Modelling and verification of predictable data flow in real-time systems

by

Boris Madzar

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
October 2015

Copyright © Boris Madzar, 2015
Abstract

As vehicle electronic control systems become responsible for an increasing number of functions, developers of such systems find themselves pulled in two directions: the need to perform more and more computation while at the same time reducing cost, weight and power consumption. Traditionally, safety isolation and fault containment of software tasks has been achieved through either physical or temporal segregation. This approach is reliable but inefficient in terms of processor utilization. Recently, there has been a move towards systems which achieve better processor utilization by executing all tasks together and guaranteeing safety isolation and fault containment by means of formal verification.

First, a definition for one such system was developed, based on PharOS and Giotto. The system modelled attempts to ensure observably deterministic data flow (i.e., inputs from and outputs to other systems), but allows for non-observable non-determinism. It also allows for sporadic deadline overruns and accounts for criticality during fault handling.

Second, a method for verifying a given input system definition (system, set of tasks and fault model) for predictable data flow was developed. A stand-alone tool was made that performs this verification and displays the results graphically. The core exhaustive verification is performed by the Spin model checker.
Acknowledgments

I would like to thank my supervisor, Juergen Dingel, for all his help and guidance. I would also like to recognize the generous support of Tom Fuhrman and Ramesh S of General Motors Research and Development.

Funding for this project was provided by the Network for the Engineering of Complex Software-Intensive Systems for Automotive Systems (NECSIS) and the Natural Sciences and Engineering Research Council of Canada.
Contents

Abstract i

Acknowledgments ii

Contents iii

List of Tables vi

List of Figures vii

Chapter 1: Introduction 1
   1.1 Aim .......................................................... 3
   1.2 Contribution .................................................. 3
   1.3 Organization of Thesis ...................................... 4

Chapter 2: Background 5
   2.1 Real-time Operating Systems ................................ 5
   2.2 Task Scheduling .............................................. 6
      2.2.1 Priority Assignment ................................... 7
      2.2.2 Task Selection .......................................... 8
   2.3 Tasks ........................................................ 9
   2.4 Schedulability ............................................... 13
   2.5 Scheduling Algorithms ...................................... 14
   2.6 Zero-Slack Scheduling ...................................... 17
   2.7 Predictability ............................................... 18
   2.8 Predictable Data Flow ...................................... 20
   2.9 AUTOSAR ................................................... 23
   2.10 Exhaustive Verification .................................... 24
      2.10.1 Spin ................................................... 25
   2.11 Conclusion .................................................. 26

Chapter 3: Related Work 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 PharOS</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Giotto</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Schedulability Verification</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Conclusion</td>
<td>35</td>
</tr>
<tr>
<td><strong>Chapter 4: Problem Definition</strong></td>
<td><strong>36</strong></td>
</tr>
<tr>
<td>4.1 System Description</td>
<td>37</td>
</tr>
<tr>
<td>4.2 Scheduling</td>
<td>40</td>
</tr>
<tr>
<td>4.3 Faults and Fault Handling</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1 Task Failure</td>
<td>41</td>
</tr>
<tr>
<td>4.3.2 Budget Violation</td>
<td>42</td>
</tr>
<tr>
<td>4.3.3 Deadline Violation</td>
<td>43</td>
</tr>
<tr>
<td>4.4 Sporadic Deadline Overruns</td>
<td>43</td>
</tr>
<tr>
<td>4.5 Conclusion</td>
<td>45</td>
</tr>
<tr>
<td><strong>Chapter 5: Modelling and Verification</strong></td>
<td><strong>46</strong></td>
</tr>
<tr>
<td>5.1 Guarantee</td>
<td>47</td>
</tr>
<tr>
<td>5.2 Model Definition</td>
<td>47</td>
</tr>
<tr>
<td>5.3 Verification Techniques</td>
<td>48</td>
</tr>
<tr>
<td>5.3.1 Direct Approach</td>
<td>48</td>
</tr>
<tr>
<td>5.3.2 Indirect Approach</td>
<td>49</td>
</tr>
<tr>
<td>5.4 Conclusion</td>
<td>52</td>
</tr>
<tr>
<td><strong>Chapter 6: Implementation</strong></td>
<td><strong>54</strong></td>
</tr>
<tr>
<td>6.1 Choice of Model Checker</td>
<td>54</td>
</tr>
<tr>
<td>6.2 Initial Exploration</td>
<td>57</td>
</tr>
<tr>
<td>6.3 Final Design</td>
<td>59</td>
</tr>
<tr>
<td>6.3.1 Assumptions</td>
<td>59</td>
</tr>
<tr>
<td>6.3.2 Observations</td>
<td>59</td>
</tr>
<tr>
<td>6.3.3 Changes to Model Design</td>
<td>61</td>
</tr>
<tr>
<td>6.4 Extensions to Spin</td>
<td>62</td>
</tr>
<tr>
<td>6.4.1 Static Annotations</td>
<td>64</td>
</tr>
<tr>
<td>6.4.2 Dynamic Annotations</td>
<td>65</td>
</tr>
<tr>
<td>6.4.3 Spin Source Code</td>
<td>67</td>
</tr>
<tr>
<td>6.5 Promela Skeleton</td>
<td>67</td>
</tr>
<tr>
<td>6.5.1 Operating System</td>
<td>68</td>
</tr>
<tr>
<td>6.5.2 Scheduler</td>
<td>68</td>
</tr>
<tr>
<td>6.5.3 Task Template</td>
<td>69</td>
</tr>
<tr>
<td>6.6 Graphical Interface</td>
<td>70</td>
</tr>
<tr>
<td>6.7 Visualization</td>
<td>71</td>
</tr>
<tr>
<td>6.7.1 Trace File Annotations</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Visualization annotation specification. . . . . . . . . . . . . . . . . . 73

7.1 Active safety demonstration task set. . . . . . . . . . . . . . . . . . . 94
7.2 Active safety demonstration data values. . . . . . . . . . . . . . . . . 95
7.3 First implementation performance. . . . . . . . . . . . . . . . . . . . 97
7.4 Second implementation performance. . . . . . . . . . . . . . . . . . . 98
List of Figures

2.1 The terminology used to describe task timing and execution. . . . . . . 12
2.2 Scheduling algorithm comparison. . . . . . . . . . . . . . . . . . . . . 16
2.3 Sample execution trace, with full observability – first version. . . . . . 27
2.4 Sample execution trace, with full observability – second version. . . . . 28
2.5 Sample execution trace, with limited observability – first version. . . . 29
2.6 Sample execution trace, with limited observability – second version. . . 30

4.1 Demonstration of write order. . . . . . . . . . . . . . . . . . . . . . . 39
4.2 Demonstration of deadline violation handling. . . . . . . . . . . . . . 45

5.1 Absorbing task overruns. . . . . . . . . . . . . . . . . . . . . . . . . . . 53

6.1 A pseudo-code version of the Promela skeleton. . . . . . . . . . . . . 84
6.2 A sample counterexample trail file. . . . . . . . . . . . . . . . . . . . . 86
6.3 A sample counterexample trail file visualization. . . . . . . . . . . . . 87
6.4 The data-entry view of the first implementation. . . . . . . . . . . . . 88
6.5 The results view of the first implementation. . . . . . . . . . . . . . . 88
6.6 The data-entry view of the second implementation. . . . . . . . . . . . 89
6.7 The task-entry view of the second implementation. . . . . . . . . . . . 90
6.8 The results view of the second implementation. . . . . . . . . . . . . . 90
7.1 Data flow diagram of the active safety demonstration task set . . . . 96
7.2 The four-core configuration state counts. . . . . . . . . . . . . . . . . 98
Chapter 1

Introduction

The miniaturization and commoditization of computer technology driven by consumer demand for smarter, more capable products has led to embedded processors being integrated into many everyday products. What was previously accomplished by means of purpose-built mechanical or electric designs can now be achieved through general-purpose computers and software at reduced cost. While opinions on the merits of such progress can sometimes be divided, it is impossible to deny that this is the direction the world is moving: from programmable thermostats to smartwatches, software is permeating every aspect of our lives.

This modernization can be seen in the world of motor vehicles as well: as recently as the 1980s, vehicle systems were almost entirely mechanical and electric. Today, a car can have dozens of independent processors running hundreds of millions of lines of code. This software controls everything from the radio to the airbags, and its depth and breadth is increasing each year due to the combined pressures of consumer demand, regulatory compliance and competitive advantage. On the horizon are vehicles in which the computer systems take over the role of the driver entirely – autonomous vehicles are the next major innovation in automotive technology.
The forces driving development of the embedded engine control units (ECUs) that run the real-time software in vehicles are pulling in different directions. On one hand, there is pressure to handle an increased number of software components, which themselves are increasing in complexity. On the other, there is pressure to reduce the size, weight, power consumption and – most importantly – cost of the hardware being used. Essentially, to do more with less. While improvements to processor and semiconductor design and economies of scale help in this endeavour, some changes do need to be made on the software level to reduce the resource footprint and improve flexibility.

As is the case with any large software system, especially one that is responsible for the safety of people and equipment, vehicle ECUs run many different software components that are built and tested to different standards and have different levels of criticality. These components are developed by different teams and different companies – some may even be commercial off-the-shelf software. It is therefore very difficult to predict how each piece of software will behave under all possible conditions. For this reason, it is crucial that the operating systems and supervisory code that manages these independent components be designed to withstand faults and failures, and not allow one component to negatively affect another.

A common approach in the industry is to design systems such that they provide “run-time” guarantees, i.e., faults are isolated at run-time for all possible software components, and extensive testing is not necessary. This could involve dedicated hardware, or time-slicing algorithms. These approaches provide good guarantees, but at the expense of under-utilizing hardware and therefore driving up cost.

An alternative approach is to design systems such that they provide “design-time”
guarantees, i.e., faults are not isolated at run-time so guarantees must be made when
the software components are being integrated into a single system. It must be shown
that it is impossible for failure to occur in all plausible cases. This approach requires
more extensive testing, as the behaviour is dependent on the software components
being run. The guarantees made by these approaches are weaker, but they are much
better at maximizing hardware utilization. It is just such an approach that was the
focus of this research.

1.1 Aim

The aim of this research was to explore the feasibility of using model-checking tools
and techniques in providing run-time guarantees for a specific system design, and to
create a proof-of-concept verification tool to provide these guarantees.

1.2 Contribution

The main contributions of the work described in this thesis are the system design,
the verification technique and the actual verification tool.

The specific design of the system was heavily influenced by previous work done on
PharOS and Giotto, as well as contributions from industry partners. Certain parts,
notably the approach to deadline overrun handling, are novel. The design of the
system was not based on any existing operating system, but rather synthesized from
a collection of behaviours that are of interest to the automotive industry.

A technique for verification of this system was designed, based on several insights
into how best to verify such systems using a model checker.

Finally, the actual proof-of-concept tool that was produced ties the first two points
together and represents an actual concrete implementation of the theory. While only a proof of concept, it was tested on a realistic example from the automotive domain.

1.3 Organization of Thesis

The remainder of this thesis is structured as follows: background information is provided in Chapter 2; related work is discussed in Chapter 3; descriptions of the system under study and the approach used to model and verify it are provided in Chapters 4 and 5, respectively; the implementation of the tool is discussed in Chapter 6; testing is covered in Chapter 7; and a reflection on the work and possible future extensions are discussed in Chapter 8. Finally, a summary and conclusion is provided as Chapter 9.
Chapter 2

Background

The system design under consideration is one that is commonly found in engine control units (ECUs) in motor vehicles. This type of system is composed of a real-time operating system, low-level software that provides common services (I/O, monitoring, etc.), and a set of periodic tasks which comprise the “business logic” of the ECU. A single vehicle may have multiple ECUs; communication between which is accomplished by means of a common network such as a CAN (controller-area network) bus.

The hardware of an ECU is similar to that of most general-purpose computers in use today: it contains a CPU comprised of one or more cores, volatile RAM, and read-only and read-write persistent storage. The work described in this thesis, while done with the automotive domain in mind, is equally applicable to other similar domains and other similar systems.

2.1 Real-time Operating Systems

The operating systems used in ECUs are different than typical desktop operating systems, such as Mac OS X or Microsoft Windows, that most people are familiar
with. These real-time operating systems (RTOS) are designed to be able to handle input data nearly instantaneously, with no buffering delays.

An RTOS is an operating system that is designed to provide low latency, predictable execution and the ability to reliably meet deadlines. It should prioritize the timing constraints above all else, including the complete execution of tasks. RTOSes are designed such that context switching latency is low, and they only run a small number of software tasks. An RTOS may need to run for long periods of time without intervention, therefore it must be able to recover from common faults. This is all in contrast to the typical desktop operating system, where tasks timing does not need to be guaranteed but correct task behaviour should be, and larger latencies are tolerated in favour of greater throughput.

In addition to the automotive domain, RTOSes are found in many other embedded-computing applications such as consumer electronics and control systems for appliances. Examples of RTOSes include Green Hills Software Integrity\footnote{http://www.ghs.com/products/rtos/integrity.html}, QNX\footnote{http://www.qnx.com/} and FreeRTOS\footnote{http://www.freertos.org/}.

### 2.2 Task Scheduling

One of the jobs of the operating system is task scheduling: determining which tasks should be granted the processor at any given time. A system may contain one or more processor cores. In the types of systems that were considered, tasks only require one core and do not care which core they are assigned. The cores are symmetric, i.e., each core is indistinguishable from any other.
2.2. TASK SCHEDULING

If a system contained as many processor cores as tasks, the operating system could simply assign one task per core and be done. However, this would be very wasteful: tasks do not need to be executing constantly, so for much of the time the core assigned to the task would be doing nothing. Additionally, if a core were to fail, the system would have no recourse.

