*Smart*OS: An OS Architecture for Sustainable Embedded Systems

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

Leandro Batista Ribeiro
lbatistaribeiro@tugraz.at
Graz University of Technology
Graz, Austria

Tim Sagaster
tim.sagaster@student.tugraz.at
Graz University of Technology
Graz, Austria

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

ABSTRACT

The number of embedded devices is growing, and so are the concerns about dependability and sustainability. However, the life-span of modern devices is commonly very short, due to their lack of long-term maintainability in both hardware and software. This yields an increased amount of e-waste, as the individual devices are commonly very cheap and can therefore easily be replaced in case of (partial) obsolescence.

In this work, we show an operating system architecture which is designed to make embedded systems more sustainable and prepared for long-term use. To do so, we implement a general basic architecture alongside extended concepts and special features within the operating system. Our approach is based on hardware/software co-design and the opportunity to update software as well as hardware in a modular way at runtime. Therefore, logic reconfiguration of the host platform, dynamic software composition and integration, as well as formal methods for verification and portability are supported.

KEYWORDS

operating system, embedded systems, sustainability, partial updates, partial reconfiguration, formal methods

1 INTRODUCTION

This paper addresses the question how future Operating System (OS) architectures can support the development of *sustainable* software and hardware. Focus is on the aspects *long-term maintenance* and *dependability* of individual HW/SW components as well as the composed overall systems. In particular, new or currently underrepresented technological and methodological possibilities will be included in the architecture of the system software.

The impulse for the considerations is given by the overarching goal of keeping future embedded systems operational for decades with continuously guaranteed dependability [10, 12, 36, 58]. During development and operation, (limited) resources shall be used responsibly. Therefore, e-waste shall be avoided and existing systems shall be more easily reusable, i.e., efficiently adaptable to new functions and be able to operate in mixed networks of old and new systems in the long-run.

This overarching goal leads to more extensive or entirely new *demands* for the architecture of such systems and their components: Over the entire lifetime, (D1) partial software updates shall be facilitated and there shall be (D2) support for updating or modifying even the hardware. For each update, (D3) the integration of new or modified HW/SW shall be done automatically, and (D4) hard correctness guarantees for the new composition shall be provided. In order to be successful in the long run, (D5) the operating system itself shall be efficiently portable for new and changing target architectures as well as their derivatives in order to permanently guarantee a dependable basis for changing software compositions.

In this respect, technological and methodological progress in the areas of hardware design and formal methods opens up new *Possibilities*: Future embedded systems will (P1) increasingly support partial reconfiguration of logic at runtime [11, 67] and (P2) increasingly use formal methods for software development and maintenance [22, 25].

Today's operating systems are based on architectures which have virtually not changed for decades and do not or not sufficiently support the demands D1-D5. In particular,

FGBS '22, March 17–18, 2022, Hamburg, Germany

https://doi.org/10.18420/fgbs2022f-01

they do not yet take advantage of the available possibilities P1-P2 for ensuring continuous sustainability of systems.

In the following, we discuss some potential approaches for future operating system architectures to enable maintenance (in the sense of portability, support for reconfigurable hardware, and partial software updates) and guarantee dependability (in the sense of runtime updates and verification capability) of future embedded systems. Initial insights on first implementations and applicability are shown by the example of *Smart*OS.

The remainder of this paper is structured as follows: Section 2 summarizes current and previous work in related research areas and motivates the presented concepts. Subsequently, Section 3 shows the architecture, the main concepts as well as the utilized methodologies of our operating system *Smart*OS. Section 4 details the implemented features and resulting characteristics of *Smart*OS. In Section 5, an application scenario shows, how the features of *Smart*OS can help to overcome system shortcomings in practice, whereas the final Section 6 concludes the paper.

2 MOTIVATION AND RELATED WORK

Being able to update software in a deployed embedded system is not a new concept. There are multiple extensions available on top of widely used OSs like FreeRTOS [23], Device OS [55], QNX [13], VxWorks [71], or TinyOS [66] that provide some form of updates in the field. While most of those extensions will replace the firmware as a monolithic binary, almost all require a reboot of the system to finish the update [24]. There has been some work regarding the partial update of application code at runtime as well as of parts of the OS code itself [29, 43]. Some OSs have the built-in ability to update some modules and programs at runtime like Contiki [21]. Regarding the OS design, there is also some work using Domain-Specific Language (DSL) and formal methods [57] to model an OS that can be updated at runtime.

Since update-induced reboots or down times can lead to undefined conditions like timing violations or consequences on other systems [52], *Smart*OS supports runtime modularity in the upper layers.

While reconfigurable hardware (in, e.g., FPGAs) is becoming more common, it is still rare compared to fixed devices (ASICs). However, we see an increasing demand for hardware updates in future embedded devices and propose to implement the idea of "updating hardware like software" directly into the OS. Such an option would allow to fix hardware bugs [33, 38], improve algorithms or add new ones (e.g., for artificial intelligence or machine learning), or adapt to changing requirements and legislation [56]. Today, the hardware is either replaced or software workarounds are applied in such situations. Even if Field-programmable Gate Arrays (FPGAs) are used, we observe the same measures as for software: Logic updates are either monolithic or not offered at all. When it comes to OS support, only few kernels can exploit the flexibility of embedded FPGAs to enable dynamic Instruction Set Architectures (ISAs) [18] or extensible on-chip peripherals of soft core processors.

To support a seamless integration of hardware adaptation already in the OS, *Smart*OS can benefit from OS-awareness and partial reconfiguration in the host platform. This enables rebootless hardware updates at runtime [62].

