Applying Formal Verification to an Open-Source Real-Time Operating System

Andrew Butterfield[1(✉)] and Frédéric Tuong[2]

[1] Trinity College Dublin, Dublin, Ireland
butrfeld@tcd.ie
[2] Simon Fraser University, Burnaby, BC, Canada
ftuong@sfu.ca

Abstract. This paper describes work done using formal methods to verify parts of the RTEMS real-time operating system, as part of an activity sponsored by the European Space Agency to qualify multi-core processors for spaceflight. A variety of formalisms were investigated, keeping in mind the need to be a good fit with the RTEMS community in general. The technique that was deployed used Promela to model aspects of the operating system behavior, and the SPIN model-checker to do test generation. This involved developing Promela models, which are formal artifacts, and then developing a simple machine-readable observation language that made it easy to connect model behavior to the generation of C test code. The observation language was then refined to code using a dictionary mapping observable elements to test code snippets. Neither the observable language of the dictionary mapping are formal, so this paper also explores how these might be given UTP semantics, and linked together, in which the research of He Jifeng plays a key role. It finishes defining a future research agenda that uses this work with a real-world application to drive the research.

AQ1 AQ2 AQ3

Keywords: Unifying theories of programming · Model checking · Promela/SPIN · Test Generation · Real-Time Operating Systems · RTEMS

1 Introduction

The first author's first academic interaction with He Jifeng was his 2002 paper with Adnan Sherif on Circus Time [42]. This was a key starting point for early work on Slotted Circus [16]. One earliest memory of meeting He Jifeng in person was at UTP2008 which was hosted in Trinity College Dublin, at which he gave the keynote talk [29] (among others!). This paper explores how Unifying Theories of Programming (UTP [33]) might be used to give an overarching description of work we did using formal methods to verify parts of RTEMS [11]. In particular, we look at how the work of He Jifeng provided, and continues to provide, key material towards fulfilling this aim.

Supported by ESA Contract No. 4000125572/18NL/GLC/as, and assistance from Lero and the RTEMS community.

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 1–19, 2023.
https://doi.org/10.1007/978-3-031-40436-8_13

This paper describes work we did to introduce formal methods into an activity sponsored by the European Space Agency (ESA) to qualify a new version of the RTEMS[1] open-source real-time operating system [5], tailored for multi-core processors. It also looks at how Unifying Theories of Programming (UTP) [33] could be used to fill the formal semantic gaps that arose as a result of our approach.

The ESA activity, *Qualification of RTEMS Symmetric Multiprocessing (SMP)*, was led by Thales Edisoft, involving also Embedded Brains, the CISTER Research Centre at U. Porto, Jena Optronik, and ourselves. A single-core version of RTEMS had previously been qualified by Edisoft, and this new activity looked at upgrading that to cover multi-core, and to also provide tooling to ease the cost of the testing, reporting, and document generation involved. Edisoft, Embedded Brains and CISTER worked on these aspects, and they had to meet the standards expected, as defined by the relevant standards for software assurance [24]. All the artifacts that they would produce were to be made available, open-source, in the RTEMS git repositories [2], in a manner that adhered to the RTEMS community guidelines [46, §1.3]. Jena Optronik used a proprietary real-world application of theirs to assess the methodology developed by the other partners.

Our role was to explore how best to use formal methods to support qualification, in a way that would also fit with RTEMS community guidelines. After an initial survey and review of suitable formalisms, we elected to use the model-checker Promela/SPIN[35] to do test generation. This focussed on parts of the RTEMS API that dealt with task synchronization facilities, including signalling events, synchronization barrier, message-passing, and semaphores. The emphasis was on the functional correctness of the relevant API calls, when invoked by concurrent tasks. Also, the success criteria for the qualification effort as a whole was not tied to our efforts, in that the other partners had to meet the relevant standards, including those regarding test coverage, independently of what we achieved.

In the Background section (Sect. 2) we give an overview of the RTEMS qualification project, and introduce the Promela language and SPIN model-checker. In the Formal Models section (Sect. 3) we give more details of the formal approach and give an overview of one the models we constructed. In the Refinement section (Sect. 4) we explain how we map the counter-examples generated from SPIN into RTEMS C test code. In all of the above we explore how UTP might be applied to formally connect all the pieces. We then discuss Related Work (Sect. 5) and finish with Conclusions and future work (Sect. 6).

2 Background

2.1 RTEMS

RTEMS[5,11] is a real-time operating system aimed mainly at embedded systems. It is open-source and freely available [2], mainly under a BSD-2 license.

[1] Real-Time Executive for Multiprocessor Systems.

It is an operating system of choice for ESA, who have funded a number of past initiatives [3,4] to bring RTEMS up to the quality standards they require These are European-based standards for critical space software [23,24], involving a lot of testing, traceability and documentation, but no formal methods. The *qualification* so obtained covered versions of RTEMS aimed at single-core processors.

However, the widespread availability of multi-core processors, including those that are space-hardened, has led to increased demand for their approval for space missions. In addition, two recent new multi-core adaptations of existing scheduler algorithms were implemented for RTEMS. These were the $O(m)$ Independence-Preserving Protocol (OMIP) [12], and the Multiprocessor Resource Sharing Protocol (MrsP) [13]. Both were merged and added to RTEMS [18] and have been updated and enhanced several times since [28]. The move to multi-core support caused a large increase in the complexity of the code, especially as far as scheduling was concerned. This ranges from the implementation of synchronisation primitives such as semaphores, barriers, events, messaging, and a key component underlying these: *thread queues*.

