A Formal Modeling Approach for Portable Low-Level OS Functionality*

Renata Martins Gomes[1], Bernhard Aichernig[1][0000−0002−3484−5584], and Marcel Baunach[1][0000−0002−3716−2682]

renata.gomes@tugraz.at, aichernig@ist.tugraz.at, baunach@tugraz.at
Graz University of Technology, Austria

Abstract. The increasing dependability requirements and hardware diversity of the Internet of Things (IoT) pose a challenge to developers. New approaches for software development that guarantee correct implementations will become indispensable. Specially for Real Time Operating Systems (RTOSs), automatic porting for all current and future devices will also be required. As part of our framework for embedded RTOS portability, based on formal methods and code generation, we present our approach to formally model low-level operating-system functionality using Event-B. We show the part of our RTOS model where the switch into the kernel and back to a task happens, and prove that the model is correct according to the specification. Hardware details are only introduced in late refinements, which allows us to reuse most of the RTOS model and proofs for several target platforms. As a proof of concept, we refine the generic model to two different architectures and prove safety and liveness properties of the models.

Keywords: Event-B· RTOS · Portability · Refinement · Verification

1 Introduction

The amount of devices in the Internet of Things (IoT) (e.g. autonomous vehicles, smart infrastructures, automated homes and production facilities), is expected to increase exponentially, along with the diversity on both the hardware and the software side [20,15]. Operating System (OS) developers, who currently focus on just a couple of different computing platforms, will face a huge variety of devices, ranging from simple single-core to more complex multi or many-core systems, including specialized ASIC or even reconfigurable FPGA components [18,12].

While high-level code can more easily be compiled for another hardware, low-level functionality (i.e. context switches, system initialization routines, interrupt handling) are still handwritten for each architecture. To support a new Instruction Set Architecture (ISA), for example, an OS must have many low-level parts completely rewritten, which requires in-depth knowledge of both software and hardware, including a deep understanding of their interaction. From our own experience and industry cooperations, supporting additional MCU families or just

* This work is partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments"

variants is not straightforward, even if there are only a few differences to existing ports. This increases the development time, often limiting OS support to a low number of devices. Even though the code base of many OSs is modular, there are noticeably often just a few complete and directly usable ports available. Especially when developed under time pressure, wrong implementations, new bugs, and security breaches are common. Besides, changes in the logic of low-level software must be manually introduced to all implementations, which also hinders or slows down important improvements of the OS. Since dependability is key for the IoT [15,36,4], we propose an approach to improve the portability of embedded Real Time Operating Systems (RTOSs) based on formal models, refinement, and code generation to target-specific code.

We report on the first part of the approach: the modeling and verification of low-level functionality. Parts of the model of an RTOS are presented that focus on the switch into the kernel and back to a task, detailing the operations that happen in these transitions. We chose the context switch as demonstrating example, because it is architecture-specific, requires reimplementation for each architecture, and its correctness is crucial: corrupted task contexts, resulting from incorrect implementations, may produce errors that are hard to find and compromise the OS's ability to properly interleave concurrently running tasks.

First, we model the OS execution flow and functionality incrementally through formal refinements. Then, as a proof of concept, we further refine the generic OS model to two different architectures: an MSP430 and a RISC-V. In order to verify safety (something bad must *not* happen) and liveness (something good *should* eventually happen) properties [25], we (1) prove that our RTOS models do not corrupt any task's context by properly saving and loading them, even though the process for saving and restoring a context differs for different MCU architectures; (2) prove that the kernel runs in the appropriate CPU state and changes it as specified for task execution; (3) prove that the kernel executes in the correct order and finishes execution. Since we only introduce hardware details in late refinements, most of those proofs need only be done once, on the generic RTOS model. The refinement to each architecture, as well as their proofs follow and become much simpler, as we will show.

Contributions. To the best of our knowledge, this is the first time that OS low-level functionality is formally modeled with focus on its portability. Our main contributions in this paper are: (1) we decouple low-level functionality from hardware specifics; (2) a generic formal RTOS model with context switches; (3) safety and liveness verification of two instantiations of the generic model via interactive theorem proving and model checking.

Structure. Section 2 discusses related work and Section 3 provides background on the tools and the two target HW architectures. Section 4 presents the requirements and the modeled RTOS. Section 5 introduces the general idea of the modeling process and our refinement strategy, while the model itself is detailed in Section 6. Section 7 discusses verification. We conclude in Section 8.

2 Related Work

Many works investigate how to formally model and verify OSs. Brandenburg [6] introduces a new concept to develop RTOSs with time predictability. Craig [8,9] specifies an OS using formal models, also showing it can be refined to executable code. However, he either assumes several HW details or leaves further refinements to be done, conceding that "the hardware specification poses a slight problem". Novikov and Zakharov [29] verify Linux to detect faults in the entire monolithic kernel. Alkhammash *et al.* present modeling guidelines of FreeRTOS [2], and Cheng *et al.* model FreeRTOS' task model [7]. Stoddart *et al.* [33] model an interrupt driven scheduler in B. Danmin *et al.* [11] also use B to model an OS. Su *et al.* [34] design an RTOS memory manager in Event-B. With correctness proofs from abstract kernel function specification down to the binary file, seL4 [16] does not model the direct hardware instructions, assuming correctness of hardware-specific code, such as handwritten assembly code, boot code, etc [24]. Syeda and Klein [35] tackle the modeling of low-level cached address translation, whose correctness is usually left as an assumption, aiming to eventually integrate their model into seL4's verification. Baumann *et al.* [3] develop a method to model the underlying hardware of embedded systems such that the detailed modeling of each hardware component can be delayed, while still allowing the verification of high level security properties. OpenComRTOS [37], was designed with TLA+ and has been ported to several targets, however code is always handwritten.

Others tried to generate code from models: Hu *et al.* [21] report on problems found when trying to synthesize an OS. Fathabadi *et al.* [14] design a platform-independent formal model of an OS module that interacts with the HW and automatically generates C code. Dalvandi *et al.* [10] generate verifiable code from formal models. Popp *et al.* [31] generate information about device properties and addresses. Méry [28] presents formal modeling design patterns later translated to software. The generated software in these works is semantically dependent on the HW, but the syntax is pure C, not solving the problem of generating low-level OS code, that must, at least partially, be handwritten in assembly. Wright [40,41] formally specifies an entire ISA in Event-B and automatically generates a virtual machine capable of simulating it. Borghorst *et al.* [5] generate code for low-level OS functions using abstract assembly to describe the software and a HW architecture description to generate code, but without formal methods.

