Ap] \) (formed by \( T_c \)) of the system’s run-queue.

**Proof.** This Theorem is divided in two parts, the: (a) definition of a range of priorities; (b) identification of blocked threads in this range. Consider the first part of the Theorem. Initially, the active priority for any thread \( T \) in the system is equal to its base priority \( (T.Ap = T.Bp) \). The only case where the active priority of the thread becomes different (and always greater) than the base priority occurs when the active priority of the thread is boosted, because it is executing inside a shared resource, due to: (i) an inherited priority (when using the PI protocol); (ii) the setting of a ceiling priority (when using the PP protocol). Firstly, consider the PI protocol. In this case, if the thread inside the shared resource has its active priority boosted, this means the thread has inherited the priority of one or more already blocked threads, which are waiting to enter in the shared resource. Secondly, consider the PP protocol. When the thread enters in a shared resource, its
active priority is set to the ceiling of that shared resource. In this case, the active priority for the thread becomes greater than its base priority. The only exception occurs when the ceiling is equal to the thread’s base priority. But in this case, the thread that entered in the shared resource is not blocking other higher priority threads, since it has the highest possible priority for executing inside that specific shared resource. According to both protocols PI and PP, the boosted active priority \(T_c.Ap > T_c.Bp\) for the current thread \(T_c\) executing inside the shared resource defines a range of priority. Therefore, threads waiting for execution in the priority range \([T_c.Bp + 1, T_c.Ap]\) of the system’s run-queue, will not be allowed to do so, as we assume priority queues are ordered by FIFO.

This leads to the second part of the Theorem, where Definition 2.1 of the priority inversion problem is followed. If a thread \(T_b\) has its base priority in the priority range \([T_c.Bp + 1, T_c.Ap]\) of the system’s run-queue\(^3\), defined by the current thread \(T_c\) running in the system, this necessarily means that thread \(T_b\) has a higher base priority than the current thread \((T_b.Bp > T_c.Bp)\). Therefore, thread \(T_b\) is being blocked by thread \(T_c\).

Using Theorem 4.1, it is possible to derive an algorithm to calculate, during run time, the blocking time for each of the threads of the system that are being blocked by the current low-priority thread running in the system. Unfortunately, Theorem 4.1 does not tackle the case where a low-priority thread, which is already blocking one or more higher priority threads, self-suspends (e.g., calls a sleep function) inside a shared resource.

In order illustrate why the previous Theorem does not work for the special case where threads can self-suspend inside a shared resource, consider the scenario depicted in Figure 4.6. In this scenario, a shared resource \(L_0\) (using the PI protocol) is shared between a high-priority thread \(T_0\) and a low-priority thread \(T_2\). Besides, another medium-priority thread \(T_1\) can execute in the system. According to the scenario, thread \(T_2\) starts and locks

\(^3\)As defined previously, the system’s run-queue includes threads that have been released and are currently ready to execute as well as threads that are currently blocked.
resource $L_0$. Thread $T_2$ is preempted by $T_1$, which is eventually preempted by thread $T_0$. When $T_0$ tries to lock $L_0$, it increases the priority of thread $T_2$, which is currently using the resource $L_0$.

![Diagram showing thread interactions](image)

Figure 4.6: Problem with thread self-suspending inside a shared resource.

Then, thread $T_2$ sleeps inside the shared resource $L_0$, letting thread $T_1$ execute in the system. This is the event of interest, which shows that the previous Theorem is not capable of monitoring threads that can self-suspend inside their shared resource. Note that, the first condition (where the active priority of the current thread is greater than its base priority) of the previous Theorem is not satisfied when thread $T_2$ sleeps in the shared resource. This happens because thread $T_1$ has its active priority equal to its base priority. Therefore, the blocking time for thread $T_0$ is not being monitored during the sleep time of $T_2$. However, this time should be accounted for the blocking time of thread $T_0$ since, following the priority inversion problem, thread $T_1$ (currently running the processor) has a base priority lower than thread $T_0$. In order to deal with this special case, the following Theorem 4.2 is specified.

**Theorem 4.2.** Consider that the current thread assigned to the processor is blocking
4.4 Detection of Blocking Time Violations

other threads in the system, i.e., it has its active priority greater than its base priority (according to the previous Theorem 4.1). Once this thread self-suspends itself inside a shared resource, it gives up on the processor (becomes the previous thread $T_p$), and a new thread $T_c$ is assigned to the processor. When this case occurs in the system, the possible range of priorities for threads being blocked by $T_c$ has to be extended to consider the priority range $[T_c.Ap + 1, T_p.Ap]$ (with respect to the system’s run-queue).

**Proof.** When the thread $T_p$ self-suspends, the new thread $T_c$ selected to execute the processor is either (a) blocking other threads (where $T_c.Ap > T_c.Bp$) or (b) not (with $T_c.Ap = T_c.Bp$). In the first case (a), the new thread defines a range of priorities for blocked threads (following Theorem 4.1). But a thread $T$ inside the priority range of the previous thread $T_p$ ($[T_p.Bp + 1, T_p.Ap]$), which self-suspended, may have the base priority (i) lower or (ii) greater than the base priority of the current thread $T_c$. In (i), the thread is not considered to be blocked by $T_c$ but, instead, has added interference (since $T_c.Bp < T.Bp$). However, in (ii) the thread is considered to be blocked by $T_c$ (since $T_c.Bp > T.Bp$). Nevertheless, the current range of priorities for $T_c$ may not include this thread, since the active priority of the previous thread $T_p$ may be greater than the active priority of $T_c$ ($T_p.Ap > T_c.Ap$). For dealing with (ii), the possible range of blocked threads for $T_c$ has to be extended to also deal with the higher priorities (when $T_p.Ap > T_c.Ap$) defined by the previous thread $T_p$. In other words, a new priority range $[T_c.Ap + 1, T_p.Ap]$ (with respect to the system’s run-queue) has to be monitored. Case (b) is a subset of (a), in the sense that the only range of priorities that has to be monitored is $[T_c.Ap+1, T_p.Ap]$, as the current thread $T_c$ is not blocking any other thread in the system ($T_c.Ap = T_c.Bp$).

In order to provide an algorithm for calculating the blocking time of threads in the system, three basic steps are followed. Firstly, the question of *when*, in the scheduling process
of the system, the algorithm for calculating the blocking time should be called has to be defined. This is shown in the requirements for the algorithm, depicted in Algorithm 4.1. For the definition of the algorithm, the computational model notation specified previously is used. The requirements for the algorithm start with the global variables necessary for the execution of the algorithm. Then, the next requirement specifies when the algorithm should be executed in the system, defining the first step of the algorithm. It defines that the algorithm should be called (i) every time a thread is added to the run-queue of the system and (ii) when a context switch occurs (a new thread is assigned to the processor). According to the assumption regarding the computational model, the run-queue of the system is composed by both threads that are ready to run and also blocked threads. This way, it is possible to guarantee that by looking to the run-queue, all possible threads that could be potentially blocked by the current thread in the system are identified.

Algorithm 4.1 Requirements for calculating the blocking time.

Require:
- The following system global variables:
  - \textit{Current\_Time} \{provides the current absolute time of the system\}
  - \textit{T} \{temporally holds the reference to a given thread in the system\}
  - \textit{Tc} \{contains the reference to the thread scheduled to the processor\}
  - \textit{BP\textsubscript{previous}} \{base priority for the previous thread running in the system\}
  - \textit{AP\textsubscript{previous}} \{active priority for the previous thread running in the system\}
  - \textit{AP\textsubscript{suspended}} \{maximum active priority for the threads in the suspended queue that have the base priority greater than the active priority\}
  - \textit{AP\textsubscript{previous\_suspended}} \{maximum active priority for the threads in the suspended queue that have the base priority greater, in the previous context switch, than the active priority\}
- When to call the algorithm by the run time system:
  - Every time a thread is added to the system’s run-queue
  - When a new thread is assigned to the processor, i.e., a context switch occurs

Secondly, it is necessary to specify where the system should start and stop monitoring the blocking time for the threads. And the third step is to identify what to monitor in the system. These two steps follow Theorems 4.1 and 4.2 provided previously, where
the threads that are being blocked in the system in a given range of priorities are identified. More specifically, Algorithm 4.2 depicts the algorithm based on Theorem 4.1 and Algorithm 4.3 implements the ideas of Theorem 4.2.

Algorithm 4.2 Algorithm for calculating the blocking time (part 1).

1: if $Ap_{(previous)} > Bp_{(previous)}$ then
2:   for $I$ in the priority range $[Bp_{(previous)} + 1, Ap_{(previous)}]$ do
3:     $T \leftarrow GetHeadOfPriorityQueue(I)$
4:     while $T \neq null$ do
5:         if $T.Bp > Bp_{(previous)}$ then
6:             $T.Blk.Time_Total \leftarrow T.Blk.Time_Total + (Current\_Time - T.Blk.Time_Start)$
7:             \{Stop blocking timer for this thread here\}
8:         end if
9:     end while
10:   end for
11: end if
12: if $T_c.Ap > T_c.Bp$ then
14:     $T \leftarrow GetHeadOfPriorityQueue(I)$
15:     while $T \neq null$ do
16:         if $T.Bp > T_c.Bp$ then
17:             $T.Blk.Time_Start \leftarrow Current\_Time$
18:             \{Set blocking timer for this thread\}
19:         end if
20:     end while
21:   end for
22: end if
23: \{Set global variables for the next execution of the algorithm\}
24: $Bp_{(previous)} \leftarrow T_c.Bp$

The code in Algorithm 4.2 is now explained. At lines 1 and 12 the stop and start conditions, respectively, for monitoring the blocking time are set. This defines the second step of the algorithm, i.e., where to start and stop monitoring. The idea of these conditions
is to check if the active priority of the thread is greater than its base priority. As shown in line 1, the active priority \((Ap_{\text{previous}})\) and the base priority \((Bp_{\text{previous}})\) for the previous thread that was executing the processor are compared. Similarly, line 12 checks if the current thread assigned to the processor has its active priority \((T_c.Ap)\) greater than its base priority \((T_c.Bp)\). It should be noted that, when the system starts its execution, the initial value of both \(Ap_{\text{previous}}\) and \(Bp_{\text{previous}}\) are zero. Moreover, instead of checking the start condition for the algorithm first, the stop condition is evaluated. This is devised to guarantee that the blocking time monitoring is never started more than once, which could happen if the start condition was evaluated before the stop condition. If any of these stop and start conditions are evaluated to true, the third and last step of the algorithm is triggered: checking what is the possible range of priorities where threads might be being blocked by the current thread.

From lines 1 to 11 the stop condition for the blocking time monitoring is shown. Line 2 checks the range of priorities where a thread might be being blocked. If a thread is found to be blocked in the priority range of the previous thread executing the processor (line 5), the \(Blk_{\text{Time}}_{\text{Total}}\) is set for that thread to be the total blocking time so far \((Blk_{\text{Time}}_{\text{Total}})\) plus the system’s current time \((Current_{\text{Time}})\) minus the time that this thread started being blocked \((Blk_{\text{Time}}_{\text{Start}})\). Also, the blocking timer, which signals the violation of the blocking time, is stopped. From lines 12 to 23 the priority range for the current thread that is assigned to the processor (line 13) is checked. If a thread in this range is being blocked by the current thread (line 16), the \(Blk_{\text{Time}}_{\text{Start}}\) is set for that thread to be the system’s current time \((Current_{\text{Time}})\). Moreover, the timer is set at line 18 for signalling any blocking time overrun for that particular thread. The \(Blk_{\text{Time}}\) for the thread is set to the \(Current_{\text{Time}}\) plus the maximum allowed blocking time for the thread \((Blk_{\text{Time}}_{\text{Max}})\) minus the total blocking time used so far by the thread \((Blk_{\text{Time}}_{\text{Total}})\).
Algorithm 4.3 has the same structure as Algorithm 4.2 just depicted. However, it focuses on implementing the ideas of Theorem 4.2 by extending the range of priorities for blocked threads in the system when threads are in a self-suspended state.

**Algorithm 4.3** Algorithm for calculating the blocking time (part 2).

1: if $A_{p\text{(previous\_suspended)}} > A_{p\text{(previous)}}$ then
2: for $I$ in the priority range $[A_{p\text{(previous)}} + 1, A_{p\text{(previous\_suspended)}}]$ do
3: $T \leftarrow \text{GetHeadOfPriorityQueue}(I)$
4: while $T \neq \text{null}$ do
5: if $T.Bp > B_{p\text{(previous)}}$ then
6: $T.\text{Blk\_Time\_Total} \leftarrow T.\text{Blk\_Time\_Total} + (\text{Current\_Time} - T.\text{Blk\_Time\_Start})$
7: \{Stop blocking timer for this thread here\}
8: $T \leftarrow \text{GetNextThreadInPriorityQueue}(I)$
9: end while
10: end for
11: end if
12: $A_{p\text{(suspended)}} \leftarrow \text{GetHighestActivePriorityInSuspended}()$
13: if $A_{p\text{(suspended)}} > T_c.Ap$ then
14: for $I$ in the priority range $[T_c.Ap + 1, A_{p\text{(suspended)}}]$ do
15: $T \leftarrow \text{GetHeadOfPriorityQueue}(I)$
16: while $T \neq \text{null}$ do
17: if $T.Bp > T_c.Bp$ then
18: $T.\text{Blk\_Time\_Start} \leftarrow \text{Current\_Time}$
19: \{Set blocking timer for this thread\}
20: $T.\text{Blk\_Timer} \leftarrow \text{Current\_Time} + (T.\text{Blk\_Time\_Max} - T.\text{Blk\_Time\_Total})$
21: end if
22: $T \leftarrow \text{GetNextThreadInPriorityQueue}(I)$
23: end while
24: end for
25: end if

\{Set global variables for the next execution of the algorithm\}

The algorithm described in this section provides a mechanism to identify higher priority threads that are being blocked, during the run-time execution of the system, by lower priority threads and calculate the total blocking time for those threads. It also provides
a mechanism for signalling, by setting timers, when the maximum blocking time for a specific thread has been reached. This way, timers can be set to the maximum blocking time assigned for the specific thread when the start condition of the blocking time becomes true. In case an overrun occurs, the system will be signalled by this blocking timer. In other words, the signal is sent when the blocking time measurement is reached. Otherwise, the system continues working as usual and the stop condition of the algorithm is used to stop timers that have been previously set.

4.4.2 Detecting Violations

For presenting a possible detection mechanism for the blocking time overrun of a particular thread, the example of a periodic thread (shown in Section 3.3) is now extended. With the implementation of the algorithm described previously, it is possible to use blocking-time clocks and timers, which facilitates the implementation of a detection model for blocking time overruns. The description of the implementation of the algorithm is presented in Chapter 7. In Figure 4.7 an abridged version of the application code, which only displays the scope of the periodic thread, is presented. This code follows the same idea of execution time overrun detection shown in Section 4.2, only focusing on the blocking time feature of the thread.

The blocking time overrun detection for a periodic thread is implemented by initially creating a blocking-time timer (using the `timer_create` function, with the `CLOCK_THREAD_BLKTIME_ID` option). This timer is then immediately set to the value of the allowed blocking time for the thread, using the `timer_settime` function. If the timer expires during the execution of the periodic thread, a pre-defined signal is generated. In particular, the code for handling this signal is not shown here, but it can either use a specific thread that waits for incoming signals (using the `sigwait` function) or establish a specific signal

---

4 This constant defines that the blocking-time clock for the current thread is to be used.
void *periodic_thread (void *args) {
    struct timespec next_release, remaining_time;
    timer_t blocking_timer;
    struct release_parameters *pr;
    pr = (struct release_parameters *)args;
    /* Create and set blocking-time detection timer */
    timer_create (CLOCK_THREAD_BLKTIME_ID, &pr->event, &blocking_timer);
    timer_settime (blocking_timer, 0, &pr->blocking, NULL);
    /* Wait until start time */
    next_release = pr->start;
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next_release, &remaining_time);
    while(1) {
        /* Thread does computation and enters in shared resources */
        add_timespec(&next_release, &next_release, &pr->period);
        /* Set blocking-time timer */
        timer_settime (blocking_timer, 0, &pr->blocking, NULL);
        /* Wait until next period */
        clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next_release, &remaining_time);
    }
}

Figure 4.7: Detection of blocking time overrun for a periodic thread model.

handler to be executed upon the generated signal (using the sigaction function). Once the signal has been received, the default action upon the execution time overrun can be taken. Again, similar to the other handlers, the variable pr->event can have added information that tells the handler which thread the signal was related to.

4.5 Summary

According to the detection model for POSIX/C specified in this research, four different types of timing errors could be generated in the system. This chapter discussed the POSIX/C support for detecting each one of the timing errors considered, including an example of their use.

Firstly, the detection of deadline miss was presented, where regular POSIX/C real-time timers can be used for implementing the detection model. Secondly, the detection of execution time overruns for a thread was discussed. The detection mechanism presented
focused on the use of execution-time clocks and timers. Thirdly, the possibility of having the MIT constraints, for a given thread, being violated was considered. The detection mechanism presented for such timing errors was based on the use of real-time monotonic timers. Also, the support for sporadic and aperiodic threads provided by POSIX/C in terms of a Sporadic Server was shown.

Moreover, the last error detection mechanism considered in this chapter was related to the execution overrun of the WCET of a thread inside a shared resource. In such aspect, this thesis contends that POSIX/C does not provide adequate support; therefore, a mechanism for monitoring the time a higher priority is blocked by a lower priority one was discussed. Such a mechanism was defined in terms of blocking-time clocks and timers, providing the basis for implementing a detection model for such type of timing error. Indeed, it was shown how to use blocking-time clocks and timers to detect the violation of blocking time for a higher priority thread.
The RTSJ computational model is a complex one, with many interactions between its features. Some of its features have not been implemented and, consequently, it is difficult to have confidence in their correctness. A major feature not widely explored by implementation (nor analysis) is the cost monitoring and enforcement model. This is a very important topic to this thesis. In order to explore the cost monitoring model, some formal descriptions are required for the behaviour of real-time threads defined in the RTSJ. This chapter focuses on presenting the behaviour of real-time threads in the RTSJ, building a formal model and analysing its consistency.

5.1 Real-Time Threads

The RTSJ model for defining real-time threads with aperiodic and sporadic release parameters, as well as periodic release parameters, has been proposed in [37]. The key idea
behind the approach is to extend the current behaviour specified for periodic real-time threads. According to the proposed behaviour for aperiodic and sporadic release parameters in the RTSJ, the following methods are incorporated to the `RealTimeThread` class [37]:

- `waitForNextRelease()` (and `waitForNextReleaseInterruptible()`): indicates the finish of a release and is subject to the semantics defined for aperiodic and sporadic release control in the RTSJ [18] (see Section 5.4);
- `release()`: signals that the associated real-time thread should be released;
- `deschedule()` and `schedule()`: respectively, deschedules and schedules a thread using the same semantics of the methods `schedulePeriodic()` and `deschedulePeriodic()` defined for periodic threads in the RTSJ [18].

By incorporating these methods using the semantics described, it becomes possible to use the deadline monitoring and cost monitoring mechanisms of the RTSJ for both aperiodic and sporadic threads. In Figure 5.1 the requirements and definitions for the behaviour of periodic, sporadic and aperiodic real-time threads are presented. The behaviour for real-time threads acts as if the following variables were added to the thread model: boolean `descheduled`, integer `pendingReleases`, integer `missCount`, and boolean `lastReturn`.

As presented in Figure 5.1, releases are affected by the operations defined in (a). After the execution of the initial release, the next releases are characterized by the generation of arrivals in the case of sporadic and aperiodic parameters (b1), or release events when periods are due for periodic parameters (b2). In the case of sporadic and aperiodic threads, for each release event that is accepted, an entry is made in the thread’s arrival-time queue. The deadline of a real-time thread is relative to the release event and determined at the release event time, according to the value of the deadline contained in the real-time thread release parameters. This value does not change, except for threads with sporadic and
5.1 Real-Time Threads

(a) The release characteristics of real-time threads are defined by the: invocation of the `start()` method; invocation of the `waitForNextRelease()`, `schedule()`, and `deschedule()` methods; occurrence of deadline misses and whether or not a miss handler is available; (b1) calling of the `release()` method that generates non-periodic release events; (b2) the generation of periodic releases due to the passage of time;

(b1) Each call to the release method characterizes an arrival: if the real-time thread has aperiodic release parameters, then the arrival may become a release event for the real-time thread, according to the semantics defined for Aperiodic Release Control (see Section 5.4). The same occurs for a real-time thread with sporadic release parameters, but the semantics follow the behaviour for Sporadic Release Control (see Section 5.4);

(b2) If the real-time thread has `PeriodicParameters`, subsequent release events occur as each period falls due (except as described in (c)(5)): if the $i^{th}$ release event occurred at time $t_i$, then the $(i+1)^{th}$ release event occurs at time $t_i + T_i$, where $T_i$ is the value of the period specified by the `PeriodicParameters` class;

(c) The behaviour of a real-time thread is defined as below:
   (1) initially descheduled is false, pendingReleases is 0, missCount is 0, and lastReturn is true;
   (2) when the method `deschedule()` is invoked: descheduled is set to true;
   (3) when the method `schedule()` is invoked: descheduled is set to false. If the thread is blocked-for-release-event: pendingReleases is set to 0 and cost monitoring is reset for this thread;
   (4) when descheduled is true, the thread is said to be descheduled;
   (5) if the thread is descheduled and blocked-for-release-event, no release events can be received and no deadline misses can occur, until a call to the method `schedule()`. The descheduling of a thread has no effect on its initial release;
   (6) When each release event is due:
      (i) if the thread is blocked-for-release-event and descheduled: do nothing; else increment pendingReleases, inform cost monitoring that the next release event occurred, and make the thread eligible for execution;
      (ii) otherwise, increment pendingReleases, and inform cost monitoring that the next release event occurred;
   (7) On each deadline miss:
      (i) if a thread has deadline miss handler: deschedule is set to true, release the handler with its fireCount increased by missCount+1, and missCount is set to 0;
      (ii) otherwise, increment missCount;
   (8) When the method `waitForNextRelease()` is executed:
      (i) if missCount is greater than 0: decrement missCount if lastReturn is false and apply any pending parameter changes, decrement pendingReleases, inform cost-monitoring that thread has completed and return false; else, set lastReturn to false and return false;
      (ii) otherwise, apply any pending parameter changes, inform cost-monitoring that thread has completed and wait while descheduled is false or pendingReleases is 0. After that, lastReturn is set to true, decrement pendingReleases, and return true.

Figure 5.1: Semantics for real-time threads (based on [37]).

aperiodic release parameters using the REPLACE policy for MIT violation or arrival-time queue overflow. The policies used for sporadic and aperiodic release control are described in Section 5.4, where the formal model of sporadic and aperiodic real-time threads is specified.
The basic behaviour for real-time threads is explained in Figure 5.1 (c). In particular, on the invocation of the `schedule()` method (c)(3), the cost monitoring can be reset for the current thread being monitored. Depending on the current state of the real-time thread, different actions can occur on each release event (c)(6). On the case of deadline misses, miss handlers can be released (if available), otherwise a variable is used to count the number of deadline misses (c)(7). Finally, the behaviour of the real-time thread when a call to the `waitForNextRelease()` method is invoked is shown in (c)(8).