To successfully integrate software updates, all involved processes should be done automatically. This includes dependency resolution, resource availability analysis (e.g., memory and OS data structures), and module management. While such features are standard in non-embedded devices, only few embedded OSs allow modular application updates at runtime due to the extra resource overhead. Still, some approaches allow updates to the bytecode of VM-based applications [37] or to dynamically link and load native code [20, 61].

In *Smart*OS, the so-called pluggability check is performed upon updates. New modules/versions can only be integrated if the target system has enough memory and free slots on the OS management data structures. Missing dependencies are automatically installed in the scope of an update protocol [8].

A reliable software integration also demands guarantees that the resulting software composition meets all system requirements. There are some approaches regarding schedulability [51, 64] as well as on guarantees about mutual exclusion properties and freedom from deadlocks [74] after updates. The demand for provable correctness as well as the occurrence of increasingly efficient algorithms to do so has led to a rise in popularity of formal methods in the area of embedded systems. An example for a formally verified kernel is sel4 [32], but it does not allow for runtime updates. Reconfiguring components of an embedded system at runtime in a safe manner is discussed in [57].

In *Smart*OS, we have proposed the interoperability check, a term that refers to operations performed to evaluate whether all system requirements will still be met in case an update was applied [8].

Portability of OSs is critical w.r.t. both coding effort and (non-) functional correctness for all supported target platforms. As detailed by [44], this goal can be achieved by applying formal methods for model-based design, verification, and code generation. However, even more recent OSs that declare portability as an inherent design feature, like Zephyr [73] or Atomthreads [4], only provide more or less detailed developer guides [5, 72] on how to port the OS to new processors or hardware platforms. In the automotive industry,

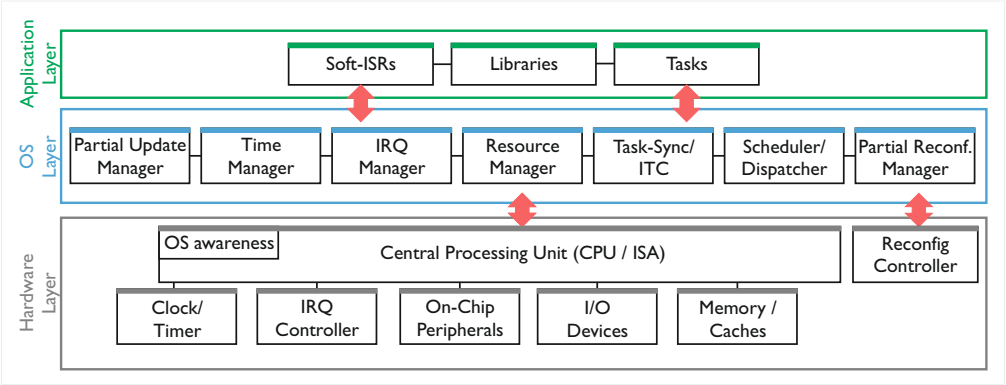

Figure 1: *Smart*OS layering with basic and extended concepts.

the problem of often error-prone and time-consuming re-implementation work [15] has been overcome by the AUTOSAR standard [6].

For *Smart*OS we seek to facilitate the porting process by formally modelling hardware-dependent kernel code and relevant parts of the target architectures. Code can then be generated from composed HW/SW models, allowing formal verification and avoiding error-prone re-implementation work.

For few existing embedded OS, the mentioned requirements or features do already exist independently from each other - often, however, only as extensions or workarounds. This motivates the creation of a sustainable OS that inherently incorporates features for all the introduced demands D1-D5 by design. To do so, we design *Smart*OS as an embedded OS, not only supporting rebootless updates for both software and hardware, but also compositional software aspects alongside portability and verification techniques. This way, we align to the paradigm proposed by European Union Agency for Network and Information Security (ENISA), which says that "the research in the area of patching and updating equipment without disruption of service and tools" [36] is of importance to future devices.

3 OS ARCHITECTURE AND BASIC CONCEPTS

*Smart*OS is a Real-Time Operating System (RTOS) for resource-constrained embedded devices, consisting of a microkernel that supports some basic features as explained next. The source code is supported by an increasingly complete set of formal models, representing all those features as well its target architectures for both formal verification and code generation (cf. Section 4.3). Furthermore, there exists an extensive build environment [45] that supports code separation for the selected target architecture. Currently, the main supported Microcontroller Unit (MCU) architectures are MSP430 [65] and RISC-V [60], but there are also ports for ARMv7E-M [3] (in MSP432 and STM32H7), SuperH [59], and Aurix [31]. Apart from research, *Smart*OS is also used in teaching.

*Smart*OS-based embedded systems feature a strict **layering** across the entire system stack that is depicted in Figure 1. Hardware components like the CPUs (which comes with optional OS-awareness, cf. Section 4.1.3) and its peripherals, as well as the (optional) Reconfiguration Controller (cf. Section 4.1.1) are featured in the Hardware Layer. In the OS Layer the *Smart*OS kernel with all its different Managers is situated. Application tasks, libraries and soft-ISRs belong to the Application Layer.

The *Smart*OS microkernel implements six central concepts, on which the extended concepts explained in Section 4 build. Table 1 summarizes the corresponding syscalls and their behaviour.

The **Internal Timeline** of *Smart*OS is directly driven by a hardware clock. On any supported architecture, it is 64 bits wide and has a resolution of 1 μs. The notion of time is provided to higher software layers through a temporal semantics of many kernel functions, that allow tasks to e.g., sleep or wait on events and resources with absolute or relative deadlines.