This led to the establishment in 2018 of an activity called "Qualification of RTEMS Symmetric Multiprocessing (SMP)" to perform a pre-qualification[2] of two- and four-core processors for spaceflight. This activity had a number of key goals beyond just producing a pre-qualified version of multi-core RTEMS. These included:

- Developing tools to automate the generation and reporting of the evidence needed to demonstrate that the qualification standards had been achieved.
- Providing all the tooling, and code improvements in a form that could be fed back into the RTEMS open-source repositories.

The second goal is key. The purpose of this work went beyond just the needs ESA had for qualifying software, but also included the desire by the RTEMS community to gain expertise themselves in the ability to perform safety critical certification and qualification. The idea is that the techniques could be reused by RTEMS users in other safety critical application areas.

This second goal meant that all qualification materials should fit in with RTEMS community guidelines [46, §1.3],[45, §2]. One key consequence of this is to consider the needs of all users, from hobbyist to safety-critical system developers. This means that large complex software entities with many dependencies should be avoided where possible. Another principle is that licensing must have a BSD-2 flavour, rather than something like GPL—this is because it is expected that companies will link RTEMS with proprietary applications.

Formal Methods for ESA for RTEMS. The investigation into the use of formal methods had three phases: an initial exploration of available and suitable formal techniques; apply the chosen techniques to selected parts of

[2] ESA uses qualification to refer to an entire mission. RTEMS is a sub-component of a mission, so such partial treatments are called pre-qualifications.

```
stmt ::= skip                    // do nothing
       |  v = e                  // assignment
       |  e                      // expression-statement
       |  assert(e)              // assertion
       |  run procName(e,..,e)   // start a process
```

Fig. 1. Simplified syntax for atomic Promela statements.

```
stmt +:= stmt ; stmt    // sequencing
       |  stmt -> stmt  // sequencing (alt.)
       |  if            // conditional
          :: stmts1
             ...
          :: stmtsN
          fi
       |  do            // iteration
          :: stmts1
             ...
          :: stmtsN
          od
       | atomic{stmts}
```

Fig. 2. Simplified syntax for composite Promela statements. Here `stmts` denotes one or more sequenced `stmt`s.

RTEMS; and producing a final report. The initial investigation explored a range of techniques, including, among others, Isabelle/HOL [39], Frama-C [36], and Promela/SPIN[35]. The outcome was a decision to focus on using Promela/SPIN to perform test generation, as this technology is a good fit with the RTEMS guidelines, and the 2009 survey paper by Hierons et al. [32] makes a very good case for using model-checkers like SPIN in this way.

2.2 Promela/SPIN

Promela/SPIN is easy to install (spinroot.com), requiring little more than a C compiler along with the `lex` and `yacc` utilities. This made it a good fit for RTEMS users in terms of its size and installation. Promela is the modelling language, while SPIN is the model-checker. The Promela language, based loosely on C, is imperative in character, with a notion of state defined by variables, and notation that allows concurrent process behavior to be defined. Behavior is defined by statements which can be atomic (Fig. 1), such as assignment, or composite (Fig. 2), like conditionals or iteration. Communication between processes can be via shared global variables, or using CSP-like channel-based message passing. The semantics is based on arbitrary interleaving of the sequence of atomic actions performed by each process. Each process has a "program counter" that identifies the next statement to be executed. The state of a Promela model is

defined by the values of all the variables and the program counters of the live processes.

The above description could be of a concurrent programming language, but Promela is for modelling, and so its semantics differs in crucial ways. The first key difference is the notion of "executability". Some language constructs are always ready to run, while others may only run in certain model states. The `skip`, assignment and assertions are always executable. While `assert(e)` is always executable, if `e` evaluates to false, then the model run/analysis aborts, reporting a violation. A "bare" expression can occur where a statement is expected. It is blocked in any state in which it evaluates to zero. If not blocked, it can proceed, and behaves like skip. In effect it waits for itself to become true, which means there is no need in many cases to model waiting with some kind of busy-waiting loop. The `run` statement is blocked if the current number of processes equals the maximum allowed. If this is not the case, then it is an atomic action that starts an instance of the named process.

For composite statements, executability depends on that of the atomic statements that are first in line to execute. For sequential composition, the whole is executable if the first statement is. The conditional and iteration notation is very similar to that in Dijkstra's Guarded Command Language[22]. The only difference here is that the first (guard) component can be a general statement, and need not be an expression. Both the `if` and `do` statement are executable if at least one of their statement sequences is executable. If more than once choice is executable, then a non-deterministic choice is made between them. The `if`-statement terminates when its chosen branch does, while the `do` will repeat the whole choice process. There is a special atomic statement `break`, only valid in loops, that terminates the loop. The `atomic` keyword makes its enclosed statements execute atomically (no other process can run). The exception is if a sub-statement is not executable, in which case the atomicity breaks to allow other processes run. When that statement once more becomes executable, then it resumes running atomically.

