Software and Systems Modeling (2024) 23:289–315
https://doi.org/10.1007/s10270-023-01144-y

SPECIAL SECTION PAPER

A framework for embedded software portability and verification: from formal models to low-level code

Renata Martins Gomes[1] · Bernhard Aichernig[1] · Marcel Baunach[1]

Received: 3 June 2022 / Revised: 20 September 2023 / Accepted: 12 December 2023 / Published online: 1 February 2024

Abstract
Porting software to new target architectures is a common challenge, particularly when dealing with low-level functionality in drivers or OS kernels that interact directly with hardware. Traditionally, adapting code for different hardware platforms has been a manual and error-prone process. However, with the growing demand for dependability and the increasing hardware diversity in systems like the IoT, new software development approaches are essential. This includes rigorous methods for verifying and automatically porting Real-Time Operating Systems (RTOS) to various devices. Our framework addresses this challenge through formal methods and code generation for embedded RTOS. We demonstrate a hardware-specific part of a kernel model in Event-B, ensuring correctness according to the specification. Since hardware details are only added in late modeling stages, we can reuse most of the model and proofs for multiple targets. In a proof of concept, we refine the generic model for two different architectures, also ensuring safety and liveness properties. We then showcase automatic low-level code generation from the model. Finally, a hardware-independent factorial function model illustrates more potential of our approach.

Keywords Embedded systems · RTOS · Formal methods · Event-B · Verification · Code generation · Portability

1 Introduction

The amount of computing devices in the Internet of Things (IoT) (in, e.g., autonomous vehicles, smart infrastructures, automated homes, and production facilities), is expected to increase exponentially, along with the diversity on both the hardware (HW) and the software (SW) side [23, 32]. 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-core or many-core systems, including specialized ASIC or even reconfigurable FPGA components [19, 29].

While high-level code can more easily be compiled for a new or different 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, 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 cooperation, supporting additional MCU families or just variants is not straightforward, even if there are only a few differences to existing ports [53]. 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, faulty implementations, new bugs, and security holes are common. Besides, changes in the logic of low-level software must be manually introduced

Communicated by Antonio Cerone and Frank de Boer.

Note: This paper results from a combination of [27], a publication that led to the invitation for this special section, and our subsequent paper [28]. We have significantly extended the content with unpublished work, including a new example and more details on how code is generated from formal models.

✉ Marcel Baunach
baunach@tugraz.at

Renata Martins Gomes
renatamgomes12@gmail.com

Bernhard Aichernig
aichernig@ist.tugraz.at

1 Graz University of Technology, Graz, Austria

to all implementations, which also hinders or slows down important improvements of the OS.

These difficulties in porting software clearly signal that new approaches must be sought to make the porting process easier and more reliable. This would not only increase the number of supported platforms, but could also have a positive impact in the quality of ports, reduce costs, and even open new horizons on software evolution. As an example, model-driven software development has been largely used in the embedded software industry. It can improve quality while reducing costs and development time. Simple sketches and more elaborate modeling approaches, such as UML, are very common [4], and formal modeling approaches are gaining traction since a while. Formal methods have been experimented on software development, bringing a new perspective and showing that considerable improvements with respect to correctness proofs and verification can be achieved [46, 82]. Despite several obstacles regarding their applicability for complex real-world systems [57], industry has been successfully exploring the benefits of introducing formal methods to their development processes [11, 79]. Such methods have been used for specification and verification, and also for testing and code generation [7]. Even for OS development, several works apply formal methods with great and promising results on, e.g., functional safety and security [31, 42, 84].

However, to the best of our knowledge, no one has tried to use formal methods to do verification and to improve software portability at the same time for embedded system software, such as OS in, e.g., the IoT. We propose a change of perspective to what software development is, which has the potential of not only improving portability, but also overall maintainability and dependability. While this is a big challenge and an ongoing work, we have investigated a novel OS development framework based on formal methods and code generation. This paper presents the framework and its concepts, along with a proof of concept that shows how an OS context switch can be modeled in a generic way and how the code can still be automatically generated for two different target architectures. Besides the low-level context-switch, we have also applied the framework on a high-level mathematical function that calculates the factorial of a given number. This demonstrates that our framework is not limited to low-level OS functionality, and can potentially be applied to a wide range of software functionality.

The first part of our approach addresses a research question on verifying low-level functionality. *RQ1: How can low-level software development support guaranteed dependability while not undermining portability?*

The methodology is based on incrementally refined formal models to prove safety and liveness properties. We present parts of the model of an Real Time Operating System (RTOS) that focus on the switch into the kernel and back to a task, detailing the operations that happen during these transitions. We chose the context switch as demonstrating example, because it is architecture-specific, typically requires reimplementation for each new 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: MSP430 and RISC-V. In order to verify safety (something bad must *not* happen) and liveness (something good *should* eventually happen) properties [44], 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 eventually 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 late refinement to each architecture, as well as their proofs follow and become much simpler, as we will show.

The second part of our approach addresses a research question on code generation: *RQ2: How can low-level software be engineered to be more portable to different hardware architectures?*

The methodology is based on using the verified models from the first part as an input to EB2LLVM, a tool that resulted from our research and automatically generates LLVM intermediate code which is eventually compiled with a target-specific compiler backend. To demonstrate how code is generated from Event-B models, we first present another model, applying the entire framework on the generally known high-level algorithm to calculate the factorial of a number. The factorial example presents EB2LLVM's basic features, and all stages of code generation for the same target architectures as for the OS model, i.e., MSP430 and RISC-V. We also compare the automatically generated code with equivalent code compiled from the function implemented in C.

Finally, we show how EB2LLVM generates code from the presented OS model, including register and array accesses, and, step-by-step, how generic parts of the OS model can generate target-specific code considering the target's hardware model.

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, and code is automatically generated for two targets. 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; (4) assembly code is automatically generated and compiled into binaries for the two model instantiations; (5) a formal model of the factorial function, with its automatically generated code.

Structure. Section 2 discusses related work and Sect. 3 provides background on the tools and the two target HW architectures. Section 4 presents an overview of the framework. Section 5 presents the first part of the framework, including the requirements, the general idea of the modeling process and our refinement strategy, the modeled RTOS, and its verification. The second part of the framework is presented in Sect. 6, with the model and code generation for the factorial example, and the code generation for the RTOS model presented in the first part. We conclude in Sect. 7.

2 Related work

Throughout the history of software development, portability has been a difficult to tackle issue. Porting is a common source of bugs in drivers [14] and OSs in general [49, 61]. Code duplication, for example, is often difficult to maintain and often leads to faults [24, 38, 39]. These problems can be tracked to the porting process itself [13, 34]. Several works on porting experiences report difficulties in the low-level code [69] and the need to consider several aspects of the target [58], and often the ports miss some functionality [53, 83]. Even the generally considered portable Linux and UNIX OSs present important challenges [9, 41, 74, 75]. Solutions for improving software portability in different domains have been proposed [48, 67]. Some works try to improve aspects of portability with formal methods and code generation [22, 60]. While the generated code is semantically hardware-dependent, low-level assembly code still cannot be generated.

Formal methods can also aid in OS verification, and several works investigate how to formally model and verify OSs [5, 10, 12, 15, 16, 18, 56, 71, 72]. The most known formally-verified microkernel is probably sel4 [51], whose modeling and verification strategies have been largely studied [42, 73]. OpenComRTOS [76] develops implementations from formal models, though these implementations are still handwritten. It is also possible to formally specify an entire Instruction Set Architecture (ISA) and automatically generate a simulator for it [80, 81].

With modular UML models, Besnard et al.'s work [8] decouples system and environment models, deploying the same system model for simulation, verification, and execution. Also focused on the software deployment, Rivera et al. [64] present a tool that automates the transformation of Platform-Independent Models (PIMs) into Platform-Specific Models (PSMs). These approaches are interesting for our use-case, however we have not further explored UML as a modeling language.

Automatic generation of code from formal models has also been investigated [6, 17, 54, 55, 65, 68]: However, the state-of-the-art modeling approaches rarely take care of separating software from hardware concepts. When high-level functionality is modeled, this does not present an issue, and the model can be detailed enough that (high-level) code can be generated and then compiled to different hardware platforms. However, as soon as direct interaction with hardware is necessary, they either model it directly, which means that the model does not or only with considerable effort work for different hardware; or they do not detail the model enough to be able to automatically generate any code from it. Such generated high-level code does usually not support direct references to, e.g., CPU registers. Apart, corresponding tools like MATLAB [54] commonly lack built-in support for formal verification in the way dedicated formal methods tools like model checkers or theorem provers do. Even if, the verification effort must be repeated for each target, as the hardware-specific models largely differ. An exception in the tooling landscape is AutoFOCUS [6] which comes with formal semantics, code generation and support for hardware deployment. However, AutoFOCUS has been designed for component-based development of embedded applications, its focus is not on low-level and portable operating system development.

Finally, Holland [33] proposes an approach to synthesize machine-specific OS assembly code by introducing a substantial number of new specification languages and tools to handle compiler-specific and machine-specific calling conventions, data types, memory layouts, etc. While these languages are tailored to, e.g., the special requirements of context switching, the verification step only considers the correct interaction between the assembly code and the C compiler that calls the generated sequences. The assembly code generation must be manually adjusted for each new architecture.

In contrast, our approach is to generate LLVM intermediate code from formal software and hardware models. We then use the standard compiler toolchain which ultimately generates low-level assembly code for a wide range of supported target architectures. Since the intermediate representation language is unified, our code generation algorithm needs no adaptation to new hardware. Apart, we allow to directly reuse the software model (along with already accomplished proofs) for all target architectures by strictly separating it from their hardware models. For a new hardware architecture, only its hardware model and the instantiation needs to be implemented and verified against the specification. This is more efficient compared to implementing a new and mixed hardware/software model for each new target architecture.

3 Background

3.1 Event-B and Rodin

Event-B is a formal method for system-level modeling and analysis [2, 21] derived from the B formalism [1]. Based on set theory and state transitions, Event-B supports refinements to model different abstraction levels, while mathematical proofs verify correctness and consistency between refinements. The Rodin Platform [37], 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). POs are mathematical formulas that must be proved (discharged) to ensure correctness of the model. Several plugins are available to Rodin that aid in, e.g., automatic PO discharging and model checking.

In order to ease the understanding of readers unfamiliar with Event-B, we introduce the components of Event-B models and then present a model with most of these components to show how they can model a sequential program. Listing 1 and Listing 2 present the model explained later, and can also be used as reference to the keywords and components explained next. We assume the reader is familiar with basic set theory notation.

