School of Computer Science and Statistics

SPIN/Promela-Based Test Generation Framework for RTEMS Barrier Manager

Jerzy Jaśkuć

April 14, 2022

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MCS (Integrated Computer Science)

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Signed: ____________________ Date: ____________________

Abstract

Testing is one of the essential parts of software development. It provides guarantees in terms of system behaviour in specific scenarios, which are covered within the tests. That is why tests are a basis of a reliable system. Nevertheless, what if the system becomes increasingly large while the guarantees of its behaviour are the cornerstone of its existence? Such is the case for RTEMS, an open-source Real-Time Operating System (RTOS). RTEMS is commonly used in the embedded system in industries such as medicine, air and space. Many use cases in these industries require the system to have deterministic, predictable behaviour and guarantee to perform some tasks within time limit constraints. That makes writing tests a formidable task and makes it harder to verify whenever developed test suites are comprehensive enough.

This dissertation presents the framework that utilizes Promela/SPIN formal verification tools to model RTEMS behaviour and generate possible execution scenarios based on this model. These scenarios are then used together with templates to automatically generate the test suite for the modelled part of the system. Using SPIN/Promela allows the system to be formally verified through modelling while it also speeds up the development process of creating the test suites. Since the process is largely automated, it also decreases the chance of accidental human error and allows for better system scalability.

This work continues on previous research done as a part of the ESA RTEMS Qualification Project and is based on the framework used to model the Events Manager of RTEMS. The framework was overhauled, extended and applied to the RTEMS Barrier Manager to generate the test suite. The project resulted in a test suite with comprehensive code coverage and an improved framework that is cleaner and more extendable.

Acknowledgements

I would like to thank my supervisor Prof. Andrew Butterfield and RTEMS Community for all the support and guidance throughout the project. Your knowledge and help was invaluable!

Secondly, I would like to thank my family and friends for all the years of continuous support. Thank you for being there with me!

Finally, I would like to thank myself for being the unwavering sole source of motivation and determination to go through and keep on achieving throughout these five college years. Keep on going!

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Project Overview . . . 2

1.3 Contributions . . . 2

1.4 Dissertation Structure . . . 3

2 Background **4**

2.1 Real-Time Executive for Multiprocessor Systems . . . 4

2.2 SPIN and Promela . . . 7

2.3 State-of-the-Art Work . . . 8

2.4 Alternatives Considered . . . 9

3 Design **11**

3.1 RTEMS Configuration . . . 11

3.2 Test Generation . . . 14

3.2.1 SPIN/Promela Modelling . . . 14

3.2.2 Model Refinement . . . 15

3.2.3 Supporting Files . . . 16

3.3 RTEMS Barrier Manager . . . 17

4 Implementation **19**

4.1 Model . . . 19

4.1.1 Modelled Directives . . . 22

4.1.2 Scenarios . . . 28

4.1.3 Generating log files . . . 32

4.2 Refinement & Test Generation . . . 33

4.3 RTEMS Tests . . . 35

4.4 RTEMS Configuration & Test Deployment . . . 36

5 Evaluation **41**

5.1 Model Validation 41
5.2 Test Code Coverage 42
5.3 Comparison with existing test suites 44
5.4 Extensibility 45

6 Conclusion **49**
6.1 Future Work 50

A1 Appendix **55**
A1.1 Refinement Dictionary File 55
A1.2 Spin2Test Language Definition 58
A1.3 Generated Test Segment 60
A1.4 Complete Project Code 60

List of Figures

2.1 Organization of RTEMS Classic API Managers. (1) 5

4.1 Barrier Manager directive mappings inside `model-barrier-mgr-rfn.yml` dictionary file. 39
4.2 Overview of the SPIN/Promela test suite framework and files involved. . . . 40

5.1 Contents of `sparc-gr712rc-smp-user-qual.yml`, needed to execute tests on SIS simulation tool 48

A1.1 Full contents of `model-barrier-mgr-rfn.yml` dictionary file. 58
A1.2 Example test segment for barrier manager generated with refinement dictionary file. 61

List of Tables

4.1 Processes and their corresponding Promela IDs. 21
4.2 Directive execution in default Manual Barrier Operations scenario. 31
4.3 Directive execution in default Automatic Barrier Operations scenario. 31
4.4 Directive execution in Automatic Barrier with Timeout scenario. 32

5.1 Test coverage for GR712RC SMP using Promela/SPIN framework generated barrier manager tests. 43
5.2 Test coverage for GR712RC UNI using Promela/SPIN framework generated barrier manager tests. 43
5.3 Test coverage for GR712RC SMP when using existing barrier manager tests. 44
5.4 Test coverage for GR712RC UNI when using existing barrier manager tests. . 45

1 Introduction

1.1 Motivation

Testing is becoming an increasingly important topic throughout recent years. With the increase of computational power and the ability to instantly compile and execute written code, it is no longer a problem to produce a large amount of working code. However, that code is ultimately broken down into a binary sequence of 1's and 0's and executed by the machine. There's no guarantee that the execution result is what the programmer intended it to be when writing the program; the machine does not care. That is why there are usually tests in place. Tests typically are scenarios that range from standard execution patterns to edge cases that stress test or check system behaviour in unusual situations. The larger the system, the more tests are needed, and the more challenging it is to test all the system components' interactions. Manually writing tests is then becoming increasingly tedious and time-consuming. It becomes especially evident in cases where the tested application or system is *mission critical*. A failure of *mission critical* application or system can potentially lead to the complete, irreversible loss of the whole machine that it was controlling, or in the worst case, even human life. Examples of this would be controlling software of spacecraft or planes.

One such system is Real-Time Executive for Multiprocessor Systems (RTEMS). RTEMS is an open-source Real-Time Operating System (RTOS) that provides all the necessary tools and algorithms to support the work of *mission critical* applications in both uni-processor and multiprocessor configurations. It is widely used in industries such as aerospace, healthcare and embedded systems, as well as it supports a wide variety of processor architectures and configurations (2).

In systems such as RTEMS, it is not only desired but is a requirement to have full code coverage when writing tests due to the importance of the system's continuous operation. However, while this practice ensures that the system is well tested, it still doesn't guarantee that it is bug-free. The problem here is that it is also very time consuming to figure out all of the edge cases and write tests for each of them, while manually written tests can themselves contain accidental bugs. Generating whole test suites instead of manually

developing them is a potentially much more efficient and robust solution.

1.2 Project Overview

One way of doing the test generation is with the help of Promela/SPIN verification tools. Promela allows to create models of a system, which can be used by the SPIN Verification tool to produce potential counter-examples that show where the system would fail. Promela inherently supports the modelling of multi-threaded systems with multiple processes, making it even more suitable for the task.

The problem with modelling RTEMS directly as a whole is that the system is massive. It is a mature software that has been constantly worked on for almost 30 years. Fortunately, its functionality is split into *'Managers'*. Each such manager is responsible for a different part of the operating system's functionality. Therefore, we argue that it is much more optimal to model each manager separately, potentially finding a common modelling pattern for all or most of them. Separately taking care of each manager allows for a higher degree of precision within the model, a more manageable scope and allows to avoid State Space Explosion (3).

Furthermore, historically there was not much research done to verify RTEMS behaviour on the multiprocessor systems. While there has been a large amount of work put into developing multi-processing scheduling and resource sharing algorithms over the past seven years (4, 5, 6), the formal verification of those is still relatively new. Creating verification models is a step towards validating the system behaviour on such configurations.

Therefore, this project's objective is to create a Promela model for one of the main RTEMS Managers and then use this model to generate a comprehensive RTEMS test suite for that manager. The model-based generated tests will then be compared to existing test suites within RTEMS to evaluate their effectiveness. At the same time, the developed methodology will be experimented with on other managers to establish how extensible and scalable it is and whenever there is a potential for a general approach to modelling all of the RTEMS managers.

1.3 Contributions

The results and findings of this project are expected to be relayed to the RTEMS community and included into RTEMS source code (7). The generated test suite will provide additional testing, and Promela models will help advance formal verification of the system. This will increase the system's robustness and can potentially help find potential bugs and inconsistencies within the system.

Furthermore, this research will help further improve the existing test framework and can save precious development time during test creation. Simplifying the test suite creation process will also make it more accessible for the open-source community to contribute.

Finally, this work will help with the efforts to transition towards more comprehensive support of multiprocessor systems from uni-processor ones (6) and formally verify such configurations.

1.4 Dissertation Structure

Chapter 2: Background goes into in-depth details about what is RTEMS, the difference between supported processor configurations and architectures in the context of RTEMS, what is Promela and SPIN, as well as it touches on the existing state-of-the-art solution.

Chapter 3: Design presents a detailed overview of how the proposed test generation framework works, the steps necessary to achieve final results, as well as which part of the system is being modelled and under which configuration.

Chapter 4: Implementation contains the exact details of how the steps described in *Chapter 3* are implemented. It delivers the code examples, configuration details, commands and instructions that can be used to reproduce the work presented.

Chapter 5: Evaluation assesses the accuracy of the model, the effectiveness of the test generation framework through comparison with existing RTEMS test suites and evaluating the potential scalability and possibility to extend it to other parts of the RTEMS system.

Chapter 6: Conclusion summarizes the results of the projects, as well as defines potential out-of-scope improvements and directions for future research.

2 Background

Before going into the details, it needs to be established what is, in fact, RTEMS, the difference between Promela and SPIN and what research has been done in the field already.

2.1 Real-Time Executive for Multiprocessor Systems

As already mentioned in **Section 1.1**, RTEMS itself is a Real-Time Operating System. Like any Operating System (OS), it provides a layer between the application software and underlying hardware. That generally allows application developers to omit knowing the in-depth details of the hardware they operate on and instead focus on writing the application. It is then OS's responsibility to allocate hardware resources to the application, such as processor time and memory. Suppose more than one application or an application with multiple processes is involved. In that case, the OS will be deciding what gets to use the processor first, and for how long, based on the scheduling algorithm it is utilizing. OS also exposes services that can be used by the application developers through the API endpoints, such as memory management, synchronization controls, scheduling algorithms, tools for communication between processes or networking protocols.

The "Real-Time" part of the name means that it supports applications where the timing of the application execution can be of utmost importance. That can be concerning internal processes, as well as interaction with external signals or devices, such as sensors (8). A good example could be radiation level sensors at the Nuclear Facility. The readings must be performed at the exact time intervals and no later. Otherwise, the data is considered outdated, and it is assumed that there is a problem with a sensor. That makes sense since it can potentially lead to radiation poisoning of the personnel on-site and possible result in death.

The Nuclear Facility example is very vivid and over-simplified. Not every application requires such a level of strictness in terms of timings. Sometimes it is more along the lines of "it would be best if this task was finished before this deadline, but otherwise, it should still be fine". Generally, as mentioned in (1, 8), it is possible to split the application requirements in

terms of meeting the deadline into three categories:

- Soft real-time - where missing a deadline does not necessarily mean negative consequences, and the results still may be of some value to the application
- Hard real-time - where missing a deadline will most likely result in negative consequences, and the results of the execution are useless to the application
- Safety-critical - where missing a deadline will result in negative consequences, potentially endangering the integrity of the whole system and even human life.

Since timings are so important when working in real-time, unlike typical OS, RTOS guarantees non-determinism in its operation (8). That means that whenever the application developer uses APIs exposed by RTOS, they have the guarantee that the operation will always take the same amount of time to execute, regardless of the system's current state. Additionally, RTOS can provide scheduling algorithms that are not only priority-based but also consider the deadline of the process.

RTEMS also supports a variety of APIs and libraries. It contains POSIX API (9), as well as it is POSIX compliant in terms of operations that are focused on applications with a single process and a single user involved (8). It also has support for *newlib* and *binutils* that are available to be loaded and installed through RTEMS Source Builder (10).

RTEMS exposes most of its native services through managers and their directives, also known as Classic API (1). The general structure of managers is presented in **Figure 2.1**

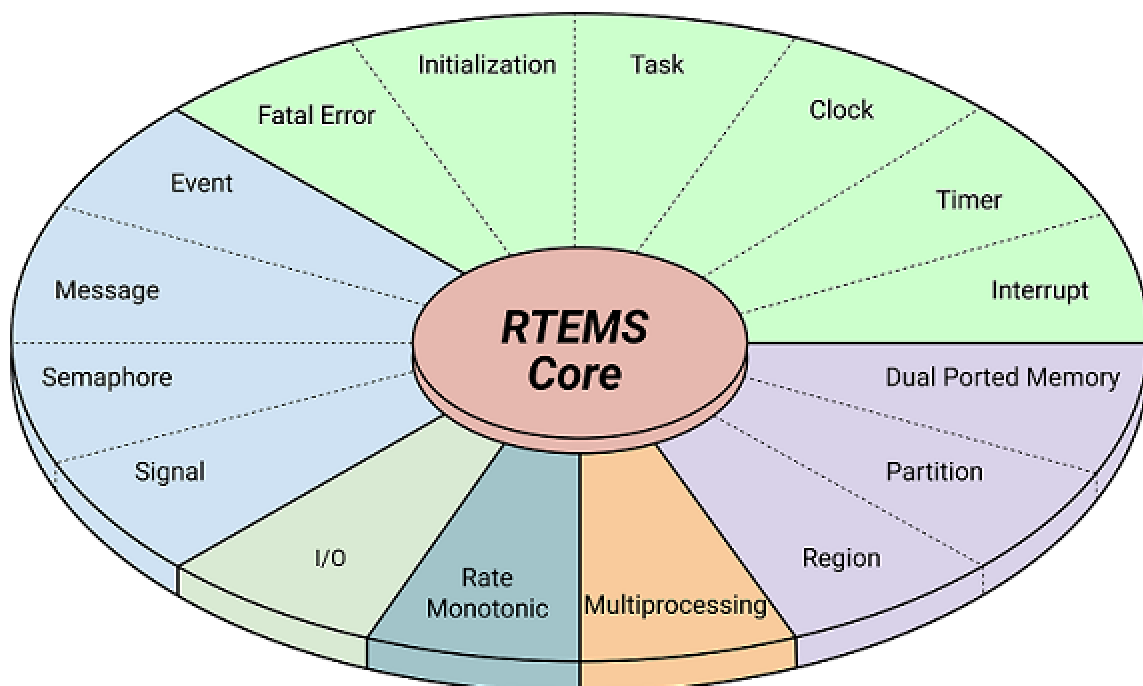

Figure 2.1: Organization of RTEMS Classic API Managers. (1)

Some managers are grouped under similar categories, often called "resource groups". There are no official names for these groups, but in **Figure 2.1** it is possible to see the split through the use of colours. *Green* represents managers that help coordinate scheduling and management of the processes. *Blue* represents tools for communication between processes and their synchronization. *Purple* ones deal with memory, and the rest are self-contained. Nonetheless, all of the managers internally use a set of core instructions provided by RTEMS core, such as object manipulation or thread dispatching (1).