Promela has datatypes similar to those found in C, with some variants where it is possible to specify the number of bits. It also allows one-dimensional arrays, and a record notation very similar to defining C "structs".

The scope for procedural abstraction is quite limited. Process types are defined using the `proctype` keyword, take named parameters, and can define local variables. However these define complete processes, and can't be used to abstract a part of the process behaviour. The Promela language uses the C preprocessor, and also the `inline` construct, which has named parameters, but performs syntactic substitution.

The SPIN model-checker takes a Promela file and compiles it into a C model-checking program tailored to the model defined in that file. It then can perform a wide range of exhaustive analyses of that model, looking for deadlock, livelock, starvation, unfairness, and failing `assert` statements. In addition, it can take temporal properties described using Linear Temporal Logic (LTL) and check those for possible violations. The models used by SPIN are extended Büchi

automata [35, Chp. 6,7]. These are finite-state machines to which a criterion for accepting infinite sequences has been added, namely that every cycle in the model involves visiting a designated accepting state. It is interesting to note that He Jifeng has been involved in explorations of the relationship between LTL and Büchi automata [37].

3 Formal Models Of RTEMS

We being with an overview of the formal approach adopted, followed by using one of our models as an example of what is actually involved. We present a high level overview, and then use one model/test scenario to discuss some of the complexity that arises, and finish describing other processes used to model OS behavior.

3.1 Formal Approach

Our overall approach was to start with the RTEMS documentation, most notably that contained in the Classic API Guide[44]. This has sections that cover key concepts, as well as specific sections describing services in terms of *Managers*. These sections typically define an Application Interface (API) by specifying the relevant C prototypes, describing how to call these, what their effects are, and what kinds of error or success indicators get returned.

The Chains API [44, §34] was used initially to figure out the end-to-end methodology, from a Promela model to running passing tests on both simulators and real hardware. The Promela model would describe correct behavior, and also specify desirable properties using assertions and LTL. We would use SPIN to check it in the usual way to ensure correctness of our model (deadlock freedom, assertion checks, etc.) We then used the fairly obvious idea[17] of taking each property, negating it, and re-running the model-checker. It would then report a violation and issue a counterexample. However, this counterexample is an example of a correct run of the system, and hence can be used as a scenario for test generation.

Normally SPIN stops once any error is found and returns the relevant counterexample, but, it can also be asked to continue checking the entire model to find *all* errors. This can be exploited to get a collection of scenarios that give a range of correct behaviours. If we have models that are guaranteed to terminate, then they can be used to generate *all* possible correct behaviours by adding `assert(false)` at the end.

All of the models completed to date are ones that terminate. This is because, in addition to the Chains API, we have focussed on modelling Managers associated with task synchronisation of some form: events, barriers, messages, semaphores. None of these require models that run forever, because the requirements focus on the outcomes of making the various API calls, in terms of side-effects and return codes. What is important is the interaction between such

calls performed by concurrent tasks. The key metric being used to gauge test quality is code coverage. This does not require models to be non-terminating.

In addition, we need to ensure that the scenarios we produce from our models do terminate, because a test is not helpful if it fails to terminate (in practice test frameworks put timeouts in place to abort looping tests).

We now describe how we modelled parts of RTEMS using Promela, in a manner that would support test generation, using the Events Manager [44, §15] as a running example. The Chains API got the basics going, but only involved one RTEMS task, while this Manager involves at least two tasks in general, and also has requirements regarding priority and preemption for these tasks.

3.2 The RTEMS Event Manager

The Events Manager [44, §15] allows RTEMS tasks to send and receive event-sets, where an event is a number between 0 and 31 inclusive. The meanings of these numbers are application-specific, so the Events Manager is only concerned with their transmission, and does not care what they might mean. There are two API calls in this Manager (Send,Receive):

- (Send) `rc = rtems_event_send(id,events)` sends the event-set `events` to the task with identifier `id`.
- (Receive) `rc = rtems_event_receive(wanted,options,ticks,rcvptr)` is called by a task looking to receive events in event-set `wanted`, and if successful, will find the obtained events at the location pointed to by `rcvptr`. The parameters `options` and `ticks` are used to specify waiting criteria and a timeout interval.

Both calls return an RTEMS status code, shown above as `rc`.

We modelled all behavior of these API calls, including the situations that resulted in error status codes. These include invalid values for parameters such as task identifiers `id` or the receive pointer `rcvptr`. The Receive operation has various waiting options (none,timeout,forever) so can report being unsatisfied or having timed out. These can all be checked with a test that just calls one or other API appropriately.

The gist of correct behavior is as follows: every RTEMS task has an associated pending event-set variable, initially empty. The effect of sending an event-set is to add those events into the pending set. When, or if, the receiver is satisfied, the satisfying events only are removed from the pending set and are written to the location pointed to by `rcvptr`. The tests need to verify that these pending event-sets are modified correctly. This can be done using the Receive call, specifying an empty set for `wanted` which simply returns the current value of the pending set without modifying it.

In our Promela model, we only needed at most four events, to get all relevant test combinations, that exercise all paths through the code. For a given call of Receive, one model event models all unwanted events, then we have two model events to capture that it may take more than one send to satisfy the receive.

Any combination of 32-bit wanted and sent event sets can be refined down to 4-bit sets that capture the same pattern of behavior.