A flexible scheduling system, wherein tasks are assigned processor time only when they need it, is preferable both from a cost-saving and safety perspective: on one hand, the hardware investment is reduced by requiring fewer cores; and on the other, the workload can shift to the remaining cores in the case of a failure. However, since we have more tasks than processor cores, the operating system needs to queue and schedule the tasks.

Task scheduling and schedulability is a well-developed area of computer science, and only the briefest of overviews is provided here.

The scheduling process is divided into two halves: priority assignment and task selection, both handled by the operating system. Priority assignment happens first. Once all tasks have a priority assigned, tasks can be selected for assignment to processor cores.

2.2.1 Priority Assignment

All tasks that are ready to execute need to be assigned a priority. Priorities can either be static – the task is always assigned the same priority value – or dynamic. In a dynamic-scheduling scheme, the operating system needs to calculate the priority of a given task based on certain parameters. The parameters used are dependent on the scheduling algorithm employed. Scheduling algorithms are discussed in greater detail
2.2. TASK SCHEDULING

in Section 2.3

2.2.2 Task Selection

The task selection process is very simple: for a system with $n$ cores, the $n$ highest-priority tasks that are ready to execute are assigned to these cores. The scheduler executes and re-calculates the core assignment every time the system reaches a pre-emption point. A preemption point is defined as any point at which a task preemption might occur, i.e., the core assignment may change. This will happen when a task becomes ready to execute, a task stops being ready to execute, or a ready-to-execute task’s priority changes. The frequency of preemption points is dependent on the nature of the tasks and the scheduling algorithm in use.

The task scheduler itself is software that needs time and resources to execute. As such, the tasks granted processor time must be stopped for the scheduler to run. There are two approaches to how this can be handled:

Cooperative Multitasking In a system that employs cooperative multitasking, it is the responsibility of the task that is assigned the processor to yield control back to the operating system at the appropriate time. This is the simplest to implement (as it requires no hardware support), but can be dangerous. A faulty task can lock up the entire system, and the only way to recover would be to forcibly reset the hardware. An example of systems using cooperative multitasking would be some consumer-grade operating systems from the early 1990s, such as Microsoft Windows 3.1 [21] and Apple System 7 [25].
2.3. TASKS

Preemptive Multitasking  A system that employs preemptive multitasking leverages hardware interrupts and interrupt handlers to cede control to the operating system without requiring cooperation from the tasks. The operating system requests the hardware raise an interrupt after a certain amount of time; when this interrupt occurs, the processor stops execution of the tasks and switches to operating-system-provided interrupt handling code. The operating system is then able to re-assign tasks to cores as needed. Essentially, this process allows the operating system to pause and un-pause time from the point of view of the tasks – the tasks are not aware a context switch has occurred. Most modern consumer operating systems use preemptive multitasking; examples include Microsoft Windows 95 and later [21] and Apple OS X [25].

The systems that are being considering here all employ preemptive multitasking. Since task code is considered untrusted and potentially faulty, it is not possible to make guarantees in the same way for systems that use cooperative multitasking.

2.3  Tasks

Tasks are responsible for the “business logic” of the ECU. The task sets that were explored were written in C, a low-level, compiled language. Due to the hard real-time constraints of the domain and the low performance of the embedded processors in use, C is a good choice. Tasks are permitted to read from and write to memory, and access hardware devices via the system’s base software. For the purposes of this work, all hardware I/O is considered to be memory-mapped and processes communicate by means of shared memory alone. This reduces the complexity of the problem by only considering memory reads and writes.
2.3. TASKS

Tasks report their status to the operating system: a task may be schedulable (ready) or not schedulable (not ready or sleeping). A schedulable task will be assigned processor time when it becomes available. Tasks move from the schedulable to the not schedulable state based on internal logic; the work that a task does between becoming schedulable and becoming not schedulable is defined as a unit of work. Tasks move from the not schedulable state to the schedulable state based on operating system decisions.

Tasks may also be marked as terminated. A task that has been terminated is considered unable to ever again leave the not schedulable state and allows for any resources attached to this task to be released.

Tasks can be divided into two categories, depending on under what circumstances they are marked as schedulable: event-driven and periodic. Event-driven tasks are marked as schedulable by the operating system when a specific internal or external event occurs; for example, a timer expires or data becomes available on a given I/O channel. Periodic tasks are marked as schedulable at regular intervals. This work does not consider event-driven tasks, only periodic. Event-driven tasks are mentioned here for completeness; unless explicitly indicated otherwise, the remainder of this document will be referring to periodic tasks.

Each periodic task has associated with it some metadata that represents its timing characteristics. This metadata is static and is loaded by the operating system, commonly in the form of a scheduling table stored in a file. The operating system uses this metadata to schedule the task and ensure it stays within allowable parameters. The specific elements of metadata that are of interest are described in more detail below as well as being visualized in Figure 2.1.
• **Period** The task’s *period*, expressed in wall-clock time, is the interval at which the task becomes schedulable. The task is expected to complete one unit of work within a period.

• **Execution Window** A task’s *execution window* is usually the same as its period: It is the window of wall-clock time between the start of one period and the start of the next period during which a task must complete its unit of work. The start of the execution window is known as the baseline, and the end of the execution window is known as the deadline. In the common case where the execution window is equal to the period, the deadline of the current window overlaps the baseline of the next. However, it can somewhat simplify the discussion if they are considered separately.

• **Budget** The task’s *budget*, expressed in processor time, is the maximum amount of time that a task may consume to complete a unit of work. The consumed time is reset at the start of each period, i.e., the budget is defined per period. Therefore, the budget should not be greater than the period; and in fact, the budget generally is much shorter than the period. The budget may also be referred to more fully as the *execution budget*.

• **Priority** If a static-priority scheduling scheme is in use, the *priority* of the task must be provided to the operating system. If a dynamic-priority scheduling scheme is used, the priority will be calculated, though some static inputs may still be required. The operating system schedules the highest-priority available tasks to execute at each preemption point.
2.3. TASKS

- **Criticality** Whereas priority represents a technical concern, *criticality* represents a business or safety concern. The criticality of a task comes into play when the system is starved of resources (CPU time, memory, etc.): in these situations, resources must be granted preferentially to critical tasks.

![Figure 2.1: The terminology used to describe task timing and execution.](image)

In addition to the above properties, there are some additional properties that can be defined for a task but are not provided to the operating system:

- **Best- and Worst-Case Execution Time** Abbreviated *BCET* and *WCET*, respectively, these represent estimates and expectations for the amount of processor time that a task will take to complete a given unit of work. These values are calculated by employing automatic analysis on the source code or compiled binary, or estimated by engineers. There is a balance that needs to be struck here between resource utilization and a desire for schedulability guarantees. For
instance, the BCET or WCET may depend on the code going down a path that is only followed if some edge-case input is received. The BCET or WCET may be much different if outliers are discarded, but these outliers still represent valid potential execution paths. A decision that needs to be made when designing real-time systems is whether or not to support all tasks’ actual WCETs, or if it is acceptable to allow for periodic overruns which can be handled by protection mechanisms built in to the operating system. This decision is frequently determined by the criticality of the task or ECU.

- **Slack Time** Any time where one or more processor cores are idle is referred to as slack time. This happens in systems that are less than fully-loaded. More specifically, there is slack time for a given task in a given execution window if the task starts execution earlier than it absolutely needs to so as to be able to complete its unit of work before its deadline.

### 2.4 Schedulability

Given the definitions listed in the previous section, a definition for schedulability can now be given. This is a term that is used often in this document and thus warrants a dedicated section.

A schedulable system is one in which all tasks are able to complete one unit of work with each execution window, including cases where the tasks experience a fault which is within the acceptable range of faults. Depending on the requirements, schedulability is not necessarily required for correct operation, as will be discussed in later sections.

Should a task take longer that its budget to complete a unit of work, a budget...
overrun occurs. A task’s budget is a safety property, not one that directly determines schedulability. A task that has exceeded its budget is usually faulty or otherwise misbehaving. While the system may remain schedulable, some action may be required to correct the fault.

Should a task not be able to complete a unit of work within an execution window, a deadline overrun occurs. A system in which one or more tasks experience an overrun is not schedulable. See Figure 2.1.

2.5 Scheduling Algorithms

The operating system needs to determine the priority of all tasks before it can schedule some for execution. Static-priority schemes can have arbitrary priorities assigned by engineers, though in practice an algorithm is used to calculate the values. Dynamic-priority schemes must have an algorithm that can be implemented in software to determine the priorities.

There are many scheduling algorithms defined in the literature; some common ones that were considered are listed below. A graphical comparison of these algorithms can be seen in Figure 2.2.

- **Rate-Monotonic Scheduling (RMS)** The priority of a task is proportional to its period: A task with a shorter period will have a higher priority. If the period does not change at runtime, this is a static-priority algorithm. [19]

- **Deadline-Monotonic Scheduling (DMS)** The priority of a task is proportional to its execution window size; commonly this is the same as RMS. If the execution window size does not change at runtime, this is a static-priority algorithm. [19]
• **Earliest Deadline First (EDF)** The priority of a task is proportional to the amount of time left until its upcoming deadline. Tasks that are closer to their deadline are assigned a higher priority. This is a dynamic-priority algorithm, as the priority values change over time. [19]

• **Criticality as Priority Assignment (CAPA)** The priority of a task is proportional to its criticality level: High-criticality tasks will have higher priority than lower-criticality tasks. If the criticality level does not change at runtime, this is a static-priority algorithm. CAPA is not considered an efficient approach, as it achieves poor processor utilization. [8]

• **Zero-Slack Scheduling (ZS)** A hybrid algorithm designed to combine the efficiency of EDF, DMS or RMS with the safety-first approach of CAPA. Tasks will be assigned priorities according to EDF, DMS or RMS until it is determined that there is a risk that a high-criticality task will not be able to complete its unit of work by its deadline. At this point, the algorithm switches to CAPA. While most of the calculations for ZS can be done in advance, some runtime support is required. (Note: “EDF, DMS or RMS” is a choice that must be made ahead of time; the same algorithm is used for all tasks and does not change at runtime.)

Zero-slack scheduling is less well-known than the other algorithms listed. It is discussed in more details in Section 2.6.

Note that the convention is to assign a *lower* numeric priority value to a task that has *higher* priority; e.g., task $T_1$ with priority 1 will be scheduled in preference of task $T_2$ with priority 3. This convention is followed in this document. To alleviate
Figure 2.2: A visualization of how the five different algorithms schedule tasks; RMS, DMS and CAPA produced the same execution and are therefore combined to save space.

Top row is in regular (fault-free) execution, bottom row is with an overrun in the low-priority task.

The dashed lines represent ticks where the tasks are ready to execute, the blocks represent ticks where the tasks are executing, the shaded regions are the overruns and the horizontal bars are the deadlines.
the inevitable confusion, the numeric priority is sometimes known as the “niceness” value of a task or process – a task with a lower “niceness” value will less readily yield the processor to other tasks or processes (i.e., be less nice). This terminology was not used in the works that form the basis for this research, and as such will be avoided here. The reader is cautioned to pay close attention to descriptions of priorities and priority values.

2.6 Zero-Slack Scheduling

Zero-slack scheduling is a scheduling algorithm developed by de Niz et. al. of Carnegie Mellon University \[8, 16\] to address the criticality-inversion problem that may occur with mixed-criticality systems. The zero-slack scheduling algorithm stands on the observation that task criticality, and the associated protection mechanisms, are only applicable when the system is in a state where certain assumptions or limits have been violated (e.g., if a task’s execution budget or WCET are exceeded). A correctly-designed system operating within its parameters should never experience criticality inversion. It is therefore not necessary to apply protection measures until a violation of one of these parameters is observed.

The zero-slack algorithm guarantees that all tasks will be able to execute up to their execution budget so long as no higher-criticality tasks exceeds its worst-case execution time. To achieve this, task execution is divided into two modes: normal mode and critical mode. The task should always be in normal mode unless an overload occurs. In normal mode, protection measures are disabled and task scheduling is done using a priority-based preemptive scheduler (e.g., DMS, EDF) which attempts to maximize resource utilization. If the scheduler determines that the system has
reached a state where there are only enough free cycles available for a given task to execute up to its execution budget, it switches to critical mode. In critical mode, all tasks with lower criticality are suspended. At the instant where the switch occurs, no slack time remains and therefore this point in time is known as the \textit{zero-slack instant}.

In systems with predictable scheduling – either fixed-priority or dynamic-priority – the zero-slack instants can be pre-calculated and a timer mechanism can be used to perform enforcement at runtime. This makes this approach very efficient in terms of reducing run-time overhead. The authors provide a sample algorithm for rate-monotonic scheduling (RMS) in [8].

As the systems analyzed for this thesis did not have predictable scheduling – configurations could change at runtime – a simpler, two-criticality-level version of the algorithm was implemented and the calculations were performed dynamically. This is not necessarily something that can be implemented in a real embedded system due to excessive resource consumption by the scheduler (a claim made in the literature), however it was sufficient for proof-of-concept purposes.

2.7 Predictability

The term “predictability” has many closely-related but subtly different interpretations. For this reason, it is beneficial to spend some time defining what “predictability” means in the context of this work, otherwise there is a risk that further discussion of “predictable data flow” may lead to confusion and misunderstanding.

Predictability implies deterministic behaviour. That said, a system that behaves non-deterministically can still be predictable from a certain vantage point. For example, a multi-server system that includes load-balancing software that distributes
requests randomly among a pool of web servers can be equally called non-predictable and predictable: it is non-predictable because it is not possible to predict which server will serve the request but it is predictable from the point of view of the user – they are always served the same web page for a given address.

In his work on the subject [12], Henzinger defined four sources of non-determinism that are present in a software system: input non-determinism, unobservable implementation non-determinism, don’t-care nondeterminism and observable implementation non-determinism.