RTEMS is continuously under development, so it has developed a few more managers since the release of the structure in the figure. These managers are:

- Barrier Manager - that provides directives for bulk synchronization of the tasks. It goes into *green* resource group.
- Cache Manager - that provides directives that help manage data and instruction caches. It goes into *purple* resource group.
- User Extensions Manager - that allows supplementing the system with extension routines made by the application developer. This manager is self-contained.

Portability is one of the major RTEMS advantages as there are no assumptions regarding underlying processor architecture. Most of the hardware-specific dependencies are covered by RTEMS and, to a degree, are irrelevant to the application developer (1). At the same time, it provides tools that allow for custom user extensions to be easily augmented to the system if necessary. Due to this isolation and the fact that RTEMS is open-source, it supports a wide variety of microprocessor architectures and specific boards built on those.

Currently, RTEMS supports over 20 different architectures while also allowing some room for customization of the ports. The supported architectures vary between non-embedded systems specific to embedded system-specific, including those used in both. The most popular ones are ARM, RISC-V, x86 or Xilinx MicroBlaze. (11).

In terms of support for the specific boards under these architectures, RTEMS supports them through Board Support Packages (BSP). There are currently more than 200 BSPs fully available and ready to be installed (12). Generally, BSP covers all the necessary hardware configuration and controls, such as initialization, handling of board devices, as well as handling of the interrupts and data transfers within the board (8).

It is also important to note that since RTEMS is open-source, the code is fully available to the users and can be modified as much as needed.

2.2 SPIN and Promela

Promela (Process or Protocol Meta Language) is a **not** a programming language. It is a **modelling language** that is very C like and was introduced by Gerard J. Holzmann. It extensively supports the modelling of asynchronous processes and allows for dynamic creation and control of those. It also includes tools that allow communication between those processes. Due to that, its primary use case is the modelling of complex concurrency problems and systems that rely on parallel computing. Even though Promela supports most of the standard C features, it does not support pointers, which need to be modelled by other means chosen by the programmer. Additionally, there are quite a few evident differences, reminding that Promela is a modelling language and not a programming one. Such examples are the execution of `if` and `do` statements, where any non-blocking statement may be chosen to execute, or statements that block process execution while they are false. The important language features used in this work are explained and showcased in detail in further sections.

On the other hand, SPIN (Simple Promela Interpreter) is a verification tool that uses Promela models to verify the modelled system. SPIN can run in two modes:

- Simulation Mode - in this mode, SPIN picks one of the possible executions non-deterministically and simulates its run. During the run, it will typically print out the execution log and finish when all the processes terminate. If the processes do not terminate, the simulation can potentially run forever.
- Exhaustive Verifier - in this mode, SPIN performs model checking on the given Promela model. During the verification, all state variable values that are possible within the defined model are checked. It means that the SPIN verifier will be going through all of the possible execution scenarios to try and find a counter-example where the defined model does not hold. If all the statements within the model are holding in all of the checked possibilities, the model holds. Otherwise, the counter-example is printed, showing the execution in which the model does not hold. This property will be later used by the approach in this project and is described in more detail in **Section 3.2.1**.

In both modes, a complete execution order of processes within the model is non-deterministic. This property helps to establish whenever the order of execution can break the model. At the same time, it can lead to a massive increase in the possible states that a system can be in during specific points throughout execution. If this is not desired, an appropriate process synchronization can be implemented. This way, the number of states is reduced since access to shared resources is given and critical sections of the model are executed in a predefined order, regardless of the order of other operations. Implementing

process synchronization in the model can also help establish whether chosen synchronization mechanisms would perform as expected in the real system.

SPIN utilizes C code to perform the actual verification, which is why it is possible to utilize many of the C features within the Promela modelling language. That also makes it relatively lightweight and optimized performance-wise.

Using both Promela and SPIN allows to perform model checking to determine whether the modelled systems hold and execute as intended. Systems are also checked for any adverse execution situations such as deadlocks, where all processes are waiting for some variant of a shared resource that is locked, or starvation, where some processes are denied shared resources, such as CPU time, in favour of others, unintended. Model checking can also establish whenever all of the code is executed and whenever a system is in desired states during specific execution points or never is in undesired ones.

Both of these tools are well established within the industry and are widely used, with continuous updates happening since the 1990s. Combined with their inherent support of verification of parallel processes and low pre-requisites needed to work, it makes it a good choice for this project.

2.3 State-of-the-Art Work

Historically, RTEMS was not designed to support Multiprocessing (6). The "Multiprocessing System"' part in its name stood for "Missile Systems" or "Military Systems". Most of its scheduling and resource sharing protocols assumed the uni-processor environment. Once multi-core systems started to become increasingly popular, it became apparent that the support for these systems would have to be included. However, that was not an RTEMS specific problem and research was done in the field to solve the issue of performing resource sharing in multi-processor scenarios. One such popular solution is Multiprocessor Resource Sharing Protocol (MrsP), also generally known as *priority-ceiling protocol* (5). Even then, the protocol still had its flaws, like those pointed out in (4).

RTEMS picked up on that work. However, as mentioned in (6), attempts to modify the existing system to support multiprocessing were relatively unsuccessful and instead, it required a general overhaul. Currently, the state-of-the-art provided by RTEMS is called Symmetric Multi-Processing (SMP). While work is still being done to improve and expand it, it supports most of the functionality provided by RTEMS, such as Classic and POSIX APIs, as well as it supports it for most common platforms, such as ARM or SPARC. Additionally, it has a choice of SMP supported scheduling algorithms and works with some of the parallel programming languages, such as 'OpenMP' (13).

While the support for multiprocessing in RTEMS is growing, system guarantees for these

supports are lacking. Since RTEMS is an RTOS, its SMP configurations must be tested and qualified, just like uni-processor configurations. As per *IEEE* definition, qualification is "Formal testing to demonstrate that the software meets its specified requirements" (14). However, as RTEMS development is open-source, so is its qualification. There are, however, ongoing efforts and guidelines to standardize the process (15, 16).

The project in this dissertation is heavily based on the most recent RTEMS qualification effort sponsored by European Space Agency (ESA). It is a continuation of work done by Prof. Andrew Butterfield at Trinity College Dublin. As the ESA project was only recently released on January 26th 2022, the reports are not yet publicly available on ESA GitLab (17), nor they are published. They can, however, be accessed within the qualification package itself and the overview documentation is available online (18).

The work on automating the test framework within the ESA qualification package focuses primarily on using the Promela model and then refining the output of model verification into the tests. The process was done by using additional python scripts and template C code. The project covered Chains API, Event Manager and Thread Queue services of RTEMS Classic API as they are extensively used within multiprocessing environment (17).

The original project focused on SPARC architecture, and therefore it was chosen for the project in this dissertation. SPARC stands for Scalable Processor Architecture and is a CPU instruction set architecture primarily based on a Reduced Instruction Set Computer (RISC). It was designed by Sun Microsystems, and as they mention in (19) "SPARC was designed as a target for optimizing compilers and easily pipelined hardware implementations". It also featured a sliding Register Window (20), which was highly innovative at the time. Even though SPARC is an old architecture dating back to the 1980s and 90s, it is still widely used in embedded systems and the aerospace industry.

2.4 Alternatives Considered

RTEMS is not the only Real-Time Operating System. Alternatives exist, although most are either not as popular or are closed source. However, two other promising alternatives were looked at.

One of them is LITMUS^{RT}, which is an extension of the Linux kernel to support real-time operations. However, it last released was in 2017, and it generally serves as a proof-of-concept and is more used in an experimental way to test multiprocessing related algorithms and structures. Otherwise, it does not see much use within the industry and lacks stability to be used in real-world applications (21).

Another, more popular alternative is FreeRTOS. FreeRTOS is one of the leading Real-Time Operating Systems. It is open source under the MIT license and is widely used in

micro-controllers and Internet-of-Things devices due to its small memory requirements and low power consumption. Just like RTEMS, it receives constant updates and has support for multiple architectures and platforms. Due to its wide use, it also has a wide range of user-made libraries and APIs that can be utilized (22). However, native FreeRTOS interfaces are not as clearly separated into modules as RTEMS managers, and much of the non-core behaviour is propriety. While FreeRTOS functionality is well supported in terms of documentation, it is not as exhaustive as the one of the RTEMS. The process of sending patches to the system and documentation is also not described as well as it is within RTEMS Community.

Due to that, RTEMS seemed like the best choice for the project. Additionally, as mentioned in **Section 2.3**, there was already groundwork done in the SPIN/Promela field during ESA sponsored project. Basing this dissertation on existing work from the ESA project was more appropriate in scope, considering the time constraints, rather than starting research from scratch on another RTOS.

3 Design

Using Promela to do test generation requires multiple supporting files and configuration steps in order to work. This section will describe in detail the configuration used, the general overview of the test generation process, and describe the details of RTEMS Barrier Manager. The design is utilising a scenario-based approach (23). It is primarily based on the previous work done in RTEMS Events Manager by Prof. Andrew Butterfield from Trinity College Dublin, which was funded by ESA in the project mentioned in **Section 2.3**. That work is described in the document called "FV-201 Formal Verification Artefacts (Architecture, Models, Assumptions, Traceability, Supporting Tests)", which can be found in ESA Qualification Data Pack (QDP) (18).

3.1 RTEMS Configuration

To start off, the RTEMS source and Tool Suite version must be chosen. For the purpose of this project, the latest version of RTEMS was chosen, which is RTEMS 6. The chosen architecture is SPARC since, as mentioned in **Section 2.3**, previous work was based on this architecture, and due to time constraints porting the framework onto a different architecture is out of the scope of this project. Acquiring RTEMS sources (7) allows the development of applications using the functionality provided by RTEMS. At the same time, installing the RTEMS Tool Suite allows one to build and compile such applications. It also provides some additional tools like simulators or test coverage report generation.

To be able to run the RTEMS application, Board Support Package (BSP) has to be installed. As mentioned in **Section 2.1**, BSPs provide all the necessary software for RTEMS applications to work on a specific piece of hardware (12). This project's tests are performed on the "gr712rc" system on a chip based on "Leon3" SPARC conformant architecture models. "Gr712rc" is equipped with two cores. There is also a more powerful version of the processor based on the SPARC "Leon4" architecture model. It is the "gr740", equipped with four cores. These chips are widely used in the aerospace industry and therefore are used in this, and previous research (18). RTEMS provides BSPs for both of those systems, ready to be installed. To run the tests, the SIS SPARC simulator tool is used. It is capable of

simulating Leon based systems and is installed together with SPARC Tool Suite. The SIS simulator can simulate arbitrary processor configuration so that, for example, "gr712rc" is not limited to two cores and can instead execute on four cores. However, "gr740" is used in the real world instead when four cores are needed. While the work is mainly done on "gr712rc", it can also be executed on "gr740" and yields identical results. Both processors are capable of running in a uni-processor configuration with only one core active if needed.

Before touching on implementation details, it is important to note the application configuration that the generated tests will be executed on since RTEMS supports a vast range of those. Specifically, this project is focused on testing using Symmetric Multiprocessing (SMP) configuration. SMP allows for true concurrency since it can handle configuration with multiple processors running tasks in parallel, unlike in its uni-processor counterpart, where only one task can run at one time, and the parallelism is simulated by the operating system (24). SMP supports scheduling on multiple processors by using one or more of the scheduling algorithms. The algorithms available, as per (25), are:

- "Earliest Deadline First SMP Scheduler" - where there is a split between tasks with a deadline and "background" tasks that do not have the deadline. Every task with a deadline defined has a higher priority than any task without a deadline. The closer the deadline, the higher priority the task has. The fixed-priority scheduling rules apply within the background tasks, where a task with the highest priority gets executed.
- "Deterministic Priority SMP Scheduler" - which is an implementation of a fixed priority scheduler with a First-In-First-Out (FIFO) task queue per priority level.
- "Simple Priority SMP Scheduler" - which works the same as the deterministic one but instead has only one queue for tasks, and that queue is sorted by task's priority levels.
- "Arbitrary Processor Affinity Priority SMP Scheduler" - which works the same as the deterministic one with an addition of support for arbitrary task processor affinities.

The one used by our system is the default "Earliest Deadline First SMP Scheduler" (EDF SMP). However, since our application does not use deadlines, this scheduling protocol becomes a fixed-priority SMP scheduler. In our configuration, there is only one scheduler at work unless four or more processors are configured, in which case there are three schedulers involved, all using EDF SMP scheduling. That means that different code that handles SMP will be involved when executing tests on four or more processors, which is desirable for the greater test code coverage. This scheduler setup is defined inside the application configuration by using the following macros:

```
#if defined( RTEMS_SMP ) && \
```

```
( CONFIGURE_MAXIMUM_PROCESSORS == 4 || CONFIGURE_MAXIMUM_PROCESSORS == 5 )

#define CONFIGURE_SCHEDULER_EDF_SMP
RTEMS_SCHEDULER_EDF_SMP(a);
RTEMS_SCHEDULER_EDF_SMP(b);
RTEMS_SCHEDULER_EDF_SMP(c);
```

There is also support for Distributed Multiprocessing in RTEMS, where the notion of "nodes" is introduced (26). In this configuration, the system consists of multiple "nodes" of processors, and there is communication between them through the network messages. However, this use case has not received any updates within the last two years and is being slowly deprecated by the RTEMS community. Due to that, it is of no interest to this project. While there are still references to "local" and "global" nodes present in some parts of the documentation, the notion applies solely to Distributed Multiprocessing. In SMP, all processors are grouped or clustered and are considered one "local" node.

