Formal Modeling and Verification of Low-Level AUTOSAR OS Specifications: Towards Portability and Correctness

Vignesh Manjunath
Pro2Future GmbH
Graz, Austria
vignesh.manjunath@pro2future.at

Tobias Scheipel
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Marcel Baunach
Graz University of Technology
Graz, Austria
baunach@tugraz.at

Abstract

Embedded automotive software development increasingly uses formal methods to guarantee correctness and dependability. Generally, we can use models of high-level (hardware-independent) specifications to generate high-level code. However, we need to model low-level (hardware-dependent) specifications individually for each hardware architecture, which significantly increases modeling complexity and verification effort and limits the re-usability of these low-level models to generate hardware-specific code. The problem is even more significant when dealing with complex low-level functionalities (e.g., context switch, system initialization) of an AUTOSAR Operating System (OS).

In this paper, we propose a refinement-based formal modeling approach to address this problem. First, we use UML-B and Event-B formal methods to implement a hardware-generic model of the low-level specification, which abstracts hardware-specific details. Next, we refine the generic model by including the hardware-specific details in the final refinements. We verify the hardware-generic and -specific models using theorem proving and Linear Temporal Logic (LTL) model checking. This way, our approach facilitates portability and re-usability while ensuring the correctness of low-level functionalities. As proof of concept and evaluation, we model and verify the context switch of an AUTOSAR OS and refine the hardware-generic model of the use case for AURIX TriCore and RISC-V architectures.

CCS Concepts

• **Computer systems organization → Embedded systems**; **Real-time operating systems**; *Reliability*; *Maintainability and maintenance*; • **Software and its engineering → Formal software verification**.

Keywords

Formal methods, Verification, AUTOSAR, OS Portability

ACM Reference Format:
Vignesh Manjunath, Tobias Scheipel, and Marcel Baunach. 2025. Formal Modeling and Verification of Low-Level AUTOSAR OS Specifications: Towards Portability and Correctness. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3672608.3707867

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707867

Artifacts Availability

The Event-B models, requirement specifications and use case analysis documents, design diagrams, and the LTL formulas are available in the artifacts at https://dx.doi.org/10.6084/m9.figshare.26963377.v1. In this paper, we omit intricate model details for better readability. In such cases, the symbol 🗎 indicates that details can be found in the artifacts.

1 Introduction

The increasing complexity of automotive software motivates the use of verification methods to guarantee dependability and correctness. Therefore, modern automotive software development increasingly uses formal methods to mathematically model and verify the software for correctness against its requirement specifications.

In general, we can model and verify high-level or hardware-independent functionalities and use the models to generate high-level code that is independent of hardware architectures. However, since low-level functionalities are hardware-dependent, we must model their behavior individually for each target architecture, which significantly increases the modeling complexity and verification effort and limits the re-usability of low-level models to generate hardware-specific code. This limitation reduces the overall portability of the Operating System (OS). The problem is even more significant when dealing with the low-level functionalities of an AUTOSAR OS, which are complex and safety-critical. We formulate this problem through the following Research Questions (RQn) and address them in our paper:

- RQ1 How can we formally model the behavior of a complex low-level AUTOSAR OS functionality while facilitating its portability to different hardware architectures?
- RQ2 How can we verify the resulting formal models to ensure correctness w.r.t. the requirement specifications?

To address these research questions, we propose a refinement-based formal modeling approach. First, we systematically analyze the given low-level requirement specification in natural language and design the specification as a Finite State Machine (FSM). In addition, we logically derive Linear Temporal Logic (LTL) formulas [39] from the requirements. Second, we implement a hardware-generic model of the low-level specification as an FSM using UML-B [43] and Event-B [1]. The hardware-generic FSM represents the low-level functionality while abstracting the hardware-specific details. This way, we can reuse the hardware-generic model across different hardware architectures and facilitate portability (addressing RQ1). Next, we refine the hardware-generic FSM by including the hardware-specific details in the final refinements. We use LTL model checking and theorem proving to verify the models at their respective stages of implementation/refinement (addressing RQ2). Then, we can use the hardware-specific FSM to generate the hardware-specific code. The process of generating code from the hardware-specific models and modeling of high-level specifications (e.g., scheduler, system call implementation) are beyond the scope of this paper.

Contribution. We propose an approach to formally model and verify low-level specifications of an AUTOSAR OS while facilitating portability. As a proof of concept and evaluation, we model and verify the context switch functionality of EB Tresos AutoCore OS [20] (an industrial classic AUTOSAR OS), and refine the hardware-generic model for AURIX TriCore [5] and RISC-V [26] architectures. Our main contributions in this paper are: (1) Design and implementation of a hardware-generic formal FSM model of the AUTOSAR OS context switch functionality; (2) Structured derivation of LTL formulas from the given requirement specifications; (3) Verification of functional correctness of the formal model using LTL model checking and theorem proving; (4) Verification of *safety*, *liveness*, and *reachability* properties using LTL formulas; (5) Two hardware-specific refinements of the hardware-generic model for AURIX TriCore and RISC-V architectures.

Outline. We organize the remainder of this paper as follows: Section 2 gives a general background on AUTOSAR OSs, the formal methods, and the hardware architectures used in our approach. In Section 3, we demonstrate the underlying modeling and verification methodology using the context switch functionality and present the verification results. Section 4 compares related work, and in Section 5, we discuss the benefits and limitations of our work. Section 6 summarizes the paper and provides an outlook on future work.

2 Background

In this section, we provide an overview of AUTOSAR OSs, a brief background on the formal methods and tools we use in our approach, and an insight into the AURIX TriCore and RISC-V hardware architectures.

2.1 AUTOSAR OS

An AUTOSAR OS complies with the specifications defined by classic AUTomotive Open System ARchitecture (AUTOSAR) [7] and is designed to meet the real-time and safety-critical requirements of automotive applications. Among many other features, the OS implements preemptive and priority-based scheduling of concurrent tasks and allows multiple activations of tasks. In our proof of concept, we refer to a specific implementation of AUTOSAR OS, the EB Tresos AutoCore OS.

A context switch in AutoCore OS is triggered on a system call (abbreviated as *syscall*) invocation. The kernel context switch functionality considers the task state (e.g., SUSPENDED, TERMINATING, READY, RUNNING, WAITING) and conditionally saves/restores the task's context [7, 37]. For example, when a task chains itself (i.e., activates a new instance of the task), the kernel need not save the task context. In addition, the functionality includes kernel mode handling, task state and priority management, interrupt enable/disable, and kernel stack management.

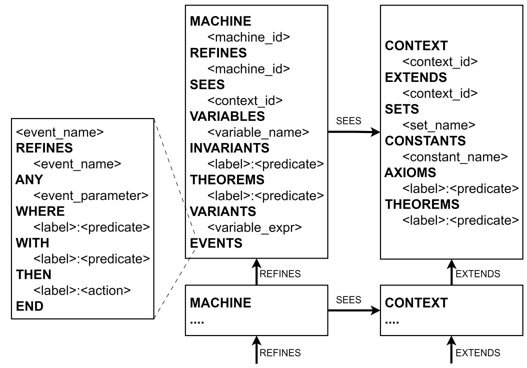

Figure 1: Comprehensive Event-B modeling notation.