The two top elements of a model in Event-B 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*. In contrast, just *context* refers to task context in operating systems. Event-B contexts describe all static information about the system and can have carrier *sets* and *constants*. *Axioms* are the predicates that the constants obey and will be available as hypotheses in the POs. It is also possible to add *theorems* which must be proved. If an Event-B context *extends* others, it can reference sets and constants of the extended Event-B contexts and any they also extend. Dynamic information about the system is described in the *machines*. A machine can *see* Event-B contexts, such that it can use their sets and constants to relate static and dynamic information, as well as axioms and theorems to discharge POs. Machines can have *variables* that obey predicates given by their *invariants*, and *theorems* can be added and proved. Considering machine states as sets of the variables' values, each *event* represents a state transition. An event is said to be enabled if the state satisfies its *guard* condition. The mandatory event INITIALISATION serves as a starting point, has no guards, and only runs once to initialize all variables. Event-B does not define which event will be executed in case more than one is enabled simultaneously. The execution of an event modifies the current state according to its *actions*, which are executed simultaneously. It is possible to limit how often an event may be enabled using a *variant*, which is either a numeric expression or a finite set whose free identifiers are constants or concrete variables. Events always

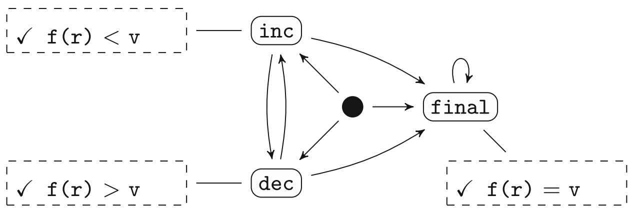


Fig. 1 : Visualization of the binary search Event-B model

have a *status*, which can be either *ordinary* (default status, the event may occur arbitrarily often), *convergent* (the event must decrease the variant), or *anticipated* (the event must not increase the variant). When a machine *refines* another one, it may introduce details or more variables. The machine events must be refined or kept as they were. In special cases, when the event can never be enabled, it may be deleted from the refined machine. POs define what is to be proved for a model and are automatically generated by Rodin. For example, the POs guarantee that invariants always hold and that refined events do not contradict abstract ones.

Listing 1 and Listing 2 reproduce an example from Abrial's book: the binary search in a sorted array (Section 15.4 of [2]). We have only created a Rodin project and copied the Event-B model from the book into it, correcting some typos present in the original and discharging the POs Rodin generated. To complete our example here, we also added the Event-B context c1 (Listing 1b), which is not present in the original model. With this example, we can show Event-B's notation and model components, as well as some concepts for modeling sequential programs introduced by Abrial and used in our models. In this model, we only have four events and the state is composed of a few variables. Still, it can be challenging to understand how state transitions occur and which effect events may have on each other. Therefore, it is helpful to visualize the model graphically. Many tools can create visualizations from Event-B models, producing intricate and detailed graphs. For the sake of simplicity, however, we present simple state-machine-like graphs representing our models. As an example, Fig. 1 presents a visualization of the concrete Event-B model of the binary search. The nodes represent the events' guards, and edges represent the possible state transitions caused by their actions when they execute. The dashed boxes are the guard expressions defining the state that enables the respective event.

Listing 1a shows the Event-B context with the binary search pre-conditions: f is an array of natural numbers with size n, and v is a value in the range of f (i.e., an element of the array). The program (modeled in the machine, explained later) will search for the index r in the array, such that f(r) = v. The axiom axm0_4 tells us that the array f is sorted in a non-decreasing way, and the theorem thm0_1 states that there is at least one element in the array. Listing 1b is not

```
CONTEXT c0
CONSTANTS
  n
  f
  v
AXIOMS
  axm0_1: n ∈ ℕ
  axm0_2: f ∈ (1..n)→ℕ
  axm0_3: v ∈ ran(f)
  thm0_1: n ≥ 1 theorem
  axm0_4: ∀ i,j · i ∈ 1..n ∧ j ∈ 1..n
    ∧ i ≤ j ⇒ f(i) ≤ f(j)
END
```

(a) Event-B context `c0` – pre-conditions

```
CONTEXT c1
EXTENDS c0
SETS
  S
CONSTANTS
  s1
  s2
  s3
AXIOMS
  nProB: n = 5
  vProB: v < 10
  setDef: partition(S,{s1},{s2},{s3})
END
```

(b) Event-B context `c1` – *extends* `c0`

```
MACHINE m0
SEES c0
VARIABLES
  r
INVARIANTS
  inv0_1: r ∈ ℕ
EVENTS
 INITIALISATION
  THEN
   act0_init_1: r :∈ ℕ
  END
 progress anticipated
  THEN
   act0_prog_1: r :∈ ℕ
  END
 final
  WHERE
   grd0_fin_1: r ∈ 1..n
   grd0_fin_2: f(r) = v
  THEN
   skip
  END
END
```

(c) Event-B machine `m0` – post-condition and progress

Listing 1: Binary search as Event-B model – initial model

part of the model. However, it shows how an Event-B context can extend another to use (and further detail) elements of the extended context, for example, defining the size `n` of the array `f`. It also defines a carrier set `S` and new constants. The axiom `setDef` defines `S` as a partition of three subsets with one element each, effectively creating an enumerated set of three elements. The machine of the initial model (Listing 1c) *sees* the Event-B context `c0` and declares the variable `r`, which represents the index that the program must find. Besides the mandatory `INITIALISATION` event, we add two others. Event `final` represents the specification of the program, i.e., its post-condition. It has no actions (`skip`), and its guards represent the state where the program has found `r`. Note that `axm0_3` states that `v` is in the range of `f`, guaranteeing the program will find `r`. The other event we add is the anticipated event `progress`, which modifies `r` in a non-deterministic way. Its refinements will detail how `r` is to be found and will eventually be made convergent on a variant, such that the loop it creates is guaranteed to terminate eventually. In the first refinement `m1` (Listing 2a), two new variables, `p` and `q`, are introduced, representing search indexes of the array `f`. The abstract event `progress` is split into events `inc` and `dec`, which are made convergent on the variant `q`$-$`p`. Since `inv1_3` has been introduced, the guard `grd0_fin_1` in `final` of Listing 1c always holds and can therefore be removed from the refined machine `m1`. The second refinement `m2` (Listing 2b) defines how `r` is to be found within `inc` and `dec`, defining the increment and decrement intervals that were non-deterministic in `m1`. Note that, since event `final` remains exactly the same as its abstraction, its guard is omitted and the word `REFINES` is replaced by `EXTENDS`. The goal state is still `f(r)` $=$ `v`.

While we assume the reader is familiar with basic set theory, our models use some set and relation operators that are less generally known. To ensure understanding of the models, we explain the meaning of these operators with an example. Listing 3 defines two sets, `S1` and `S2`, where `S2` is a subset of `S1`. The set `S1` is a partition of constants `a` to `e` (definition omitted for simplicity). A relation `r` is a set of pairs where its domain `dom(r)` is the set of all elements occurring on the left side of the pairs and its range `ran(r)` is the set of elements on the right side. A function is a special case of relation, where each element of the domain is uniquely related to one element of the range. The two relations defined in Listing 3, `r1` and `r2`, are partial functions from, respectively, `S1` and `S2` to ℕ. Table 1 shows the values in each relation range. Columns `r1` and `r2` show the range values defined for these relations, each cell representing the value mapped to the cor-

```
MACHINE m1
REFINES m0
SEES c0
VARIABLES
  r
  p
  q
INVARIANTS
  inv1_1: p ∈ 1..n
  inv1_2: q ∈ 1..n
  inv1_3: r ∈ p..q
  inv1_4: v ∈ f[p..q]
VARIANT
 q−p
EVENTS
 INITIALISATION
  THEN
   act1_init_1: r :∈ 1..n
   act1_init_2: p := 1
    act1_init_3: q := n //typo: q := 1
  END
 inc convergent
  REFINES progress
  WHERE
   grd1_inc_1: f(r) < v
  THEN
   act1_inc_1: r :∈ r+1..q
   act1_inc_2: p := r+1
  END
 dec convergent
  REFINES progress
  WHERE
   grd1_dec_1: v < f(r)
  THEN
    act1_dec_1: r :∈ p..r−1
   act1_dec_2: q := r−1
  END
 final
  REFINES final
  WHERE
   grd1_fin_2: f(r) = v
  THEN
   skip
  END
END
```

(a) Event-B machine m1

```
MACHINE m2
REFINES m1
SEES c0 // c1 for extended
VARIABLES
  r
  p
  q
EVENTS
 INITIALISATION
  THEN
   act2_init_1: r := (1+n)/2
   act2_init_2: p := 1
   act2_init_3: q := n
  END
 inc
  REFINES inc
  WHERE
   grd2_inc_1: f(r) < v
  THEN
   act2_inc_1: r := (r+1+q)/2
   act2_inc_2: p := r+1
  END
 dec
  REFINES dec
  WHERE
   grd2_dec_1: v < f(r)
  THEN
   act2_dec_1: r := (p+r−1)/2
   act2_dec_2: q := r−1
  END
 final
  EXTENDS final
  THEN
   skip
  END
END
```

(b) Event-B machine m2

Listing 2: Binary search as Event-B model – first and second refinements

responding set element. If an element is not in the relation's domain, the cell is marked with "-". If it is, but there is no mapping for it in the relation the cell is left blank. The other columns represent the result of some operations with r1: The domain subtraction S2 ⩤ r1 removes all S2 from r1's domain, leaving only two mapped values. Domain restriction S2 ◁ r1 is the inverse, only leaving values mapped to elements present in S2. The overwrite operation r1 <+ r2 overwrites the r1 relation with values from r2. Finally, Listing 3 also shows the results of two other operations as theorems: S1 minus S2 results in a set with all elements from S1 that are not in S2 (thm1). Brackets are used to access a relation element, as exemplified by thm2.

3.2 Model checking and ProB

Temporal logic [52, 59] 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 [30]. It is particularly useful for expressing properties such as *safety* (given a precondition,

Table 1 Relation operators and example values

Set element	r1	r2	Dom.Subtraction S2 ⩤ r1	Dom.Restriction S2 ◁ r1	Overwrite r1 <+ r2
a	1	–	1	–	1
b	3	–	3	–	3
c	5	2	–	5	2
d	7		–	7	7
e		4	–		4

```
AXIOMS
    // definitions
    defS1: partition(S1,{a},{b},{c},{d},{e})
    defS2: S2 = {c,d,e}
    typeR1: r1 ∈ S1⇸ℕ
    defR1: r1 = {a↦1,b↦3,c↦5,d↦7}
    typeR2: r2 ∈ S2⇸ℕ
    defR2: r2 = {c↦2,e↦4}
    // theorems that exemplify other
     operations
    thm1: S1\S2 = {a,b} theorem
    thm2: r1(a) = 1   theorem
END
```

Listing 3: Example of set and relation operators

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 [47] that integrates as a plugin-in Rodin and can be easily used for LTL model checking of our RTOS models.

3.3 Hardware architectures

Next, we present the main characteristics of the two HW architectures we use as examples for the architecture-specific instantiations of our model in Sect. 5.3.

MSP430 The TI MSP430 [36] 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 variant. 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 [63] 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 [78] 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) [77]. 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`.

3.4 LLVM

The LLVM Compiler Infrastructure is a collection of compiler and toolchain components designed as a set of reusable libraries [50]. All components have well-defined interfaces, and one of the most important aspects of LLVM for this paper is the LLVM Intermediate Representation (LLVM IR). The LLVM IR is a language with well-defined semantics and a RISC-like virtual instruction set that supports simple instructions like add, subtract, compare, and branch. It is strongly typed and uses an infinite set of temporary registers prefixed with a % character. LLVM IR is defined in three forms: a textual format of the language in `.ll` files, an in-memory data structure used by optimizers, and an on-disk binary "bitcode" [45], with tools provided to transform one form into another.

An LLVM compiler has three distinct phases: the frontend is responsible for parsing the input code and translating it into LLVM IR; this IR can then go through optimizers and analyzers that improve the IR code; a backend finally produces native machine code from the IR code. To take advantage of the available LLVM backends (and eventually of some optimizers and analyzers), we have written a frontend that takes Event-B "code" and produces LLVM IR. We then use `llc`, the LLVM static compiler, to produce architecture-specific assembly code that can be used by a native assembler and linker to generate native binaries.