These works advance the state of the art in model-based OS development either by modeling and proving properties of high-level parts of the kernel (such as scheduling, task management, or resource management), or assuming HW characteristics that render the model HW-specific. Some code can often be generated for different programming languages, but the low-level code remains to be ported manually, and the models are not usable for different HW platforms. Our work aims at covering this gap on low-level software, providing an approach for its modeling and automatic code generation.

3 Background

Model Checking and ProB. Temporal logic [30,27] is a logic system that formally describes properties of time. Linear Temporal Logic (LTL) is the most popular and widely used temporal logic in computer science to specify and verify the correct behavior of reactive and concurrent programs [19]. It is particularly useful for expressing properties such as *safety* (given a precondition, then undesirable states that violate the safety condition will never occur), *liveness* (given a precondition, then a desirable state will eventually be reached), and *fairness* (involves combinations of temporal patterns of the form a predicate holds “infinitely often” or “eventually always”). ProB is an animator, constraint solver, and model checker for the B-Method [26] that integrates as a plugin-in Rodin and can be easily used for LTL model checking of our RTOS models.

```
SETS // sets block
 S
VARIABLES // variables block
 x ⊆ S
 f ∈ S ⇸ DATA // f is a partial function
 b ∈ S ⤖ DATA // b is a bijective function
eventName // event block
ANY // parameter block
 p ⊆ x
WHERE // guards block
 p ≠ ∅
 x ⩤ b = x ⩤ f // b and f are equal for all domain elements not in x (domain subtraction ⩤)
THEN // actions block.
 f := f  (p ◁ b) // f is overwritten () by the pairs in b whose first element is in p (domain restriction ◁)
 x := x \ p // set subtraction
```

Fig. 1: Event-B notation

Event-B and Rodin. Event-B is a formal method for system-level modeling and analysis [1,13]. The Rodin Platform [23], an Event-B IDE based on Eclipse, supports the development and refinement of models with automatic generation and partial discharging of mathematical Proof Obligations (POs). Event-B is based on set theory and state transitions. The two top elements of a model are *contexts* and *machines*. The term *context* is overloaded in the domains of OSs and formal methods. Hence, we will always refer to a context in Event-B as *Event-B context*. Event-B contexts describe all static information about the system. They are composed by carrier sets, constants, and axioms. Dynamic information is represented by the machines, which are composed of variables, invariants, and events. Event-B machines can *see* Event-B contexts, such that the machine can use their constants and axioms to relate static and dynamic information as well as to discharge POs. Considering machine states as sets of the variables’ values, each event represents a state transition. Events are enabled if the state satisfies the corresponding *guard* condition and modify the current state according to their *actions*, which are executed in parallel when the event is enabled. When a machine refines another one, it must refine (or keep) its events, adding details or more variables; POs are automatically generated by Rodin to e.g. guarantee that invariants always hold and that refined events do not contradict the abstract ones. In order to ease the understanding for readers not familiar with Event-B, Figure 1 presents a summary of the Event-B notation used in our models. We assume the reader is familiar with basic set theory notation.

Next, we present the main characteristics of the two HW architectures we use as examples for the architecture-specific instantiations of our model in Section 6.

MSP430. The TI MSP430 [22] family of MCUs comprises a range of ultra-low power devices featuring 16 and 20-bit RISC architectures with a large variation of on-chip peripherals, depending on the model. Among its 16 registers are general purpose registers, the program counter `PC`, the stack pointer `SP`, and the status register `SR`, which stores status flags, such as interrupt enabled, overflow, etc. The MSP430 offers a very simple architecture with only one execution mode and a fully orthogonal instruction set. There is no privileged mode, nor any memory protection or memory management unit. Once an interrupt occurs, the `PC` and `SR` registers are pushed onto the stack. Then, further interrupt requests (IRQs) are disabled and the `PC` is overwritten with the address of the first instruction of the corresponding interrupt handler. The return from interrupt instruction, i.e., `RETI`, restores `SR` and `PC` from the stack and finally continues where the handler has interrupted the regular execution flow.

RISC-V. The open RISC-V instruction set architecture [32] was originally developed by UC Berkeley and is meanwhile supported by a highly active community of software and hardware innovators with more than 100 members from industry and academia. The RISC-V is a load-store architecture, and its specification [39] defines privilege levels used to provide protection between different components of the software stack. We refer to an implementation that supports user and machine modes, with 32-bit integer and multiplication/division instructions (RV32IM) [38]. There are 32 registers available in all modes, including a zero register and the program counter `pc`. The calling convention specification assigns meanings to the other registers, such as a stack pointer `sp`, function arguments and return values. Additionally, Control and Status Registers (CSRs) with special access instructions are available for e.g. managing the CPU or accessing on-chip peripherals in defined privilege levels. An IRQ switches the CPU into a higher privilege level, while software can issue an `ECALL` instruction for that. In both cases, returning to user level is done by the instruction `URET`.

4 Requirements

We modeled the *MCSmart*OS [17], an RTOS we have developed and used for many years. This section presents its architecture and requirements. An important concept to understand is the *context* (not in the sense of Event-B, but in the sense of operating systems): A *context* is a set of information and configuration of a CPU or a CPU core that is required to control the execution flow of software, i.e. code sequences. Depending on the CPU state and external events, cores can usually switch between different code sequences by loading their respective context. To be able to continue an earlier code sequence from the interrupted instruction, its context is saved before the switch. The actual switching process as well as the composition of the contexts is defined by the interrupt concept of the CPU; in any case, the hardware automatically saves and loads the contexts. If, in addition to interrupts of the hardware, an OS supports preemptive, i.e. interleaved executable tasks (or threads or processes), the *context* is extended. In order to switch between tasks, this extended information is saved (previous task) and loaded (next task) by the kernel. Next, we describe our assumptions about the computing platform and present *MCSmart*OS's requirements.

4.1 Hardware Assumptions

Even though we aim on keeping the OS model initially independent from the hardware, a target architecture must have certain features in order to be capable of running an operating system. Focusing only on the relevant aspects for our RTOS model, we define data storage and interrupt handling features as environmental assumptions, numbering and labeling them ENV. Different OSs might require other features, but that does not affect our general concept.

ENV1 The CPU provides means to store/load data to/from referable locations. These locations can be, e.g., registers or memory addresses.