**Input Non-Determinism**  This type of non-determinism is present in any system that interacts with the environment. The system designer cannot influence or predict the inputs provided to the system. The user from the earlier example would be a source of this type of non-determinism: the system cannot predict the next web request.

**Unobservable Input Non-Determinism**  This is non-determinism that is present within the system but does not influence the uniqueness of the output. The load balancer in the earlier example is a source of this type of non-determinism.

**Don’t-Care Non-Determinism**  This is non-determinism that does affect the uniqueness of the output, but only in instances when the particular output value has no significance or can be discarded. From the web example: if the specific web server that generated the webpage also added a hidden identifier for debugging purposes, this would be an example of this type of non-determinism. Even though the output is not unique, the difference can be ignored.
2.8. PREDICTABLE DATA FLOW

Observable Implementation Non-Determinism This is non-determinism that affects the uniqueness of the output in a significant way. If the different web servers had different versions of the same page, that would be an example of this type of non-determinism. The user could not predict which version of the page would be returned for two identical requests.

When designing and analyzing predictable systems, only the fourth type of non-determinism—observable implementation non-determinism—needs to be considered. Input non-determinism is a property of the environment while unobservable and don’t-care non-determinism are inconsequential. Only observable implementation non-determinism can cause problems when analyzing systems for correctness.

This observation leads to the definition of predictability that will be used from here on in:

a predictable system is one which for each unique set of timestamped inputs, the system produces a unique set of timestamped outputs [12]

This definition places a restriction on both values and time, which is important in real-time embedded systems. In practice, the business rules allow for the restrictions on time to be somewhat flexible, though for the purposes of simplifying the analysis this flexibility is not taken into consideration.

2.8 Predictable Data Flow

“Data flow” refers to the path of data through the system. That is, inputs coming in and outputs going out. While most of this work focuses on scheduling and schedulability, it is not predictable scheduling that we are after. In fact, we are trying to
avoid having to enforce predictable scheduling, as allowing the scheduler to be more flexible when faced with changing conditions can produce systems that achieve better hardware utilization. The decisions made by the scheduler are a form of unobservable implementation non-determinism: when analyzing predictability, we need to treat the behaviour of tasks and the operating system as a black box, and only consider the inputs from and outputs to the external environment.

It may seem surprising that a system with a known set of tasks and fixed scheduling policy will experience variability in timing and processor load. In theory, a given set of input values should always result in the same code path which should always take the same amount of time to execute on a given piece of hardware. However there are two sources of variability that must be considered: unpredictable environmental influences, and redeployment on different hardware.

The environment in which the code is executing may influence the execution timing. For instance, the processor may select a lower clock speed if the ambient temperature is high so as to prevent overheating. Other software events that occur may also influence the timing, e.g., if the system is busy handling an interrupt or performing some maintenance operation.

The second source of variability is redeployment onto different hardware. While this is an obvious case of variability, it would still be useful to not have to repeat analysis and verification processes for these sorts of changes. Should a vendor suddenly not have stock of a given component, having to repeat verification processes might be an unwelcome hindrance to the business.

A basic real-time operating system, such as those currently used in many vehicles, does not have predictable data flow. Tasks read from and write to memory as needed,
with no guarantees on the order or timing of these events.

The programmer, of course, has control over when data is read in and written out when the task is performing its unit of work. To reduce the potential for conflicting inputs to the task, programmers can arrange the code so that all inputs are read at the start of the code representing the unit of work and outputs are written at the end of this code. The view of the outside world within the code will therefore be consistent, making the task of reasoning about the operations the code is performing less difficult.

However, this approach cannot guarantee true predictable data flow. The programmer cannot control when the task is scheduled; that is up to the operating system. To achieve predictable data flow, communication within the system and between the system and the outside world must happen at pre-defined instants measured according to a global clock. This approach eliminates the variability that can be present in the timing of task execution.

The approach toward data flow handling just described is the “full observability” approach: at any given time, each task has full view of the current state of the data written by all other tasks. It is difficult to guarantee predictable data flow in such a system.

One way to ensure predictable data flow is to limit observability, as in PharOS and Giotto. In a limited observability approach, the data that a task sees are limited by the operating system. This is the basis of the implementation that was analyzed in this work.

The limited observability approach uses the existing task baselines and deadlines as the points at which data becomes visible to other tasks. As baselines and deadlines
always happen at the same instants in time according to the global clock, they are 
unaffected by timing variations that can be introduced by the sources mentioned 
earlier.

In this approach, tasks do not directly access shared memory. Instead, task reads 
and writes are captured by the operating system. The operating system ensures that 
tasks read values as they were at the baseline and that writes performed by the task 
are not made visible to other tasks until the writing task’s deadline. Writes happen 
before reads, and coincident writes are handled in an implementation-defined but 
completely deterministic order. Combined with the requirement that each task must 
perform one and only one unit of work within an execution window, the time and 
value of each read and write will always be identical for a given set of input values. 
This is the definition of predictability.

A more detailed explanation is provided in Chapter 4. A visualization of the 
difference between a full observability and limited observability approach can be seen 
in Figures 2.3/2.4 and 2.5/2.6 respectively.

2.9 AUTOSAR

An example of an operating system design and specification for automotive embedded 
systems is the AUTOSAR standard. The AUTOSAR standard is an open standard 
developed by a consortium of major automotive manufacturers (OEMs) and suppliers 
. This specification is used to design and build software that runs in vehicles.

\[http://www.autosar.org/\]
The AUTOSAR standard, at version 4 at the time of this writing, does not implement any limited observability techniques nor does it provide predictability guarantees. However, it was used as a reference for the state of the art in embedded operating systems in the automotive domain as it is used by many manufacturers. Claims made about how things are done in the field are made specifically with AUTOSAR in mind.

2.10 Exhaustive Verification

To verify a property such as predictability, an exhaustive verification technique was employed. This is a well-known and proven approach, and there exist many software tools (including Spin, which was used) that support such verification.

The control flow of a given software system can be broken down into a finite collection of states and transitions between these states. Additionally, the data stored by this system can be represented as a vector of values, known as the state vector. If a state has two or more outgoing transitions that are all active (i.e., the conditions for taking these transitions are satisfied), this represents a non-deterministic choice: the system can equally proceed down any of the paths represented by the active transitions. Starting at the initial state, a search tree can be built, with each branch representing a non-deterministic choice and each node consisting of the control-flow state and state vector. Verifying a property is a matter of exploring this search tree and ensuring the property holds at each node. This is the “exhaustive” part of “exhaustive verification” – all possible states and paths are explored.

For a non-terminating system, such as the ones under consideration here, the search tree is technically infinitely deep. However, since the set of states used to build the search tree is finite, there will eventually be repetition: after enough time,
the system will end up at a node that is identical to one it has seen previously. At this point, the exploration of the given branch can stop as there is no new information to be gathered by exploring further.

The search tree may be explored breadth-first or depth-first. The choice of algorithm depends on the nature of the problem, ideally the algorithm will uncover cases where the given property does not hold relatively quickly. In the case of predictability verification, it was found that a depth-first strategy yielded better results.

2.10.1 Spin

The predictability verification was done using the Spin model checker [15]. Spin is a reliable tool that is well-known in the field: it has been around for over 25 years and is still actively maintained. It reads text-based input written using Promela (process meta-language), and generates the corresponding verifier code in C. This code can then be compiled and executed resulting in very fast verification runs. Spin supports parallel verification on multi-core or multi-processor systems.

Spin runs on Linux, OS X and Windows through the use of the Cygwin POSIX compatibility environment. This makes it a good choice if portability is a concern.

Unlike regular imperative languages such as C or Java, Promela is a language that is designed for expressing state machines in a way that is familiar to imperative programmers and does not require the use of graphical interfaces. Also supported is the embedding of C code fragments into the Promela models, allowing the expression of properties and behaviours that are not supported in vanilla Promela such as pointer manipulation. The embedded C code is included verbatim in the generated verifier source.
2.11 Conclusion

This chapter provided a brief overview of the various topics touched upon in this project. This includes task scheduling, predictability and verification of software models. The next chapter contains a discussion of related work.
Figure 2.3: A single-core system with three tasks (left to right): Task 1, Task 2 and Task 3. The tasks have periods 10, 8 and 12 and budgets 3, 2 and 3. Relative priorities are assigned according to DMS: Task 2 > Task 1 > Task 3. Task behaviour is very simple: they just copy their input to their output, with no transformation.
Inputs (on left) occur at the exact same points in time for each execution. The outputs (on right) vary.
This figure displays one possible valid execution, with no task violating any constraint (budget or deadline). Figure 2.4 displays a different, but equally valid, execution.
Figure 2.4: A different, but equally valid, execution of the task set described in Figure 2.3. All tasks still meet their deadlines and budgets. With full observability: despite there being no change in the timing or value of inputs, the outputs differ in both value and timing.

<table>
<thead>
<tr>
<th>Input</th>
<th>t</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td>29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 2.5: The example from Figure 2.3 with limited observability.
Figure 2.6: With limited observability, in the example from Figure 2.5 neither the output values nor their timings change.
Chapter 3

Related Work

3.1 PharOS

The conceptual operating system and data flow synchronization mechanisms used in this work are taken from PharOS. PharOS is a full description of an operating system for embedded automotive applications that has been developed by Aussaguès et. al. of CEA LIST [7]. There has been work completed on creating a full, AUTOSAR-compliant implementation of PharOS [5].

Many of the behaviours exhibited by the system under study in this thesis were taken from PharOS, but this system is not PharOS or a derivative. PharOS can be seen as more of a source of inspiration than a foundation in this regard.

A full discussion of PharOS would not be productive, as the parts that are of interest are essentially the same as what has already been described in Section 2.8 with some variations in terminology. Significant differences between PharOS and the system that was studied are discussed below, though this is not an exhaustive list. For more information, please consult the referenced literature.
3.1. PHAROS

Level of Detail  PharOS provides a full definition of an operating system, and therefore it provides a considerably more granular definition of the various parts – tasks, scheduler, operating system, etc. This includes defining how the operating system is implemented, how protection mechanisms function, and even what system calls are available. The work described in this thesis is at a much higher (less granular) level, and therefore few concrete assumptions are made and limiting factors of the implementation considered. The capabilities put forward as being possessed by the operating system in this thesis may be beyond what is actually possible in a PharOS implementation.

Event-Triggered (ET) tasks  PharOS supports two types of tasks: time-triggered, which are activated by a timer; and event-triggered, which are activated by an interrupt event. Only the former were considered here.

Behaviour monitoring  PharOS supports monitoring of task behaviour based on pre-determined execution patterns. This was not considered by the work described in this thesis.

Quota and deadline overruns  PharOS does not permit tasks to exceed their quotas (execution budgets) or overrun their deadlines. This work explicitly does allow this as this was one of the areas of interest for the research.

Advanced temporal constraints  PharOS supports various temporal constraints to be defined on tasks and data values, such as the maximum age of a value or minimum time between executions. These are monitored or enforced by the operating
3.2. GIOTTO

The typical embedded-systems development process happens in two phases: a high-level control design phase, during which a control engineer designs the system using mathematical models; and a low-level code-writing phase, during which a software engineer takes the control engineer’s designs and implements them in platform-specific code. Giotto introduces an intermediate level of abstraction, in the form of a language, compiler, library and architecture definition. The aim of Giotto is to take the high-level control design and create a platform-agnostic intermediate-level design, which contains definitions of software tasks and timing. The software engineer can use automation to perform automatic platform-specific code generation from Giotto, and then simply “fill in the gaps”. In this way, Giotto reduces the amount of low-level work required, reducing the chance for errors to be introduced at this stage. Additionally, it provides a common language for both the control engineer and software engineer, thereby facilitating communication.

Giotto only supports time-triggered, periodic tasks.

The work done on Giotto is very interesting, though it is only tangentially related to the work described in this thesis. Giotto is referenced here since it takes the same approach to predictable data flow as PharOS and the description in Section 2.8. Specifically:

The Giotto semantics requires that a Giotto task reads the values in its
3.3. SCHEDULABILITY VERIFICATION

input ports exactly at the time of its invocation and writes its results to its output ports exactly at the end of its period. [13]

This matches the definition given in this thesis, with perhaps slightly different terminology.

The relationship between the system under study and Giotto is much the same as the one with PharOS – inspiration rather than foundation.

A formal definition of Giotto is available in [14], and a less-formal description can be found in [13].

3.3 Schedulability Verification

Verifying schedulability through formal methods and model-checking is a well-explored research area.

Toshiaki Aoki describes the implementation of method to verify software execution on a real-time operating system conforming to the $\mu$ITRON specification [24]. While different in details, the same general approach was followed as in this thesis. Aoki also used Spin and Promela, and implemented reusable Promela components. More details can be found in [2].

Fang et al. describe an approach to modelling the AUTOSAR operating system in [9]. They describe the creation of an abstract model which was in turn implemented in Promela and tested against dynamically-generated test cases. This is a higher-level (more abstract) approach.
3.4 Conclusion

This chapter provided an overview of related work, some of which was used as the basis for this project. The next chapter begins the discussion of the project itself by describing the design of the operating system that was studied.
Chapter 4

Problem Definition

The research problem that this work tries to address is the requirement to be able to verify the predictability of data flow in a system that consists of a set of tasks and an operating system that supports the limited observability behaviour discussed in Section 2.8. While such systems will be predictable if all tasks abide by the rules and limitations stated therein, it is not known ahead of time if any given set of tasks and timing properties will abide by these limitations. An exhaustive verification will prove this.