In order to be able to perform testing, the BSPs have to be built with the `` `BUILD_TESTS=True` `` option. Additionally, to enable SMP support, the option `` `RTEMS_SMP = True` `` needs to be added as well.

The generated tests are designed in a way that will pass in both uni-processor and SMP configurations, regardless of the chosen scheduling algorithms. That reduces the test flakiness due to configuration changes. This compatibility is possible due to `` `#if defined(RTEMS_SMP)` `` macro. It allows declaring conditional code blocks, such as scheduler or core count checks, that are executed only when SMP support is enabled. This macro can be used within C test code, as shown in the example below:

```
#if defined(RTEMS_SMP)
sc = rtems_scheduler_ident_by_processor( 1, &ctx->other_sched );
T_rsc_success( sc );
T_ne_u32( ctx->runner_sched, ctx->other_sched );
#endif
```

However, it is important to note that not all code, specifically the one that handles the behaviour when SMP is enabled, is executed when running in uni-processing mode. This behaviour can result in smaller reported code coverage when generating coverage reports for the tests. When executing in uni-processing mode, the scheduler uses the default "Deterministic Priority Scheduler", also known as the "Fixed Priority Scheduler" scheduling algorithm.

3.2 Test Generation

The general approach to generating and executing tests through Promela models consists of three main components, done in the following order:

1. Modelling of the tested functionality in Promela.
2. Refinement of SPIN generated output into tests by using dictionary and templates.
3. Creation of supporting test runner, configuration and specification files within RTEMS to compile and run the tests.

3.2.1 SPIN/Promela Modelling

SPIN/Promela modelling is typically used to generate counter-examples to the modelled behaviour. Given the Promela model, the SPIN model checker will show whenever it is possible to find the counter-example where the system would break, stall or be in an undesired state. Otherwise, it will succeed if it is impossible to determine that in a finite number of checked scenarios, indicating that the model holds.

However, for the purpose of this research, we will be using this process in a different direction. Since tests usually indicate how the program *should* work, the generated scenarios will show us how a healthy program should behave. This reversal is achieved by adding an `` `assert(false);` `` statement at the end of the model, which always fails. This trick 'lies' to the SPIN, deliberately indicating that the model does not hold at that point. Because of that, SPIN will consider it as a counter-example and produce a trail of execution for this specific order of space states all the way up to the point of the failed assertion statement.

With Promela, we are trying to create a model that mimics the system's behaviour as close to reality as needed. It is used to capture all of the behaviour and functionality available to the user through the given manager or API. It models the execution of all of the directives, including all possible options, edge cases and all of the possible return values. However, the model does not have to strictly follow the procedures performed in a real system, as long as the details of all critical steps and system states are captured. That comes from the fact that the model file itself is not used in the later refinement and test generation process. This way, we can introduce some shortcuts and placeholders, such as modelling `` `NULL` `` with 0 or giving arbitrary IDs to the objects. This freedom allows to factor out the internal context switching with memory and object allocations, as well as it allows to avoid modelling some of the C language boilerplate code.

In order to be useful for test generation, the Promela model contains `` `printf` `` statements in the relevant points of the model and is executed using SPIN Verifier. The statements

contain `@@@` prefix annotations so that logs starting with these are known to be for test generation purposes. Running SPIN produces a complete log of execution. Additionally, categories are assigned to the log messages to be easily differentiated whenever the message contains state, function call or parameter values. An example below illustrates a log message, followed by a call to a directive:

```
@@@ 4 LOG WAIT 1 Over
@@@ 4 CALL barrier_ident 1 1
```

There are multiple log files produced, and each log file represents an execution path with a specific set of parameters - a scenario. There can be as many scenarios as required to test the system comprehensively. Each of the scenarios is used to generate a separate test.

Due to the fact that model execution is non-deterministic unless strictly controlled, the generated log files can differ slightly. However, given that the critical states are logged regardless and code blocks where precedence matters are utilising synchronisation mechanisms, such as semaphores, it does not have any influence on the generated tests.

Using the log files instead of actual Promela files allows us to create specific scenarios without the burden of creating complicated language processing systems. Directly parsing Promela files would require its own language processing program and would possibly need to include a way of determining the execution precedence. While it could potentially make test generation more general, as there would be no need to adjust the refinement file to the new system under test, it would require significantly more effort and time to create such a system and is therefore out of the scope of this project.

3.2.2 Model Refinement

Refining the generated logs require a few vital components to create a test suite.

One of them is a dictionary file. It is called a 'dictionary' since it contains a list of keywords that appear within the generated logs and their mapping to the corresponding blocks of the 'C' code. The keywords in the log can also be followed by the scalar parameters, which are imported into the code as well. This conversion allows automatic placement of the directives and function calls that were called during the model run into the test to check if the return values and task states correspond to those predicted within the model. The generated blocks are put together into "test segments". Each such segment corresponds to an execution of a single process, be it an application task or a system task.

However, these segments are not the only things that are needed for a complete test. In addition to the "varying" part of the test, there are additional blocks of code that always

have to be present. Three following template files provide them:

- Preamble - that contains all of the needed header and options. It is placed before the code part refined by the dictionary.
- Post-amble - that contains the functions that provide a starting point for the created tasks that will run the generated test segments.
- Runner code template - that contains all of the code necessary to set up the test environment, such as the test fixture and all of the necessary objects used during the test execution.

All of the above are combined through the refinement python program written in "Coconut" functional language. It creates a test case file for each scenario log previously generated by SPIN.

3.2.3 Supporting Files

There is still one more step needed to complete the test suite. Many basic functions and object operations within the separate generated test files can be bundled into helper functions. However, it would not make sense to duplicate those across all of the generated tests. Therefore, there are "test runner" header and code files containing all of the function definitions and hosting all of the necessary structures populated and used by the generated tests. They also contain some of the test fixture code, such as cleanup, scope and teardown functions. Therefore in actual tests, the functions can be simply linked to a fixture and executed by the testing framework when the test is finishing and shutting itself down. Additionally, there must be a main "test case" file containing all of the generated "test cases" as macro calls to the RTEMS test framework. It acts as a wrapper to a generated part of the suite and calls the "runner" functions of the generated test case files.

Finally, for the new test suite to be picked up by the RTEMS build system, compiled and then executed, it needs a configuration and specification file. The configuration file contains all of the necessary information about memory allocation, scheduling algorithms used, and it defines the limits on the creation of RTEMS objects. The specification file is the binding element for all the test suite files. It is written in *YAML* and contains linking instructions for the system to combine all of the "test case", "test runner", and test configuration files into an executable. Executable is then compiled and built during the BSP build process and can be executed by using one of the simulators, such as "SIS", or the "rtems-test" tool.

3.3 RTEMS Barrier Manager

As pointed out in **Section 2.3**, one of the models in the previous research was based on RTEMS Events Manager. It provides one of the main tools that allow synchronisation and communication between tasks. To follow the pattern of testing functionality that is crucial to SMP and one that involves synchronisation of tasks, it was decided to model RTEMS Barrier Manager of Classic API.

The Barrier Manager of RTEMS provides one of the main synchronisation tools available within the operating system. As the name suggests, it provides a "barrier" on which the tasks can block and wait to be released. When the barrier is released, all of the tasks waiting on it are no longer blocked and are available for execution according to the scheduling policy in place. This mechanism can help synchronise multiple tasks in scenarios when all or a certain number of them have to reach a specific stage in their execution for the whole system to proceed or when tasks have to wait until something is completed before they can execute (27). Since this manager provides synchronisation mechanisms for the RTEMS tasks, similarly to Event Manager, it made it a suitable candidate to try and extend the existing test generation framework to it.

The Event Manager has a clear separation between system and non-system operations. Handling of the system events has its own, slightly different code implementation, separate from the primary directives. This separation allows for the creation of the second set of events used internally by the RTEMS system only. The behaviour of system directives is practically identical, and therefore they share the same Promela model. However, tests for both system and regular directives had to be generated to achieve the best code coverage. In Barrier Manager, there is no such separation. Therefore the directives provided by the primary barrier implementation are the only ones used by both system and applications and are the only ones that need to be tested.

There are two types of barriers - Automatic and Manual. Manual barriers have to be released by the manual release directive and will block any task waiting on it, regardless of the number of tasks waiting. On the other hand, automatic barriers have the number of maximum waiters specified and are automatically released once the number of waiting tasks reaches that threshold. That means that the task that broke the threshold will not wait on the barrier but rather trigger the release of it (27).

Tasks can also have an option of a Timeout. By default, there is no timeout set for the task waiting at the barrier, which means that it can potentially wait forever since the barrier may never be released. However, if that is not desirable, the task can be specified to timeout after a specific amount of time. It will then continue its execution, with the return code indicating that it timed out rather than being released by a directive (27).

In terms of directives that barrier manager offers, there are 5 of them. For all of them, if the directive is successful, it returns a return code indicating that; otherwise, the return status code will indicate why the directive failed. The directives are following:

- `rtems_barrier_create` - creates a barrier object with a given name and the attribute indicating if the barrier is manual or automatic. If successful, it places the barrier ID of the created barrier into the passed variable.
- `rtems_barrier_ident` - identifies a barrier object by given name. If successful, it places the barrier ID of the identified barrier into the passed variable.
- `rtems_barrier_wait` - makes the calling task block at the barrier. If it fails or tasks are released by a timeout or deletion of the barrier, it is indicated by the return code.
- `rtems_barrier_release` - releases all the tasks waiting at the barrier.
- `rtems_barrier_delete` - deletes the barrier object and releases the tasks waiting on it.

All of the above directives are included in a Promela model, and all of their possible return values are modelled. However, while covered implicitly to a degree, the model does not perform checks on tasks correct preemption when dealing with varying priorities since, technically, it is outside of the Barrier Manager's scope. However, it can be easily extended to do so in the future if such a need arises.

4 Implementation

Based on the design in **Chapter 3** and on the Event's Manager model and template files made by Prof. Andrew Butterfield and Sebastian Huber, which are contained in ESA QDP (18), all three components of the test generation framework - model, refinement file and test generation templates were adopted, generalised and cleaned where possible. Most of the code was overhauled to work with the Barrier Manager, except for generic functionality, which could be reused between the manager models and tests.

4.1 Model

For our purpose, we decided to utilise three tasks. One of the tasks is the main 'Runner' task, and the other two are worker tasks. This split means that the 'Runner' task is synchronised to always run before the workers. While this designation does not affect much within the Promela model itself, the task precedence plays an important role later in the generated tests. Within the model, tasks can have five different states:

- `Ready` - which means that the task is ready to be executed or is executing.
- `Zombie` - which means that the task has finished its execution.
- `BarrierWait` - which means that the task is blocked waiting on the barrier.
- `TimeWait` - which means that the task is waiting on the barrier but has the timeout option activated.
- `OtherWait` - which means the task was preempted and is waiting to be readied again by the system. Being "preempted" means that the task had to yield to the processor, usually favouring the task with higher priority.

Worker and runner processes are the primary processes within our model. Each of them is a task that uses the Barrier Manager directives in a controlled way, decided by the scenario chosen. It is configured using the `TaskInputs` options. While there is not much difference between the structure of the three tasks, the difference between the runner and worker tasks becomes crucial within the generated tests. Task semaphores control the

precedence of execution. In general, the `Runner` task is the task that will run first and the one that will create the barrier. After that, the Runner will block on its own semaphore in most scenarios so that other tasks have a chance to run to, e.g. call the wait directive on the barrier. That has to be explicitly done since the scheduling algorithm used will allow the task to continue executing until it blocks or is preempted in favour of a higher priority task. After `Runner` is blocked, the `Worker0` runs and then `Worker1` runs. At the point when all tasks get to execute, the rest of the execution order can be non-deterministic or depend on a specific scenario. In the end, however, the `Runner` task will block waiting on the worker semaphores, which will be released when worker tasks finish.

The System Process is looping through all tasks, ensuring that any preempted task is placed back into its `Ready` state at some point. It also checks whenever all tasks are Zombies, which would indicate the end of the model; thus, the System Process would finish as well.

The Clock Process works similarly to the System Process. However, it loops through the tasks to see if any of them are waiting for a potential timeout and reduces their timeout ticks by one if so. This process is specifically helpful in the scenario where the task blocks on the barrier but has a timeout period specified. Once the timeout period specified by the task ends, which is indicated by the task tick count reaching 0, the Clock will place it back to the `Ready` state. However, if the task returns from the directive before timing out, it will no longer be in the `TimeWait` state and can no longer time out. The Clock Process finishes when the System Process is finished.

Only one primary process is allowed to run at a given time within the Promela model. System and Clock processes are considered to be *background* processes and can always run. `Runner` and `Worker` can only be executed when they are in the `Ready` state. To simulate potential preemption of the processes, when, e.g. higher priority tasks are released from a barrier or timeout, the check on priorities of all non-background tasks is performed through `preemptIfRequired(callerid)` call. The convention where a lower number indicates higher priority is used to decide the task state. The excerpt from the check is listed below:

```
(...)
if
:: tasks[callerid].preemptable && tasks[t_index].prio < tasks[callerid].prio &&
   tasks[t_index].state == Ready ->
     tasks[callerid].state = OtherWait;
     printf("@@@ %d STATE %d OtherWait\n",_pid,callerid);
     callerid = t_index;
:: (!tasks[callerid].preemptable || tasks[t_index].prio > tasks[callerid].prio) &&
```

```
    tasks[t_index].state == Ready ->
      tasks[t_index].state = OtherWait;
      printf("@@@ %d STATE %d OtherWait\n",_pid,t_index);
::  else -> skip
fi
(...)
```

After the check, the task calls `waitUntilReady(id)` with the potentially blocking statement to yield the processor to another task if necessary:

```
tasks[taskid].state == Ready;
```

All of the processes have their own Promela IDs, which do not change between runs as they get assigned in order of creating within the model. These are described in **Table 4.1**

Table 4.1: Processes and their corresponding Promela IDs.

