



# A Modeling Concept for Formal Verification of OS-Based Compositional Software

Leandro Batista Ribeiro<sup>1</sup> (✉), Florian Lorber<sup>2</sup>, Ulrik Nyman<sup>2</sup>,  
Kim Guldstrand Larsen<sup>2</sup>, and Marcel Baunach<sup>1</sup>


<sup>1</sup> Graz University of Technology, Graz, Austria  
{lbatistaribeiro,baunach}@tugraz.at

<sup>2</sup> Aalborg University, Aalborg, Denmark  
{florber,ulrik,kgl}@cs.aau.dk

**Abstract.** The use of formal methods to prove the correctness of compositional embedded systems is increasingly important. However, the required models and algorithms can induce an enormous complexity. Our approach divides the formal system model into layers and these in turn into modules with defined interfaces, so that reduced formal models can be created for the verification of concrete functional and non-functional requirements. In this work, we use UPPAAL to (1) model an RTOS kernel in a modular way and formally specify its internal requirements, (2) model abstract tasks that trigger all kernel functionalities in all combinations or scenarios, and (3) verify the resulting system with regard to task synchronization, resource management, and timing. The result is a fully verified model of the operating system layer that can henceforth serve as a dependable foundation for verifying compositional applications w.r.t. various aspects, such as timing or liveness.

**Keywords:** Embedded Systems · Real-Time Operating Systems · Formal Methods · Uppaal · Software Composition.

## Availability of Artifacts

All UPPAAL models and queries are available at <https://doi.org/10.6084/m9.figshare.21809403>. Throughout the paper, model details are omitted for the sake of readability or due to space constraints. In such cases, the symbol  indicates that details can be found in the provided artifacts.

## 1 Introduction

Embedded systems are everywhere, from simple consumer electronics (wearables, home automation, etc.) to complex safety-critical devices. e.g., in the automotive, aerospace, medical, and nuclear domains. While bugs on non-critical devices are at most inconvenient, errors on safety-critical systems can lead to catastrophic

consequences, with severe financial or even human losses [19,21]. Therefore, it is of utmost importance to guarantee dependable operation for safety-critical systems at all times. Common practice in industry to validate safety-critical systems is still extensive testing [4]. However, this approach only proves the absence of errors in known cases, but it cannot prove general system correctness.

While general correctness can be proven with formal methods, they still face resistance from practitioners [24], as they are considered resource-intensive and difficult to integrate into existing development processes [14]. However, potential cost reduction or strict regulations might contribute to their adoption. For example, the use of formal methods can facilitate the acceptance of medical devices by regulatory agencies [13], and is already prescribed as part of future development processes in some domains [30,31].

The software running in embedded devices is commonly composed of applications running on top of an Operating System (OS). Throughout the device life cycle, there are usually many more updates on the application than on the OS. Moreover, the application software is tailored for specific needs, while the OS is a foundation that diverse applications can use. Therefore, it is highly desirable to have a formally verified OS, which does not need to be re-verified when applications are modified. The complete formal verification of software involves the creation of models and their verification. Furthermore, all transition steps from models to machine code must be verified.

In this paper, we focus on the modeling stage by using the model-checking tool UPPAAL [23] to model typical features and functionality of modern real-time operating systems and to formally specify requirements to verify the model. Once the OS model is proven correct, it can be used by OS-based software models and reduce the verification effort, since OS requirements do not need to be re-verified.

**Our contributions** in this paper are (1) an approach that allows the modularization of formal models with defined interfaces, so that these can be assembled as models of the overall system; (2) based on this, guidelines to create a self-contained OS model that facilitates the creation of application models, which can be combined to verify various aspects of the overall software; (3) a concept for creating abstract task models to verify the OS model against the specified requirements.

**As a proof of concept and to evaluate** our approach in terms of performance and scalability, we formally model typical syscalls that represent the kernel interface towards the higher software levels. We then verify the modeled kernel features under all conceivable situations. For this, we create models that abstract the full software stack, and then verify timing, task synchronization, and resource management with feasible resource expense. The result is a formally verified OS model that can henceforth be used as a foundation for the modeling and verification of complex OS-based applications.

In this paper, **we do not address** the correctness of concrete OS implementations or the completeness of specified requirements, i.e., this paper does not aim to prove the correctness of the code-to-model translation, or that all require-

**Table 1.** Common task states on RTOSes.

State	Description
Running	Task is currently being executed.
Waiting	Task is waiting for an event, a resource, or a timeout.
Ready	Task could be executed, but higher priority tasks or the OS are running.
Suspended	Task is terminated or not yet started.

ments are specified. Still, the provided models and requirements are sufficient to demonstrate the proposed concept.

The remainder of this paper is organized as follows: in Section 2 we present relevant concepts for our proposed approach. In Section 3 we describe our approach to model the software layers modularly. In Section 4 we introduce abstract tasks and discuss the verification of OS requirements. In Section 5, we analyze and evaluate the proposed concept. In Section 6 we present related work. Finally, Section 7 summarizes this paper and shows potential future work.

## 2 Background

### 2.1 Real-Time Operating System (RTOS)

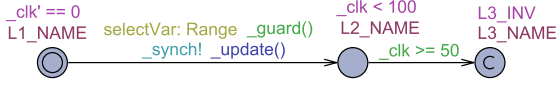
Complex OSES quickly lead to state explosion when model-checking. Therefore, we focus on a small common set of features of modern RTOSes that enables real-time behavior, namely preemptive multitasking, priority-driven scheduling, task synchronization, resource management, and time management. Priority inheritance protocols are not addressed in this paper, because they are not necessary to demonstrate our proposed concept. However, they can be integrated by modifying the related syscalls.

**Tasks** are the basic execution unit of RTOS-based software. They run in user mode and have fewer privileges than the kernel, which runs in kernel mode. Tasks have individual priorities and execute concurrently, and interact with the OS via *syscalls*. Tasks can be in one of the four states shown in Table 1. Specific implementations might not contain all states. For example, in this paper we model tasks as infinite loops, which never terminate. Thus, they have no suspended state. RTOSes commonly contain an idle task, which runs when no other task is in the ready state.

**The Kernel** is responsible for providing services to tasks and for interacting with the hardware. It initializes the system on startup and switches between tasks at runtime. Kernel execution can be triggered by tasks or interrupts through a fixed interface only.