In this chapter, the system that needs to be verified is described. This does not necessarily represent an existing system that is found in industry, but rather a theoretical approach to task scheduling that could be implemented for use in the automotive domain. The ideas for the system were sourced from the work done on PharOS and Giotto and from experts in the industry. Certain parts of the system description were then extended or fleshed out to create a full definition encompassing all edge cases.
4.1 System Description

The system under consideration consists of a set of periodic tasks scheduled by a real-time operating system and executing on one or more processor cores. The operating system maintains a global, monotonically-increasing clock. This clock is incremented once per scheduling tick, though this is an abstraction as the modelling is not done at the level of timers and interrupts.

The operating system assigns each task an execution window during which the task must complete a unit of work. Most commonly, this involves the task code executing from start to finish; however, suspending and restoring a task’s state allowing for partial execution is also supported. Note that this is not preemption; tasks may choose to suspend themselves temporarily in much the same way as in a cooperative multitasking system.

As the tasks are periodic, when one execution window ends, the operating system grants the task another execution window. There is no gap between execution windows (i.e., the deadline of one execution window becomes the baseline of the next; or, the relative deadline is equal to the period), though gaps do not affect the guarantee of predictability. The initial execution window granted to the task may be delayed by an initial offset after system start-up to allow for staggering of task starts.

The operating system is free to schedule the task anywhere within the execution window. Tasks are scheduled according to the scheduling algorithm in use, relative priorities of other tasks, and available resources. It is important that the execution window size is much longer than the amount of processor time required by the task to complete its unit of work, to allow for other tasks to use the processor. Tasks are assigned execution budgets on a per-execution-window basis. The operating system
can be configured to enforce budgets and take some action if a task exceeds its budget.

Each execution window for a given task is specified with three parameters, known as a configuration: the execution window size, the budget and the criticality level. The budget is generally proportional to the execution window size, and as such will usually need to change when the execution window size changes. The criticality level does not have an effect during regular operation, but can be used in an overload or fault condition to prioritize tasks for load-shedding.

In theory, each task can have an infinite possible number of configurations. As this is impractical for verification and determining schedulability, the number of valid configurations is limited and defined in advance. The operating system stores a static lookup table containing all the possible configurations for each task. Each task may, during execution, request one of these configurations be used for the next execution window – the configuration of the current window cannot be changed. If a task does not request a new configuration, it is assumed by the operating system that the currently active configuration is still acceptable.

Communication between tasks is aligned to the baselines and deadlines by the operating system in a way that is entirely transparent to the task. No matter where the task is executing within its window, it will see an identical “snapshot” taken at its baseline of the values in shared memory written by other tasks. Any changes made after the task’s baseline will not be visible within the current execution window. Similarly, any values written to shared memory by the task will not become visible to other tasks until the deadline of the writing task. Writes are applied in the order of the deadlines, rather than when the task actually writes the value (see Figure 4.1). If a deadline and a baseline are coincident (i.e., a write and a read of the
same location in shared memory), the write is to happen before the read. If two deadlines are coincident (i.e., two writes of the same location scheduled during the same scheduler tick), the order of writes is left up to the implementation but must always be consistent for a given pair of tasks.

Figure 4.1: Task 1 produces a value before Task 2, but Task 2’s is the first to become visible because its deadline is sooner.

Tasks can request termination at any point within an execution window. The operating system immediately terminates the task and does not schedule it again within the window, however it still waits for the task’s deadline before making any values written to shared memory available to other tasks. Similarly, it does not make known to other tasks that the task in question has terminated until the deadline. A
terminated task is not granted any further execution windows; it is assumed that the
task is done and will not need to be executed again until the system is restarted.

The system as described here does not include blocking operations. Due to the
limited observability approach, the state of a shared resource as seen by a task is
always the same during its execution window. For this reason, it does not make
sense to block waiting on the resource: it will not become available until the task’s
deadline at the earliest. This prevents deadlock, but may result in starvation. This
is acknowledged but not analyzed in this thesis.

Overhead, including time for the scheduler to run and context switch time, is also
not considered. While these are important factors to take into account when building
an actual system, they would only serve to complicate the analysis here.

4.2 Scheduling

Tasks are scheduled on one or more symmetric processor cores. For simplicity, tasks
do not have affinity for specific cores and migrating a task between cores does not
have any associated cost. Task priorities are assigned dynamically, and recalculated
by the operating system scheduler at every preemption point. After the priorities are
recalculated, the top \( n \) tasks are assigned to the \( n \) processor cores. Should multiple
tasks be assigned equal priorities, it is up to the implementation to determine which
to schedule.

Three different scheduling algorithms were modelled:

- **Earliest-Deadline First (EDF)** EDF is the optimal scheduling algorithm for
  this type of system [19]. The priority is inversely proportional to the number
  of scheduler ticks remaining until the task’s deadline.
• **Dynamic Deadline-Monotonic Scheduling (DMS)** Regular DMS utilizes static priorities. The concept behind “dynamic” DMS is exactly the same; the “dynamic” qualifier emphasizes that the priority for a given task can change when the configuration changes. Each configuration for a given task can be assigned a priority value. Unlike the other two algorithms, the priority remains the same within an execution window, i.e., the absolute priority of the task is the same at the start of the window as at the end. The priority is inversely proportional to the execution window size.

• **Simplified Zero-Slack (ZS)** The zero-slack algorithm was simplified to use only two criticality levels so the zero-slack instant could be efficiently recalculated at each preemption point. Priorities are initially assigned using EDF in “normal mode”, however in “critical mode” all high-criticality tasks are scheduled before all low-criticality tasks. Within a given criticality level, EDF is used.

### 4.3 Faults and Fault Handling

There are three possible failure modes that the system can experience: task failure, budget violation and deadline violation.

#### 4.3.1 Task Failure

Task failure is the most obvious failure mode: a task either incorrectly calculates its outputs based on its inputs or suffers some other catastrophic failure, such as an unexpected termination.

Handling this type of failure is beyond the scope of this work. It is assumed
that the tasks provided have been sufficiently tested and will not crash or attempt to interfere with other tasks or the operating system. No provisions are made for handling a task that has terminated unexpectedly.

4.3.2 Budget Violation

The budget defined in the configuration of the execution window represents the maximum amount of time that a task should execute within an execution window in correct operation. Exceeding this budget is considered a fault of the task itself (incorrect operation) or of the system developers (incorrect budget).

The operating system monitors tasks for budget violations. If a violation is detected, the operating system can take one of two possible actions: either the budget violation is ignored or the offending task is terminated.

Ignoring the violation is possible as the task set should not be designed to use 100% of the available processor time. Ideally, the overrun will eventually be absorbed by the available slack (idle ticks) and the system will return to normal.

Terminating the task ensures that the assumptions made by the operating system when scheduling the tasks continue to hold, however the behaviour of the system as a whole could become wildly incorrect. It is possible to handle budget violations of critical and non-critical tasks differently. For instance, the system may be more sensitive to the unexpected termination of a critical task and may elect not to terminate a critical task on a violation for which a non-critical task would have been terminated.
4.3.3 Deadline Violation

A task must complete a unit of work before its deadline. Violating the deadline (i.e., not completing a unit of work) is considered a fault, but not necessarily of the task in question: it is possible that the task missed its deadline due to other tasks misbehaving, or due to an incorrect configuration (e.g., the task set requires more processor time than is available).

Deadline violation is more serious than budget violation and some compensatory action must be taken by the operating system in all cases in order to preserve predictability. The simplest option is to terminate the offending task, with the same caveats as in the budget violation case.

The other option is to delay the deadline until the task has finished executing. In this case, the current deadline and the subsequent baseline are delayed by the operating system until the task enters a completed state. The next deadline is not delayed: rather, the next execution window is shortened. This is described in more detail in the next section.

4.4 Sporadic Deadline Overruns

Rather than use strict deadline monitoring, it is possible to allow sporadic deadline overruns and still preserve predictability. In practice, a deadline overrun would happen so infrequently that the system could conceivably recover without any ill effects. Having all tasks meet their deadlines is a sufficient condition for predictability, as shown by PharOS. However, it is not a necessary one: if the overrun can be absorbed such that the final outputs still occur when they are supposed to, predictability can be preserved despite schedulability being violated due to the overrun. The chances
that an overrun can be absorbed are good, as overruns are generally very short and there should always be some slack (idle time) in the system that can be used to absorb the overrun.

To handle sporadic deadline misses in a way that preserves predictability, baselines and deadlines of all tasks in the system must always occur in the same relative order. This guarantees that each task still sees the same snapshot of shared data at its baseline. The operating system achieves this by delaying any baseline and deadline of any task that occurs during the overrun, and then applies them in the same order in which they would normally occur after the overrun has completed. The next windows of these tasks are also shortened in the same way as the next window of the overrunning task.

An example of this behaviour is shown in Figure 4.2.

By shortening subsequent windows, it is possible that a cascade of deadline violations will be created. However, much like in the budget violation case, the slack present in the system should ideally cause the overruns to be absorbed and the system will return to normal operation. See Figure 5.1.

For this mechanism to work, the system cannot operate under full load. In addition to the overrun itself, during the overrun period certain processor cores may remain unused. A safety margin must be included in the system design, with more processor capacity available than is required by the tasks under normal operation (though still less than in systems with run-time guarantees). The exact size of this safety margin is dependent on the allowable overrun; this must be decided by the system developers.
Figure 4.2: Task 2 overruns its deadline (diagonal lines) and causes a delay (arrow; to thicker line). The baseline of Task 1 occurs within this overrun period and it is delayed as well. The deadlines of Task 1 and Task 2 remain at the same position relative to their respective original deadlines, with the effective sizes of the windows reduced appropriately.

4.5 Conclusion

This chapter described the design of the operating system scheduler for which predictable data flow needs to be verified. The techniques that were employed to perform this verification are covered in the next chapter.
Chapter 5

Modelling and Verification

This chapter will cover the approach used and challenges faced in taking the system definition given in Chapter 4 and creating a model that can be exhaustively verified in a reasonable amount of time and without using an excessive amount of memory.

The first thing to note is that this type of system is predictable by design, for any task set, so long as all tasks complete before their deadlines (i.e., the system is schedulable). This includes cases where execution budget violations occur if this in turn does not also cause deadline violations. This was shown by PharOS and is discussed a bit more in Section 6.3.2. It will be taken as granted moving forward.

The more interesting cases are those where the task set and fault model are not schedulable. These task sets could still be predictable: deadline violations could be absorbed by utilizing idle processor time or output values of certain task could be ignored in the given execution mode. Exhaustive verification is required to determine this.
5.1 Guarantee

The guarantee given is a design-time guarantee of predictable data flow. The guarantee given is for a specific set of tasks, timing values and hardware configuration. Additionally, it is only given up to a certain worst-case scenario (the fault model). No guarantees are made for cases where the maximum allowable fault count and timing are exceeded.

5.2 Model Definition

Before progressing to the discussion of verification techniques and the design of the model, it is important to define what the inputs are that need to be provided. This set of inputs is referred to as the “input system”. It is comprised of:

- The number of symmetric processor cores available;
- The scheduling algorithm in use;
- The task set; and
- The fault model.

The “task set” is comprised of:

- The static configuration table, each configuration consisting of execution window size, budget and criticality;
- The initial offset before the first execution window; and
- Sufficient logic to determine when and how to modify output values based on input values.
Finally, the “fault model” is comprised of:

- The maximum number of tasks that will violate their budgets; and
- For each configuration of each task, by how much the task will exceed its budget should it enter a fault state.

The verification tool needs to be able to accept the definition of an input system as a configurable parameter or set of parameters.

5.3 Verification Techniques

The basic task of the verification process for a given input system is to exhaustively check that for each set of valid, timestamped inputs the system always produces the same set of timestamped outputs – with all possible combinations of faults allowed by the fault model. The verification process either concludes that the system is predictable in all cases, or provides a counterexample in which predictability is not preserved. Part of the challenge involved in this task was determining the best approach for accomplishing this task: which properties should the model checker be looking for?

5.3.1 Direct Approach

The most direct approach for checking that a given input system satisfies the criteria for predictability would be to check it directly: take all possible sets of timestamped inputs, combine them with all possible failure modes, and check that the same set of timestamped outputs is produced for each set of inputs.

This is not as daunting a task as it may seem at first. The tasks are periodic and the number of configurations is limited. The system will eventually start to repeat
5.3. VERIFICATION TECHNIQUES

previously-explored states at which point the search can stop. Additionally, as task
behaviour is assumed to be correct (verifying tasks is the responsibility of the task
developer), the full range of possible inputs does not need to be examined. It is
sufficient to only use a small set of values; only enough values are required such that
it can be unambiguously determined which inputs produced which output.

This approach, while theoretically possible, consumes much time and resources
as well as being difficult to implement. Verifying that the system produces a pre-
determined set of timestamped outputs is simple; however, verifying a less direct
property such as “the system always produces an identical set of timestamped out-
puts” is more difficult. It would necessitate modifying the operation of the model
checker and therefore require a proof of soundness as the one guaranteed by the au-
thors of the model checker would no longer hold. This is covered in more detail in
Section 6.2.

5.3.2 Indirect Approach

An alternate approach that is equally correct but much more efficient is to verify
predictability indirectly. As mentioned earlier, the system is predictable by design
when it is schedulable. We can disregard the case where a fault-free execution is
not schedulable, as that only happens with a mis-configured system. Therefore, to
verify predictability it is necessary to consider which behaviours the system exhibits
in a case where a fault has occurred, and which of those behaviours will result in a
violation of predictability. The three points that make up this approach are detailed
below.
5.3. VERIFICATION TECHNIQUES

Inputs only happen at read times. Each input and output value has two parameters associated with it, the modification of which would result in a violation of predictability: data and time.

The data value of inputs is beyond the influence of the system and is assumed to always be correct and valid. This is a form of input non-determinism, which, as already mentioned, is not considered for predictability.

The timing of input arrival is technically beyond the control of the system as well; however, the system controls when it reads in the value of the input parameter. This is a reduction of many possible cases to one: the input can arrive anywhere between reads, but all these scenarios are effectively equivalent to the input arriving at the read time. If the system is predictable with inputs that are timestamped at the read time, it is also predictable when the actual arrival times are used.