Moreover, the RTSJ also describes in the semantics of real-time threads, how the dynamic change of release parameters should take place. For instance, what happens to a periodic thread when it changes to a sporadic thread. This is described in Section 5.5, with formal models used to specify these aspects.

## 5.2 Modelling Architecture

In order to formally model the behaviour of real-time threads proposed by the RTSJ, a modelling architecture using the Timed Automata (TA) formalism of the UPPAAL tool is defined in this section (Figure 5.2). A brief introduction to the UPPAAL model checker can be found in Appendix A. The formal model of real-time threads is described in terms of two different models, one for Periodic Real-Time Threads (called PRTT subsystem, which is described in the next section) and one for Aperiodic/Sporadic Real-Time Threads (called ASRTT subsystem, which is described in Section 5.4). The change of release parameters is also specified in Section 5.5.

Because the focus is on the analysis of the specific behaviour of real-time threads, the model comprises the generic behaviour of a real-time thread. This way, the model does not include several threads running concurrently, but only a generic behaviour for the thread and possible interactions that can happen with this thread. The basic idea of the
architecture is to decompose the functionalities of the behaviour of real-time threads into three different levels (each level can be modelled with more than one automaton):

- **application level**: responsible for modelling the interactions of the application that can change the behaviour of the PRTT and ASRTT subsystems at the component level;

- **component level**: defines the main behaviour of the PRTT and ASRTT subsystems;

- **scheduler level**: represents the basic functions and interactions of the scheduler, substituting the need to model a real-time kernel. In both PRTT and ASRTT subsystems, it deals with the generation of release events and monitoring deadlines.

The automata defined at the application level are used to non-deterministically generate inputs to the component level (through synchronization channels). Non-deterministic inputs are inputs that could be generated at different times for different execution runs. In particular, the specification of this application level plays an important role in the architecture. They also model the interaction with any released event handlers. Therefore, all the possible combinations for these inputs during verification can be checked, taking the full advantage of using model checking as the analysis method.

The modelling architecture following these ideas is shown in Figure 5.2. Both PRTT and ASRTT subsystem have an internal global variable, called \( rtt_{cs} \), used to represent the current state of the PRTT or ASRTT subsystem. This variable is utilised in both component and scheduler levels, but is only set at the component level. It is also used to correctly synchronize the actions between architecture levels. The interactions represented by arrows in Figure 5.2 are specified through synchronization channels, whose names are depicted next to each arrow. The architecture also represents a possible change of release parameters, switching the execution of a PRTT subsystem to an ASRTT subsystem, and
vice-versa. This release parameter change model is detailed in Section 5.5. Note that, in the modelling architecture of Figure 5.2, the automata that compose each level of the architecture and the possible synchronization channels used between them are omitted.

![Figure 5.2: Modelling architecture for real-time threads.](image)

5.3 Formal Model of Periodic Real-Time Threads

In order to specify the PRTT subsystem, five automata are defined. Firstly, the possible values (constants in the UPPAAL tool) for the current state of the PRTT are presented. These values are set through the global variable $rtt\_cs$ and are as follows:

- **INIT**: the PRTT has not been selected for execution, only used to define an initial state;
• EXEC: the PRTT is eligible-for-execution or executing. There is no distinction between these states, because a run time model for a system, where many threads may be eligible-for-execution, is not specified;

• BFRE: the PRTT is blocked-for-release-event;

• BFS: the PRTT is blocked-for-schedule. This represents a state where the PRTT is blocked-for-release-event and also descheduled, waiting for a call to `schedule()` to become scheduled again;

• BLK: the PRTT is blocked for some other event.

In addition, in the next chapter, two other states are used in the extended models for interacting with the cost monitoring model of the RTSJ:

• BBCO: the PRTT is blocked-by-cost-overrun;

• PBBCO: a pending for blocked-by-cost-overrun has been set.

**Automata for the PRTT subsystem at the application level:** Specify the methods that can affect the behaviour of the PRTT. In Figure 5.3 (a) the automaton `RTT_SD` (RTT Schedule/Deschedule), representing calls (channels synchronizations) to `schedule()` (`sd!`) and `deschedule()` (`dd!`) methods, is presented. Figure 5.3 (b) shows the automaton `RTT_wFNR` (RTT waitForNextRelease) representing calls to the `waitForNextRelease()` method (`wfnr!`).

Figure 5.3: (a) RTT_SD and (b) RTT_wFNR automata.
• Automata for the PRTT subsystem at the component level: Specify the interface between the schedule/deschedule calls at the application level and the PRTT. They also define the behaviour of the PRTT. The interface is represented by the automaton RTT\_SDI (RTT Schedule/Deschedule Interface) depicted in Figure 5.4 (a). This automaton has two main transitions that are activated, respectively, by the schedule (sd?) and deschedule (dd?) of the PRTT triggered from the application level. Depending on the current state of the PRTT (variable rtt\_cs), it may trigger different actions in the RTT\_Main automaton, setting the variable d to true (when PRTT is descheduled) or false (when PRTT is scheduled).

![Diagram of RTT SDI and RTT Main automata](image)

Figure 5.4: (a) RTT\_SDI and (b) RTT\_Main automata.

Figure 5.4 (b) shows the automaton RTT\_Main, describing the main behaviour of the PRTT. The automaton is started by a new release event (execute?). When executing, it can non-deterministically change to the blocked state. Once in the blocked state, a
PRTT can become (non-deterministically) unblocked. The main behaviour of a PRTT is defined by the execution of the `waitForNextRelease()` method \((wfnr?)\) that, depending on the current value of its internal variables, triggers different transitions. If the PRTT has not missed a deadline \((mc == 0)\), and it is executing the current \(i^{th}\) release \((pr == 0)\) and is descheduled, it becomes blocked-for-schedule. If it is not descheduled, it becomes blocked-for-release-event, and also informs the scheduler level that it has completed its current release \((completed!)\). Once blocked-for-schedule, it can become blocked-for-release-event \((schedule?)\). If blocked-for-release-event, it can change to the state: (i) blocked-for-schedule \((deschedule?)\); (ii) executing \((execute?)\), decrementing the pending releases (variable \(pr\)) and setting the last return (variable \(lr\)) to true. Moreover, three other transitions can also happen when a call to the `waitForNextRelease()` method is executed. All these transitions end up in the executing state and they can inform the scheduler level \((completion!)\) or not (this means that the PRTT is trying to recover from a deadline miss).

- **Automaton for the PRTT subsystem at the scheduler level:** Defines the generation of releases and monitors deadlines for the PRTT. It is modelled using the automaton \(RTT_{PRD}\) (RTT Periodic Releases/Deadlines) shown in Figure 5.5. This automaton keeps track of the deadline and period for the PRTT, generating release events when periods are due. It starts the first release of the PRTT with the synchronization `execute!`. When the current time equals the deadline for the PRTT \((time == dl)\) it can (based on the state of the PRTT): increment the number of miss counts (variable \(mc\)); deschedule the PRTT \((d\) is set to true\) and reset the number of miss counts; or do nothing. Moreover, when a period is due \((time == p)\) it can (also depending on the current state of the PRTT): increment the pending releases; increment the pending releases and make the PRTT eligible-for-execution; or do nothing.

In order to guarantee that the model is analysable, it is necessary to ensure that
variables cannot have their values growing indefinitely. Thus, a guard is used to control the maximum number of pending releases. When the model reaches a maximum of NR pending releases, the transition \( S3 \rightarrow S5 \) takes place. Whenever this transition occurs, the whole model stops and no more transitions (except for \( S5 \rightarrow S5 \)) can occur. Using this strategy the number of pending releases becomes bounded, providing a well-defined behaviour once the maximum value for pending releases (NR) is reached. Even with this restriction, the model still generates all the possible interesting execution combinations (while pending releases is lower than NR) for the analysis.

### 5.4 Formal Model of Aperiodic and Sporadic Real-Time Threads

An event is responsible for the release of aperiodic and sporadic (non-periodic) real-time threads. This happens when a call to the `release()` method is issued by the system. The event is termed an *arrival* and the time it occurs is the *arrival time* [18]. According to the RTSJ, the base scheduler maintains a queue, called the arrival time queue, for each non-periodic real-time thread in the system. The arrival time queue maintains information about each of the arrivals and their times until the associated release completes [18]. The
5.4 Formal Model of Aperiodic and Sporadic Real-Time Threads

arrival time queue has an initial size (defined by the associated release parameter) that grows with time and can become full. Once full, an overflow policy can be set for the particular real-time thread.

Specifically, for aperiodic real-time threads four policies are defined [18]: (i) IGNORE, the arrival is not accepted (no release event occurs) and is ignored silently; (ii) EXCEPT, the arrival is not accepted (no release event occurs) but an ArrivalTimeQueueOverflowException is generated; (iii) REPLACE, if the completion associated with the last release event in the queue has not yet occurred and the deadline has not been missed, then the release event is replaced with the arrival time of the new arrival, otherwise it acts as the IGNORE policy; (iv) SAVE, the queue expands and accepts the new arrival (a release event occurs).

Different from aperiodic real-time threads, sporadic real-time threads have to maintain an MIT between releases, which characterizes the expected frequency of releases for the thread. When an arrival is accepted, the scheduler calculates the earliest time that a next arrival can be accepted by adding the current MIT to the arrival time of the accepted arrival [18]. In order to enforce this behaviour, two mechanisms are employed [18]: (i) arrival-time regulation, controls the work-load by letting only one release occurring per MIT – MIT violation policies IGNORE, EXCEPT, and REPLACE are supported (they work just like the policies described before for aperiodic real-time threads); (ii) execution-time regulation, occurs when the SAVE policy is in effect, in this case all arrivals are accepted but the restriction of only one release per MIT is enforced. Moreover, when changes are made to the MIT value of the sporadic thread, it only affects the next expected arrival time for release events that occur after the change [18].

In order to specify the model of aperiodic and sporadic real-time threads, the ASRTT subsystem is defined, according to the modelling architecture of Figure 5.2. The ASRTT subsystem uses the same automata defined for both application and component levels of
the PRTT subsystem. The model introduces an automaton to keep track of the time and, specifically, changes the scheduler level of the PRTT subsystem to deal with arrivals, instead of the passage of time, and handle the possible policies defined for the arrival queue. These automata are described.

- **Automaton for the ASRTT subsystem for tracking the global time:** In order to analyse the behaviour of threads with aperiodic and sporadic release parameters, the model has to cope with the notion of arrivals, by making it possible to add timestamps for the arrival times. Since it is not possible to obtain the current time from a clock and store it in an integer variable using the TA formalism, the TIME automaton (see Figure 5.6) is provided.

This automaton provides a (finite) global variable \(ct\) that is used by the scheduler level in order to acquire the current global time of the model. The constant MAXT is utilized to define the maximum global time that can be reached in the model. This way, when the MAXT value is reached the transition \(S1 \to S2\) takes place and the model stops executing, because only the transition \(S2 \to S2\) is possible in the model. Moreover, the bigger the value for MAXT, higher is the complexity of the model.

- **Automata for the ASRTT subsystem at the scheduler level:** Defines the generation of releases and monitors deadlines for the ASRTT component. It is modelled using three different automata, shown in Figure 5.7. Firstly, the automaton ASRTT_GA
5.4 Formal Model of Aperiodic and Sporadic Real-Time Threads

(ASRTT Generate Arrivals), presented in Figure 5.7 (a), is used to generate the possible arrivals for the ASRTT subsystem (arrival!). Secondly, the automaton ASRTT_GAI (ASRTT Generate Arrivals Interface), shown in Figure 5.7 (b), is defined to deal with the arrivals. It specifies how to handle the arrivals according to the policies IGNORE and REPLACE, in both conditions of arrival time queue overflow (for both aperiodic and sporadic threads) or MIT violation (for arrival time regulation in sporadic threads). The SAVE policy for the arrival time queue overflow and MIT violation (execution-time regulation in sporadic threads) is not modelled in this research work because it forces a dynamic expansion of the arrival time queue in order to accommodate new arrivals, this feature could render the model infinite and not verifiable.

Figure 5.7: (a) ASRTT_GA, (b) ASRTT_GAI and (c) ASRTT_ARB automata.
In the automaton ASRTT_GAI the variable MIT is used to specify the thread’s Minimum Interarrival Time. This variable is key to model an aperiodic (where MIT is set to zero) or sporadic (where MIT is greater than zero) thread, without having to define a whole new automaton. The ASRTT_GAI automaton, shown in Figure 5.7 (b), starts with an arrival from the environment if the current state of the thread is different from blocked-for-release-event (via channel arrival, transition S0 \rightarrow S1). The constant AQS is used to represent the maximum number of arrivals possible in the model and the variable next holds the allowed MIT for the next arrival. If the maximum number of arrivals allowed in the model has not been reached (aqs < AQS) and current arrival not violates the MIT (ct \geq next), the arrival is accepted (transition S1 \rightarrow S2 \rightarrow S3 \rightarrow S0). If the maximum number of arrivals is reached or the current arrival does violate the MIT, the regulation policies are used to deal with them.

According to the IGNORE policy, if any of the conditions mentioned before occur, the arrival is ignored (transition S1 \rightarrow S0). In the REPLACE policy, the arrival is also ignored (transition S1 \rightarrow S0) if the previous arrival in the arrival queue has already started (aqs == 0) or the last arrival in the arrival queue has already missed its deadline (aq[aqs - 1] + dl \geq ct). But it can be accepted (transition S1 \rightarrow S2 \rightarrow S3 \rightarrow S0) if the last arrival in the arrival queue has not missed its deadline (aq[aqs - 1] + dl < ct) – replacing the last arrival with the new one. Whenever a new arrival is accepted, a synchronisation to the channel accepted! is triggered. This brings the attention to the third automaton used to model aperiodic and sporadic threads.

The automaton ASRTT_ARD (ASRTT Arrival Releases/Deadlines), presented in Figure 5.7 (c), is modelled to generate releases and monitor deadlines for the ASRTT. The variable t in the automaton is a clock that is never reset and is used to monitor the deadline for the execution of the current arrival. It receives the accepted arrival (accepted?) and calculates the relative deadline for this release (variable ndl). Then, it starts the
5.5 Formal Model of Release Parameters Changes

The proposed model of real-time threads of the RTSJ specifies the possibility of dynamically changing the release parameter for a real-time thread. The idea behind this release parameter change is the adoption of a pseudo re-start of the thread [37], where the real-time thread looses its current state (even if it has releases in its queue or missed deadlines) and starts from scratch with its new release parameter. Of course, this only occurs when the thread finishes its current release, by calling the `waitForNextRelease()` method.

Four possible combinations of release parameter changes between the models of real-time threads are considered in this research. A change from periodic to either aperiodic
or sporadic, and vice-versa. Note that, when a sporadic real-time thread changes to an aperiodic parameter, it is simply changing the value of its MIT to zero and no changes to the arrival queue are carried out. Therefore, this model is not considered in the release parameter changes of the RTSJ and is not modelled here. In order to specify these release parameter changes, the models defined for periodic real-time threads (PRTT subsystem) and sporadic and aperiodic real-time threads (ASRTT subsystem) are extended. The automata for the application level (RTT_SD and RTT_WFNr) of both PRTT and ASRTT subsystems are used without changes. The automaton for the component level, which is the same for both PRTT and ASRTT subsystems, is modified. With respect to the scheduler level, the automata for both PRTT and ASRTT subsystems are extended according to the release parameter change they are modelling, this equates to two different models: (i) from the ASRTT subsystem to the PRTT subsystem; (ii) from the PRTT subsystem to the ASRTT subsystem.

- **Extended automata for the PRTT and ASRTT subsystems at the component level:** Figure 5.8 shows the extended version of the two automata that compose the component level of both PRTT and ASRTT subsystems. In order to incorporate release parameter changes, the new model introduces the boolean variables \( rpc \) (release parameter change) and \( started \). The variable \( rpc \) is set to true when the change of release parameters is triggered, being set to false when the actual change of parameters occur. The variable \( started \) is set to true, and continues true, when the thread starts executing its first release with the new release parameter, otherwise it is false. Therefore, when a thread is having its release parameter changed (\( rpc \) is true) or has not started yet (\( started \) is false), it is not possible to deschedule it. This change is reflected in the extended automaton RTT_SD of Figure 5.8 (a), which has certain transitions added and modified when the thread is descheduled.
In Figure 5.8 (b) the extended automaton RTT\textsubscript{Main}, implementing the main behaviour of the real-time thread, is presented. The behaviour added to this automaton is the handling of a release parameter change. When a change of release parameter is pending (\textit{rpc} is true) and the thread executes the \textit{waitForNextRelease()} method, instead of continuing its normal execution, the thread goes to state S8 and triggers the immediate change of release parameters (via channel \textit{change}). This is in accordance to the pseudo-restart of the thread, defined in the RTSJ.

Figure 5.8: Extended automata RTT\textsubscript{SDI} (a) and RTT\textsubscript{Main} (b).
• Extended automata for the release parameter change from the ASRTT subsystem to the PRTT subsystem at the scheduler level: For modeling a release parameter change, the first thing that has to be added to the model is a mechanism to generate this desired change. This is achieved by extending the automaton ASRTT\_GA, shown in Figure 5.9, by incorporating a synchronization to the channel \texttt{rpchange!}. This automaton is also used in the other model of release parameter change, which is presented later.

![Figure 5.9: Extended automaton ASRTT\_GA.](image)

In this model, when an aperiodic or sporadic thread becomes periodic, the synchronization for release parameter change (\texttt{rpchange?}) occurs with the automaton ASRTT\_ARD (Figure 5.10). Depending on the current state of the thread, two possible paths can be taken. Firstly, if the thread has started and is blocked-for-release-event, the release parameter change happens immediately by contacting the PRTT subsystem via channel \texttt{change}. This process initiates the execution of the thread using periodic parameters. Secondly, if the thread has started and is not blocked-for-release-event, a pending to change the release afterwards is set and the actual change takes place when the automaton RTT\_Main described previously executes the method \texttt{waitForNextRelease()}. It is also necessary to extend the automaton ASRTT\_GAI (Figure 5.11), responsible for monitoring the deadlines of aperiodic and sporadic threads. This has to be done in such a way that, once the thread changes its release, the thread stops monitoring the execution
and generating new releases (if the arrival queue is not empty). This is accomplished by inserting certain guards that check if the variable `changed` is set to true or not. The variable `changed` is used to define if the aperiodic or sporadic thread has changed its release parameter to periodic. If `changed` is true, the automaton goes to state $S0$ and remains there, because no further arrivals will be accepted (channel `accepted`).

From the periodic thread’s point of view (Figure 5.12), when a change of parameters takes place (transition $S0 \rightarrow S1$), it first waits for the start time to reach $(time == s)$ and immediately starts the execution of the first release of the periodic thread (`execute!`). After this, the normal behaviour for monitoring the deadline and releasing the thread each period is specified. However, another modification is done. The stop condition for the model, which was based on the maximum number of pending releases (see previous Figure 5.5), is changed to execute up to the maximum allowed number of time (constant `MAXT`). This change is reflected in the model via comparisons of the `MAXT` value to the clock `ct` – a clock that is never reset in the system.
Extended automata for the release parameter change from the PRTT subsystem to the ASRTT subsystem at the scheduler level: In this new model, the system starts executing as a periodic thread, later changing to an aperiodic or sporadic thread. This way, the automaton PRTT_PRD executes normally (Figure 5.13), but once a change of parameters happens (channel rpchange?) the following transitions can occur. If the current release of the periodic thread has not finished (is not blocked-for-release-event), a pending to change release is set and the actual change takes place when the
method `waitForNextRelease()` is executed. Otherwise, if the current release of the periodic thread has finished (it is blocked-for-release-event), it immediately applies the change of parameters by contacting the automaton `ASRTT_GAI`, shown in Figure 5.14, via channel `change`.

The automaton `ASRTT_GAI` shown in Figure 5.14 represents the start of the aperiodic or sporadic thread. It waits for the actual change (`change?`) and, when this occurs, it immediately generates a new arrival and the aperiodic or sporadic thread starts executing. The variable `started` is immediately set to true, meaning that the change of release has completed.
5.6 Formal Evaluation

For the formal evaluation of the models described for real-time threads, two steps have to be followed when using model checking as the formal analysis technique. Firstly, a scenario (or scenarios) providing the possible interesting behaviours of the models have to be specified. Secondly, formal properties about the proper behaviour of the component have to be defined and analysed against the defined scenario(s). In this section, scenarios are specified for each of the previous models: PRTT subsystem, ASRTT subsystem and release parameter changes (PRTT and ASRTT subsystems). Then, properties are specified for the models. The verification results for the properties are presented at the end of this section.

Verification scenarios for the PRTT subsystem

In the PRTT subsystem model, there are attributes related to the PRTT that need to have their values initialised. These attributes comprise the start time (variable $s$), period value (variable $p$) and the deadline time (variable $d$). Moreover, in order to make the model finite and analysable, it is also necessary to restrict the maximum allowed number of pending releases (constant NR) for the PRTT.
For defining the scenario, different values for the described variables were provided. In particular, during the verification process it was observed that both the start time and the period value do not impact on the complexity of the verification process. This happens because during the start time no interactions that change the state of the system occur and the tool can efficiently abstract this time, reducing state space. The same is true for increasing the period value, after the PRTT stops executing in a given period. Both these variables do not change the validity of the results for the verifications. Therefore a fixed value is set for the start time to zero and period to ten in the verification scenarios.

Indeed, only two different scenarios are provided by setting the deadline time to five time units in the first scenario and ten time units (equal to the period) in the second scenario. The complexity of the model (and consequently its memory size) increases, as the allowed number of pending releases (constant NR) is increased. Therefore, different variations of the scenarios (where the size of NR was increased) were analysed during verification. Though, only the verification information for the most significant model (capable of being handled in the used machine), where NR is 100, is presented.

**Verification scenarios for the ASRTT subsystem**

For the ASRTT subsystem model, the attributes related to the ASRTT that need to have their values initialised include: the MIT (constant MIT); the deadline (variable $d$); the maximum execution time of the model (constant MAXT); and the maximum number of possible arrivals (constant AQS). The first scenario is defined for the aperiodic model. MIT is set to zero and both deadline and possible number of arrivals are set to two. The second scenario specifies the model of a sporadic thread. The MIT, the deadline and the maximum number of arrivals are set to two.

In these scenarios, the increase of the constant MAXT leads to a complexity increase. Therefore, different variations of the scenarios (where the size of MAXT was increased)
were analysed during verification. Like for the scenarios of the PRTT subsystem, only the verification information for the most significant models (capable of being handled in the used machine), where MAXT is 12 (in the scenarios with aperiodic threads) and 40 (in the scenarios with sporadic threads), are presented. Moreover, variations of these scenarios using the overflow policies modelled (IGNORE and REPLACE) are defined, leading to a total of 4 different scenarios.

Verification scenarios for release parameter changes (PRTT and ASRTT subsystems)

Because this model joins both PRTT and ASRTT subsystems, the main variable that defines the complexity of the verification is the constant MAXT. Unfortunately, as it will be presented in the verification results, the MAXT constant does not reach high values. Therefore, in order to provide a scenario where all possible behaviours can be analysed, the period value (variable \( p \)) of the PRTT subsystem is defined to two. The MIT is defined to zero, when aperiodic threads are modelled, and to two when sporadic threads are modelled. Also, the maximum number of arrivals (constant \( AQS \)) is defined to two.