4 Framework overview

Our approach relies on the separation of software functionality from details of the hardware platform on which this functionality shall run. The idea is to (formally) model the functionality (i.e., the OS we want to implement), abstracting away the hardware details that would render the model specific to a certain target platform, such as registers, interrupt handling, memory specifications, etc.

We can think of this as explaining someone else what the code they should implement must do. We can explain functionality in an abstract way, such that others (e.g., experts) can presumably "understand" what is to be done, without the need to know all the details of the code. For instance, in order to support concurrent tasks, a preemptive OS must implement context switches, i.e., it must save the currently running task's runtime information and load the next one's into the CPU. Experts will probably assume that this can be implemented using interrupts, and that the interrupt controller will save the return address of an interrupted code sequence somewhere. It might change some CPU configuration, such as the ability to take further interrupts, the stack, its execution mode, etc., saving the previous values as well. Finally, it will start executing another defined code sequence (e.g., the handler specified for that interrupt). To understand the general idea of a context switch, we do not need to know exactly which and how registers are modified, or how they might be saved. When we implement the OS context switch, we know we must save the "rest" of the context that is not automatically saved by the hardware, and change the remaining CPU configuration according to what the OS needs it to be.

What we do in our approach is to formalize this abstract description, especially about low-level functionality. Figure 2 shows an overview of the framework: While the software model has a very abstract view of the hardware, these specifics are separately modeled for each target architecture in the hardware models. They are then combined (*instantiated*) into architecture-specific models, from which our code generator can generate each target's specific code. In addition, the models are verified through formal proofs, and most of these proofs are reused to verify each arch-specific model.

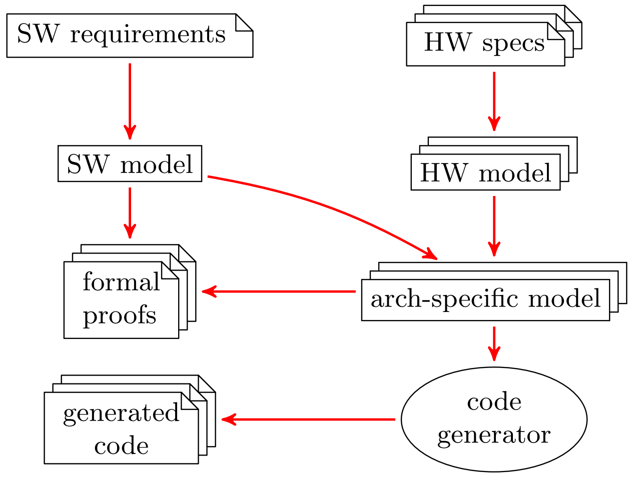


Fig. 2 : Framework overview

In our framework, software and hardware are modeled and verified with Event-B, a formal method for system-level modeling and analysis [2]. Code is then generated with a tool (EB2LLVM) that resulted from our research and parts of the LLVM compiler infrastructure, taking advantage of the backend targets that are already available.

As usual in formal modeling, the software model starts from a very abstract view of the modeled software, and is incrementally *refined* to describe all requirements. The refinements integrate, step by step, the software requirements, and we also add invariants that guarantee correctness. We also use model checking to verify liveness properties, and find errors, inconsistencies, or underspecification problems early in the modeling, so that they can be fixed in early phases of the modeling and will not become bugs in the code. Since instantiations are nothing more than late refinements of the generic model to arch-specific models, their verification also profits from the verified generic model, and is almost automatic. Other aspects to be verified, such as timing, security, functional safety, etc. could also be added to the models, and similarly verified at both the generic and the hardware-specific levels. This, however, is part of future work.

Once the software is modeled and verified, it can be easily instantiated to different targets, verified again, and the corresponding code automatically generated. When the software must be ported to a new hardware (e.g., a new architecture), one only needs to create or adapt the hardware model, instantiate the generic software model and verify/generate again. While manual porting requires the OS developer to have deep knowledge of both software and hardware, creating a hardware model with just the details required for applying our approach is much simpler, as described in Sect. 5.3.2. If OS requirements change, new features are needed, or the logic must be changed, one only needs to change the generic OS model, verify the changes, and generate the code again for all targets. This can even accelerate the process of optimizing the software design, as different options can be quickly evaluated on real hardware, without the need for the software developer to apply the change to all ports individually. It also guarantees that the ports are always consistent to each other and according to the requirements.

Finally, to complete our approach to portability, code is automatically generated from the models. While our framework aims at generating all code for the models, including linker files, generic and specific C code, etc., this paper focuses on the low-level functionality. For the context switch that serves as proof-of-concept, we only generate assembly code. The tool we have written, EB2LLVM, takes as input the Event-B models, and generates target-specific LLVM IR. The LLVM IR files generated by EB2LLVM can then be compiled with `llc`, LLVM's static compiler. While other Event-B code generators only understand basic arithmetics, EB2LLVM also understands the model's sets and axioms, such that they are used to generate low-level code that refers to target-specific registers and instructions. Taking advantage of LLVM's backend, we can generate machine code for several different targets.

The next sections present details of our framework. In Sect. 5, we use the example of the context switch in SmartOS [66], an RTOS we have developed and used for many years, to show how the software can be modeled independently from the hardware, how it is later instantiated, and how the model is verified. Section 6 shows how we generate the context switch and factorial code for RISC-V-based architectures [63] and the MSP430 [36].

5 Modeling

This section presents the first part of the framework: modeling and verification. Section 5.1 presents the requirements for the context switch and Sect. 5.2 shows the refinement strategy we used in the model. Section 5.3 details the generic model, while Sect. 5.3.2 presents the model instantiations to the target architectures. The verification we have done on this model is presented in Sect. 5.4.

5.1 Requirements

We modeled SmartOS [66], an embedded RTOS we have developed and used for many years. This section summarizes 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 general assumptions about the computing platform and present SmartOS's requirements.

5.1.1 Hardware assumptions

Even though we aim at keeping the OS model initially independent from the hardware, a target architecture must have certain general 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.

5.1.2 Software specification

SmartOS 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 next task to be executed 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 the *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 being the same as it was when that task was running last, and never in kernel mode.

OS7 A part of the *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 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*.

Table 2 : Model and requirements

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

5.2 Refinement strategy: from abstraction to detailed specification

The model has several refinements and showing all would be too cumbersome. So, we divide it into six levels of abstraction (referred to as *Level*). Each Level is composed of several refinements and addresses a new set of requirements (Table 2).[1] Up to and including Level 4, the model remains generic, only requiring the generic hardware features described in Sect. 5.1.1. We only introduce further hardware details in Level 5, where we instantiate the model for specific target architectures. This section introduces the general idea of each Level, while Sect. 5.3 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 Sect. 5.1.

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

[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].

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 Here, we refine the model to a generic hardware that automatically saves and loads a subset of the context, and the software that complements the switches.

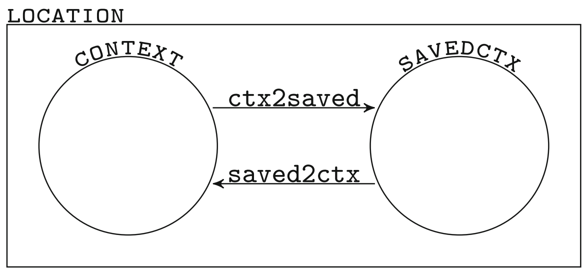
Level 5 Finally, we refine the model into architecture-specific models from which OS code can be generated (See Sect. 6).

5.3 Context switch model

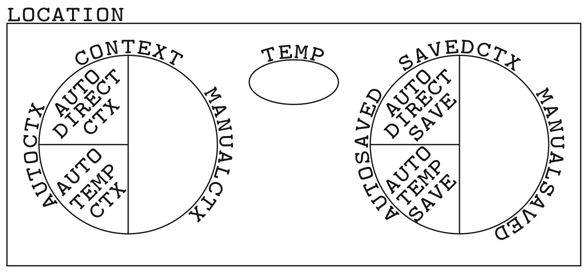
This section details the generic part of the context switch model, i.e., Levels 0 to 4 from Sect. 5.2. Please, refer to Sect. 3.1 for the Event-B notation used in the following listings.

5.3.1 Hardware-independent model

Level 0 First, we define the carrier set LOCATION (Fig. 3a), 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 $\in$ CONTEXT$\rightarrowtail\!\!\!\!\twoheadrightarrow$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



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



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

Fig. 3 : Diagrams of LOCATION

```
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
```

Listing 4: Level 0 – Initial abstraction

CONTEXT to DATA, while a saved context is a relation from SAVEDCTX to DATA.

The initial abstraction (Listing 4), sees the context switches as two context copies: one copies old context to saved $\in$ SAVEDCTX$\nrightarrow$DATA, and the other loads new context to loaded $\in$ CONTEXT $\nrightarrow$ 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 finished its execution. 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 repre-

```
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
```

Listing 5: Level 1 – 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
```

Listing 6: Level 1 – Kernel exit

sents 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 idea is that, since the context is copied in different steps by HW and software, this event can be refined into these steps. The Proof Obligations (POs) generated by Rodin verify that the refinements of this abstraction (Levels 1 to 5) are correct. If we can prove that the model always reaches `osFinal`, we prove that the desired state after OS execution is reached. These proofs are shown in Sect. 5.4.

Level 1 Listing 5 and Listing 6 show 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 (Fig. 4), and their guards define that entry must happen before exit, and exit may only start after entry is done.

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 (Listing 5, Line 11). This guarantees that, eventually, the save pro-

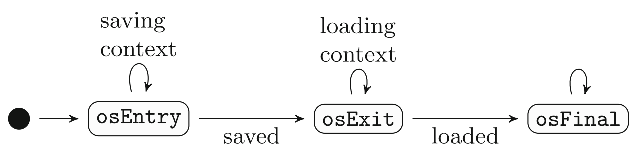


Fig. 4 : Level 1 states and transitions

cess 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 (Listing 6, 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. 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 in Listing 7 the functions `transform` (axm1) and `ctxTransform` (axm5). 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` (axm3), modifying the values stored in each location according to `ctxTransform`. This is modeled by the axiom axm7 in Listing 7.

With these definitions in Event-B contexts, we also refine the machine variables and events. 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 `rTask` ∈ 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 `rTask` to save the context. Similarly, as the CPU context is only one, we create `cpuCtx` ∈ CONTEXT → DATA to represent it, and its relation to `oldCtx` and `loaded` is given by the glue invariants `toSave` ◁ `oldCtx` = `toSave` ◁ `cpuCtx` and `loaded` = `toLoad` ⩤ `cpuCtx`. This way, all three kernel parts (*entry*, *body*, and *exit*) deal with the same variables, simplifying code genera-