Tasks and their entry functions are the main building blocks of any *Smart*OS application. Tasks execute in task mode, are preemptive at any time, and can synchronize on each other (through events and resources) as well as on the hardware (through interrupts). Thus, they can be in running, ready, or waiting state (cf. Figure 2). To support proper interleaving by the scheduler, tasks have individual stacks of fixed size and variable priorities (both defined or pre-selected at compile time). *Smart*OS always provides an idle task that

Table 1: *Smart*OS syscalls. T is the invoking task.

Syscall	Behavior
`waitEventUntil(ev, deadline)`	T goes to waiting state until `ev` is set or the absolute `deadline` is reached. If `ev` is already set, T consumes the event and continues executing. If `deadline` is reached, T goes to ready state.
`waitEventFor(ev, timeout)`	Like `waitEventUntil`, but with a relative `timeout` instead of the absolute `deadline`.
`waitEvent(ev)`	Like `waitEventUntil`, but with an infinite `deadline`.
`setEvent(ev)`	T triggers `ev`: If `ev` was already set, nothing happens. If other tasks are waiting for `ev`, the one with highest priority immediately consumes it and goes to ready state. If no tasks are waiting, `ev` is set for later consumption.
`notifyEvent(ev)`	T triggers `ev`: If `ev` was already set, nothing happens. If other tasks are waiting for `ev`, they all immediately consume it and go to the ready state. If no tasks are waiting, `ev` is set for later consumption.
`getResourceUntil(res, deadline)`	T goes to waiting state until `res` is free or the absolute `deadline` is reached. If it is already free T takes the ownership of `res` and continues executing. If it is already owned by T, T increases the ownership of `res` by one and continues executing. If `deadline` is reached, T goes to ready state.
`getResourceFor(res, timeout)`	Like `getResourceUntil`, but with a relative `timeout` instead of the absolute `deadline`.
`getResource(res)`	Like `getResourceUntil`, but with an infinite `deadline`.
`releaseResource(res)`	T decreases its ownership of `res` by one. If it is already free or if T does not own it, nothing happens. If other tasks are waiting for it, the one with highest priority takes ownership of `res` and goes to ready state.
`getCurrentTime()`	Returns the current system time.
`sleepUntil(deadline)`	T goes to waiting state, and after absolute `deadline` is reached it goes to ready state.
`sleep(timeout)`	Like `sleepUntil`, but with a relative `timeout` instead of the absolute `deadline`.
`yield()`	T is removed from and immediately reinserted into the ready queue, handing over to another task with the same priority in ready state.

runs at lowest priority and is responsible for, e.g., power management. A hypothetical task can be seen in Listing 1.

System Calls are functions to request OS services from the higher software layers. When called by a task, they switch to kernel mode first and then execute the contained code in an atomic fashion. If supported by the CPU, *Smart*OS makes use of hardware acceleration or applies special protection mechanisms to enforce the isolation of the OS and the higher software layers [70].

Events allow Inter-Task-Communication (ITC) between two or more tasks and signalling between ISRs and tasks. While tasks and ISRs can trigger events, only tasks can wait for events (with or without deadline). This allows the simple

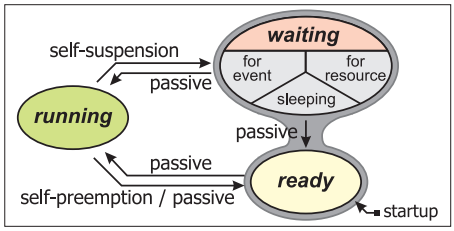

Figure 2: Task states and transitions in *Smart*OS.

Listing 1: A hypothetical *Smart*OS task.

```
1  OS_TASKENTRY (task1) {
2    [...]
3    while (1) {
4      waitEvent (ev1);
5
6      getResource (res1);
7      [...]
8      releaseResource (res1);
9
10     setEvent (ev2);
11   }
12 }
```

mapping of IRQs to responsible tasks – including the consistent reflection of IRQ priorities (HW) in relation to the tasks priorities (SW) [46, 48]. In any case, triggering an event can resume either the highest prioritized task or all tasks waiting for it.

Resources allow tasks to exclusively allocate shared physical or virtual objects, like on-chip peripherals, data structures, etc. Resources can be created at any software layer,

e.g., by drivers for their specific hardware component or by tasks just to synchronize with others. In any case, each resource can only be held by one task at a time, but a task may hold multiple resources at once or the same resource several times. Requesting a resource from the kernel can be done with or without deadline: If a resource is currently not available, the requesting task transits to waiting state until the resource becomes free or the deadline is reached. *Smart*OS implements various resource management protocols (PIP, PCP, HLP; cf. [16]) that take task priorities into account and feature different deadlock avoidance/resolution strategies. This includes a special extension that signals tasks that prevent higher priority tasks from running due to an allocated resource to release the resource – which reduces priority inversion and resolves deadlocks [9, 48]. While managing resources is different from protecting them, *Smart*OS can use related hardware security mechanisms as described in [40].

Interrupts are always handled in a unified way: The kernel provides a unique ISR for all IRQ sources, captures the timestamp on occurrence, and then calls a soft-ISR that can be provided by any software layer at runtime. This gives the opportunity to relate to the timestamp (e.g., for real-time systems) or to demultiplex IRQ sources (e.g., to introduce one soft-ISR per pin of a multi-pin GPIO port with just one shared IRQ vector). In any case, ISRs and soft-ISRs are executed atomically in kernel context and on the kernel stack. While this obviates the prophylactic over-provisioning of task stacks, ISRs can neither be nested nor are they allowed to wait for events or allocate resources. If such synchronization is needed, they should trigger a task for the actual IRQ processing.