Due to the nature of the model, which is the change of release parameters, four models are provided. The first one changes from periodic thread to sporadic thread, the second from periodic to aperiodic, the third from sporadic to periodic and the fourth from aperiodic to periodic. For all these models, the deadline is set to two. Furthermore, each one of the scenarios has their MAXT constant increased to extend the search space. Like for the previous scenarios of PRTT and ASRTT subsystems, only the verification information for the most significant models (capable of being handled in the used machine), where MAXT is 10 (in the scenarios where periodic threads become aperiodic threads), 9 (in the scenarios where aperiodic threads become periodic threads), 60 (in the scenarios where periodic threads become sporadic threads) and 40 (in the scenarios where sporadic threads
become periodic threads), are presented. Like for the scenarios of the ASRTT subsystem, variations of these scenarios using the overflow policies that were modelled (IGNORE and REPLACE) are defined. This leads to a total of 8 scenarios.

Verification properties

In order to guarantee that the model is in accordance with the proposed behaviour of the RTSJ, the following two properties about the proper behaviour of the model are defined. They take into account two important variables defined for a real-time thread, \( mc \) (missed count) and \( pr \) (pending releases).

Property 1. The number of deadline misses is always greater than or equal to zero.

The safety property \( A[] (mc \geq 0) \) is specified. It is important to guarantee this property since a call to the \textit{waitForNextRelease()} method decrements the \( mc \) variable, and a negative value would show a potential problem in the model. Moreover, another safety property is provided.

Property 2. The number of pending releases is always greater than or equal to zero.

Similar to the previous property, the safety property \( A[] (pr \geq 0) \) is defined. The variable \( pr \) (pending releases) is also decremented on calls to the \textit{waitForNextRelease()} method and incremented when releases for the real-time thread are generated. Therefore, it is important to guarantee its consistency. In order to explore the behaviour of the models, a property about deadlock situations is defined.

Property 3. The real-time thread model never enters in a deadlock situation.

This property is specified using the \textit{deadlock} keyword available in the UPPAAL tool, leading to the formula \( A[] (not \text{deadlock}) \). When verified, all properties reached a true result for all models.
Verification results

The machine used in the verification of the properties was an AMD Athlon 64 X 2 Dual Core 2.2 GHz processor. Although this is a multiprocessor, only one of the two processors was used by the UPPAAL tool. The system was running the Slackware Linux OS (Linux kernel 2.6.24.2) with 3 Gb of RAM. The UPPAAL tool version 3.4.10 was used. In particular, the aggressive space optimisation option found in the UPPAAL tool was set for the verification. Note that both memory and time results for all the verifications presented in this section are provided as approximate results. This happens because UPPAAL does not give time and memory results, instead, one has to use a tool to monitor this data. Therefore, results may get slight different from verification to verification.

The results for the properties verified against the PRTT subsystem are presented in Table 5.1. Scenario 1 is the scenario with the deadline time set to five time units. Scenario 2 is the scenario with the deadline time set to ten time units (equal to the period). As it can be seen, both memory requirements and time used are relatively low. Indeed, increasing the size of the number of releases (constant NR) did not influence so much in the results, as the UPPAAL tool handled very well the verification of the model.

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario</th>
<th>Memory (in Megabytes)</th>
<th>Time (in seconds:milliseconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>≈ 22.6</td>
<td>≈ 02:30</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 22.7</td>
<td>≈ 02:20</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>≈ 22.7</td>
<td>≈ 02:30</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 22.1</td>
<td>≈ 02:30</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>≈ 22.5</td>
<td>≈ 05:20</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 22.7</td>
<td>≈ 05:10</td>
<td>true</td>
</tr>
</tbody>
</table>

The results for the properties verified against the ASRTT subsystem are presented in Table 5.2. Scenario 1 is the scenario for aperiodic real-time threads. The first variation 1a of the scenario uses the IGNORE policy, where the second variation 2b uses the REPLACE policy. Scenario 2 is the scenario for sporadic real-time threads, with variations for the
5.6 Formal Evaluation

IGNORE (2a) and REPLACE (2b) policies. Differently from the verification of the PRTT subsystem, the verifications consumed a lot more memory and time. This happens because the model has to count the time, due to the nature of the releases, and the UPPAAL tool was not efficiently as in the previous model.

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario</th>
<th>Memory (in Megabytes)</th>
<th>Time (in seconds:milliseconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1a</td>
<td>≈ 2185.6</td>
<td>≈ 1127.07</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2185.6</td>
<td>≈ 1131.72</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>≈ 2483.0</td>
<td>≈ 1149.15</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>≈ 2483.0</td>
<td>≈ 1151.30</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>1a</td>
<td>≈ 2185.6</td>
<td>≈ 1130.87</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2185.6</td>
<td>≈ 1133.04</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>≈ 2483.0</td>
<td>≈ 1148.24</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>≈ 2483.0</td>
<td>≈ 1149.82</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>1a</td>
<td>≈ 2185.9</td>
<td>≈ 1556.48</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2185.9</td>
<td>≈ 1561.72</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2a</td>
<td>≈ 2483.3</td>
<td>≈ 1552.08</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2b</td>
<td>≈ 2483.3</td>
<td>≈ 1550.46</td>
<td>true</td>
</tr>
</tbody>
</table>

The results for the properties verified against the change of release parameters are presented in Table 5.3. Note that, variations of the scenarios using the IGNORE (a) and REPLACE (b) policies are defined. The scenario 1 describes the change of release from periodic to aperiodic. In scenario 2, the change of release from periodic to sporadic is presented. Scenario 3 is used to describe the change of release from aperiodic to periodic. Finally, scenario 4 describes the change of release from sporadic to periodic. Similar to the verification of the ASRTT subsystem, the verifications consumed a large amount of memory and time. This occurred due to the need to count the time and also due to the added complexity of having more automata (from the PRTT subsystem) in the same verification model.
Table 5.3: Properties verified against changes to the release parameter.

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario</th>
<th>Memory (in Megabytes)</th>
<th>Time (in seconds:milliseconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1a</td>
<td>≈ 2470.5</td>
<td>≈ 1153:64</td>
<td>true</td>
</tr>
<tr>
<td>1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2a</td>
<td>≈ 2682.2</td>
<td>≈ 1185:37</td>
<td>true</td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3a</td>
<td>≈ 2112.2</td>
<td>≈ 737:05</td>
<td>true</td>
</tr>
<tr>
<td>3b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4a</td>
<td>≈ 2394.0</td>
<td>≈ 698:99</td>
<td>true</td>
</tr>
<tr>
<td>4b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1a</td>
<td>≈ 2470.5</td>
<td>≈ 1152:25</td>
<td>true</td>
</tr>
<tr>
<td>1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2a</td>
<td>≈ 2682.2</td>
<td>≈ 1183:76</td>
<td>true</td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3a</td>
<td>≈ 2112.2</td>
<td>≈ 736:37</td>
<td>true</td>
</tr>
<tr>
<td>3b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4a</td>
<td>≈ 2394.0</td>
<td>≈ 699:95</td>
<td>true</td>
</tr>
<tr>
<td>4b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1a</td>
<td>≈ 2470.8</td>
<td>≈ 1616:41</td>
<td>true</td>
</tr>
<tr>
<td>1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2a</td>
<td>≈ 2682.5</td>
<td>≈ 1624:24</td>
<td>true</td>
</tr>
<tr>
<td>2b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3a</td>
<td>≈ 2112.5</td>
<td>≈ 1082:87</td>
<td>true</td>
</tr>
<tr>
<td>3b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4a</td>
<td>≈ 2394.2</td>
<td>≈ 1078:02</td>
<td>true</td>
</tr>
<tr>
<td>4b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.7 Summary

This chapter has presented the behaviour of real-time threads with periodic, aperiodic and sporadic release parameters proposed in the RTSJ. A modelling architecture was proposed to model the behaviour using the Timed Automata (TA) formalism available in the UPPAAL tool (more information on the UPPAAL tool and the TA formalism can be found in Appendix A). The formal models were defined for: (a) periodic threads; (b) aperiodic and sporadic threads, where an aperiodic thread has a Minimum Inter-arrival Time (MIT) equal to 0 and the sporadic has an MIT greater than 0; (c) release parameter change from periodic to aperiodic or sporadic; and, (d) release parameter change from aperiodic or sporadic to periodic. Then, certain properties, with respect to the definition of the behaviour of real-time threads in the RTSJ were defined for the models and analysed (using model checking).
When using model checking as the formal analysis method, one important consideration is the definition of the scenarios, which the properties of the system are evaluated against. Using these, the formal analysis showed that the behaviour of real-time threads is correct under the assumed scenarios. Some problems are easier to use model checking than others. For this research, when analysing the behaviour of real-time threads using model checking, it was possible to explore the properties of interest in a given scenario—the scenario contained the behaviours of interest. However, several problems do not fall in this category. One example, relative to this research, is the formal analysis of the priority inheritance protocol proposed in the RTSJ, which is described in [77]. During the formal analysis of the protocol in [77], in order to explore the behaviour of the protocol, several different verification scenarios were used. Model checking can be an effective analysis method. But it depends if the behaviours of interest, from the problem being analysed, can be generalised using verification scenarios.
CHAPTER 6

Timing Error Detection in the RTSJ

This chapter presents different error detection models, following the considered timing errors, for the RTSJ. The contributions of this chapter are threefold. Firstly, it describes in more details the monitoring mechanisms provided by the RTSJ for the error detection of:

- deadline miss: a mechanism integrated to the scheduling model whereby the user sets the relative deadline for a given thread. Once a deadline is missed, an Asynchronous Event Handler (AEH) is released to handle the deadline miss;

- execution overrun of the thread’s WCET: the RTSJ provides a model, called cost monitoring and enforcement, which aims to provide damage confinement for the overrun of the pre-defined WCET (called cost) for a given thread;

- violation of the thread’s MIT assumptions: as described in the previous chapter, the RTSJ provides a mechanism for the detection of MIT violation and executing
sporadic and aperiodic real-time threads. Moreover, it also provides support for
the execution of threads within a temporal isolation model by providing a form of
Deferrable Server. This takes the form of \textit{ProcessingGroupParameters}.

The second contribution of this chapter concerns the cost monitoring and enforcement
model of the RTSJ. Since this model is considered optional, a formal description to analyse
its functionality is provided. It is then shown that, under certain execution scenarios, the
current cost monitoring and enforcement model of the RTSJ may not correctly confine the
behaviour of errant threads. Therefore, a revised model, which corrects this problem, is
proposed.

The third contribution of this chapter is the provision of a mechanism for the detection
of the following timing error:

- execution overrun of the thread’s WCET inside a shared resource (or blocking time
  violation for a thread): by using the blocking time monitoring mechanism proposed
  in Chapter 4 for POSIX/C, it is possible to provide a blocking time monitoring
  mechanism for the RTSJ. Following the usual constructs of the RTSJ, such a moni-
  toring model can be integrated to the scheduling model, releasing specific handlers
  when overruns occur.

Figure 6.1, previously presented in Section 3.4.1, summarises the RTSJ mechanisms
that can be used to detect the considered timing errors. It also depicts other components
of the system that are assumed to implement the correct functionality of the model. These
include the use of a uniprocessor architecture, FPS and priority inheritance algorithms.

The structure of this chapter is defined as follows. Section 6.1 describes the deadline
monitoring model for real-time threads of the RTSJ. In Section 6.2 the cost monitoring and
enforcement model of the RTSJ is presented and formally analysed. With the discovery of
a problem in the model, a revised model is proposed along with its formal model. Using
formal analysis, it is shown that this revised model corrects the previously found anomaly.

The detection of MIT violations and the support for the temporal isolated execution of sporadic and aperiodic threads is shown in Section 6.3. In order to provide a detection model for the violation of the blocking time for a thread in the RTSJ, Section 6.4 proposes a blocking time monitoring model.

### 6.1 Deadline Monitoring

The RTSJ integrates both scheduling and deadline monitoring mechanisms for a thread. This was shown in the previous chapter, where a formal model for real-time threads in the RTSJ was defined. When a real-time thread is created, the deadline value is passed as a release parameter (or one of its subclasses), which sets the deadline monitoring of the underlying RTSJ’s JVM. Figure 6.2 presents the attributes for the release parameter class and its subclasses, used for periodic, aperiodic and sporadic release parameters. With respect to deadline monitoring, the \textit{ReleaseParameters} class has two attributes: \textit{deadline}, which specifies the deadline value for the thread once it has been released; \textit{missHandler}, the event handler that is released to contend with a deadline miss.
Figure 6.2: Release parameter class and subclasses.

By integrating the scheduling and deadline monitoring mechanisms, the RTSJ provides a pre-defined behaviour for deadline misses. This was presented in the previous chapter, alongside with the behaviour of real-time threads. It indicated that, whenever a deadline is missed, the system should either: (a) release a miss handler, if a handler is available and the thread is not descheduled; (b) increment the variable \( \text{missCount} \) (which maintains the number of missed deadlines for the thread), if a miss handler is not available. Different from POSIX/C, the RTSJ already provides a framework for deadline monitoring, where the timing error (the deadline miss) is automatically considered.

6.2 Cost Monitoring and Enforcement

The RTSJ uses the term \textit{cost monitoring and enforcement}\footnote{Also, the term cost monitoring is used interchangeably.} to mean the process of tracking the amount of Central Processing Unit (CPU) consumption for a given real-time thread.
and, when necessary, taking appropriate actions over its behaviour (an enforcement model) upon cost overruns (execution time violations). The cost monitoring mechanism, associated with each thread, monitors in phases. When the thread is executing its $i^{th}$ release, and the $i^{th} + N$ release event (where $N > 0$) has not occurred, an $i^{th}$ cost monitoring phase will be associated with this release of the thread. The current CPU consumption is the amount of CPU consumed by the real-time thread in the current cost monitoring phase. In certain specific conditions, it is possible for the thread not to finish its $i^{th}$ release (make a call to the $\text{waitForNextRelease()}$ method) although the $i^{th} + N$ release event has already occurred. In this scenario, the thread is allowed to continue executing (although it exhausts the budget for its current $i^{th}$ release), because it can steal computation time from the release events that have already occurred. The rationale to let a thread steal computation time from next releases (that already occurred), is to allow for the thread the possibility to “catch-up” with the already occurred release events. Therefore, every time the thread finishes the budget for a given release, and uses the budget allocated for the next release, the cost monitoring mechanism starts considering a new cost monitoring phase. When the budget for a given release is exhausted, i.e., the current CPU consumption of the thread being monitored becomes equal to, or greater, than the cost (maximum execution time), a cost overrun is triggered, possibly releasing a handler to deal with the overrun. Moreover, the current CPU consumption value is reset whenever the cost monitoring mechanism enters a new monitoring phase.

As presented in the previous section about deadline monitoring, the RTSJ uses the $\text{ReleaseParameters}$ class to set attributes related to the execution of a real-time thread. Indeed, as shown in the previous Figure 6.2, the cost and $\text{overrunHandler}$ attributes compose the attributes related to the cost monitoring model of the RTSJ. The definitions and requirements for the cost monitoring and enforcement model of the RTSJ are depicted in Figure 6.3.
6.2 Cost Monitoring and Enforcement

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) On initial release</td>
<td>A new cost monitoring phase is started, the CPU consumption is set to zero (and will increase as the thread executes);</td>
</tr>
<tr>
<td>(b) Cost overrun</td>
<td>Occurs when the current CPU consumption becomes equal or greater than cost; the overrun asynchronous event handler, if any, is released;</td>
</tr>
<tr>
<td>(c) On the occurrence of a cost overrun</td>
<td></td>
</tr>
</tbody>
</table>
| (1) if the most recent release of the thread is the $i^{th}$ release and ((the $i^{th} + 1$ release event has not yet occurred) or (the $i^{th} + N$ release event has occurred and the $i^{th} + N$ cost monitoring phase has been started, where $N > 0$)); | (i) if the thread is *executing* or *eligible-for-execution*, then it becomes *blocked-by-cost-overrun*;  
(ii) if the thread is blocked for a reason other than *blocked-by-cost-overrun*, then a pending *blocked-by-cost-overrun* is set; |
| (2) else if the $i^{th} + N$ (where $N > 0$) release event has occurred and the $i^{th} + M$ (where $N > M$) cost monitoring phase has not yet started, then the $i^{th} + M$ cost monitoring phase is started and the CPU consumption is set to zero, and the thread remains in its current state; | |
| (d) On occurrence of the $i^{th}$ release event for a thread | (1) if the thread is *blocked-by-cost-overrun*, then the $i^{th}$ cost monitoring phase is started and the CPU consumption is set to zero, the thread is made *eligible-for-execution*;  
(2) if the thread is blocked for a reason other than *blocked-by-cost-overrun*: | (i) if a pending *blocked-by-cost-overrun* is set, then the pending is cleared, the $i^{th}$ cost monitoring phase is started, the CPU consumption is set to zero, and the thread remains in its current state;  
(ii) otherwise, no cost monitoring actions occurs; |
| (3) otherwise, no cost monitoring actions occurs; | |
| (e) On completion of the $i^{th}$ release for a thread | If the $i^{th} + N$ (where $N > 0$) cost monitoring phase has not started, it is started and the CPU consumption is set to zero; otherwise no action is taken; |
| (f) On changes to the cost value | (1) if the new value is lower or equal to consumption and the old value was greater than consumption, then a cost overrun occurs;  
(2) if the new value is greater than consumption: | (i) if the thread is *block-by-cost-overrun*, then the thread is made *eligible-for-execution*;  
(ii) if the thread is blocked and a pending for *block-by-cost-overrun* is set, then the pending is cleared;  
(iii) otherwise, no cost monitoring actions occurs; |
| (g) On reset of the cost monitoring | If the current cost monitoring phase is the $m^{th}$ phase and the thread is in the $n^{th}$ release ($n > m$), then a reset makes the cost monitoring start the $n^{th}$ phase (reset) and current CPU consumption is set to zero. |

Figure 6.3: Requirements and definitions for cost monitoring (based on [18]).

Following Figure 6.3, as soon as the thread starts its initial release, the cost monitoring starts to monitor the CPU consumption (a). If a cost overrun occurs for the $i^{th}$ release (b), any associated cost overrun event handler is released. When a cost overrun occurs for the $i^{th}$ release, the cost monitoring mechanism must know if the $i^{th} + 1$ release event has
already occurred (c). Based on this information, different actions may be triggered by the cost monitoring (c)(1) or (c)(2). On the occurrence of the \(i^{th}\) release event, the thread may have its consumption value reset: becoming eligible-for-execution (d)(1); or having a pending blocked-by-cost-overrun cleared (d)(2)(i). Otherwise, no action from the cost monitoring occurs. When the thread completes the current \(i^{th}\) release, the consumption value is reset and the cost monitoring enters its \(i^{th} + N\) phase (e). The cost value of the thread being monitored can be dynamically changed, triggering different actions by the cost monitoring (f). Finally, if the cost monitoring has started the \(m^{th}\) phase and the thread is in the \(n^{th}\) release, where \(n > m\), the state of the cost monitoring can be reset to consider the \(n^{th}\) release (g). This can happen in the RTSJ due to threads being descheduled and (re)scheduled by the application. In the next sections, a formal model of this cost monitoring model is provided.

### 6.2.1 Modelling Architecture

In order to formally analyse the cost monitoring model of the RTSJ, a modelling architecture using the TA formalism of the UPPAAL tool is defined in this section (Figure 6.4). This architecture is an extension to the modelling architecture of Section 5.2. It introduces the Cost Monitoring (CM) subsystem in the architecture, which provides the behaviour of the cost monitoring model of the RTSJ to control the behaviour of Periodic Real-Time Threads and Aperiodic/Sporadic Real-Time Threads, represented by the PRTT and AS-RTT subsystems, respectively. Like in the previous architecture, the basic idea is to decompose the functionalities of the components (real-time threads and cost monitoring) into three different levels (each level can be modelled with more than one automaton):

- **application level**: responsible for modelling the interactions of the application that can change the behaviour of the PRTT, ASRTT and CM subsystems at the component level;
6.2 Cost Monitoring and Enforcement

- component level: defines the main behaviour of the PRTT, ASRTT and CM subsystems. Also specifies the interaction between these two subsystems, i.e., the monitoring and enforcement features;

- scheduler level: represents the basic functions and interactions of the scheduler, substituting the need to model a real-time kernel. The portion defined for the PRTT and ASRTT subsystems deals with the generation of release events and monitoring deadlines. Differently, the portion specified for the CM subsystem is used to keep track of the release events generated for the PRTT and ASRTT subsystems.

Again, the automata defined at the application level are used to non-deterministically generate inputs to the component level. It is responsible for generating non-deterministic inputs to the PRTT, ASRTT and CM subsystems at the component level (through synchronization channels). They also model the interaction with any released event handlers.

The modelling architecture following these ideas is shown in Figure 6.4. The CM subsystem has a global variable, called \( cm\_cs \), used to represent the current state of PRTT or ASRTT subsystem being monitored. This variable is utilised in both component and scheduler levels, but is only set at the component level. It is also used to correctly synchronize the actions between architecture levels. The interactions represented by arrows in Figure 6.4 are specified through synchronization channels, whose names are depicted next to each arrow. The channels that are used to synchronize between the two components of the architecture are named with a prefix (\( rtt \) for the PRTT and ASRTT subsystems and \( cm \) for the CM subsystem) that indicates the subsystem responsible for triggering the synchronization. Note that, in the modelling architecture of Figure 6.4, the automata that compose each level of the architecture and the possible synchronization channels used between them are omitted.
Next, modified versions of both periodic real-time threads and aperiodic and sporadic real-time threads presented in the previous chapter are specified. The new models include certain transitions that are used to interact with the CM subsystem introduced in the modelling architecture.

### 6.2.2 Extended Formal Models of Real-Time Threads

The models for Periodic Real-Time Thread (PRTT subsystem) and Aperiodic/Sporadic Real-Time Thread (ASRTT subsystem) are extended in this section with new transitions that enable its interaction with the cost monitoring model (CM subsystem) that will be presented in the next Section. Basically, the introduced transitions are at the component level and the scheduler level of both PRTT and ASRTT subsystems. This means that the application level of these subsystems is left unmodified.
• Automaton for the PRTT and ASRTT subsystems at the component level:
  At this level, two automata (RTT\_SDI and RTT\_Main) are used to specify the behaviour of PRTT and ASRTT subsystems. The RTT\_SDI automaton is left unchanged, it still deals with calls to schedule and deschedule the thread from the application. Differently, the RTT\_Main automaton, shown in Figure 6.5, is extended with two new states: \texttt{bbco} (blocked-by-cost-overrun), a state where the thread has exceeded its maximum execution time and has become blocked; and \texttt{pbbco} (pending for blocked-by-cost-overrun), which denotes a state where the thread is blocked, but a cost overrun has occurred (due to a change of cost value). Also, the synchronizations to the broadcast channels \texttt{descheduled!}, completion! and completed! used in the ASRTT subsystem are substituted with synchronizations, respectively, to the channels \texttt{rtt\_descheduled!}, rtt\_completion! and rtt\_completed!. Besides being used in the ASRTT subsystem, these synchronizations are also utilized for informing the cost monitoring about the operations that were realized.