```
axm1: transform ∈ (CONTEXT→DATA)⤖
(SAVEDCTX→DATA)
axm2: invTransform ∈ (SAVEDCTX→ DATA)⤖
(CONTEXT→DATA)
axm3: ctx2saved ∈ CONTEXT⤖SAVEDCTX
axm4: saved2ctx ∈ SAVEDCTX⤖CONTEXT
axm5: ctxTransform ∈ CONTEXT→(DATA→DATA)
axm6: ctxInvTransform ∈ SAVEDCTX→(DATA→
DATA)
axm7: ∀ ctx,el · ctx ∈ CONTEXT→DATA ∧ el
∈ CONTEXT ⇒ transform(ctx)(ctx2saved(el)) =
ctxTransform(el)(ctx(el))
axm8: ∀ sctx,el · sctx ∈ SAVEDCTX→DATA ∧
 el ∈ SAVEDCTX ⇒ invTransform(sctx)(
saved2ctx(el)) = ctxInvTransform(el)(
sctx(el))
```

Listing 7: Generic model axioms in Level 2

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

Listing 8: Level 2 – Kernel entry

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

Listing 9: Level 2 – Kernel exit

tion. We can finally replace the abstract save action in Listing 5 (Line 10) by the action in Listing 8 (Line 9).

The context to be loaded during *kernel exit* actually comes from the saved context of `rTask`, thus we replace the abstract load from Listing 6 (Line 10) by the load in Listing 9 (Line 9). The functions `transform`, `ctx2saved`, and `ctxTransform`, used for saving a context have their inverses also defined in Listing 7: `invTransform` (axm2), `saved2ctx` (axm4), and `ctxInvTransform` (axm6),

```
osFinal (guards)
cpuCtx = invTransform(t_saved(rTask)
 )
t_saved(oldTask) = transform(oldCtx)
toSave = ∅
toLoad = ∅
```

Listing 10: Level 2 – `osFinal`

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

Listing 11: Level 2 – `osBody`

respectively. Similarly to axm7 for the save functions, axm8 models how `invTransform` converts a saved context into a context.

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 (Listing 8, 9, and 10).

Finally, the new event `osBody` models *kernel body* (Listing 11), abstractly representing the scheduler (OS4). We do not model the typical functionality of the *kernel body* (e.g., scheduling, resource management, etc.) in this work, but will refine it to guarantee its execution according to the OS requirements.

Level 3 Though we do not model *kernel body* with all its functionality, 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. Modifications of these variables in *kernel entry* and *exit* remain nondeterministic, since they are highly hardware-dependent. Listing 12 shows the new sets and variables, Listing 13 shows the new axioms, and Listing 14 shows the new guards for `osBody`. We also create

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

Listing 12: Level 3 – New carrier sets and variables

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

Listing 13: Level 3 – New axioms

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

Listing 14: Level 3 – New guards in `osBody`

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 (manually) copying it to the task's saved context in *kernel entry*. To model this, we partition CONTEXT and SAVEDCTX in three sections, as shown in Fig. 3b: MANUALCTX and MANUALSAVED for the locations that are 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` in Listing 8 and `loadSet` in Listing 9) 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 Fig. 5: `osAutoEntry` saves values from AUTODIRECTCTX into AUTODIRECTSAVE and copies from AUTOTEMPCTX to the new variable `temp` $\in$ TEMP $\rightarrow$ DATA. For architectures that only copy parts 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 TEMP\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 action that represents this is shown in Listing 15a. The structure `t_saved` gives us the locations where the context is saved for each task, and `rTask` represents the running task, whose context must be saved. The saved context is overwritten ($\Leftarrow$) in two steps: one takes the values from `cpuCtx` $\in$ CONTEXT $\rightarrow$ DATA, filtering only the MANUAL part with a domain restriction operator ($\lhd$), while the other takes the values from `temp` $\in$ TEMP $\rightarrow$ DATA, which has been written by `osAutoEntry` with the AUTOTEMPCTX values from `cpuCtx`. 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. The generic model we conclude in this Level is shown in Fig. 6.

5.3.2 Hardware-specific model instantiation

Level 5 This section details the instantiation of the context switch model, i.e., Level 5, and the hardware models for the target architectures we instantiate for, MSP430 and RISC-V.

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 target, we extend the Event-B context into hardware models. We 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`. The MSP430 model defines the CONTEXT subsets as shown in Listing 16a, while the same subsets are defined for the RISC-V in Listing 16b. Not shown here are the definitions of the SAVEDCTX subsets, which simply mirror the CONTEXT elements, and their relations. The relation `ctx2saved`, that maps each

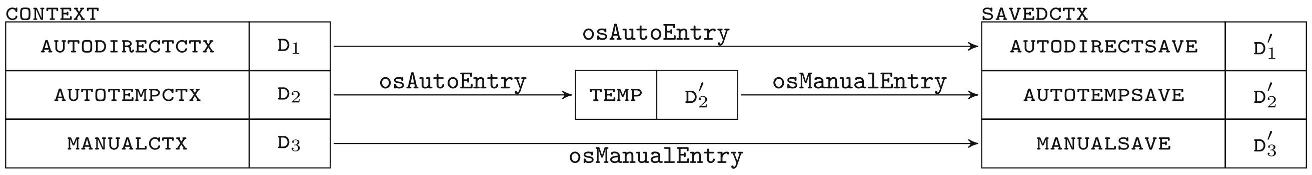


Fig. 5 : Context saving during *kernel entry*. $\mathtt{D}_x \subset \mathtt{DATA}$ and $\mathtt{D}'_x = \mathtt{ctxTransform(D}_x\mathtt{)}$

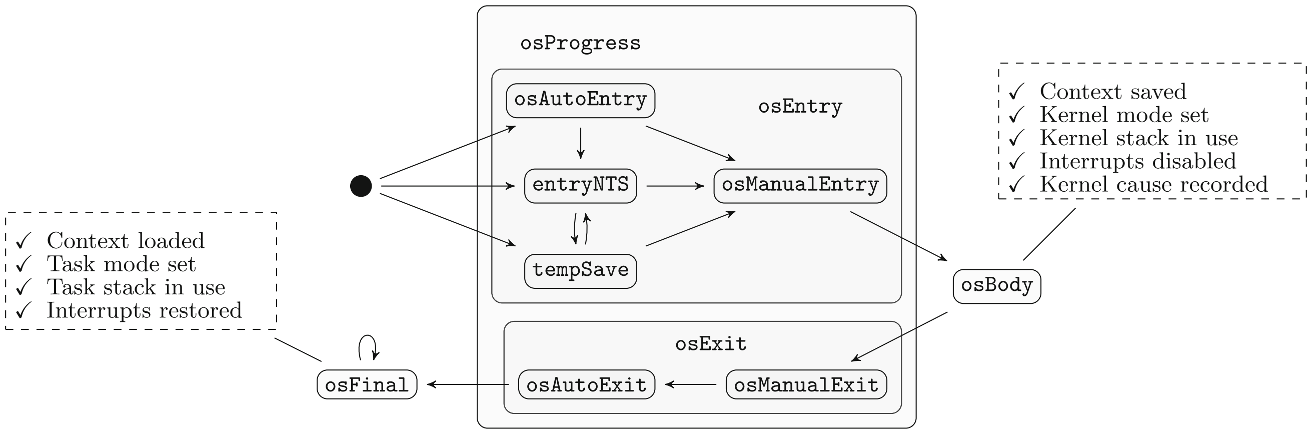
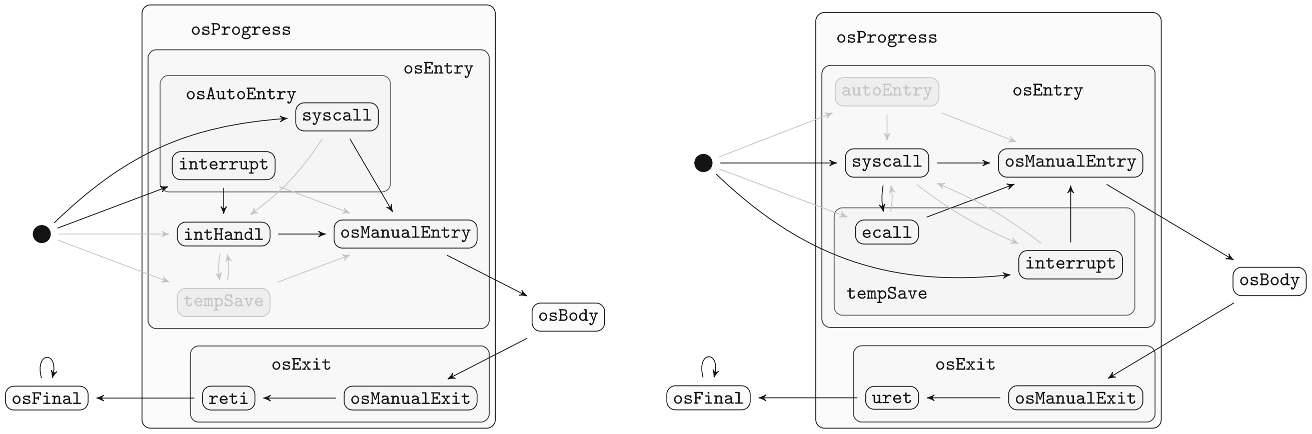


Fig. 6 : Level 4 – Most detailed but still hardware-independent (generic) model of the context switch



(a) MSP430 (b) RISC-V

Fig. 7 : Level 5 – target-specific refinements of Level 4 (Fig. 6)

CONTEXT element to its correspondent in SAVECTX, is only shown for the MSP430.

To instantiate the generic model, we refine, for each target, the last Level 4 machine. It `sees` the Event-B context correspondent to the target, and the architecture-specific actions from Level 3 are made deterministic.

MSP430. Figure 7a depicts the architecture-specific model for the MSP430. Comparing it to Fig. 6, one can see that the former is a target-specific refinement of the latter. The guards shown in the generic model are still present (and verified) in the architecture-specific model, but omitted in the figure for clarity. When the MSP430 accepts an interrupt, it pushes the return address (next program counter in R0) and status register (SR/R2) onto the stack, disables interrupts, and starts executing the corresponding interrupt handler. Because each task has its own stack, we can consider R0 and R2 as automatically saved on interrupt. Therefore, we refine `osAutoEntry` to represent the interrupt. The event will also be refined to represent the syscalls, becoming two different (but equivalent to their abstraction) events. Since no registers are temporarily saved by the MSP430, `tempSave` is removed. Once an interrupt is accepted, the MSP430 takes the interrupt handler's address from the corresponding index in the interrupt vector table and executes the handler. The

```
mSave:t_saved(rTask) := t_saved(rTask)
  ◁- (MANUALSAVED ◁ transform(cpuCtx))
  ◁- (autoTempTEMPSVDtransform(temp))
```

(a) Formal definition of the manual save action

```
t_saved(rTask)(mR1) := cpuCtx(R1)
t_saved(rTask)(mR4) := cpuCtx(R4)
t_saved(rTask)(mR5) := cpuCtx(R5)
...
```

(b) Unrolled save on MSP430

```
t_saved(rTask)(mX1) := cpuCtx(x1)
t_saved(rTask)(mX2) := cpuCtx(x2)
...
t_saved(rTask)(mMEPC) := cpuCtx(mepc)
t_saved(rTask)(mMSTAT) := cpuCtx(mstatus)
...
```

(c) Unrolled save on RISC-V

Listing 15: Generic (a) and target-specific unrolled (b,c) save actions

```
axm4:AUTODIRECTCTX = {R0,R2}
axm26:TEMP = ∅
axm3:AUTOTEMPCTX = ∅
axm5:MANUALCTX = {R1,R4,R5,R6,R7,R8,R9,
 R10,R11,R12,R13,R14,R15}
axm22:ctx2saved = {R0 ↦ mR0,R1 ↦ mR1,
 R2 ↦ mR2,R4 ↦ mR4,...,R15 ↦ mR15}
```

(a) MSP430