ENV2 The context is a well-defined subset of locations and their stored values that the CPU requires for execution of a code sequence. It must be saved when the code sequence is interrupted, so that it can later be resumed from the same point.

ENV3 The CPU has an interrupt enabled flag. Interrupts will only be accepted if the interrupt enabled flag is set.

ENV4 When an interrupt is accepted, the values of the context or a part of it are automatically copied into other locations defined by the architecture.

ENV5 The CPU offers a "return from interrupt" instruction that automatically loads the context with the values automatically saved when the interrupt was accepted (ENV4).

ENV6 The saving process (ENV4) is allowed to modify the context values before they are saved, according to a well-defined and CPU-specific function.

ENV7 The restoring process (ENV5) must reverse the modification of ENV6.

4.2 Software Specification

*MCSmart*OS provides, among many other features, a preemptive and priority-based scheduler for concurrent tasks. The kernel is invoked when an interrupt occurs or a syscall is called, and is divided into three parts: (1) the *kernel entry* is responsible for stack management and context saving. It unites both entry points, enters kernel mode, and continues to (2) the *kernel body* which handles the actual interrupt or syscall request and runs the scheduler that selects the task to be executed next. Finally, (3) the *kernel exit* executes a context switch by loading the selected task's context and returning to task mode.

In this work, we only model the context switches in *kernel entry* and *exit*, and the conditions required by the OS to execute its other functions. High-level kernel functionality, such as scheduling, task management, etc. is out of scope. The requirements for correct context switches and kernel execution (OS) are:

OS1 (A) A task executes on the context defined in ENV2. When not running, the values of its context are stored in locations reserved for context saving. (B) Each element of the context has its correspondent in the saved context.

OS2 (A) Once the kernel is invoked, *kernel entry* saves the old task's context. (B) On *kernel exit* the next task's context is loaded into the CPU.

OS3 (A) Each task has dedicated locations for context saving. (B) These locations with their stored values are the task's saved context, where contexts are saved to and loaded from (OS2).

OS4 The scheduler chooses the new task and is implemented in *kernel body*.

OS5 The cause for kernel invocation, unambiguously identifying which interrupt or syscall has occurred, must be recorded for use within *kernel body*.

OS6 (A) The *kernel body* always runs in kernel mode, with interrupts disabled, and on the OS stack. (B) Each task runs on its own stack, with the interrupt flag the same as it was when that task was running last, and never on kernel mode.

OS7 A part of *kernel entry* context saving and CPU preparation is automatically executed by the hardware (ENV4). The rest must be executed in software after the automatic part.

OS8 The kernel is exited with a return from interrupt instruction (ENV5). The task selected by the scheduler shall continue execution and where it was preempted before.

OS9 A part of *kernel exit* context loading and CPU preparation is automatically executed by the return from interrupt instruction (ENV5, OS8). The rest must be executed in software before the automatic part.

OS10 (A) If the values copied on interrupt (ENV4) are copied into task-specific locations, these locations and their data are considered a part of the saved context. (B) Otherwise, it is the OS's responsibility to save those values into the task's save context, and to copy them back where the CPU expects them to be when returning from an interrupt (ENV5).

OS11 If the architecture provides a privileged mode, *kernel body* runs in it, while tasks run in less privileged modes. Switching the mode must be done on *kernel entry* and *kernel exit*.

5 Refinement Strategy: from Abstraction to Detailed Specification

The model has several refinements and showing all would be too cumbersome. So, we divide it into 6 levels of abstraction (referred to as *Level*). Each Level is composed of several refinements and addresses a new set of requirements (Table 1). [1] Up to Level 4, the model remains generic, only requiring the generic hardware features described in Section 4.1. We only introduce further hardware details in Level 5, where we instantiate the model for specific target architectures.

[1] Model artifact at https://figshare.com/s/0f262342284eada236f5. The relationship between refinements and levels can be found in the README file. Model elements are referenced as [component.label].

This section introduces the general idea of each Level, while Section 6 details how each level was modeled. This model focuses on the interface between hardware and software in order to model the kernel's interleaved execution of concurrently running tasks. The goal is to prove that the kernel does not corrupt any task's context by properly saving and loading them, as well as to guarantee that tasks and *kernel body* run in the appropriate conditions described by the requirements from Section 4.

Level	**ENV**	**OS**
0	ENV1, ENV2	OS1
1	-	OS2
2	ENV6, ENV7	OS3, OS4
3	ENV3	OS5, OS6
4	ENV4, ENV5	OS7, OS8, OS9, OS10
5	Target-specific	OS11

Table 1: Model and requirements.

The state of an Event-B machine is the set of its variables' values, and state transitions are represented by the machine's events. In our model, these events represent the different parts of the kernel, building a state machine that starts with the switch into the kernel and finishes with the switch back to a task. The events, therefore, are modeled such that their order is well-defined, in the order the kernel parts must run: *kernel entry* executes first, then *body*, and finally *exit*; and the automatic part of *entry* executes before the manual part that must be executed in software (OS7), and in *exit* manual executes before automatic (OS9).

Level 0 In this initial abstraction, we only present the expected result of the OS execution, i.e., that an old context is saved and a new one is loaded, without modeling how this will be achieved. We also define the basic Event-B sets and their relations, used along the refinements.

Level 1 In the first refinements, we define the entry and exit parts of the kernel simply as two context copies: one in *kernel entry* for saving a context, and another one in *kernel exit* for loading a context. At this level, we do not yet define where those contexts are copied from or to, nor do we have any notion of tasks or conditions for proper task and *kernel body* execution.

Level 2 Next, we introduce tasks, their saved contexts, and *kernel body*. This level also defines where the context is saved to and loaded from.

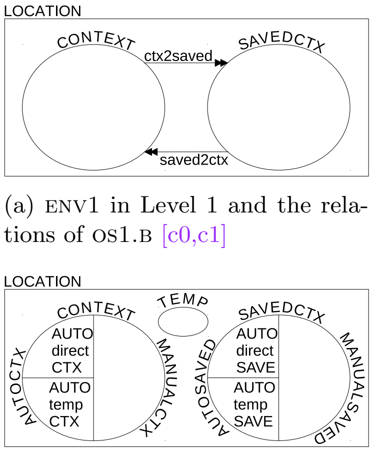
Level 3 Then, we introduce and set up the variables that control the conditions for proper task and *kernel body* execution (interrupts disabled, kernel mode, running on its own stack, and cause for the kernel execution).

Level 4 Refines the model to a generic hardware that automatically saves and loads a subset of the context, and the software that complements the switches.