![Extended RTT\_Main automaton diagram](image)

Figure 6.5: Extended RTT\_Main automaton.

Furthermore, the following behaviour is incorporated to the automaton. When ex-
executing, an overrun can occur (\textit{cm\_overrun?}) or it can non-deterministically change to the blocked state, informing the cost monitoring (\textit{rtt\_block!}). Once in the blocked state, a thread can have a pending for blocked-by-cost-overrun (\textit{cm\_pbbco?}) or become (non-deterministically) unblocked, informing the cost monitoring about it (\textit{rtt\_unblock!}). If a pending for blocked-by-cost-overrun is set for the thread and: (i) it unblocks, it becomes blocked-by-cost-overrun (\textit{rtt\_unblock!}); (ii) a new release occurs, the pending is cleared (\textit{cm\_clearpbbco?}). If the thread is blocked-by-cost-overrun and the cost value is increased or a new release occurs, the cost monitoring makes the thread eligible-for-execution (\textit{cm\_eligible?}). Moreover, if the thread is: blocked-for-schedule and becomes blocked-for-release-event, it resets the cost monitoring (\textit{rtt\_reset!}); blocked-for-release-event and becomes blocked-for-schedule, it informs the cost monitoring about it (\textit{rtt\_descheduled!}).

- **Automaton for the PRTT subsystem at the scheduler level:** The extended automaton PRTT\_RD (PRTT Releases/Deadlines) is presented in Figure 6.6. Basically, it is extended with three synchronizations to the channels \textit{rtt\_release!} that are used to inform the cost monitoring that the PRTT has been released. These synchronisations take place in transitions \(S_1 \rightarrow S_2\), \(S_4 \rightarrow S_2\) and \(S_3 \rightarrow S_2\). The rest of the model remains unchanged.

![Figure 6.6: Extended PRTT\_RD automaton.](image-url)
• **Automata for the ASRTT subsystem at the scheduler level:** The ASRTT subsystem defined in Section 5.4 consisted of four different automata. Two of those automata, ASRTT.GAI and ASRTT.ARD, have to be modified in order for the ASRTT subsystem to interact with the cost monitoring. The extended automata are shown in Figure 6.7. The automaton ASRTT.GAI is extended with a synchronization to the channel \texttt{rtt} release!, in transition \texttt{S3} → \texttt{S0}, which informs the cost monitoring that the ASRTT has been released. The other automaton ASRTT.ARD has the synchronizations to the broadcast channels \texttt{descheduled?}, \texttt{completion?} and \texttt{completed?} substituted with synchronizations, respectively, to the channels \texttt{rtt}descheduled?, \texttt{rtt}completion? and \texttt{rtt}completed?. These modifications are in accordance with the changes presented previously, for the extended RTT.Main automaton, in Figure 6.5.

![Figure 6.7: Extended (a) ASRTT.GAI and (b) ASRTT.ARD automata.](image-url)
6.2.3 Formal Model of the Cost Monitoring

The TA model for the CM subsystem is composed of five automata. As discussed in the modelling architecture, the current state of the CM subsystem is set through the global variable $cm_cs$. Because the idea of this model is to monitor the execution of the real-time threads (modelled using the PRTT and ASRTT subsystems), it reflects on its behaviour the current state of the thread being monitored. The interaction between both CM and PRTT or ASRTT subsystems needs to be correct otherwise the model may become inconsistent. The possible values for the state of the thread include: INIT, EXEC, BBCO, PBBCO, BFRE, BFS and BLK.

- **Automaton for the CM subsystem at the application level**: Specifies the method that can affect the behaviour of the CM. The automaton CM\_Cost is presented in Figure 6.8. This automaton represents calls (channel synchronizations) from the application that can change the current cost value (budget) of the thread being monitored ($changecost!$).

![Figure 6.8: CM\_Cost automaton.](image)

- **Automata for the CM subsystem at the component level**: Specify the interface between the change of cost calls at the application level and the CM subsystem. It also defines the main behaviour of the CM component. The interface is represented by the automaton CM\_CI (CM Cost Interface) depicted in Figure 6.9 (a). This automaton has two main transitions that are activated by the change of the cost ($changecost?$) from the application level. Depending on the current state of the CM component (variable $cm_cs$), it may trigger different actions in the CM\_Main automaton (including overruns, see actions...
6.2 Cost Monitoring and Enforcement

changeoverrun! depicted next. Also, it swaps values between the current cost (variable cost) and the new cost (variable ncost). Using this technique of swapping values it is possible to verify all possible interleaving executions where the old and new cost values are different from each other.

Figure 6.9: (a) CM_CI, (b) CM_Main and (c) CM_Counter automata.

Figure 6.9 (b) shows the automaton CM_Main, describing the main behaviour of the
CM component. It specifies the actions triggered by the cost monitoring in the occurrence of cost overruns and also resets the CPU consumption to zero. Since it is not possible to store the value of a clock using the TA formalism, the automaton CM.Counter shown in Figure 6.9 (c) is defined. This automaton counts, every one time unit (this suffices to capture the execution profile of real-time threads), the CPU consumption ($cpr$) and generates overruns ($overrun!$) when the current CPU consumption equals the current cost set for the thread.

The CM.Main automaton starts by monitoring the thread that was initially released ($monitor?$) whilst counting the CPU consumption ($start!$). During execution, when the thread becomes blocked ($rtt\_block?$), the cost monitoring stops counting the CPU consumption value of the thread. If the thread changes from a blocked state to executing ($rtt\_unblock?$), the cost monitoring starts counting the CPU consumption again. If a cost overrun occurs and the thread is blocked ($changeoverrun?$), a blocked-by-cost-overrun is left pending. Either when a new release event ($monitor?$) or a change to the cost value (if the new cost is greater than the current CPU consumption) occurs ($clearpbco?$), the thread has the blocked-by-cost-overrun pending cleared ($cm\_clearpbco!$) and considers the thread to be in the blocked state. Otherwise, if the thread changes from executing straight away, it becomes blocked-by-cost-overrun ($rtt\_unblock?$).

When the thread is executing and a cost overrun in a release occurs (due either to consumption-release ($overrun?$) or change in the cost value ($changeoverrun?$)) and the release being monitored is the $i^{th}$ release ($cor == 1$), the thread becomes blocked-by-cost-overrun. Otherwise, the cost monitoring enters its $i^{th} + 1$ phase ($cor > 1$), resetting the consumption value and not changing the state of the thread. If a thread is blocked-by-cost-overrun, it can change to the executing state when its cost changes to a value greater than the current CPU consumption ($makeefe?$) or a new release occurs for the thread ($monitor?$). When a thread completes a release, if the release being monitored is
6.2 Cost Monitoring and Enforcement

The $i^{th}$ release and the current release for the thread is the $i^{th}$ release ($\text{rtt}_{\text{completed}}$), the thread becomes blocked-for-release-event. Otherwise ($\text{rtt}_{\text{completion}}$), the number of outstanding releases is decremented (variable $\text{cor}$), the consumption value is reset, and the thread continues its execution. If the thread is descheduled ($\text{rtt}_{\text{descheduled}}$), it can eventually become scheduled again, where the cost monitoring is reset ($\text{rtt}_{\text{reset}}$).

- **Automaton for the CM subsystem at the scheduler level:** This is used to keep track of the release events generated for the PRTT or ASRTT subsystems. It is modelled using the automaton CM_MR (CM Monitor Releases) shown in Figure 6.10. This automaton counts the number of outstanding releases (variable $\text{cor}$, meaning count outstanding releases) that represents the $i^{th}$ monitoring phase for the mechanism. When a new release occurs ($\text{rtt}_{\text{release}}$) the number of outstanding releases is increased and the cost monitoring is informed to start monitoring ($\text{monitor!}$). If the current state of the thread is blocked-for-release-event, the number of outstanding releases is reset ($\text{cor} := 1$). This means that the $i^{th}$ cost monitoring phase is related to the $i^{th}$ release of the thread.

![Figure 6.10: CM_MR automaton.](image)

- **Variables for verification purposes:** In the automata shown in Figures 6.9 and 6.10 three variables were introduced for verification purposes. The variable $tpr$ (time per release) is used to count the current CPU consumption value for a given release of the thread until its completion. The variable $tpp$ (time per period) counts the CPU consumption value in a period. The variable $oh$ (overrun handler) is set to true when a cost overrun occurs,
and set to false when the thread completes a release.

6.2.4 Formal Evaluation

Since model checking is used as the formal analysis method, scenarios that are used in the verification process have to be initially specified. Then, two types of properties are specified for the cost monitoring model. The first set is called consistency properties and are used to guarantee that the model does specify the correct behaviour (defined in the RTSJ). Having these properties verified to true in the models, another set of properties, called evaluation properties, are defined. Evaluation properties aim to analyse the behaviour of the model. The verification results of both set of properties are shown at the end of this section.

Verification scenarios for interacting with the PRTT subsystem

In the model, certain attributes need to have their values initialised. In the PRTT subsystem, these attributes comprise the start time (variable $s$), period value (variable $p$) and the deadline time (variable $d$). In the CM subsystem, the attributes include the initial cost (variable $cost$) and the possible new cost (variable $ncost$). Moreover, in order to make the model finite and analysable, it is also necessary to restrict the maximum allowed number of pending releases (constant NR) for the PRTT.

For defining the scenario, different values for the described variables are provided. In particular, during the verification process it was observed that both the start time and the period value do not impact on the complexity of the verification process. This happens because during the start time no interactions that change the state of the system occur and the tool can efficiently abstract this time, reducing state space. The same is true for increasing the period value, after the PRTT stops executing in a given period. Also, both these variables do not change the validity of the results for the verifications. Therefore a
fixed value is set for the start time to zero and period to ten in the verification scenario.

Since a system based on a uniprocessor architecture is being analysed, it is not desirable to have a cost equal to the period. This would model a system where only one thread would be executing without stopping – if the thread has the highest priority. Therefore, the initial cost value is set to four time units and it can dynamically change to two time units. This way, due to the period value of ten, the PRTT initially executes up to 40% of the CPU, but once a new cost is determined, it can execute up to 20% of the processor. Moreover, another variation of the scenario, where the cost value remains constant with four time units, is specified. This is used for the analysis of property e4. The deadline time is set to a fixed value of five time units. The complexity of the model (and consequently its memory size) increases, as the allowed number of pending releases (constant NR) is increased. Therefore, different variations of the scenarios (where the size of NR was increased) were analysed during verification. Though, only the verification information for the most significant model (capable of being handled in the used machine), where NR is 6, is presented.

Verification scenarios for interacting with the ASRTT subsystem

Besides setting the attributes (variables cost and ncost) for the CM subsystem, the following attributes for the ASRTT subsystem are initialised: the MIT (constant MIT); the deadline (variable d); the maximum execution time of the model (constant MAXT); and the maximum number of possible arrivals (constant AQS). Only one scenario, for the sporadic model, is defined. In the aperiodic model, the added complexity is very high, and the verification does not complete – it becomes out of memory. Moreover, the formal analysis of the sporadic model also provides confidence over the correctness of the aperiodic model. In the defined scenario, the MIT is set to 4, the deadline is set to 3 and the maximum number of arrivals is set to two. The cost (variable cost) is set to 2 and the new cost
(variable $ncost$) is set to 1.

In the scenario, the increase of the constant MAXT leads to a complexity increase. Therefore, different variations of the scenarios (where the size of MAXT was increased) were analysed during verification. Though, like for the scenarios of the PRTT subsystem, only the verification information for the most significant models (capable of being handled in the used machine), where MAXT is 18 (in the scenarios with sporadic threads). Variations of this scenario using the overflow policies modelled (IGNORE and REPLACE) are defined, leading to a total of 2 different scenarios. Moreover, another variation of the scenario, where the cost value remains constant with two time units, is specified. This is used for the analysis of property $e4$.

Model consistency properties

Because the behaviour of a thread is modelled, in the first two properties it is checked that no inconsistency behaviours from a thread point of view occurs. The attention is focused in two key variables of the behaviour for threads: $mc$, used to count the number of deadline misses; and $pr$, which counts the number of pending releases.

Property c1. The number of deadline misses is always greater than or equal to zero.

This property specifies that the variable $mc$ must never have a negative value, such condition would indicate a faulty behaviour in the model. The following safety property $A[] (mc \geq 0)$ is specified – meaning that in all states of possible executions, the value of $mc$ is always greater than or equal to zero.

The second property focuses in the $pr$ variable (pending releases).

Property c2. The number of pending releases is always greater than or equal to zero.

In this property it is important to make sure that the value of the $pr$ variable is greater than zero. The safety property $A[] (pr \geq 0)$ is defined.
All properties ended up being valid (they had a true result) in the verification process. After checking that threads present the expected behaviour, the attention is focused on guaranteeing that the interaction between the CM and PRTT or ASRTT subsystems is consistent with the expected behaviour. For the rest of the consistency properties, liveness properties using the pattern \((\Phi \Rightarrow \phi)^2\) found in the UPPAAL tool are specified. In the next two properties the possible behaviours for a thread, when a cost overrun occurs and the thread is in the executing state, are analysed.

The first property aims to check that in the occurrence of cost overrun (per release) and the thread is executing the current \(i^{th}\) release, its behaviour is enforced (the thread becomes blocked-by-cost-overrun).

**Property c3.** If a thread is in the executing state and the cost monitoring is in its \(i^{th}\) phase (and the \(i^{th} + N^3\) release event has not occurred), on the occurrence of a cost overrun (where the consumption value is greater than or equal to the cost) the cost monitoring always puts the thread in the blocked-by-cost-overrun state.

For this property it is important to make sure that the thread is currently in the executing state \((CM_{Main}.exec)\), the cost monitoring is in its \(i^{th}\) phase \((cor == 1)\) and that a cost overrun per release occurs \((cpr \geq cost)\) – this specifies the left-side. Having all those conditions true, it is also necessary to make sure that the thread is changed to the blocked-by-cost-overrun state \((RTT_{Main.bbco})\) – this defines the right-side. This leads to the liveness formula \(((CM_{Main}.exec) \&\& (cor == 1) \&\& (cpr \geq cost)) \Rightarrow (RTT_{Main.bbco})\).

The next property aims to check that in the occurrence of a cost overrun when the thread is in the executing state, but the \(i^{th} + N\) release event has already occurred, its behaviour is not enforced, because the replenishment policy allows it to continue executing.

\(^2\)See Appendix A.

\(^3\)It is assumed that \(N > 0\) for the properties defined in this section.
Property c4. If a thread is in the executing state and the cost monitoring is in its $i^{\text{th}}$ phase (and the $i^{\text{th}} + N$ release event has already occurred), on the occurrence of a cost overrun (where the consumption value is greater than or equal to the cost) the cost monitoring does not change the current state of the thread.

For this property it is necessary to make sure that the thread is currently in the executing state ($CM_{Main.exec}$), the cost monitoring is in its $i^{\text{th}} + N$ phase ($cor > 1$) and that a cost overrun per release occurs ($cpr \geq cost$) – this defines the left-side. Having all those conditions true, it is specified that the thread continues in the same executing state ($RTT_{Main.exec}$) because its cost must have been replenished – this specifies the right-side. This leads to the liveness formula \((CM_{Main.exec}) \land (cor > 1) \land (cpr \geq cost) \Rightarrow (RTT_{Main.exec})\).

In the previous two properties it was analysed the behaviour upon cost overruns when the thread is in the executing state. In the next two properties the idea is to analyse the behaviour of cost overruns when the thread is in the blocked state. It should be noted that, even if a thread is not using the CPU, an overrun can happen since the application may change the cost to a lower value and, therefore, trigger an overrun.

The next property aims to check that in the occurrence of a cost overrun (per release) when the thread is blocked, but in the current $i^{\text{th}}$ release, a pending for blocked-by-cost-overrun is set for the thread.

Property c5. If a thread is in the blocked state and the cost monitoring is in its $i^{\text{th}}$ phase (and the $i^{\text{th}} + N$ release event has not occurred), on the occurrence of a cost overrun (where the consumption value is greater than or equal to the cost) the cost monitoring sets a pending for blocked-by-cost-overrun in the thread.

In this property is is necessary to make sure that the thread is currently in the blocked state ($CM_{Main.blk}$), the cost monitoring is in its $i^{\text{th}}$ phase ($cor == 1$) and that a cost overrun per release occurs ($cpr \geq cost$) – this specifies the left-side. Having all
those conditions true, it is specified that a pending for blocked-by-cost-overrun is set for
the thread, changing its state to \((RTT_{Main.pbbco})\) – this defines the right-side. This
leads to the liveness formula \(((CM_{Main.blk}) \&\& (cor == 1) \&\& (cpr \geq \text{cost})) \rightsquigarrow
(RTT_{Main.pbbco}).\)

For the next property it is defined that in the occurrence of a cost overrun when the
thread is in the blocked state, but the \(i^{th} + N\) release event has already occurred, the
pending for blocked-by-cost-overrun is not set since the replenishment policy allows it.

**Property c6. If a thread is in the blocked state and the cost monitoring is in its \(i^{th}\) phase
(and the \(i^{th} + N\) release event has already occurred), on the occurrence of a cost overrun
(where the \text{consumption} value is greater than or equal to the cost) the cost monitoring
does not change the current state of the thread.**

For this property it is important to make sure that the thread is currently in the
blocked state \((CM_{Main.blk})\), the cost monitoring is in its \(i^{th} + N\) phase \((cor > 1)\) and
that a cost overrun per release occurs \((cpr \geq \text{cost})\) – this defines the left-side. Having all
those conditions true, it is ensured that the thread continues in the same blocked state
\((RTT_{Main.blk})\), because its cost must have been replenished and a pending for blocked-
by-cost-overrun is not set – this specifies the right-side. This leads to the liveness formula
\(((CM_{Main.blk}) \&\& (cor > 1) \&\& (cpr \geq \text{cost})) \rightsquigarrow (RTT_{Main.blk}).\)

In summary, in properties **c1** and **c2** it is guaranteed that no deviations occur in the
model of a thread. Two safety properties were specified, about its key variables used to
count the number of deadline misses (variable \(mc\)) and pending releases (variable \(pr\)).
With respect to the cost monitoring and enforcement, properties **c3** to **c6** have addressed
the behaviour triggered by cost overruns due to release overruns using liveness formulas.
Since all those properties had a true result on the verification runs, it is possible to state
that the model is coherent and consistent with the definitions found in the RTSJ and the
proposal.
Model evaluation properties

Now, certain properties used to explore different behaviours of the model and evaluate the correctness of the cost monitoring and enforcement approach are specified. In the first property, the model checking tool is used to generate all possible executions for the scenarios and check that no deadlock situations can occur.

**Property e1.** The use of cost monitoring with a thread must never leads to a deadlock situation.

In order to specify this property, the *deadlock* keyword available in the UPPAAL tool is used. This keyword is used to identify a state in the system where no transitions can occur (deadlocked state). Using this keyword the safety formula $A[] (not\ deadlock)$ is specified. When verified, this property ended up being valid for the scenarios.

The next property focuses on analysing the enforcement of behaviour imposed by the cost monitoring when a thread exceeds its cost (budget) in a given release.

**Property e2.** A thread cannot execute on the CPU for more than its cost per release.

For the specification of this property it is necessary to monitor the CPU execution time for the thread during a release. The variable $tpr$ is used to count the consumption of a thread until it makes a completion for the current release. Then, the proposition $(tpr \leq cost)$ asserts that the value of the $tpr$ variable is always lower than or equal to the current maximum possible cost value of the scenario being verified. The following safety formula is specified $A[] (tpr \leq cost)$. This property was not valid during the verification process and a counter-example, using one of the verification scenarios, is shown in Figure 6.11.

In the counter-example, the thread starts executing with a budget of 4. Then, it executes its entire budget in the first period, generating a cost overrun and becoming blocked-by-cost-overrun (enforced behaviour). Because it did not complete the first release in the first period (calling the `waitForNextRelease()` method), it resumed execution in the
second period (using the budget of the second release). Therefore, it exceeded its execution budget in a given release. Nevertheless, it is important to note that, when the thread exceeded its cost in the first period, a cost overrun handler was fired for that release (guaranteeing that actions may be taken by the program to correct the behaviour). Also, the thread was suspended (because of the enforced behaviour). Consequently, there is no impact on lower priority threads.

Since the behaviour of the cost monitoring mechanism seems to be working fine, according to the counter-example, another property about the behaviour of the mechanism when a thread exceeds the cost in a given release is specified.

This property is used to ensure that if the cost is exceeded in a release, specific measures (generation of an overrun handler) are taken by the cost monitoring mechanism.

**Property e3.** *If a thread executes the CPU for more than its cost per release a cost overrun handler is released.*

In order to specify this property the pattern \((\Phi \rightsquigarrow \phi)\) is utilized. It is important to make sure that during execution the thread exceeds its budget per release \((tpr \geq cost)\) – defines the left-side of the formula. Making sure that this proposition holds, there is the consequence of the generation of an overrun handler \((oh == true)\) – specifying the right-side of the formula. The following liveness property is specified \((tpr \geq cost) \rightsquigarrow (oh == true)\). This property ended up being valid for the scenarios in the verification process,
guaranteeing that an overrun handler is always released whenever the thread exceeds its budget per release.

The last property aims to guarantee that during a given period, instead of a release, only one budget is permitted.

**Property e4.** *A thread cannot execute on the CPU for more than its maximum allowed cost per period.*

For the specification of this property it is necessary to monitor the CPU execution time for the thread during a period. The variable \( tpp \) is used to count the consumption of a thread per period. Then the proposition \( (tpp \leq cost) \) asserts that the value of the \( tpp \) variable is always lower than or equal to the possible cost value. Note that, for the verification of this property, the scenario with a fixed cost value was used. If a scenario with dynamic change of costs was used, there could be a situation where the thread has the current cost of 4 time units and executed for 3 time units in a period. Because of the dynamic changes of cost, it could change to a cost of 2 time units, a situation where \( tpp \) is greater than the current cost, but not an error condition from the cost monitoring mechanism. The safety formula \( A[\](tpp \leq cost) \) is defined.

This property has a false result and reveals a possible anomalous behaviour in the current definition of the cost monitoring approach of the RTSJ. An execution example that shows the anomalous behaviour is depicted in Figure 6.12. The scenario used during verification to generate the counter-example consists of a thread executing with a period of 10 time units, a deadline of 5 time units, and a cost of 4 time units.

As shown in Figure 6.12, the thread starts executing and becomes blocked at time one (say, as a result of an I/O operation), unblocking and restarting execution at time 11 (in the second period). It continues executing and an overrun occurs at time 14 (generating a handler) when the budget of the first release finishes. The current cost monitoring and enforcement model of the RTSJ supports a full replenishment of cost when the \( i^{th} + 1 \)
release event has already occurred. In this case, at time 14, the $i^{th} + 1$ cost monitoring phase starts and the thread can continue executing for 4 more time units. Indeed, it can exceed the allowed budget in the period and execute from time 15 up to time 18. At time 18 it stops because an overrun for the second release occurs and the thread becomes blocked-by-cost-overrun. This way, in this particular run, the thread is allowed to execute up to seven time units in the second period (70% of the CPU in a uniprocessor architecture because the period is of 10 time units).