```
axm4:AUTODIRECTCTX = ∅
axm11:TEMP = {mepc,mstatus}
axm3:AUTOTEMPCTX = {invisiblePC,
 invisibleSTATUS}
axm5:MANUALCTX = {x1,x2,x3,x4,x5,x6,x7,x8
 ,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,
 x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,
 x29,x30,x31}
```

(b) RISC-V

Listing 16: Hardware models for CONTEXT in Level 5

MSP430 does not record the interrupt ID anywhere, so we must save it in the specific interrupt handler before proceeding to `osManualEntry`. We thus refine `entryNTS` to save the interrupt ID and rename it to `intHandler`.

The refinement of `osAutoEntry` to `syscall` executes the actions of both `interrupt` and `intHandler` events, i.e., it saves the AUTOCTX, disables interrupts, and stores the syscall ID. In `osManualEntry` the rest of the context is saved, the stack is switched to kernel stack, and kernel mode is set to true. Since the MSP430 does not have differently privileged modes, we simulate the switch into kernel by setting a simple variable.

RISC-V. The RISC-V architecture model is depicted in Fig. 7b. The interrupt concept of RISC-V is quite different: when an interrupt is accepted, the return address is stored in the special register `mepc`, the status in `mstatus`, and the interrupt number goes into `mcause`. The CPU then switches to a more privileged mode and continues execution from the address stored in the so called trap vector. We refine `kernelMode` to the privilege levels, adding an invariant that relates `kernelMode` = TRUE to machine mode and `kernelMode` = FALSE to user mode (OS11). Since the special registers where the AUTOCTX is stored are not task-specific, they belong to the AUTOTEMPCTX, and must be permanently saved later. Because nothing is automatically saved (but only "temporarily" stored), `tempSave` is refined to represent the interrupt and `osManualEntry` follows after the interrupt. Event `osAutoEntry` is never enabled and we can remove it.

For the syscalls, we need to be able to emulate the previously described interrupt process. The `ecall` instruction is designed exactly for that, as it causes an exception, just like a hardware interrupt. We refine `tempSave` again, now to represent the instruction `ecall`. However, there is a difference: what is recorded in `mcause` is not the ID of the syscall itself, as the architecture can only register the occurrence of the instruction. Since each syscall must pass its ID to the kernel and call `ecall` explicitly, we refine `entryNothingToSave` to do exactly that, also renaming it to `syscall`.

5.4 Proofs and model checking

This section shows the properties we verified via theorem proving in Rodin and LTL model checking.

From the requirements in Sect. 5.1.2, we elaborate some 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). Details for verifying the other OS requirements are omitted in this paper since the concept is the same. Instead, we also show how to verify some liveness properties: To guarantee that the model reaches the intended states, we verify that the steps resulting from the OS requirements are executed in the appropriate orders. (**L1**) The kernel executes in the correct order, and (**L2**) always finishes execution by always reaching `osFinal`.

5.4.1 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 execu-

Table 3 : Number of POs discharged

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%**

tion, namely that an old context is saved and a new context is loaded. This state must be reached by `osProgress`, which models the OS. Thus, 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 Sect. 5.4.2 we prove that 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 3 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 guarantee that the save and load processes do save `oldCtx` and load the `rTask`'s saved context:

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

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 an extra 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 manual save action from Level 4 (Listing 15a) simulates its abstract correspondent of Level 2 (Listing 8, Line 9). We replace the temporary 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◁ cpuCtx)
```

After proving the theorem, the SIM PO is much easier to discharge.

5.4.2 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: *entry* first, 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 complete verification. 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, where LTL found 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

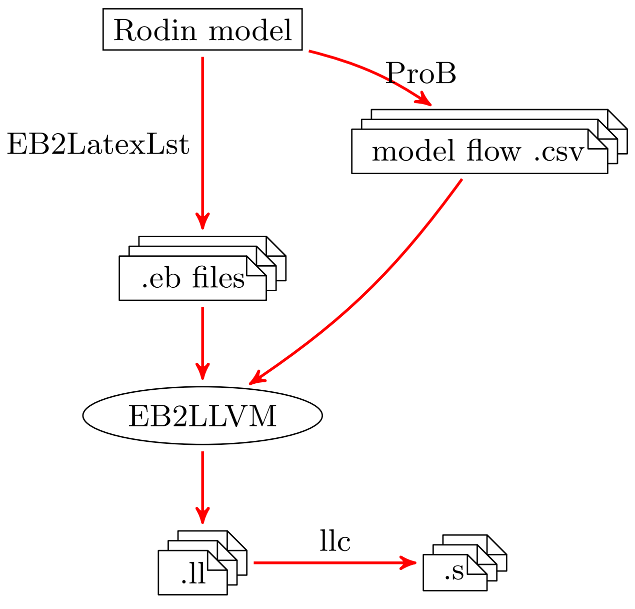


Fig. 8 : Overview of code generation

`entryNothingToSave` was introduced, prompting us to make it convergent and create a variant as explained in Level 4 (Sect. 5.3).

Model checking Level 4 also revealed that the execution order of events was 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 ∩ `toSave` = ∅ forces this order. Similarly, `osAutoExit` may only execute after all MANUALCTX is loaded (OS9), thus the new guard MANUALCTX ∩ `toLoad` = ∅ was introduced.

With these modifications to the models and the discharging of all proofs, we can formally prove that all requirements from Sect. 5.1 are fulfilled and the OS model is correct.

6 Code generation

With generic and target-specific models complete and verified, we can generate code for the target architectures. Figure 8 shows an overview of the code generation process. We start by exporting all Event-B contexts and machines to text format. The plain text format (`.eb`) allows EB2LLVM to work outside of Rodin. The Rodin plugin we use to export the model is a modified version of the B2Latex plugin [20], that we call EB2LatexLst.

Using Flex and Bison [25, 26], EB2LLVM can read and parse these `.eb` files into abstract syntax trees that it can process to generate code. For each concrete Event-B machine, EB2LLVM will create one LLVM IR file, containing one function named after the Event-B machine, with one code block for each of its events. In Event-B, control flow between events is not directly modeled, but defined within the event's guards. To extract this control flow information, we use the event enabling analysis of the model provided by ProB [47]. At the end of each block, compare and jump instructions are created to direct the flow to the appropriate next block.

Each block of code will be filled with code generated from the actions of the corresponding event. As in most code generators for Event-B, actions with basic arithmetics, variable assignment, etc. can be converted to their corresponding instructions. Additionally, EB2LLVM also supports array accesses, direct register accesses, and is capable of *unrolling* set expressions and relations, as well as *solving* equations[2]. For that, it uses definitions and declarations (axioms) from the concrete model and all its abstractions.

The code generation for the OS model will be presented in Sect. 6.2. Before, in Sect. 6.1, we give an overview of the entire framework on a generally known algorithm, the mathematical function factorial. The factorial example will show the more basic features of EB2LLVM, while the OS example digs deeper in the more complex features that allow EB2LLVM to generate code that communicates directly with the hardware, such as for the context switch.

6.1 Factorial example

The factorial of a natural number n, denoted $n!$, is the product of all natural numbers from 1 to n. The factorial of 0 is, by definition, $0! = 1$. Factorial is often implemented recursively, where `f(n) = n * f(n-1)`. Our Event-B model will use recursion for the model checking. However, the intended implementation is an iterative one, as Listing 17 shows. This C code is not relevant for code generation, but just to (1) present the algorithm in comparison to the model and (2) to compare the final binaries compiled from the C code and our generated LLVM IR.

Following our modeling strategy described in Sect. 5, the model's machine is shown in Listing 18. It contains the `fFinal` event that models the goal of the factorial function in its guards, and has no actions (`skip`). The guard [fFinal.modelGoal] assures the correctness of the calculated result. Since some parts of the model are not intended for code generation, such as this guard, any statement with the label starting with "model" will be ignored by EB2LLVM. The event where the factorial of `input` is calculated is `calc` (Listing 18, Line 16). This event is equivalent to the `while` block in Listing 17.

[2] Computational equation solving is a rather complex topic that we have not explored fully. EB2LLVM's capabilities are limited to the specific equations we needed for this proof of concept.

```
int input;
int cnt, res;

void factorial() {
  cnt = 0;
  res = 1;

  while(cnt < input) {
    cnt = cnt+1;
    res = cnt*res;
  }
}
```

Listing 17: C code of the factorial function. Parameters and return value passed via global variables

```
MACHINE factorial
SEES factorialCtx
VARIABLES
  cnt
  res
INVARIANTS
  cntType: cnt ∈ 0.. maxInput
  resType: res ∈ DATA
  resInv: res = fatFunc(cnt)
EVENTS
 INITIALISATION
  THEN
    initCnt:cnt := 0
    initRes:res := 1
  END
 calc
  WHERE
    loopCond:cnt < input
  THEN
    mulRes:res := (cnt + 1) ∗ res
    incCnt:cnt := cnt + 1
  END
 fFinal
  WHERE
    loopStop:cnt = input
    modelGoal:res = fatFunc(input)
  THEN
   skip
  END
END
```

Listing 18: Event-B machine of the factorial model

The model sets the function's pre-conditions in the axioms of Listing 19. Therefore, the generated function `factorial` will assume that these pre-conditions hold, and will not, e.g., check if `input` ∈ 0..`maxInput`.

Since the factorial model does not make any use of, e.g., registers or interrupts, its instantiation consists of only defining the `DATA` type as a function of the target's bit-width. It is important to notice that, currently, EB2LLVM only supports signed integers with the architecture's bit-width. Therefore, even though the model's variables could potentially be represented, e.g., by unsigned long integers, the instantiation must consider them signed integers. It follows that `DATA` $\in 0..(2^{b-1}-1)$, where b is the bit-width of the target. For the 16-bit MSP430, this means `maxInt` $= (2^{15}-1)$, while for the 32-bit RISC-V, `maxInt` $= (2^{31}-1)$.

```
CONTEXT factorialCtx
CONSTANTS
  DATA
  maxInt
  fatFunc
  input
  maxInput
AXIOMS
  maxIntType: maxInt ∈ ℕ
  dataType: DATA = 0..maxInt
  maxInt1: maxInt ≥ 1
  maxInType: maxInput ∈ DATA
  inParam: input ∈ 0..maxInput
  fType: fatFunc ∈ 0..maxInput → DATA
  fCalc: ∀ x · x ∈ 0..maxInput ∧ x > 0 ⇒
   fatFunc(x) = x ∗ fatFunc(x−1)
  fZero: fatFunc(0) = 1
  maxIn: fatFunc(maxInput) ≤ maxInt
END
```

Listing 19: Event-B context for the factorial model

We will now explain the LLVM IR generated from our models and shown in Listing 20 for the MSP430[3]: In the enabling analysis generated by ProB (Table 4) the connection between `fFinal` with itself is *syntactic_independent*, since once the event executes, the state does not change anymore. EB2LLVM interprets this as a return instruction. The *possible* connections in the enabling analysis will all generate conditional branches at the end of the origin block. The condition is generated from the guards of the destination. All possible destinations will be considered: having `INITIALIZATION` as origin, both `calc` and `fFinal` are *possible*. Therefore, two conditional branches will be generated. The first, in Listing 20 (Lines 7–10) branches to `calc` if $cnt < input$, as required by the guard [calc.loopCond]. Otherwise, it jumps to a newly created block `INITIALIZATION_fFinal` (Lines 32–36) that then jumps to `fFinal` if $cnt = input$ (Line 26). Otherwise, it jumps to the default `error` block (Line 29). Similarly, `calc` may either jump back to itself (Lines 21–24) or to `calc_fFinal` (Lines 38–42).