3.3 High Level Model Overview

It is clear that our Promela model needs to capture the correct behavior of the two API calls, based on a careful reading of the documentation. We determined how these could then be orchestrated to produce useful tests by looking at existing RTEMS test code for the Events Manager. The basic structure of a test was that an initial runner task would be started which would initialise the test state and also start a number of worker tasks, as needed to participate in the test. The runner task would then call parts of the API, while the worker tasks would typically do something complementary. For the Events Manager, the runner played the role of a task doing Receive, while one worker did one or two Sends. When the test was done, the runner task would perform the appropriate teardown procedure.

We use Promela processes to model RTEMS runner and worker tasks. So we chose to model a situation that had two RTEMS tasks, one that would perform between zero and two event sends (`Send`), while the other performed at most one receive operation (`Receiver`). We wanted to support a range of scenarios, from those that checked error-reporting for individual API calls, to those that mixed a receive call with up to two send calls. We defined general scenario types using Promela's only enumeration type:

```
mtype = {Send,Receive,SndRcv,SndRcvSnd,...};
mtype scenario;
```

The idea is to specify that the scenario choice is nondeterministic. We do this using a conditional statement where each guard is an always executable assignment:

```
if
::  scenario = Send;
::  scenario = Receive;
::  scenario = SndRcv;
::  scenario = SndRcvSnd;
::  ....
fi
```

The value of `scenario` would then be used by deterministic conditionals to initialize variables that determined detailed flow of control.

3.4 Modelling Send;Receive;Send

We will now look at a single scenario where the worker performs a Send first, and then the runner does a Receive, where it opts to wait either for a timeout or indefinitely, and finally the worker does a second Send. We assume that the

first Send does not satisfy the Receive, but that the second Send adds in what was missing. The description of Send and Receive given earlier focussed on the receiver's `wanted` and `pending` event sets, but this is not the full picture. We have two tasks synchronizing over these event sets, when the receiver, when called, is not satisfied. So it blocks, either indefinitely or for a specified timeout interval. These are different blocking circumstances that can lead to different return code outcomes, so we need to model this distinction in our Receive API model.

While the behavior of the Send seems simple, just being an update of the `pending` set, we do in fact also need to model that this may unblock a waiting receiver. In effect, we need to have a variable `state` associated with each Promela process that models the corresponding tasks RTEMS scheduler state (executing, ready, blocked, dormant, and non-existent [44, §5.2.5]). In practice we need to model `Ready`, and three variants of being blocked (`EventWait`,`TimeWait`, `OtherWait`). The first two model Receive waiting indefinitely or for a timeout. The third models a case where the Send can be forced to wait, due the following requirement for `rtems_event_send`:

> "The calling task will be preempted if it has preemption enabled and a higher priority task is unblocked as the result of this directive. "[44, §15.4.1, Notes]

Clearly we needed to model priorities as well but we don't discuss this here.

```
inline event_send(self,tid,evts,rc) {
atomic{
    if
    ::  tid >= BAD_ID -> rc = RC_InvId
    ::  tid < BAD_ID ->
        tasks[tid].pending = tasks[tid].pending | evts
        unsigned got : NO_OF_EVENTS;
        bool sat;
        satisfied(tasks[tid],got,sat);
        if
        ::  sat ->
            tasks[tid].state = Ready;
            preemptIfRequired(self,tid) ;
            waitUntilReady(self);
        ::  else -> skip
        fi
        rc = RC_OK;
    fi }}
```

Fig. 3. Promela specification of `rtems_event_send`

The resulting behavior for Send is modelled by the Promela `inline` definition in Fig. 3. There are three other `inline`s called by `event_send`:

- `satisfied` encodes when a receiver is satisfied.
- `preemptIfRequired` checks if the sender is required to be preempted, and if so sets its state to `OtherWait`.
- `waitUntilReady` blocks internally on the expression statement `state == Ready`, waiting for something else to make it so.

In the Receiver, we use `satisfied`, can set state to `TimeWait` or `EventWait`, and call `waitUntilReady`.

Given that both Send and Receive can block, we need some other mechanism to unblock them. A satisfying Send can unblock a waiting Receiver, but so can a timeout. We also need to model a preempted Send being eventually free to run again, once the higher priority Receive is done. This achieved by adding two Promela processes called `Clock` and `System`. The `Clock` process emits regular clock ticks and decrements timeout data associated with processes in state `TimeWait`, setting their state to `Ready` when the timeout reaches zero. The `System` process models relevant parts of the RTEMS scheduler, mainly the fact that processes in state `OtherWait` eventually become `Ready`. The Send and Receive processes terminate and set their state to `Zombie`, and the system process watches for this and then ends the model run when all processes are done. The last line in the model used for test generation is:

```
assert(false);
```

3.5 Towards a UTP Semantics for Promela

We can summarise the subset of Promela that we use with the following abstract syntax. We assume appropriate types t, expression syntax e, and process names p. We then start with statements:

$$\begin{aligned} s ::= {} & II \mid e \mid x := e \mid \mathbf{assert}\ e \mid s_1; s_2 \\ \mid {} & \mathbf{if}\ s_1, \ldots, s_n\ \mathbf{fi} \mid \mathbf{do}\ s_1, \ldots, s_n\ \mathbf{od} \\ \mid {} & \mathbf{atm}\ s \mid \mathbf{run}\ p(e_1, \ldots, e_n) \end{aligned}$$

The language here is very reminiscent of *stateful-failure reactive designs* in Foster et al. [25], with some laws like the following:

$$\mathbf{if} i \in I \bullet b(i) \pfun P(i) \mathbf{fi} = (\prod_i b(i) \pfun P(i)) \sqcap ((\neg \bigvee_i b(i)) \pfun \mathbf{chaos})$$

However, guards b above can be general statements s here, with the Promela notion of executability. In particular, an **if** with all branches blocked is simply blocked itself, and does not behave like **chaos**, and their $b \pfun P$ becomes $b; P$ in our language.