2.2 Event-B

Event-B [1] is a formal method based on set theory, abstraction, and refinement. First, an abstract model of the system is implemented and then incrementally refined to a concrete model. We leverage this refinement-based modeling technique to enhance the re-usability of low-level models.

The Rodin Platform [2], an Eclipse IDE, supports to implement and verify Event-B models. Rodin automatically generates mathematical Proof Obligations (POs) for the models and partially discharges (i.e., satisfies) them using theorem provers. POs are mathematical formulas that need to be discharged to ensure the correctness of the model. We use Rodin plugins like ProB [31], AtelierB [30], and SMT solvers [16] such as Z3 and CVC3 to interactively satisfy the undischarged POs.

A model in Event-B comprises *contexts* and *machines*. We will refer to a *context* and *machine* in Event-B with the terms *Event-B context* and *Event-B machine*, respectively. An Event-B context represents static information about the system, i.e., carrier sets, constants, axioms, and theorems. An Event-B machine represents dynamic information about the system, i.e., variables, variants, and events, and static information, i.e., invariants and theorems that must hold for the variables throughout the model. When an Event-B machine *sees* an Event-B context, the machine can use the modeling elements of the Event-B context (e.g., constants, axioms) to relate static and dynamic information as well as to discharge POs. Variables in an Event-B machine represent the system state, and events specify state transitions. An event is enabled when a state satisfies an event's guard conditions, and the event's actions modify the state.

Model Refinement. When an Event-B machine *refines* an abstract machine, it must refine or retain the events. We can also extend functionality by adding more variables, new events, or strengthening guard conditions. Rodin automatically generates POs to ensure that the refined machine does not contradict the abstract one (e.g., the refined machine preserves the invariants in the abstract machine). When an Event-B context *extends* an abstract context, we can add more carrier sets, constants, axioms, and theorems or refine the elements in the abstract context (e.g., define the members of an abstract set).

Figure 1 represents a comprehensive Event-B modeling notation with machine-context relationship. The `WHERE` section of an event contains its guard conditions, the `THEN` section contains its actions, and the `WITH` section provides *witness* (gluing predicates) for the disappearing abstract variable or an event parameter. The `INITIALIZATION` event initializes the variables of an Event-B machine and contains only `WITH` and `THEN` sections. Further details on Event-B *constructs* and notations (including set theory notations) are available in [18, 36].

2.3 UML-B

UML-B [43] is an extension of Event-B with a UML-like diagrammatic formal modeling notation and supports incremental refinement and verification by proof (via translation to Event-B). UML-B is integrated with Rodin (as iUML-B state machine plugin [42]) and provides a graphical front-end and automatic translation of UML-B models into Event-B. In addition, UML-B supports the graphical arrangement of events within the model, which simplifies event ordering and helps to avoid errors that can occur when textually ordering the events using Event-B alone.

In our approach, we use a UML-B state machine diagram to model the behavior of low-level specifications as an FSM. The states in the FSM are represented as enumerated values in Event-B, and a state machine variable manages these states. The transitions between the states are translated into events.

Model Refinement. Each Event-B machine corresponds to only one state machine, and when the machine refines, the state machine also refines. The refined state machine can retain the states and events or add new events and nested state machines but it cannot add new states.

2.4 LTL Model Checking

Linear Temporal Logic (LTL) [39, 44] is a widely used temporal logic to describe properties of linear-time sequences. It operates on the principle that at each time instant, there is only one possible future timeline. LTL formulas compose a finite set of atomic propositions, the logical operators (e.g., $\neg$, $\wedge$, $\vee$), and the temporal operators (e.g., **G** (globally $\square$), **X** (next $\bigcirc$), **F** (finally $\lozenge$)).

LTL formulas express properties such as *safety* (i.e., given a precondition, a system never enters an unsafe state), *reachability* (i.e., a particular state is reachable from a given state), and *liveness* (i.e., given a precondition, a desirable state will eventually be reached). We use ProB [31], a model checker and a constraint solver integrated into Rodin Platform for LTL model checking.

2.5 AURIX TriCore Architecture: TC3xx

AURIX TriCore [6] is a widely used hardware architecture in embedded automotive systems. It fulfills safety-critical requirements and supports achieving a high level of dependability of automotive software. We refer to TC3xx [4, 5], a 32-bit AURIX TriCore family of microcontrollers. The General Purpose Register (GPR) file in TC3xx contains 16 32-bit address registers (A0-A15) and 16 32-bit data registers (D0-D15). The CPU context comprises the GPRs, Program Counter (PC), Program Status Word (PSW), and Previous Context Information (PCXI). The context is subdivided into upper context (A10-A15, D8-D15, PSW, and PCXI) and lower context (A2-A7, D0-D7, A11, and PCXI).

2.6 RISC-V Architecture

RISC-V [25] is a load-store architecture with an open standard Instruction Set Architecture (ISA), increasingly used in the automotive industry due to its flexibility and scalability. We refer to the base RV32I [26] implementation with no extension. The RV32I instruction set includes basic 32-bit integer instructions and supports user and machine modes. The GPR file in RV32I contains 32 registers (X0-X31 of 32-bit each), and the CPU context comprises the GPRs, a PC, and a set of Control and Status Registers (CSRs) (e.g., MEPC, MSTATUS) with special access instructions to modify, e.g., the interrupt enable state in defined privilege levels.

3 Methodology and Evaluation

In this section, we present and demonstrate the underlying modeling and verification methodology using the context switch functionality of the AUTOSAR OS. The context switch is a complex, safety-critical, and real-world low-level use case, which makes it suitable for us to demonstrate and evaluate our methodology.

3.1 Context Switch Requirements

Based on the AUTOSAR OS specifications and AutoCore OS internal documentation, we manually derive the following requirement specifications (OSn) of the context switch functionality and its subparts (Z):

- OS1 (A) When a task is interrupted, its context data must be saved in the memory locations reserved for context saving. When saving a task's context, manually save only the part of the context that is not automatically saved by the hardware. (B) When a task terminates or completes its execution or chains itself or will continue its execution, the context save is not required. (C) When there is no running task, the context save is not required.
- OS2 (A) Before a task is resumed, its saved context data must be loaded from the respective memory locations into the CPU context. When restoring a task's context, manually restore only the part of the context that is not automatically restored by the hardware. (B) Context initialization is required when a task starts a new execution.
- OS3 When a task is interrupted and the same task is resumed, context save (defined in OS1(A)) and restore (defined in OS2(A)) are not required.
- OS4 When a task triggers a syscall, the task execution is interrupted, and the kernel is invoked. The kernel is also invoked when a task is completed or terminated.
- OS5 Once the kernel is invoked, the kernel entry saves a part of the context that might be modified by the kernel (e.g., callee-saved registers), saves the caller stack, switches to the kernel stack, and calls the corresponding kernel function of the syscall. For a preemptive kernel in AUTOSAR, the kernel entry must save the previous kernel mode, then set the current kernel mode to active, and enable interrupts before invoking the kernel function of the syscall.
- OS6 (A) On return from syscall, if the previous kernel mode is not active, then disable interrupts, and based on OS1 and OS3 conditions, the kernel body saves the context, including the part of the context saved in OS5, the syscall result, and

proceeds to Kernel Exit. (B) On return from syscall, if the previous kernel mode is active, then restore the previous kernel mode and proceed to Kernel Exit.