In this example, one can also notice several `load` and `store` instructions generated from Event-B variable reads and assignments. Basic arithmetics are also translated to their LLVM equivalents. In the factorial example, we have addition and multiplication being converted to, respectively, the instructions `add` and `mul`.

[3] The RISC-V code is the same, except for the bit-widths.

```
...
; Function Attrs: naked
define void @factorial() naked {
INITIALISATION:
  store i16 0, i16* @cnt, !label !{!"initCnt"}
  store i16 1, i16* @res, !label !{!"initRes"}
  %0 = load i16, i16* @cnt
  %1 = load i16, i16* @input
  %compare = icmp slt i16 %0, %1
  br i1 %compare, label %calc, label %INITIALISATION_fFinal, !label !{!"
   possible"}

calc:        ; preds = %calc, %INITIALISATION
  %2 = load i16, i16* @res
  %3 = load i16, i16* @cnt
  %4 = add i16 %3, 1
  %5 = mul i16 %4, %2
  store i16 %5, i16* @res, !label !{!"mulRes"}
  %6 = load i16, i16* @cnt
  %7 = add i16 %6, 1
  store i16 %7, i16* @cnt, !label !{!"incCnt"}
  %8 = load i16, i16* @cnt
  %9 = load i16, i16* @input
  %compare2 = icmp slt i16 %8, %9
  br i1 %compare2, label %calc, label %calc_fFinal, !label !{!"
   possible_disable"}

fFinal:      ; preds = %calc_fFinal, %INITIALISATION_fFinal
  ret void, !label !{!"syntactic_independent"}

error:       ; preds = %calc_fFinal, %INITIALISATION_fFinal
  ret void

INITIALISATION_fFinal: ; preds = %INITIALISATION
  %10 = load i16, i16* @cnt
  %11 = load i16, i16* @input
  %compare1 = icmp eq i16 %10, %11
  br i1 %compare1, label %fFinal, label %error, !label !{!"possible"}

calc_fFinal:   ; preds = %calc
  %12 = load i16, i16* @cnt
  %13 = load i16, i16* @input
  %compare3 = icmp eq i16 %12, %13
  br i1 %compare3, label %fFinal, label %error, !label !{!"possible_enable"}
}
```

Listing 20: MSP430 generated LLVM IR for the factorial model. The generated IR code for RISC-V is the same, except for the bit-widths.

The last step in the code generation is to compile the LLVM IR into assembly with `llc`, LLVM's static compiler. For the factorial example, we use the default optimization `-O2` to generate assembly code for the targets MSP430 and RISC-V. The generated code is shown in Listing 21a and Listing 22a respectively. The code generated from the Event-B model, can be compared to the code compiled from the C factorial function in Listing 17. For the MSP430, we used the `MSPgcc 6.2.1` compiler, and the resulting assembly is shown in Listing 21b. A `clang` compiler was available for RISC-V, and we used version `RISC-V rv32gc clang 9.0.0` (with the `-march=rv32i` option), the same LLVM version we use in our code generation. The compiled code is shown in Listing 22b. Both used the same level of optimization `-O2`.

The code compiled from C for the MSP430 (Listing 21b) is a few instructions longer than the one we generated from the Event-B model (Listing 21a). One of the reasons is that the `gcc` compiler inserted prologue and epilogue to save and restore registers used within the function. Since our code generator aims at generating exactly what is modeled, these

Table 4 Control flow generated by ProB

Origin/destination	calc	fFinal
INITIALISATION	possible	possible
calc	possible_disable	possible_enable
fFinal	syntactic_unchanged	syntactic_independent

```
 1 factorial:               ; @factorial
 2 ; %bb.0:                 ; %INITIALISATION
 3 mov #1, &res
 4 clr &cnt
 5 cmp #1, &input
 6 jl .LBB0_2
 7 .LBB0_1:                 ; %calc
 8  ; =>This Inner Loop Header: Depth=1
 9 mov #1, r10
10 add &cnt, r10
11 mov &res, r13
12 mov r10, r12
13 call #__mspabi_mpyi
14 mov r12, &res
15 mov r10, &cnt
16 cmp &input, r10
17 jl .LBB0_1
18 .LBB0_2:                 ; %calc_fFinal
19  cmp &input, &cnt
20 ret
```

(a) Compiled from auto-generated IR for MSP430 (Event-B $\xrightarrow{\text{EB2LLVM}}$ LLVM $\xrightarrow{\text{llc}}$ ASM)

```
factorial():
PUSHM.W #4, R10
PUSHM.W #1, R4
MOV.W   R1, R4
MOV.W   #0, &cnt
MOV.W   #1, &res
MOV.W   &input, R7
MOV.B   #0, R12
CMP.W   R7, R12
JGE .L1
MOV.W   R7, R8
ADD.W   #1, R8
MOV.B   #1, R10
MOV.W   R10, R12
MOV.W   #__mspabi_mpyi, R9
.L3:
MOV.W   R10, R13
CALL    R9
ADD.W   #1, R10
CMP.W   R8, R10
JNE .L3
MOV.W   R7, &cnt
MOV.W   R12, &res
.L1:
POPM.W  #1, r4
POPM.W  #4, r10
RET
```

(b) Compiled from the C code for MSP430 (C $\xrightarrow{\text{gcc}}$ ASM)

Listing 21: MSP430 assembly code for factorial

registers are silently overwritten.[4] Further, the gcc version has more instructions to prepare for the while loop, however the instructions sequence inside the loop is more optimized.

A note about the compare instruction in Listing 21a, Line 19: our code generator correctly generated it from Listing 20, Lines 39 to 41, however the corresponding branch in Line 42 was omitted. This is functionally correct, since both destinations of this branch (fFinal and error) would result in the execution of a return instruction, which follows the compare in the generated assembly code. However, this renders the compare instruction useless. A similar situation happens in the code generated for RISC-V: all the load instructions between Lines 24 and 31 in Listing 22a are generated from Listing 20, Lines 39 to 41, but never used in the compare/branch for which they were intended.

[4] Support for function parameters and return values are future work for EB2LLVM.

Also similarly to the MSP430, the assembly code generated by clang for the RISC-V (Listing 22b) has prologue and epilogue, while the code generated from Event-B (Listing 22a) does not. The while loop is also more optimized, and instead of storing partial results in each iteration, clang only stores the end values before return. Though slightly less efficient, the code generated from Event-B for both MSP430 and RISC-V can be compiled into binaries that execute as intended.

6.2 OS code generation

Generating code for our context switch model requires more complex features from EB2LLVM. Since the context switch must communicate with the hardware directly, the code generator must know how to, e.g., translate and interpret register accesses, interrupts, and specific instructions. The model also uses array accesses, sets and relations, and equations that

```
factorial: # @factorial
 .cfi_startproc
# %bb.0:   # %INITIALISATION
 lui s2, %hi(cnt)
 sw zero, %lo(cnt)(s2)
 lui s1, %hi(res)
 addi a0, zero, 1
 sw a0, %lo(res)(s1)
 lui s3, %hi(input)
 lw a1, %lo(input)(s3)
 blt a1, a0, .LBB0_3
.LBB0_1:   # %calc
# =>This Inner Loop Header: Depth=1
 lw a0, %lo(cnt)(s2)
 addi s0, a0, 1
 lw a1, %lo(res)(s1)
 mv a0, s0
 call __mulsi3
 sw a0, %lo(res)(s1)
 sw s0, %lo(cnt)(s2)
 lw a0, %lo(input)(s3)
 blt s0, a0, .LBB0_1
# %bb.2:   # %calc_fFinal
  lui a0, %hi(input)
 lw a0, %lo(input)(a0)
 lui a1, %hi(cnt)
 lw a1, %lo(cnt)(a1)
 ret
.LBB0_3:   # %INITIALISATION_fFinal
 lw a0, %lo(input)(s3)
  lw a1, %lo(cnt)(s2)
 ret
```

(a) Compiled from auto-generated IR for RISC-V (Event-B $\xrightarrow{\text{EB2LLVM}}$ LLVM $\xrightarrow{\text{llc}}$ ASM)

```
factorial(): # @factorial()
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      s0, 8(sp)
        sw      s1, 4(sp)
        lui     a0, %hi(cnt)
        sw      zero, %lo(cnt)(a0)
        lui     a0, %hi(res)
        addi    a1, zero, 1
        sw      a1, %lo(res)(a0)
        lui     a0, %hi(input)
        lw      s1, %lo(input)(a0)
        blt     s1, a1, .LBB0_4
        mv      s0, zero
.LBB0_2: # =>This Inner Loop Header:
 Depth=1
        addi    s0, s0, 1
        mv      a0, s0
        call    __mulsi3
        mv      a1, a0
        blt     s0, s1, .LBB0_2
        lui     a0, %hi(res)
        sw      a1, %lo(res)(a0)
        lui     a0, %hi(cnt)
        sw      s0, %lo(cnt)(a0)
.LBB0_4:
        lw      s1, 4(sp)
        lw      s0, 8(sp)
        lw      ra, 12(sp)
        addi    sp, sp, 16
        ret
```

(b) Compiled from the C code for RISC-V (C $\xrightarrow{\text{clang}}$ ASM)

Listing 22: RISC-V assembly code for the factorial function

need to be used in the code generation, as opposed to the factorial model, where these were only used for the model verification.

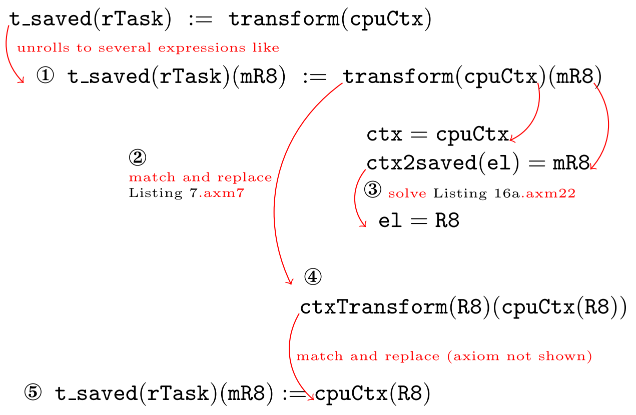
For the two architectures shown in Fig. 7, each inner-most box (representing the concrete events) results in one block of code. They are executed sequentially, and the enabling analysis from ProB reflects that dubbing each connection *guaranteed*, while other possible connections are *impossible*. The *guaranteed* connections generate unconditional branches from the origin block to the destination.

In our models, events model what the hardware does when an interrupt occurs and when a return from interrupt instruction is issued. The actions within these events should, therefore, not produce any code. Furthermore, an event modeling a specific instruction, such as the return from interrupt, should be translated to the equivalent instruction. This is achieved by using pre-defined names for these events to indicate to the code generator which instruction to use. For example, the code generator will ignore events called `interrupt`, and automatically generate a return from interrupt instruction when it finds an event named `reti`.

The context switch model is much larger than the factorial, and, to demonstrate its code generation, we selected one crucial snippet of it: the save action from Listing 15a. Besides showing the remaining features of EB2LLVM, this is also the action that allows the model to remain mostly generic up to refinement Level 4 and yet detailed enough to generate architecture-specific code from refinement Level 5. In the save action, the context is saved into `t_saved`. We know from the model that `t_saved` is a two-dimensional array, with one block of SAVEDCTX elements for each task in the system (see Fig. 3b). The running task `rTask` is our reference here, and each context element is to be stored in the array `t_saved[rTask]`. EB2LLVM will use the HW models (Listing 16) to unroll the action into several store instructions, according to the other elements of the expression. It applies `transform` to save each `cpuCtx` and `temp` value into their appropriate saved context index, filtering only MANUAL* `cpuCtx` values due to the restriction operator (◁).