**Syscalls and Interrupt Service Routines (ISRs)** are special functions that are exclusively provided by the kernel and define its interface. While user mode software can only interact with the OS through syscalls, ISRs can only be triggered by the hardware. The modeled syscalls and ISR are covered in Section 3.

**Time Management** is an important feature of RTOSes. The kernel (1) maintains an internal timeline to which all tasks can relate, and (2) allows tasks to specify timing requirements.



**Fig. 1.** A general UPPAAL timed automaton template.

**Events** can be used for inter-task communication and to react on interrupts. They provide a unified synchronization mechanism across hardware and software, in which tasks can signal each other, and interrupts can trigger tasks.

**Resources** coordinate the access of tasks to exclusively shared components, like hardware (e.g., I/O peripherals) or virtual entities (e.g., data structures). They can be requested from the OS and are assigned depending on availability and the priority of waiting tasks.

**The Scheduler** is responsible for coordinating the interleaving of tasks according to one or more predefined policies, such as fixed-priority, Rate-Monotonic Scheduling (RMS), and Earliest Deadline First (EDF).

## 2.2 UPPAAL

For modeling and verification, we choose the model-checking tool UPPAAL [23], in which systems are formalized as a network of timed automata with additional functions and data structures that are executed and changed on edges. Since we model preemptive tasks, we use UPPAAL 4.1, which supports stopwatch automata [10] and enables the elegant modeling of preemption. While a formal definition of timed automata is provided in [7], we still describe the features relevant for this work. Examples in this section refer to Fig. 1.

**Timed automata** are composed of (labeled) *locations* and *edges*. In UPPAAL, timed automata are specified with the concept of *templates*, which are similar to classes in object-oriented programming. For the verification, the templates are instantiated into *processes* (analogous to objects). All instantiated processes execute concurrently in a UPPAAL model. However, they can still be modeled in a fashion that executes them sequentially, which we adopted in our models.

**Locations.** *Standard locations* are represented by a circle ( $L2\_NAME$ ). The *initial location* ( $L1\_NAME$ ) is represented by a double circle. *Committed locations* ( $L3\_NAME$ ) have a letter “C” within the circle, and they are used to connect multi-step atomic operations. Different from standard locations, time does not pass while any automata are in a committed location. Locations can have names and invariants. Location names can be used in requirement specifications, and ease the readability of automata. A location invariant (e.g.,  $\_clk < 100$ ) is an expression that must hold while the automaton is in that corresponding location.

**Edges** connect locations in a directional manner. Edge transitions are instantaneous, i.e., they introduce zero time overhead. Edges can have a select statement ( $selectVar : Range$ ), a guard ( $\_guard()$ ), a synchronization ( $\_synch!$ ), and an update operation ( $\_update()$ ). A select statement non-deterministically chooses a value from a range of options and assigns it to a variable. A guard controls whether or not its edge is enabled. An update operation is a sequence of

```

1 typedef int [5 , 10] from5to10_t;
2
3 const from5to10_t VALID = 10;
4 from5to10_t invalid = 4; // verification failure
5
6 typedef struct {from5to10_t var1;} newStruct_t;

```

**Listing 1.1.** Bounded types and data structures in UPPAAL.

expressions to be executed. Finally, processes can synchronize and communicate via channels.

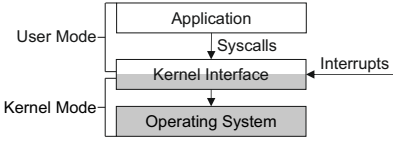
**Communication channels** (`_synch`) allow processes to send output (`_synch!`) or listen for input (`_synch?`). UPPAAL supports handshake and broadcast communication. When a synchronizing transition is triggered, both the sender and the listener(s) move to the next location simultaneously, assuming their guards allow for the transition to be taken. The update operation happens first at the sender side, allowing the sender to communicate numeric values via shared variables. In our approach, this is used to pass function/syscall parameters and return values between model modules.

**Time** is modeled with clock variables (`_clk`). The timing behavior is controlled with clock constraint expressions in invariants and guards. For example, the invariant `_clk < 100` and the guard `_clk >= 50` indicate that the transition from `L2_NAME` to `L3_NAME` happens when `_clk` is in the interval `[50, 100)`. In general, all clock variables progress continuously and synchronously. However, the stopwatch feature of UPPAAL 4.1 provides a way to stop one or more clocks in any location, namely by setting the clock derivative to zero (`_clk' == 0`). When the derivative is not written in the location invariant, its default value (1) is used and the clock progresses normally. For our system models, stopwatches are used to measure and verify the execution time of preemptive tasks. A task's clock progresses only if the task is in the running state, otherwise it is stopped.

**Functions, data structures and bounded data types** are defined in UPPAAL in a C-like language. Bounded types are very convenient for detecting unwanted values during the verification, which is immediately aborted in case a variable is assigned a value outside its type range. The syntax is exemplified in Listing 1.1.

**Formal verification.** UPPAAL performs symbolic model-checking to exhaustively verify the specified system requirements. The UPPAAL specification language allows expressing liveness, safety, and reachability properties.

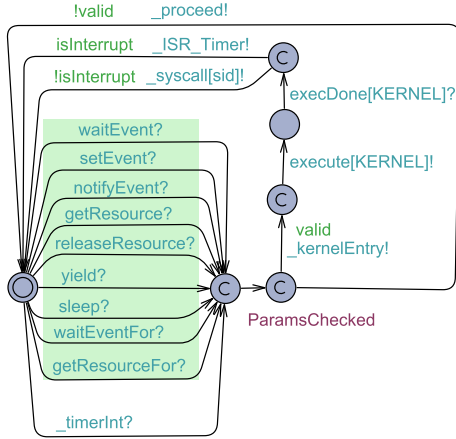
An important operator offered by UPPAAL is "`-->`" (leads to): `p --> q` means that whenever `p` holds, `q` shall also eventually hold. This notation is particularly useful to detect task starvation: if a task in the ready state does not lead to its running state, it is starved. A **deadlock** in the UPPAAL verification query language is used to detect system states that are not able to progress, i.e., states of the model in which no edges are enabled. Throughout this paper, such situations are referred to as UPPAAL *deadlock*. It must not be confused with *deadlock*, which refers only to task deadlocks due to cyclic waiting on resources.