OS7 (A) Once in Kernel Exit, when the previous kernel mode is active, restore the part of the context saved in OS5, prepare the switch to the caller stack, dispose of the kernel context, return from the trap handler, and resume the execution of a task. (B) Once in Kernel Exit, when the previous kernel mode is not active, and there is a valid task to resume, based on OS2 and OS3, either execute the context restore/initialization or just restore the part of the context saved in OS5, prepare the switch to the caller stack, dispose of the kernel context, return from the trap handler, and resume the execution of a task. (C) Once in Kernel Exit, when the previous kernel mode is not active, and there is no task to dispatch, execute the idle loop. (D) Once in Kernel Exit, when the previous kernel mode is not active, the current running task is valid, and the task stack is out of bounds, proceed to Error State.

3.2 Hardware-generic Assumptions

With respect to the requirement specifications (OSn), we define the following generic hardware assumptions (HWn) to help us realize a hardware-generic model for the context switch. The assumptions represent a basic set of hardware features necessary for the context switch functionality, and the target architecture must satisfy these assumptions.

HW1 The architecture defines a subset of locations (e.g., registers) as context, and the CPU uses/modifies the context data during code execution.

HW2 The architecture provides mechanisms for saving and restoring context data from referable locations such as registers or memory addresses.

HW3 Before the CPU serves a trap, the data of a part of the context are automatically saved into locations defined by the architecture.

HW4 The architecture provides instructions that automatically restore the context with the data that were automatically saved during HW3, e.g., "return from trap" or "return from exception" instructions.

HW5 When the code execution is interrupted, the architecture provides instructions to manually store the context data in a set of well-defined memory locations, e.g., "store" instructions.

HW6 To resume the code sequence from the same point, the architecture provides instructions to manually load the context with the data from the respective memory locations where it was saved during HW5, e.g., "load" instructions.

HW7 The architecture provides an interrupt enable flag, and the interrupts will only be served if the interrupt enable flag is set.

HW8 The architecture has a stack pointer register that points to the top of the current stack.

For every hardware assumption (HWn), we identify the relevant TC3xx and RISC-V architecture specifications (🗎). We use these hardware specifications to refine the hardware-generic model.

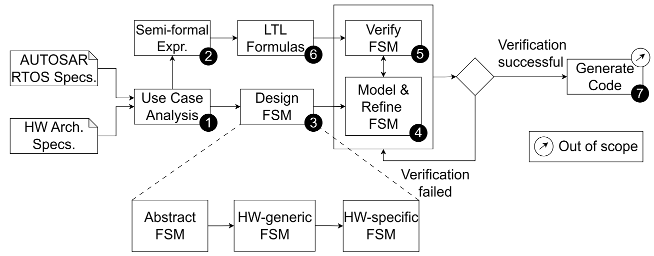

Figure 2: Overview of the proposed approach.

3.3 The Overall Approach

Figure 2 illustrates an overview of the proposed approach and the No. denote the sequence of steps involved. In 1, we use the specifications (OSn), the assumptions (HWn), and the corresponding hardware specifications to perform use case analysis. Corresponding to every (OSn), we first derive the prerequisite conditions and the possible use cases. Second, we determine the necessary condition checks and actions that an (OSn) specifies. The actions represent the hardware-generic updates w.r.t. (HWn). Next, we determine the required hardware-specific updates based on the identified target hardware specifications. Third, we logically transform the hardware-generic and -specific updates as post-condition checks.

Next, in 2, we use the use case analysis to logically derive semi-formal expressions, which are formalized natural language requirements of the pattern: It is always the case that if *prerequisite* and *condition check* holds, then next/eventually *post-condition check* holds as well. Table 1 shows the results of a use case analysis for the (OS1(A)) specification and its corresponding semi-formal expression. For clarity, we use C-like expression notation for the condition checks, updates, and semi-formal expressions.

In 3, we use the results of the use case analysis to design the context switch functionality as an FSM. We first choose the base set of states and transitions and create an abstract FSM design representing the overall functional flow. The state transitions in the FSM perform all condition checks and actions, and choosing the number of states is a design choice. Next, we elaborate on the abstract FSM design into a detailed hardware-generic FSM design, where the guard conditions of the state transitions correspond to *prerequisite* and *condition checks*, and the actions correspond to hardware-generic updates derived in 1. We add/modify the hardware details based on the hardware-specific updates and the corresponding target hardware specifications and derive the hardware-specific FSM design. For traceability and completeness of the designs, we map (annotate) the state transitions and actions to the corresponding requirement specifications and their use cases.

In 4 and 5, we model and verify the designed FSMs. Modeling and verification is an iterative process where, for every iteration, the model gets more refined (i.e., with more details and functionality). For every model refinement, we satisfy the undischarged POs generated by Rodin to verify that the refinement is correct. We proceed with further model refinement only after discharging all the POs.

In 6, we logically deduce LTL formulas from the semi-formal expressions (derived in 2) by substituting model variables and their

Table 1: Example of a use case analysis.

Requirement	OS1(A)
Prerequisite	RunningTask execution is interrupted (i.e., RunningTask is not NULL)
Use case	When the TaskQHead is not same as the RunningTask and the RunningTask is not in TERMINATING or SUSPENDED state
Condition check	TaskQHead != RunningTask && RunningTask.state != TERMINATING && RunningTask.state != SUSPENDED
Hardware-generic update	1. Save the context of the RunningTask (Ref.: HW5) 2. If RunningTask.State == RUNNING then RunningTask.State = READY, else, do nothing
TriCore-specific update	Save only lower context (Ref.: TC_HW5) and save the PCXI register data to the task. (Ref.: TC_HW2)
Hardware-generic post-condition check	1. RunningTask's saved context == CPU context data 2. RunningTask.state != RUNNING
TriCore post condition check	RunningTask.PCXI == CPU.PCXI
Semi-formal expression	It is always the case that if (RunningTask != NULL && TaskQHead != RunningTask && RunningTask.state != TERMINATING && RunningTask.state != SUSPENDED) holds, then eventually (RunningTaskSavedContext == CPUContext && RunningTask.state != RUNNING) holds as well
Hardware-generic LTL formula	**G**({rTask ≠ NULLTASK ∧ taskQHead ≠ rTask ∧ taskState(rTask) ≠ ts_TERMINATING ∧ taskState(rTask) ≠ ts_SUSPENDED} ⇒ **F**({t_saved(rTask) = transform(cpuCtxt) ∧ taskState(rTask) ≠ ts_RUNNING}))

Note: TaskQHead refers to the task at the head of the ready queue (i.e., task to be dispatched), RunningTask (or rTask) refers to the currently running task, and RunningTask.PCXI represents the running task's saved PCXI register data. In the LTL formula, `taskState(rTask)` represents the state of the running task, `t_saved(rTask)` represents the running task's saved context data, and `cpuCtxt` represents the current CPU context data. The model is discussed in detail in Section 3.4.

abstractions. The hardware-generic LTL formula for the (OS1(A)) use case is shown in Table 1. In addition, we define *safety*, *reachability*, and *liveness* properties (refer to Section 2.4) based on the (OSn) requirements and derive the LTL formulas that express these properties. Further details about these properties are provided in Section 3.5. Next, we use these LTL formulas to verify the hardware-generic model. Then, we include hardware-specific post-condition checks and adapt the LTL formulas to verify the hardware-specific model. If the LTL check fails or we are unable to discharge the POs, then we need to modify the model/refinements (or even change the design) accordingly.