But the problem could be worst. For instance, consider that (for whatever reason) the thread gets blocked for a number of $R$ release events. In this case, there is the possibility of the thread executing uninterrupted for: $R$ release events multiplied by the thread’s cost. Depending on the current state of the application, this behaviour could have a major impact on lower priority threads and potentially undermine the feasibility analysis of the application. Of course, if the application has specified a deadline miss handler and an overrun handler, these will be released at each missed deadline and overrun. However, there is no automatic recovery action in such a handler that can immediately stop the execution of the thread, and consequently isolation of the timing fault cannot be guaranteed. In the next section, a solution is proposed to this problem alongside a formal representation of the model. Also, a definition of this model’s behaviour in terms of the notation used in
the RTSJ is provided.

In summary, in property $e_1$ it was analysed that no deadlock situations could occur in the model, reaching a true result in the verification process. Then, in property $e_2$, it was considered that it was not possible for a thread to execute for more than its cost in a given release. This property ended up being false in the verification, leading to the specification of property $e_3$ which stated that: if the thread executes for more than its cost in a release, it will always release a handler to correct the situation. Having reached a true result with this property, property $e_4$ was defined, which specified that it is not possible for a thread to execute for more than its maximum cost value in a period. This property was false and revealed a possible anomaly in the cost monitoring approach of the RTSJ, where the behaviour upon the errant thread is not enforced (the thread does not become blocked-by-cost-overrun).

Verification results

Like in the verification of the models in the last chapter, the machine used in the verification of the properties was an AMD Athlon 64 X 2 Dual Core 2.2 GHz processor. Also, the system was running the Slackware Linux OS (Linux kernel 2.6.24.2) with 3 Gb of RAM, and the UPPAAL tool version 3.4.10 was used. In particular, the aggressive space optimisation option found in the UPPAAL tool was set for the verification. Note that both memory and time results for all the verifications presented in this section are provided as approximate results.

Table 6.1 shows the verification results for the properties against the model interacting with the PRTT subsystem. Scenario 1 is the scenario with the deadline time set to five time units. Scenario 2 is the scenario with the deadline time set to ten time units (equal to the period). In particular, the property $e_4$ is verified for both scenarios 1 and 2, but with the cost value fixed. As it can be seen, both memory requirements and time used are
relatively low. Indeed, increasing the size of the number of releases (constant NR) did not influence so much in the results, as the UPPAAL tool handled very well the verification of the model.

The results for the properties verified against the model interacting with the ASRTT subsystem are presented in Table 6.2. Scenario 1 is the scenario for sporadic real-time threads. The first variation 1a of the scenario uses the IGNORE policy, whereas the second variation 2b, uses the REPLACE policy. As before, the property e4 is verified for scenario 1 and its variations, but with the cost value fixed. Differently from the verification of the model interacting with PRTT subsystem, the verifications here consumed a lot more memory requirements and time. This happens because the model has to count the time, due to the nature of the releases, and the UPPAAL tool can not be used as efficiently as in the previous model.
Table 6.2: Properties verified against the model with the ASRTT subsystem.

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario</th>
<th>Memory (in Megabytes)</th>
<th>Time (in seconds:milliseconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>1a</td>
<td>≈ 2185.1</td>
<td>≈ 616:38</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2185.1</td>
<td>≈ 608:43</td>
<td>true</td>
</tr>
<tr>
<td>c2</td>
<td>1a</td>
<td>≈ 2185.1</td>
<td>≈ 615:97</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2185.1</td>
<td>≈ 606:42</td>
<td>true</td>
</tr>
<tr>
<td>c3</td>
<td>1a</td>
<td>≈ 2365.9</td>
<td>≈ 628:40</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2365.9</td>
<td>≈ 618:18</td>
<td>true</td>
</tr>
<tr>
<td>c4</td>
<td>1a</td>
<td>≈ 2225.2</td>
<td>≈ 612:14</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2225.2</td>
<td>≈ 605:03</td>
<td>true</td>
</tr>
<tr>
<td>c5</td>
<td>1a</td>
<td>≈ 2275.3</td>
<td>≈ 611:22</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2275.3</td>
<td>≈ 603:90</td>
<td>true</td>
</tr>
<tr>
<td>c6</td>
<td>1a</td>
<td>≈ 2307.5</td>
<td>≈ 608:12</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2307.5</td>
<td>≈ 601:46</td>
<td>true</td>
</tr>
<tr>
<td>c1</td>
<td>1a</td>
<td>≈ 2185.5</td>
<td>≈ 1015:28</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2185.5</td>
<td>≈ 991:19</td>
<td>true</td>
</tr>
<tr>
<td>c2</td>
<td>1a</td>
<td>≈ 1.5</td>
<td>≈ 00:10</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1.5</td>
<td>≈ 00:10</td>
<td>false</td>
</tr>
<tr>
<td>c3</td>
<td>1a</td>
<td>≈ 3108.3</td>
<td>≈ 273:47</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 3108.2</td>
<td>≈ 269:26</td>
<td>true</td>
</tr>
<tr>
<td>c4</td>
<td>1a</td>
<td>≈ 4.6</td>
<td>≈ 00:20</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 5.3</td>
<td>≈ 00:20</td>
<td>false</td>
</tr>
</tbody>
</table>

6.2.5 A Revised Cost Monitoring and Enforcement Model

In general, there are two goals for any cost enforcement approach:

1. notify the application about any overrun execution state (faulty behaviour), so that measures can be taken by the application;

2. ensure that the run time behaviour of the application does not undermine any performed feasibility analysis without the explicit intervention of the application.

It was observed that there are some scenarios in which the RTSJ does not guarantee the second property. In order to solve the anomalous behaviour discussed previously and enforce that a thread cannot exceed its budget in a period, the first solution sought was for the cost monitoring mechanism to not allow outstanding releases. This leads to a cost per period mechanism used to enforce the behaviour upon the thread. Specifically, on each new $i^{th} + 1$ release occurrence, the cost monitoring system would reset the current consumption of the thread to zero and consider this $i^{th} + 1$ release to be the new $i^{th}$
6.2 Cost Monitoring and Enforcement

cost monitoring phase. Therefore, whenever the current consumption exceeds the cost in
the current period, an overrun occurs and the thread becomes blocked-for-release-event.
Unfortunately, as a consequence of not considering outstanding releases, this model can
no longer guarantee that an overrun handler will be fired when the thread exceeds the
budget per release (a property that is guaranteed by the current model in the RTSJ). This
fails to ensure the first property listed before, where the application should be informed
that an overrun has occurred.

For instance, one possible situation that can happen using this model is shown in
Figure 6.13. The scenario consists of a thread executing with a period of 10 time units,
a deadline of 5 time units, and a cost of 4 time units. In the example, the thread enters
in a cycle (executing for 3 time units, blocking and unblocking) and can execute in each
period without exceeding its budget per period. Though, the thread presents an errant
behaviour by not making a call to the \texttt{waitForNextRelease()} method and, therefore, not
finishing its first release. Because the current release is being considered in this run (which
exceeds its cost at time 12), the system is not able to inform the (faulty) behaviour to the
application since a cost overrun never occurs at time 12 (for the release) and no overrun
handler is ever released (although a deadline miss handler, if present, will be released).

![Figure 6.13: Problems with only considering a cost per period enforcement.](image)

So far two main enforcement attributes can be identified, based on per release or per
period mechanisms, which can be used to enforce the CPU budget for a thread. On
the one hand, using the per release mechanism guarantees that overruns are notified to the application – first property previously described. On the other hand, the per period mechanism enforces that errant threads cannot impact on other threads execution, maintaining the feasibility analysis – second property previously presented. Four possible combinations of these two attributes are possible. In the first approach no cost enforcement model is introduced during execution and the thread can execute freely. The second approach is the one found in the current version of the RTSJ. The approach discussed before is the third approach, where only per period based enforcement is added to the system. In the fourth approach a combination of both per release and period enforcements is found. This solution is consistent, in the sense that it combines strengths found in both second and third approaches. Indeed, this is the solution proposed in this research work.

The idea is to guarantee that an overrun handler will be released when the consumption exceeds the cost per release and that a thread becomes blocked-by-cost-overrun whenever it exceeds the budget per period (possibly releasing a handler as well). The current model of cost monitoring system of the RTSJ is extended by adding another consumption value, which is used to monitor the CPU consumption during the current period of the thread. When this consumption exceeds the cost in a given period, a cost overrun (due to the period) occurs and the thread becomes blocked-by-cost-overrun. In Figure 6.14 an example of the solution is presented. This shows the same run of Figure 6.12, where the anomaly was depicted. The difference is that, instead of continuing the execution up to time 18, the thread becomes blocked-by-cost-overrun at time 15 due to an overrun for the period. This guarantees that the behaviour is correctly enforced upon the thread.

Based on this approach for solving the anomalous behaviour, Figures 6.15 and 6.16 present a modified definition of the cost monitoring system defined in the RTSJ. The main modification is the addition of another counter to keep track of the CPU consumption value per period, which is renamed to consumption-period. Moreover, the current consumption
value used in the release is called consumption-release. Due to this addition it is needed to: (i) change some of the rules already defined by the RTSJ to cope with the consumption counter; (ii) add other rules to deal with this new counter. An explanation of the proposed definition follows.

| (a) Initial release: consumption-release and consumption-period are set to zero and will increase as the thread executes;  
| (b) Cost overrun: if consumption-release or consumption-period becomes equal or greater than cost – due to execution of the thread or change in the cost value – the overrun asynchronous event handler, if any, is released;  
| (c1) On the occurrence of a cost overrun (due to a release overrun):  
| (1) if the most recent release of the thread is the $i^{th}$ release and ((the $i^{th} + 1$ release event has not yet occurred) or (the $i^{th} + N$ release event has occurred and the $i^{th} + N$ cost monitoring phase has been started, where $N > 0$)):  
| (i) if the thread is executing or eligible-for-execution, then it becomes blocked-by-cost-overrun;  
| (ii) if the thread is blocked for a reason other than blocked-by-cost-overrun, then a pending for blocked-by-cost-overrun is set;  
| (2) else if the $i^{th} + N$ (where $N > 0$) release has occurred and the $i^{th} + M$ (where $N > M$) cost monitoring phase has not yet started, then the cost monitoring system starts its $i^{th} + M$ phase, CPU consumption-release is set to zero and the thread remains in its current state;  
| (c2) On the occurrence of a cost overrun (due to a period overrun):  
| (1) if the thread is executing or eligible-for-execution, then it becomes blocked-by-cost-overrun;  
| (2) if the thread is blocked for a reason other than blocked-by-cost-overrun, then a pending for blocked-by-cost-overrun is set;  

As presented in Figure 6.15, as soon as the thread starts its initial release the cost monitoring starts to monitor the CPU consumption (per release and per period) (a). If a cost overrun occurs (b), any associated cost overrun handler is released. When a cost
(d) **Occurrence of the $i^{th}$ release event for a thread:**

1. If the thread is blocked-by-cost-overrun, then the $i^{th}$ cost monitoring phase is started, the CPU consumption-release and the CPU consumption-period are set to zero, the thread is made eligible-for-execution;

2. If the thread is blocked for a reason other than blocked-by-cost-overrun:
   
   (i) If a pending for blocked-by-cost-overrun is set, then the pending is cleared, the $i^{th}$ cost monitoring phase is started, consumption-release and consumption-period are set to zero, the thread remains in its current state;

   (ii) Otherwise, the CPU consumption-period is set to zero;

3. Otherwise, consumption-period is set to zero;

(e) **Completion of the $i^{th}$ release for a thread:** if the $i^{th} + N$ (where $N > 0$) cost monitoring phase has not started, it is started and the CPU consumption-release is set to zero; otherwise no action is taken;

(f) **Changes to the cost value:**

1. If the old value was greater than consumption-release and consumption-period and if the:

   (i) New value is lower or equal to consumption-release and greater than consumption-period: a cost overrun due to a release overrun occurs;

   (ii) New value is lower or equal to consumption-release and consumption-period: a cost overrun due to a release and a period overrun occurs;

   (iii) New value is lower or equal to consumption-period and greater than consumption-release: a cost overrun due to a period overrun occurs;

2. If the new value is greater than consumption-period:

   (i) If the thread is block-by-cost-overrun, then the thread is made eligible-for-execution;

   (ii) If the thread is blocked and a pending for blocked-by-cost-overrun is set, then the pending is cleared;

   (iii) Otherwise, no cost monitoring actions occur;

(g) **Reset the cost monitoring:** if the most recent release being considered is the $m^{th}$ release and the thread is in the $n^{th}$ release ($n > m$), then a reset makes the cost monitoring start its $n^{th}$ phase (reset) and consumption-release and consumption-period are set to zero.

Figure 6.16: Requirements and definitions for the revised cost monitoring (part 2).

overrun in a release occurs, different actions may be triggered by the cost monitoring (c1). The semantics of the behaviour for a cost overrun in a period are shown in (c2).

As shown in Figure 6.16, on the occurrence of a new $i^{th}$ release event, the thread has its consumption-release and consumption-period values reset: becoming eligible-for-execution (d)(1); or having a pending for blocked-by-cost-overrun cleared (d)(2)(i). Otherwise, only the consumption-period is reset. When the thread completes the $i^{th}$ (current) release and the $i^{th} + N$ cost monitoring phase has not started, it is started and the consumption-release value is reset (e). Furthermore, the cost value of the thread being monitored can
be dynamically changed, triggering different actions by the cost monitoring (f), affecting both consumption-release and consumption-period values and triggering the respective overruns. Finally, if the cost monitoring is in the \( m^{th} \) phase and the thread is in the \( n^{th} \) release, where \( n > m \), the state of the cost monitoring is reset to consider the \( n^{th} \) release (g) – resetting both consumption-release and consumption-period counters.

From an implementation point of view, this model considers that only one type of handler is generated when an overrun occurs. By doing so it is possible to maintain the current interface proposed by the RTSJ. Nevertheless, in order to take appropriated measures, it is necessary for the developer to be able to distinguish what type of overrun triggered the handler. In this sense, the idea is to reflect in a special variable (representing the state of the thread) the last overrun state of the thread. According to the model shown in Figures 6.15 and 6.16, three different states are possible: (i) overrun in a release; (ii) overrun in a period; (ii) overrun in a release and period. Using this state the developer is able to take the correct measures in the system. Moreover, this approach introduces a lower overhead in the system because only one type of handler is ever released.

6.2.5.1 Changes to the Formal Model

Based on the modified version of the cost monitoring mechanism shown in Figures 6.15 and 6.16, modified a TA model, defined for the previous model of cost monitoring, which reflects the incorporated changes is presented. Only the component level of the CM subsystem is changed.

- **Automata for the revised CM subsystem at the component level:** Specify the interface between the change of cost calls at the application level and the CM subsystem. It also defines the main behaviour of the CM component. The interface is represented by the automaton CM_CI (CM Cost Interface) depicted in Figure 6.17 (a). This automaton
has three main transitions that are activated by the change of cost \((\text{changecost}\?)\) from the application level. Depending on the current state of the CM component (variable \textit{cm\_cs}), it may trigger different actions in the CM\_Main automaton (including overruns, see actions \textit{changeoverrun!} and \textit{changeoverrunpp!}). Also, it swaps values between the current cost (variable \textit{cost}) and the new cost (variable \textit{ncost}). The use of this technique of swapping values allows for the verification of all possible interleaving executions where the old and new cost values are different from each other.

Figure 6.17 (b) shows the automaton CM\_Main, describing the main behaviour of the CM component. It specifies the actions triggered by the cost monitoring in the occurrence of cost overruns and also resets the CPU consumption-release and consumption-period to zero. Two integer variables are inserted to keep track of the current CPU consumption-release (variable \textit{cpr}) and consumption-period (variable \textit{cpp}) in order to detect the overruns. Since it is not possible to store the value of a clock using the TA formalism, the automaton CM\_Counter shown in Figure 6.17 (c) is defined. This automaton counts, every one time unit (this suffices to capture the thread’s execution profile), the CPU consumption-release/period (\textit{cpr} and \textit{cpp}) and generates overruns (\textit{overrun!} and \textit{overrunpp!}) when the current CPU consumption equals the current cost set for the thread.

The CM\_Main automaton starts by monitoring the thread that was initially released (\textit{monitor?}) whilst counting the CPU consumption (\textit{start!}). During execution, when the thread becomes blocked (\textit{rtt\_block?}), the cost monitoring stops counting the CPU consumption value of the thread. If the thread changes from a blocked state to executing (\textit{rtt\_unblock?}), the cost monitoring starts counting the CPU consumption again. If a cost overrun in a release or period occurs and the thread is blocked (\textit{changeoverrun?} or \textit{changeoverrunpp?}), a blocked-by-cost-overrun is left pending. Either when a new release event (\textit{monitor?}) or a change to the cost value (if the new cost is greater than the current CPU consumption) occurs (\textit{clearpbco?}), the thread has the blocked-by-cost-overrun
Figure 6.17: Revised (a) CM_CI, (b) CM_Main and (c) CM_Counter automata.
pending cleared (cm\_clearbbco!) and considers the thread to be in the blocked state. Otherwise, if the thread changes from executing straight away, it becomes blocked-by-cost-overrun (rtt\_unblock?).

When the thread is executing and a cost overrun in a release occurs (due either to consumption-release (overrun?) or change in the cost value (changeoverrun?)) and the release being monitored is the \textit{i}^{th} release (cor == 1), the thread becomes blocked-by-cost-overrun. The same transitions can be triggered during a cost overrun in a period (channels overrunpp? and changeoverrunpp?). Otherwise, the cost monitoring enters its \textit{i}^{th} + 1 phase (cor > 1), resetting the consumption-release value and not changing the state of the thread. If a thread is blocked-by-cost-overrun, it can change to the executing state when its cost changes to a value greater than the current CPU consumption-release (makeefe?) or a new release occurs for the thread (monitor?). Note that, whenever a new release occurs (monitor?), the CPU consumption-period is reset (cpp := 0). When a thread completes a release, if the release being monitored is the \textit{i}^{th} release and the current release for the thread is the \textit{i}^{th} release (rtl\_completed?), the thread becomes blocked-for-release-event. Otherwise (rtl\_completion?), the number of outstanding releases is decremented (variable cor), the consumption-release value is reset, and the thread continues its execution. If the thread is descheduled (rtl\_descheduled?), it can eventually become scheduled again, where the cost monitoring is reset (rtl\_reset?).

\textbf{• Variables for verification purposes:} In the automaton shown in Figure 6.17 three variables are used for verification purposes. Besides using the two variables \textit{tpr} (time per release) and \textit{oh} (overrun handler) defined in the previous model, a new variable \textit{ohpp} (overrun handler per period) is introduced. This variable is set to \textit{true} on the occurrence of a cost overrun in a period, being set to \textit{false} when a new period starts or the cost for the period is increased.
6.2.5.2 A New Formal Evaluation

In order to guarantee that the proposed cost monitoring and enforcement model does correctly solve the encountered anomalous behaviour, in this section the properties specified previously are verified against the new model. Furthermore, two new properties about the consistency of the model, with respect to the period enforcement are presented. The verification scenarios used in the analysis are the same ones utilized in the previous analysis.

Model consistency properties

The same properties $c_1$ to $c_6$ used to analysed the model consistency for the previous model were analysed again, having the same true results as before. Moreover, two other properties are specified for the model, related to the introduction of the consumption-period variable.

In the first added property, it is specified that in the occurrence of a cost overrun per period, when the thread is in the executing state, its behaviour is enforced (the thread becomes blocked-by-cost-overrun).

**Property c7.** *If a thread is in the executing state, on the occurrence of a cost overrun (where the consumption-period value is greater than or equal to the cost) the cost monitoring always puts the thread in the blocked-by-cost-overrun state, irrespective of whether the $i^{th} + N$ release event has already occurred or not.*

Firstly, it is defined that, in the left-side of the liveness formula, the thread is in the executing state ($CM_{\text{Main.exec}}$). Also, it is necessary to characterise the occurrence of a cost overrun per period at the left-side of the formula. According to the behaviour of the model, two more propositions are defined. The first concerns the consumption-period exceeding the cost ($cpp \geq cost$) and the second is the release of an overrun handler due to a cost overrun per period ($ohpp == true$). The generation of an over-
run handler per period is defined because it may be possible, in certain executions, that consumption-period becomes greater than cost but as soon as this happens, the thread may increase its cost and continue the execution. Using the generation of an overrun enables the exact characterisation of an overrun per period in the model. At the right-side of the formula it is defined the thread behaviour being enforced to blocked-by-cost-overrun (RTT_Main.bbco). Putting the propositions together yields the following liveness formula 

\[(\text{CM}_\text{Main}.\text{exec}) \land (\text{cpp} \geq \text{cost}) \land (\text{ohpp} == \text{true}) \Rightarrow (\text{RTT}_\text{Main}.\text{bbco}).\]

This property analyses the behaviour upon cost overruns when the thread is in the executing state. In the next property, the idea is to analyse the behaviour of cost overruns when the thread is in the blocked state. As before, it should be noted that even if a thread is not using the CPU, an overrun can happen. Since the application may change the cost to a lower value and, therefore, trigger an overrun. It is important to check that in the occurrence of a cost overrun per period, when the thread is in the blocked state, its behaviour is enforced (a pending for blocked-by-cost-overrun is set).

Property c8. If a thread is in the blocked state, on the occurrence of a cost overrun (where the consumption-period value is greater than or equal to the cost) the cost monitoring sets a pending-by-cost-overrun in the thread, irrespective of whether the \(i^{th} + N\) release event has already occurred or not.

For this property, in the left-side of the liveness formula, it is defined that the thread is in the blocked state (\(\text{CM}_\text{Main}.\text{blk}\)). The occurrence of a cost overrun per period at the left-side of the formula is characterised using \((\text{cpp} \geq \text{cost}) \land (\text{ohpp} == \text{true})\). Then, at the right-side of the formula the pending for blocked-by-cost-overrun is set for the thread (RTT_Main.pbbco). Putting the propositions together yields the following liveness formula 

\[(\text{CM}_\text{Main}.\text{blk}) \land (\text{cpp} \geq \text{cost}) \land (\text{ohpp} == \text{true}) \Rightarrow (\text{RTT}_\text{Main}.\text{bbco}).\]

In summary, the previous specified properties, when analysed had a true result. With respect to the proposed cost monitoring and enforcement, two new properties c7 and c8
are specified. They addressed the behaviour triggered by cost overruns due to period overruns. These new properties had a true result on the verification runs, so that it is possible to state that the model is coherent and consistent with the definitions found in the RTSJ and also the proposal.

**Model evaluation properties**

The same properties $e_1$ to $e_3$ used to evaluate the previous model were analysed again using the proposed model, having the same true (for properties $e_1$ and $e_3$) and false (for property $e_2$) results as before. However, property $e_4$, which specified that the maximum allowed budget must not be violated in a given period, had a true result against the new model. This guarantees that the previous encountered anomaly in the cost monitoring of the RTSJ does not happen in the proposed model of this research work.

**Verification results**

The same verification environment (computer and UPPAAL tool) used for the analysis of the previous models is used in the verification of the new model. Table 6.3 shows the verification results for the properties against the model interacting with the PRTT subsystem. Scenario 1 is the scenario with the deadline time set to five time units. Scenario 2 is the scenario with the deadline time set to ten time units (equal to the period). In particular, the property $e_4$ is verified for both scenarios 1 and 2, but with the cost value fixed. As it can be seen, both memory requirements and time used are relatively low. Indeed, increasing the size of the number of releases (constant NR) did not influence so much in the results, as the UPPAAL tool handled very well the verification of the model.