Name	PID
System	1
Clock	2
Runner	3
Worker0	4
Worker1	5

Binary semaphores are utilised to synchronise the tasks. In the current structure, there is one such semaphore per task. Semaphores, by definition, do not have an owner and can be obtained and released by any of the tasks. Binary Semaphores can only be in either acquired or released state. When the semaphore is created, it is in the acquired state by default, both in the model and in the actual application. Semaphores have two mechanisms within the model. One is for obtaining the semaphore:

```
inline Obtain(sem_id){
  atomic{
    printf("@@@ %d WAIT %d\n",_pid,sem_id);
    semaphore[sem_id] ;         // wait until available
    semaphore[sem_id] = false; // set as in use
    printf("@@@ %d LOG WAIT %d Over\n",_pid,sem_id);
    }
  }
```

And one is for releasing the semaphore:

```
inline Release(sem_id){
  atomic{
    printf("@@@ %d SIGNAL %d\n",_pid,sem_id);
    semaphore[sem_id] = true ; // release
    }
  }
```

The task is blocked on the semaphore by executing `semaphore[sem_id];`. In Promela, if the statement value is `false`, it cannot be executed, and the process blocks on it until the statement becomes *enabled* to be executed. So in this situation, if the semaphore value under `sem_id` is `false`, the task will block until the value is `true`. Therefore, a semaphore is 'acquired' by setting its value to `false` and released by setting it to `true`.

In theory, priorities could also be used to synchronise the tasks. However, unlike semaphores, they can yield unpredictable results when changing priorities during the test execution. This behaviour is especially evident when working in the SMP configuration with multiple schedulers since the priorities are not shared between the schedulers. Therefore if two tasks are dispatched by different schedulers, their priorities relative to each other will not have any effect. On the contrary, binary semaphore behaviour is consistent irrespective of application configuration. It is also the most straightforward synchronisation mechanism available and requires minimal effort to configure and use it.

In terms of barriers, we are operating on a single barrier and assuming that the actual RTEMS application is configured with `CONFIGURE_MAXIMUM_BARRIERS 1`. There would be no additional code covered if we were to use more than one barrier. It would also be harder to test an overflow in barrier creation and clean up created barrier objects in the actual RTEMS test application.

4.1.1 Modelled Directives

As described in **Section 3.3**, we model all of the five directives provided by the RTEMS Barrier Manager. They are defined as `inline`s within the Promela code. Keyword `inline` indicates a macro, which defines the replacement piece of code or text instead of a symbolic name. It also means that everything defined within `inline` is within the scope of the process where it was called. Due to that, while we can pass parameters to the `inline`, we cannot return anything from it. Therefore we pass the return code and any additional return values as a parameter, which will be available from outside of the `inline` when 'returned' (28).

The following describes how the directives are modelled within the Promela model:

- `rtems_barrier_create`

  Represented as `inline barrier_create(name,attribs,max_waiters,id,rc)` within the model.

  The 'create' directive requires only the name and the attributes of the barrier to be created. The attributes indicate whenever the barrier will be manual or automatic, with automatic cases requiring the maximum number of waiters to be passed to the directive call.

  New barrier IDs start at 1, since 0 is reserved for the `NULL` value:

  ```
  int new_bid = 1;
  ```

  Then after the parameter check, the global barrier array of the model is traversed to find the next open slot, which is indicated by `barriers[new_bid].isInitialised` value being `false`. If all is well, the directive creates a barrier object in this slot. The creation looks as follows in the case of a manual barrier:

  ```
  barriers[new_bid].b_name = name;
  barriers[new_bid].isAutomatic =  barriers[new_bid].isAutomatic | attribs;
  barriers[new_bid].waiterCount = 0;
  barriers[new_bid].isInitialised = true;
  id = new_bid;
  rc = RC_OK;
  printf("@@@ %d LOG %d Created {n: %d, auto: false}\n" ,_pid, new_bid, name);
  ```

  And similarly for automatic barrier:

  ```
  if
  ::  max_waiters > 0 ->
      barriers[new_bid].maxWaiters = max_waiters;
  ::  else ->
      rc = RC_InvNum;
      break;
  fi
  barriers[new_bid].b_name = name;
  barriers[new_bid].isAutomatic = barriers[new_bid].isAutomatic | attribs;
  barriers[new_bid].waiterCount = 0;
  barriers[new_bid].isInitialised = true;
  id = new_bid;
  ```

```
rc = RC_OK;
printf("@@@ %d LOG %d Created {n: %d, auto: true, mw: %d}\n"
        ,_pid, new_bid, name, max_waiters);
```

The index of the barrier in the array is the ID of that barrier. The barrier object holds information about the properties of the barrier and keeps track of the tasks waiting on that barrier and their count. After that, the new barrier ID and `RC_OK` status code, corresponding to `RTEMS_SUCCESSFUL` code in real RTEMS application, are assigned to 'return' variables passed.

In the case where the preconditions for creating a new barrier were not met, the barrier is not created. The passed barrier ID is reset to 0, and the return code indicating an error is returned. Depending on the failing precondition, the return code might be one of the following:

- `RC_InvName` corresponding to `RTEMS_INVALID_NAME`, caused by the passed name being equal to 0. In technical terms, any `'rtems_name'` equal to 0 is deemed invalid.

- `RC_InvAddr` corresponding to `RTEMS_INVALID_ADDRESS`, caused by the passed pointer to the barrier ID variable being `NULL`. In the Promela model, `NULL` is represented by 0.

- `RC_InvNum` corresponding to `RTEMS_INVALID_NUMBER`, caused by passed `max_waiters` parameter being equal to 0, while the barrier is configured as being automatic.

- `RC_TooMany` corresponding to `RTEMS_TOO_MANY`, caused by trying to create the barrier, while the amount of barriers active is at the configured cap. In the model, it is returned if there is no available index in the global barrier array.

All is done in an atomic fashion within the model.

- `rtems_barrier_ident`

  Represented as `inline barrier_ident(name,id,rc)`

  The 'identify' directive provides a way to find the barrier ID through its name. It is important to note that in the case where two or more barriers have the same name, the one that has the lowest index part of the ID will be found first and returned by this directive (27).

  The directive is simply modelled through a `do` loop that iterates over the global barrier array trying to satisfy the following condition:

```
barriers[b_index ].isInitialised && barriers[b_index ].b_name == name
```

If the directive is successful, then the barrier ID and `RC_OK` status code, corresponding to array index and `RTEMS_SUCCESSFUL` respectively, are assigned to 'return' variables passed. Otherwise, the barrier ID returned will be equal to 0, and one of the following error codes is returned:

– `RC_InvAddr` corresponding to `RTEMS_INVALID_ADDRESS`, caused by the passed pointer to the barrier ID variable being `NULL`.

– `RC_InvName` corresponding to `RTEMS_INVALID_NAME`, caused by the passed name being equal to 0 or when the barrier with the given name does not exist.

Again, the whole directive is executed in an atomic fashion.

- `rtems_barrier_wait`

  Represented as `inline barrier_wait(self,bid,interval,rc)`

  The 'wait' directive allows the task to be blocked on the barrier, waiting for its release. The barrier is referred to by its ID within the call. Additionally, the timeout interval can be specified for the task, which means that if the task is not released within that time interval, it will timeout and stop waiting on the barrier. If the passed interval equals 0, the waiting task can potentially wait forever to be released.

  When the task is blocked on the barrier during this directive, it is placed into `BarrierWait` state:

```
tasks[self].state = BarrierWait;
printf("@@@ %d STATE %d BarrierWait\n",_pid,self)
```

If the directive had timeout interval specified, the task is instead placed into `TimeWait` state:

```
tasks[self].tout = false;
tasks[self].ticks = interval;
tasks[self].state = TimeWait
printf("@@@ %d STATE %d TimeWait %d\n",_pid,self,interval);
```

It is important to note that the only case where the task will not be immediately blocked when calling the directive is when it calls it on the automatic barrier and is the task that will go over the automatic release threshold:

```
(...)
if
::  barriers[bid].isAutomatic ->
    bool sat;
    satisfied(bid, sat) // check if enough waiters to release barrier
    if
    ::  sat ->
        barrierRelease(self,bid);
        rc = RC_OK
(...)
```

Nevertheless, if this action releases a task with higher priority, the calling task will get preempted and put into `OtherWait` state.

The task is blocked by executing a statement, which is only *enabled* when the task is back in `Ready` state:

```
tasks[tid].state == Ready;
```

This directive's status code is returned only when the task is released and is executing again. If the task is released by either automatic release or `rtems_barrier_release` directive, the status code return is `RC_OK`. In other cases, it will return:

– `RC_Timeout` code, corresponding to `RTEMS_TIMEOUT`, in the case where the task timed out when waiting on the barrier release.

– `RC_InvId` code, corresponding to `RTEMS_INVALID_ID`, in the case where the barrier under passed ID does not exist.

– `RC_Deleted` code, corresponding to `RTEMS_OBJECT_WAS_DELETED`, in the case where the task was released because the barrier was deleted rather than released. In the model, this is established by checking whether the barrier still exists after the task returns from the wait and is executed.

The check is performed right after the task is unblocked:

```
if
::  tasks[tid].tout ->
    barriers[bid].waiters[tid] = false; // remove self from waiters
    rc = RC_Timeout
    tasks[tid].tout = false; // reset timeout in case we reaquire
```

```
    barriers[bid].waiterCount--;
    preemptIfRequired(tid);
    waitUntilReady(tid) // if we were higher prio, need to pre-empt others
::  !barriers[bid].isInitialised ->
    rc = RC_Deleted
::  else -> // normally released
    rc = RC_OK
fi
```

As per usual, the whole operation is done in an atomic manner. However, the task will break out of atomic behaviour if it yields itself as the result of calling this directive.

- `rtems_barrier_release`

  Represented as `inline barrier_release(self,bid,nreleased,rc)`

  The 'release' directive releases the tasks waiting on the barrier. The barrier that is supposed to be released is referred to by its ID. While the directive is primarily used to release manual barriers, it can also be used to release automatic barriers manually.

  When the release is successful, all of the tasks waiting on the barrier are released and put into `Ready` state:

```
int tid = 1;
do
:: tid < TASK_MAX ->
    if
    ::  barriers[bid].waiters[tid] ->
        barriers[bid].waiters[tid] = false;
        tasks[tid].state = Ready
    ::  else -> skip
    fi
    tid++
:: else -> break
od
barriers[bid].waiterCount = 0;
```

  At the end of the directive, the preemption check is called. If a task with a higher priority was released, the calling task is preempted and put into `OtherWait` state. Otherwise, the calling task continues to execute, and all released tasks are placed into `OtherWait` state instead.

  While not through a direct call of the directive, when the automatic barrier reaches the release threshold in both our model and the real system, the inner release

functions are identical to those called during the 'release' directive.

The `RC_OK` status code is returned, and the number of released tasks is assigned to the passed `nreleased` variable.

If the directive were unsuccessful, the number of released tasks would be 0, and one of the following status codes would be returned:

- `RC_InvAddr` corresponding to `RTEMS_INVALID_ADDRESS`, caused by the passed pointer to the `nreleased` variable being `NULL`. Just as with the ID in previous directives, we model `NULL` to be represented by 0.
- `RC_InvId` corresponding to `RTEMS_INVALID_ID`, in the case where the barrier with the given ID does not exist.

Once again, while the whole directive is within the `atomic` block, the task will break out of atomic behaviour if it has to yield to the task with higher priority.

- `rtems_barrier_delete`

  Represented as `inline barrier_delete(self,bid,rc)`

  The 'delete' directive releases the barrier and then deletes the object by setting `isInitialised` value within the array of barriers to `false`:

```
barriers[bid].isInitialised = false;
```

  In both our model and real-world application, the barrier is released by the same internal function calls as during `rtems_barrier_release` directive. The only difference in the RTEMS application is the passed status code filter. Therefore the preemption mechanism is exactly the same as in the 'release' directive.

  On successful execution of the directive, `RC_OK` status code is returned, and the barrier is deleted. Otherwise, the only possible error can be passing a non-existing barrier ID into the directive, which results in `RC_InvId` status code.

  Identically to the `rtems_barrier_release`, the directive is within the `atomic` block but breaks out of atomic behaviour if it has to yield to the released task with higher priority.

4.1.2 Scenarios

Capturing all of the behaviour and functionality that the Barrier Manager has to offer in a single model execution would be impractical. That would require creating a large number of threads or having an extremely long model with a lot of code duplication. That would also

make generated RTEMS tests very long and unreadable while increasing the potential for bugs and making it hard to fix them. So instead, we split model runs into scenarios. That allows to appropriately scope each test and avoid using `goto` statements, a readability improvement over the Events Manager model. Additionally, some of the scenarios have their own sub-scenarios, which test what happens when the directives do not succeed and if the behaviour in these cases is correct.

So far in this work, `if` blocks were used for conditioning the execution, with strictly one condition passing at any time. However, unlike in well-known programming languages, such as C, the `if` statement blocks are not executed top-down. All statements that have a passing 'guard' condition or do not have any associated condition can be executed. If there is one or more of them, any of them can be chosen (29). Suppose the following example:

```
if
:: cond1 -> statement1
:: statement2
:: else -> statement3
fi
```

In this situation, `statement2` is always available to be chosen since it does not have a condition associated with it, also called *guard*. However, if `cond1` is satisfied, `statement1` could also be chosen to execute. In that case, SPIN will choose either `statement2` or `statement1` non-deterministically. The `else` statement is only satisfied if no other statements can be executed. In the example above, this will never happen since `statement2` is always available. If there is no `else` within the `if` block and no statements are available for execution, the process will block until one or more options become enabled. The same behaviour goes for `do` blocks, with the difference being that the block is looped through until the process `break` out of it.