Once all the POs and LTL formulas are satisfied, we can use the hardware-specific model to generate code in 7. The code generation process is out of the scope of this paper. In the following section, we provide a detailed explanation of the modeling and verification process.

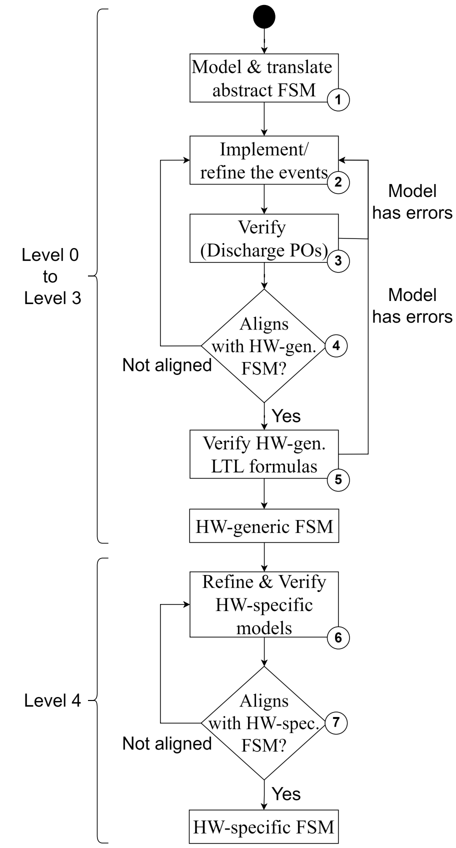

Figure 3: Modeling and verification process.

3.4 Modeling and Verification

In this work, we model and verify only the software part of the low-level context switch functionality. We abstract high-level functionalities such as kernel syscall functions, idle loop, and error handler in the model. In addition, we model the hardware context save/restore part (refer to HW3 and HW4) to verify if the complete task context is correctly saved and restored.

Our modeling process involves the following five levels of refinement from Level 0 to Level 4. In Level 0 to Level 3, we model the hardware-generic FSM, and in Level 4, we include hardware details and refine the model into a hardware-specific FSM.

- **Level 0** models the abstract FSM, basic sets (e.g., context, saved context), and their relations using UML-B and Event-B. In addition, the level models the abstract context save and restore (hardware and manual save/restore).
- **Level 1** introduces the running task (refers to the currently running task) and refines the abstract context save and restore to/from the task-specific memory.
- **Level 2** models the volatile context save and restore. The volatile context is a subset of the context that might be modified by the kernel (e.g., callee-saved registers).
- **Level 3** models the task queue head, task state and priority changes, kernel mode management, syscall kernel function invocation, and kernel stack switching. In addition, we map (annotate) the events to the corresponding requirement specifications

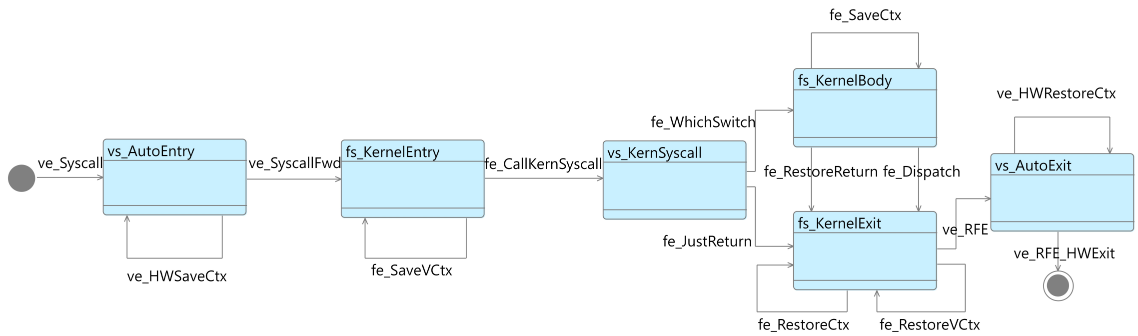

Figure 4: Abstract FSM in UML-B.

and their use cases for traceability. Note: the task queue head (or taskQHead) refers to the task at the head of the OS's ready queue.

- **Level 4** refines the model into the hardware-specific FSM by including hardware-specific details (e.g., CPU registers).

Now we explain the modeling and verification process, where Figure 3 illustrates this process as a sequence of steps (denoted by No.). The designs (created in the step 3, Figure 2) serve as the input to our modeling process. Steps 1 to 5 correspond from Level 0 to Level 3 model refinements and steps 6 to 8 correspond to Level 4 hardware-specific model refinement.

In 1, we model the abstract FSM design in UML-B and translate it into Event-B (via the iUML-B plugin; refer to Section 2.3). The UML-B abstract FSM model is shown in Figure 4 and represents the overall functional flow (from `vs_AutoEntry` to `vs_AutoExit`). The blue boxes are states, and the directed arrows are events (in Event-B). Table 2 provides the event descriptions and maps them to the corresponding specifications (OSn) and assumptions (HWn). Further details of the events are available in the artifacts (🗎). We classify the events into two types: (1) functional events (with prefix `fe_`) that represent functional logic, and (2) virtual events (with prefix `ve_`) that are model-only elements without any functional logic. Similarly, we also classify the states into functional states (with prefix `fs_`) and virtual states (with prefix `vs_`). We primarily use the virtual states and events to model abstractions of the high-level functionalities that interact with the context switch (e.g., idle loop). In addition, the virtual events help in modeling the hardware behavior relevant to the context switch functionality (e.g., the `ve_HWSaveCtx` event represents the hardware context save (HW3)).

Next, in 2, we create the basic sets and their relations required for modeling the context switch. For this, we adapt the basic sets and their relations from an existing work [21]. For an easy understanding of the abstraction and refinement concept, we explain only the refinement corresponding to the hardware context save (HW3). The complete UML-B and Event-B models are available as artifacts (🗎).

Basic Sets. We create the set `LOCATION`, an abstract representation of memory addresses and registers. We define the `LOCATION` as three non-overlapping subsets namely: (1) `CONTEXT` representing the context defined in (HW1), (2) `SAVEDCTX` representing the set of locations reserved for saving the context data (OS1(A)), and (3) `SVDVOLATILECTX` representing the set of locations reserved for saving the volatile context data (OS5). The cardinality of `CONTEXT` and `SAVEDCTX` are the same. To decouple the hardware-managed (HW3 and HW4) from the manually-managed context part, we define $\mathtt{CONTEXT} = \mathtt{AUTOCTX} \cup \mathtt{MANUALCTX}$. Similarly, we also define $\mathtt{SAVEDCTX} = \mathtt{SVDAUTOCTX} \cup \mathtt{SVDMANUALCTX}$. We define `DATA`, a subset of $\mathbb{Z}$ (integer values) to represent the data in these locations (e.g., $\mathtt{CONTEXT} \longrightarrow \mathtt{DATA}$). In addition, we create a set called `TASKS` to represent the task set within the system. This set includes a `NULLTASK` (invalid task) and a set of valid tasks known as `VTASKS`.