Both the data value and timing of output values can be directly affected by system behaviour.

Only monitor “edge” task timing. All reads and writes happen at a task’s baseline and deadline, respectively. All reads of input values and writes of output values happen at the baseline of input tasks and the deadline of output tasks, respectively. It is therefore possible to verify data flow predictability properties of the system by monitoring these “edge” tasks.

As we have neither control over nor method of predicting the input, we must assume the worst case where it is constantly changing. Every time you read from the input, the value is different. Moving the baseline of an input task may result in a violation of predictability, as a different value will be read in at the baseline’s new time. This is not a guarantee that predictability is violated, but is a sign that the
configuration needs to be analyzed further.

On the other hand, if an output task deadline gets moved, there will be at least one point in time where an output channel will not contain the correct value, which is a violation of predictability. (Note: in the systems being considered, only one task can be writing to a given output channel at the same time – writes are queued. It is possible to devise a contrived example where a system is predictable even when an output task deadline gets moved if this restriction is lifted.)

Therefore, if either input baseline or output deadline has to be moved to satisfy the data flow requirements, it results in a violation of predictability. See Figure 5.1 for a visualization.

**Flag outputs of forcibly terminated tasks** In the most permissive operating mode – where execution budget violations are ignored and deadline violations result in delays – checking data values is not necessary as the system guarantees correct data flow by eschewing timing enforcement. However, in a more restrictive mode that involves task termination, the data values do need to be checked. The check is for one specific property: was a task that was terminated by the operating system involved in calculating this value? If a terminated task was involved anywhere in the chain, the output is no longer reliable as it is not guaranteed the task completed the calculation successfully before being terminated.

It is sufficient to add a “tainted” flag to data values as they are passed between tasks. This tainted flag is set when a task that is supposed to write a given value is forcefully terminated and it is in turn propagated by all other data-dependent tasks. If a tainted value is observed among the output, it results in a violation of predictability.
5.4 Conclusion

This chapter covered the design of the model used to verify predictability. It included a discussion of the parameters required, the techniques used to verify, and the properties that are checked. The actual implementation of these verification techniques is covered in the next chapter.
Figure 5.1: Two tasks: Task 1 and Task 2. Task 2 overruns its deadline. The delays cascade, and eventually get absorbed by the slack in the system.
Chapter 6

Implementation

The main contribution of this work is a tool that could be used to verify predictable data flow for arbitrary input systems. The tool was created as a proof of concept, and guided in its development by the requirements collected through collaboration with General Motors. The tool accepts an input system definition as input and performs an exhaustive verification: returning either a counterexample showing a predictability violation, or indicating that no predictability violation could be found.

The layout of this chapter mirrors the timeline followed during the development of the tool. The work involved three main programming tasks: extending Spin, creating a graphical interface, and creating the “skeleton” Promela code that represents the operating system.

6.1 Choice of Model Checker

From the start, it became clear that this was a problem that existing model checking tools could be made to solve. However, there are many such tools available out there, all tuned for a slightly different problem. The basic requirements for a model checker were:
6.1. CHOICE OF MODEL CHECKER

1. **Ease of integration with other tools.** A text-based, command-line interface was preferred.

2. **Performance.** Fast execution and low memory consumption were desired, as well as the potential for parallelized exploration.

3. **Portability.** The tool would need to execute on Windows, OS X and Linux.

4. **Familiarity.** Preference was given to tools which were familiar or for which there was easy access to the required expertise.

These conditions narrowed the set of options down to the following five:

1. A custom implementation in Java

2. A custom implementation in C

3. UPPAAL

4. Java Path Finder (JPF)

5. Spin

**Custom Java Implementation** A custom implementation in Java would satisfy most of the requirements, and not have any overhead involved with attempting to fit a general-purpose tool to a specific problem. A custom implementation would also allow for detailed visualizations and result output. Java is a familiar, high-level language that would make development quick and easy. It is cross-platform, with only minor portability issues.
On the other hand, a custom Java implementation would take time to develop and would need to be proven correct. Additionally, the performance characteristics of Java are not as good as those of lower-level languages such as C. Additionally, the available sample task code is written in C, so would need to be ported to Java.

**Custom C Implementation**  A C implementation has a similar cost/benefit profile as a Java one. A C implementation would take longer to develop as C is a lower-level language; however, this generally means better performance. Additionally, a C implementation would not require porting of existing task code to a different language.

**UPPAAL**  UPPAAL is a model development and verification tool and environment designed explicitly for real-time (or time-sensitive) systems. UPPAAL seems like a good candidate on the surface, however it was disregarded for several reasons.

First, UPPAAL is designed for time-sensitive systems and has a notion of continuous time. Predictability verification does not require this: our basic unit of time is the scheduling tick (or time slice), which is an entirely artificial construct. All times are expressed as an integer number of scheduling ticks, and the entire simulation moves forward in fixed steps. We do not need to simulate events that happen between these fixed time points.

Second, UPPAAL is not familiar. Learning and adapting the system would have been required, and the available documentation is not always sufficient.

Third, integration of UPPAAL into other text-oriented tools was determined to be more difficult than some of the other options, even though it was possible.
Java Path Finder  Java Path Finder is a tool that checks existing Java applications as written. It is a very powerful tool, but primarily targets verification of production code and not purpose-built models. Early attempts were made at creating a schedulability verification tool backed by JPF. It was primarily its performance characteristics that ruled it out as a solution.

Spin  Of all the options surveyed, Spin seemed to be the closest fit to the requirements: text-based interface, performant verifiers, portability and familiarity. However, Spin was not perfect: it did not really support inspection of the verification process as it proceeded, and required porting existing logic to Promela from C. It would be later determined that modifications to Spin were not simple to make due to the highly-optimized nature of the code.

Spin was the final choice; it was the model checker used in the rest of the project. Reflections on this choice are provided in Chapter 8.

6.2 Initial Exploration

The initial exploration of the problem took place before the indirect verification properties (described in Section 5.3) were developed. Indeed, it was the failure of these initial attempts that led to the development of the indirect approach.

The initial exploration attempts involved only Spin. The task code and skeleton code wrapping the task code and representing the operating system were written and updated by hand. The SpinRCP development environment from the University of Maribor [6] was used to make the process less time consuming.

The immediate problem that was encountered was that there was no way to describe the predictability property in a way that Spin could understand. If the desired
outputs were known ahead of time, it would be simple: ensure that variable $x$ has value $y$ when $t$ scheduling ticks have elapsed. This can be expressed as a simple assertion or LTL formula, understood by Spin. However, the property that needed to be verified was that variable $x$ always has the same value when $t$ scheduling ticks have elapsed. This does not easily fit into the search-tree model: the different nodes of the tree would need to share data, and maintain global state. While Spin could be made to do this by utilizing some unsafe features of Promela, soundness of the verification would no longer be guaranteed – since the verifier is unaware of what is happening, it would no longer be a given that all possible paths had been explored when the search terminates.

A rudimentary version of the model was developed using the direct properties. It used unsafe Promela features that allow the storing of global state outside of the purview of the exploration mechanism. It also relied on implementation details relating to the depth-first exploration of the search tree and the way non-deterministic choices were made: specifically, a non-deterministic choice within a range of values will always first explore the lowest value, then the second-to-lowest, and so forth. The same tree could therefore be explored twice, storing a global value in the first pass and comparing against it in the second.

This approach proved to be both very slow, memory intensive and possibly unsound, that is, the guarantee that all possible paths would be explored during the exhaustive search possibly no longer held as the mechanisms used by the model checker to ensure this were being subverted. It was decided that it was much too inelegant, so the implementation did not proceed any further and an alternate approach was sought.
6.3 Final Design

After the failure of the initial design, the problem was analyzed more closely to determine if predictability can be verified indirectly.

6.3.1 Assumptions

To arrive at the observations listed in the next section, certain task properties and behaviours must be assumed. Tasks are assumed to be “well-behaved” – that is, they do not attempt to subvert the protection measures in place and their execution is entirely deterministic. A task that uses random numbers in an externally-visible way would not be considered well-behaved by our definition. It is presumed that such tasks are uncommon, as they would be more difficult to test thoroughly.

6.3.2 Observations

The first observation relies on the fact that if all tasks always satisfied their deadlines, the design was correct-by-construction and no analysis is needed. A simple way to become convinced of this fact is to disregard the actual task execution entirely: if we assume the tasks will always meet their deadlines, we know exactly when the data reads and data writes will happen – at the baseline and deadline, respectively. The data flow is entirely deterministic and predictable through a system where neither baselines nor deadlines are changed non-deterministically.

This first observation is, therefore, that only cases where deadlines or baselines are moved can result in a predictability violation.

The second observation builds on the first and ties into the definition of predictability used (see Section 2.7). The operating system will delay deadlines in case
6.3. FINAL DESIGN

of a fault to ensure the order of reads and writes is always consistent, giving preference
to correct values over correct timing. We do not care what happens between the input
and the output: this is a form of non-observable implementation non-determinism.
Therefore, we can safely ignore the behaviour of tasks that are not “edge tasks”, i.e.,
visible to the outside world. Since the operating system guarantees the correctness
of values and we only need to consider edge tasks, the only time a predictability
violation can occur is when the baseline or deadline of an edge task is moved.

The second observation refines the first: only cases where deadlines of tasks that
write to the output or baselines of tasks that read from the input are moved can
result in a predictability violation.

The third observation ties into the limited observability property of this design.
Tasks that read from the input effectively sample the input values at their baseline.
Any changes to the input values that happen between consecutive baselines have no
effect on the behaviour of the system as they are never read in. They can thus be
ignored, and we can safely collapse all the cases that consider changes to the values
or timing of inputs between the baselines into one: the case where the given value
arrives at the baseline.

The third observation is that inputs can be treated as always and only arriving
at the baseline of the task that will read them in.

The fourth and final observation handles a specific behaviour that may be ex-
hibited by the operating system and that could violate predictability in a way that
wouldn’t otherwise be caught: premature task termination. The operating system
may, for whatever reason, force a task to terminate unexpectedly or uncleanly. In this
case, no baseline or deadline will be moved but predictability may still be violated as
6.3. FINAL DESIGN

the task may not have completed its work. Any downstream task that depends on the output of the terminated task could in turn produce incorrect output itself; this may cascade all the way to the final outputs of the system. The outputs of a terminated task must be flagged as such, and these flags need to be propagated through any intermediate calculation. If the final outputs of the system are so flagged, this may indicate a predictability violation.

The fourth observation is that unexpected task termination can result in a predictability violation.

6.3.3 Changes to Model Design

Based on the above observations, the final model of the operating system added flags to the edge tasks – input and output – as well as a “tainted” flag to data values which would be propagated through intermediate calculations. This was inspired by the way the special NaN value is propagated through calculations involving IEEE 754 floating-point numbers [1].

During verification, if the simulated operating system needed to delay the baselines or deadlines of the flagged values, an error would be raised. Similarly, if a tainted value was seen in the output, an error would be raised. These do not represent a guaranteed predictability violation (e.g., a terminated task may have output the correct values), but show weaknesses in the design of the input system that should be addressed by the developers.
6.4 Extensions to Spin

The Spin model checker, while it has many positive qualities, is very opaque. It does not expose many internal details at runtime. One of the aims for the verification tool was to build the models programmatically, and therefore eliminate the need for the end-user to be familiar with the actual Promela code that Spin receives as input. Spin is not designed with this use case in mind – it is difficult to monitor the progress of verification, and any output that can be provided is either only useful for debugging or is very closely tied to the Promela file supplied as input. As this was not satisfactory, some extensions to Spin needed to be made.

To understand the requirement for the extensions, it is necessary to be familiar with how Spin communicates its counterexamples. When Spin encounters a violation of the property that it is checking, it outputs a “trail file” – a machine-readable text file that contains information about the path through the state machine that was followed to arrive at the violation. This text file contains information that is meaningful to Spin, including state numbers and process numbers. By reading the trail file back into the generated verifier, a simulation of the fault condition can be reproduced.

Unfortunately, the trail file is nearly useless for the problem we were trying to solve. It does not contain any information about the state vector, and the data that it does contain is only meaningful to the dynamically-generated verifier. The state numbers used are generated in the process of taking the source Promela code and turning it into a state machine suitable for verification. They are not and cannot be specified by the user, and the mapping between Promela code and state number is not easily obtainable in advance.
The simple solution is to obtain the mapping from the state number to the location in the original Promela after the fact – this is possible and the generated verifier can produce this information on the demand. With some processing, it would be possible to automatically match the two sides. However, the second part of the problem becomes evident here: the Promela code is itself automatically generated and therefore meaningless to the user! It is possible to show the chain of Promela statements to the user, but this would be almost as unhelpful as showing the state numbers.

Essentially, there are two opaque transformations taking place: the user’s input to Promela code, and the Promela code to an optimized state machine. The only valuable information for the user is information expressed in terms of the original input: the language of tasks and deadlines.

To provide this information, one of two paths could be taken: either map the end product back onto the initial data or carry enough metadata through to the end product to make it meaningful. The first option is already half-done by Spin itself: as mentioned, the generated verifier can map states back onto Promela statements. However, the second half proved to be hard to achieve. The transformation of the users’ input into Promela was done through three separate chained tools (Java frontend, C preprocessor and Spin), and the mapping would have to reverse each of these processes.

It was therefore decided that the best course of action would be propagating metadata through to the output. This was done by adding annotations to the Promela code.
6.4. EXTENSIONS TO SPIN

6.4.1 Static Annotations

The first task was adding static state annotations. These annotations would be stored along with the states, and not the state vectors – they are referred to as “static” because they are determined at compile-time and cannot include state-vector data.