**Fig. 2.** Model layers as abstraction of the software stack.



**Fig. 3.** Modeling of kernel execution time.



**Fig. 4.** Kernel interface template. Syscalls highlighted  $\color{blue}\boxed{\color{blue}}$ .

### 3 Model Design

In this section, we propose a general modular approach to model OSes and (abstractions of) application tasks. Our overall goal is to formally prove that a system meets all (non-)functional requirements, which we divide into OS-internal and overall software composition requirements. The characteristics of each category are described in Section 4.

We logically divide the UPPAAL model into three layers, as shown in Fig. 2. The application<sup>3</sup> contains tasks that run in user mode and can use OS services through syscalls. The kernel interface is responsible for switching between user and kernel mode, and to invoke the appropriate OS services or functionality upon syscalls or interrupts.

In this paper, we primarily focus on the operating system layer and how to model it with the goal to simplify the later modeling of the application layer. The result is a strict layering of the overall software model, where modules above the OS layer can be added, removed or updated without re-verifying the OS itself.

To demonstrate the applicability of our approach, we create an OS model  $\color{blue}\boxed{\color{blue}}$  (composed of sub-models) based on common features of modern RTOSes: preemptive multitasking, priority-driven scheduling, and syscalls for task synchronization, resource management, and time management. The modeling techniques are generic, and any concrete OS can be similarly modeled.

#### 3.1 Naming Convention

For readability, there is a naming convention for communication channels and variables throughout the entire model: Channels starting with an underscore

<sup>3</sup> For this paper, user libraries and middleware services are abstracted into the application layer and are not discussed separately.

(e.g., `_proceed!` in Fig. 4) represent internal kernel communication or are used for interrupt handling. Similarly, variables starting with an underscore represent internal kernel data structures. As for real code, the application layer must not directly access such OS-internal functions or variables. Channels and variables that can be accessed by the application layer as part of the OS interface start with a letter (e.g., `sleep?` in Fig. 4). Unfortunately, UPPAAL does not support such scope separation and the naming convention is used only as visual aid.

### 3.2 The Kernel Interface

The kernel interface must offer all possibilities to switch from user to kernel mode, modeled with communication channels. Triggering such channels from automata in the application layer represents a syscall in the real code.

Fig. 4 depicts our modeled kernel interface. A context switch (`_kernelEntry!`) occurs either upon syscalls, if the parameters are valid (`valid` [\[4\]](#)), or upon a timer interrupt (`_timerInt`). Supporting more interrupts (or syscalls) can be achieved by adding their corresponding automata, and respective edges into the kernel interface.

**Kernel Execution and Kernel Overhead.** Our modeling approach can precisely reflect the runtime overhead introduced in a preemptive system by the OS kernel itself. This allows a more accurate verification of the behavior of embedded systems compared to approaches that abstract away the OS layer. While different types of OS overhead can be modeled, we initially focus on timing.

Therefore, the kernel interface in Fig. 4 triggers a separate automaton for the kernel timing (`execute[KERNEL]!`), as shown in Fig. 3. The execution time interval [`bcet`, `wcet`] contains the time required to enter the kernel, process the invoked syscall or ISR, execute further kernel functions (e.g., the scheduler), and exit the kernel. This concentrated timing computation is possible because the kernel executes atomically (in contrast to the preemptive tasks).

Next, after taking kernel timing into consideration (`execDone[KERNEL]?`), we trigger the automata for the functional part of the actual syscall or ISR. The variable `sid` in `_syscall[sid]!` is updated along the syscall edges [\[4\]](#) and identifies the ID of the invoked syscall. The same approach can be used for modeling multiple interrupts.

### 3.3 The Operating System

The OS model must contain the internal data structures as well as the UPPAAL templates for the scheduler and for all syscalls. For this paper, we created the OS model based on the *SmartOS* [28] implementation.

**Data Structures and Tight Bounds.** We must declare all OS variables and arrays with data types of the tightest possible boundaries, according to the system parameters. Listing 1.2 shows a few examples from our OS model.

A beneficial consequence is a strict verification that does not tolerate any value out of range. In such cases, the verification immediately fails and aborts.

```

1 // 1 - System Parameters
2 const int NTASKS, NEVENTS, NRESOURCES, MMGR;
3 // 2 - Type Definitions
4 typedef struct {
5     int[0,NTASKS] qCtr; // the number of tasks in ready queue
6     ExtTaskId_t readyQ[NTASKS]; // the ready queue containing all tasks
7                               // in ready state sorted by priority
8 } SCB_t; // Scheduler Control Block
9 typedef int [0 , NTASKS - 1] TaskId_t;
10 // 3 - Declaration of Control Blocks
11 TCB_t _TCB[NTASKS]; // Task CBs
12 RCB_t _Res[NRESOURCES]; // Resource CBs
13 SCB_t _sched; // Scheduler CB

```

**Listing 1.2.** Tight bounds on type and array definitions [\[4\]](#)

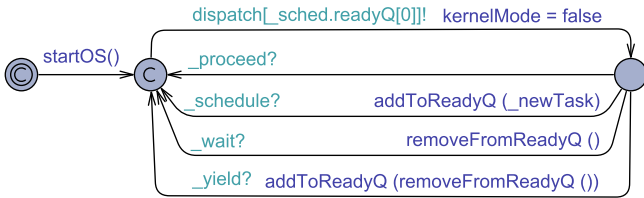
In other words, if the verification finishes, there is a guarantee that no boundary violation has occurred.

**The Scheduler** must be the only part of the OS model allowed to manipulate the ready queue (see Listing 1.2) and dispatch Ready tasks for execution.

Before the first task is dispatched, the system must be fully initialized. To ensure this, we must use a single initial committed location, from which an initializing edge transition occurs. Fig. 5 shows this behavior on the scheduler. The function `startOS()` initializes all the internal data structures of the OS. Next, because the following location is also committed, the scheduler immediately dispatches the highest priority Ready task, and switches to user mode (uppermost edge). The scheduler then must wait for instructions (`_proceed?`, `_schedule?`, etc.), which are issued by syscalls or ISRs, and must adapt the ready queue accordingly [\[4\]](#).