Level 5 Finally, we refine the model into architecture-specific models from which OS code can be generated (code generation is not in the scope of this paper).

6 Kernel Model

This section details the levels from Section 5. Please, refer to Figure 1 for the Event-B notation used in the following listings.



(a) ENV1 in Level 1 and the relations of OS1.B [c0,c1]

(b) ENV1 in Level 4 [c2,c3,c4]

```
EVENTS
 osProgress anticipated
 THEN
  act1:loaded :∈ CONTEXT⇸DATA
  act2:saved :∈ SAVEDCTX⇸DATA
 END
 osFinal
 ANY
  new ∈ CONTEXT→DATA
  old ∈ SAVEDCTX→DATA
 WHERE
  grd3:loaded = new
  grd4:saved = old
 THEN
  skip //state not changed
```

(c) Level 0

Fig. 2: Diagrams of LOCATION and initial abstraction

Level 0 First, we define the carrier set LOCATION (Figure 2a), an abstract representation of memory addresses and registers. In combination with DATA, a subset of $\mathbb{Z}$, memory and registers can be represented according to ENV1. Two non-overlapping subsets with the same cardinality, CONTEXT and SAVEDCTX represent the subset of locations that compose the context (ENV2) and the subset of locations of a saved context (OS1.A), respectively. The bijective function ctx2saved ∈ CONTEXT⤖SAVEDCTX relates each context location to where it is saved, while saved2ctx is its inverse (OS1.B). The context is defined as a relation from CONTEXT to DATA, while a saved context is a relation from SAVEDCTX to DATA.

The initial abstraction (Figure 2c), sees the context switches as two context copies: one copies old context to saved ∈ SAVEDCTX ⇸ DATA, and the other loads new context to loaded ∈ CONTEXT ⇸ DATA. The old and new contexts that are copied are simply event parameters, that will later be refined into the actual contexts that are copied. The OS is modeled in the event osProgress. The event osFinal is not a part of the OS, but is only introduced to model the state where the OS has successfully executed. This event is composed only of guards, that is, it is enabled once the state represented in its guards is reached but does not change it anymore. The event osProgress, that represents the OS kernel, is allowed to change the variables saved and loaded, but does not yet describe how the copies are made. It is made *anticipated*, which, in Event-B, means that it may execute several times, but must eventually give up control and allow the model to reach osFinal. Refinements of an anticipated event must *converge*, i.e., decrease a variant, thereby proving that it eventually gives up control. The

```
1 osEntry REFINES osProgress
2 ANY
3  saveSet ⊆ toSave
4  old ∈ SAVEDCTX→DATA
5 WHERE
6  saveSet ≠ ∅
7  saved = toSave ⩤ old
8  loaded = ∅ //load not started
9 THEN
10  saved := saved ∪ (saveSet ◁ old)
11  toSave := toSave\saveSet
```

(a) Kernel entry

```
1 osExit REFINES osProgress
2 ANY
3  loadSet ⊆ toLoad
4  new ∈ CONTEXT→DATA
5 WHERE
6  loadSet ≠ ∅
7  loaded = toLoad ⩤ new
8  toSave = ∅ //save complete
9 THEN
10  loaded := loaded ∪ (loadSet ◁ new)
11  toLoad := toLoad\loadSet
```

(b) Kernel exit

Fig. 3: Level 1

idea is that, since the context is copied in different steps by HW and software, this event can be refined into these steps. The POs generated by Rodin verify that the refinements of this abstraction (Levels 1 to 5) are correct. If we prove that the model always reaches `osFinal`, we prove that the desired state after OS execution is reached. These proofs are shown in Section 7.

Level 1 Figure 3 shows the refinement of `osProgress` into two events: `osEntry` (OS2.A: *kernel entry* responsible for saving the old context), and `osExit` (OS2.B: *kernel exit* is responsible for loading the new context). They model state transitions (Figure 4), and their guards define that entry must happen before exit, and exit may only start after entry is done.

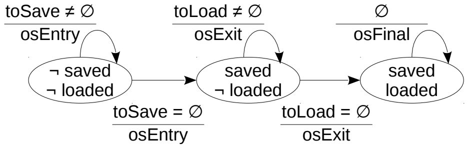


Fig. 4: Level 1 states and transitions

The new variable `toSave` ⊆ `SAVEDCTX` keeps track of the context yet to be saved, while the parameter `saveSet` defines the context subset saved in each run of `osEntry`. The event is made convergent on the variant `toSave`, and `saveSet` is subtracted from `toSave` in each run (Figure 3a, Line 11). This guarantees that, eventually, the save process will complete and we move to a *saved* state. Similarly, the new variable `toLoad` ⊆ `CONTEXT` represents the context yet to be loaded, while `loadSet` defines the context subset loaded in each run of `osExit`. The event is made convergent on the variant `toLoad`, and `loadSet` is subtracted from `toLoad` on each run (Figure 3b, Line 11), allowing the event to eventually reach the *loaded* state. Event `osFinal` remains unchanged.

Level 2 Next, we define the input constants `oldTask` ∈ `TASKS` and `oldCtx` ∈ `CONTEXT` → `DATA` that represent the old task and its context when the kernel was requested. The saving operation in `osEntry` must save `oldCtx` into `oldTask`'s save space. In order to save the context, we must map it to a saved context.

```
1 osEntry REFINES osEntry
2 ANY
3  saveSet ⊆ toSave
4 WHERE
5  saveSet ≠ ∅
6  toSave ⩤ t_saved(runningTask) =
    toSave ⩤ transform(oldCtx)
7  loaded = ∅ // load not started
8 THEN
9  t_saved(runningTask) := t_saved(
    runningTask) ◁- (saveSet ◁
    transform(oldCtx))
10 toSave := toSave\saveSet
```

(a) Kernel entry

```
1 osExit REFINES osExit
2 ANY
3  loadSet ⊆ toLoad
4 WHERE
5  loadSet ≠ ∅
6  loaded = toLoad ⩤ invTransform(
    t_saved(runningTask))
7  toSave = ∅ //save complete
8 THEN
9  loaded := loaded ∪ (loadSet ◁
    invTransform(t_saved(runningTask)))
10 toLoad := toLoad\loadSet
```