Promela supports attaching arbitrary labels to states by using C-inspired label: syntax. These labels can be used as the target of a goto statement, or, if they have certain prefixes, to assign special meaning to certain states. Promela code does not map one-to-one to generated states; therefore there are certain restrictions as to where the labels may be placed. In practice, this did not seem to be much of an issue though it did necessitate adding states with empty outgoing transitions (via skip statements) in certain places.

Spin does not make use of these labels as annotations, though it seemed like a good way to extend it to do what was needed. The labels would pass through the input-to-Promela transformation without a problem, and the Spin lexer and parser already were equipped to handle labels and attach them to the correct internal data structures. These values could, with simple modifications, be made available in the verifier and output as part of the trail file.

The only transformation that would have an effect on the label annotations would be the state machine optimization pass. Spin first generates a full state machine by producing a state from every Promela statement that can produce a state. Then, it merges states that can be merged without changing the correctness of the verification. For instance, consider two states that may form part of a larger state machine: S1 and S2. If S1 has only one outgoing transition, to state S2, and S2 has only one incoming transition, from S1, then S1 and S2 can safely be merged without changing
the behaviour of the system. Likewise, all states within an explicitly labelled fully-deterministic sequence (\texttt{dstep}) would be rolled up into one.

During the state merging process, some or all of the labels would be dropped. As Spin only uses the labels as a guide for creating transitions between states, they are no longer required after the initial creation of the state machine. The solution was to add a special prefix to annotation labels (following the example of other special label types, such as \texttt{end}), and combine such labels during state merging through string concatenation rather than dropping them.

The static labels now made it all the way through the process, and were included in the trail files produced. This meant that the trail file could be parsed and the labels extracted, allowing a meaningful mapping of trail files to input data. Since the labels were stored on the state machine, of which there is only a single copy, the memory consumption was negligible.

The main drawback of this approach was that the labels were static, and therefore could not use information from the state vector. This necessitated applying some contrived modifications to the Promela code so labels could be applied as needed to capture all the crucial data, however some data that would be nice to have was impossible to obtain. It was decided that it might be best to implement fully dynamic annotations in addition to the static ones.

### 6.4.2 Dynamic Annotations

Dynamic annotations were more complex than static annotations: they were not merely labels, they required processing by the verifier. Fortunately, Promela already contains the special \texttt{printf()} statement, which behaves much like its C standard
library namesake. The \texttt{printf()} statements are propagated through to the verifier, and behave almost identically during verification as they do during simulation. The handling of the \texttt{printf()} statement was used as a model for the new \texttt{annotation()} statement – right down to the syntax; \texttt{annotation()} used the \texttt{snprintf()} function from the C standard library, which behaves very similarly to \texttt{printf()}. Like static annotations, these would pass through the user-input-to-Promela stage unchanged.

Dynamic annotations required additions to the Spin parser, and were otherwise handled the same way as the regular \texttt{printf()} statements, meaning they passed through the optimization phase intact.

The code changes required in the generated verifier were more involved. The annotations had to be collected, and stored in the state vector. Multiple annotations may need to be stored together, as there could be multiple \texttt{annotation()} statements in the same deterministic step. The annotations had to be stored in the same way as the trail of visited states, and output in the correct locations in the produced trail file. This required modification of various parts of the verifier code.

An issue that was encountered and never fully resolved was that of static vector size: for efficiency, Spin uses a static state vector structure, the size of which is known at compile time. Therefore, the amount of memory that could be used by annotations had to also be fixed. This makes the implementation somewhat inefficient (in most cases, it was necessary to over-specify the annotation buffer), and had the potential for lost data (once the buffer fills up, any additional data had to be dropped). Fixing this issue is left as future work; changing this to be dynamic proved to be a more difficult task than first thought, as Spin uses a hash of the entire state vector to perform state-matching.
6.4.3 Spin Source Code

The source code for Spin is publicly available. It is written in ANSI C (also known as C89) \[10\], which is an older standard but is known to be widely supported and portable. Due to its age and portability requirements, Spin does not rely on many external libraries. To attain good performance, the code relies on various “tricks” and non-standard ways of performing certain tasks. Additionally, as the verifier code is generated dynamically, most of it is stored as strings within the Spin binary. This makes it very difficult to leverage the tools provided by modern IDEs (Integrated Development Environments) to help develop and debug the code. Modification therefore proved to be an unexpected challenge.

6.5 Promela Skeleton

The Promela code required to simulate the behaviour of the scheduler and operating system was written separately, as a re-usable “skeleton” or template with special placeholders that could be replaced with user-supplied data. This expedited the development process, as it meant that issues with the Promela and issues with the programmatic code generation could be debugged separately.

The template placeholders were one of two types: simple, which would just be replaced directly by the corresponding value (e.g., \$cores\$); and looping, which contained a block of code to include either for each task or for each core (e.g., \$perCore\$...code...\$). This permitted a complete separation of concerns between the user interface and the Promela skeleton code.

The Promela code is separated into three parts: one each representing the operating system, the scheduler, and the task template. A high-level pseudo-code description
of the skeleton can be seen in Figure 6.1.

6.5.1 Operating System

The file representing the operating system contains the scheduler loop. It initializes the tasks, and then calls the scheduler and the scheduled tasks in an infinite loop. Each iteration though the loop represents one scheduler tick (equivalent to a time slice) and is therefore treated as always taking the same amount of simulated time. This assumption is valid as we are not considering operating system overhead; if variable operating system overhead and context switch time came into play, it could no longer be assumed that a tick takes the same amount of wall-clock time.

The operating system code is also responsible for making non-deterministic choices regarding task failure. The fault model supplied as part of the input system definition contains the maximum number of tasks that can fail which is stored in the remaining failures counter. While this counter is greater than zero, at each iteration of the scheduler loop for each task that is not already in a failed state, a non-deterministic choice is made whether that task will experience failure. If the task is determined to have failed, the remaining failures counter is decremented by one. A failed task remains in the failed state until it has completed its unit of work.

6.5.2 Scheduler

The scheduler code is entirely deterministic. It runs once per scheduler tick, and is responsible for most of the logic underlying the implementation. It:

- Moves tasks into the ready state at their baseline;
- Copies a data snapshot into a task-specific buffer at the tasks’ baselines;
• Copies changed data values out of the task-specific buffers at the tasks’ deadlines;

• Handles task termination, as needed;

• Handles deadline overruns, as needed;

• Handles budget overruns, as needed;

• Calculates the priorities of all tasks that are ready to execute; and

• Contains assertions for predictability properties.

The zero-slack algorithm implementation is written in C rather than Promela as it is a language better suited to this type of problem. The implementation was included as an embedded C fragment. This did not affect the verification as the code that determines priorities is always fully deterministic and can be modelled as a black box. Spin does not need to know the details of the implementation.

6.5.3 Task Template

The task template file is very simple, and contains only the Promela code needed to define a task. Most of the task template is filled in programmatically based on user input.

Additionally, there is a configuration file that contains some global definitions, e.g., the exact numeric representation of certain sentinel values, and a bootstrap file that combines the individual files, allowing a single, combined output file to be produced.
6.6 Graphical Interface

The graphical user interface (GUI) to the tool allows the user to specify the input system, as well as incorporates code to merge the user input with the pre-written Promela templates to create a Promela file that can be fed to Spin. The ultimate goal for this component was to be easy to use by people unfamiliar with Promela.

The functionality that was deemed useful to have as part of the interface was:

- Allow a complete input system to be entered.
- Allow saving and loading previously-saved input systems. Ideally, they would be stored as regular files to allow distribution.
- Build and execute the verifier.
- Collect, parse and display the results produced by the verifier.
- Display the results to the user, including a graphical execution trace.

The programming language and toolkit chosen was Java with the Swing widget toolkit. Other contenders were C# and Windows Forms, as well as a PHP web application. The former was discarded as it was not cross-platform, and the latter did not meet the approval of industrial collaborators who required something fully self-contained. Java and Swing were then chosen for their portability and familiarity.

The design and development of the graphical interface, while it did take a non-trivial amount of time, does not bear extended discussion here as it was a straightforward programming task, aided by the UI-building toolkit available in the NetBeans 8 IDE. The final products are discussed in more detail in Sections 6.8 and 6.9 below.

https://netbeans.org/community/releases/80/
6.7 Visualization

The final artifact produced by the verification tool was a counter-example trace visualization, displaying the executions in which faults were found. This would allow a user to see exactly which case was problematic, and make adjustments accordingly. The visualization generation can be divided into two sub-problems: developing a structured, machine-readable output from the verifier; and parsing the output to generate an image.

6.7.1 Trace File Annotations

As discussed in Section 6.4, the trace files produced by Spin have been extended to include custom, dynamic annotations. The nature of the annotations is not limited by Spin (except in length: they must fit in the fixed-length buffer, as discussed). However, for the annotations to be useful, they must be machine-readable so they can be parsed to display the data in a user-friendly way.

To this end, a specification was developed for the output annotations. The specification required that the annotations may be human-readable, should the need arise. The ability to allow a human to read through and verify the annotations produced were correct trumped more technical concerns, such as memory usage. A binary (non-human readable) format would have been much more information-dense, and therefore would have consumed less memory. For example, a 16-bit numeric value can be stored in just 2 bytes, but requires up to 5 bytes when stored in a human-readable, ASCII-compatible encoding.

The annotations are simple text strings encoded using the ISO 8859-1 character encoding scheme, commonly known as Latin1. This character encoding uses a single
byte per character. A single-byte encoding was chosen for simplicity: multi-byte encodings (especially variable-length encodings such as UTF-8) would have added an additional level of complexity without providing any real benefit. ISO 8859-1 (or the nearly-identical Windows-1252) is the standard single-byte encoding used by the majority of English-speaking North America.

Each annotation represented an event that may be of interest to the user. The annotation string was comprised of a single-character event code, followed by zero or more parameters separated by spaces (ISO 8859-1 0x20). The event codes were all uppercase letters. Initially, a four-character event code was used, but this did not provide any additional information and introduced a cost of three extra bytes per annotation. Parameters could not themselves contain spaces (there was no escaping syntax). This did not prove to be a problem.

The full set of annotations is shown in Table 6.1. This table is referenced extensively in Section 6.7.2.

6.7.2 Image Generation

The second step in generating the execution trace visualization is parsing the trail file and producing an image. This process takes four passes through the data:

Pass 1, String Parsing The trail file is read in and parsed line-by-line. Spin-specific data (three integers, separated by the colon character, :) is stripped. The rest of the line is the annotation data: as some lines can contain multiple annotations, each line is split on the annotation separator string (double underscore, _). Annotations are inserted into a variable-length array in order: reading the trail file top to bottom and left to right will always produce the correct ordering of the annotations. Each
### Table 6.1: Full set of visualization annotations that are supported.

A variable number of parameters are supported; even though there are four columns, not all events have four parameters. All parameters listed are mandatory.

Note 1: The initialization annotation (code C) occurs once, and is output before any other code executes. This annotation contains as its parameters the numeric values of internal constants. In this way, these values can be identified when they occur.

Note 2: The first parameter for annotation code S is defined as the tick number. In full verification mode, this will always be 0. In fixed-size trail generation mode, this will be a monotonically increasing integer value, starting at 0, representing the number of scheduler ticks that have elapsed.

<table>
<thead>
<tr>
<th>Event</th>
<th>Code</th>
<th>Param. 1</th>
<th>Param. 2</th>
<th>Param. 3</th>
<th>Param. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note or comment</td>
<td>A</td>
<td>Any</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialization_note 1</td>
<td>C</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task running, no fault or overrun</td>
<td>E</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task running, fault, no overrun</td>
<td>F</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data copied in to task buffer</td>
<td>I</td>
<td>Task ID</td>
<td>Data ID</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Task killed</td>
<td>K</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task deadline/baseline</td>
<td>L</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task initialization</td>
<td>N</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data copied out from task buffer</td>
<td>O</td>
<td>Task ID</td>
<td>Data ID</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Task priority re-calculated</td>
<td>P</td>
<td>Task ID</td>
<td>New priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task read value from task buffer</td>
<td>R</td>
<td>Task ID</td>
<td>Data ID</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Scheduler executed_note 2</td>
<td>S</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task status polled (task tick)</td>
<td>T</td>
<td>Task ID</td>
<td>Current conf.</td>
<td>Next conf.</td>
<td>Status</td>
</tr>
<tr>
<td>Task running, no fault, overrun</td>
<td>V</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task wrote value to task buffer</td>
<td>W</td>
<td>Task ID</td>
<td>Data ID</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Task running, fault and overrun</td>
<td>X</td>
<td>Task ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File version number</td>
<td>Z</td>
<td>Version</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
annotation is stored as a data structure composed of an event code and zero to eight parameters.

**Pass 2, Tick Separation**  The ordered list of event annotations is separated by tick number. All events between two successive $S$ events are grouped together into one tick. The $S$ events themselves are dropped, serving only as delimiters. Their one and only parameter (tick number) is disregarded; ticks are numbered sequentially starting at zero.

Additionally, if any $Z$ events are encountered, the file format version is checked. If the version of the tool in use does not support the version of the file format specified, the process aborts. The $Z$ events are a way of ensuring that the tool will produce a sensible error message if the trail file format changes in the future and the user attempts to read in a new-format file. The $Z$ events are dropped.

**Pass 3, Data Flow Calculation**  The data flow through the system is expressed as a set of “messages”. A message is a six-tuple of values: the data item ID, the data value, the task ID of the writer, the tick number at time of writing, the task ID of the reader and the tick number of the read. For each value read, there is therefore one message that links the read to the write that produced that value. Many read events can be linked to one write event, and a task can read and write multiple values during the same scheduler tick. Messages are subdivided into three types: copy-in to read (event type $I$ to $R$), write to copy-out (event type $W$ to $O$) and copy-out to copy-in (event type $I$ to $O$). The third pass steps through the list of event annotations and pairs up the relevant events into messages.
Pass 4a, Image Generation from Events This pass steps through the event annotation list one last time, and makes the appropriate library calls to build an image. The different event types are handled as indicated:

- **Event types A, C, I, O, R, T, W** These events are ignored.