**Syscalls.** Each syscall must have a dedicated UPPAAL template, which models its semantics, i.e., the manipulation of related OS data structures, and interactions with the scheduler. Syscalls can be triggered (1) from the kernel interface (`_syscall[sid]!`) or (2) from other syscalls. Their general structure is an initial non-committed location, followed by a sequence of transitions through committed locations, making the syscall execution atomic, as shown in Fig. 6.

**Task slices.** While syscall automata model the behavior of the OS, task slices model different aspects of task execution, as shown in Fig. 7. They can directly communicate with task models (e.g., in Fig. 7(c), start/end a real-time block), or progress upon kernel operations (e.g., in Fig. 7(d), state change upon scheduler actions). The latter is completely transparent to task models. The use of task slices facilitates the modeling of tasks (Section 3.4) and the formal specification and verification of requirements (Section 4).



**Fig. 5.** The priority-driven scheduler [\[4\]](#).



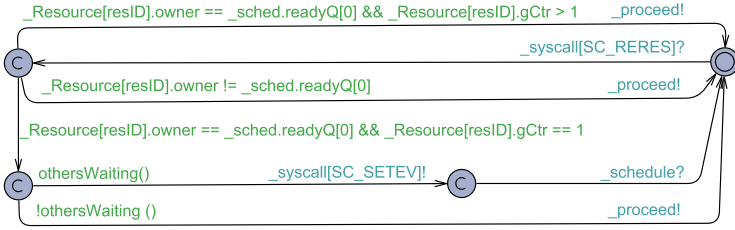


Fig. 6. The releaseResource syscall model [4].

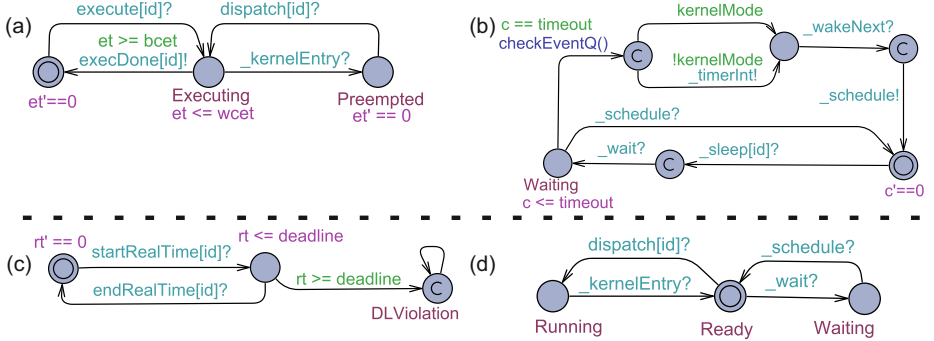


Fig. 7. Modeled task slices [4]: (a) Task Execution, (b) Task Timeout, (c) Task Real-Time, (d) Task States.

**Task Execution Time.** This task slice represents the user-space execution time of (code blocks within) a task. It abstracts away the code functionality, but allows the modeling of a  $[bcet, wcet]$  range. While the specification of the range itself is shown in Section 3.4, the helper template is shown in Fig. 7(a). Its structure is similar to the kernel execution time template in Fig. 3. However, we cannot assure that the execution of code in user mode is atomic, and must therefore consider preemption: If a `_kernelEntry!` occurs while a task is in the **Executing** location, it goes to **Preempted**, where the task execution is paused, i.e., the execution time clock `et` is paused (`et'==0`).

**Task Timeout.** This task slice is responsible for handling timeouts of syscalls (e.g., `sleep`), and thus it must trigger timer interrupts. Our version is depicted in Fig. 7(b)<sup>4</sup>. The clock `c` is used to keep track of elapsed time. The location **Waiting** can be left in two different ways: either the timeout expires (edge with `c==timeout`), or the task receives the requested resource/event (edge with `_schedule?`) before the timeout. If `c==timeout`, a timer interrupt is generated (`_timerInt!`) if the system is not in kernel mode. Otherwise, we directly proceed to the next location, where we wait for a signal from the scheduler (`_wakeNext?`) indicating that the task can be scheduled again. Finally, we instruct the scheduler to insert the current task into the ready queue with `_schedule!`.

<sup>4</sup> In our model, all syscalls with a timeout internally use `_sleep[id]` [4]. Other approaches might require multiple outgoing edges from the initial state.

**Task Real-Time.** This task slice is used to verify real-time behavior, as it can detect deadline violations. This task slice acts as an observer of the response times during verification, and has no influence on OS data structures or locations.

As shown in Fig. 7(c), there is a local clock `rt`, which is used to compute the response time of a code sequence. It remains paused unless `startRealTime[id]?` is triggered by the corresponding task. This happens in the task model (as shown in Section 3.4) and indicates that the task is about to start the execution of a code sequence with timing constraints. `rt` then progresses until the task triggers `endRealTime[id]?`. If this happens before the `deadline` is reached, the process returns to its initial state and is ready for another real-time block. Otherwise, the system goes to the `DLViolation` error state. The self-loop in the error state is used to avoid a UPPAAL deadlock<sup>5</sup>.

**Task States.** This task slice allows the detection of task starvation. A task starves if it never runs (again). A special case of starvation is task deadlock, which can be detected by additionally analyzing the OS internal data structures and identifying cyclic waiting on resources. Fig. 7(d) shows the modeled task states (as locations) and the actions that trigger state transitions.

The use of task slices is an extensible modeling concept: Extra task slices can be added to enable the verification of other (non-)functional requirements, e.g., energy/memory consumption.

### 3.4 Simple Application Modeling

The OS model, kernel interface, and task slices are designed with a common goal: Simplify the modeling of application tasks and make the overall system verification more efficient. With our concept, task models just need to use the provided interfaces (channels) and pass the desired parameters.

In summary, a task can be modeled with three simple patterns, as exemplified in Fig. 8:

- ❶ syscalls: invocation by triggering the corresponding channel, then waiting for `dispatch[id]?` (from the scheduler),
- ❷ execution of regular user code between `execute[id]!` and `execDone[id]?` (from Task Execution Time task slice),
- ❸ specification of real-time blocks between `startRealTime!` and `endRealTime!`.

As an example, Fig. 8 models the task source code from Listing 1.3 as a UPPAAL task. The variables `p1` and `p2` are used to pass data between different processes, e.g., for syscall parameters.