(b) Kernel exit

Fig. 5: Level 2

Additionally, the context might be modified during the saving process (ENV6), e.g. the stack pointer is changed before it is saved when the architecture automatically pushes some registers onto the stack. We must account for this modification, since what is finally saved is the transformed version of the values, and not the original input values. Thus, we define the functions `ctxTransform` ∈ `CONTEXT`→(`DATA`⤖`DATA`) and `transform` ∈ (`CONTEXT`→`DATA`)⤖(`SAVEDCTX`→`DATA`). The architecture-specific function `ctxTransform` is only declared at this level, and represents the modification of each value in the context. The function itself is only fully specified in Level 5. The function `transform` converts a context into a saved context according to `ctx2saved`, modifying the values stored in each location according to `ctxTransform`. This is modeled by the axiom

```
c1.axm7: ∀ ctx,el · ctx ∈ CONTEXT→DATA ∧ el ∈ CONTEXT ⇒
transform(ctx)(ctx2saved(el)) = ctxTransform(el)(ctx(el))
```

With these definitions in Event-B contexts, we also refine the machine variables and events (Figure 5). To represent the saved contexts of all tasks (OS3.A), `saved` is refined into `t_saved` ∈ `TASKS`→(`SAVEDCTX` → `DATA`), with glue invariant `saved` = `toSave` ⩤ `t_saved(oldTask)`. The new variable `runningTask` ∈ `TASKS` is equal to the constant `oldTask` before *kernel body* is run, and represents the new scheduled task after the scheduler has run. While `osFinal` always uses `oldTask` to check if the context has been correctly saved, `osEntry` refers to `runningTask` to save the context. This way, all three kernel parts (*entry*, *body*, and *exit*) deal with the same variable, simplifying code generation. We can finally replace the abstract save action in Figure 3a (Line 10) by the action in Figure 5a (Line 9).

```
1 osFinal (guards)
2  loaded = invTransform(t_saved(runningTask))
3  t_saved(oldTask) = transform(oldCtx)
4  toSave = ∅
```

Fig. 6: Level 2: `osFinal`

The context to be loaded during *kernel exit* actually comes from the saved context of `runningTask`, thus we replace the abstract load from Figure 3b

(Line 10) by the load in Figure 5b (Line 9). The functions `ctxTransform` and `transform` used for saving a context have their inverses, used for the load process, defined as `ctxInvTransform` ∈ `SAVEDCTX`⇸(`DATA` ⤖ `DATA`) and `invTransform` ∈ (`SAVEDCTX` → `DATA`)⤖(`CONTEXT` → `DATA`), related with the axiom

```
c1.axm8: ∀ sctx,el · sctx ∈ SAVEDCTX⇸DATA ∧ el ∈ SAVEDCTX ⇒
invTransform(sctx)(saved2ctx(el)) = ctxInvTransform(el)(sctx(el))
```

Now, we can refine the `old` and `new` parameters to reflect the real source and destination of the context copies (OS3.B) in Level 2 (Figure 5a, 5b, and 6).

Finally, the new event `osBody` models *kernel body* (Figure 7), abstractly representing the scheduler (OS4). We do not model *kernel body* in more detail in this work, but will refine it to guarantee its execution according to the OS requirements.

```
osBody
WHERE
 toSave = ∅
 toLoad = CONTEXT
THEN
 runningTask :∈ TASKS
```

Fig. 7: Level 2: `osBody`

Level 3 Though we do not model *kernel body*, we want to guarantee that *kernel entry* prepares the CPU to start its execution. Analogously, we do not model tasks, but want to guarantee that *kernel exit* prepares the CPU to run them. Thus, we introduce the variables that control the conditions for proper task and *kernel body* execution (OS5, OS6): `kernelMode` is a flag that indicates when the kernel has been entered. `osBody` can only be enabled if it is true, and `osFinal` if it is false; `kernelCause` records why the kernel has been invoked. It unambiguously identifies each interrupt and syscall, and must be valid within `osBody`; `interruptEnable` is the interrupt enabled flag (ENV3). It must be false in `osBody`, and loaded from the next task's saved context during *kernel exit*; `currStack` indicates the stack currently in use, abstractly representing a kernel or a task stack. `osBody` is enabled if `currStack` indicates kernel stack, while `osFinal` requires it to indicate task stack. We strengthen `osBody` and `osFinal` guards to fulfill OS5 and OS6. Modification of these variables in *kernel entry* and *exit* remain nondeterministic, since they are highly hardware-dependent. Figure 8 shows the new sets, variables, and guards. We also create the event `entryNothingToSave`, that mimics `osEntry` and is explained in Level 4.

Level 4 Now, *kernel entry* and *exit* are divided in two parts: one models what is *automatically* done by the hardware, via an interrupt acceptance or a return from interrupt instruction (ENV4, ENV5). This may save some registers, turn off the interrupt enabled flag, switch the CPU mode, etc. The remaining actions of *kernel entry* (OS7) and *exit* (OS8, OS9) are fulfilled by their *manual* parts.

This Level still does not refer to specific details of a potential target architecture. Therefore, the model must support different behaviors: the hardware might, on interrupt, copy a set of its registers into another set of registers designed for that (OS10.B), or it might copy them to memory, for example pushing them onto the stack (OS10.A). In the first case, we call this a temporary save, since the destination is the same for all tasks and must, therefore, still be made permanent by copying it to the task's saved context in *kernel entry*. To model

```
SETS (new)
 STACKS
 KERNELCAUSES
VARIABLES (new)
 kernelMode ∈ BOOL
 kernelCause ∈ KERNELCAUSES
 interruptEnable ∈ BOOL
 currStack ∈ STACKS
```

(a) New carrier sets and variables

```
AXIOMS (new)
 partition(STACKS,KERNELSTACK,TASKSTACKS)
 partition(KERNELCAUSES,{kCause_invalid},
  KCAUSE_SYSCALLS,KCAUSE_FLOWINTS)
```

(b) New axioms

```
osBody (new guards)
 kernelMode = TRUE
 kernelCause ≠ kCause_invalid
 interruptEnable = FALSE
 currStack ∈ KERNELSTACK
```

(c) New guards in body

Fig. 8: Level 3: Additions to model

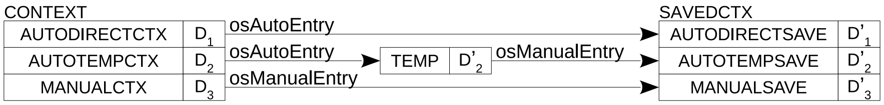


Fig. 9: *Kernel entry* context save. $\texttt{D}_x \subset \texttt{DATA}$ and $\texttt{D}'_x = \texttt{ctxTransform}(\texttt{D}_x)$

this, we partition CONTEXT and SAVEDCTX in three sections, as shown in Figure 2b: sections MANUALCTX and MANUALSAVED for the locations only manipulated in the manual part, and two others for those automatically handled by the hardware. AUTODIRECTCTX and AUTODIRECTSAVE for those locations permanently saved by the hardware, and AUTOTEMPCTX and AUTOTEMPSAVE for those first copied to/from a TEMP location. TEMP is another subset of LOCATION, created in this level.