Set relations. To save the context, we convert (using bijection operator ($\rightarrowtail\!\!\!\twoheadrightarrow$)) the context data to the saved context data using the function $\mathtt{transform} \in (\mathtt{CONTEXT} \longrightarrow \mathtt{DATA}) \rightarrowtail\!\!\!\twoheadrightarrow (\mathtt{SAVEDCTX} \longrightarrow \mathtt{DATA})$. We split the `transform` function into $\mathtt{autoTransform} \in (\mathtt{AUTOCTX} \longrightarrow \mathtt{DATA}) \rightarrowtail\!\!\!\twoheadrightarrow (\mathtt{SVDAUTOCTX} \longrightarrow \mathtt{DATA})$ for converting hardware-managed context data and $\mathtt{manualTransform} \in (\mathtt{MANUALCTX} \longrightarrow \mathtt{DATA}) \rightarrowtail\!\!\!\twoheadrightarrow$

Table 2: Event descriptions and specifications mapping.

Events	Description	Specifications
ve_Syscall	Syscall invocation	OS4
ve_HWSaveCtx	Save the automatic context	HW3
ve_SyscallFwd	Connecting transition	OS4
fe_SaveVCtx	Save volatile context	OS5
fe_CallKernSyscall	Compute syscall address and invoke syscall	OS5
fe_WhichSwitch	Save the syscall result and check save context conditions	OS6 (A)
fe_SaveCtx	Save the manual context	OS6 (A), OS1 (A), HW5
fe_JustReturn	Save the syscall result and restore previous kernel mode	OS6 (B)
fe_RestoreReturn	Connecting transition	OS6 (A), OS3
fe_RestoreVCtx	Restore volatile context	OS7 (A) (B), OS3
fe_Dispatch	Check valid dispatch task	OS7 (B)
fe_RestoreCtx	Restore the manual context	OS7 (B), OS2 (A), HW6
ve_RFE	Connecting transition	HW4
ve_HWRestoreCtx	Restore the automatic context	HW4
ve_RFE_HWExit	Connecting transition	HW4

```
1  ve_HWSaveCtx
2  ANY
3   autoSaveSet
4  WHERE
5   autoSaveSet ⊆ toSave
6   autoSaveSet ⊆ SVDAUTOCTX
7   autoSaveSet ≠ ∅
8   ...
9  THEN
10  saved := saved ∪ (autoSaveSet ◁
        autoTransform(AUTOCTX ◁ cpuCtxt))
11  toSave := toSave\autoSaveSet
12  ...
13 END
```

Listing 1: Abstract ve_HWSaveCtx event.

(SVDMANUALCTX ⟶ DATA) for converting manually-managed context data. Similarly, to restore the context, we define `invTransform`, the inverse of the `transform` function, and split the function into `autoInvTransform` and `manualInvTransform`. We define these relations as axioms in the Event-B context.

Abstract hardware context save. In the initial abstraction (Level 0), we create `toSave` and `saved` variables in the Event-B machine, where `toSave` ⊆ SAVEDCTX keeps track of the context yet to be saved and `saved` ∈ SAVEDCTX ⇸ DATA represents the context data that is saved already. The `cpuCtxt` variable represents the CPU context data, i.e., `cpuCtxt` ∈ CONTEXT ⟶ DATA. Initially, `toSave` = SAVEDCTX and `saved`=∅.

The ve_HWSaveCtx (Listing 1) event saves the AUTOCTX data. The context save or restore is not always a single action. For example, in TriCore, we can save (lower/upper) context in one single action (e.g., using SVLCX instruction). However, in RISC-V, we need to save one register at a time (e.g., using `store` instruction). Therefore, we use an event parameter `autoSaveSet` (Line 3) to save only a subset of the auto (HW3) context data at a time (Lines 5 & 6). Line 10 in the listing models the abstract save action in 2 steps: (1) The `autoTransform` function extracts (using the domain restriction operator (◁)) the data from the AUTOCTX part of the `cpuCtxt` and converts the data into SVDAUTOCTX data; (2) Only a subset of the SVDAUTOCTX data (restricted by `autoSaveSet`) is added to the `saved` variable. Then, Line 11 deducts the saved subset (`autoSaveSet`) from the `toSave` variable. This saving process repeats until `autoSaveSet` = ∅ (Line 7). In other words, the hardware context save process ends when the AUTOCTX part of the `cpuCtxt` is completely saved.

Refined hardware context save. In Level 1, we define the `rTask` variable to represent the currently running task and the `t_saved` variable to represent the valid tasks' saved context data, i.e., `rTask` ∈ TASKS, and `t_saved` ∈ (TASKS \ {NULLTASK}) ⟶ (SAVEDCTX ⟶ DATA). Then, we replace the `saved` variable (from Level 0) and refine Line 10 in Listing 1 to:

```
t_saved(rTask) := t_saved(rTask) ⊲- (autoSaveSet ◁
autoTransform(AUTOCTX ◁ cpuCtxt))
```

The action: (1) extracts (using ◁) data from the AUTOCTX part of the `cpuCtxt` and converts the data into SVDAUTOCTX data; (2) replaces the data (using the relational overriding operator (⊲-)) in the SVDAUTOCTX part of the running task's saved context with a subset of the converted data (restricted by `autoSaveSet`). In addition, we also define a new guard condition to ensure that the save action is performed only if the running task is valid, i.e., `rTask` ∈ VTASKS.

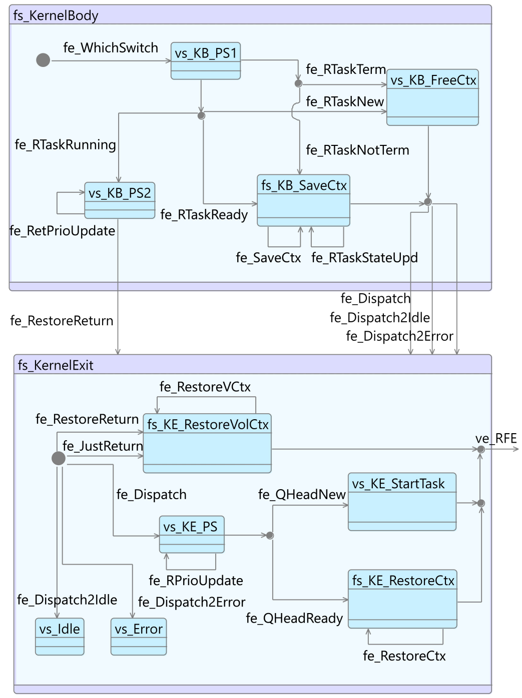

Figure 5: Refined Kernel Body and Kernel Exit states with nested state machines.