For ❶ and ❷, the use of the guard `amIRunning(id)` is crucial for the correct behavior of the task. It allows a task to proceed only if it is `Running`. The absence of this guard would allow any task to execute, regardless of priorities or task states.

For ❸, this guard is not necessary when starting or ending real-time blocks, though. If a task reaches the beginning of a real-time block, the response time

---

<sup>5</sup> In our approach, an UPPAAL deadlock indicates a modeling mistake.

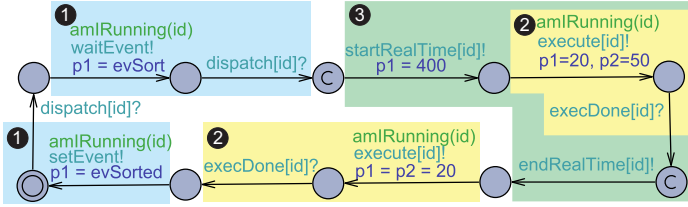


Fig. 8. UPPAAL model of the code from Listing 1.3.

```

1 OS_TASKENTRY (taskSort){
2   while (1){
3     ❶ waitEvent (evSort);
4     ❷ // START: Real-Time Task block. Deadline=400
5     ❸❸ quickSort (buffer, BUFSIZE); // Execution Block: BCET=20, WCET=50
6     ❹ // END: Real-Time Task block
7     ❺ for (...) printf ("\n%u", buffer[i]); // Execution Block: BCET=WCET=20
8     ❶ setEvent (evSorted);
9   }
10 }

```

Listing 1.3. Source code of a task.

computation must be immediately started, even if the task is preempted. Similarly, after the execution of a real-time block, the response time computation must be stopped immediately.

## 4 Requirements and Verification

### 4.1 Composition Requirements

These requirements refer to task properties that are influenced by other tasks running in the system, such as freedom from starvation and from deadline violations [4].

If a composition requirement is violated, the underlying cause is usually a badly composed or implemented task set, which makes it impossible for all tasks to coexist. However, it is also possible that an error in the OS leads to a violation of the composition requirements. In order to exclude this second possibility when verifying the complete system model, we must formally verify the OS model first.

### 4.2 OS Requirements

The OS requirements refer to OS properties that must always hold (invariants), regardless of the number of tasks in the system or of how these tasks interact with the OS (or with each other through the OS). As described in Section 3.3, the OS model is composed of data structures and multiple UPPAAL templates, which must be consistent at all time (general requirement). For example, if a task is in the **Waiting** location in the task timeout task slice, it must also be in the **Waiting** location in the task states task slice. In UPPAAL, we can verify this requirement with the query:

A[] forall (Tasks) TaskTimeout.Waiting imply TaskStates.Waiting [4]

This example shows an important point when extending our concept: Whenever new task slices are added to verify other (non-)functional requirements of the application, additional OS requirements must be specified to verify the consistency of the new task slice with pre-existing parts of the OS model.

### 4.3 Verifying the Requirements

For a given software (i.e., OS and application), we can prove correctness w.r.t. the OS and composition requirements by verifying all associated queries. However, we cannot yet claim that the OS model is correct in general (i.e., independent from the task composition), because we do not know if all possible OS operations were considered in all possible scenarios during the verification. Therefore, a complete re-verification of both layers is required in case the application changes.

To avoid the repeated and resource-expensive re-verification of the OS requirements for each task set, we must prove that the OS model is correct in general. We can then limit the re-verification to the application layer. To achieve this goal, we need to make sure that all possible OS operations are verified in all possible scenarios and execution orders. One possible strategy is to create different task sets to reach different scenarios, similar to test case generation. However, this strategy requires the prior identification of relevant scenarios, and the creation of the corresponding task sets. Additionally, it is hard to guarantee that all scenarios were indeed identified. Therefore, we introduce a new concept that inherently covers all scenarios: abstract tasks. They unite all possible behaviors of concrete tasks, i.e., they can trigger any action at any time. A task set with  $N$  abstract tasks thus represents the behavior of all possible task sets with  $N$  (concrete) tasks. Thus, by definition, all possible scenarios will be reached (UPPAAL exhaustive approach).

**Abstract Tasks.** Real tasks, as exemplified in Listing 1.3, are strictly sequential. Thus, a (concrete) task model is a predefined sequence of steps, as discussed in Section 3.4, and shown in Fig. 8. Their key characteristic is that only one outgoing edge is enabled in any location at any point in time.

The abstract task is depicted in Fig. 9. Unlike a concrete task, it has multiple outgoing edges enabled, which open all possible options to progress: ❶ syscalls with valid parameters and ❷ user code execution (`execute[id]!`). Thus, the behavior of any concrete task can also be achieved with the abstract task.

While different actions are performed by taking different edges, the parameters are non-deterministically chosen in the select statements for each syscall. The UPPAAL state space exploration mechanisms guarantee that all values of the select statements are considered for each edge.

Select statements are not necessary for the timing parameters `EX_TIME` and `SL_TIME`. Fixed values have less impact on the state space, and are enough to fire all edges from the task execution and task timeout (Fig. 7(a) and Fig. 7(b), respectively). We define the timing parameters `⌈` in a way that all edges are eventually fired and the state space remains small enough for a feasible verification.

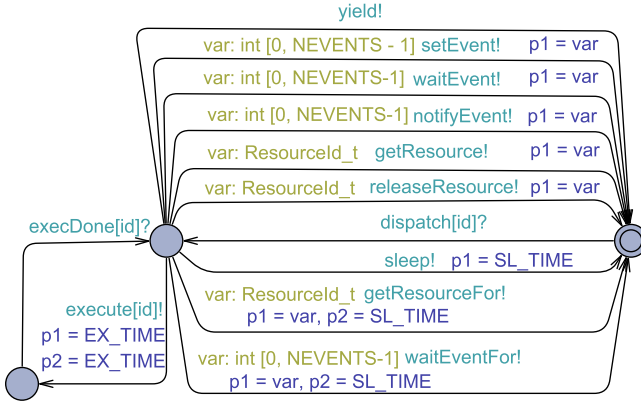


Fig. 9. The abstract task model [\[6\]](#).