He Jifeng and his colleagues have been exploring semantics for Verilog for two decades [31,41,50] A common feature of this work is using UTP to help link the

different semantic forms: algebraic, denotational, and operational. In recent work on MDESL, a Verilog-like language, they address shared variable concurrency using *pre-emption points* [40, Defn. 2.1]. These occur at specific points related to timing and parallel constructs, and are the only places where the scheduler can allow the environment to run. A discrete time model is presented as time-stamped sequences of sequences of snapshots. A chop operator ($P \frown Q$) defines sequential composition on this model, and then auxiliary design variables (e.g. *ok*) are added, and MDESL sequential composition is defined using chop. The final result is a healthy process

$$\mathbf{H}(\neg div(P) \vdash wait(P) \lhd wait' \rhd ter(P))$$

which also has the form of Foster et al.'s *reactive contract* [26].

In Promela, every basic action is a pre-emption point, with the exception of inside an `atomic`, which corresponds closely to the notion of *atomic action* in the MDESL semantics. In a sense Promela is very close to maximally interfering shared-variable concurrency, as modelled in our work on UTCP[15]. However this is very low-level, and needs to have abstractions built on top in order to be useable. Perhaps MDESL could be such an abstraction?

Finally, we note that the concept of model-checking and associated temporal logics has been given a UTP formulation by Anderson et al. [8].

4 Refining Promela to C

4.1 Observing SPIN Counterexamples

Once satisfied that the Promela model is correct by using SPIN to verify properties, we then negated those properties in order to obtain test scenarios. The counter-example output produced by SPIN is designed to be read by the model authors, and reports state values using Promela syntax, as well as line-numbers in the model text. However, we wanted to automate the process of converting a counterexample into a test, so we needed to have a more generic way to see what was happening in the model, using a notation that was easy to parse.

Promela has a `printf` statement that supports a simple subset of the one available in C. It has no effect when SPIN performs a verification run, but does produce output when SPIN is run in simulation mode, or when counterexamples are being displayed. We defined a simple observation language that we used to generate an appropriate textual *abstraction* of Promela state.

As an example, consider invoking `event_send` in the model. We want to know that we have called it, what its inputs were, and what its return code was when it returned. So we bracket its invocation with two `printf` statements:

```
printf("@@@ %d CALL event_send %d %d %d sendrc\n",
       _pid,taskid,sendTarget,sendEvents);
event_send(taskid,sendTarget,sendEvents,sendrc);
printf("@@@ %d SCALAR sendrc %d\n",_pid,sendrc);
```

Note here that the parameter `taskid` is the index into a task array, and is *not* the corresponding Promela process id. We start every such output with a marker string `@@@`, which is used to filter these statements out of SPIN's own reporting material. We then output the Promela process number, denoted by special Promela variable `_pid`. This is very important as we need to know when the running process changes if we are to generate test code that reproduces this scenario.

The next component is a keyword indicating what kind of observation is being presented. We use `CALL` to denote a function call, and `SCALAR` to denote a simple value. The function call then displays the arguments for the `inline` call (including the *name* of the `sendrc` placeholder). The scalar value will be the value of the return code. There is a wide range of other keywords that cover declarations, initialisation, atomic and structured values, task management, and logging.

An example output might be:

```
@@@ 3 CALL event_send 1 2 10 sendrc
@@@ 3 SCALAR sendrc 0
```

Here we see that Promela process 3 performed a call to `event_send` in which it identifies itself as being RTEMS task 1, with RTEMS task 2 as target, passing the event set $\{3, 1\}$, and storing its return code in variable `sendrc`. We then see that `sendrc` is a scalar variables whose current value is zero.

4.2 Refining `printf` Observations

In order to get test code we need to define a *refinement relation* we use to get from model output to C test code. We use a Python dictionary that maps names with arguments to a text item into which those arguments can be substituted. The keywords like `CALL`, `SCALAR`, and others, determine precisely how both the dictionary lookup and the resulting substitution is done. The dictionary is itself stored as a YAML file.

The process for refining both these observations is to lookup the name that immediately follows the keyword, substitute the arguments into the retrieved text, and add it to the code being generated. The refinement entries for `event_send` (simplified) and `sendrc` are:

```
event_send: |
  {3} = rtems_event_send( {1}, {2} );
sendrc:
  T_rsc( sendrc, {0} );
```

The `CALL` observation is refined by substituting in `2`, `10`, and `sendrc` into the `event_send` entry, while the `SCALAR` observation involves looking up `sendrc` and substituting in `0`. This results in the following C test code snippet:

```
  sendrc = rtems_event_send( 2, 10 );
  T_rsc( sendrc, 0 );
```

Here, `T_rsc` is a test function that checks that a return code has the specified value.