- **Event types E, F, L, N, P, V, X** These events produce a graphical element: either some text, a line or a coloured block.

- **Event type K** The occurrence of this event is logged, and for each subsequent tick a graphical element will be produced indicating the task is no longer running.

- **Event types S, Z** These event types were dropped in Phase 2; they are included here for completeness only.

Pass 4b, Image Generation from Messages Finally, the messages calculated in Pass 3 are drawn as lines on the image indicating the flow of data. The image is now complete.

The image that is produced is a standard PNG-format image file, created using the Java standard library image processing routines. It can be read in by any application that can read PNG files. This approach proved to be much simpler and more performant than generating a “live” graphic using interactive elements. This latter approach is still useful as an interactive visualization can allow the user to inspect and query different events, e.g., view all messages for a given data item. This is left as future work.

A sample trail file and the corresponding trace visualization image are shown in Figure 6.3.
6.8 Implementation 1: Simple Tasks

The first implementation of this tool focused on simple tasks. A simple task was one that had only a single configuration, and all of its inputs would always affect all of its outputs. This allowed the definition of a task to be simplified to a vector of values. These values are:

- Task name (for display purposes)
- Window size
- Execution budget
- Initial offset
- Data items read (list)
- Data items written (list)
- Criticality
- Writes are externally-visible? (Is output?)
- Reads from external locations? (Is input?)
- Overrun – if a fault occurs, it is assumed that the task spends this amount of time in excess of its execution budget to complete a unit of work

A unit of work for the task was assumed to always take an amount of time equal to its execution budget.

Tasks are assigned identification numbers (IDs) automatically. Data items can be specified using any unique string of characters. The same string appearing within
6.8. IMPLEMENTATION 1: SIMPLE TASKS

two different data item lists is assumed to refer to the same data item. Internally, the data items are also assigned numeric IDs for conversion to Promela. The textual values entered are kept for display purposes only.

It may seem slightly illogical to flag tasks rather than data items as being externally-visible. This is true; however it was done this way to make the translation to Promela simpler. The end result is no less correct as long as the user entering the task set does not make a mistake – as this was a proof-of-concept project rather than an actual production-grade tool, this was deemed an acceptable risk.

6.8.1 User Interface

The user interface was divided into two views: The data-entry view and the results view.

Data-Entry View

The data-entry view consists of a set of form fields where the input system definition is entered. The input system definition consists of:

- Number of symmetric processor cores
- Scheduling policy in use (EDF, DMS, or ZS)
- Action taken by the simulated operating system on a deadline violation (terminate, terminate if non-critical, terminate if critical, no action)
- Action taken by the simulated operating system on an execution budget violation (same as above)
- Maximum number of execution budget overruns
• Should faults in critical tasks be simulated as well as faults in non-critical tasks?

• The set of task vectors

Additionally, a pair of parameters (vector size and maximum search depth) is exposed to the user. These are passed to Spin directly; modification of these parameters only needs to be done if either Spin or the generated verifier reports that the default values are insufficient. They represent the sizes of various statically-allocated data structures used by the verifier.

A screenshot of the data-entry view is shown in Figure 6.4.

Results View

The results view is displayed after verification is complete. It consists of a simple two-panel window, with the console output from Spin displayed on the left and the counterexample trail file visualization on the right. If there is no counterexample (i.e., verification has determined that the input system is always predictable), then nothing is displayed in the right panel.

A screenshot of the results view is shown in Figure 6.5.

6.8.2 Loading and Saving

Input systems can be loaded from and saved to disk in XML format. XML was chosen as it was a relatively simple, human-readable format that would be familiar to a sufficiently technically knowledgeable person. It also has very good, “out-of-the-box” support in the Java standard library. Alternatives that were considered but discarded were JSON (library support not as extensive), a custom format (would not be immediately familiar to users), or serialized binary data (not editable by hand).
The structure of the XML files mapped nearly one-to-one with the form controls presented to the user. Each XML file stores one and only one input system definition.

The generated XML could be validated by using a supplied document type definition (DTD) file. All generated XML files were strictly compliant with this DTD, though the input parser was configured to not validate documents and attempt to parse even documents with structural errors. Syntax errors were more difficult to overcome, and therefore a syntactically-invalid input file would produce an error.

Saving input systems as flat files allowed the input system definitions to be distributed separately from the application package.

### 6.8.3 Flow of Execution

The process by which a given input system is verified for predictability is as follows:

1. Read value of all form fields into an internal data structure.

2. Read Promela skeleton code from the main file. Scan this file for C-preprocessor \#include statements. If any are found, replace them with the contents of the corresponding file. This is only done one level deep (i.e., the included files are not themselves scanned for \#include statements).

3. Replace placeholders in the Promela code with their corresponding values from the form-field data structure.

4. Write out the final Promela code to a single file with a unique file name.

5. Pass the path of this file, along with user-supplied parameters, to Spin running in generate-and-verify (i.e., -run) mode. Capture the console output of Spin,

---

"Be liberal in what you accept, and conservative in what you send" \[^{1}1\] via \[^{22}2\]
6.9. IMPLEMENTATION 2: EXTENDED TASKS

and wait for the application to terminate.

6. Once Spin terminates, scan the relevant directory for any trail files. These will have names that can be determined from the name of the Promela file generated earlier.

7. Generate a visualization for the first trail file found.

8. Display the results window, including the generated visualization image (if any) and console output from Spin.

6.9 Implementation 2: Extended Tasks

The second implementation of the tool focused on more complex tasks. It was built following feedback received regarding the first implementation. Complex tasks were tasks that could have arbitrary behaviour, i.e., tasks that behaved more like their production counterparts. These tasks could alter their behaviour and outputs based on inputs, as well as requesting different configurations from the operating system.

The task definition was changed to consist mainly of five blocks of Promela code; the full definition is:

- Task name (for display purposes)

- Initialization code – executes once, at system startup. Meant to initialize the task descriptor structure.

- Per-tick code – executes once per tick, regardless of task status. Meant to simulate external events.
6.9. IMPLEMENTATION 2: EXTENDED TASKS

- Non-fault unit of work code – executes once per tick that the task is given processor time and is not marked as experiencing a fault.

- Fault unit of work code – executes once per tick that the task is given processor time and is marked as experiencing a fault.

- Termination code – executes when the task is terminated by the operating system.

The per-tick code was required for simulation purposes; it does not have a real-life counterpart. It is used to simulate external events, such as hardware interrupts.

Tasks are once again assigned automatic identification numbers.

As the behaviour of the tasks is no longer known ahead of time, the user is provided with a set of simple commands that must be placed at the appropriate places in the code. These commands are used to mark locations where events that are significant for the verification occur, e.g., where an edge task has its deadline moved. All five blocks come pre-populated with Promela code that will achieve equivalent behaviour to that of the first implementation, and provides users with a guide of how to structure the task behaviour.

6.9.1 User Interface

The user interface was once again divided into two views: The data-entry view and the results view.

Data-Entry View

The data-entry view consists of a set of form fields where the input system definition is entered. The input system definition consists of:
6.9. IMPLEMENTATION 2: EXTENDED TASKS

- Number of symmetric processor cores
- Scheduling policy in use (EDF, dDMS, or ZS)
- Action taken by the simulated operating system on a deadline violation (terminate, terminate if non-critical, terminate if critical, no action)
- Action taken by the simulated operating system on an execution budget violation (same as above)
- Maximum number of execution budget overruns
- Should faults in critical tasks be simulated as well as faults in non-critical tasks?
- The set of tasks

Entering task data was separated into a separate window, due to the expected length of the Promela blocks.

Additionally, several parameters (vector size, annotation size and maximum search depth) are exposed to the user. These are passed to Spin directly; modification of these parameters only needs to be done if either Spin or the generated verifier reports that the default values are insufficient. Users may also choose for all possible counterexample trail files to be generated, and not have Spin terminate after the first error is encountered.

A screenshot of the data-entry view is shown in Figure 6.6. A view of the task-entry window is shown in Figure 6.7.
Results View

The results view is displayed after verification is complete. The simple view from the first implementation was updated to allow the user to choose one of the counterexample trail files to display. The graphic was also separated into toggleable layers, allowing the user to show and hide certain parts of the visualization.

A screenshot of the results view is shown in Figure 6.8.

6.9.2 Loading, Saving and Flow of Execution

The second implementation was based on the first, and as such some parts remain unchanged. For input system definition file information, see Section 6.8.2; for control flow, see Section 6.8.3.

6.10 Conclusion

This chapter covered the design and development of the verification tool, starting from the choice of technology, continuing with a discussion of the hurdles faced during design and development, and ending with descriptions of the two implementations. The next chapter will cover testing and performance.
Figure 6.1: A pseudo-code version of the Promela skeleton.

```plaintext
for (task in tasks) {
    task.init();
}

global.init();

// Begin scheduling loop
while (true) {

    // Pre-tick
    if (!global.deadline_overrun) {
        for (task in tasks) {
            if (task.elapsed >= task.window_size) {
                if (task.terminated) {
                    task.clear();
                    task.status = terminated;
                } else {
                    task.elapsed = task.elapsed - task.window_size;
                    task.executed = 0;
                    task.faulty = false;
                    task.current_configuration = task.next_configuration;
                    task.status = sleeping;
                    task.data = global.get_data();
                }
            }
        }
    }

    global.tick();

    for (task in tasks) {
        task.tick();
    }

    // Tick
    task_ordered_queue.clear();
    for (task in tasks) {
        if(task.status == ready) {
            task_order_queue.enqueue(task, scheduler.calculate_priority(task));
        }
    }
}
```
for (0..global.core_count) {
    if (task_order_queue.is_empty()) {
        break;
    }
}

task = task_ordered_queue.dequeue();
task.faulty = global.random_fault();

if (task.faulty) {
    task.run();
} else {
    task.fault();
}

task.executed++;
}

// Post-tick
global.deadline_overrun = false;

for (task in tasks) {
    if (task.status != terminated) {
        task.elapsed++;
    }

    if (task.status != terminated & task.status != sleeping) {
        if (task.elapsed < task.window_size) {
            task.status = ready;
        } elseif (task.elapsed < (task.window_size * 5)) {
            task.status = ready;
            global.deadline_overrun = true;
        } else {
            task.status = sleeping;
            task.kill();
        }
    } elseif (task.status == sleeping & !write_queue.contains(task)) {
        write_queue.enqueue(task);
    }
}

if (!global.deadline_overrun & !write_queue.is_empty()) {
    global.merge_data(write_queue.dequeue().data);
}
6.10. CONCLUSION

Figure 6.2: A partial counterexample trail file. Lines between tick 10 and tick 41 have been omitted for brevity. The entire run is 50 ticks long.

Reading order: down the columns, starting at the leftmost column.
6.10. CONCLUSION

Figure 6.3: A sample counterexample trail file visualization. The numbers running vertically along the far left side are the tick numbers, the task names run along the top. The names are pulled from the currently-loaded configuration file, to make the visualization more useful. Solid blocks represent tasks executing, and the number superimposed on the left side of the block is the task priority value that was calculated. Values that are written and copied out are indicated by a small “name=value” on the right hand side of the block. Finally, data flow is visualized using translucent solid lines. This diagram is best viewed in colour.
Figure 6.4: The data-entry view of the first implementation.

Figure 6.5: The results view of the first implementation.
Figure 6.6: The data-entry view of the second implementation.
Figure 6.7: The task-entry view of the second implementation.

Figure 6.8: The results view of the second implementation.
Chapter 7

Testing and Performance

In addition to functionality tests used for debugging, the tool and model were tested using a small but realistic example from the automotive domain. This task set definition was provided by industrial collaborators as being representative – in terms of task count and data flow – of a real-world task set used in production systems.

Various system configurations were tested – different numbers of cores and allowable faults, along with different scheduling algorithms and enforcement modes. The size of the state space and therefore both the execution time and memory requirements increase as the number of permitted faults increases. This is understandable, as each fault represents a non-deterministic choice that needs to be made once per scheduling tick per task, causing a branch in the search tree. The search tree becomes very broad with many faults, but the depth remains reasonable. No other parameter has this sort of effect, as every other behaviour is entirely deterministic.

For the purposes of this discussion, a more formal test was conducted. This is discussed below. However, performance was not a primary consideration during development: it was only necessary that the verifier complete in a reasonable amount of time and use a reasonable amount of memory. Optimization of the model, such as
reducing the number of states and the size of the state vector, is left as future work.

7.1 Active Safety Demonstration Task Set

7.1.1 Background

The task set used for testing represents a simple integration of various active safety features with standard manual driving controls. “Active safety” is a term used in the industry to refer to technologies being developed and included in new vehicles that use computer technology in attempting to prevent accidents, either by alerting the driver or taking direct control of the vehicle. It is a field in which extensive research and development is taking place.

Adaptive Cruise Control One of the active-safety features included in newer vehicles is adaptive cruise control. Classic cruise control allows the driver to set a speed, and for the vehicle to maintain that speed automatically, i.e., without driver input. It is commonly used in freeway driving, to reduce driver fatigue and fuel consumption. Adaptive cruise control systems extend this functionality by monitoring the speed of traffic surrounding the vehicle and adjusting the pre-set speed automatically [23].