The results for the properties verified against the model interacting with the ASRRTT subsystem are presented in Table 6.4. Scenario 1 is the scenario for sporadic real-time threads. The first variation 1a of the scenario uses the IGNORE policy, whereas the
Table 6.3: Properties verified against the model with the PRTT subsystem.

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario</th>
<th>Memory (in Megabytes)</th>
<th>Time (in seconds:milliseconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>1</td>
<td>≈ 1472.6</td>
<td>≈ 401:51</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1147.4</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>c2</td>
<td>1</td>
<td>≈ 1472.6</td>
<td>≈ 400:85</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1147.4</td>
<td>≈ 362:10</td>
<td>true</td>
</tr>
<tr>
<td>c3</td>
<td>1</td>
<td>≈ 1518.6</td>
<td>≈ 510:55</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1173.3</td>
<td>≈ 383:81</td>
<td>true</td>
</tr>
<tr>
<td>c4</td>
<td>1</td>
<td>≈ 1532.1</td>
<td>≈ 607:09</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1180</td>
<td>≈ 346:78</td>
<td>true</td>
</tr>
<tr>
<td>c5</td>
<td>1</td>
<td>≈ 1483.1</td>
<td>≈ 715:82</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1157.9</td>
<td>≈ 409:78</td>
<td>true</td>
</tr>
<tr>
<td>c6</td>
<td>1</td>
<td>≈ 1494.4</td>
<td>≈ 459:73</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1169.1</td>
<td>≈ 330:26</td>
<td>true</td>
</tr>
<tr>
<td>c7</td>
<td>1</td>
<td>≈ 1472.9</td>
<td>≈ 518:16</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1147.7</td>
<td>≈ 392:54</td>
<td>true</td>
</tr>
<tr>
<td>c8</td>
<td>1</td>
<td>≈ 1472.9</td>
<td>≈ 555:93</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1147.7</td>
<td>≈ 363:84</td>
<td>true</td>
</tr>
<tr>
<td>c9</td>
<td>1</td>
<td>≈ 1473.9</td>
<td>≈ 852:39</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1148.3</td>
<td>≈ 562:51</td>
<td>true</td>
</tr>
<tr>
<td>e2</td>
<td>1</td>
<td>≈ 1.5</td>
<td>≈ 00:11</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1.5</td>
<td>≈ 00:10</td>
<td>false</td>
</tr>
<tr>
<td>e3</td>
<td>1</td>
<td>≈ 1795.7</td>
<td>≈ 179:28</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 1411.1</td>
<td>≈ 135:13</td>
<td>true</td>
</tr>
<tr>
<td>e4</td>
<td>1</td>
<td>≈ 369.7</td>
<td>≈ 72:19</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>≈ 263</td>
<td>≈ 47:69</td>
<td>true</td>
</tr>
</tbody>
</table>

second variation 2b, uses the REPLACE policy. As before, the property e4 is verified for scenario 1 and its variations, but with the cost value fixed.

### 6.3 Monitoring MIT Violations

The RTSJ already integrates a mechanism for detecting MIT violations for sporadic real-time threads. Indeed, in Section 5.4 of the previous chapter, a formal model of aperiodic and sporadic real-time threads was defined. During the definition of the formal model, the notion of arrival queue in the RTSJ was depicted. Together with the arrival queue, the developer can also set a specific policy towards the overflow of this queue (in the case of aperiodic threads) or the violation of MIT constraints (for sporadic threads). Reviewing, in order to deal with MIT violations to the arrival queue of a sporadic thread, the RTSJ provides the following policies:
6.3 Monitoring MIT Violations

Table 6.4: Properties verified against the model with the ASRTT subsystem.

<table>
<thead>
<tr>
<th>Property</th>
<th>Scenario</th>
<th>Memory (in Megabytes)</th>
<th>Time (in seconds:milliseconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>1a</td>
<td>≈ 1728.1</td>
<td>≈ 402:00</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1728.1</td>
<td>≈ 398:44</td>
<td>true</td>
</tr>
<tr>
<td>c2</td>
<td>1a</td>
<td>≈ 1728.1</td>
<td>≈ 398:44</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1728.1</td>
<td>≈ 388:13</td>
<td>true</td>
</tr>
<tr>
<td>c3</td>
<td>1a</td>
<td>≈ 1767.9</td>
<td>≈ 405:28</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1768</td>
<td>≈ 394:20</td>
<td>true</td>
</tr>
<tr>
<td>c4</td>
<td>1a</td>
<td>≈ 1768.2</td>
<td>≈ 401:20</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1768.2</td>
<td>≈ 396:97</td>
<td>true</td>
</tr>
<tr>
<td>c5</td>
<td>1a</td>
<td>≈ 1748.4</td>
<td>≈ 399:23</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1748.4</td>
<td>≈ 396:16</td>
<td>true</td>
</tr>
<tr>
<td>c6</td>
<td>1a</td>
<td>≈ 1750.5</td>
<td>≈ 399:83</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1750.5</td>
<td>≈ 399:86</td>
<td>true</td>
</tr>
<tr>
<td>c7</td>
<td>1a</td>
<td>≈ 1728.6</td>
<td>≈ 398:40</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1728.4</td>
<td>≈ 402:29</td>
<td>true</td>
</tr>
<tr>
<td>c8</td>
<td>1a</td>
<td>≈ 1728.4</td>
<td>≈ 398:41</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1728.4</td>
<td>≈ 391:12</td>
<td>true</td>
</tr>
<tr>
<td>e1</td>
<td>1a</td>
<td>≈ 1728.1</td>
<td>≈ 707:34</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 1728.1</td>
<td>≈ 692:85</td>
<td>true</td>
</tr>
<tr>
<td>e2</td>
<td>1a</td>
<td>≈ 26.6</td>
<td>≈ 01:80</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 26.6</td>
<td>≈ 01:80</td>
<td>false</td>
</tr>
<tr>
<td>e3</td>
<td>1a</td>
<td>≈ 2237.4</td>
<td>≈ 171:29</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 2237.4</td>
<td>≈ 172:39</td>
<td>true</td>
</tr>
<tr>
<td>e4</td>
<td>1a</td>
<td>≈ 664.9</td>
<td>≈ 110:26</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>1b</td>
<td>≈ 664.9</td>
<td>≈ 103:06</td>
<td>true</td>
</tr>
</tbody>
</table>

- **mitViolationIgnore**: the violation is ignored, silently;

- **mitViolationExcept**: an exception of type *MITViolationException* is thrown by the system;

- **mitViolationReplace**: this policy depends on the current state of the arrival queue. That is, if the last release event has not completed and its deadline has not been missed, the arrival time for this particular release event is replaced with the time of this new arrival. In any other condition, this policy simply ignores the new arrival violating the MIT (behaving like the *mitViolationIgnore* policy);

- **mitViolationSave**: once this policy is set, the system is said to be under an execution time regulation policy. This happens because all arrivals are accepted, but still the arrival times have to be constrained to the thread’s MIT. Therefore, if an arrival
comes before the MIT, it will be set to have arrived when the next MIT is due. Note, however, that the deadline for the particular arrival is related to the original arrival time, and not the constrained one.

The particular policy towards the arrival queue of a sporadic thread is set using the SporadicParameters class (shown in the previous Figure 6.2). Moreover, once a new policy is set, its effect in the system takes place immediately upon the next arrival.

6.3.1 Support for Aperiodic and Sporadic Real-Time Threads

In order to limit the amount of time (budget) dedicated to a sporadic or aperiodic thread in the system in a given period of time, several forms of servers can be found in the literature (see Chapter 2). The RTSJ introduces the notion of ProcessingGroupParameters, where a specific budget for a given period is shared between a group of real-time threads. A ProcessingGroupParameters object has six attributes: (i) start time; (ii) the replenishment period; (iii) the execution time budget; (iv) the deadline; (v) an execution time overrun handler; (vi) and a deadline miss handler. These attributes basically amount for the definition of a periodic real-time thread.

The execution time budget allocated to the ProcessingGroupParameters object is only consumed when one of the threads associated to it starts executing the CPU. At this point, the budget starts decreasing and only stops if: (a) the thread finishes its execution; (b) the budget finishes. In case (a), when the thread finishes, it may be possible that another thread associated to the ProcessingGroupParameters object is eligible for execution. Therefore, it could start executing and the execution time budget starts decreasing from where it stopped earlier. In the second case (b), no thread associated to the ProcessingGroupParameters object is allowed to execute, a suspend behaviour is enforced. This behaviour is enforced until the next period of the ProcessingGroupParameters object is due, where the budget for the group is replenished.
Moreover, when a budget is exhausted, the associated execution time overrun handler is released. The handler can then be used to increase the budget for the group. If, for instance, a real-time thread associated to the `ProcessingGroupParameters` object is executing the CPU when the deadline is due, the associated deadline miss handler is released. The association of a `ProcessingGroupParameters` object can be done at the creation of the real-time thread, or using the `setProcessingGroupParameters()` method provided for real-time threads in the RTSJ.

#### 6.4 Blocking Time Monitoring

The term blocking time monitoring refers to the process of tracking the amount of CPU consumption that a higher priority real-time thread is not allowed to execute whilst a lower priority thread is executing the processor. This research proposed the introduction of blocking time clocks and timers to the POSIX/C standard in Section 4.4. In this section, a monitoring mechanism for blocking time in the RTSJ is proposed. From the developers point of view, there are mainly two functionalities to be provided by a blocking time monitoring model in the RTSJ. Firstly, the addition of methods in the RTSJ that allows getting the amount of blocking time already imposed on a thread. This can provide a mechanism for developers to quickly obtain data relative to the maximum blocking times that the threads in the system are subject to. Secondly is the provision of a high-level mechanism for the detection of blocking time violations, which takes actions whenever a blocking time violation occurs. Indeed, this approach is similar to that of cost monitoring and enforcement described in Section 6.2. However, differently from cost enforcement, when dealing with shared resources, the idea is not to stop threads from executing, but rather provide a way to execute an asynchronous event handler to deal with the timing error (the violation of the estimated blocking time for the thread). This happens because
stopping a thread inside a shared resource could potentially stop important threads in the system, which are waiting to enter in the shared resource. Another case also occurs if a thread inside a shared resource stops and others threads are allowed to enter, therefore leaving the critical section in an inconsistent state.

The blocking time monitoring mechanism for the RTSJ proposed in this research has two objectives: (a) monitor the violations of the expected blocking times for the threads in the system; (b) profile data related to the minimum, average and maximum blocking times for the threads in the system, which can then be obtained by the developer. The blocking time monitoring mechanism following these objectives is presented in Figure 6.18. The mechanism, associated with each thread, monitors in phases. The phases are related to the current release of the real-time thread. This means that the system only considers a new release when the thread finishes the current release, i.e., it calls the `waitForNextRelease()` method.

(a) **On initial release**: a new blocking time monitoring phase is started, the blocking time consumption is set to zero (and will increase as the thread is subject to blocking by lower priority threads);
(b) **Blocking violation** occurs when the current blocking consumption becomes equal or greater than the blocking value; the blocking violation asynchronous event handler, if any, is released;
(c) **On the occurrence of a blocking violation**: no action is taken upon the behaviour of the thread. It is up to the released asynchronous event handler to do that;
(d) **On occurrence of the \(i^{th}\) release event for a thread**: no blocking time monitoring actions occurs;
(e) **On completion of the \(i^{th}\) release for a thread**: the \(i^{th} + 1\) blocking time monitoring phase is started and the blocking consumption is set to zero. Information about the minimum, average and maximum blocking times for the previous releases of the thread are gathered;
(f) **On changes to the blocking value**:
   (1) if the new value is lower or equal to the current blocking consumption and the old value was greater than the current blocking consumption, then a blocking violation occurs;
   (2) if the new value is greater than the current blocking consumption, the blocking value is updated and no further action is taken.

Figure 6.18: Requirements and definitions for blocking time monitoring.

In Figure 6.18 the definitions and requirements for the blocking time monitoring mechanism are presented. As soon as the thread starts its initial release, the blocking time monitoring sets to 0 the blocking time consumption (a). If a blocking time violation oc-
curs ((b) and (c)), any associated blocking time violation event handler is released. On the occurrence of the \(i^{th}\) release event no action is taken (d). When the thread completes the \(i^{th}\) current release (e), a new \(i^{th} + 1\) blocking time monitoring phase is started to monitor the new \(i^{th} + 1\) release of the thread that will occur. Moreover, when the thread completes its release (e), information about the minimum, average and maximum blocking times for the past releases is gathered. The blocking time value of the thread being monitored can be dynamically changed, triggering different actions by the blocking time monitoring mechanism (f).

Different from the cost monitoring and enforcement model of the RTSJ, where a formal model was built and analysed, a formal model of the blocking time monitoring mechanism proposed for the RTSJ is not provided. The reasons for this are twofold. Firstly, the blocking time monitoring model has a lower complexity than the cost monitoring and enforcement model. Mainly because the cost monitoring model has to confine errant threads, so that they cannot affect others in the system. Differently, the blocking time monitoring model aims to alert the system (the threads that are being blocked) and not enforce any behaviour, since the blocking thread is inside a shared resource. Secondly, as it will be described in the next chapter, the implementation of the monitoring mechanism presented in this section is done at a lower level using the blocking time monitoring algorithm depicted in Section 4.4. Therefore, its correctness has already been provided in Theorems 4.1 and 4.2.

### 6.5 Summary

This chapter has presented in more details the timing error detection model for the RTSJ. According to the error detection model, four different types of timing errors could develop during the execution of the system. This chapter then discussed the RTSJ support for
detecting them. The integrated deadline monitoring approach for the RTSJ was first presented. In this model, the deadline associated to a particular thread is defined during its creation or by changing the release parameters of the thread. Whenever a deadline is missed, the system generates a handler (if one is available).

Secondly, the cost monitoring and enforcement model provided by the RTSJ was considered. With the definition of an enforcement model, the RTSJ goes beyond the provision of monitoring capabilities, guaranteeing that a thread that violates its maximum execution time is stopped and does not impact on the execution of lower priority threads. Nevertheless, this model is considered an optional facility in the RTSJ and is not thoroughly tested. Therefore, a formal model of the RTSJ cost monitoring approach was defined and analysed. During the analysis, it was shown that the enforcement model was inconsistent, because in some rather specific scenarios it is possible for a thread to execute more time than expected in a given period. This can potentially impact in the execution of lower priority threads. Based on these results, a revised model was presented in this chapter. This model corrected the anomaly behaviour found in the original model.

Moreover, with respect to the enforcement model, which says that a thread should suspend its execution, it is important to note that other softer approaches could be used. For instance, the thread could have its priority lowered so that it could still run, after all threads in the system finish their execution. This idea is discussed in the next chapter, where the implementation of the RTSJ cost monitoring and enforcement approach is provided.

Thirdly, the RTSJ approach for handling MIT violations was described, together with the support given for executing sporadic and aperiodic threads using processing group parameters. Similar to the other timing error detection mechanisms of the RTSJ, the detection of MIT violations is also associated to real-time threads, having pre-defined behaviours taken upon violations. In fact, this model was described in more details in the
previous chapter, where the formal model of aperiodic and sporadic real-time threads was presented.

Fourthly, the last error detection mechanism considered in this chapter was related to the execution overrun of the WCET of a thread inside a shared resource. The RTSJ did not provide any support for monitoring and detecting the overruns, as the cost monitoring mechanism of the RTSJ cannot be set to only measure the time a thread executes inside a shared resource. This way, a model that enables to monitor and detect overruns, releasing event handlers to deal with the timing errors, was defined. In addition, the proposed model also provided means to profile data related to blocking time, considering the minimum, average and maximum blocking times for the previous releases of a real-time thread.
So far, in this thesis, different error detection models for the POSIX/C standard and the RTSJ have been discussed. The models focused on the four different timing errors assumed in this work. With respect to the POSIX/C standard, the main contribution of the thesis is the definition of an algorithm, which provides the basis to implement the notion of blocking-time clocks and timers. In the RTSJ, the main contributions include: (a) the identification of an anomaly in the enforcement model of the cost monitoring mechanism of the RTSJ, followed by the proposal of a revised model to solve the anomaly; (b) the definition of a blocking time monitoring mechanism to alert the system on the occurrence of blocking time violations. This chapter aims to provide a prototype implementation of these mechanisms. With their implementations, an evaluation of the overhead incurred in the system when using the mechanisms is obtained.

Initially, the run time execution environment used in the implementation of the mechanisms is considered in the next section. This run time execution environment is based on
the open-source softwares MaRTE OS [59, 7] and jRate [31, 6]. In Section 7.2, the imple-
mentation and evaluation of blocking-time clocks and timers in the POSIX/C standard is
presented. This is followed by the implementation and evaluation of the cost monitoring
mechanisms (Section 7.3) and the proposed blocking time monitoring mechanism for the
RTSJ (Section 7.4).

7.1 The Run Time Environment

In order to implement the proposed timing error detection models, a run time environment
is provided. When defining this environment, a basic requirement was specified:

*The run time environment shall be solely based on open-source implement-
ations of both POSIX/C and the RTSJ. This provides a platform that can
be modified and extended, without restriction, by the author or any other re-
searcher.*

Because several open-source implementations of POSIX/C can be found in the lit-
erature, the focus was to find an open-source implementation of the RTSJ. Indeed, few
open-source implementations of the RTSJ can be found in the literature, the main ones
being jRate [31, 6] and the OVM Virtual Machine [52, 8]. From these implementations,
jRate was the chosen one to be used in this work. This comes from the fact that jRate: (a)
has an implementation easier to understand and modify when compared to OVM (used to
build virtual machines); (b) has been previously used by the real-time research group at
York [30]. Nevertheless, jRate requires running on top of a Real-Time Operating System
(RTOS), so that real-time capabilities can be achieved.

Previous work carried out by Mario Aldea Rivas has ported jRate to the open-source
RTOS MaRTE OS, running on top of a Linux distribution [60]. With such an implementa-
tion, only an emulated version of real-time applications using jRate can be accomplished,
since the compiled system executes as a stand-alone application in the Linux distribution. Nevertheless, one of the main advantages of using this implementation is that MaRTE OS already provides an implementation of the POSIX/C standard, which can be further extended (especially for the implementation of blocking-time clocks and timers defined in this thesis). Indeed, this is the basic run time environment selected to be used in this thesis.

Unfortunately, as it stands, this run time environment is not sufficient for the implementation and evaluation of the proposed error detection algorithms. This is because the statistical data gathered in the evaluation process only equates to the emulation of the system, which is subject to several interferences coming from the Linux distribution. Therefore, one of objectives achieved in this work was to port this basic run time environment to execute in a bare machine, instead of using the Linux distribution. In the next section, the jRate environment is presented in more details. Section 7.1.2 describes MaRTE OS. In the last section, the basic run time environment of jRate and MaRTE OS, running on top of the Linux distribution, is shown. Also, the modifications carried out in this thesis to enable the execution of the run time environment on top of a bare machine are detailed.

7.1.1 jRate

jRate [31, 6] is an open-source implementation of the RTSJ. It is an extension to the GNU Compiler for Java (GCJ) [2], providing functions necessary for the execution of RTSJ-compliant applications. jRate is an ahead-of-time middleware, where the RTSJ application is not interpreted by a Virtual Machine (VM). Instead, the application is compiled into native code and then executed. The jRate architecture is depicted in Figure 7.1.

As shown in Figure 7.1, an application compiled using jRate needs an RTOS to properly execute and meet its timing requirements. Currently, jRate has been tested in several
7.1 The Run Time Environment

Figure 7.1: The jRate architecture (taken from [31]).

Linux implementations, which are not RTOSs. The following RTSJ functionalities are implemented in jRate’s Version 0.3.4 (used in this research) [31]:

- memory areas: support for scoped memory (both linear and variable time) and immortal memory;

- real-time threads and scheduling: `RealtimeThread` support with a priority pre-emptive scheduler;

- asynchrony: asynchronous event handling is supported, using priorities as the execution eligibility;

- high resolution time and clock: resolution from nanoseconds up to microseconds can be obtained, depending on the underlying hardware and RTOS;

- timers: provide support for both periodic and one-shot timers.

Though, some of these features are limited due to the run time environment provided in this thesis. These limitations are described later, in Section 7.1.3.1.

7.1.2 MaRTE OS

MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) [59, 7] is an open-source real-time kernel for embedded applications. Its implementation extends the
Minimal Real-Time POSIX.13 subset [54] with execution-time clocks and timers. When designing MaRTE OS, the main requirements formulated were [59]: (i) target applications with system resources known at compile time; (ii) services with bounded response time; (iii) no protection boundaries between application and kernel, for efficiency purposes; (iv) multi-platform, using cross-development tools; (v) multi-language, adding support for Ada and C. Currently, MaRTE OS runs on a Personal Computer (PC) platform. The architecture of MaRTE OS, for the execution of C applications, is presented in Figure 7.2.

As shown in Figure 7.2, MaRTE OS provides header files for the application to access a standard C library and the MaRTE OS kernel. Device drivers can also be added to the kernel. Note that, the abstract hardware interface provides a clean interface between the kernel and the hardware. In fact, as presented in [49], by modifying this layer it is possible to run MaRTE OS (as a library) under a Linux distribution. This way, it is possible to test the system in an emulation environment and, later, run in a bare machine. This architecture, for C applications, is shown in Figure 7.3. Differently from the previous architecture, when running MaRTE OS in Linux, the application can access several libraries provided by the Linux system, and not supported by MaRTE OS.

Following an extended POSIX.13 minimal profile, the features supported by MaRTE OS include [59]:

![Figure 7.2: Original MaRTE architecture for C applications (taken from [59]).]
7.1 The Run Time Environment

![MaRTE Architecture Diagram]

Figure 7.3: MaRTE architecture for C applications under Linux (taken from [60]).

- thread management: POSIX thread creation, finalization and attributes. Since it targets the implementation of embedded systems, the number of threads and stack sizes are configurable;

- scheduling: support FIFO (First-In First-Out) and Round-Robin priority scheduling;

- mutexes and condition variables: implements both Priority Inheritance and Priority Ceiling (or Priority Protection), and also condition variables for wait synchronizations;

- signals: supports basic event notification;

- clocks: implements POSIX real-time clocks;

- execution-time clocks and timers: supports execution time budgeting;

- console I/O: provides support for I/O using PC console. Though, I/O drivers can be added;

- time services: implements thread suspension with absolute and relative high-resolution delays;

- dynamic memory management: supports features for dynamic memory management.
The implementation of blocking-time clocks and timers defined in this thesis can benefit from the provision of execution-time clocks and timers, found in MaRTE OS, as the basic infrastructure for handling events and the interface for the functions are already defined. Moreover, most of the described features are used in order to give support for the execution of jRate on top of MaRTE OS.