We refine the events osEntry and osExit by splitting each in two, and refining their parameters (saveSet and loadSet in Figure 5) to differentiate the partitions in CONTEXT and SAVEDCTX. The transform and invTransform functions are also refined to reflect the refinements of this level and the separation of the different levels of context copy. The events are refined according to Figure 9: osAutoEntry saves values from AUTODIRECTCTX into AUTODIRECTSAVE and copies from AUTOTEMPCTX to the new variable $\texttt{temp} \in \texttt{TEMP} \rightarrow \texttt{DATA}$. For architectures that only copy part of the context to temporary locations, not saving anything, we refine entryNothingToSave into tempSave, adding a copy to temp and making it convergent on the variant $\texttt{TEMP}\setminus\texttt{dom(temp)}$ (must add elements to temp). We also keep entryNothingToSave only modifying variables from Level 3. Then, osManualEntry saves MANUALCTX into MANUALSAVED and copies temp into the AUTOTEMPSAVE location, completing the saving of the temporary part of the context. The inverse operation is modeled in *kernel exit*: First, osManualExit loads from MANUALSAVED into MANUALCTX and copies from AUTOTEMPSAVE into temp, then osAutoExit loads all AUTOCTX from temp and AUTODIRECTSAVE.

Level 5 - Architecture-specific Instantiations Having intentionally modeled the OS independent from the hardware so far, we finally introduce hardware details in a new refinement level per target architecture. For each, we extend the Event-B context and define, within LOCATION, all registers available in the architecture. At the same time, we also define which of them are part of CONTEXT (and to which subset), TEMP, etc. We also define the locations for saved contexts and the CPU-specific functions ctxTransform and ctx2saved. For each target, one Event-B machine refines the last Level 4 machine, *see* the correspondent Event-B context, and the architecture-specific actions from Level 3 are made deterministic. On interrupt, RISC-V copies some registers into temp, not saving anything directly, therefore osAutoEntry is never enabled and we can remove it. tempSave is refined into interrupt and ecall, which are very similar: both switch into kernel (machine) mode, disable interrupts and copy data into temp. Additionally, interrupt registers the interrupt ID (kernelCause). For a syscall, we need to do this before ecall, so we refine entryNothingToSave into __syscall. We can refine kernelMode to the privilege levels, adding an invariant that relates kernelMode = TRUE to machine mode and kernelMode = FALSE to user mode (OS11). On MSP430, registers are pushed onto the task's stack, and there are no temporarily saved registers, so tempSave is removed. Without privileged modes, kernelMode is a variable, which is set to TRUE in osManualEntry and to FALSE in osManualExit. osAutoEntry is refined into interrupt and __syscall. The latter imitates the former, additionally registering which syscall triggered the kernel. Since the interrupt can not do this automatically, entryNothingToSave is refined to interruptHandler, which is enabled after the interrupt.

7 Proofs and Model Checking

This section shows the properties we verified via theorem proving in Rodin and LTL model checking. From the requirements, we elaborate safety properties to be proved: (**S1**) Contexts are never corrupted by the kernel (OS2), (**S2**) osBody always runs in the specified conditions for its execution and osFinal is reached with the specified conditions for task execution (OS5, OS6). And liveness properties guarantee the model reaches the intended states: (**L1**) The kernel executes in the correct order, and (**L2**) always finishes execution (always reaches osFinal).

7.1 Theorem Proving

All refinements in our model must correspond to their abstraction, which is proved with discharging the POs generated by Rodin. The initial abstraction, Level 0, defines the state (osFinal guards) we want to achieve after OS execution, namely that an old context is saved and a new context is loaded. This state must be reached by osProgress, which models the OS. Event osProgress is refined into the three main parts of the kernel (entry, body, and exit). Entry and exit are responsible, respectively, for saving the old task's context and loading the new task into the CPU. The first abstraction is modeled such that osProgress can not run forever. The idea is that it must change the state until it finally enables `osFinal`, i.e. the desired terminal state. Through the refinements, we model how exactly this happens, splitting `osProgress` into several events, and creating invariants and actions that model the OS requirements.

Some of the discharged POs guarantee that the events refining `osProgress` also give up control, and in Section 7.2 we prove they indeed modify the state such that it eventually reaches `osFinal`. Other POs prove that actions always respect the invariants (INV POs), or that a concrete event's actions do simulate the abstract correspondents (SIM POs). There are several other rules for PO generation, which we do not detail here. Table 2 summarizes the number of POs generated in each level of abstraction, and shows how many of them were automatically or manually discharged. The manually discharged ones are differentiated according to their discharging complexity: *simple* POs only required a few steps to be discharged, while the *complex* POs required more experience with the proving system and the PO's breakdown in several proving steps.

In Level 2, two invariants to guarantee that the save and load processes do save `oldCtx` and load the `runningTask`'s saved context:

```
m05.inv2:toSave = ∅ ⇔ saved = transform(oldCtx)
m07.inv4:toLoad = ∅ ⇔ loaded = invTransform(t_saved(runningTask))
```

Discharging the related INV POs proves that, for every refinement, when our model considers the old context as saved and the new context as loaded, they indeed are. Those INV POs were always automatically discharged, except in few refinements, where they were manually discharged in a few steps.

The SIM POs involving save and load actions, however, were rather complex, especially in Level 4. In particular for events `osManualEntry` and `osAutoExit`, we had to create a new parameter and a theorem in order to discharge the SIM POs. We detail here the proof strategy for the save action SIM PO in `osManualEntry`. The same strategy was applied to `osAutoExit`. We must prove that the action modeling the `osManualEntry` arrows in Figure 9 as described in Level 4, simulates its abstract correspondent in Figure 5a (Line 9):

Level	#POs	Auto	Simple	Complex
0	8	8	0	0
1	17	15	2	0
2	63	52	11	0
3	14	14	0	0
4	83	39	35	9
5	70	58	6	6
TOTAL	255	186	54	15
	100%	**73%**	**21%**	**6%**

Table 2: Number of POs discharged

```
m12.osManualEntry.act2: t_saved(runningTask) := t_saved(runningTask) <+ (
 autoSaveSet ◁ autoTempTEMPSVDtransform(temp)) <+(manualSaveSet ◁
 manualTransform(MANUALCTX ◁ oldCtx))
```

We replace the automatic save part of the action by the parameter `aux = autoSaveSet ◁ autoTempTEMPSVDtransform(temp)` and add to the event's guards the theorem `aux = autoSaveSet ◁ autoTempTransform(AUTOTEMPCTX◁ oldCtx)`. After proving the theorem, the SIM PO is much easier to discharge.