Lane Keeping Another active-safety feature being developed is lane-keeping, an extension to lane departure warning systems. Lane departure warning systems use computer vision technologies to monitor the lane markers painted on the road and alert the driver when they are veering out of the lane, as in [17]. Lane-keeping technologies take this a step further by taking control of the steering system to keep the vehicle centred in the lane.
7.2. TESTING METHODOLOGY

7.1.2 Task Set

The parameters of the various tasks that make up the task set can be seen in Table 7.1. Explanations of the data values read and written are available in Table 7.2. A data flow diagram can be seen in Figure 7.1.

7.2 Testing Methodology

To test the performance of the verifier, the task set described in Section 7.1 was read into the GUI and system parameters were set as indicated in the results tables. Each test was performed five times and the arithmetic mean of the five sets of completion times was reported. No statistical analysis was performed as this was an informal benchmark.

Testing was performed on a desktop computer running Microsoft Windows 7 Professional (64-bit). The CPU used was a six-core Intel Core i7 3930K at 3.2GHz with 28GB of RAM. Attempts were made to limit the resource utilization of other tasks on the system during the tests.

All optimizations performed by Spin when generating the verifier (e.g., state merging) were enabled, the default state. Parallel execution of the verifier was disabled due to observed issues on Microsoft Windows and Cygwin.

Critical tasks were permitted to fail. Each scheduling algorithm produced broadly similar results, so a full test was only done using EDF.

7.3 Results

The results of the test runs are shown in Table 7.3 for the first implementation (simple tasks) and Table 7.4 for the second implementation (extended tasks). As expected, the
Table 7.1: The active safety demonstration task set.  
Initial offset is 0.

<table>
<thead>
<tr>
<th>Task</th>
<th>Criticality</th>
<th>Window</th>
<th>Budget</th>
<th>Overrun</th>
<th>Values Read</th>
<th>Values Written</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptiveCruise</td>
<td>Hi</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>APP BPP CO CSS</td>
<td>ABT APT</td>
</tr>
<tr>
<td>AutoSteering</td>
<td>Hi</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>ASO LMP SWA</td>
<td>AST</td>
</tr>
<tr>
<td>InputProcessing</td>
<td>Hi</td>
<td>40</td>
<td>6</td>
<td>4</td>
<td>(Environment)</td>
<td></td>
</tr>
<tr>
<td>ManualBraking</td>
<td>Lo</td>
<td>40</td>
<td>6</td>
<td>4</td>
<td>BPP</td>
<td>MBT</td>
</tr>
<tr>
<td>ManualPropulsion</td>
<td>Lo</td>
<td>40</td>
<td>6</td>
<td>4</td>
<td>APP</td>
<td>MPT</td>
</tr>
<tr>
<td>ManualSteering</td>
<td>Lo</td>
<td>40</td>
<td>6</td>
<td>4</td>
<td>SWA</td>
<td>MST</td>
</tr>
<tr>
<td>OutputArbitration</td>
<td>Hi</td>
<td>40</td>
<td>6</td>
<td>4</td>
<td>APP ABT APT ASO AST BPP CO MBT MPT MST SWT</td>
<td></td>
</tr>
<tr>
<td>OutputProcessing</td>
<td>Hi</td>
<td>40</td>
<td>6</td>
<td>4</td>
<td>BT COI PT SOI ST</td>
<td>(Environment)</td>
</tr>
</tbody>
</table>
Table 7.2: Description of the input and output values used in Table 7.1.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABT</td>
<td>Automatic braking torque</td>
</tr>
<tr>
<td>APP</td>
<td>Accelerator pedal pressure</td>
</tr>
<tr>
<td>APT</td>
<td>Automatic propulsion torque</td>
</tr>
<tr>
<td>ASO</td>
<td>Automatic steering enabled</td>
</tr>
<tr>
<td>AST</td>
<td>Automatic steering torque</td>
</tr>
<tr>
<td>BPP</td>
<td>Brake pedal pressure</td>
</tr>
<tr>
<td>BT</td>
<td>Braking torque</td>
</tr>
<tr>
<td>CO</td>
<td>Cruise control enabled</td>
</tr>
<tr>
<td>COI</td>
<td>Cruise control indicator light</td>
</tr>
<tr>
<td>CSS</td>
<td>Cruise control set speed</td>
</tr>
<tr>
<td>LMP</td>
<td>Lane marking sensor input</td>
</tr>
<tr>
<td>MBT</td>
<td>Manual braking torque</td>
</tr>
<tr>
<td>MPT</td>
<td>Manual propulsion torque</td>
</tr>
<tr>
<td>MST</td>
<td>Manual steering torque</td>
</tr>
<tr>
<td>PT</td>
<td>Propulsion torque</td>
</tr>
<tr>
<td>SOI</td>
<td>Automatic steering indicator light</td>
</tr>
<tr>
<td>ST</td>
<td>Steering torque</td>
</tr>
<tr>
<td>SWA</td>
<td>Steering wheel angle</td>
</tr>
<tr>
<td>SWT</td>
<td>Steering wheel torque</td>
</tr>
<tr>
<td>TD</td>
<td>Distance to vehicle ahead</td>
</tr>
<tr>
<td>VS</td>
<td>Vehicle speed</td>
</tr>
</tbody>
</table>

Environment: The electrical interface between the ECU and other hardware.

The performance of the first implementation was reasonable, with most runs completing in under a second. The second implementation took more time, both due to its more complex generated models and less-optimized code.

The 4-core configuration was schedulable for all the fault cases that were tried, so it was a good for comparison. The growth rate seems to be polynomial, with a slight aberration between no faults and one fault. This can be seen in Figure 7.2.
Figure 7.1: Data flow diagram of the active safety demonstration task set. See Tables 7.1 and 7.2 for details on the tasks and data values.
Table 7.3: First implementation performance.
The first two columns represent the configuration (number of cores simulated and number of allowed faults, respectively). The third column is the result of the verification, as a binary yes or no. If the “no” is followed by an asterisk (*), the configuration was not predictable because it was not schedulable, even with no faults. The fourth column is the number of states explored. Finally, the fifth column is the mean time to completion as reported by Spin (excluding visualization generation).

<table>
<thead>
<tr>
<th>Cores</th>
<th>Faults</th>
<th>Predictable?</th>
<th>States</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>No*</td>
<td>2178</td>
<td>0.013</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Yes</td>
<td>2231</td>
<td>0.003</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Yes</td>
<td>2231</td>
<td>0.002</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>No*</td>
<td>2180</td>
<td>0.013</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Yes</td>
<td>19486</td>
<td>0.022</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Yes</td>
<td>20844</td>
<td>0.023</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>No*</td>
<td>2182</td>
<td>0.015</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td>87971</td>
<td>0.096</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Yes</td>
<td>95394</td>
<td>0.106</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>No*</td>
<td>2186</td>
<td>0.012</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>No</td>
<td>44369</td>
<td>0.059</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Yes</td>
<td>632906</td>
<td>0.707</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>No*</td>
<td>2198</td>
<td>0.013</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>No</td>
<td>44137</td>
<td>0.060</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>Yes</td>
<td>3059085</td>
<td>3.54</td>
</tr>
</tbody>
</table>

7.4 Conclusion

This chapter covered the testing and performance of the two implementations described in the previous chapter. The next chapter contains a look back on lessons learned as well as a look forward at potential future extensions to this work.
### Table 7.4: Second implementation performance. Columns are the same as in Table 7.3.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Faults</th>
<th>Predictable?</th>
<th>States</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>No*</td>
<td>1039</td>
<td>0.024</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Yes</td>
<td>7078</td>
<td>0.201</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Yes</td>
<td>7078</td>
<td>0.199</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>No*</td>
<td>1039</td>
<td>0.023</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Yes</td>
<td>55688</td>
<td>1.57</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Yes</td>
<td>59656</td>
<td>1.61</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>No*</td>
<td>1039</td>
<td>0.021</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td>249550</td>
<td>7.24</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Yes</td>
<td>272027</td>
<td>7.71</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>No*</td>
<td>1039</td>
<td>0.020</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>No</td>
<td>249812</td>
<td>7.28</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Yes</td>
<td>1793057</td>
<td>56.4</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>No*</td>
<td>1039</td>
<td>0.019</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>No</td>
<td>1039</td>
<td>0.024</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>Yes</td>
<td>8471546</td>
<td>203</td>
</tr>
</tbody>
</table>

**Figure 7.2**: The four-core configuration state counts.
Chapter 8

Reflections and Future Work

The initial problem motivating this research was the requirement for a formal verification of a new approach to data flow for embedded systems. The main task was to create a proof-of-concept tool for performing such a verification. This was achieved, with two functional prototype tools that were tested and demonstrated to industrial partners. However, there is still much more that can be done – extensions made to existing work, mistakes corrected – that is not achievable within the limited time frame allotted for this research. This is what will be discussed in this chapter.

8.1 Custom Code, Existing Tools

The path chosen for developing the software was to use an existing, proven verification platform (Spin) to perform the core verification duties and integrate that with custom code to provide the ancillary functionality (e.g., GUI). This was seen as a sensible approach with perhaps a slight nod to the UNIX philosophy of system design [20], allowing the focus to be on the single problem at hand and not on solving already-solved problems. It was hoped that this would speed up development as well.

In retrospect, this was perhaps a poor decision.
First, the final solution uses very few of Spin’s many features, and extensions had to be made to Spin to add features that were required but not present. It does not seem that Spin was as well-suited to this task as initially thought.

Second, the integration process ended up being needlessly complex. Spin brought with it Promela, and required a wholly-unnecessary conversion step: the original task code is in C, which had to be converted manually to Promela, which was then automatically converted back to C. It would have been simpler, both for development and for the end-users who do not know Promela, to stay with C all the way through and apply some restrictions as to what features of the language can be used.

Third, adding Spin as a dependency introduced a second code base that had to be checked and fixed to ensure portability between OS X, Windows and Linux systems. Changes were necessary despite the claims of the Spin maintainers that Spin is fully portable as-is.

Fourth, many features that would have been simple to implement had the code been fully custom (e.g., dynamic visualization) were very difficult or impossible since control was ceded to Spin.

Finally, on a personal note, relying on Spin meant that many of the more interesting programming tasks were already handled while increasing the number of uninteresting programming tasks. The programming tasks that were left were much more mundane and tedious: form-based interface development, debugging portability issues, writing glue code.

It would probably have been best to develop a single tool that contains all the components necessary. The verification process might have been slower as it would no longer be done in compiled code, but verification time was never a major issue in
the first place – if verification time were doubled or even quadrupled, it would still be sufficient for a proof-of-concept tool. A fully-custom implementation would have allowed for more fine-grained control over the interaction between the verification and the visualization, as well as tasks written in C. Portability issues would have been limited to one familiar codebase, and bugs would be easier to track down. A more stable and fully-featured tool could have been produced if this path was followed.

8.2 Spin

Spin was chosen as the model checker due to its performance, familiarity and proven track record. In the end, this proved to be a good choice compared to the other model checkers initially considered, issues mentioned in the previous section notwithstanding. Interoperability with Spin was easy to achieve, and verification was quick. While there was work necessary to run Spin on some platforms, it was nowhere near as arduous a task as getting some larger pieces of academic software to run can be. While not ideal, Promela proved to be sufficiently expressive for the relevant use cases.

Extending Spin proved more difficult than it perhaps needed to be. The Spin source code was very convoluted, with years of changes obscuring any architecture design that may have been present initially. Decisions seemed to have been made based on a desire for quick turnaround or minimizing resource consumption rather than concern for best practices or future maintainability. This may be due to the fact that Spin was initially developed at a time where hardware limitations were much more severe than today.
8.3 Future Work

This project leaves open several possible avenues for extension and additional research. These include formalizing the concepts, re-creating the existing proof-of-concept implementations and creating a production-grade tool that can be used in production.

The predictable data-flow approach described in this work is based on various pieces of published work, each of which themselves have formalizations and some of which have proofs. However, the entire approach as put together has not been formalized, including concepts that are unique to this work such as the forcibly delayed deadlines. It would be beneficial for those who wish to make use of this concept – and much more convincing to others – if this were to be formalized.

The existing implementation of the verification tool has several limitations, as already discussed. A potential future continuation of this work would be to re-implement the proof-of-concept tools as a single, custom-written application or to better integrate Spin or another model checker. The end product may be useful to a broader audience and allow for more opportunities for extension.

Finally, the current implementation was intended to be simply a high-level demonstration of how model-checking could be used to provide schedulability and predictability guarantees for a real-time system that requires predictable data flow. It does not approach the complexity required to perform this type of analysis on actual task sets used industry. More complex models must be made to more accurately reflect the different parts of a real-time operating system (e.g., shared resources). Integration with tools used by engineers is also a must. This may, however, be a task for commercial enterprise and not for academic work.
8.4 Conclusion

This chapter contained a reflection on the work that was done as well as potential ways to extend it in the future. The next and final chapter wraps up this document with a summary and conclusion.
Chapter 9

Conclusion

The vehicle of today is increasingly computerized, with manufacturers responding to the demands of an increasingly technology-oriented generation by adding more and more features and automation. At the same time, the traditional constraints of automotive design – weight, power consumption and cost – have not gone away. One way to reconcile these conflicting requirements is by making existing hardware do more: better utilize processor resources by using design-time rather than run-time guarantees of fault isolation.

This work proposed an operating system design that utilizes concepts from PharOS and Giotto to achieve predictable data flow even with variations in task scheduling. This approach restricts what data tasks can see, and arranges for task communication to happen at fixed points in time. At the same time, tasks are allowed to occasionally be delayed and overrun their deadlines as long as slack available in the system can be used to make up the time. This is all achieved without requiring involvement of the task developers themselves.

A method by which to verify the predictability of a given task set running on the above-described system was developed using the Promela modelling language and
Spin model checker. A proof-of-concept cross-platform tool was built and tested with a small, representative example from industry. As this was only a proof-of-concept tool, there exists the possibility to extend this work to a full production-grade tool used for verification of actual tasks used in vehicles.
Bibliography