**Non-Goals of Verification with Abstract Tasks.** With abstract tasks, it is meaningless to verify if **composition requirements** are satisfied at task level. Abstract tasks – by definition – lead to states where composition requirements are violated<sup>6</sup>. The goal of abstract tasks is to ensure that the OS itself works correctly even if the task composition is flawed, e.g., if it leads to starvation or livelocks. This is achieved by verifying the OS requirements in all conceivable scenarios (in the end of Section 4.4, we show how to verify that flawed composition scenarios are also reached). Additionally, we do not explore **invalid values** of variables./parameters. Out-of-bound values lead to verification failure, and when invalid syscall parameters are detected in the kernel interface, no functionality is triggered in the OS. Thus, checking for invalid values would increase the state space without adding new behaviors.

#### 4.4 OS Model Verification

A single set of abstract tasks provides a reliable way of verifying scenarios that could otherwise only be reached with numerous concrete task sets. To fully verify the OS model, we must compose the abstract task set so that it triggers all OS operations in all possible scenarios (covering all corner cases).

Within our model, we can control four system parameters that affect the OS verification: `NTASKS`, `NEVENTS`, `NRESOURCES`, and `MMGR`<sup>7</sup>, cf. Listing 1.2. We use a **short notation** to represent the system configuration. For example, 5-3-4-2 represents a configuration with `NTASKS` = 5 (idle task + 4 others), `NEVENTS` = 3, `NRESOURCES` = 4, and `MMGR` = 2. The goal is to find the minimal configuration that reaches all possible scenarios, and thus allows the complete verification of the OS model with minimal verification effort.

<sup>6</sup> Unless the OS offers guarantees by design, e.g., if it implements the Highest Locker Protocol (HLP), task deadlock scenarios must not be reachable.

<sup>7</sup> Maximum multiple `getResource`, i.e., the upper limit of the resource counter.

**Model Coverage.** In order to cover the whole model, the verification must traverse all edges, and entirely cover the C-like code of update operations.

*Edge Coverage.* If there is at least one edge in the model that is not traversed during verification, the model is surely not fully verified; unreachable edges could also indicate design flaws in the model. Therefore, the first step of the verification addresses the edge coverage. We add boolean markers in strategic edges, which are set to true when the corresponding edge is taken. We then verify if all markers are ever true:

```
E<> forall (i : int [0, NEDGES-1]) edge[i]==true
```

*Edge Scenarios.* A single edge can be traversed in multiple scenarios, due to composite guards (with the pattern (A or B or C ...)) or update operations (parameter passing or functions). For the composite guards, we must verify that each of its components is reachable with queries with the following pattern [\[4\]](#):

```
E<> Location and A
```

For the update operations, we ensure that an edge is traversed with all possible parameter values via select statements, which cover all valid parameter values. The functions demand a more careful analysis. It is necessary to identify all corner cases, and verify their reachability. For example, to verify the corner cases of a list insertion, we can use the following queries:

```
E<> InsertLocation and firstPosInsertion
E<> InsertLocation and lastPosInsertion
E<> InsertLocation and intermediatePosInsertion
```

After an iterative process of increasing the configuration and verifying the aforementioned properties, we found the smallest configuration that entirely covers our OS model: 4-1-1-2.

**OS and Composition Requirements.** The goal of the verification of the OS model is to guarantee that all OS requirements are met. In conjunction with the full model coverage verification, we prove that they are met regardless of the operations performed by individual tasks on top of the OS.

However, to ensure that the OS model is correct, we still must prove that the OS requirements are also met in states where composition requirements are violated. For that, we must identify all situations that violate composition requirements, and verify their reachability. For example, the reachability of a deadlock scenario can be verified with the query [\[4\]](#):

```
E<> Res1.owner == Task1 and Res2.owner == Task2 and
Task1.waits == Res2 and Task2.waits == Res1
```

The deadlock scenario reveals that 4-1-1-2 is not sufficient to reach all composition scenarios, since at least two resources are required to cause it. For the modeled OS features, all composition scenarios are reachable with 4-1-2-2.

---

**Algorithm 1** Finding the minimal configuration.

---

```

1: procedure FINDMINIMALCONFIG (FEATURES)
2:    $minConf \leftarrow 0-0-\dots-0$ 
3:   foreach (f: Features)
4:      $conf \leftarrow \text{getMinimalFeatureConfig}(f)$ 
5:      $minConf \leftarrow \text{getMaxParams}(minConf, conf)$ 
6:   return  $minConf$ 

```

---

## 5 Analysis and Evaluation

So far, we verified 4-1-2-2<sup>8</sup>, and confirmed that it satisfies all specified OS requirements and the necessary aspects discussed in Section 4: (1) traverse all model edges at least once; (2) invoke syscalls with all possible parameters; (3) reach all corner cases of edge update operations; (4) satisfy all of the components of composite guards; (5) reach valid and invalid composition scenarios; In this section, we analyze how the minimal configuration is obtained in the general case, and the scalability of the approach. We then reason why bigger configurations are not necessary for the verification.

### 5.1 Compositional Approach to Deriving the Minimal Configuration

The verification of the OS model is essentially the verification of its set of supported features. Thus, the composition of all minimal configurations needed to verify individual features is used to verify properties of the entire OS.

We assume that feature developers/experts provide the minimal configuration based on the corner cases and composition scenarios of their feature. We then build the minimal configuration by using the highest value of each parameter of each analyzed feature, as described in Algorithm 1. For example, the dominating features<sup>9</sup> in our OS model are resource management (3-0-2-2) and event passing (4-1-0-0), which lead to the resulting configuration 4-1-2-2.

### 5.2 Scalability: Resource Consumption for Verification

First, we show a concrete analysis of our approach, namely the number of explored states, CPU Time, and memory consumption during verification. Additionally, we show how each system parameter influences these values.

The verification was performed with UPPAAL 4.1.26 x64, running on a machine with Ubuntu 18.04.5 LTS, a 16 core Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, 64GB DDR4 memory @ 1600MHz, and 8GB swap.

**State Space.** In order to explore all states with a low processing overhead, we verify the query "A[] **true**". Fig. 10 and Table 2 show the number of explored states with different system configurations. The leftmost point ( $\Delta = 0$ ) in

<sup>8</sup> see Section 4.4 for configuration notation.