7.2 LTL Model Checking

For the liveness verification, we encode a set of LTL formulas that guarantee the specified execution order and that osFinal is eventually enabled. The model shall (1) eventually reach osFinal, staying there forever, (2) not reach a state where all events are disabled, (3) always have exactly one event enabled, and (4) implement the specified execution order: first, *entry*, then *body*, and finally *exit*, and manual save after auto save (OS7) and manual load before auto load (OS9).

Since the model's axioms are rather complex, we need to create a minimal set of CONTEXT and SAVEDCTX elements to represent the locations that compose contexts and saved contexts, otherwise the state space explodes and ProB cannot run. For this, we extend the Event-B contexts with the constant instantiations, and refine the machines we want to check. These machines are not modified any further, except for the model checks of Level 3 and 4, where the nondeterministic actions introduced in Level 3 would cause the checks to fail, since paths would exist in which osBody and osFinal could not be reached. As our intention is to leave this determinism to the architecture-specific models, the actions are modified to enforce the correct execution path. In Level 5, all actions are left unmodified, and we can check if the variables have been correctly set.

One error was found in Level 2: LTL finds a counterexample for reachability, so the model may never reach osFinal. An infinite loop is possible, because osBody does not decrease any variant and does not modify any variables that affect its guards. Thus, we introduce a new boolean variable osBodyRun, initialize it with FALSE, and add the guard osBodyRun = FALSE and an action osBodyRun := TRUE. A similar error was found when entryNothingToSave was introduced, prompting us to make it convergent and create a variant as explained in Level 4.

Model checking Level 4 also revealed that the execution order of events as not as intended: one formula fails because we forgot to strengthen osManualEntry's guards to require it to only be enabled after all AUTODIRECTSAVE elements have been saved, as required by OS7. The new guard AUTODIRECTSAVE $\cap$ toSave $= \varnothing$ forces this order. Similarly, osAutoExit may only execute after all MANUALCTX is loaded (OS9), thus the new guard MANUALCTX $\cap$ toLoad $= \varnothing$ was introduced.

With these modifications to the models and the discharging of all proofs, we prove that the requirements are fulfilled and the model is correct.

8 Conclusion and Future Work

We have presented the first step in our approach towards portability of embedded RTOS based on formal methods and code generation. We have shown a generic formal RTOS model in Event-B with context switches that decouples low-level functionality from hardware specifics. This allows us to reuse the model and its proofs for several architectures. Then, we instantiated the model for two architectures and verified them via interactive theorem proving and model checking. The safety and liveness verification of the models (1) proved that the generic model and its instantiations do not corrupt task contexts by having them properly saved and loaded; (2) proved that the kernel and the tasks run in the appropriate CPU states and privilege levels by having them properly changed; (3) proved that the kernel executes in the correct order and finishes execution.

To the best of our knowledge, this is the first time that OS low-level functionality is formally modeled for portability and verification. With the target-specific models, we can already generate significant parts of the OS assembly code for the MSP430 and RISC-V architectures, however this is still an ongoing work. Besides the code generation and automatic porting of low-level code, we are also working on the modeling and verification of additional aspects in the OS, such as security, timing, and energy consumption.

There is still much to do to make automatic porting a reality: Among other issues, the effort of modeling is not negligible, specially for the average software developer, who often lacks a background in formal methods. Besides, the correctness proofs can only be as good as the model itself, so the modeling process must be thorough. Additionally, modeling and verifying an entire OS, including all its low-level components, will require considerable effort. Nevertheless, it has been proved that formal modeling in software improves its quality and can reduce costs. Furthermore, architectures with completely different concepts would require the model to be adapted. While the effort must still be investigated, the hardware requirements of our current model should be fulfilled by most modern architectures. For maintainability, specially for porting, we expect that it will not only be beneficial, but also crucial within the IoT. The effort invested in modeling can be mitigated by increasing the number of ports and partially replacing testing by verification for guaranteed dependability during the development process. Another issue we must mention is the time and computation power required for model checking. The axioms in the presented model already cause state explosion in ProB if all registers available in the target architectures are included, which prompted us to create a minimal set for model checking. With bigger and more complex models, even a minimal set will eventually not avoid state explosion. We hope that advances in formal methods will eventually solve this problem. Other methods, such as TLA+, Isabelle/HOL, and HOL4 are potentially suitable for the model presented in this work, and should be investigated in future works.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. Alkhammash, E.H., Butler, M.J., Cristea, C.: International conference on communication, management and information technology, chap. Modeling guidelines of FreeRTOS in Event-B, pp. 453–462. CRC press (2017)
3. Baumann, C., Schwarz, O., Dam, M.: Compositional verification of security properties for embedded execution platforms. In: Kühne, U., Danger, J.L., Guilley, S. (eds.) PROOFS 2017. 6th International Workshop on Security Proofs for Embedded Systems. EPiC Series in Computing, vol. 49, pp. 1–16. EasyChair (2017). https://doi.org/10.29007/h4rv, https://easychair.org/publications/paper/wkpS