When choosing scenario and potential sub-scenarios, this non-determinism of the `if` clause is utilised. There is an option within `if` block for each execution scenario, which creates an additional state space and results in an additional counter-example for it. The same goes for sub-scenarios within the scenarios. Within these sub-scenarios, the process input options are modified to change the execution path of the processes or to change directive parameters passed. The following example shows the excerpt of *Automatic Aquisition* scenario and its sub-scenario:

```
(...)
::  scenario == AutoAcq ->
      task_in[TASK1_ID].isAutomatic = true;
      task_in[TASK1_ID].maxWaiters = MAX_WAITERS;
```

```
      if
      ::  task_in[TASK1_ID].maxWaiters = 0;
          printf( "@@@ %d LOG sub-senario bad create, max_waiters is 0\n", _pid);
      ::  skip
      fi
(...)
```

In terms of multiprocessing scenarios, there is an additional following `if` construct after the scenario selection. That means each scenario available can be executed in either uni-processor or multiprocessor setting. That ensures that there is a test for either configuration. The code block is as follows:

```
if
::  multicore = true;
    task1Core = THAT_CORE;
    printf( "@@@ %d LOG sub-senario multicore enabled, cores:(%d,%d,%d)\n",
              _pid, task1Core, task2Core, task3Core );
::  skip
fi
```

As said in **Section 4.1**, every process receives `TaskInputs` containing the options that define which directives the task will execute, as well as what parameter values will be passed to these directive calls. The structure contains the following options that define the process:

```
typedef TaskInputs {
  bool doCreate; // will task create a barrier
  bool isAutomatic; // if doCreate is true, specifies barrier type
  int maxWaiters; // if isAutomatic is true, specifies max barrier waiters
  byte bName; // Name of barrier to create or identify
  bool doAcquire; // will task aquire the barrier
  byte timeoutLength; // only relevant if doAcquire is true, gives wait time
  bool doDelete; // will task delete the barrier
  bool doRelease; // will task release the barrier
  byte taskPrio; // task priority
  bool taskPreempt; // task preemption mode
  bool idNull; // indicates whenever passed id is null
  bool releasedNull // indicates whenever nreleased param is null
};
```

By default, the first process always creates the barrier, and all processes acquire it. The

barrier's type by default is 'manual'. Within the model, a task can get the barrier ID through either `rtems_barrier_create` directive or `rtems_barrier_ident` directive. The execution of other directives is controlled directly by the options, but they go in order of `rtems_barrier_wait` then `rtems_barrier_release` and finally `rtems_barrier_delete`.

There is a total of 11 scenarios, including sub-scenarios. When adding multi-core equivalents, it amounts to 22 possible execution paths. While within each scenario the actual execution sequence of each task can vary between the runs by a small margin, the sequence of directive calls remains the same, so this inconsistency is irrelevant.

The main scenarios are following:

- Manual Barrier Operations as presented in **Table 4.2**.

  Table 4.2: Directive execution in default Manual Barrier Operations scenario.

Task	Create	Identify	Wait	Release	Delete
Runner	True	False	False	True	True
Worker0	False	True	True	False	False
Worker1	False	True	True	False	False

  Additionally, there are seven sub-scenarios that are testing erroneous behaviour, such as passing `NULL` values instead of pointers, as well as testing edge cases, such as using non-existent barrier ID in directive calls, using an invalid name in creation or identification directives, checking if the behaviour is consistent when calling directives, such as `rtems_barrier_release`, on a barrier with no waiters and creation of too many barriers. There are so many sub-scenarios due to the fact that this scenario was picked to test all of the behaviour that is not automatic barrier specific. Manual barrier configuration allows for easier control of the execution and testing of edge behaviour.

- Automatic Barrier Operations as presented in **Table 4.3**.

  Table 4.3: Directive execution in default Automatic Barrier Operations scenario.

Task	Create	Identify	Wait	Release	Delete
Runner	True	False	True	False	False
Worker0	False	True	True	False	False
Worker1	False	True	True	False	False

  Furthermore, there is also one sub-scenario that tests an automatic barrier specific edge case where the value of `max_waiters` passed is equal to 0.

- Automatic Barrier with Timeout as presented in **Table 4.4**.

Table 4.4: Directive execution in Automatic Barrier with Timeout scenario.

Task	Create	Identify	Wait	Release	Delete
Runner	True	False	False	False	False
Worker0	False	True	True	False	False
Worker1	False	True	True (Timeout)	False	True

There are no sub-scenarios here since other scenarios already cover all other functionality.

4.1.3 Generating log files

To run Promela models, the SPIN model checker is used. To run a single scenario picked non-deterministically, the following command should be used:

```
$ spin model-barrier-mgr.pml
```

While it is good to see that it runs as expected, it is not quite useful for the test generation. Instead, it is possible to generate `.trail` files for every single possible scenario and sub-scenario within our model by using the following command:

```
$ spin -run -E -c0 -e model-barrier-mgr.pml
```

Additional options mainly ensure proper formatting.

However `.trail` files are still not useful for test generation purpose, as they just contain step, Promela ID and transition numbers, which are not very readable. Instead, we can use them to generate annotations using SPIN on each file:

```
$ spin -T -t1 model-barrier-mgr.pml > model-barrier-mgr-0.spn
  ...
$ spin -T -tN model-barrier-mgr.pml > model-barrier-mgr-{N-1}.spn
```

Where `-T` option normalizes indentation and `-tN` indicate N-th `.trail` file.

After this, we end up with `.spn` files, which contain a complete log of execution made using `printf` annotated statements within our model. These can now be used for test generation. All the described steps are included and sequenced automatically in the test generation step of `testbuilder.py` python script made by Prof. Andrew Butterfield and Robert Jennings (18).

4.2 Refinement & Test Generation

Refinement is the next step toward generating tests. For the refinement of produced logs into RTEMS tests, a few additional files are required:

- `` `model-barrier-mgr-rfn.yml` `` - this file is a dictionary that allows for refinement between keywords in log files produced by running the model and test C code. It connects keywords to the replacement code and allows to pass the numeric parameters into that code.
- `` `model-barrier-mgr-pre.h` `` - preamble material. It contains the initial `` `#includes` `` required for the generated tests to work.
- `` `model-barrier-mgr-run.h` `` - test setup material. This file contains the setup code that is not generated from the model but is required for the tests to work. It sets up directive calls and pushes the test fixture, which contains a call to the setup code for all the necessary tasks and objects, such as semaphores, and schedulers, if SMP is enabled. The fixture also links to cleanup and teardown code. The test setup template takes the test number as a parameter. Therefore all function calls will be unique, as they are assigned the number of a scenario for which the test is generated. This assignment allows us to avoid duplicating function names and bundle everything under a single header file later.
- `` `model-barrier-mgr-post.h` `` - post-amble material. It contains the C template for Runner and Worker tasks that execute refined code. These tasks do not take any parameters and will always look the same; therefore, they are separated.

The above files are used to assemble a single test file for each scenario generated by the Promela model. All of them are used by the existing python program called `` `spin2test` `` written in Coconut python extension language by Prof. Andrew Butterfield (18). It utilises the dictionary to replace the keywords from a Promela model run log and then combines that with the templates to create a test-runner C file for each scenario.

The template files are simply copy-pasted into the file. The only file that allows us to create some variation within the tests is the refinement `` `.yml` `` dictionary file. Therefore, it is the most important one since it provides a mapping from the output of the model execution into the RTEMS C test code. **Figure 4.1** presents the most relevant contents of the dictionary file that are used in barrier manager directive testing.

Full dictionary code is available in **Figure A1.1**. It's important to note how each replacement code block of C code is mapped to exact keyword that is found within the model execution log and is expecting predefined amount of parameters. However the keywords and passed parameters cannot be arbitrary and are predefined by `` `spin2test` ``

program language definition. The full definition is described in **Appendix A1.2**.

While it may seem that there are not that many keywords that can be used, one can successfully test any part of the system with only a handful of those. It also ensures that the `spin2test` is robust and consistent. Adding more manager specific refinement code could overcomplicate the process on both modelling and refinement levels. Having consistent code here makes it easier to scale and extend to handle other managers without much of the extra programming involved.

The refinement process does not allow for complete freedom regarding what we can put into the dictionary within the replacement C code blocks. The dictionary file does not accept direct assignments within it, such as `bid = 0;`. Helper functions have to be utilised instead.

In the test part generated through refinement using the dictionary, each task gets its own test segment to run, contents of which are dependent on the scenario and dictionary replacements. Test segment number in C file matches with Promela ID number within the model. `TestSegment0` is the only segment that is not task-specific and is used for all the necessary initialisation. In the case of the Barrier Manager, it contains only the initialisation of semaphores, while in the Events Manager tests, it can have something like the initialisation of pending events. Other segments are `TestSegment3`, `TestSegment4` and `TestSegment5`, which are executed by `Runner`, `Worker0` and `Worker1` tasks respectively, as per post-amble template. The example test segment is presented in **Figure A1.2**.

An important note should be made regarding suspending the worker tasks before they finish their execution. If the tasks created through RTEMS Classic API fall off the bottom of the task's body function, that will produce a fatal error. Therefore in this situation, the worker tasks have to be suspended right before they finish until the point where they are deleted. This suspension can be done through, e.g. issuing a wait for an event, such as:

```
(...)
rtems_event_receive(RTEMS_ALL_EVENTS,RTEMS_DEFAULT_OPTIONS,0,&events);
```

which can wait indefinitely until the task is deleted.

To combine everything into tests the `spin2test` has to be executed on each `.spn` file previously produced:

```
$ spin2test model-barrier-mgr 0
  ...
$ spin2test model-barrier-mgr {N-1}
```

Alternatively, this step is also included into the test generation step of `testbuilder.py` script. To use the script in order to generate the tests, the following command has to be issued within the working directory where the Promela model is:

```
$ {testbuider-directory}/testbuilder.py generate model-barrier-mgr
```

This script will go through all mentioned steps till this point and generate `.trail`, `.spn` and actual test case `.c` files.

It is important to note that when refinement was worked on, instead of changing each of the files separately and seeing if the whole test suite works for the new model, one of the existing generated Event Manager test-case files was taken. It was then gradually changed to fit one of the scenarios generated by the Barrier Manager model. This way, it was significantly easier to see which elements should be changed and which should stay. Thanks to temporarily scoping down the task, finding and fixing bugs in the model and test code was much more manageable. It also allowed for some experimentation with reducing the unnecessary boilerplate present in the Event's Manager test code. After the test was running fine, the relevant parts were copied into templates used to generate the rest of the tests. This approach resulted in a faster development time for this step and improved robustness and readability of the code.

4.3 RTEMS Tests

Going through previous steps would generate N number of C test-runner files named `tr-model-barrier-mgr-N.c`, where N is the scenario number. To execute the generated tests, three more files are needed:

- `tc-model-barrier-mgr.c` - which is the test-case setup C file. It sets up the test case calls to all of the generated tests. It used to contain a wrapper to the directives as well, but it is no longer necessary. An example of a test case for a generated test of scenario number 0 would look like this:

  ```
  T_TEST_CASE( RtemsModelBarrierMgr0 )
  {
    RtemsModelBarrierMgr_Run0();
  }
  ```

- `tr-model-barrier-mgr.h` - the test-runner header file. As a header file in C would normally do, it contains the prototypes of all the helper functions required within the tests. It also includes the context object, which saves the state of all the relevant variables within the test scope, such as semaphore, task and scheduler IDs.

Additionally, it stores the wrapped function calls to the directives tested. The header file is linked to generated tests through the preamble file previously explained in **Section 4.2**.

- `tr-model-barrier-mgr.c` - the test-runner setup C file. This file contains the implementation of all the functions defined within the header file. Helper functions include the ones needed for creating, deleting and operating semaphores and tasks. Additionally, they contain teardown and cleanup functions used by the pushed test fixture, which delete any leftover objects such as barriers or semaphores and prevent resource leaking between the tests. If the leakage occurs, it can fail the tests since, for example, a barrier that should not exist is still present from the previous test, and the task successfully interacts with it, where normally it would fail.

The test suite is complete once all of the above files are present, including the generated test cases. However, the test suite still needs to be configured to be able to be compiled and run.

4.4 RTEMS Configuration & Test Deployment

To run the tests, the Board Support Package on which the test will be running needs to be built. In RTEMS version 6 'waf' is used for that purpose. Only a `config.ini` file needs to be defined inside

```
$HOME/{rtems_directory_prefix}/src/rtems
```

directory in order to use it. The `{rtems_directory_prefix}` represents a user chosen installation directory of RTEMS. Below is an example of contents of `config.ini` file:

```
[sparc/gr712rc]
RTEMS_SMP = True
BUILD_TESTS = True
```

This example is for 'gr712rc' BSP for SPARC architecture, as indicated in the first line of the configuration file. However, this file would look the same for 'leon3' or 'gr740' BSPs, with the only difference being the BSP name in the first line of configuration.

The final step in creating a working test suite for RTEMS would be configuring the application, adding it to the compilation specification and running the executable.

First of all, the C generated test files and test-case and test runner files would need to be included into the RTEMS source files. Therefore all of the files with `tr` and `tc` prefixed

need to be moved under

```
$HOME/{rtems_directory_prefix}/src/rtems/testsuites/validation
```

The tests are part of the effort to formally validate the system; thus, they should go under the 'validation' directory. However, for the execution purpose, any directory with tests would work.

Additionally, a test-setup file is also required in this directory. The file contains the configuration of the RTEMS ecosystem, in which the tests will be executed, such as scheduler configuration, limits on object creations as well as defining clock lengths. Normally, most test applications only have their test name inside and are simply importing `ts-default.h` configuration, as it fits most use cases. However, for the Barrier Manager, a separate `ts-model-barrier-mgr-0.c` file was created, which copies most of the contents from the default test-setup header file, with the exception of having `CONFIGURE_MAXIMUM_BARRIERS` option set to 1. That allows testing of an edge case where too many barriers are created by using only two barriers. It also allows better control over barrier object resources within the test as there is only a single barrier to take care of.

For the tests to get included into the compilation process when building BSP, they need to be added to the build specification. Build specifications are written in *YAML* and specify which files sources should be included into the executable when compiled, as well as what type of dependency this is. In this case, this is a build dependency, as it needs to be created every time we build the BSP.

The specification should be created under following directory:

```
$HOME/{rtems_directory_prefix}/src/rtems/spec/build/testsuites/validation
```

As with test files 'validation' directory is the semantically correct choice. However, any directory with test specifications would work.