7.1.3 jRate on Top of MaRTE OS

In previous work [60], Mario Aldea Rivas provided a port of jRate to run on top of MaRTE OS. This port only considered the version of MaRTE OS that runs as a library under a Linux distribution. This leads to a basic run time environment that is only capable of emulating the application, as the system is subject to any interference that might occur in the underlying Linux distribution. The main ideas of this port, which can be found at [7], provided by Mario Aldea Rivas are now described. In the next subsection, the changes carried out in this thesis to remove the underlying Linux distribution of the port, allowing for the execution of the run time environment on a bare machine, are presented.

The first step in the port is to compile jRate using the header files for the POSIX library found in MaRTE OS. Since jRate is an extension of GCJ with several features available, different libraries also have to be provided by the underlying support system, in order to achieve the correct compilation. Unfortunately, MaRTE OS’s POSIX library is not as bountiful as the Linux library. Not providing all the libraries needed for the correct compilation. It is at this point, that the use of MaRTE OS running as a library in Linux is important, because the compilation process of jRate can use all the necessary libraries of the Linux distribution. This architecture was depicted in the previous Figure 7.3, but in terms of a C application.

The second main modification is to insert some specific MaRTE OS code into the jRate and GCJ run time system in order to create and handle the threads using MaRTE OS,
instead of the Linux distribution. The compilation process for the application is then specified. Firstly, the system produces object files of the Java application, using the jRate compiler. This is then linked together with the jRate library and the MaRTe OS POSIX library. The result is an executable Linux application.

From the RTSJ’s point-of-view, the main features provided by this port include periodic real-time threads, asynchronous event handlers and timers. Albeit not following the definitions provided by the RTSJ, a basic version of the cost monitoring and enforcement model of the RTSJ was also implemented. One of the main restrictions of this port was that priority inheritance protocols were not supported. This restriction is removed in the new version of the port provided in this thesis.

7.1.3.1 The Bare Machine Port

Essentially, four steps are followed in order to port jRate to the MaRTe OS version running in a bare machine. Firstly, it is necessary to make sure that the functionalities not found in MaRTe OS are not used in the compilation of the jRate run time system (or middleware). This is accomplished by modifying both jRate’s and GCJ’s source code and configuration files. In the second step, the run time configurations for MaRTe OS (running as a library in Linux) already in use had to be ported to the MaRTe OS version running in a bare machine. These included the increase of the memory size allocated during execution for jRate and the maximum number of threads allowed for creation in MaRTe OS.

The third step involved the modification of certain MaRTe OS functions, which enables the jRate middleware to run on MaRTe OS. Indeed, in this step certain code had to be inserted into MaRTe OS so that constructors and destructors used in GCJ are correctly allocated in the generated code. The final step relates to the proper linkage of the application with the middleware and RTOS in order to provide a executable form that can be executed in a PC (x86 architecture). In this step, the previous compilation
process used in the library version of the port was modified in such a way that the correct libraries were used. Moreover, the environment provided in this thesis (in both library and bare machine versions) was extended in such a way that the priority inheritance protocols available in MaRTE OS can be supported in a Java application, via modifications to the jRate middleware.

7.2 POSIX/C Blocking-Time Clocks and Timers

The concepts of blocking-time clocks and timers for the POSIX/C standard were described in Chapter 4. This section aims to provide details about the implementation of the concepts in MaRTE OS and the overhead incurred when using this monitoring mechanism during run time.

7.2.1 Implementation

The basic idea followed in the implementation of blocking-time clocks and timers in MaRTE OS is to extend the mechanism already found for execution-time clocks and timers. Therefore, for implementing the idea of blocking-time clocks and timers in MaRTE OS, it is necessary to: (i) introduce the monitoring algorithm shown in Chapter 4 (Section 4.4) at specific points in the system; (ii) add new fields to the thread’s Task Control Block (TCB), which are necessary for the execution of the monitoring algorithm; (iii) extend the signalling model to deal with blocking-time timers; (iv) change basic functions, related to clocks and timers, to take into consideration the notion of blocking-time clocks and timers.

According to algorithm in Section 4.4, the monitoring code has to be introduced at every point in the system where: a thread is added to the system’s run-queue; and a new thread is selected for execution (a context switch occurs). This is not straightforward in MaRTE OS, because the run-queue in MaRTE OS only includes threads that are ready
to run, excluding information of threads that are current blocked in the system (this information is held by the shared resource itself). To solve this problem, a new queue is added to MaRTE OS, namely the *blocked_queue*, which holds information about the threads that are currently blocked in the system.

The algorithm is introduced into the scheduling procedure of MaRTE OS (called *Do_Scheduling*). This tackles both the case where a new thread is selected for execution (a context switch occurs) and the case where a thread is added to the run-queue and not selected for execution. In fact, the latter case is only because the interruption handling procedure of MaRTE OS (called *Timer_Interrupt_Handler*), which handles the signals used for releasing (adding to the run-queue) previously suspended threads, always calls the *Do_Scheduling* procedure.

The execution of the algorithm needs certain fields associated with the threads of the system. The thread’s TCB in MaRTE OS is extended with the fields discussed in the Section 4.4: *Blk_Time_Start*, *Blk_Time_Max*, *Blk_Time_Total* and *Blk_Timer* (the timer field is substituted for a queue of timers, enabling the definition of more than one blocking-time timer for a specific thread). Also, according to the algorithm, two global variables representing the active and base priorities for the previous thread that was executing the processor are defined.

There is still another step to conclude the basic implementation of blocking-time clocks and timers in MaRTE OS. The extension of the signalling model to deal with the introduced blocking-time timers\(^1\). This particular step is not covered by the monitoring algorithm provided in Section 4.4, since it is tightly coupled to implementation aspects. On the occurrence of a timer signal in the system, the function *Extract_Expired_Event* is called. In this function it is decided if the generated signal came from other timers in

\(^1\)In particular, when using a Pentium processor, a timer in MaRTE OS is implemented using the timestamp counter. This requires only a single machine instruction, and is very fast [59].
the system (e.g. an execution-time timer), apart from blocking-time timers. If that is not
the case, the signal could be generated by a blocking-time timer. Therefore, the possible
range of priorities for threads that are being blocked (defined by the current thread) is
analysed, looking for expired timers.

Moreover, in order to provide a complete implementation of blocking-time clocks and
timers, the functions used by applications to interface with clocks and timers in MaRTE
OS have to be modified. This includes functions used for creating, setting and deleting
clocks and timers. Indeed, the following functions have to be modified: `clock_gettime`;
`clock_settime`; `timer_create`; `timer_gettime`; and `timer_settime`. Furthermore, a function
is defined for obtaining the id of a blocking-time clock for a particular thread, called
`pthread_getblkclockid()`, and a constant `CLOCK_THREAD_BLKTIME_ID`, which indicates
to the system (when a timer is created) that the blocking-time clock associated to the
current thread is to be used.

In order to correctly implement the interface, the id used by blocking-time clocks has to
be defined. Like execution-time clocks, blocking-time clocks are associated with a specific
thread. In MaRTE OS, the id for an execution-time clock matches the id of the respective
thread in the system. This results in a clean and efficient implementation of execution-
time clocks, but introduces a problem for extending it to blocking-time clocks, since the
ideal case is to also match the id of the blocking-time clock to the thread’s id. To solve
this problem with the least possible modifications to the kernel, the id of a blocking-time
clock is defined to have a negative value that, when multiplied by -1, always match the
corresponding thread id. It is important to ensure that, whenever information regarding
a blocking-time clock is needed inside the kernel, the correct conversion from a negative
(blocking-time clock id) to a positive (thread id) value is provided.
7.2.2 Overhead Evaluation

The notion of blocking time is directly associated with shared resources and, specifically for the purposes of the blocking time algorithm proposed in this thesis, both priority inheritance protocols used in POSIX/C. In order to evaluate the overhead introduced by the implementation of blocking-time clocks and timers, two different experiments are specified. The first experiment aims to measure the overheads of the protocol when using the PI protocol, whilst the second experiment focuses on the PP protocol.

**PI experiment.** The experiment for the PI protocol is defined as follows. There is only one shared resource amongst several threads. The lowest priority thread is released and enters the shared resource before any other thread in the system. One msec after it started executing, another thread (with a higher priority than its predecessor) is released by the system. Because the lowest priority thread is executing inside the shared resource, the released thread tries to enter the shared resource, becoming blocked and raising the active priority of the thread inside the shared resource. This periodic release of new threads continues until all threads in the system have been released (from lowest to highest priority). Moreover, the first released thread (with the lowest priority) does not finish executing inside the shared resource until all threads have been released. This is achieved by having the thread executing a loop whose condition only becomes false when the last thread (with the highest priority) tries to enter the shared resource. With this experiment, all threads in the system become blocked by the lowest priority thread. Hence, the maximum possible range of priorities for calculating the blocking time for the threads in the system is obtained.

Two time figures, relative to two distinct context switches, are obtained from the experiment. Firstly, the interval (context switch) where the highest priority thread in the system tries to enter the lock and the lower priority thread inside the shared resource
starts executing. This accounts for the time the algorithm starts monitoring the blocking time. Secondly, the interval (context switch) where the lowest priority thread is about to leave the shared resource and the highest priority thread enters the shared resource. This accounts for the time the algorithm stops monitoring the blocking time.

**PP experiment.** As in the previous experiment, in the experiment for the PP protocol only one shared resource is shared amongst several threads. The lowest priority thread is released and enters the shared resource before any other thread in the system. Once entering the shared resource, the active priority of the thread is raised to the ceiling of the lock. One msec after it started executing, another thread (with a higher priority than its predecessor) is released by the system. However, it is not allowed to execute, since the thread inside the shared resource now has the highest priority in the experiment. This periodic release of new threads continues until all threads in the system have been released (from lowest to highest priority). The first released thread does not finish executing inside the shared resource until all threads have been released. This is again achieved by having the thread executing a loop on the time where the last thread in the experiment is released. With this experiment, all threads in the system are eligible for execution. Hence, the maximum possible range of priorities for calculating the blocking time for the threads in the system is obtained.

As in the previous experiment, two time figures, relative to two distinct context switches, are obtained. Firstly, the interval (context switch) where the thread inside the shared resource yields the processor and the last released thread in the experiment starts executing (as it has the same priority of the shared resource ceiling). This accounts for the time the algorithm stops monitoring the blocking time. Secondly, the interval (context switch) where the last released thread yields the processor and the thread inside the shared resource starts executing again. This accounts for the time the algorithm starts
monitoring the blocking time.

**Results for the experiments.** The experiments were executed with a different number of threads, ranging from 2 (only 1 blocked thread) to 25 (24 blocked threads), in two distinct implementations, one with blocking-time clocks enabled and another without it. Each experiment was executed for 10000 times, so that the figures converge. The tests were carried out in a 1.6GHz Centrino T2050 Core Duo processor. Although a dual core processor was used, MaRTE OS only supports a uniprocessor architecture. Therefore, only one of the two processors is used during execution. Figures 7.4 and 7.5 show different graphics for the start and stop monitoring conditions of the protocol for experiments with the PI protocol and the PP protocol, respectively. The implementation without blocking-time clocks does not include the start and stop monitoring conditions of the protocol.

In the experiments with the PI protocol, the overhead for the start monitoring part of the protocol implementation increases on the average by 4.35% per each new thread added to the experiment. In comparison, the overhead for the implementation without blocking-time clocks is quite lower, on the average an increase of 0.57% per each new thread that is added to the experiment. The stop monitoring part of the protocol shows an average increase of 7.88% per new thread in the experiment with blocking-time clocks, compared to an average increase of 3.72% in the implementation without blocking-time clocks. The experiments with the PP protocol using blocking-time clocks show overheads on average of 4.55% (for the start monitoring part) and 4.17% (for the stop monitoring part) per each new added thread. Differently, the experiments with the PP protocol implementation without blocking-time clocks present no overhead increase. This is an expected result, as in the experiments with the PP protocol the system does not have to deal with threads that are entering and leaving the shared resource – the data is gathered by threads yielding the processor.
Figure 7.4: Overhead for different settings of the PI experiment in the POSIX/C.
Figure 7.5: Overhead for different settings of the PP experiment in the POSIX/C.
From these experiments, it is clear that the usage of blocking-time clocks may impact in the system’s overall performance in special cases where a large number of threads are being blocked by a lower priority thread. However, it is assumed that such cases are rare in a system – intuitively, developers will aim to design system where the priority difference between the threads sharing a resource is as close as possible. Therefore, the overhead caused by blocking-time clocks could be absorbed by the system without major issues. Nevertheless, this overhead can be reduced with more efficient implementations of blocking-time clocks. The overhead for the experiments in the PP protocol is quite lower than the overhead in the PI protocol. This comes from the fact that in the experiments with the PI protocol, the system has to add and remove threads from a blocked queue and also search this queue in addition to the system’s run-queue. This is a constant and costly operation, which could be improved.

For instance, the implementation of priority inheritance algorithms in the Shark OS [32] uses the concept of shadow tasks [32]. Using a shadow task, a high priority thread that becomes blocked in a shared resource is not removed from the run-queue of the system. Instead, a shadow field, which is incorporated in the thread’s structure, is set to point to the thread currently using the shared resource and that inherits the priority of the high priority thread. Once the high priority thread is selected for execution and the shadow field is not empty, the thread pointed by the shadow field is selected for execution. This mechanism could improve the efficiency of the implementation because: (i) there is no need to add and remove threads from a blocked queue; (ii) the only queue that the algorithm has to search for blocked threads is in the system’s run-queue.
7.3 Cost Monitoring in the RTSJ

In Chapter 6, the cost monitoring and enforcement mechanism of the RTSJ was formally analysed. A revised model was proposed to solve a possible anomaly that can occur, under certain scenarios, in the current model. This section provides details about the implementation of the cost monitoring mechanisms – for the one already defined in the RTSJ and the revised one proposed in this research work. Moreover, detail about the overhead introduced when using both models is described.

7.3.1 Implementation

The provision of a cost monitoring mechanism in the RTSJ starts with the definition of an interface for developers to interact with the mechanism. Since one of the intentions in this work is not to modify the current interface of the RTSJ, the interface already defined for cost monitoring is left unchanged. The interface is defined in the abstract class `ReleaseParameters`. Figure 7.6 summarises the important characteristics of the `ReleaseParameters` class with respect to the cost monitoring mechanism.

```java
public abstract class ReleaseParameters extends java.lang.Object implements java.lang.Cloneable {
    // Attributes used in cost monitoring
    private RelativeTime cost;
    private AsyncEventHandler overrunHandler;
    ...

    // Constructor used with cost monitoring
    protected ReleaseParameters(RelativeTime cost, RelativeTime deadline,
                                AsyncEventHandler overrunHandler, AsyncEventHandler missHandler);
    ...

    // Methods used for interfacing with cost monitoring
    public RelativeTime getCost();
    public void setCost(RelativeTime blocking);
    public AsyncEventHandler getCostOverrunHandler();
    public void setCostOverrunHandler(AsyncEventHandler handler);
    ...
}
```

Figure 7.6: Abridged `ReleaseParameters` class with cost monitoring.
The \textit{ReleaseParameters} class includes a \textit{cost} attribute that is used to set the maximum execution time (cost) that a thread is subject to, during one specific release or period (following the new model proposed in this thesis). The attribute \textit{costOverrunHandler} holds a reference to an asynchronous event handler, which should be released when an overrun of this maximum time occurs. The constructor of the class includes parameters used for initialising these two attributes. Methods for setting and getting the values of these attributes are defined as well.

Moreover, following a recent trend in the RTSJ [38] to introduce, in the next version, the possibility of obtaining information about the execution time of a real-time thread, this research work also provides such an interface. The summary of an interface for getting values related to the current, minimum, average and maximum execution time values used in a given release for a real-time thread are shown in Figure 7.7\textsuperscript{2}.

```java
public class RealtimeThread extends java.lang.Thread implements Schedulable {
    ...
    // Attributes used in cost monitoring
    private RelativeTime minimumExecution;
    private RelativeTime averageExecution;
    private RelativeTime maximumExecution;
    ...
    // Methods for interfacing with cost monitoring
    public RelativeTime getCurrentExecution();
    public RelativeTime getMinimumExecution();
    public RelativeTime getAverageExecution();
    public RelativeTime getMaximumExecution();
    ...
}
```

Figure 7.7: Abridged \textit{RealtimeThread} class with cost monitoring.

Three different attributes are included in the \textit{RealtimeThread} class. These are used to store, respectively, the minimum, average and maximum execution times that the thread was subject to during its past releases. These attributes are handled by the RTSJ imple-

\textsuperscript{2}Although this thesis focuses on the \textit{RealtimeThread} class, similar extensions could be defined for all schedulable objects by using the \textit{Schedulable} class.
The developer is capable of getting these values and also obtaining the current execution time used by a thread via the respective methods shown in Figure 7.7. Note that, the implementation provided in this work modifies the original one defined in the RTSJ. It enables the calculation of the average execution time and also does not restrict the methods to be static, as they are defined originally in the RTSJ. In particular, the use of non-static methods allows other threads in the system to obtain data, relative to execution times, from other threads in the system.

So far, the interface provided to developers for interacting with the cost monitoring mechanism has been described. Since the underlying OS for jRate used in the environment is MaRTE OS, the execution-time clocks and timers found in MaRTE OS can be used to implement the mechanism in jRate. Therefore, in the implementation of the environment, every real-time thread in jRate has a corresponding thread in MaRTE OS. At the middleware (the jRate run time system), the behaviour of real-time thread specified in the RTSJ is implemented. Moreover, in the implementation of MaRTE OS, every thread has an associated execution-time clock.

This means that, for the purposes of implementing the cost monitoring mechanism, every real-time thread at the middleware has either one execution-time timer associated to it (in the implementation of the cost monitoring mechanism of the RTSJ) or two (in the proposed cost monitoring mechanism of this thesis). With respect to the behaviour of the real-time thread, certain actions are then executed by the middleware at specific points in time. For the cost monitoring mechanism described in the RTSJ, these include:

- setting an execution-time timer (associated to the release time) to the execution value specified in the real-time thread when it (i) is firstly released, (ii) completes subsequent releases, (iii) an overrun occurs and the next monitoring phase associated to the real-time thread has already occurred;
• calculate the minimum, average and maximum execution times (by accessing the
execution-time clock that is associated to the thread in MaRTE OS) when a release
is completed by the real-time thread.

For the cost monitoring mechanism proposed in this research, besides executing all the
previous actions, the following action has to be included as well:

• setting an execution-time timer (associated to the period time) to the execution
  value specified in the real-time thread, every time a period occurs.

With respect to the enforcement model, the implementation currently stops the thread
by lowering its priority to a value that is never executed in the system. Such enforcement
may not be the best policy to react to the timing error itself.

7.3.2 Overhead Evaluation

In order to evaluate the overhead introduced by the two different cost monitoring and
enforcement mechanisms, the following experiment is defined. There are $n$ periodic threads
in the system, leading to 5 different executions of the experiment, where $n$ is equal to 5,
10, 15, 20 and 25. All the threads in the experiments are released at the same time,
and they have different (increasing) priorities. This corresponds to the highest possible
interference that lower priority threads can suffer. The period of the threads is equal to
their deadlines, which is 30 $\text{msec}$.

Each periodic threads executes for 1 $\text{msec}$ and then finishes its release, by calling
the $\text{waitForNextRelease()}$ method. The cost values for all the threads are set to 3 $\text{msec}$. Although the cost could be set to 1 $\text{msec}$, it is necessary to account for the execution
time that special threads need to use in order to monitor the experiment. Therefore, the
highest priority thread obtains the start time of the experiment when it starts executing,
and the lowest priority thread obtains the end time of the experiment when it finishes its
7.3 Cost Monitoring in the RTSJ

Figure 7.8: Result for the experiments with cost monitoring mechanisms.

release. This time accounts for the execution of the highest priority thread and all the other threads in the system. In terms of the overhead introduced by a cost monitoring and enforcement mechanism, this corresponds to: (a) the time the system has to set the execution-time timers; and (b) the expiration of the previously set execution-time timers (generation of interrupts).

Results for the experiments. The experiments were executed for 10000 times, so that figures converge. As in the previous experiments of this chapter, the tests were carried out in a 1.6GHz Centrino T2050 Core Duo processor. Although a dual core processor was used, MaRTE OS only supports a uniprocessor architecture. Therefore, only one of the two processors is used during execution. Figure 7.8 shows the results for the experiments, where three different implementations are used. Firstly is the implementation without the addition of a cost monitoring mechanism. Secondly is an implementation of the cost monitoring mechanism defined in the RTSJ. Thirdly is the implementation of the revised cost monitoring mechanism proposed in this research.
As it can be observed in Figure 7.8, the results of the experiments are almost identical. Indeed, the \textit{msec} level of time is the same for all the experiments. This means that the overhead is only visible at the \textit{µs} level of time. When compared to the experiments executed in the system not implementing cost monitoring, the average overhead for the experiments using the cost monitoring mechanism of the RTSJ were in the range of 0.18%. This average overhead increased to 0.21% compared to the revised cost monitoring mechanism proposed in this research work. This can be considered a very small overhead, below 0.5% to the overall system. Moreover, when comparing the overhead results with the cost monitoring mechanisms, the overhead introduced by the cost monitoring mechanism proposed in this thesis is very small, below 0.05%.

7.4 Blocking Time Monitoring in the RTSJ

A blocking time monitoring mechanism for the RTSJ was described in Chapter 6. This section aims to describe details of the implementation of the proposed model and evaluate the overhead that this model introduces in the system.

7.4.1 Implementation

In order to provide a mechanism for blocking time monitoring in the RTSJ, the first step is to define an interface for an application to interact with the monitoring mechanism. In this sense, the most straightforward step for defining the interface is to add attributes and methods similar to those currently used in the RTSJ for cost monitoring and enforcement (described in the last section). These are mainly defined in the abstract class \textit{ReleaseParameters}. Following this strategy, Figure 7.9 summarises the important characteristics of the extended \textit{ReleaseParameters} class, modified to interface with the blocking time monitoring mechanism.
7.4 Blocking Time Monitoring in the RTSJ

The modified `ReleaseParameters` class includes a `blocking` attribute that is used to set the maximum blocking time that a thread is subject to during one specific release. The attribute `blockingHandler` holds a reference to the asynchronous event handler, which should be released if a violation of this maximum time occurs. The constructor of the class is also modified to include parameters that are used for initialising these two attributes. Moreover, methods are defined for setting and getting the values of these attributes as well.

Note that, these changes only tackle the blocking time violation problem and do not present the possibility of getting values related to the minimum, average and maximum blocking time values, used in a given release for the real-time thread. In order to provide such feature, an extension of the current real-time thread class is also proposed. Figure 7.10 presents a summary of the changes needed to provide support for these functionalities.

---

3Although this thesis focuses on the `RealtimeThread` class, similar extensions could be defined for asynchronous event handlers. In fact, the class `Schedulable` could be used, so that all subclasses (real-time threads and asynchronous event handlers) would have access to these features.
public class RealtimeThread extends java.lang.Thread implements Schedulable {
...
// Attributes used in blocking time monitoring
private RelativeTime minimumblocking;
private RelativeTime averageblocking;
private RelativeTime maximumblocking;
...
// Methods for interfacing with blocking time monitoring
public RelativeTime getMinimumBlocking();
public RelativeTime getAverageBlocking();
public RelativeTime getMaximumBlocking();
}...

Figure 7.10: Abridged RealtimeThread class with blocking time monitoring.