The results of this process for both MSP430 and RISC-V are shown in Listing 15b and Listing 15c, respectively.

However, the definition of `transform` involves further axioms and its evaluation during code generation is rather complex. Listing 23 shows it step by step for saving one MSP430 register. First, the save action is unrolled, resulting in several expressions like ①. Then, the code generator matches the right-hand side (②) to `axm7` from Listing 7, also using the hardware-specific definition of `ctx2saved` (`axm22` in Listing 16a) to solve ③. Finally, `ctxTransform` (④) is evaluated in a similar way. In this particular case it does nothing (i.e., `ctxTransform(el)(ctx(el))` = `ctx(el)`), so the final expression that will be translated into LLVM IR shows in ⑤. If a modification was defined by `ctxTransform` in the model, it would be taken into account and the appropriate instruction would be generated.


t_saved(rTask) := transform(cpuCtx)
unrolls to several expressions like
① t_saved(rTask)(mR8) := transform(cpuCtx)(mR8)
ctx = cpuCtx
ctx2saved(el) = mR8
② match and replace Listing 7.axm7
③ solve Listing 16a.axm22
el = R8
④
ctxTransform(R8)(cpuCtx(R8))
match and replace (axiom not shown)
⑤ t_saved(rTask)(mR8) := cpuCtx(R8)

Listing 23: Internal evaluation of the save action

The save operation can then be translated into LLVM IR[5] as shown in Listing 24. The register access `cpuCtx(R8)` (Listing 23, ⑤) is converted to a call to the intrinsic LLVM function `read_register` (→ Line 1). In Lines 4 and 7 we can see that the accesses to elements in a relation are converted to LLVM's GEP instruction[6], and the value from the register is finally stored in the appropriate array position `mR8`. Note that the save index `mR8` is defined, in the model, as a constant value, and therefore expressed as the immediate value `5` in Line 7. This is repeated for each element that must be saved, according to the model. The LLVM IR code for RISC-V is similar to Listing 24: The register name and its index, as well as the data sizes are replaced according to the RISC-V model.

```
1 %R8 = call i16 @llvm.read_register.i16(r8)
2 %rTsk = load i16, i16* @rTask
3 // t_saved[rTask]
4 %gepRT = getelementptr i16, i16* @t_saved,
5  i16 %rTsk
6 // t_saved[rTask][mR8], with mR8=5
7 %gepMR8 = getelementptr i16, i16* %gepRT,
8  i16 5
9 store i16 %R8, i16* %gepMR8
```

Listing 24: LLVM IR generated from Listing 23 , ⑤

```
1 mov &rTask, r12
2 add r12, r12
3 mov r8, t_saved
  +10(r12)
```

(a) MSP430: save `r8` (see Listing 24)

```
1 lui a0, %hi(t_saved)
2 addi a0, a0, %lo(t_saved)
3 lui a1, %hi(rTask)
4 lw a1, %lo(rTask)(a1)
5 slli a1, a1, 2
6 add a0, a0, a1
7 sw s0, 28(a0)
```

(b) RISC-V: save register `x8` (`s0`)

Listing 25: Generated assembly code to save one register

Finally, assembly code is generated from the automatically generated LLVM IRs by `llc`. This time, we turn off optimizations (`-O0`) to assure that unintended modifications to the model's logic will not occur. For the MSP430, each block that saves one register (such as in Listing 24) is translated to assembly as shown in Listing 25a. The index for the array access in Line 3 is automatically calculated by `llc` as $\mathtt{mR8} * ByteWidth$, and since the MSP430 is a 16-bit (2 bytes) architecture: $5*2 = 10$ (Line 3). Listing 25b shows the RISC-V generated code to save register `x8` in the save index $\mathtt{mX8} = 7$. Our model only uses the architecture's specified register names, but `llc` uses the Abstract Binary Interface (ABI) names (`x8` = `s0`). The index calculated for RISC-V's 32-bit (4 bytes) architecture is $7 * 4 = 28$ (Line 7).

The generated code could still be optimized: Lines 1 to 2 in Listing 25a and Lines 1 to 6 in Listing 25b are repeated for each register saved, though they could be present only once. Furthermore, while the model assures, for example, that no register will be overwritten before it is saved, the code generator might need to use temporary variables for temporary storage of register values. Thus, a tighter integration between EB2LLVM and `llc` is needed to assure that any modification caused by a generated instruction is intentional according to the model.

7 Conclusion and outlook

We have presented our framework for OS portability based on formal methods and code generation.

First, with regard to RQ1 from the introduction, we have shown a generic formal RTOS model in Event-B with context switches that decouples low-level functionality from hard-

[5] To make the LLVM IR listings more readable, we have replaced metadata with their actual values, renamed virtual registers to better reflect what they represent, and added comments to some lines.

[6] The `getelementptr` (GEP) instruction is often misunderstood, but here it is enough to understand it as calculating the address of an array element.

ware 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. This verification also helped us detect and eliminate issues early in the design.

Within the proof of concept for the next stage of the framework, and with regard to RQ2 from the introduction, we showed how code is automatically generated from the models: EB2LLVM generates LLVM IR code from the verified models. The IR is then compiled with the target-specific compiler backend, generating assembly code for the context switch of SmartOS. Additionally, we presented another, hardware-independent model, applying the entire framework on the generally known high-level algorithm to calculate the factorial of a number. We compared the automatically generated code with equivalent code compiled from the function implemented in C. Though less efficient, the code generated from Event-B for both MSP430 and RISC-V can be compiled into binaries and work as intended, successfully generating a binary that corresponds to the model.

An important aspect of our approach is to initially keep the software models and the hardware models independent from each other. From a potential pool of OS models and architecture models we can then pick one, respectively, and only combine them in a last instantiation step before code generation. This way, most of the modeling and verification efforts are concentrated in the generic parts of the models, and supporting a new architecture requires much less effort compared to traditional approaches, where hardware and software are inherently mixed in one model which must then be created from scratch. In addition, the automatic code generation from one OS model to many architectures assures consistency between the ports, and makes it easier to adapt to new or changing OS specifications. The framework can thus accelerate the optimization of the software, as different design options can be quickly evaluated on real hardware without the need to adapt all ports individually and manually. This even shows that RQ1 and RQ2 can be addressed in combination.

Formal modeling of software is still not very common in real-world projects – especially when it comes to generating executable code. The effort often seems too high compared to traditional implementation and acceptance on the developers' side must first be built. However, studies have already shown that formal methods can drastically increase software quality and have a great positive impact on its overall dependability. In fact, we have shown how the same formal models can be be used for both code generation (RQ2) *and* verification (RQ1). On the long-term, we believe that the approach will essentially simplify the creation and maintenance of software. This will not only save costs, but it will also be beneficial for future systems in various domains (e.g., automotive, avionics, the IoT, etc.), where guaranteed dependability is crucial. 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. Therefore, we continue our work to support more target architectures and to verify the generated code.

With regard to the validity and applicability of our approach, a potential issue we must mention is the time and computation power required for model checking. The axioms in the presented model already cause a 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 of, e.g., bigger and more complex OS kernels or targets, even a minimal set might 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.

We have already investigated more complex architectures, such as the Infineon Aurix [35]. In fact, though the work is not yet complete, our initial models do already consider some Aurix-specific concepts, and show the general applicability of our approach. In order to actually generate code for it, the instantiation step will likely be more complex than for the two simple architectures we have presented in this work. Since the Aurix supports special instructions and a linked list data structure in hardware to handle contexts, the code generator would also have to be extended to generate the appropriate code.

In any case, we still need to work on verifying the code generator itself, so we can guarantee the correct translation of the model into LLVM IR. Regarding the final compilation step into an executable binary, we rely on the LLVM community, where several works already try to provide verification for the compiler backend [3, 40, 43]. The verification of the toolchain parts we have developed is still open and not part of this work.

Apart, we have already started to extend our modeling concept to also support the verification of (non-)functional aspects in (other) OS kernels and application software, such as timing and liveness. We are therefore working on modeling timed automata to analyze the interaction of the application and OS layers through syscalls as well as on the response times of kernel functions and concurrently running tasks [62]. A possible threat to the applicability of our approach to general OSs could be their sheer set of features and the commonly very complex interactions in between. Regarding SmartOS, we aim to model the entire kernel by separating its parts and features into distinct model modules with well-defined input/output interfaces. This way, we can model and verify all modules independently first (e.g., the *kernel body*), and only later verify their interaction (e.g., with the context switch model presented here). Early experiences with AUTOSAR [70] have shown that other low-level functionality of the OS, such as device drivers, can also be modeled and verified similarly to the context switch presented here. The individually verified model modules would then be combined for the verification of overarching aspects in complete software stacks.

Funding Open access funding provided by Graz University of Technology.

References

1. Abrial, J.-R.: The B Book-Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
3. Ahmed, A.: Verified compilers for a multi-language world. In: Ball, T., Bodik, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds) 1st Summit on Advances in Programming Languages (SNAPL 2015), volume 32 of Leibniz International Proceedings in Informatics (LIPIcs), pp 15–31, Dagstuhl, Germany, (2015). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik
4. Akdur, D., Garousi, V., Demirörs, O.: A survey on modeling and model-driven engineering practices in the embedded software industry. J. Syst. Architect. **91**, 62–82 (2018)
5. Alkhammash, E.H., Butler, M.J., Cristea, C.: International Conference on Communication, Management and Information Technology, Chapter Modeling Guidelines of FreeRTOS in Event-B, pp. 453–462. CRC Press (2017)
6. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: tooling concepts for seamless, model-based development of embedded systems. In: ACES-MB&WUCOR@MoDELS 2015, CEUR Workshop Proceedings, pp. 19–26. CEUR-WS.org (2015)
7. Basile, D., ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F., Piattino, A., Trentini, D., Ferrari, A.: On the industrial uptake of formal methods in the railway domain. In: Furia, C.A., Winter, K. (eds.) Integrated Formal Methods, pp. 20–29. Springer, Cham (2018)
8. Besnard, V., Jouault, F., Brun, M., Teodorov, C., Dhaussy, P., Delatour, J.: Modular deployment of uml models for v&v activities and embedded execution. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20, New York. Association for Computing Machinery (2020)
9. Bodenstab, D.E., Houghton, T.F., Kelleman, K.A., Ronkin, G., Schan, E.P.: The UNIX system: UNIX operating system porting experiences. AT T Bell Lab. Tech. J. **63**(8), 1769–1790 (1984)
10. Brandenburg, B.B.: The case of an opinionated, theory-oriented real-time operating system. NGOSCPS19, 04 (2019)
11. Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.-F., Voisin, L.: The first twenty-five years of industrial use of the B-Method. In: ter Beek, M.H., Ničković, D. (eds.) Formal Methods for Industrial Critical Systems, pp. 189–209. Springer, Cham (2020)
12. Cheng, S., Woodcock, J., D'Souza, D.: Using formal reasoning on a model of tasks for FreeRTOS. Formal Aspects Comput. **27**(1), 167–192 (2014)
13. Cho, D., Bae, D.: Case study on installing a porting process for embedded operating system in a small team. In: 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion, pp. 19–25 (2011)
14. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. SIGOPS Oper. Syst. Rev. **35**(5), 73–88 (2001)
15. Craig, I.D.: Formal Refinement for Operating System Kernels. Springer, New York, Secaucus (2007)
16. Craig, I.D.: Formal Models of Operating System Kernels, 1st edn. Springer Publishing Company, Incorporated, Berlin (2010)
17. Dalvandi, M., Butler, M.J., Fathabadi, A.S.: SEB-CG: code generation tool with algorithmic refinement support for event-b. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds) Formal Methods. FM 2019 International Workshops-Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I, volume 12232 of Lecture Notes in Computer Science, pp 19–29. Springer (2019)
18. Danmin, C., Yue, S., Zhiguo, C.: A formal specification in b of an operating system. Open Cybern. System. J. **9**(1), 1125–1129 (2015)
19. 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 (2016)
20. Event-B. B2Latex—Event-B. https://wiki.event-b.org/index.php/B2Latex
21. Event-B. Event-B and the Rodin Platform. www.event-b.org
22. 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. J. Syst. Architect. **82**, 12–23 (2018)
23. 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 (2015)
24. Gabel, M., Yang, J., Yu, Y., Goldszmidt, M., Su, Z.: Scalable and systematic detection of buggy inconsistencies in source code. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA10, pp. 175–190. Association for Computing Machinery, New York (2010)
25. GNU Project. Bison. https://www.gnu.org/software/bison/
26. GNU Project. The fast lexical analyzer. https://github.com/westes/flex