All files with `tr`, `tc` and `ts` prefixes need to be included into specification under `source`. After the `.yml` specification is made, it must be linked with the main specification file. Since the tests are under the 'validation' group category, the new specification needs to be added to the corresponding `grp.yml` specification file in that directory. The following lines should be added to the file under `links` section:

```yaml
- role: build-dependency
  uid: model-barrier-mgr-0
```

The last steps left are compiling and running the tests. To compile the tests, the following commands need to be executed in the main RTEMS source directory:

```
$ ./waf configure
```

and then

```
$ ./waf install
```

That creates the executable for the generated tests. SIS SPARC simulator can then be used to execute them. Simulator is located in the RTEMS Kernel. It can be executed using:

```
$ ./sparc-rtems6-sis -leon3 -r s -m 4
$HOME/{rtems_directory_prefix}/src/rtems/build/sparc/gr712rc \
/testsuites/validation/ts-model-barrier-mgr-0.exe
```

The `-m` option defines the number of processors in the simulation. The directory path prefix of the executable file depends on the BSP used, and the suffix is consistent with paths used for specification and tests files.

If the tests execute successfully, there will be a line:

```
*** END OF TEST BarrierModel0 ***
```

displayed at the end of the execution log. Otherwise, if something fails during the execution, the line will not be present, or the test will crash or deadlock in the middle of the execution. Any failures are indicated by a log line starting with `F:`.

Instead of manually executing all of the commands, it is possible to use `testbuilder.py` script for that purpose:

1. `testbuilder.py copy model-barrier-mgr` - will copy all the relevant test files from the working directory into respective RTEMS directories and modify the *YAML* specification file accordingly.
2. `testbuilder.py compile` - will compile and build the tests using 'waf'.
3. `testbuilder.py run` - will run the tests using the SIS simulation tool.

All directory paths and filenames can be altered in the `testbuilder.yml` file.

Figure 4.2 graphically presents the general overview of the entire test suite generation process described in this chapter and showcases the role of each file in it.

```
(...)
barrier_create: |
  T_log(T_NORMAL, "Calling BarrierCreate(%d,%d,%d,%d)", {0}, {1}, {2}, {3} );
  rtems_id bid;
  initialise_id(&bid);
  rtems_status_code rc;
  rtems_attribute attribs;
  attribs = mergeattribs({1});
  rc = rtems_barrier_create({0}, attribs, {2}, {3} ? &bid : NULL);
  T_log(T_NORMAL, "Returned 0x%x from Create", rc );

barrier_ident: |
  T_log(T_NORMAL, "Calling BarrierIdent(%d,%d)", {0}, {1} );
  rtems_id bid;
  initialise_id(&bid);
  rtems_status_code rc;
  rc = rtems_barrier_ident({0} , {1} ? &bid : NULL);
  T_log(T_NORMAL, "Returned 0x%x from Ident", rc );

barrier_wait: |
  rtems_interval timeout = {1};
  T_log(T_NORMAL, "Calling BarrierWait(%d,%d)", bid, timeout );
  rc = rtems_barrier_wait(bid, timeout);
  T_log(T_NORMAL, "Returned 0x%x from Wait", rc );

barrier_delete: |
  T_log(T_NORMAL, "Calling BarrierDelete(%d)", bid);
  rc = rtems_barrier_delete(bid);
  T_log(T_NORMAL, "Returned 0x%x from Delete", rc );

barrier_release: |
  T_log(T_NORMAL, "Calling BarrierRelease(%d,%d)", bid, {1});
  uint32_t released;
  rc = rtems_barrier_release(bid, {1} ? &released : NULL);
  T_log(T_NORMAL, "Returned 0x%x from Release", rc );
(...)
```

Figure 4.1: Barrier Manager directive mappings inside `model-barrier-mgr-rfn.yml` dictionary file.

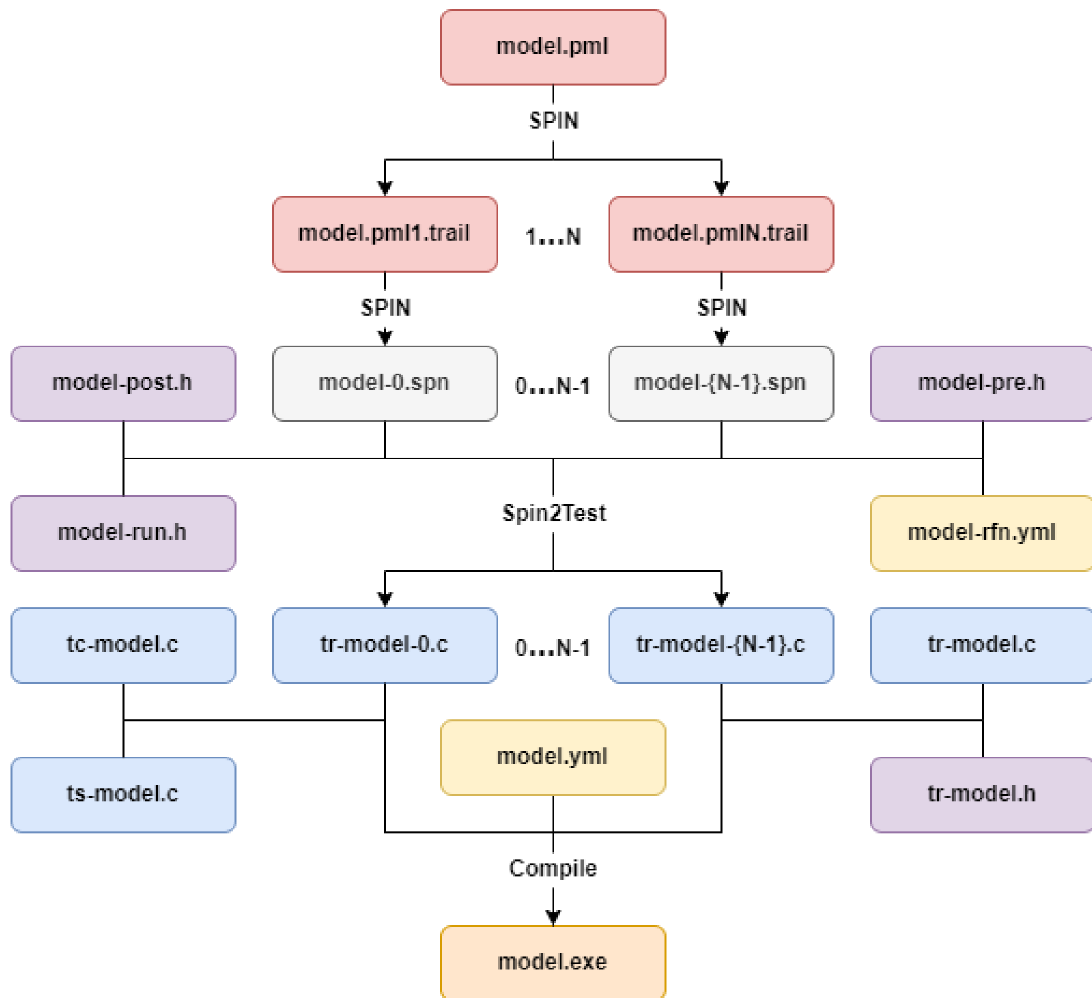

Figure 4.2: Overview of the SPIN/Promela test suite framework and files involved.

5 Evaluation

5.1 Model Validation

The developed model is a faithful representation of the real world system described in the documentation, but it still has to be validated to ensure it holds. To do that, the `` `assert(false)` `` at the end of the Promela model has to be temporarily removed, and SPIN has to be executed in the exhaustive verifier mode. That requires the following command:

```
$ spin -a model-barrier-mgr.pml
```

which would create verifier C code that can then be executed with:

```
$ cc pan.c
$ ./a.out
```

If the line in the log with the error count equals 0, then the model holds. E.g:

```
State-vector 448 byte, depth reached 752, errors: 0
```

Through this verification, we know that the model holds. However, there is no real way of knowing that it represents the actual RTEMS system. For this, the model has to be cross-checked by using generated tests. If tests fail, there is a chance that it is not a system bug, but the tests are incorrect because the model reflected the real system incorrectly or there is a discrepancy in the documentation. In this case, the test and model have to be manually analysed to find the problem. However, it is no different from manual test development and considering how crucial correct system behaviour is in RTEMS, any opportunity to detect system bugs is invaluable. Keeping in mind that generating tests from a model is much more efficient, it still brings significant value over manual test suite development.

5.2 Test Code Coverage

Test code coverage can be a good metric for establishing the efficiency of the tests in question. Especially, since as mentioned in **Section 1.1**, full code coverage is a requirement in RTEMS.

In order to evaluate the effectiveness of generated tests in terms of code coverage, the Qualification Data Pack (QDP (18)) was utilised. The reason for that is because at the moment of performing evaluation for this project, the RTEMS coverage report generation tool was not working, and it was not clear if it would be in the near future. Therefore the QDP qualification process was adapted instead to generate the reports. While all the tools were available, a few configuration steps had to be done to make it work for newly generated tests.

By default, the QDP is configured to work on target hardware. To make it work on the SIS simulation tool, the configuration file `` `qual-tool/config-variants/sparc-gr712rc-smp-user-qual.yml` `` has to be changed to contain the configuration presented in **Figure 5.1**.

The qualification package offers two types of RTEMS sources. One is the full RTEMS build that is similar to the one from the master branch of RTEMS git(7). Another one is the package that contains only pre-qualified interfaces of RTEMS scoped for space profiling, as described in (30). Since barrier manager is one of the pre-qualified interfaces, it is possible to use the scoped down version of RTEMS to speed up coverage report generation for our tests. Additionally, that means that no further configuration changes are needed; otherwise, all the relevant configuration scripts would need to be changed to link to the full RTEMS build. Alternatively, it is also possible to pre-build the test executable files and include them into the coverage report generation step.

To include the new test sources into the build, the steps from **Section 4.4** have to be followed, with the exception that the BSP build will now be handled by QDP tools. The only steps needed are copying the sources into the RTEMS source directory and adjusting configuration files. Unlike in regular RTEMS build, sources need to be copied into `` `src/rtems-qual-only` `` directory since the evaluation is performed on the pre-qualified package only.

To perform report generation, one needs to be in `` `qual-only` `` directory and follow execution commands laid out in Section 11.3 'Guidance for RTEMS Qualification in User's Environment' found in documentation supporting the QDP (30). The commands are the following:

```
$ make env
```

```
$ . env/bin/activate
$ ./qdp_config.py config-variants/sparc-gr712rc-{mode}-user-qual.yml
$ ./qdp_build.py --log-level=DEBUG \
  build-sparc-gr712rc-{mode}-user-qual \ 2>&1 | tee log.txt
```

The `{mode}` is replaced with either `smp` or `uni`, depending on the package used.

To evaluate only the tests generated by our framework, all of the other tests were unlinked in the specification files. The reports were generated for GR712RC SMP and UNI configurations and results are presented in the **Table 5.1** and **Table 5.2** respectively.

Table 5.1: Test coverage for GR712RC SMP using Promela/SPIN framework generated barrier manager tests.

Filename	Code Covered	Branches
`cpukit/include/rtems/rtems/barrierimpl.h`	100%	0
`cpukit/include/rtems/score/corebarrierimpl.h`	100%	0
`cpukit/rtems/src/barriercreate.c`	100%	10
`cpukit/rtems/src/barrierdelete.c`	100%	2
`cpukit/rtems/src/barrierident.c`	100%	0
`cpukit/rtems/src/barrierrelease.c`	100%	4
`cpukit/rtems/src/barrierwait.c`	100%	2
`cpukit/score/src/corebarrier.c`	100%	0
`cpukit/score/src/corebarrierwait.c`	100%	2

Table 5.2: Test coverage for GR712RC UNI using Promela/SPIN framework generated barrier manager tests.

Filename	Code Covered	Branches
`cpukit/include/rtems/rtems/barrierimpl.h`	100%	0
`cpukit/include/rtems/score/corebarrierimpl.h`	100%	0
`cpukit/rtems/src/barriercreate.c`	100%	10
`cpukit/rtems/src/barrierdelete.c`	100%	2
`cpukit/rtems/src/barrierident.c`	100%	0
`cpukit/rtems/src/barrierrelease.c`	100%	4
`cpukit/rtems/src/barrierwait.c`	100%	2
`cpukit/score/src/corebarrier.c`	100%	0
`cpukit/score/src/corebarrierwait.c`	100%	2

Only the files relevant to the barrier manager scope were included in the table. However, coverage checks are also done for any additional files from the core RTEMS system used to keep the system running and execute the directives involved. Nonetheless, it is still essential to take a general look into them.

By looking solely into the barrier manager related files, it is clear that the generated tests achieved the required full code coverage in both uniprocessor and SMP configurations.

On the contrary, the code coverage of the system files varied between SMP and uniprocessor configurations. SMP configuration covered more of the SMP supporting files, while uniprocessor configuration covered more of the standard implementation code. Coverage of the system files varied and often was not 100%. However, these files are out of the scope of barrier manager testing, and therefore there is no requirement of reaching full coverage on them.

5.3 Comparison with existing test suites

To fully understand how good the test coverage evaluated in **Section 5.2** is, it is compared with the code coverage of the existing test suite for the RTEMS barrier manager.

For that, QDP is utilised again. Since it is a fresh release, some of the test suites that it contains are more up-to-date than the ones at the master branch of RTEMS, as they are still in the process of being integrated at the point when this work was written. Moreover, we are only interested in *space profiling* scope only, which is precisely covered by *qual-only* pre-qualified version of RTEMS within the QDP. This scope also means that only the official test suite is compared, and any third-party suites are out of this project's scope.

Code coverage reports were generated for the existing tests that cover the RTEMS barrier manager to perform the comparison. The technique used is the same as described in **Section 5.2**.

The reports were generated for GR712RC SMP and UNI configurations and results are presented in the **Table 5.3** and **Table 5.4** respectively.

Table 5.3: Test coverage for GR712RC SMP when using existing barrier manager tests.