4 EXTENDED CONCEPTS AND SPECIAL FEATURES

Apart from the basic concepts described in Section 3, our research in various directions has resulted in several extended concepts and special *Smart*OS features that tackle the demands D1-D5 from Section 1. These concepts can be divided into three different overarching topics: OS-specific hardware support and reconfiguration (Section 4.1), compositional software design and partial updates (Section 4.2), and formal methods for verification and portability (Section 4.3). *Smart*OS runs on all aforementioned platforms without restrictions. However, if it is used with our extended concepts and features in both hardware and software, there are several advantages.

4.1 MCU/OS Co-Design

The interaction between hardware and software requires mutual support on *both* sides. While "software follows hardware" is the prevalent credo in most domains, many embedded systems can benefit significantly from applications-specific hardware extensions. Thus, we do not only tailor our OS concepts to common hardware concepts, but we also investigate how OS kernels can benefit from specific MCU extensions, namely:

- *partial reconfiguration* of the host computing platform (i.e., processors, peripherals) at runtime (cf. Section 4.1.1),
- *hardware security* features (e.g., memory protection) within the host MCU (cf. Section 4.1.2), and
- *OS-aware* logic (cf. Section 4.1.3) of the host processor.

4.1.1 Partial Hardware Reconfiguration at Runtime. If a processor's logic changes during code execution, we talk about partial reconfiguration at runtime. With our goal to support dynamic composition in software *and* hardware, our concept of partial logic updates [62] allows to update and extend the ISA of processors as well as their on-chip peripherals on-the-fly without resynthesis of the entire logic and even without halting or resetting the system. As this significantly influences the software execution flow up to the application layer, the OS must support and manage such modifications, and support throughout all software layers must be introduced. In this section, we describe the underlying OS/MCU concepts at the example of the RISC-V-based *more*MCU.

To understand the concept, assume an application that calls an assumed instruction `cinsi` that might or might not be implemented by the processor. This is shown in Line 12 in Listing 2. Within the original RISC-V ISA [68, 69], this instruction is unknown and leads to an illegal instruction exception within the processor. In this case, the OS is responsible for emulating the unknown instruction's functionality in software. To do so, the currently running task is interrupted and kernel mode is entered. Then, the OS processes the binary encoding of the instruction causing the exception and checks its internal data structures for a corresponding emulation function that was registered by an application. If the emulation was registered before, this functionality is injected into the task's execution flow by the OS, and executed after the kernel returns to task mode. In case this look-up fails, the instruction is illegal and internal error handling steps in, and the task is most likely killed.

If, however, the instruction was natively supported by the processor, no further steps must be taken. Therefore, *Smart*OS is able to manipulate the ISA of the underlying computing platform actively. This is supported by our specifically tailored *more*MCU architecture, designed as a soft core for FPGA. As it features a runtime partial reconfiguration controller, parts of the CPU pipeline and the peripherals can

```
10    [...]
11    addi  t1, zero, 6
12    cinsi t0, t1,   2    ; unknown
          instr.
13    [...]
```

Listing 2: Unknown instruction execution code.

be hot-swapped between hardware realization and emulation on the fly. The OS can therefore decide, which instructions shall be added to the hardware, and which functionality shall be emulated in software. While the applications have to provide and register the emulation code or FPGA bitstreams to benefit from this feature, they can introduce new and application-specific hardware acceleration even during updates after hardware deployment.

4.1.2 Security Feature Support. A common design approach, especially in resource-constrained embedded devices is to have applications, OS components, and device drivers or libraries reside in a single non-isolated address space, which represents one vast attack surface. In order to reduce complexity, cost, and energy usage, embedded devices often lack memory protection mechanisms. Apart, programming flaws can easily lead to malfunctions or crash the whole system. As embedded devices are increasingly integrated in critical infrastructures, cyber-security attacks can have devastating consequences, including impacts on human lives or the environment [2, 30, 34].

Efficient memory protection in embedded devices can be achieved only if hardware and software components are co-designed to cooperate. We have designed and implemented lightweight hardware extensions for RISC-V-based MCUs which work in synergy with the *Smart*OS kernel extensions to provide several memory protection concepts. First, we enabled a memory protection mechanism which confines tasks to their own code and data memory regions, while still enabling full access to peripherals and protected access to shared memory for communication. Then, we enforced the kernel's resource management protocol in hardware by locking memory-mapped peripherals and protecting them from unauthorized task access [41]. Furthermore, we extended this concept to enable fine-grained protection of peripherals with multiple channels [40]. Finally, in order to protect the system from malfunctioning device drivers, we provided a device driver isolation mechanism, which limits the memory regions that drivers are allowed to access [39].

We show that these protection mechanisms have small hardware and software footprints. They do not impose significant run-time overhead and are suitable for maintaining the existing real-time properties of the system.

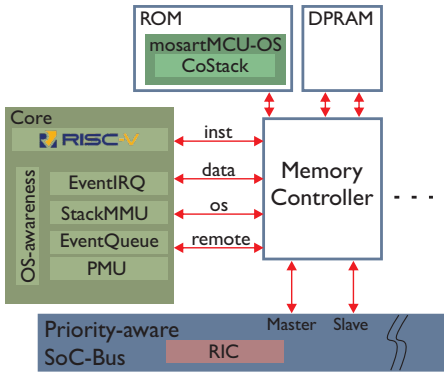

Figure 3: Overview of the *mosart*MCU.

4.1.3 OS-aware Processor Extensions. To support OS features in hardware, we introduced the mosartMCU [49, 50], being a flexible multi-core MCU architecture. Based on the RISC-V ISA [60], its implementation as soft core for FPGAs allows us to conduct hardware/software co-design for application-specific computing platforms. By introducing OS-awareness for improved dependability and composition aspects (cf. Figure 3), the *mosart*MCU and *Smart*OS support each other, but can also be used independently.