The YAML refinement dictionary is not the whole story. In addition to including the test framework, where `T_rsc` is defined, we also have to define some C functions that support the refinement. The actual test program will consist of a preamble in which such functions are defined, followed by the test code generated by the YAML refinement, and finishing off with a postamble that does proper test teardown.

4.3 Refining Task Switches

So far we have not discussed the use of the `_pid` number that follows the `@@@` marker. Consider the following (very) simplified extract from one of the Event scenarios, with added line numbers. This is the `SndRcvSnd` scenario, where the sender process sends some events to the receiver, then the receiver asks to receive some of those events plus others not just sent, and finally the sender sends more events that satisfy the receiver.

```
1. @@@ 3 CALL event_send 1 2 2 sendrc
2. @@@ 4 CALL event_receive 10 1 1 0 2 recrc
3. @@@ 4 STATE 2 EventWait
4. @@@ 3 CALL event_send 1 2 8 sendrc
5. @@@ 4 SCALAR recrc 0
```

Line 1 shows the sender sending event set {1} to the receiver. Lines 2–3 shows the receiver request to receive all events in {3}, but then blocks because it is not satisfied and hence enters the state `EventWait`. Line 4 shows the sender running again and sending event-set {3}. This satisfies the receiver. Line 5 shows the receiver with a success return code.

The first use of `_pid` is to partition the refined C code into distinct segments, one for each value of `_pid` that occurs. In the above example, the refinement of lines 1 and 4 will be added to segment 3 (Worker), while those in lines 2, 3, and 5 will be added to segment 4 (Runner).

The temporal sequencing of the three Event API calls here is important. These means that the corresponding C test code needs to suspend and waken the two RTEMS Tasks at appropriate points. In particular, the RTEMS test code needs to be *reproducible* in that it always interleaves concurrent code execution the same way every time. Effectively *all* the non-determinism has to be refined away.

The standard way of doing this is to used a so-called *simple binary semaphore*, that has two API calls, one to obtain such a semaphore, another to release it. Only one task can obtain it at a time, but any task can release it. Simple binary semaphores are suitable for task synchronisation, and the Promela model includes models of binary semaphores which are used appropriately, reported using `CALL`, and refined to calls to the RTEMS equivalent.

4.4 Towards a UTP Semantics of Promela-to-C Refinement

There are two stages to the refinement from the Promela model to C test code. The first stage links the semantics of the model that of the observation language used in the `printf` statements (e.g. `CALL`, `SCALAR`, etc.). The second stage links the observation semantics to that of the C code itself. In each case we need UTP semantics for the two parts along with a linking Galois connection [33, Chp. 4].

UTP Semantics of Model Observables. The observation semantics is fairly straightforward as it is really just a simple way of reporting basic Promela events, such as calling an inline definition, or reporting the observed value of a model variable. The definition of the linking predicates will need to make use of the contents of the refinement YAML file, and this observation semantics.

UTP Semantics of C. The first step in this stage is to have a UTP semantics for C. We don't need to work with the whole C language, but can restrict ourselves to the subset that is actually used in safety critical systems, for example, the widely used MISRA-C Standard [9]. These forbid the use of C constructions that are semantically problematic, such as an expression in an assignment that calls functions that update global variables. ESA mandates the use of coding standards [24], while RTEMS has its own [45, §6.3], which also result in safe C code.

This means we can treat the sequential parts of C as being essentially UTP Designs [33, Chp. 3], with the addition of separation logic, to deal with C pointers. We already have UTP material on separation logic in Woodcock et al. [49], which treated it as a sub-theory in a setting of heterogenous theories.

Concurrency semantics is also needed to cover both concurrent RTEMS Tasks, and hardware-level concurrency with a software impact, most notably interrupts. Again, it looks like our work on UTCP [15], and the work by He Jifeng and colleagues on MDESL [41] may help here also. Another key area to explore is the linkages between denotational and operational semantics, that have been explored by extensively by He Jifeng and colleagues down the years [30,33,40, Chp. 10].

5 Related Work

Promela Semantics. Most of the formal semantics material for Promela/SPIN is operational in nature. An early example was the notion of a *symbolic labeled transition system* [38]. This then inspired work using ACL2 for abstract syntax which defined the semantics as a functional program [10]. Another approach used SOS notation to build a three-layered operational semantics [48]. In the SPIN book [35], there are sections on its semantics, based on extended Büchi automata, with states defined with representations of variables, messages, processes and other system attributes. It then defines a *semantic engine*, which is basically a program in pseudo-code over this state space.

Formal Methods and Testing. In 1995 Gaudel wrote a seminal paper relating formal methods and testing [27]. This established a formal framework for talking about the relationship between formal specifications and test code. Here we shall discuss the key concepts in the context of our work with RTEMS. Key points made in that paper include the fact that a specification generally can not serve as a test oracle, and some form of refinement needs to be established between it and test code. For our work we need to construct a refinement from Promela via observations to C code. This provides what Gaudel terms the *conformance relation*. It is possible to formally define an exhaustive test set, which has all valid behaviours and is usually infinite in nature. but techniques need to be found to shrink this to a finite test set that is adequate. In our case, model-checking requires us to produce a finite model, and test generation requires limiting it to finite behaviours, so we address this during Promela model design. Papers about formal methods and testing in UTP include works by Cavalcanti and Gaudel [19,20] and Aichernig et al. [6,47]. These all explore the conformance relation concept. Also related is work by Aichernig and Jifeng on mutation testing [7].