<sup>9</sup> No other feature has higher parameter values.

**Table 2.** Verification time (minutes) and memory consumption (MB).

Configuration	State space	A[] true		OS Requirements	
		Time	Memory	Time	Memory
C-(51-50-2-2)	574,266	1.5	640	76.0	738
4-1-2-2	5,369,534	0.6	470	30.7	470
4-1-2-4	14,963,367	2.5	1,787	124.3	1,788
5-1-2-2	85,077,164	13.2	6,655	644.2	6,656
4-3-2-2	116,606,955	14.7	10,189	689.0	10,189
4-1-4-2	570,284,574	75.8	47,800	3,774.6	47,800

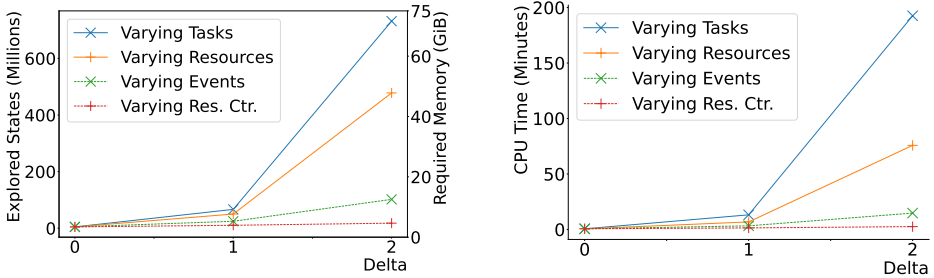

**Fig. 10.** Verification overhead for different configurations.

Fig. 10 represents our proposed minimal system configuration 4-1-2-2. We then vary one of the parameters, while all others are constant. For example, the "Varying Events" line on Delta = 1 shows the number of states for 4-2-2-2; and the "Varying Res. Ctr." line on Delta = 2 the number of states for 4-1-2-4.

The curves from Fig. 10 show that NTASKS has the biggest impact in the state space, and that MMGR has the lowest. While MMGR affects only the upper bound of the resource counters, NTASKS affects all kernel data structures, since each task can call any of the syscalls, which drive the modifications on the kernel data structures. In fact, the verification of 6-1-2-2 did not finish. It required more than 72GB of RAM, and the process was killed by Linux. Until short before, we could already count 950 million explored states.

It is important to highlight that the scalability is much better when simple concrete tasks are modeled. To demonstrate it, we modeled a concrete task set with sequential execution (without preemption) and used the configuration C-(51-50-2-2) [6], where C- indicates it is a configuration for a concrete task set. Table 2 shows that verifying "A[] true" explored only 574,266 states. Additionally, ongoing research on reducing the state-space, like for instance with partial-order reduction [22], will enable the verification of ever larger systems.

**Memory consumption and CPU time.** For the tested configurations, memory and CPU time follow a pattern similar to the number of explored states (Fig. 10). However, the number of states is not the only factor influencing resource consumption. The verification of C-(51-50-2-2) took longer and used more memory than the verification of 4-1-4-2, even though the state space is almost 10 times smaller (see Table 2). The size of individual states also plays



an important role, because they are stored/read into/from memory during the verification. In our OS model, `NTASKS`, `NEVENTS`, and `NRESOURCES` contribute to the state size, since bigger values increase the size/amount of data structures.

### 5.3 Sufficiency of 4-1-2-2 Configuration for our OS Model

We cannot run the verification of the OS model with arbitrarily big system configurations, due to the state space explosion problem. Therefore, we reason that, despite creating a larger state space, bigger configurations do not create any new scenarios in the OS layer.

As discussed in Section 3.3, the bounds of all data types are as tight as possible, and are defined according to the system parameters. Thus, when a parameter is increased, the bounds of the variables are adapted accordingly, avoiding out-of-bounds errors.

Since the bounds of data types and arrays are already covered by design, we just need to assure that no extra corner cases arise on queue operations.

**More abstract tasks.** With more tasks, the capacity of OS internal queues increases. Thus, there are more positions in which a new element can be inserted. However, these new possibilities do not add any new corner cases.

**More events or resources.** More events or resources lead to more queues in the system, but do not change the capacity of the queues. Thus, these parameters do not affect queue operations w.r.t. verification.

**Higher limit for counting resources.** When a task `T` (that already owns a resource `R`) requests `R` once again, `R`'s internal counter is incremented. Still, a higher limit does not create new corner cases w.r.t. verification.

**Composition Scenarios.** Bigger system configurations do not create new scenarios, but only new settings for the existing ones, e.g., starvation of different tasks, or deadlocks involving different sets of tasks and resources.

## 6 Related Work

Similar to our approach, with the goal to verify compositional requirements, Ironclad [18] covers the full software stack. It uses Dafny [25] and Boogie [6] to verify assembly code, but it addresses only security requirements. Borda *et al.* [8] propose a language to model self-adaptive cyber-physical systems modularly and a technique to support compositional verification. However, timing requirements are not addressed. Giese *et al.* [12] address compositional verification of real-time systems modeled in UML. Components are verified in isolation, and the correctness of the system is derived by ensuring that the composition is syntactically correct. However, this is only possible if the components do not share resources. UPPAAL has been used for schedulability analysis of compositional avionic software [17], and for conformance testing with requirements specified as pre- and post-condition functions [29].

Regarding modeling and verification of OSeS, on a more abstract level, Alkhamash *et al.* [5] propose guidelines for modeling FreeRTOS [1] using Event-B [3]. Cheng *et al.* formally specify the behavior of FreeRTOS tasks [11] and verify

it using the Z/Eves theorem prover[26], but, unlike our approach, they do not address timing, resource sharing, or interrupts.

On a less abstract level, closer to the real implementation, seL4 [20] proves the functional correctness of the C code of the kernel. Furthermore, it guarantees that the binary code correctly reflects the semantics of the C code. Hyperkernel [27] formally verifies the functional correctness of syscalls, exceptions and interrupts. The verification is performed at the LLVM intermediate representation level [32] using the Z3 SMT solver[9]. CertiKOS[16] is the first work that formally verifies a concurrent OS kernel. They use the Coq proof assistant[2], a C-like programming language, and a verified compiler [15]. These approaches focus exclusively on the functional correctness of the OS kernel.