When used in combination, the MCU can concurrently access and modify internal OS data structures to temporally bound or entirely avoid cross-core priority inversions [48, 63]. Similarly, the co-design of OS/MCU memory management concepts and data structures improves shared stack memory usage to facilitate predictable software execution times [47]. In general, the OS-awareness enables system properties that could not be achieved through pure software solutions, and opens entirely new research opportunities for OS/MCU co-design.

4.2 Compositional Software and Automatic Integration

At build time, software composition of embedded systems is usually modular, i.e., different pieces of software (applications, libraries, OS, etc.) are combined to generate a binary image with the desired features and behavior. However, this image is mostly monolithic, and updates require the creation of another full image, be it deployed through complete replacement or by differential approaches. Our research efforts aim to provide concepts to support dynamic software (re-)composition at runtime [8]. This includes two steps: (S1) partially update the running software in a modular way while preserving code dependencies, and (S2) check if all functional and non-functional requirements would still be satisfied in case a given update was applied. Updates will

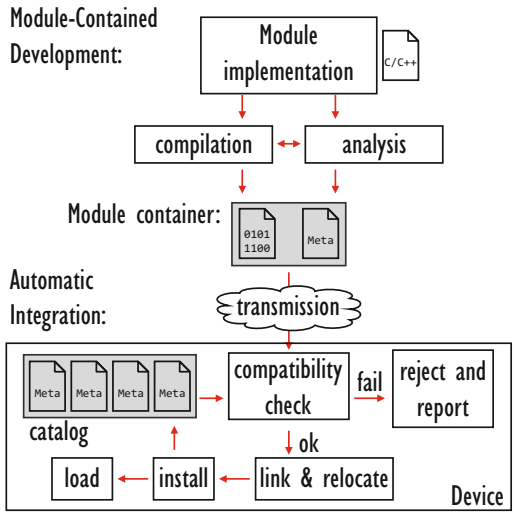

Figure 4: Module development and automatic integration.

only be applied in case all compatibility checks are successful, as shown in Figure 4.

Partial updates demand extra OS functionality. Already for step S1, it is necessary to keep track of all installed modules, handle dependencies, manage memories, and offer an accessible update service. For S2, complex algorithms – often based on formal methods – need to be implemented and demand additional computational resources.

From an implementation perspective of S1, *Smart*OS supports the update of application modules on conventional MCUs. However, the update of shared modules, such as drivers and libraries, is not supported, due to potentially inconsistent references. For each dependency update, dependent modules must adapt their references accordingly. To support this rather complex operation, we have worked on a hw/sw solution that supports loose coupling based on runtime relocation [42]. With our approach, all dependents remain unchanged when their dependencies are updated. It is also possible to freely move modules within the physical memory, allowing memory defragmentation. This feature is essential for long-term maintenance, since repeated updates will lead to memory fragmentation and the under-utilization of this commonly scarce resource.

When it comes to requirements checking for step S2, there are a variety of research challenges, especially regarding the mutual impacts of old and new software components w.r.t. (non-)functional requirements. For example, how to guarantee that already existing and newly installed real-time tasks will continue to meet their deadlines after a re-composition. Similar questions arise for other requirements, e.g., energy consumption, safety, security, etc. To enable the analysis of such aspects, it is necessary to generate or extract metadata from the developed modules. Our efforts in this regard include (i) a compact notation to describe the control-flow and interaction of tasks [7]; and (ii) comprehensive Uppaal models of *Smart*OS and application modules. Goal is to combine the metadata of individual modules in order to reliably conclude whether the resulting software composition meets the specified requirements. The approach that provides the strongest guarantees is formal methods, and that is the direction of our research efforts.

4.3 Formal Methods for Verification and Portability

Our approach uses formal languages to initially create independent models of application software, operating systems, and processor logic. Models of different layers of a concrete system can then be merged into an overall model to (1) verify different aspects of (non-)functional properties and (2) generate hardware-specific code for different target architectures.

For *Smart*OS, this approach currently allows us to give liveness and real-time guarantees for compositional application software, and to automatically port low-level kernel code to different target architectures.

Regarding (1), we use Uppaal [35] to model application tasks and the OS itself as a network of timed automata. Figure 5 shows how the execution block of a task is modeled (a task is composed of interactions with its execution block and with the OS). An execution block finishes when the task has executed for an interval within [BCET, WCET]. The execution time is not increased (`et' == 0`) before the execution block starts, or when the task is preempted.

One focus of the models is on the intended interaction (e.g., though explicit ITC) and implicit interference (e.g., through emerging conflicts) between tasks. Since the OS provides the central mechanisms for coordinating the tasks, we have modeled the internal functionality, timing parameters, and interfaces of the *Smart*OS syscalls (see Table 1) and interrupt concept. On top, application task models can interact with the OS model and also incorporate functional as well as non-functional aspects. This way, we are able to prove liveness (e.g., freedom from deadlocks), timing behavior, etc. throughout the entire system stack. A particular benefit of this approach is the possibility to (i) include even low-level OS details into such analysis (which is often neglected in many works on e.g., schedulability analysis), and to (ii) easily replace the OS and application models independently from each other (which allows us to verify the same application against various different mechanisms and implementation options in the kernel).

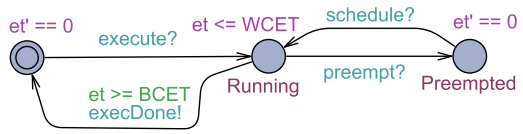

Figure 5: UPPAAL timed automaton of a task execution block.