27. Gomes, R.M., Aichernig, B., Baunach, M.: A formal modeling approach for portable low-level OS functionality. In: de Boer, F., Cerone, A. (eds) Software Engineering and Formal Methods, pp.155–174. Springer, Cham (2020)
28. Gomes, R.M., Baunach, M.: A framework for OS portability: from formal models to low-level code. In: The 37th ACM/SIGAPP Symposium On Applied Computing (2022)
29. 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 (2015)
30. Goranko, V., Galton, A.: Temporal logic. In: Zalta, E.N. (ed) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, winter 2015 edition (2015)
31. Gu, R., Shao, Z., Chen, H., Wu, X.(Newman), Kim, J. Sjöberg, V., Costanzo, D.: Certikos: an extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 653–669. USENIX Association, Savannah (2016)
32. Hahm, O., Baccelli, E., Petersen, H., Tsiftes, N.: Operating systems for low-end devices in the internet of things: a survey. IEEE Internet Things J. **3**(5), 720–734 (2016)
33. Holland, D.A.: Toward Automatic Operating System Ports via Code Generation and Synthesis. Ph.d thesis (2020)
34. 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, PLOS19, pp. 67–73. Association for Computing Machinery, New York (2019)
35. Infineon. AURIX TriCore Microcontroller. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/
36. Instruments, Texas, MSP430 ultra-low-power sensing and measurement MCUs (2019)
37. Jastram, M., Butler, P.M.: Rodin user's handbook: covers Rodin V.2.8. USA (2014)
38. Jiang, L., Su, Z., Chiu, E.: Context-based detection of clone-related bugs. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 55–64. Association for Computing Machinery, New York, (2007)
39. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: 2009 IEEE 31st International Conference on Software Engineering, pp. 485–495 (2009)
40. Kang, J., Kim, Y., Song, Y., Lee, J., Park, S., Shin, M.D., Kim, Y., Cho, S., Choi, J., Hur, C.-K., Yi, K.: Crellvm: verified credible compilation for llvm. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, pp. 631–645. Association for Computing Machinery, New York (2018)
41. Kleen, A.: Porting linux to x86-64. In: Proceedings of the Linux Symposium (2001)
42. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1) (2014)
43. Lammich, P.: Generating Verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds) 10th International Conference on Interactive Theorem Proving (ITP 2019), volume 141 of Leibniz International Proceedings in Informatics (LIPIcs), pp. 22:1–22:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2019)
44. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **SE–3**(2), 125–143 (1977)
45. Lattner, C.: Introduction to the LLVM compiler system. In: Advanced Computing and Analysis Techniques in Physics Research (ACAT) (2008)
46. Lecomte, T., Deharbe, D., Prun, E., Mottin, E.: Applying a formal method in industry: a 25-year trajectory. In: Cavalheiro, S., Fiadeiro, J. (eds.) Formal Methods: Foundations and Applications, pp. 70–87. Springer, Cham (2017)
47. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
48. Lewis, B.: Software portability gains realized with METAH and Ada95. In: Proceedings of the 11th International Workshop on Real-time Ada Workshop, IRTAW '02, pp. 37–46. ACM, New York (2002)
49. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In: OSdi, vol. 4, pp. 289–302 (2004)
50. llvm-admin team. The LLVM compiler infrastructure. https://llvm.org/
51. Lyons, A., Danis, A., Yyshen, A., Hesham, S., Stephen, Z., Amirreza, M., Kent, K., Gerwin, P., Latent, B., Joel, S., Thomas, A., Kolanski, R., Boettcher, A., Susarla, P., Brecknell, M., Waugh, J., Holzapfel, S., Guikema, C., Richardson, C., Cloudier, V., Robbie, M., Mokshasoft, N., Tim, M., Luke, M., Jesse, J., Studer, N., Millar, C.: seL4/seL4: MCS pre-release (2018)
52. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, Berlin (2012)
53. Martins, G. Renata, B.M.: A study on the portability of iot operating systems. In: Tagungsband des FG-BS Frühjahrstreffens 2021, Bonn (2021). Gesellschaft für Informatik e.V
54. MATLAB. The MathWorks Inc., Natick (2010)
55. Méry, D., Singh, N.K.: Automatic code generation from event-b models. In: Proceedings of the Second Symposium on Information and Communication Technology, SoICT '11, pp. 179–188. ACM, New York (2011)
56. 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, Cham (2018)
57. Nyberg, M., Gurov, Dilian, L., Christian, R., Andreas, W.J.: Formal verification in automotive industry: enablers and obstacles. In: Margaria, T., Steffen, B. (eds) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, pp. 139–158. Springer, Cham (2018)
58. Oikonomou, G., Phillips, I.: Experiences from porting the Contiki operating system to a popular hardware platform. In: 2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS), pp. 1–6 (2011)
59. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57 (1977)
60. 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 (2016)
61. Ray, B., Kim, M., Person, S., Rungta, N.: Detecting and characterizing semantic inconsistencies in ported code. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 367–377 (2013)
62. Ribeiro, L.B., Lorber, F., Nyman, U., Larsen, G., Baunach, M.: A modeling concept for formal verification of os-based compositional software. In: Under review at 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2022)
63. RISC-V Foundation. RISC-V
64. Rivera, L.F., Villegas, N.M., Tamura, G., Jiménez, M., Müller, H.A.: Uml-driven automated software deployment. In: Proceed-

ings of the 28th Annual International Conference on Computer Science and Software Engineering, CASCON '18, pp. 257–268. IBM Corp, USA (2018)
65. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for event-b. Int. J. Softw. Tools Technol. Transf. **19**(1), 31–52 (2017)
66. Scheipel, T., Batista Ribeiro, L., Sagaster, T., Baunach, M.: SmartOS: An OS architecture for sustainable embedded systems. In: Tagungsband des FG-BS Frühjahrstreffens 2022. Gesellschaft für Informatik e.V, Bonn (2022)
67. Smith, R., Smith, G., Wardani, A.: Software reuse in robotics: enabling portability in the face of diversity. In: IEEE Conference on Robotics, Automation and Mechatronics, 2004., vol. 2, pp. 933–938 (2004)
68. Sritharan, S., Hoang, T.S.: Towards generating spark from event-b models. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods, pp. 103–120. Springer, Cham (2020)
69. Stan, A.: Porting the core of the Contiki operating system to the TelosB and MicaZ platforms. International University, Bremen, Bachelor thesis (2007)
70. Staron, M., Durisic, D.: AUTOSAR standard. In: Automotive Software Architectures, pp. 81–116. Springer (2017). https://doi.org/10.1007/978-3-319-58610-6_4
71. 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 (2006)
72. 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 (2015)
73. Syeda, H.T., Klein, G.: Formal reasoning under cached address translation. J. Autom. Reason. (2020)
74. Takata, H., Sugai, N., Yamamoto, H.: Porting Linux to the M32R processor. In: Lockhart, J.W. (ed) Linux Symposium, pp. 398. The Linux Foundation (2003)
75. Torvalds, L.: Linux: a Portable Operating System. Master's thesis, University of Helsinki (1997)
76. Verhulst, E., Boute, R.T., Faria, J.M., Sampaio, S., Bernhard, M.: Vitaliy: Formal Development of A Network-Centric RTOS. Springer, New York (2011)
77. Waterman, A., Asanović, K.: The RISC-V instruction set manual volume I: user-level ISA version 2.2 (2017)
78. Waterman, A., Lee, Y., Avizienis, R., Patterson, D.A., Asanović, K.: The RISC-V instruction set manual volume II: privileged architecture version 1.7. Technical Report UCB/EECS-2015-49, EECS Department, University of California, Berkeley (2015)
79. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 19:1-19:36 (2009)
80. Wright, S.: Formal construction of instruction set architectures. Ph.d. thesis, University of Bristol (2009)
81. Wright, S.: Automatic generation of C from Event-B. In: Workshop on integration of model-based formal methods and tools, pp. 14 (2009)
82. Zhang, F., Niu, W.: A survey on formal specification and verification of system-level achievements in industrial circles. Acad. J. Comput. Inf. Sci. (2019)
83. Zhang, S., Kobetski, A., Johansson, E., Axelsson, J., Wang, H.: Porting an AUTOSAR-compliant operating system to a high performance embedded platform. SIGBED Rev. **11**(1), 62–67 (2014)
84. Zhou, Z., Liang, B., Jiang, L., Shi, W., He, Y.: A formal description of SECIMOS operating system. In: Gorodetsky, V., Kotenko, I., Skormin, V. (eds) Computer Network Security, pp. 286–297. Springer, Berlin, Heidelberg (2005)

Renata Martins Gomes received her engineer's degree in Computer Engineering at Universidade Federal de Itajuba, Brazil, and her Ph.D. with distinction in Information and Communications Engineering at Graz University of Technology, Austria. Her research interests focus on portability and correctness of real-time operating systems, including code generation and formal verification of hardware and software models. As software engineer in industry, she focuses on compositional and RTOS-based embedded systems.

Bernhard Aichernig is a tenured associate professor (ao. Univ.-Prof.) at Graz University of Technology, Austria. He and his research group investigates the foundations of software engineering for realising dependable computer-based systems. Bernhard is an expert in formal methods and testing. His research covers a variety of areas combining falsification, verification and abstraction techniques. Current topics include the Internet of Things, model learning, and statistical model checking. Bernhard holds a habilitation in Practical Computer Science and Formal Methods, a doctorate, and a diploma engineer degree from Graz University of Technology.

Marcel Baunach is professor for Embedded Automotive Systems and head of the EAS Group at the Institute of Technical Informatics at Graz University of Technology, Austria. His research areas include hardware and software for highly dependable and sustainable embedded systems. The focus is on real-time operating systems and processor architectures, in particular on concepts for the design, verification and portability of system software, dynamic composition of modular software, as well as application-specific and reconfigurable processors. Application areas include, e.g., future vehicles, the Internet of Things or cyber-physical systems.