The modified RealtimeThread class includes three different attributes, which are used to store, respectively, the minimum, average and maximum blocking times that the thread was subject to during its past releases. These attributes are handled by the RTSJ implementation. The developer is capable of getting the values via the respective methods that are also shown in Figure 7.10.

This concludes the interface provided to developers for interacting with the proposed blocking time monitoring mechanism. Since the underlying OS for jRate that is used in the environment is MaRTE OS, the blocking-time clocks and timers defined in this research work can be used to implement the proposed mechanism in jRate. Indeed, this is the approach taken in this thesis. In the implementation of the environment, every real-time thread in jRate has a corresponding thread in MaRTE OS. At the jRate run time system level (middleware), the behaviour of real-time thread specified in the RTSJ is implemented. Therefore, following the blocking-time clocks and timers implementation defined in MaRTE OS, every thread has an associated blocking-time clock in MaRTE OS.

This means that, for the purposes of implementing the proposed blocking time monitoring mechanism, every real-time thread at the middleware level has a blocking-time timer associated with it. Moreover, with respect to the behaviour of the real-time thread,
certain actions are done by the middleware at specific points in time:

- set the blocking-time timer to the blocking time value specified in the real-time thread when it (i) is firstly released and (ii) completes subsequent releases;

- calculate the minimum, average and maximum blocking times (by accessing the blocking-time clock associated to the thread, in MaRTE OS) when a release is completed by the real-time thread.

Different from the cost monitoring mechanism that introduces a default cost enforcement policy upon the detection of a blocking time violation, a handler is released in this case. As already described, this policy is adopted because: (a) the thread being monitored is the higher priority thread outside the shared resource; (b) stopping the lower priority thread that is executing inside the shared resource may impact seriously on the timing properties of the system.

### 7.4.2 Overhead Evaluation

The experiments used to evaluate the overhead of the blocking time monitoring mechanism proposed for the RTSJ is very similar to the experiments presented earlier in the context of the POSIX/C standard. Indeed, in order to evaluate the overhead introduced by the implementation, two different experiments are specified. The first experiment aims to measure the overheads of the protocol when using the PI protocol, whilst the second experiment focuses on the PCE protocol.

**PI experiment.** The experiment for the PI protocol is defined as follows. There is only one shared resource amongst several threads. The lowest priority thread is released and enters the shared resource before any other thread in the system. One msec after it started executing, another thread (with a higher priority than its predecessor) is released
by the system. Because the lowest priority thread is executing inside the shared resource, the released thread tries to enter the shared resource but becomes blocked – raising the active priority of the thread inside the shared resource. This periodic release of new threads continues until all threads in the system have been released (from lowest to highest priority). Moreover, the first released thread (with the lowest priority) does not finish executing inside the shared resource until all threads have been released. This is achieved by having the thread executing a loop whose condition only becomes false when the last thread (with the highest priority) tries to enter the shared resource. With this experiment, all threads in the system become blocked by the lowest priority thread hence it is obtained the maximum possible range of priorities for calculating the blocking time for the threads in the system.

Two time figures, relative to two distinct context switches, are obtained from the experiment. Firstly, the interval (context switch) where the highest priority thread in the system tries to enter the lock and the lower priority thread inside the shared resource starts executing. This accounts for the time the algorithm starts monitoring the blocking time. Secondly, the interval (context switch) the lowest priority thread is about to leave the shared resource and the highest priority thread enters the shared resource. This accounts for the time the algorithm stops monitoring the blocking time. Moreover, these are the longest intervals for the blocking time monitoring algorithm, as it has to make the maximum number of comparisons to find the threads that are being blocked by the lowest priority thread.

**PCE experiment.** As in the previous experiment, in the experiment for the PCE protocol only one shared resource is shared amongst several threads. The lowest priority thread is released and enters the shared resource before any other thread in the system. Once entering the shared resource, the active priority of the thread is raised to the ceiling
7.4 Blocking Time Monitoring in the RTSJ

of the lock. Ten \textit{msec} after it started executing, another thread (with a higher priority than its predecessor) is released by the system. However, it is not allowed to execute, since the thread inside the shared resource now has the highest priority in the experiment. This periodic release of new threads continues until all threads in the system have been released (from lowest to highest priority). The first released thread does not finish executing inside the shared resource until all threads have been released. This is again achieved by having the thread executing a loop on the time where the last thread in the experiment is released. With this experiment, all threads in the system are eligible for execution. Hence, the maximum possible range of priorities for calculating the blocking time for the threads in the system is obtained.

Just like in the previous experiment, two time figures, relative to two distinct context switches, are obtained. Firstly, the interval (context switch) where the thread inside the shared resource yields the processor and the last released thread in the experiment starts executing (as it has the same priority of the shared resource ceiling). This accounts for the time the algorithm \textit{stops} monitoring the blocking time. Secondly, the interval (context switch) where the last released thread yields the processor and the thread inside the shared resource starts executing again. This accounts for the time the algorithm \textit{starts} monitoring the blocking time.

\textbf{Results for the experiments.} The experiments were executed with a different number of threads, ranging from 2 (only 1 blocked thread) to 24 (23 blocked threads), in two distinct implementations, one with blocking time monitoring enabled and another without it. Each experiment was executed for 10000 times, so that the figures converge. Just like for the previous experiments, the tests were carried out in a 1.6GHz Centrino T2050 Core Duo processor. Figures 7.11 and 7.12 show different graphics for the start and stop monitoring conditions of the protocol for experiments with the PI protocol and the PCE protocol,
The implementation without blocking time monitoring does not include the start and stop monitoring conditions of the protocol. Compared to the experiments of blocking-time clocks and timers in POSIX (previous Section 7.2.2), the times presented in the graphics are greater. This is due to the fact that the RTSJ implementation has a more complex structure (the jRate middleware), which introduces additional overhead to the system, when compared to the execution of the system using MaRTE OS.

In the experiment with the PI protocol, the overhead for the start monitoring part of the mechanism implementation increases on the average by 2.28% per each new thread.

Figure 7.11: Overhead for different settings of the PI experiment in the RTSJ.
Figure 7.12: Overhead for different settings of the PCE experiment in the RTSJ.
added to the experiment. The overhead for the implementation without blocking time monitoring is quite lower, on the average increase of 0.43% per each new thread that is added to the experiment. The stop monitoring part of the mechanism shows an average increase of 3.55% per new thread in the experiment with blocking time monitoring, compared to an average increase of 1.67% per thread in the implementation without blocking time monitoring.

The experiments with the PCE protocol, using blocking time monitoring, show overheads on average of 1.95% (for the start monitoring part) and 1.60% (for the stop monitoring part) per each new added thread. The experiments with the PCE protocol implementation without blocking time monitoring present no overhead increase. This is an expected result, as in the experiments with the PCE protocol the system does not have to deal with threads that are entering and leaving the shared resource – the data is gathered by threads yielding the processor.

Albeit higher, the overhead increase for the experiments are similar to the overheads shown for POSIX blocking-time clocks and timers (Section 7.2.2). This is an expected result, as the blocking time monitoring is implemented using the underlying POSIX blocking-time clocks and timers. Therefore, this overhead can be reduced with more efficient implementations of blocking-time clocks.

7.5 Summary

This chapter presented several aspects related to the implementation and evaluation of the error detection mechanisms described in this thesis. Initially, the environment used for the implementation of the error detection mechanisms was described. This environment is composed by the RTSJ’s compliant middleware jRate running on top of the MaRTE OS. It was shown how this environment was ported from running as a library in Linux (an emu-
lation of the system) to running in a bare machine. Then, three different implementations and evaluations of error detection mechanisms were presented.

Firstly, the implementation and evaluation of the proposed abstraction of blocking-time clocks and timers for POSIX/C was presented. In particular, this implementation introduces a considerable overhead, especially if a large number of threads are found in the priority range defined by the current thread running the system. Though, this implementation can be improved by removing the need to add and remove threads from the blocked queue, as described previously.

Secondly, the implementation of the execution time monitoring mechanism for the RTSJ was presented. The implementation used the POSIX/C execution-time clocks and timers provided by the underlying MaRTE OS. In summary, the overhead introduced by the monitoring mechanism based on releases and periods, when compared to a system without execution time monitoring mechanism and the monitoring mechanism of the RTSJ, is very small. This occurs due to the need to have another execution-time timer for each thread in the system.

Thirdly, the implementation of the blocking time monitoring mechanism for the RTSJ was described. Like in the implementation of the execution time monitoring mechanism, this mechanism uses the POSIX/C blocking-time clocks and timers, implemented previously in MaRTE OS. Therefore, all the previous overhead of the blocking-time clocks and timers is absorbed by the system, in addition to the overhead of the middleware.
This chapter provides, in the next section, a discussion of how the detection of timing errors could be potentially used with techniques for damage confinement to improve the reliability of real-time systems. Then, Section 8.2 gives an overview of the contributions included in this thesis. Section 8.3 summarises the overall conclusion from the research. In Section 8.4, future research directions motivated by the work are presented. Finally, Section 8.5 concludes the thesis with some final remarks.

8.1 Improving the Reliability of Real-Time Systems

At the core of this thesis is the belief that, with the use of different error detection mechanisms for timing errors, the reliability of real-time systems can be improved by detecting timing errors as early as possible. So far, different error detection mechanisms for two different technologies used for the development of real-time systems, POSIX/C
and the Real-Time Specification for Java, have been described in this thesis. The key point to be discussed in this section is that, in order to improve the system’s reliability with respect to timing errors, certain policies, in many aspects particular to the system, have to be adopted upon the occurrence of timing errors. For instance, in the last chapter, one type of policy was assumed in the implementation of the execution time monitoring mechanism of the RTSJ. The adopted policy stated that, whenever an execution time overruns occurs, the overrunning thread shall be stopped. This meant that the possible temporal damage has been confined and that no interference is added to other threads in the system. Nevertheless, this gave no chance for the erroneous thread to recover in that particular release.

This thesis contends that different systems require different policies. For instance, some approaches that could be used upon execution overruns include:

- stopping the thread, so it does not interfere with the execution of other threads;
- lowering the priority of the thread (instead of stopping it), so that it could use slack time from the system, when all other threads have finished executing;
- providing another chance for the thread to recover itself (possibly executing a specific recovery code), by increasing its execution time budget of $\Delta t$ when an overrun occurs;
- depending on the importance of the thread to the system, generating an exception, and aborting the thread’s current release.

Similar policies can also be used upon the detection of blocking time violations, like:

- the system can assume a value $B + \Delta$ for the blocking time, where $B$ is the estimated blocking time and $\Delta$ is the time the system has to reconfigure itself in order to tolerate the extra blocking time;
• an exception can be generated to the thread that is being blocked, so that it can execute recovery code;

• a mechanism for detecting the lower priority thread that is violating the blocking time can be built, so that this thread can roll-back (like the approach described in [48]) its execution. Then, higher priority threads are able to execute in the system.

This thesis argues that the requirements of the system may dictate the approach that will be adopted upon the occurrence of timing errors during the run time execution of the system. Nevertheless, the key point is that by using a diversity of detection mechanisms for timing errors, the errors can be detected earlier and, therefore, policies for confinement and error recovery could be executed with more accuracy.

8.2 Contributions

This thesis focused on the detection of timing errors, within a fault tolerance framework, of a real-time system. The target system comprised a uniprocessor architecture, the use of the FPS (Fixed-Priority Scheduling) approach, and the possibility of threads interacting via shared resources based on priority inheritance algorithms. No restriction was posed over the temporal type of these threads, therefore, periodic, sporadic and aperiodic threads were assumed. For such a system, four different types of timing errors were considered during the run time execution of the system: (i) execution time overrun of a thread; (ii) execution time overrun of a thread inside a shared resource; (iii) violation of the MIT (Minimum-Interarrival Time) for a thread; (iv) deadline miss of a thread. At the core of the thesis was the idea that, with respect to fault tolerance, the typical deadline monitoring mechanism used for error detection in real-time systems, does not provide enough accuracy for damage confinement. Indeed, the motivation behind the thesis was well synthesized by its hypothesis, which stated that:
Timing errors can develop when the assumptions used in the feasibility analysis of the system are undermined during the system’s run time execution. In this context, different run time monitoring mechanisms for error detection can be used for the earlier detection of timing errors, before the occurrence of deadline misses. Where applicable, this approach can provide mechanisms to confine the temporal damage to the errant threads of the system as well as facilitate error recovery strategies.

In particular, the thesis contended that the implementation of different error detection mechanisms for timing errors could improve the reliability of real-time systems. Also, whenever possible, this approach could introduce some form of confinement for the application, so that the temporal damage cannot interfere with the execution of other threads in the system. In order to demonstrate the hypothesis, the following contributions were presented in this thesis:

(C1) development of a technique to implement blocking time overrun detection for a thread using the POSIX/C standard. Together with execution time monitoring, this provides a complete model for detecting timing errors in a uniprocessor architecture, using a finer timing granularity than previously possible in real-time systems;

(C2) development of a model for the evaluation of execution time monitoring protocol of the RTSJ. This resulted in the discovery of an error and the proposal of a new model;

(C3) development of a technique to implement blocking time monitoring for a thread using the RTSJ. The higher level constructs of the RTSJ provides the ability to detect a blocking time overrun and quickly respond to it, via a mechanism that can improve the recovery for such type of timing error;
(C4) A better understanding of the relationship between timing errors and the possibility of using techniques for damage confinement in real-time systems.

Besides these contributions, used to demonstrate the thesis hypothesis, other contributions of the thesis included:

(C5) The provision of an open-source run-time support system for the RTSJ that executes in a bare machine under the x86 architecture.

This way, in order to demonstrate the hypothesis, different error detection mechanisms, for the considered timing errors, in two technologies were presented: the POSIX/C standard; and the RTSJ (Real-Time Specification for Java). Chapter 3 focused on presenting an overview of the possible error detection models, for the assumed timing errors, for both POSIX/C and RTSJ.

In Chapter 4, the possible error detection models for POSIX/C were detailed. These included: (i) the use of clocks and timers, to detect deadline misses; (ii) the use of execution-time clocks and timers, to detect the execution overrun of threads; (iii) the use of clocks and timers, to detect MIT violations; and (iv) the use of blocking-time clocks and timers, to detect the execution overrun of threads inside shared resources. In particular, the main contribution of the thesis (C1) in the chapter for POSIX/C was the definition of an algorithm for monitoring blocking time and the introduction of blocking-time clocks and timers.

Since the RTSJ provides a complex behaviour for its real-time threads, in Chapter 5 the model for real-time threads was formally analysed via model checking. Chapter 6 continued to focus on the RTSJ, but on the detailed presentation of the possible error detection models considered. These included: (i) the deadline monitoring model; (ii) the cost monitoring and enforcement model; (iii) the use of aperiodic and sporadic policies for detecting MIT violations; (iv) the blocking time monitoring model. The first main
contribution of the thesis (C2) in this chapter was the formal analysis, via model checking, of the cost monitoring model already found in the RTSJ. The analysis showed that in some scenarios, the model can provide to a thread more execution time than defined in a given period. This lead to the definition of a revised execution time monitoring mechanism that corrects the anomaly found. Moreover, another contribution of the thesis (C3) in this chapter included the proposal of a blocking time monitoring mechanism for the RTSJ.

In Chapter 7, an environment for implementing the error detection models proposed in the thesis was described. A requirement was that all the software used in this environment was open-source. Therefore, the environment is composed by the RTSJ’s compliant middleware jRate running on top of MaRTE OS. A contribution (C5) of the thesis was to port this environment from running as a library in Linux (an emulation of the system) to running in a bare machine. Then, three different implementations and evaluations of the error detection mechanisms proposed in this thesis were described. During the evaluation of the implemented mechanisms, it becomes clear that the mechanisms introduce overhead to the system. Nevertheless, depending on the objective of the application, such overheads are compensated by the possibility of detecting specific timing errors, which can be dealt with different and specific policies for either or both damage confinement and error recovery.

Moreover, in Section 8.1, a discussion of how a real-time system’s reliability could be improved by using the proposed error detection models was presented. A key point in this discussion was the fact that, in order to improve reliability, adequate policies (and sometimes specific to a system) for dealing with the timing error have to be assumed. Indeed, another contribution of the thesis (C4) was a better understanding of this relationship between timing errors, detection and policies for confinement and error recovery.
8.3 Overall Conclusions

During the development of the thesis, different error detection mechanisms were presented, in terms of the POSIX/C standard and the RTSJ, for each one of the timing errors being considered: (a) deadline miss; (b) execution overrun of the thread’s WCET; (c) execution overrun of the thread’s WCET inside a shared resource; (d) overrun of the thread’s MIT assumptions. With respect to the use of the mechanisms for detecting timing errors, it was stated in the thesis that (in the context of both technologies):

- the deadline monitoring mechanism (used to detect the timing error (a)) should be implemented in a real-time system, because it provides the latest stage to identify a (possibly) serious erroneous condition;

- the execution time monitoring mechanism (for detecting the timing error (b)) should be used, since it has a constant overhead and provides the ability for the early detection of overrun conditions, which can potentially avoid future deadline misses in the system;

- the utilisation of a mechanism for detecting the violation of MIT assumptions (timing error (d)) should be considered to be application specific. On the one hand, if the application has sporadic events that have a high importance to the system, and an MIT violation may indicate a serious problem that has to be dealt with, using such mechanism is crucial. On the other hand, if the violation of MIT assumptions does not lead to such a scenario, it may not be needed. Though, it is important to provide the support for executing aperiodic and sporadic threads via servers, so that these threads will not impact on the execution of other threads in the system.

With respect to the timing error (c), a special protocol for detecting the overrun was proposed – with a considerable overhead imposed over the system. In particular to the
8.3 Overall Conclusions

POSIX/C standard, it is possible to monitor and detect such timing error using execution-time clocks and timers. From the overhead point-of-view, it may not be the best option to use blocking-time clocks and timers. Though, the arguments in favour of using blocking-time clocks and timers include:

(a) the simplicity of use: specially in systems with nested shared resources. The developer can just set the blocking-time timer to the maximum time to detect the timing error. Where in the execution-time clocks and timers approach for detecting such timing error, the developer would need to start and stop an execution-time timer every time a thread enters a shared resource;

(b) the different perspective for handling the timing error: where the higher priority thread that is waiting to gain access to the shared resource is signalled about the error. Using execution-time clocks and timers, the thread that is executing inside the shared resource is signalled about its own erroneous behaviour.

From these benefits, it is stated in this research that in large systems, specially using the Priority Inheritance (PI) protocol, with nested shared resources and different complex interactions, the use of blocking-time clocks and timers could provide a good mechanism for detecting this particular timing error. Since determining the blocking time in such systems can be very difficult, the mechanism can be used during the development phase of the system to calculate the blocking times. It can also be a valuable mechanism to be used during testing and debugging of the real-time system. Moreover, with the improvement of the implementation, the overhead can certainly be reduced.

In the RTSJ, no support is provided for monitoring the execution time of a specific block of code from the application. The use of a blocking time monitoring mechanism may be used in any real-time system that considers the interaction of threads using shared resource and priority inheritance protocols. Another view is to equip the RTSJ with
mechanisms similar to execution-time clocks and timers, as it is being currently considered in [38]. One argument against this approach is that this mechanism can be classified as a low-level mechanism, and does not follow the RTSJ’s high level approach of integrating timing aspects with the behaviour of the real-time thread. Nevertheless, it does introduce another option for dealing with this particular timing error.

8.4 Future Research Directions

This work has particularly focused on a uniprocessor architecture. A major trend in the architecture of computer systems is the use multi-processor architectures. This trend can be easily seen with the commercialization of due-core processors, like the one used in the evaluation of the implementations of this thesis. But the industry is aiming at developing computing architectures with a vast number of cores, which is quite an open challenge. In this context, a first future research direction that can be taken is to re-analyse the possible timing errors that can occur in such a computing architecture. This can lead to the adaptation of the timing error techniques described in this thesis or the development of new techniques, tackling timing errors that may be particular to multi-processor architectures.

Still considering the system model used in this thesis, another future research direction is to look at different policies for damage confinement and error recovery that could be used in real-time systems. Indeed, this is a natural step from the developed work, in the sense that all the basis (with respect to the considered timing errors) for the detection of erroneous behaviour of the system are provided. Therefore, providing a framework with different damage confinement and error recovery policies that can be tuned to particular applications can facilitate the task of developing fault tolerant real-time systems.

From the point of view of the RTSJ formal models defined in the thesis, one particular
interesting future research direction is to analyse the model of Asynchronous Event Handlers (AEH). This would provide a complete formal analysis for the notion of Schedulable Objects proposed in the RTSJ.

8.5 Concluding Remarks

This research was developed around the possibility of timing errors occurring in real-time systems under a uniprocessor architecture, with the FPS approach, and where threads can interact using shared resources (with priority inheritance protocols). Four different error detection mechanisms, tackling four different timing errors, were presented in the context of the technologies POSIX/C and RTSJ. Under the considered computational model, these four timing errors were enough for reflecting the possible erroneous behaviour that could be generated if the system deviates from the feasibility analysis. Though, different computational models may develop other timing errors. The research of such mechanisms is an important one to possibly improve the system’s reliability. Therefore, the research of new mechanisms for error detection for timing errors and, in particular, their connection to recovery strategies is still active.
In this research work, the UPPAAL tool [17, 10] is used for the formal analysis (via model checking) of the Timed Automata (TA) formalism. The modelling language of UPPAAL adds valuable features to the TA formalism. For the specification of properties, the tool provides a simplified version of the Computation Tree Logic (CTL). Thus, given a model and a property specification, it is possible to use the verification engine to check if the property is satisfied by the model. If the property is not satisfied, a simulation engine can be used to analyse the counter-example for that property.

Here, only a brief overview of the UPPAAL features are given – the reader is referred to [17] for a more detailed explanation. The modelling language of UPPAAL extends the TA formalism with features like [17]: bounded integer variables that can be read, written and are subject to arithmetic operations; constants, which cannot be modified and are used to assign a name to an integer value; synchronization channels, where a sender automaton (c!) can non-deterministic synchronize with a receiver automaton (c?); and
committed locations, where if a transition occurs and the destination locations is committed, the next transition must occur atomically without delay and without interleaving with other transitions in the model.

Property specifications in UPPAAL consist of a path and a state formula. The path formula is used to reason about possible execution paths of the model, and the state formula refers to specific states. Properties are classified as state, reachability, safety and liveness [17]. A state property refers to an expression ($\phi$) evaluated at a specific state. The deadlock keyword is provided and defines a state where no possible transitions can occur in the model. Reachability properties are used to check if a given state is possibly reachable ($E <> \phi$) – the syntax found in [17] is used for the specification of properties. A safety property is used to guarantee that “something bad will never happen”, i.e., the property is valid in every state of all execution paths ($A[] \phi$) or at least one or more execution paths ($E[] \phi$). Finally, liveness properties are used to guarantee that “something good will eventually happen”. They can be specified using $A <> \phi$, or the pattern $\Phi \Rightarrow \phi$ (which means that whenever $\Phi$ is satisfied, $\phi$ will be satisfied). In this pattern, when the left-side of the implication ($\Rightarrow$) is valid, the right-side must become valid as well. This way, the verification process forces certain computations to occur (using the left-side) and checks that the side effects of reaching that computation are correct (specified in the right-side).
Bibliography