Filename	Code Covered	Branches
`cpukit/include/rtems/rtems/barrierimpl.h`	100%	0
`cpukit/include/rtems/score/corebarrierimpl.h`	100%	0
`cpukit/rtems/src/barriercreate.c`	100%	10
`cpukit/rtems/src/barrierdelete.c`	100%	2
`cpukit/rtems/src/barrierident.c`	100%	0
`cpukit/rtems/src/barrierrelease.c`	100%	4
`cpukit/rtems/src/barrierwait.c`	100%	2
`cpukit/score/src/corebarrier.c`	100%	0
`cpukit/score/src/corebarrierwait.c`	100%	2

Table 5.4: Test coverage for GR712RC UNI when using existing barrier manager tests.

Filename	Code Covered	Branches
cpukit/include/rtems/rtems/barrierimpl.h	100%	0
cpukit/include/rtems/score/corebarrierimpl.h	100%	0
cpukit/rtems/src/barriercreate.c	100%	10
cpukit/rtems/src/barrierdelete.c	100%	2
cpukit/rtems/src/barrierident.c	100%	0
cpukit/rtems/src/barrierrelease.c	100%	4
cpukit/rtems/src/barrierwait.c	100%	2
cpukit/score/src/corebarrier.c	100%	0
cpukit/score/src/corebarrierwait.c	100%	2

As expected, the existing test suite yields the same results since it is required to have full coverage, and it is impossible to go above that.

In terms of covered system files, code coverage can vary in some places precisely because of the different objects used for synchronisation within tests. However, on average, newly generated tests had slightly better coverage than the existing ones. The difference, however, is negligible, and since system files are out of the scope of barrier manager, it can be said that newly generated tests are as good as existing ones.

It is important to note that there is a barrier implementation called `smpbarrierwait.c`. This barrier is a low-level synchronisation tool made for SMP systems and is outside of the scope of the barrier manager. Therefore it is not covered by either of the barrier manager test suites.

Overall the results indicate that the test suite generated using Promela/SPIN framework is as comprehensive as the existing RTEMS test suite.

5.4 Extensibility

Even though the main objective was to model and generate tests for barrier managers, the secondary objective was to try and make the framework as extendable to other managers of RTEMS as possible and leave the room for possible extensions to the existing model.

To evaluate how well the model can be extended, some experimentation involving other RTEMS managers was performed. It was measured how much the current framework has to be modified in order to fit other parts of the system.

The current design does not allow for immediate adoption of the model. That is because, within the Promela model, each directive is modelled separately. On the other hand, this modularity allows to simply re-write the modelled directives to the ones of the manager

under test. The execution flow, where a uniform set of options controlling process behaviour is passed to the process when it is created, allows the system to be extended further. With that, scenario control is reduced to simply modifying some of the passed options inside the `if` option sequences. Testing new or different behaviour can be done by merely adding or removing options. This approach is very efficient since most RTEMS managers, including the barrier manager, focus on task orchestration. Options then make it easy to synchronise these tasks and decide which one is responsible for what. However, this can be extended even to other managers, such as cache, partition or I/O managers. That is because every directive always has to be called by a process, and processes within the model do not have a specific pre-defined role; it is defined by the passed option set. Therefore the number of tasks can be easily scaled up or down depending on the model's needs. The task execution itself only needs to be slightly modified to use the options set that fits the manager in question. That means that the general structure of the model can be kept the same across all of the managers modelled. The only work remains in the modelling of the directive behaviour itself.

Other parts of the framework also would require a limited amount of modifications to fit other managers because all of the unnecessary wrappers were removed, making test calling straightforward. Wrapped directive call functions are no longer passed through test case and runner files, making the calls easily adjustable.

The only change necessary within the template files is the adjustment for the number of tasks running within the model and objects that need to be created during the setup phase.

Most of the no longer needed configuration was removed from template files, and code blocks repeated across different test cases were moved intro helper methods within the test-runner files, making it cleaner and more adjustable. When adjusting for testing of a new manager, test-runner files would need to support the creation of the necessary new objects. Additionally, test-runners contain teardown methods and execution `context`, contents of which would have to be adjusted according to the manager and the directives being tested. In terms of test scenario calls, the only things that would need to be changed are the function names, e.g. from:

```c
void RtemsModelBarrierMgr_Run0(void);
```

to, as an example for Signal Manager, this:

```c
void RtemsModelSignalMgr_Run0(void);
```

The test-case file would always have to be adjusted, as it contains calls to the generated

test-runner scenario files. Directive function calls are no longer wrapped there; there are only test case calls left. Therefore, the whole file would have to be overhauled with name changes when changing the manager under test. However, due to repeated structure within this file, it is possible to potentially automate this process or even 'find and replace' all the names with another one.

The only file that would require non-trivial changes is the refinement dictionary file. During this project, the file was cleaned, and the number of keywords used was reduced to make the file cleaner and the replacement process more straightforward. While some of the keywords relating to status code comparison and task and scheduler checks can be left in place, the replacements for the directive calls will have to be modified according to the needs. It is also necessary to keep in mind to plan out the structure of the directive calls and variables used between them, as some of the directives tested further can rely on the results of the previous calls. Due to format constraints, it is impossible to provide a direct assignment in the refinement file. So any necessary assignments, e.g. `bid = 0;`, would need to use test-runner helper methods instead.

Lastly, there is the configuration `ts-` test-setup file. Unlike other tests, the separate test-setup file was used. Technically it is a copy of `ts-default.c` used by other tests, but with a different maximum barrier allowance. That allows for control over how many barriers can be created within the application without affecting other test suites. The solution is not entirely scalable and is mostly in place as it is impossible to configure the number of barriers allowed dynamically within the test application. However, it is acceptable, considering there is currently no real reason to adjust that limit on the go, as it is used solely to test a single edge case where there are too many barriers created. Testing this edge case with a different barrier limit will not cover any additional code, just like the existence of multiple barriers does not affect the behaviour of each of them in any way. Nonetheless, a possibly more elegant solution in the future could be to use `ts-default.c` instead of copying it and find a way to adjust the barrier limit only for barrier manager tests.

```yaml
build-directory: build-sparc-gr712rc-smp-user-qual
post-process-items:
- uid: /dirs/djf-svr-deploy/dir
path: /directory
action: set
value: ${/variant:/deployment-directory}/user_doc/djf/svr
- uid: /dirs/ddf-sdd-deploy/dir
  path: /directory
  action: set
  value: ${/variant:/deployment-directory}/user_doc/ddf/sdd
- uid: /steps/build-djf-svr
  path: /config-file
  action: set
  value: rtems/djf/svr/config_user.yml
- uid: /package-build
  path: /links
  action: set
  value:
  - role: build-step
    uid: steps/build-bsp-qual-only
  - role: build-step
    uid: steps/build-bsp-qual-only-coverage
  - role: build-step
    uid: steps/run-local-target-qual-only
  - role: build-step
    uid: steps/run-local-target-qual-only-coverage
  - role: build-step
    uid: steps/build-ddf-sdd
  - role: build-step
    uid: steps/build-djf-svr
spec-paths:
- spec-spec
- spec-glossary
- config
```

Figure 5.1: Contents of sparc-gr712rc-smp-user-qual.yml, needed to execute tests on SIS simulation tool .

6 Conclusion

This work presented the framework for generating C tests for RTEMS through Promela models and SPIN verifier. This framework was then successfully applied to the RTEMS barrier manager to create a comprehensive test suite. This test suite will be incorporated into the central RTEMS repository and used as part of a community effort to verify the system.

The evaluation showed that the effectiveness of these tests is on par with the ones that were manually developed in both SMP and uni-processor configurations. However, the fact that the development process of generated tests was much shorter means that the overall speed of the development pipeline can be significantly improved when the presented framework is used.

The benefit of using the developed framework is also not limited to the generation of a test suite. SPIN/Promela are formal verification tools. Therefore by using them to model the system, this project also advanced the ongoing efforts to formally validate it. While it is still far till the whole of RTEMS is formally verified, it is an important step in that direction.

While applying the framework to the barrier manager, it was also significantly improved over the previous version used for modelling the event manager. All of the redundant code was removed to make the codebase more readable. In addition to reducing the code, the functionality was modularised, and its execution was streamlined. That allows the whole system to be easily extended or re-purposed to model other RTEMS managers. This improvement is an essential step towards generalising the modelling framework to be used in the whole of RTEMS.

The limitation of the developed approach is the fact that modelling is heavily based on the system's documentation. It assumes that documented behaviour is the ground truth, while in reality, it is prone to human errors. This limitation, to a degree, is mitigated by the fact that any potential test failures coming from incorrect modelling of the system are then analysed and consulted with the RTEMS community, which allows to establish whether it is a system or a model bug.

Additionally, current refinement scripts do not allow for a large degree of freedom. While this may simplify the task of understanding the refinement process and make it more readable, it can become a limiting factor when there is a need to model a particular behaviour.

Overall the work done in this dissertation successfully completed the research objectives. The Promela/SPIN framework for RTEMS test generation was simplified and improved, and a complete model with a thorough test suite for RTEMS barrier manager was created. Furthermore, the project went through the full path of the space software qualification process by generating coverage reports using QDP to confirm full test coverage of the generated tests.

These advancements will be helpful for the RTEMS Community and their efforts to formally verify the system, especially considering some of the most important RTEMS use cases, such as controlling aircraft or spacecraft.

6.1 Future Work

There is significant potential and multiple directions for possible future work.

The most straightforward direction would be to extend the framework to other RTEMS managers and create test suites for them. There is a total of 19 managers in RTEMS, as well as some additional APIs that can be modelled and validated through tests. While some of them are already modelled, including now barrier manager, there is still work to be done on that front. This work can also help further generalise the whole framework across all RTEMS managers and APIs.

Another possibility is improving the scope of the actual Promela model and test generation framework. Currently, the priorities and the preemption mechanisms are not modelled because they were considered out of the scope of the barrier manager. However, it might be interesting to see if preemption mechanisms and priorities work correctly in general, including barrier manager directives. Additionally, there are still some gaps in test code generation within the refinement file. While the keywords are present, there is no check performed on the current state of the task, nor there is a check on the value of some of the products of the directive calls, such as IDs. It is impossible to know these details from inside the model. Exact IDs are decided on and assigned by the system during the execution, while a task cannot check its own status when it is suspended. Therefore, it might be an interesting direction of research, which can decide if performing these checks is feasible or if they are obsolete. Test generation scripts can also be improved. Currently, they have limited functionality and only work with a very specific format, but they can be improved to be more versatile, readable, and user-friendly.

In terms of potential research on a more significant scope, it might be interesting to

automate the Promela modelling fully or even partially. One approach, as described in Formal Verification documentation in (18), could be creating `printf` annotations within the model based on some of the Promela keywords, such as `assert()`. This way developer can focus on modelling the system, while the framework will handle the decision about the critical system states that need to be captured to create a test.

However, one of the most currently discussed topics is to migrate all of the current work in this field onto ARM architecture. SPARC is a relatively old architecture and has barely advanced in recent years. Considering that the main SPARC team from Sun Microsystems got disassembled by Oracle in 2017, its support has been dwindling ever since. It has also become increasingly hard to keep tool support, such as *GCC*, intact for the SPARC platform as well. At the same time, ARM keeps on gaining popularity, especially in the embedded devices field. Therefore researching into adapting the Promela/SPIN test generation framework and related work to support ARM architecture can be very interesting and beneficial for the RTEMS community.

Finally, an interesting direction can be going outside of RTEMS. It might be possible to port the framework into other RTOS-es like FreeRTOS. Promela models run independently of the modelled system and platform while adopting the model presented in this work to model other systems is relatively straightforward. However, the way tests are generated and configured might need some additional work and changes, depending on the test framework and tools used within the target systems.

Bibliography

[1] RTEMS Project and Contributors. Rtems classic api guide, 2022. URL https://docs.rtems.org/branches/master/c-user/index.html. Accessed: 2022-01-27.

[2] The RTEMS Project. Rtems real time operating system (rtos), 2021. URL https://www.rtems.org/. Accessed: 2021-12-18.

[3] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 1–30, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-35746-6. doi: 10.1007/978-3-642-35746-6_1. URL https://doi.org/10.1007/978-3-642-35746-6_1.

[4] Sebastiano Catellani, Luca Bonato, Sebastian Huber, and Enrico Mezzetti. Challenges in the implementation of mrsp. pages 179–195, 06 2015. ISBN 978-3-319-19583-4. doi: 10.1007/978-3-319-19584-1_12.

[5] A. Burns and A.J. Wellings. A schedulability compatible multiprocessor resource sharing protocol – mrsp. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 282–291, 2013. doi: 10.1109/ECRTS.2013.37.

[6] Fernand Quartier and Paul Parisis. Development environment for future leon multi-core. Technical report, Spacebel s.a., Embedded brains GmbH, 2015.

[7] The RTEMS Project. Rtems source, 2022. URL https://git.rtems.org/. Accessed: 2022-04-02.

[8] Gedare Bloom and Joel Sherrill. Scheduling and thread management with rtems. *SIGBED Rev.*, 11(1):20–25, feb 2014. doi: 10.1145/2597457.2597459. URL https://doi.org/10.1145/2597457.2597459.

[9] RTEMS Project and Contributors. Rtems posix api guide, 2022. URL https://docs.rtems.org/branches/master/posix-users/preface.html. Accessed: 2022-01-23.

[10] RTEMS Project and Contributors. Rtems source builder, 2022. URL https://docs.rtems.org/branches/master/user/rsb/index.html. Accessed: 2022-02-14.

[11] RTEMS Project and Contributors. Rtems cpu architecture supplement, 2022. URL https://docs.rtems.org/branches/master/cpu-supplement/index.html. Accessed: 2022-03-21.

[12] RTEMS Project and Contributors. Rtems user manual - board support packages, 2022. URL https://docs.rtems.org/branches/master/user/bsps/index.html. Accessed: 2022-03-02.

[13] Alexander Krutwig and Sebastian Huber. Rtems smp final report. Technical report, Embedded brains GmbH, 2017.

[14] Rodrigo Perez. What is computer system validation and how do you do it?, 2018. URL https://validationcenter.com/computer-system-validation-how-to/. Accessed: 2022-03-23.