We follow the same abstraction and refinement concept for modeling the hardware context restore, the volatile context save and restore, and the manual context save and restore. In Level 3, we refine the abstract FSM and implement nested state machines inside the `fs_KernelBody` and `fs_KernelExit` states (using UML-B), where Figure 5 shows these refined states. We introduce states `fs_KB_SaveCtx`, `fs_KE_RestoreCtx`, and `fs_KE_RestoreVolCtx` and relocate the manual context save event (`fe_SaveCtx`), the manual context restore event (`fe_RestoreCtx`), and the volatile context restore event (`fe_RestoreVolCtx`) to these respective states. The relocation ensures that the save and restore processes are completed before any subsequent events are enabled. In addition, we use virtual Pseudo-States (PS) like `vs_KB_PS1` and `vs_KB_PS2` in `fs_KernelBody` and `vs_KE_PS` in `fs_KernelExit` to serve as synchronization points. Next, we translate the refined UML-B model and implement the remaining context switch functionalities (e.g., task state, kernel mode, etc.). We provide a comprehensive list of Event-B machine variables and their descriptions that model these functionalities in the artifacts (🗎).

Model Verification. We implement 12 model refinements (three for each level) to realize the hardware-generic FSM model. For every model refinement, we verify all POs generated by Rodin in (3) (refer to Figure 3). If the verification fails and the model contains errors, we modify the models to correct these errors and satisfy the undischarged POs. When the verification is successful, we map (annotate) the events to the corresponding use case and requirement specifications (OSn). Next, in (4), we use the annotations to check if the current model and the hardware-generic FSM design align, i.e., if all the actions and guard conditions in the design are modeled. In addition, the check helps us ensure that each event's guard conditions represent the *prerequisite* and *condition checks* and its actions represent the *hardware-generic updates* of the mapped use case and requirement. If the model and the design do not align, we must implement/refine and verify (i.e., discharge all POs) the missing components, and once they align, the hardware-generic FSM model is complete.

In ⑤, we use the LTL formulas derived in step ❻ (refer to Figure 2) to verify the correctness of the hardware-generic model w.r.t. (OSn) specifications. If the model has errors during verification, we modify the hardware-generic model to fix the errors and repeat the steps from ② to ⑤. If the model has no errors, the verification succeeds and we can refine the hardware-generic model into the target hardware-specific model.

Hardware-specific refinement. In ⑥, for each architecture, we extend the Event-B context and define the `LOCATION` set with the CPU registers (identified w.r.t. HW1 and the architecture specifications). Subsequently, we also define which registers belong to `MANUALCTX` and `AUTOCTX`. In the TriCore-specific model, we define the upper context registers (refer to Section 2.5) in `AUTOCTX` since the CPU automatically saves the upper context when serving a trap (triggered on a syscall invocation). We need to manually save the lower context registers during a context switch so the registers belong to `MANUALCTX`. In the RISC-V model, since the CPU saves PC in MEPC CSR and the previous interrupt enable state and privilege mode in MSTATUS CSR on a trap, we need to manually save MEPC and MSTATUS CSRs along with the GPRs (refer to Section 2.6). Therefore, these registers belong to `MANUALCTX` and `AUTOCTX`=∅.

In addition, the context switch functionality involves modifications to special function registers. For example, in TriCore, we enable/disable global interrupts by setting/clearing the `IE` bit of the Interrupt Control Register (ICR). Therefore, we create and define a set representing the special function registers relevant to the context switch functionality. Accordingly, we create an Event-B machine variable in the hardware-specific model to handle the special function registers' data. We apply two model refinements to the hardware-generic FSM to implement the TriCore-specific FSM and one model refinement to implement the RISC-V-specific FSM. The TriCore-specific and RISC-V-specific models for the context switch functionality are available in the artifacts (🗎).

We discharge all POs generated by Rodin for the hardware-specific model refinement, and then we use the hardware-specific LTL formulas to verify the hardware-specific models. Once the verification succeeds and the model has no errors, we map (annotate) the events to the corresponding use case and requirement specifications (OSn). Similar to ④, in ⑦, we use the annotations to check whether the hardware-specific model and the hardware-specific FSM design align. If they do not align, we repeat ⑥ to model the missing components, and once they align, the hardware-specific FSM model is complete.

3.5 Verification Results

Theorem proving. Rodin generated 401 POs for the context switch model and automatically discharged 284 (approximately 70%) of them. We manually discharged the rest 117 POs using theorem proving tools and SMT solvers (refer to Section 2.2). The timeout for each SMT solver is 10 seconds. Table 3 presents the number of discharged POs corresponding to the hardware-generic model refinements (Level 0 to 3) and the hardware-specific model refinements (Level 4) for the context switch functionality in TriCore and RISC-V architectures. The `POs` column in the table indicates the number of discharged POs, and the values in the `Auto` and `Manual` columns are subsets of the values in the `POs` column indicating, respectively, the automatically and manually discharged POs.

Table 3: Number of discharged POs.

Level	POs	Auto	Manual
0	60	50	10
1	68	37	31
2	29	21	8
3	114	86	28
4-TriCore	83	51	32
4-RISC-V	47	39	8
Total	401	284	117

The discharged POs prove that all refinements in our model correspond to their abstraction. Most of these POs ensure that the refined machine preserves the invariants in the abstract machine or prove that the refined concrete actions correctly preserve the corresponding abstract actions. Specifically, four POs were complex and required several proving steps to discharge. These four POs correspond to Level 1 model refinements and prove that the concrete context save and restore actions in the refinement correctly preserve the abstract context save and restore actions in Level 0 model refinements.

LTL model checking. We used 16 LTL formulas (derived from the semi-formal expressions) to verify the functional correctness of the hardware-generic model w.r.t. the requirement specifications (OSn). In addition, we used 13 LTL formulas to verify the *safety*, *reachability*, and *liveness* properties. These 13 LTL formulas guarantee that: (1) the context save/restore actions in the model do not corrupt the context when saving/restoring it, (2) a particular state is reachable from a given state (e.g., the `vs_KernSyscall` state is reachable from the `fs_KernelEntry` state, refer to Figure 4), and (3) the `fs_KernelExit` state is eventually reached from the `fs_KernelEntry` state. We adapted the LTL formulas to include the hardware-specific post-condition checks and verified the correctness of the hardware-specific models. The *safety*, *reachability*, and *liveness* properties and the LTL formulas are available as artifacts (🗎).

4 Related Work

In recent years, several approaches have focused on formal modeling and verification of OSs functionalities. CertiKOS [22] and seL4 [28] use a refinement-based formal modeling technique to model and verify OS specifications. In ORIENTAIS [41], the authors first implement the low-level code, translate it into an intermediate representation, and then verify interrupt and memory safety properties. Butterfield et. al. [12] formally model and verify the task synchronization mechanisms of RTEMS [11]. Baumann et. al. [8] use a deductive program verification approach to verify syscall functionalities of PikeOS microkernel [38]. Manjunath et. al. [32] present a framework for verifying the correctness of low-level RTOS code against runtime hardware effects. In [15], the authors propose an automata-based formal verification method to verify the Linux kernel code. Adelt et. al. [3] develop configurable timed automata models of high-level RTOS functionalities such as event synchronization, communication mechanisms, and the scheduler. Ding et. al. [17] present a framework for verifying the correctness of μC/OS-II [29]. These works implement formal models that abstract the existing OS code and verify their correctness. However, maintaining consistency between the model and the code is challenging and involves significant overhead. In particular, each time the OS is ported to a new hardware architecture, the models that represent the low-level code behavior must be entirely reimplemented from scratch.