We have not found a work that can verify timing, resource sharing, task synchronization, and interrupts in a compositional context. That is what our work enables, after proving the correctness of the OS model.

## 7 Conclusions and Future Work

In this paper, we presented a UPPAAL modeling approach for verifying compositional software, exemplified with an OS model containing a common set of features present in modern RTOSes. Since the proposed techniques and patterns are general, they can be used to model any concrete OS. We showed how to model the OS aiming to simplify the modeling of application tasks (Section 3). We also introduced separate *OS requirements* and *composition requirements*, and showed how they can be formally specified (Section 4) to decouple the verification of the OS and the application layer. We then proposed the concept of *abstract tasks* (Section 4.3) and reasoned that the OS model can be fully verified with a minimal set of such tasks, which interact through OS primitives (e.g., events and shared resources) and thus trigger all OS functions in all possible scenarios (Section 4.4). Finally, we evaluated the resource consumption of the verification process, reasoned about the sufficiency of the used minimal configuration, and analyzed the benefits of the proposed concept (Section 5).

With the OS model proven correct, there is no need to re-verify it when the upper layers are modified, which saves time and resources on the verification of concrete task sets. We consider this as particularly beneficial for developing and maintaining highly dependable systems, where, e.g., the task composition and functionality may change during updates. Another benefit of our approach is the potential use on test case generation for the application software.

This work opens a variety of directions for future work. We currently work on task slices to verify further (non-)functional requirements. Besides, we continuously improve the model design for a better trade-off between abstraction level and verification overhead, including the avoidance of potential state space explosions. Tools to convert between source code and UPPAAL templates shall reduce the modeling gap, i.e., the discrepancy between the formal model and the actual implementation. While our *models* allow the verification of applications on top of an OS, a limitation is that model correctness does not yet mean implementation correctness. For that, the full path from models to machine code must be verified.

## References

1. FreeRTOS. <https://freertos.org/>. [Online; accessed 20-January-2023].
2. The Coq proof assistant. <https://coq.inria.fr/>. [Online; accessed 20-January-2023].
3. Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.
4. Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. A comprehensive survey of industry practice in real-time systems. *Real-Time Systems*, 2021.
5. Eman H Alkhamash et al. Modeling guidelines of FreeRTOS in Event-B. In *Shaping the Future of ICT*. CRC Press, 2017.
6. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
7. Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on Uppaal. *Formal methods for the design of real-time systems*, 2004.
8. Aimee Borda, Liliana Pasquale, Vasileios Koutavas, and Bashar Nuseibeh. Compositional Verification of Self-Adaptive Cyber-Physical Systems. In *2018 IEEE/ACM 13th Int'l Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2018.
9. Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg, 2009.
10. Franck Cassez and Kim Larsen. The impressive power of stopwatches. In *International Conference on Concurrency Theory*. Springer, 2000.
11. Shu Cheng, Jim Woodcock, and Deepak D'Souza. Using formal reasoning on a model of tasks for FreeRTOS. *Formal Aspects of Computing*, 27(1), 2015.
12. Holger Giese et al. Towards the Compositional Verification of Real-Time UML Designs. In *9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, New York, NY, USA, 2003.
13. Mario Gleirscher, Simon Foster, and Jim Woodcock. New Opportunities for Integrated Formal Methods. *ACM Comput. Surv.*, 52(6), oct 2019.
14. Tomás Grimm, Djones Lettnin, and Michael Hübner. A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip. *Electronics*, 7(6), 2018.
15. Ronghui Gu et al. Deep Specifications and Certified Abstraction Layers. *ACM SIGPLAN Notices*, 50(1), jan 2015.
16. Ronghui Gu et al. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, November 2016. USENIX Association.
17. Pujie Han, Zhengjun Zhai, Brian Nielsen, and Ulrik Nyman. A Compositional Approach for Schedulability Analysis of Distributed Avionics Systems. In *1st Int'l Workshop on Methods and Tools for Rigorous System Design (MeTRiD@ETAPS)*, Greece, EPTCS, 2018.
18. Chris Hawblitzel et al. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014. USENIX Association.

19. Joseph Herkert, Jason Borenstein, and Keith Miller. The Boeing 737 MAX: Lessons for engineering ethics. *Science and engineering ethics*, 26(6), 2020.
20. Gerwin Klein et al. SeL4: Formal Verification of an OS Kernel. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, 2009.
21. John C. Knight. Safety Critical Systems: Challenges and Directions. In *24th Int'l Conference on Software Engineering*, ICSE '02, New York, NY, USA, 2002.
22. Kim G. Larsen, Marius Mikučionis, Marco Muñoz, and Jiří Srba. Urgent partial order reduction for extended timed automata. In Dang Van Hung and Oleg Sokol-sky, editors, *Automated Technology for Verification and Analysis*, pages 179–195, Cham, 2020. Springer International Publishing.
23. Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int'l journal on software tools for technology transfer*, 1997.
24. Thierry Lecomte et al. Applying a Formal Method in Industry: A 25-Year Trajectory. In *Formal Methods: Foundations and Applications*, Cham, 2017. Springer International Publishing.
25. K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
26. Irwin Meisels and Mark Saaltink. The Z/EVES reference manual (for version 1.5). *Reference manual, ORA Canada*, 1997.
27. Luke Nelson et al. Hyperkernel: Push-Button Verification of an OS Kernel. In *26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, 2017. Association for Computing Machinery.
28. Tobias Scheipel, Leandro Batista Ribeiro, Tim Sagaster, and Marcel Baunach. SmartOS: An OS Architecture for Sustainable Embedded Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2022*, Bonn, 2022. Gesellschaft für Informatik e.V.
29. Abhishek Singh, Meenakshi D'Souza, and Arshad Ebrahim. *Conformance Testing of ARINC 653 Compliance for a Safety Critical RTOS Using UPPAAL Model Checker*. New York, NY, USA, 2021.
30. UNECE. UN Regulation No. 156 – Uniform provisions concerning the approval of vehicles with regards to software update and software updates management system. [online] <https://unece.org/sites/default/files/2021-03/R156e.pdf>.
31. Virginie WIELS et al. Formal Verification of Critical Aerospace Software. *Aerospace Lab*, May 2012.
32. Jianzhou Zhao et al. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, New York, NY, USA, 2012. Association for Computing Machinery.

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