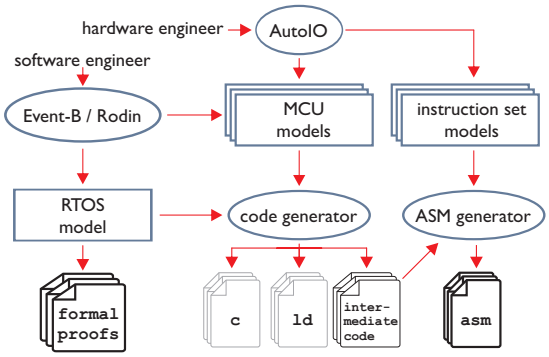

Figure 6: Model-based porting of hardware-specific code.

Regarding (2), we work on automatic generation of code for OS low-level functionalities, such as system initialization or context switches [26, 27]. The framework overview is depicted in Figure 6.

We use Event-B [1] to formally model and verify abstract versions of low-level OS operations. We also developed AutoIO [54], which is used to specify MCU architectures, including, e.g., on-chip components, buses, configuration registers, and the instruction set. We then feed the code generator with the verified OS model and the MCU model for the target architecture in order to generate architecture-independent code (in LLVM-IR). Finally, we feed the ASM generator with the generated intermediate code and the architecture instruction set model in order to create specific assembly code, proven correct. The automatic code generation guarantees that the assembly code is correctly translated from the models, and the formal verification guarantees that the models are correct. These features avoid implementation mistakes, which are very common when manually porting OSes.

5 APPLICATION SCENARIO

Having motivated requirements for future embedded systems and shown possibilities for designing suitable operating systems, we want to describe a concrete application scenario, that certainly demands sustainability, continuous dependability, and long-term maintenance.

The control systems of power plants are composed of multiple interconnected computing devices. For safety and security reasons, they are organized in levels that are mostly only accessible internally. However, external access to these devices is still required by the so-called emergency off-site facilities. Also for safety reasons, such facilities are located many miles away from the power plant [17]. Since the embedded systems of the power plant can thus be externally accessed, they could be hijacked and controlled remotely – demanding for continued and guaranteed dependability in every respect. In fact, the vast majority of computer systems rely on hardware acceleration for better performance on cryptography operations/security enforcement. However, cryptographic algorithms might become obsolete during the long lifespan of power plants. For example, in 2011 SHA-1 [19] became deprecated, thus making hardware accelerators in processors useless (cf. Intel processors [28]). Similar examples can be found in other areas which depend on the IoT as a critical infrastructure, that connects billions of devices [14].

To overcome this issue using immutable hardware, either (i) the new algorithm is implemented purely in software, and the obsolete silicon remains as useless logic, never again used; or (ii) the whole device or CPU is exchanged, which is not a sustainable choice. With reconfigurable hardware, the obsolete hardware logic could be replaced, resulting in faster operation and a more sustainable solution.

However, regardless of which changes are applied, the system's functional (e.g., new cryptographic algorithm) and non-functional properties (e.g., timing and memory consumption) will be modified, and it might be necessary to test, verify, and re-certify the updated system or submit so-called license amendment requests, which add cost and regulatory risk [53]. In this regard, the use of formal methods can be extremely beneficial, since it provides stronger guarantees than testing, and can facilitate regular updates [25].

6 CONCLUSION

In this paper, we present *Smart*OS, an OS architecture aiming to improve the sustainability of embedded systems, which is achieved through extended features on top of its basic functionality. These extended features include (i) a tightly coupled design of the OS and its underlying MCU, (ii) support for compositional software, and (iii) the use of formal methods to support software development and maintenance.

So far, approaches using the three mentioned features have not yet been fully adopted in research or industry. However, the demand for increased sustainability will create a need for these kind of features in the future. Therefore, the goal of this paper is to raise and intensify the awareness for the necessity of such concepts and simplify their use by providing an OS architecture that inherently supports them by design.

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering* (1st ed.). Cambridge University Press, New York, NY, USA.

[2] Riham Altawy and Amr Youssef. 2016. Security Tradeoffs in Cyber Physical Systems: A Case Study Survey on Implantable Medical Devices. *IEEE Access* (2016).

[3] ARM. 2021. *ARMv7-M Architecture Reference Manual.* Technical Report. ARM.

[4] Atomthreads. 2022. Atomthreads: Open Source RTOS | Free Lightweight Portable Scheduler. [Online] http://atomthreads.com/.

[5] Atomthreads. 2022. Porting Guide | Atomthreads: Open Source RTOS. http://atomthreads.com/index.php?q=node/3 [Online] http://atomthreads.com/index.php?q=node/3.

[6] AUTOSAR. 2017. Classic Platform Release 4.3.1.

[7] Leandro Batista Ribeiro and Marcel Carsten Baunach. 2019. COFIE: a regex-like interaction and control flow description. In *2019 IEEE Industrial Cyber-Physical Systems.*

[8] Leandro Batista Ribeiro, Fabian Schlager, and Marcel Baunach. 2020. Towards Automatic SW Integration in Dependable Embedded Systems.. In *Int'l Conf. on Embedded Wireless Systems and Networks (EWSN'20).* 85–96.

[9] Marcel Baunach. 2011. Dynamic hinting: Collaborative real-time resource management for reactive embedded systems. *Journal of Systems Architecture* 57, 9 (Oct 2011), 799–814. https://doi.org/10.1016/j.sysarc.2011.07.001

[10] Marcel Baunach, Renata Martins Gomes, Maja Malenko, Fabian Mauroner, Leandro Batista Ribeiro, and Tobias Scheipel. 2018. Smart mobility of the future – a challenge for embedded automotive systems. *e & i Elektrotechnik und Informationstechnik* (27 Jun 2018), 304–308. https://doi.org/10.1007/s00502-018-0623-6

[11] Christian Beckhoff, Dirk Koch, and Jim Torresen. 2012. Go Ahead: A Partial Reconfiguration Framework. In *20th Int'l Symposium on Field-Programmable Custom Computing Machines.* 37–44. https://doi.org/10.1109/FCCM.2012.17

[12] Ron Bell. 2006. Introduction to IEC 61508. In *Acm Int'l Conf. proceeding series*, Vol. 162. 3–12.