Guan et. al. [23] use Event-B to model and verify the Event-Bus functionality of IoT OSs. Similar to our approach, Gomes et. al. [33] use Event-B to model and verify the low-level functionalities of SmartOS [40] while facilitating portability. However, they model the order of events textually within the Event-B framework. While this approach might be suitable for simple use cases, the modeling complexity becomes too high, particularly when ordering the events of complex use cases (e.g., context switch of an AUTOSAR OS). Moreover, the textual ordering of events can be error-prone. In contrast, we use UML-B to graphically order the events, which significantly reduces modeling complexity and helps avoid errors during event ordering.

Our approach extends the work by Gomes et. al. [33] in three key aspects: (1) Developing progressively refined FSMs with UML-B and Event-B while simplifying the ordering of events; (2) Systematically deriving LTL formulas from the use case analysis to verify the functional correctness of the model; (3) Ensuring traceability of requirements throughout the designs and model refinements using annotations and checks.

Haur [24] formally models and verifies the scheduler and the task synchronization mechanisms of Trampoline AUTOSAR OS [9]. Béchennec et. al. [10] formally model the high-level OS functionalities as timed automata. Chiba et. al. [13] use Event-B to verify the AUTOSAR OS task memory protection. Fang et. al. [19] present a formal framework to verify task state transitions in an AUTOSAR OS. These works formally model and verify only the high-level functionalities of AUTOSAR OSs. To the best of our knowledge, there is no approach that formally models and verifies the low-level functionalities of AUTOSAR OSs, and addressing this gap is important for ensuring the overall correctness of these safety-critical OSs. Therefore, our work aims to provide an approach to formally model and verify the low-level functionalities of AUTOSAR OSs.

5 Discussion

5.1 Benefits

Improved reusability. Through 12 iterative refinements, we implemented the hardware-generic model. This model represents the low-level functionalities of the context switch while abstracting hardware-specific details, making it reusable across different architectures. In particular, we instantiated the hardware-generic model to TriCore and RISC-V architectures with minimal refinements (only two for TriCore and one for RISC-V). Furthermore, to the best of our knowledge, the hardware-generic model can be reused across other hardware architectures (in the automotive industry), such as ARM and V850, since they satisfy the hardware assumptions (HWn). Our modeling approach significantly improves the reusability of low-level models to generate hardware-specific code, thereby facilitating the overall portability of the OS.

Scalable modeling approach. We model complex low-level functionalities as progressively refined FSMs since they ensure deterministic behavior and support modularity. Furthermore, UML-B supports the graphical arrangement of events, thereby reducing the complexity of event ordering in the Event-B model. This makes our approach more scalable and less error-prone (in terms of event ordering) compared to using Event-B alone.

Traceable model completeness. We manually annotate the FSM designs and Event-B models in text with the respective use cases and requirement specifications (OSn). We use these annotations to trace and manually check whether all requirements and hardware-specific functionalities are implemented. This way, we confirm that our models are complete w.r.t. the given requirement specifications. In future work, we intend to use ProR [27], an Eclipse Requirements Modeling Framework within Rodin, to link the Event-B model elements (e.g., variables) with the requirement specifications.

5.2 Limitations

Incomplete hardware specifications. Although there is a risk that hardware specifications relevant to the low-level functionality might be incomplete, we mitigate the risk by making hardware assumptions (HWn) and identifying the relevant hardware specifications that satisfy these assumptions.

Interpretation errors. The requirement specifications are written in natural language. Despite systematically processing these requirements through our use case analysis, interpretation errors can still occur. To minimize these errors further, in future work, we plan to use requirement engineering tools [14] and integrate Natural Language Processing (NLP) techniques [34, 35, 45] to process the requirements.

Challenges in developing a hardware-generic model. To abstract hardware-specific details and develop a hardware-generic model, the developer needs expertise in: (a) the functionality to be modeled, (b) the target hardware architectures and their common features (e.g., interrupt enable/disable), and (c) the formal methods Event-B and UML-B, including how refinements work within these frameworks. Specifically, expertise in formal methods helps identify an optimal sequence of refinements to model the desired functionality.

6 Conclusion

In this work, we presented a refinement-based formal modeling approach to facilitate portability and to verify the correctness of complex low-level functionalities of an AUTOSAR OS. We addressed RQ1 (refer to Section 1) by using UML-B and Event-B to first formally model the behavior of the context switch functionality, abstracting the hardware-specific details. This model is hardware-generic and thereby reusable across different hardware architectures. Then, we refined the hardware-generic model into TriCore and RISC-V-specific models. These models can then be used to generate hardware-specific code. Thus, by improving the reusability of low-level models, our approach facilitates the overall portability of the OS. We addressed RQ2 by using theorem proving and LTL model checking to ensure the consistency of the refinements and to verify the correctness of the models w.r.t. the requirement specifications (OSn).

In the future, we plan to implement a code generation process that translates the hardware-specific models and generates assembly code in the respective hardware architecture. In addition, we intend to use NLP techniques to automate the use case analysis process and the derivation of LTL formulas.

Acknowledgment

This work has been supported by the FFG under Contract No. 881844: "Pro²Future" (Products and Production Systems of the Future), Graz University of Technology (TU Graz), and Elektrobit Automotive GmbH in the joint research project CompEAS, and by TU Graz Open Access Publishing Fund.