Formalising Pointers. There have been a number of treatments of pointers in UTP, most looking at them in an object oriented context. Hoare and He did [34] early work on a trace model. Cavalcanti et al. [21] looked at pointers where storage is an equivalence class of variables that share the same memory location. Smith and Gibbons [43] present a similar notion based on sharable and containable locations.

6 Conclusions

We have described some of the work we did applying formal methods to an ESA-sponsored qualification activity for multi-core RTEMS. We focused on our use of Promela to model synchronisation facilities in RTEMS, and on using SPIN to generate tests. This involved defining a simple but novel observation language to output information about key model events that could be interpreted as a test specification. We then mapped these observations to actual C test code snippets.

During the above, we also explored how we could extend the formality of the work beyond just that of the operational semantics of Promela. This involved identifying what pre-existing UTP theories could be used to model both Promela itself, and the refinement chain that involves the observation language and a suitable subset of the C programming language.

The current state of play is that models and test generation software for the Chains API, and the Event, Barrier and Message Managers, are now available from the RTEMS Central git repository [1], in the `formal` sub-directory. A new draft section on Formal Verification for the RTEMS Software Engineering manual [45] is under review by the community. More work has since be done mainly involving student projects, that has yet to be submitted to RTEMS for review and inclusion. This ongoing work in this area is hosted on Github [14].

6.1 Future Work

There is much work still to be done, with RTEMS. We plan to model far more of RTEMS than done so far, as well as revisiting and re-factoring the existing models. In particular, there is a need to formalise the new SMP-aware scheduler thread queue algorithms, that are considerably more complex than the single core versions, involving, for instance, task migration between cores.

In addition, the work done by Edisoft, Embedded Brains and CISTER has resulted in a new concept for requirements capture called *specification items* [45, §5], that encode enough information about RTEMS code artifacts to allow tools to build test code, run tests, collect data, and generate reports. None of this material is formal in any sense, but it does included descriptions that map abstract pre/post-conditions to test code snippets. This opens the possibility, given an appropriate UTP semantics, of being able to extract material to contribute to a formal specification in UTP.

The real plan for future work here is to use RTEMS, with the Promela models, the observation language, and the explicit use of C code, as a case study for using UTP to develop a unified semantic model of all these components, and their linkages. As has been pointed out, the work of He Jifeng has a great amount to contribute to this endeavour.

References

1. RTEMS Central GIT repository. https://git.rtems.org/rtems-central
2. RTEMS GIT repositories. https://git.rtems.org/
3. RTEMS Improvement by Edisoft. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS_EDISOFT
4. RTEMS Improvement by Embedded Brains. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Software_Systems_Engineering/RTEMS-SMP_Improvement_for_LEON_multi-core
5. RTEMS website. https://www.rtems.org/
6. Aichernig, B.K.: A testing perspective on algebraic, denotational, and operational semantics. In: Ribeiro, P., Sampaio, A. (eds.) UTP 2019. LNCS, vol. 11885, pp. 22–38. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31038-7_2
7. Aichernig, B.K., He, J.: Mutation testing in UTP. Form. Asp. Comput. **21**(1–2), 33–64 (2009). https://doi.org/10.1007/s00165-008-0083-6
8. Anderson, H., Ciobanu, G., Freitas, L.: UTP and temporal logic model checking. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 22–41. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14521-6_3
9. Banham, D., et al.: MISRA C:2012 Guidelines for the Use of the C Language in Critical Systems. MISRA Limited, March 2013
10. Bevier, W.R.: Toward an operational semantics of PROMELA in ACL2. In: SPIN'97. Twente University, Enshede, Netherlands, pp. 1–20 (1997). https://spinroot.com/spin/symposia/ws97/bevier.pdf
11. Bloom, G., Sherrill, J., Hu, T., Bertolotti, I.C.: Real-Time Systems Development with RTEMS and Multicore Processors, 1st edn. CRC Press, Boca Raton, November 2020