[13] BlackBerry. 2022. QNX. [Online] https://blackberry.qnx.com/en/.

[14] Carlo Boano, Kai Römer, Roderick Bloem, et al. 2016. Dependability for the Internet of Things: From dependable networking in harsh environments to a holistic view on dependability. *e&i* 133, 7 (2016), 6 pages.

[15] Manfred Broy. 2006. Challenges in Automotive Software Engineering. In *Proc. of the 28th Int'l Conf. on Software Engineering* (Shanghai, China) *(ICSE '06).* ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1134285.1134292

[16] Albert M. K. Cheng and James Ras. 2007. The Implementation of the Priority Ceiling Protocol in Ada-2005. *Ada Lett.* XXVII, 1 (apr 2007), 24–39.

[17] Chi-Shiang Cho, Wei-Ho Chung, and Sy-Yen Kuo. 2016. Cyberphysical Security and Dependability Analysis of Digital Control Systems in Nuclear Power Plants. *IEEE Trans. on Systems, Man, and Cybernetics: Systems* 46, 3 (2016), 356–369.

[18] Marvin Damschen, Martin Rapp, Lars Bauer, and Jörg Henkel. 2020. *i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems.* 1–36.

[19] Quynh H. Dang. 2015. *Secure Hash Standard.* Technical Report.

[20] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proc. of the 4th Int'l Conf. on Embedded networked sensor systems.* ACM, 15–28.

[21] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE Int'l Conf. on.* IEEE, 455–462.

[22] Marie Farrell, Matt Luckcuck, and Michael Fisher. 2018. Robotics and Integrated Formal Methods: Necessity Meets Opportunity. In *Integrated Formal Methods.*

[23] FreeRTOS. 2018. The FreeRTOS Kernel. [Online] https://freertos.org.

[24] FreeRTOS. 2022. AWS IoT Over the Air (OTA) Library. https://freertos.org/ota/.

[25] Mario Gleirscher, Simon Foster, and Jim Woodcock. 2019. New Opportunities for Integrated Formal Methods. *ACM Comput. Surv.* 52, 6, Article 117 (oct 2019).

[26] Renata Martins Gomes, Bernhard Aichernig, and Marcel Baunach. 2020. A Formal Modeling Approach for Portable Low-Level OS Functionality. In *Software Engineering and Formal Methods*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham, 155–174.

[27] R. M. Gomes and M. Baunach. 2019. Code Generation from Formal Models for Automatic RTOS Portability. In *2019 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO).* 271–272. https://doi.org/10.1109/CGO.2019.8661170

[28] Sean Gully, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. 2013. *New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processors.* Technical Report. Intel Corporation.

[29] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. 2013. Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems. In *5th Workshop on Hot Topics in Software Upgrades.*

[30] Abdulmalik Humayed, Jingqiang Lin, Fengjun Li, and Bo Luo. 2017. Cyber-Physical Systems Security - A Survey. *IEEE Internet Things J.* 4, 6 (2017).

[31] Infineon Techn. AG. 2018. *AURIX™ 32-bit microcontrollers for automotive and industrial applications – Highly integrated and performance optimized.*

[32] Gerwin Klein et al. 2009. seL4: Formal Verification of an OS Kernel. In *Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles (SOSP '09).* 207–220.

[33] Paul Kocher, Jann Horn, Anders Fogh, et al. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19).*

[34] Charalambos Konstantinou, Michail Maniatakos, Fareena Saqib, Shiyan Hu, Jim Plusquellic, and Yier Jin. 2015. Cyber-physical systems: A security perspective. In *20th IEEE European Test Symposium.* 1–8.

[35] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *Int'l Journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

[36] Rafał Leszczyna et al. 2011. Protecting Industrial Control Systems. Recommendations for Europe and Member States. *The European Union Agency for Network and Information Security (ENISA)* (2011). https://www.enisa.europa.eu/publications/protecting-industrial-control-systems.-recommendations-for-europe-and-member-states

[37] Philip Levis and David Culler. 2002. MatÉ: A Tiny Virtual Machine for Sensor Networks. *SIGARCH Comput. Archit. News* 30, 5 (Oct. 2002), 85–95.

[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, et al. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium.*

[39] Maja Malenko and Marcel Baunach. 2019. Device Driver and System Call Isolation in Embedded Devices. In *22nd Euromicro Conf. on Digital System Design.* IEEE, 283–290. https://doi.org/10.1109/DSD.2019.00049

[40] Maja Malenko and Marcel Baunach. 2019. Hardware/Software Co-designed Peripheral Protection in Embedded Devices. In *IEEE Int'l Conf. on Industrial Cyber Physical Systems*. 790–795.

[41] Maja Malenko and Marcel Baunach. 2019. Hardware/Software Co-designed Security Extensions for Embedded Devices. In *Proc. of the 32nd Int'l Conf. on Architecture of Computing Systems*. 3–14.

[42] Maja Malenko, Leandro Batista Ribeiro, and Marcel Baunach. 2021. Improving Security and Maintainability in Modular Embedded Systems with Hardware Support: Work-in-Progress. In *Proc. of the 2021 Int'l Conf. on Hardware/Software Codesign and System Synthesis*.