4. Boano, C.A., Römer, K., Bloem, R., Witrisal, K., Baunach, M., Horn, M.: Dependability for the Internet of Things—from dependable networking in harsh environments to a holistic view on dependability. e & i Elektrotechnik und Informationstechnik **133**(7), 304–309 (2016). https://doi.org/10.1007/s00502-016-0436-4, http://dx.doi.org/10.1007/s00502-016-0436-4
5. Borghorst, H., Bieling, K., Spinkczyk, O.: Towards versatile models for contemporary hardware platforms. In: 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'16). pp. 7–9 (July 2016)
6. Brandenburg, B.B.: The case of an opinionated, theory-oriented real-time operating system. NGOSCPS19 (04 2019)
7. Cheng, S., Woodcock, J., D'Souza, D.: Using formal reasoning on a model of tasks for FreeRTOS. Formal Aspects of Computing **27**(1), 167–192 (2014). https://doi.org/10.1007/s00165-014-0308-9
8. Craig, I.D.: Formal refinement for operating system kernels. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
9. Craig, I.D.: Formal models of operating system kernels. Springer Publishing Company, Incorporated, 1st edn. (2010)
10. Dalvandi, M., Butler, M., Rezazadeh, A., Salehi Fathabadi, A.: Verifiable code generation from scheduled Event-B models. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 234–248. Springer International Publishing (2018)
11. Danmin, C., Yue, S., Zhiguo, C.: A formal specification in B of an operating system. The Open Cybernetics & Systemics Journal **9**(1) (2015)
12. Dhote, S., Charjan, P., Phansekar, A., Hegde, A., Joshi, S., Joshi, J.: Using FPGA-SoC interface for low cost IoT based image processing. In: 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI). pp. 1963–1968 (Sep 2016). https://doi.org/10.1109/ICACCI.2016.7732339
13. Event-B: Event-B and the Rodin Platform, www.event-b.org
14. Fathabadi, A.S., Butler, M.J., Yang, S., Maeda-Nunez, L.A., Bantock, J., Al-Hashimi, B.M., Merrett, G.V.: A model-based framework for software portability and verification in embedded power management systems. Journal of Systems Architecture **82**, 12–23 (2018). https://doi.org/https://doi.org/10.1016/j.sysarc.2017.12.001, http://www.sciencedirect.com/science/article/pii/S1383762117305234
15. Frühwirth, T., Krammer, L., Kastner, W.: Dependability demands and state of the art in the internet of things. In: 2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA). pp. 1–4 (Sep 2015). https://doi.org/10.1109/ETFA.2015.7301592
16. General Dynamics C4 Systems: The seL4 microkernel (2016), https://sel4.systems/, accessed: 2020-02-05
17. Gomes, R.M., Baunach, M., Malenko, M., Ribeiro, L.B., Mauroner, F.: A co-designed RTOS and MCU concept for dynamically composed embedded systems. In: OSPERT'17 (2017)
18. Gomes, T., Pinto, S., Gomes, T., Tavares, A., Cabral, J.: Towards an FPGA-based edge device for the Internet of Things. In: 2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA). pp. 1–4 (Sep 2015). https://doi.org/10.1109/ETFA.2015.7301601
19. Goranko, V., Galton, A.: Temporal logic. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, winter 2015 edn. (2015), https://plato.stanford.edu/archives/win2015/entries/logic-temporal/

20. Hahm, O., Baccelli, E., Petersen, H., Tsiftes, N.: Operating systems for low-end devices in the Internet of Things: a survey. IEEE Internet of Things Journal **3**(5), 720–734 (Oct 2016). https://doi.org/10.1109/JIOT.2015.2505901
21. Hu, J., Lu, E., Holland, D.A., Kawaguchi, M., Chong, S., Seltzer, M.I.: Trials and tribulations in synthesizing operating systems. In: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. pp. 67–73. PLOS19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3365137.3365401, https://doi.org/10.1145/3365137.3365401
22. Instruments, T.: MSP430 ultra-low-power sensing and measurement MCUs (2019), http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview/overview.html
23. Jastram, M., Butler, P.M.: Rodin user's handbook: covers Rodin V.2.8. USA (2014)
24. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems **32**(1) (Feb 2014). https://doi.org/10.1145/2560537, http://doi.acm.org/10.1145/2560537
25. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering **SE-3**(2), 125–143 (March 1977). https://doi.org/10.1109/TSE.1977.229904
26. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. International Journal on Software Tools for Technology Transfer **10**(2), 185–203 (Mar 2008). https://doi.org/10.1007/s10009-007-0063-9, https://doi.org/10.1007/s10009-007-0063-9
27. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: specification. Springer Science & Business Media (2012)
28. Méry, D.: Modelling by patterns for correct-by-construction process. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Modeling. pp. 399–423. Springer International Publishing, Cham (2018)
29. Novikov, E., Zakharov, I.: Verification of operating system monolithic kernels without extensions. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. pp. 230–248. Springer International Publishing, Cham (2018)
30. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (Oct 1977). https://doi.org/10.1109/SFCS.1977.32
31. Popp, M., Moreira, O., Yedema, W., Lindwer, M.: Automatic HAL generation for embedded multiprocessor systems. In: Proceedings of the 13th International Conference on Embedded Software. EMSOFT '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2968478.2968493, http://doi.acm.org/10.1145/2968478.2968493
32. RISC-V Foundation: RISC-V, https://riscv.org/
33. Stoddart, B., Cansell, D., Zeyda, F.: Modelling and proof analysis of interrupt driven scheduling. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007: Formal Specification and Development in B. pp. 155–170. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
34. Su, W., Abrial, J.R., Pu, G., Fang, B.: Formal development of a real-time operating system memory manager. In: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS). IEEE (dec 2015). https://doi.org/10.1109/iceccs.2015.24

35. Syeda, H.T., Klein, G.: Formal reasoning under cached address translation. Journal of Automated Reasoning (jan 2020). https://doi.org/https://doi.org/10.1007/s10817-019-09539-7
36. Taivalsaari, A., Mikkonen, T.: A roadmap to the programmable world: software challenges in the IoT era. IEEE Software **34**(1), 72–80 (Jan 2017). https://doi.org/10.1109/MS.2017.26
37. Verhulst, E., Boute, R.T., Faria, J.M.S., Sputh, B., Mezhuyev, V.: Formal development of a network-centric RTOS. Springer US (2011). https://doi.org/10.1007/978-1-4419-9736-4
38. Waterman, A., Asanović, K.: The RISC-V instruction set manual volume I: user-level ISA version 2.2 (May 2017), https://riscv.org/specifications
39. Waterman, A., Lee, Y., Avizienis, R., Patterson, D.A., Asanović, K.: The RISC-V instruction set manual volume II: privileged architecture version 1.7. Tech. Rep. UCB/EECS-2015-49, EECS Department, University of California, Berkeley (May 2015), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-49.html
40. Wright, S.: Formal construction of instruction set architectures. Phd thesis, University of Bristol (2009), http://www.cs.bris.ac.uk/Publications/Papers/2001121.pdf
41. Wright, S.: Automatic generation of C from Event-B. In: Workshop on integration of model-based formal methods and tools. p. 14 (2009)