References

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B: system and software engineering.* Cambridge University Press.
[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. Journal on software tools for technology transfer* 12 (2010). https://doi.org/10.1007/s10009-010-0145-y
[3] Julius Adelt, Julian Gebker, and Paula Herber. 2024. Reusable formal models for concurrency and communication in custom real-time operating systems. *Int. Journal on Software Tools for Tech. Transfer* 26, 2 (2024). https://doi.org/10.1007/s10009-024-00743-4
[4] Infineon Technologies AG. 2021. AURIX™ TC3xx User Manual Part-1.
[5] Infineon Technologies AG. 2021. AURIX™ TC3xx User Manual Part-2.
[6] Infineon Technologies AG. 2023. 32-bit AURIX™ TriCore™ Microcontroller. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/
[7] AUTOSAR. 2022. Specification of Operating System. https://www.autosar.org/fileadmin/standards/R22-11/CP/AUTOSAR_SWS_OS.pdf
[8] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2009. Formal verification of a microkernel used in dependable software systems. In *Int. Conf. on Computer Safety, Reliability, and Security.* Springer. https://doi.org/10.1007/978-3-642-04468-7_16
[9] Jean-Luc Bechennec, Mikael Briday, Sébastien Faucou, and Yvon Trinquet. 2006. Trampoline an open source implementation of the osek/vdx rtos specification. In *IEEE Conf. on Emerging Tech. and Factory Automat.* IEEE. https://doi.org/10.1109/ETFA.2006.355432
[10] Jean-Luc Béchennec, Olivier Henri Roux, and Toussaint Tigori. 2018. Formal model-based conformance verification of an OSEK/VDX compliant RTOS. In *5th Int. Conf. on Control, Decision and Inf. Tech. (CoDIT).* IEEE. https://doi.org/10.1109/CoDIT.2018.8394813
[11] Gedare Bloom, Joel Sherrill, Tingting Hu, and Ivan Cibrario Bertolotti. 2020. *Real-time systems development with RTEMs and multicore processors.* CRC Press. https://doi.org/10.1201/9781351255790
[12] Andrew Butterfield and Frédéric Tuong. 2023. Applying Formal Verification to an Open-Source Real-Time Operating System. In *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday.* Springer. https://doi.org/10.1007/978-3-031-40436-8_13
[13] Yuki Chiba, Toshiaki Aoki, et al. 2018. Formalization and Verification of AUTOSAR OS Standard's Memory Protection. In *Int. Symp. on Theoretical Aspects of Software Eng. (TASE).* IEEE. https://doi.org/10.1109/TASE.2018.00017
[14] Juan M Carrillo de Gea, Christof Ebert, Mohamed Hosni, Aurora Vizcaíno, Joaquín Nicolás, and José L Fernández-Alemán. 2021. Requirements engineering tools: An evaluation. *IEEE Software* 38, 3 (2021). https://doi.org/10.1109/MS.2021.3058394
[15] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. 2019. Efficient formal verification for the Linux kernel. In *Software Eng. and Formal Methods: 17th Int. Conf., SEFM 2019, Oslo, Norway, September 18–20, 2019, Proceedings 17.* Springer. https://doi.org/10.1007/978-3-030-30446-1_17
[16] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. 2014. Introduction to the Integration of SMT-Solvers in Rodin. In *1st Int. Workshop About Sets and Tools (SETS 2014). ABZ.*
[17] Jizheng Ding, Xiaoran Zhu, Jian Guo, Xin Li, and Rongkun Yan. 2018. End-to-End Automated Verification for OS Kernels. In *25th Asia-Pacific Software Eng. Conf. (APSEC).* IEEE. https://doi.org/10.1109/APSEC.2018.00028
[18] Event-B. 2023. Event-B Summary. https://wiki.event-b.org/images/EventB-Summary.pdf
[19] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. 2012. Formal model-based test for AUTOSAR multicore RTOS. In *5th Int. Conf. on Software Testing, Verification and Validation.* IEEE. https://doi.org/10.1109/ICST.2012.105
[20] Elektrobit Automotive GmbH. 2022. AutoCore OS, Classic AUTOSAR Operating System. https://www.elektrobit.com/products/ecu/eb-tresos/operating-systems/
[21] Renata Martins Gomes, Bernhard Aichernig, and Marcel Baunach. 2020. A Formal Modeling Approach for Portable Low-Level OS Functionality. In *Software Eng. and Formal Methods*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham. doi: 10.1007/978-3-030-58768-0_9.
[22] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: an extensible architecture for building certified concurrent OS kernels. In *OSDI*, Vol. 16.
[23] Yuqian Guan, Jian Guo, and Qin Li. 2021. Formal verification of a hybrid IoT operating system model. *IEEE Access* 9 (2021). https://doi.org/10.1109/ACCESS.2021.3073398
[24] Imane Haur. 2022. *AUTOSAR compliant multi-core RTOS formal modeling and verification.* Ph. D. Dissertation. École centrale de Nantes.
[25] RISC-V International. 2023. https://riscv.org/
[26] RISC-V International. 2024. The RISC-V Instruction Set - Manual Volume I.
[27] Michael Jastram. 2010. ProR, an open source platform for requirements engineering based on RIF. *In Proc. of the Systems Eng. Infrastructure Conf.* (2010).
[28] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014). https://doi.org/10.1145/2560537
[29] Jean Labrosse. 2002. *MicroC/OS-II: The Real Time Kernel.* CRC Press.
[30] Thierry Lecomte. 2014. Atelier B. *Formal Methods Applied to Complex Systems: Implementation of the B Method* (2014).
[31] Michael Leuschel and Michael Butler. 2008. ProB: an automated analysis toolset for the B method. *Int. Journal on Software Tools for Tech. Transfer* 10 (2008). https://doi.org/10.1007/s10009-007-0063-9
[32] Vignesh Manjunath and Marcel Baunach. 2024. A framework for static analysis and verification of low-level RTOS code. *Journal of Systems Architecture* 154 (2024). https://doi.org/10.1016/j.sysarc.2024.103220
[33] Renata Martins Gomes, Bernhard Aichernig, and Marcel Baunach. 2024. A framework for embedded software portability and verification: from formal models to low-level code. *Software and Systems Modeling* (2024). https://doi.org/10.1007/s10270-023-01144-y
[34] Anmol Nayak, Hari Prasad Timmapathini, Vidhya Murali, Karthikeyan Ponnalagu, Vijendran Gopalan Venkoparao, and Amalinda Post. 2022. Req2Spec: Transforming software requirements into formal specifications using natural language processing. In *Int. Working Conf. on Requirements Eng.: Foundation for Software Quality.* Springer. https://doi.org/10.1007/978-3-030-98464-9_8
[35] Farhana Nazir, Wasi Haider Butt, Muhammad Waseem Anwar, and Muazzam A Khan Khattak. 2017. The applications of Natural Language Processing (NLP) for software requirement engineering-a systematic literature review. *Inf. Science and Appl.: ICISA* (2017). https://doi.org/10.1007/978-981-10-4154-9_56
[36] Event-B Notation. 2007. Event-B notation. https://web-archive.southampton.ac.uk/deploy-eprints.ecs.soton.ac.uk/11/3/notation-1.5.pdf
[37] OSEK/VDX. 2005. Operating System Specification 2.2.3. https://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf
[38] PikeOS. 2024. PikeOS, Certifiable RTOS & Hypervisor. Retrieved Jan. 1, 2024 from https://www.sysgo.com/pikeos
[39] Kristin Y Rozier. 2011. Linear temporal logic symbolic model checking. *Computer Science Review* 5, 2 (2011). https://doi.org/10.1016/j.cosrev.2010.06.002
[40] Tobias Scheipel, Leandro Batista Ribeiro, Tim Sagaster, and Marcel Baunach. 2022. Smartos: An OS architecture for sustainable embedded systems. *GI Fachgruppentreffen Betriebssysteme* (2022).
[41] Jianqi Shi, Jifeng He, Huibiao Zhu, Huixing Fang, Yanhong Huang, and Xiaoxian Zhang. 2012. ORIENTAIS: Formal verified OSEK/VDX real-time operating system. In *17th Int. Conf. on Eng. of Complex Computer Systems.* IEEE. https://doi.org/10.1109/ICECCS20050.2012.6299224
[42] Colin Snook. 2014. iUML-B Statemachines. *Proc. of the 5th Rodin User & Developer Workshop* (2014).
[43] Colin Snook and Michael Butler. 2006. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 15, 1 (2006). https://doi.org/10.1145/1125808.1125811
[44] Jan Strejcek. 2004. *Linear temporal logic: Expressiveness and model checking.* Ph. D. Dissertation. Citeseer.
[45] Aya Zaki-Ismail, Mohamed Osama, Mohamed Abdelrazek, John Grundy, and Amani Ibrahim. 2022. RCM-extractor: an automated NLP-based approach for extracting a semi formal representation model from natural language requirements. *Automated Software Eng.* 29, 1 (2022). https://doi.org/10.1007/s10515-021-00312-y