[43] Pedro Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. 2006. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. *Europ. Workshop on Wireless Sensor Networks*.

[44] Renata Martins Gomes and Marcel Baunach. 2021. A Study on the Portability of IoT Operating Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2021*. Gesellschaft für Informatik e.V., Bonn. https://doi.org/10.18420/fgbs2021f-01

[45] Renata Martins Gomes, Marcel Baunach, Maja Malenko, Leandro Batista Ribeiro, and Fabian Mauroner. 2017. A Co-Designed RTOS and MCU Concept for Dynamically Composed Embedded Systems. In *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 41–46.

[46] Fabian Mauroner and Marcel Baunach. 2017. EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments. In *Proc. of the 20th Euromicro Conf. on Digital System Design (DSD)*. 102–110.

[47] Fabian Mauroner and Marcel Baunach. 2018. CoStack: Collaborative Stack Sharing for Embedded Real-Time Systems. In *Proc. of the 13th Int. Conference on Systems (ICONS)*.

[48] Fabian Mauroner and Marcel Baunach. 2018. EventQueue: An Event based and Priority aware Interprocess communication for Embedded Systems. In *Proc. of the 13th Int. Symposium on Industrial Embedded Systems*. IEEE.

[49] Fabian Mauroner and Marcel Baunach. 2018. mosartMCU: Multi-Core Operating-System-Aware Real-Time Microcontroller. In *Proc. of the 7th Mediterranean Conference on Embedded Computing (MECO)*.

[50] Fabian Mauroner and Marcel Baunach. 2019. OSARM: A Resource Management Approach for an Operating-System-Aware Microcontroller. *Under review – Journal of Microprocessors and Microsystems (MICPRO)* (2019).

[51] Jami Montgomery. 2004. A model for updating real-time applications. *Real-Time Systems* 27, 2 (2004), 169–189.

[52] Imanol Mugarza, Jorge Parra, and Eduardo Jacob. 2020. Cetratus: A framework for zero downtime secure software updates in safety-critical systems. *Software: Practice and Experience* 50, 8 (2020), 1399–1424.

[53] Michael Muhlheim, Peter A Sandborn, E Quinn, Paul Hunton, Richard Edward Hale, and R England. 2019. *Development of an Obsolescence Cost Model for Nuclear Power Plants*. Technical Report. Oak Ridge National Lab.(ORNL), USA.

[54] Paul Nagele. 2017. Design and Implementation of a I/O specication Tool for MCU Architectures. Bachelor's thesis, EAS group.

[55] Particle. 2022. Device OS. [Online] https://docs.particle.io/tutorials/device-os/device-os/.

[56] Birgit Penzenstadler and Joerg Leuser. 2008. Complying with Law for RE in the Automotive Domain. https://doi.org/10.1109/RELAW.2008.3

[57] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. 2007. Experience with Safe Dynamic Reconfigurations in Component-Based Embedded Systems. 242–257. https://doi.org/10.1007/978-3-540-73551-9_17

[58] Marvin Rausand and Jørn Vatn. 2008. Reliability centred maintenance. In *Complex system maintenance handbook*. Springer, 79–108.

[59] Renesas. 2006. *Renesas 32-Bit RISC Microcomputer SuperH RISC engine Family*. Technical Report. Renesas.

[60] RISC-V Foundation. [n.d.]. RISC-V. https://riscv.org/ [Online] https://riscv.org/.

[61] Peter Ruckebusch et al. 2016. GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules. *Ad Hoc Networks* 36 (2016), 127 – 151.

[62] Tobias Scheipel, Peter Brungs, and Marcel Baunach. 2021. A Hardware/Software Concept for Partial Logic Updates of Embedded Soft Processors at Runtime. In *Proc. of the 24th Euromicro Conf. on Digital System Design*. 199–207.

[63] Tobias Scheipel, Fabian Mauroner, and Marcel Baunach. 2017. System-Aware Performance Monitoring Unit for RISC-V Architectures. In *Proc. of the 20th Euromicro Conf. on Digital System Design*. 86–93.

[64] Habib Seifzadeh, Ali Asghar Pourhaji Kazem, Mehdi Kargahi, and Ali Movaghar. 2009. A method for dynamic software updating in real-time systems. In *8th IEEE/ACIS Int'l Conf. on Computer and Information Science, 2009*. IEEE, 34–38.

[65] Texas Instruments. [n.d.]. MSP430 ultra-low-power sensing and measurement MCUs. http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview/overview.html

[66] TinyOS. 2022. TinyOS. http://www.tinyos.net/ [Online] http://www.tinyos.net/.

[67] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. 2004. The MOLEN polymorphic processor. *IEEE Trans. Comput.* 53, 11 (2004), 1363–1375. https://doi.org/10.1109/TC.2004.104

[68] Andrew Waterman and Krste Asanovic. 2020. *The RISC-V Instruction Set Manual, Volume I: Unprivileged-Level ISA, Version 20191214-draft*. Technical Report. SiFive Inc., UC Berkeley.

[69] Andrew Waterman and Krste Asanovic. 2020. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.12-draft*. Technical Report. SiFive Inc., UC Berkeley.

[70] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. https://doi.org/10.14722/ndss.2019.23068

[71] WindRiver. 2022. VxWorks. [Online] https://www.windriver.com/products/vxworks.

[72] Zephyr Project. 2022. Porting - Zephyr Project. [Online] https://docs.zephyrproject.org/latest/guides/porting/index.html.

[73] Zephyr Project. 2022. Zephyr Project. [Online] https://www.zephyrproject.org/.

[74] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. 2013. Formalization and Verification of Behavioral Correctness of Dynamic Software Updates. In *Proc. of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop*. 12–23.