[15] RTEMS Project and Contributors. Rtems qualification project, 2022. URL https://qualification.rtems.org/. Accessed: 2022-03-11.

[16] RTEMS Project and Contributors. Introduction to pre-qualification, 2022. URL https://docs.rtems.org/branches/master/eng/prequalification.html. Accessed: 2022-03-11.

[17] Andrew Butterfield and Frédéric Tuong. Fv-201 formal verification artefacts (architecture, models, assumptions, traceability, supporting tests), 2022. URL https://rtems-qual.io.esa.int/opt/rtems-6-sparc-gr712rc-smp-3/doc/fm/fva/FV2-201.pdf. Accessed: 2022-03-18.

[18] European Space Agency, Edisoft, Trinity College Dublin, Embedded Brains, Jena-Optronik, and CISTER. Esa rtems smp qualification data package, 2022. URL https://rtems-qual.io.esa.int/. Accessed: 2022-03-02.

[19] SPARC International Inc. The sparc architecture manual - version 8, 1991. URL https://www.gaisler.com/doc/sparcv8.pdf. Accessed: 2022-03-19.

[20] Peter Magnusson. Understanding stacks and registers in the sparc architecture(s), 1997. URL http://icps.u-strasbg.fr/people/loechner/public_html/enseignement/SPARC/sparcstack.html. Accessed: 2022-03-23.

[21] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmus^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126, 2006. doi: 10.1109/RTSS.2006.27.

[22] FreeRTOS Contributors and Amazon Web Services. Freertos™ real-time operating system for microcontrollers, 2022. URL https://www.freertos.org/index.html. Accessed: 2022-03-25.

[23] Andreas Ulrich, El-Hachemi Alikacem, Hesham H. Hallal, and Sergiy Boroday. From scenarios to test implementations via promela. In Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado, editors, *Testing Software and Systems*, pages 236–249, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16573-3.

[24] RTEMS Project and Contributors. Rtems classic api guide - symmetric multiprocessing (smp), 2022. URL https://docs.rtems.org/branches/master/c-user/symmetric_multiprocessing_services.html. Accessed: 2022-02-22.

[25] RTEMS Project and Contributors. Rtems classic api guide - scheduling concepts, 2021. URL https://docs.rtems.org/branches/master/c-user/scheduling-concepts/index.html. Accessed: 2021-12-29.

[26] RTEMS Project and Contributors. Rtems classic api guide - multiprocessing manager, 2022. URL https://docs.rtems.org/branches/master/c-user/multiprocessing/index.html. Accessed: 2022-02-22.

[27] RTEMS Project and Contributors. Rtems classic api guide - barrier manager, 2022. URL https://docs.rtems.org/branches/master/c-user/barrier/background.html#background. Accessed: 2022-01-16.

[28] SPINROOT and Contributors. Promela reference - 'inline', 2010. URL http://spinroot.com/spin/Man/inline.html. Accessed: 2022-02-12.

[29] SPINROOT and Contributors. Promela reference - 'if', 2010. URL https://spinroot.com/spin/Man/if.html. Accessed: 2022-02-18.

[30] European Space Agency, Edisoft, Trinity College Dublin, Embedded Brains, Jena-Optronik, and CISTER. Rtems qualification data package software configuration file. 2022. URL https://rtems-qual.io.esa.int/public_release/rtems-6-sparc-gr712rc-smp-3-scf.pdf. Accessed: 2022-03-14.

A1 Appendix

A1.1 Refinement Dictionary File

Figure A1.1 presents the full contents of the Refinement Dictionary file written in *YAML*. In addition to having directive call replacements, it contains other, more utility-focused mappings between keywords found in model execution output and RTEMS test C code. These include initialization, semaphore operations, task and priority checks, as well as return code comparisons.

```
LANGUAGE: C

SEGNAMEPFX: TestSegment{} # segnumber
SEGARG: Context* ctx
SEGDECL: static void {}( {} )  # segnamepf{segnumber}, segarg,
SEGBEGIN: " {"
SEGEND: "}"

NAME: |
 /* Test Name is defined in the Test Case code (tc-model-barrier-mgr.c) */

semaphore_DCL: rtems_id {0}[{1}];

INIT: |
  initialise_semaphore( ctx, semaphore );

Runner: |
  checkTaskIs( ctx->runner_id );

Worker0: |
  checkTaskIs( ctx->worker0_id );

Worker1: |
  checkTaskIs( ctx->worker1_id );
```

```
25
26 SIGNAL: |
27   ReleaseSema( semaphore[{}] );
28
29 WAIT: |
30   ObtainSema( semaphore[{}] );
31
32 barrier_create: |
33   T_log(T_NORMAL, "Calling BarrierCreate(%d,%d,%d,%d)", {0}, {1}, {2}, {3} );
34   rtems_id bid;
35   initialise_id(&bid);
36   rtems_status_code rc;
37   rtems_attribute attribs;
38   attribs = mergeattribs({1});
39   rc = rtems_barrier_create({0}, attribs, {2}, {3} ? &bid : NULL);
40   T_log(T_NORMAL, "Returned 0x%x from Create", rc );
41
42 barrier_ident: |
43   T_log(T_NORMAL, "Calling BarrierIdent(%d,%d)", {0}, {1} );
44   rtems_id bid;
45   initialise_id(&bid);
46   rtems_status_code rc;
47   rc = rtems_barrier_ident({0} , {1} ? &bid : NULL);
48   T_log(T_NORMAL, "Returned 0x%x from Ident", rc );
49
50 barrier_wait: |
51   rtems_interval timeout = {1};
52   T_log(T_NORMAL, "Calling BarrierWait(%d,%d)", bid, timeout );
53   rc = rtems_barrier_wait(bid, timeout);
54   T_log(T_NORMAL, "Returned 0x%x from Wait", rc );
55
56 barrier_delete: |
57   T_log(T_NORMAL, "Calling BarrierDelete(%d)", bid);
58   rc = rtems_barrier_delete(bid);
59   T_log(T_NORMAL, "Returned 0x%x from Delete", rc );
60
61 barrier_release: |
62   T_log(T_NORMAL, "Calling BarrierRelease(%d,%d)", bid, {1});
63   uint32_t released;
64   rc = rtems_barrier_release(bid, {1} ? &released : NULL);
65   T_log(T_NORMAL, "Returned 0x%x from Release", rc );
66
67 rc:
```

```
68     T_rsc( rc, {0} );
69
70   released:
71     T_eq_u32( released, {0} );
72
73   created:
74     /* This is used later for cleanup */
75     ctx->barrier_id = bid;
76
77   Ready: |
78      /* We (Task {0}) must have been recently ready because we are running */
79
80   Zombie:
81     /* Code to check that Task {0} has terminated */
82
83   BarrierWait:
84     /* Code to check that Task {0} is waiting on the barrier */
85
86   TimeWait:
87     /* Code to check that Task {0} is waiting on a timer */
88
89   OtherWait:
90     /* Code to check that Task {0} is waiting (after pre-emption) */
91
92   LowPriority: |
93     SetSelfPriority( PRIO_LOW );
94     rtems_task_priority prio;
95     rtems_status_code sc;
96     sc = rtems_task_set_priority( RTEMS_SELF, RTEMS_CURRENT_PRIORITY, &prio );
97     T_rsc_success( sc );
98     T_eq_u32( prio, PRIO_LOW );
99
100  NormalPriority: |
101    SetSelfPriority( PRIO_NORMAL );
102    rtems_task_priority prio;
103    rtems_status_code sc;
104    sc = rtems_task_set_priority( RTEMS_SELF, RTEMS_CURRENT_PRIORITY, &prio );
105    T_rsc_success( sc );
106    T_eq_u32( prio, PRIO_NORMAL );
107
108  HighPriority: |
109    SetSelfPriority( PRIO_HIGH );
110    rtems_task_priority prio;
```

```
111    rtems_status_code sc;
112    sc = rtems_task_set_priority( RTEMS_SELF, RTEMS_CURRENT_PRIORITY, &prio );
113    T_rsc_success( sc );
114    T_eq_u32( prio, PRIO_HIGH );
115
116  StartLog: |
117    T_thread_switch_log *log;
118    log = T_thread_switch_record_4( &ctx->thread_switch_log );
119
120  CheckPreemption: |
121    log = &ctx->thread_switch_log;
122    T_eq_sz( log->header.recorded, 2 );
123    T_eq_u32( log->events[ 0 ].heir, ctx->runner_id );
124    T_eq_u32( log->events[ 1 ].heir, ctx->worker_id );
125
126  CheckNoPreemption: |
127    log = &ctx->thread_switch_log;
128    T_le_sz( log->header.recorded, 1 );
129    for ( size_t i = 0; i < log->header.recorded; ++i ) {
130      T_ne_u32( log->events[ i ].executing, ctx->worker_id );
131      T_eq_u32( log->events[ i ].heir, ctx->runner_id );
132    }
133
134  SetProcessor: |
135    #if defined(RTEMS_SMP)
136    T_eq_u32( rtems_scheduler_get_processor_maximum(), 4 );
137    uint32_t processor = {};
138    cpu_set_t cpuset;
139    CPU_ZERO(&cpuset);
140    CPU_SET(processor, &cpuset);
141    #endif
```

Figure A1.1: Full contents of `model-barrier-mgr-rfn.yml` dictionary file.

A1.2 Spin2Test Language Definition

As mentioned in **Section 4.2**, the logs produced by the Promela model have to fit exact patterns that are defined within the `spin2test` program. The format has to be of the following structure.

```
`@@@ <pid> <KEYWORD> <thing1> ... <thingN>`
```

where $N >= 0$.

Possible `<thing>`s that are relevant to Barrier Manager model are:

- `<pid>` - Promela Process Id of the process that generated the log.
- `<word>` - string of alphanumeric characters containing no space.
- `<name>` - Promela variable/structure/constant identifier.
- `<type>` - Promela type identifier.
- `<tid>` - task/thread/process ID alias.
- `<value>` - numerical value.
- `_` - unused argument.

Those are then combined with the `<KEYWORD>`s:

- `LOG <word0> ... <wordN>` - log output. Log outputs do not generate any code and are there for readability and debugging purposes.
- `INIT` - initialization block. This keyword generates a call or calls to initialize any helper objects needed during the test execution, such as semaphores.
- `DEF <name> <value>` - generates a `#define` macro with a given 'name' and 'value'.
- `DCLARRAY <type> <name> <value>` - generates an array with array 'type' being based on the linking keyword from dictionary and gives it given 'name' and size of 'value'.
- `TASK <name>` - generates a check that verifies that the currently running task is indeed the one with the given 'name' identifier.
- `SIGNAL <value>` - generates a call to release a semaphore with ID equal to 'value'.
- `WAIT <value>` - generates a call to obtain a semaphore with ID equal to 'value'.
- `STATE <tid> <name>` - generates a call to check if the task under 'tid' is indeed in the state 'name'.
- `SCALAR <name> <value>` - generates a code block to operate a scalar return 'value' from a function. The exact code block is matched by 'name' in the dictionary file. Usually, it checks whenever the 'value' is equal to what is expected. However, it can also be an assignment call to make sure it happens only in certain situations, such as the success of the directive call.

- `CALL <name> <value1> ... <valueN>` - generates a code block with a call to a function, where 'name' is a keyword in a dictionary and 'value's are the function parameters.

There are also a few other keywords. They, however, are not used within the Barrier Manager model and refinement process.

A1.3 Generated Test Segment

Figure A1.2 presents an example Test Segment generated within the barrier manager test file. This segment was generated using the dictionary refinement file. Since it is `TestSegment4`, it corresponds to the segment run by `Worker0` task.

A1.4 Complete Project Code

The complete project source code can be found under the following GitHub repository: https://github.com/EZCodes/SPIN-PromelaRTEMSTestGen.

Running the tests requires RTEMS installation, sources for which can be found at https://docs.rtems.org/branches/master/user/start/sources.html.

```
//  ===== TEST CODE SEGMENT 4 =====
static void TestSegment4( Context* ctx ) {
  T_log(T_NORMAL,"@@@ 4 TASK Worker0");
  checkTaskIs( ctx->worker0_id );

  T_log(T_NORMAL,"@@@ 4 CALL NormalPriority");
  SetSelfPriority( PRIO_NORMAL );
  rtems_task_priority prio;
  rtems_status_code sc;
  sc = rtems_task_set_priority( RTEMS_SELF, RTEMS_CURRENT_PRIORITY, &prio );
  T_rsc_success( sc );
  T_eq_u32( prio, PRIO_NORMAL );
  T_log(T_NORMAL,"@@@ 4 WAIT 1");
  ObtainSema( semaphore[1] );

  T_log(T_NORMAL,"@@@ 4 CALL barrier_ident 1 1");
  T_log(T_NORMAL, "Calling BarrierIdent(%d,%d)", 1, 1 );
  rtems_id bid;
  initialise_id(&bid);
  rtems_status_code rc;
  rc = rtems_barrier_ident(1 , 1 ? &bid : NULL);
  T_log(T_NORMAL, "Returned 0x%x from Ident", rc );

  T_log(T_NORMAL,"@@@ 4 SCALAR rc 3");
  T_rsc( rc, 3 );
  T_log(T_NORMAL,"@@@ 4 SIGNAL 2");
  ReleaseSema( semaphore[2] );

  T_log(T_NORMAL,"@@@ 4 CALL barrier_wait 1 0");
  rtems_interval timeout = 0;
  T_log(T_NORMAL, "Calling BarrierWait(%d,%d)", bid, timeout );
  rc = rtems_barrier_wait(bid, timeout);
  T_log(T_NORMAL, "Returned 0x%x from Wait", rc );

  T_log(T_NORMAL,"@@@ 4 SCALAR rc 4");
  T_rsc( rc, 4 );
  T_log(T_NORMAL,"@@@ 4 SIGNAL 1");
  ReleaseSema( semaphore[1] );
  T_log(T_NORMAL,"@@@ 4 STATE 2 Zombie");
  /* Code to check that Task 2 has terminated */
}
```

Figure A1.2: Example test segment for barrier manager generated with refinement dictionary file.