12. Brandenburg, B.B.: A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In: Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013), pp. 292–302 (2013). http://www.mpi-sws.org/~bbb/papers/pdf/ecrts13b.pdf
13. Burns, A., Wellings, A.J.: A schedulability compatible multiprocessor resource sharing protocol - MrsP. In: Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS 2013) (2013). http://www-users.cs.york.ac.uk/~burns/MRSPpaper.pdf
14. Butterfield, A.: Formal RTEMS-SMP repository. https://github.com/andrewbutterfield/RTEMS-SMP-Formal
15. Butterfield, A.: UTCP: compositional semantics for shared-variable concurrency. In: Cavalheiro, S., Fiadeiro, J. (eds.) SBMF 2017. LNCS, vol. 10623, pp. 253–270. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70848-5_16
16. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-circus. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 75–97. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73210-5_5
17. Callahan, J., Schneider, F., Easterbrook, S.: Automated software testing using model-checking, pp. 118–127 (1996)
18. Catellani, S., Bonato, L., Huber, S., Mezzetti, E.: Challenges in the implementation of MrsP. In: Reliable Software Technologies - Ada-Europe 2015, pp. 179–195 (2015)
19. Cavalcanti, A., Gaudel, M.-C.: A note on traces refinement and the conf relation in the unifying theories of programming. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 42–61. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14521-6_4
20. Cavalcanti, A., Gaudel, M.-C.: Specification coverage for testing in circus. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 1–45. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16690-7_1
21. Cavalcanti, A., Harwood, W., Woodcock, J.: Pointers and records in the unifying theories of programming. In: Dunne, S., Stoddart, B. (eds.) UTP 2006. LNCS, vol. 4010, pp. 200–216. Springer, Heidelberg (2006). https://doi.org/10.1007/11768173_12
22. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
23. ECSS: ECSS-E-ST-40C - Software general requirements. European Cooperation for Space Standardization (2009). https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/
24. ECSS: ECSS-Q-ST-80C Rev. 1 - Software product assurance. European Cooperation for Space Standardization (2017). https://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/
25. Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., Woodcock, J.: Automating verification of state machines with reactive designs and Isabelle/UTP. In: Bae, K., Ölveczky, P.C. (eds.) FACS 2018. LNCS, vol. 11222, pp. 137–155. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02146-7_7
26. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Theor. Comput. Sci. **802**, 105–140 (2020). https://doi.org/10.1016/j.tcs.2019.09.017
27. Gaudel, M.-C.: Testing can be formal, too. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) CAAP 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59293-8_188

28. Gomes, R.: Analysis of MrsP Protocol in RTEMS Operating System. Master's thesis, CISTER, Departmento de Engenharia Informática, Instituto Superior de Engenharia do Porto (ISEP), Portugal (2019)
29. Jifeng, H.: Transaction calculus. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 2–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14521-6_2
30. He, J., Li, Q.: A new roadmap for linking theories of programming and its applications on GCL and CSP. Sci. Comput. Program. **162**, 3–34 (2018). https://doi.org/10.1016/j.scico.2017.10.009
31. He, J., Xu, Q.: An operational semantics of a simulator algorithm. In: Arabnia, H.R. (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, 24–29 June 2000, Las Vegas, Nevada, USA. CSREA Press (2000)
32. Hierons, R.M., et al.: Using formal specifications to support testing. ACM Comput. Surv. **41**(2), 9:1–9:76 (2009). https://doi.org/10.1145/1459352.1459354
33. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Hoboken (1998). http://unifyingtheories.org
34. Hoare, C.A.R., Jifeng, H.: A trace model for pointers and objects. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 1–18. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48743-3_1
35. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, Boston (2004)
36. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. Form. Asp. Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
37. Li, J., Pu, G., Zhang, L., Wang, Z., He, J., Guldstrand Larsen, K.: On the relationship between LTL normal forms and Büchi automata. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 256–270. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_16
38. Natajaran, V., Holzmann, G.J.: Outline for an operational semantics of PROMELA. In: SPIN'96. Rutgers University, NJ, USA, pp. 1–17 (1996). https://spinroot.com/spin/symposia/ws96/Na.pdf
39. Paulson, L.C., Nipkow, T., Wenzel, M.: From LCF to Isabelle/HOL. Form. Asp. Comput. **31**(6), 675–698 (2019). https://doi.org/10.1007/s00165-019-00492-1
40. Sheng, F., Zhu, H., He, J., Yang, Z., Bowen, J.P.: Theoretical and practical aspects of linking operational and algebraic semantics for MDESL. ACM Trans. Softw. Eng. Methodol. **28**(3), 14:1–14:46 (2019). https://doi.org/10.1145/3295699
41. Sheng, F., Zhu, H., He, J., Yang, Z., Bowen, J.P.: Theoretical and practical approaches to the denotational semantics for MDESL based on UTP. Form. Asp. Comput. **32**(2–3), 275–314 (2020). https://doi.org/10.1007/s00165-020-00513-4
42. Sherif, A., Jifeng, H.: Towards a time model for circus. In: George, C., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 613–624. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36103-0_62
43. Smith, M.A., Gibbons, J.: Unifying theories of locations. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 161–180. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14521-6_10
44. The RTEMS Project contributors: RTEMS Classic API Guide (2021). https://docs.rtems.org/branches/master/c-user/index.html
45. The RTEMS Project contributors: RTEMS Software Engineering (2021). https://docs.rtems.org/branches/master/eng/

46. The RTEMS Project contributors: RTEMS User Manual (2021). https://docs.rtems.org/branches/master/user/
47. Weiglhofer, M., Aichernig, B.K.: Unifying input output conformance. In: Butterfield, A. (ed.) UTP 2008. LNCS, vol. 5713, pp. 181–201. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14521-6_11
48. Weise, C.: An incremental formal semantics for PROMELA. In: SPIN'97. Twente University, Enshede, Netherlands, pp. 1–20 (1997). https://spinroot.com/spin/symposia/ws97/weise.pdf
49. Woodcock, J., Foster, S., Butterfield, A.: Heterogeneous semantics and unifying theories. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 374–394. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_26
50. Huibiao, Z., Bowen, J.P., Jifeng, H.: From operational semantics to denotational semantics for Verilog. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 449–464. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_34