Automatic Verification of Application-Tailored OSEK Kernels

Hans-Peter Deifel, Merlin Göttlinger,
Stefan Milius and Lutz Schröder
Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg
Email: {hans-peter.deifel, merlin.goettlinger, stefan.milius, lutz.schroeder}@fau.de

Christian Dietrich and Daniel Lohmann
Leibniz Universität Hannover
Email: {dietrich, lohmann}@sra.uni-hannover.de

***Abstract*—The OSEK industrial standard governs the design of embedded real-time operating systems in the automotive domain. We report on efforts to develop verification methods for OSEK-conformant compilers, specifically of a code generator that weaves system calls and application code using a static configuration file, producing a stand-alone application that incorporates the relevant parts of the kernel. Our methodology involves two verification steps: On the one hand, we extract an OS–application interaction graph during the compilation phase and verify that it conforms to the standard, in particular regarding prioritized scheduling and interrupt handling. To this end, we generate from the configuration file a temporal specification of standard-conformant behaviour and model check the arising formulas on a labelled transition system extracted from the interaction graph. On the other hand, we verify that the actual generated code conforms to the interaction graph; this is done by graph isomorphism checking of the interaction graph against a dynamically-explored state-transition graph of the generated system.**

1. Introduction

Embedded real-time control systems are special-purpose systems dedicated to specific, predefined tasks [1], [2]. Already now, a typical (in particular, non-autonomous) car contains up to a hundred such systems. Hence, both the hardware and the system software of each embedded control system need to be tailored to its specific needs in order to keep per-unit hardware costs as low as possible [3]. The OSEK-OS standard [4] fulfils these demands for tailorability and has been (together with its superset AUTOSAR-OS [5]) the dominant industry standard for event-triggered automotive *real-time operating systems* (RTOSs) for the last decades. What sets OSEK apart from the common POSIX-like operating systems is that it is completely statically configured. For a specific automotive application, all system objects (tasks, interrupt-service routines, resources, etc.) and their configurations have to be declared at compile-time in a domain-specific language, the *OSEK Implementation Language* (OIL) [6]. From this specification, an application-specific, highly optimized RTOS instance is derived by means of a generator.

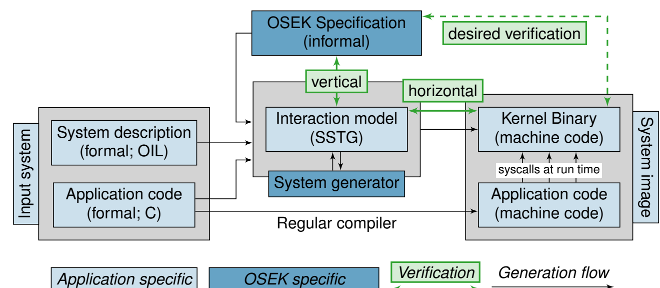


Figure 1: System generation and verification. We use the *static state-transition graph (SSTG)*, a byproduct of the OSEK system generator, as an intermediate representation for our desired kernel verification.

However, with the advent of autonomous driving features, the industry is facing new challenges with respect to functional safety; the highest safety level ISO 26262 ASIL D demands the employment of a certified RTOS, such as *RTA-OS* (ETAS), *MICROSAR OS* (Vector), or *tresos Safety OS* (EB) (vendors named in parentheses). These certified operating systems offer significantly less tailorability and thus induce higher per-unit costs. The certification of the development process of the RTOS kernel is already extremely expensive, so vendors shy away from the even higher costs of certifying a kernel generator.

Taking a step back, we argue that application developers do not need a kernel that behaves correctly in all imaginable situations. However, they are very interested in a kernel that *always* behaves correctly for their specific application and all of their kernel-usage patterns. Therefore, we replace the isolated certification of the generator with a per-instance verification of the resulting kernel binary and thus formulate our verification goal: For a given application, the generated kernel binary must expose the specified behavior when executed together with our application. This bypass of the generator in the verification process allows all kinds of highly specialized system optimizations.

We achieve the desired verification (see Figure 1) by introducing an kernel–application interaction model: Our toolchain considers not only the OIL-specified system object instances but also how these system objects actually interact with each other via the syscall interface according to the

```
TASK(Low) {
  if (test()) {
    ActivateTask(High);
  }
  TerminateTask();
}

TASK(Med) {
  TerminateTask();
}

TASK(High) {
  TerminateTask();
}

void ISR() {
  ActivateTask(Med);
}
```

Figure 2: Example System – Source Code

OSEK semantics [7]. We enumerate the application-specific kernel's state space, the *static state-transition graph* (SSTG), which is calculated and used by the system generator. The SSTG acts as an intermediate representation of kernel behavior and is the central data structure for our strategy. First, we statically verify (*vertically*, according to Figure 1) that the SSTG actually conforms to the OSEK standard. To this end, we formalize key aspects of the standard in CTL and model check the SSTG against this specification; this is feasible because the SSTG is of moderate size thanks to tailoring. Second, we dynamically verify the system against the SSTG (*horizontally*, according to Figure 1). To this end, we probe the system to explore a *dynamic state-transition graph* (DSTG) and check that it is isomorphic to the SSTG. We report on experiments with this methodology, both on systems from a standard test suite and on the control software of a quadrotor copter.

2. Background and Context

We give a brief overview of the single-core OSEK real-time operating system standard. Moreover, we describe the *static state-transition graph* (SSTG), which the dOSEK generator (dOSEK = dependable OSEK, our implementation of the OSEK standard) uses as an intermediate representation to model all possible interactions between application and kernel.

2.1. OSEK in a Nutshell. The tailored embedded systems that we are concerned with here are woven from application code and a tailored kernel instance. Kernel and application interact at runtime, typically subject to requirements on real-time performance. Depending on the current OS state, the kernel selects a control flow that is currently ready and dispatches it for execution. The application's control flows, as the kernel's counterpart, manipulate the OS state by invoking *system services* that influence the system behaviour (Table 1 gives a short overview). OSEK offers two main control flow abstractions: *interrupt-service routines* (ISRs) and *tasks* (i.e. threads). ISRs are activated by the hardware and fall into two classes: *category-1* ISRs, which are not allowed to call system services; and *category-2* ISRs, which are synchronized with the kernel. Tasks have a statically assigned priority, are allowed to use all system services, and are invoked according to a strict fixed-priority preemptive scheduling policy. On each new activation, tasks start from the very beginning and run until (self-)termination. Each task is configured to be either nonpreemptive (enforcing run-to-completion semantics) or fully preemptive. Preemption points can be either *synchronous*, for example caused by an explicit activation of a higher priority task (`ActivateTask`), or *asynchronous*, if a higher priority task is activated inside an ISR. Periodically or aperiodically recurring task activations can be triggered by means of statically configured *alarms*, which are driven by a hardware timer.

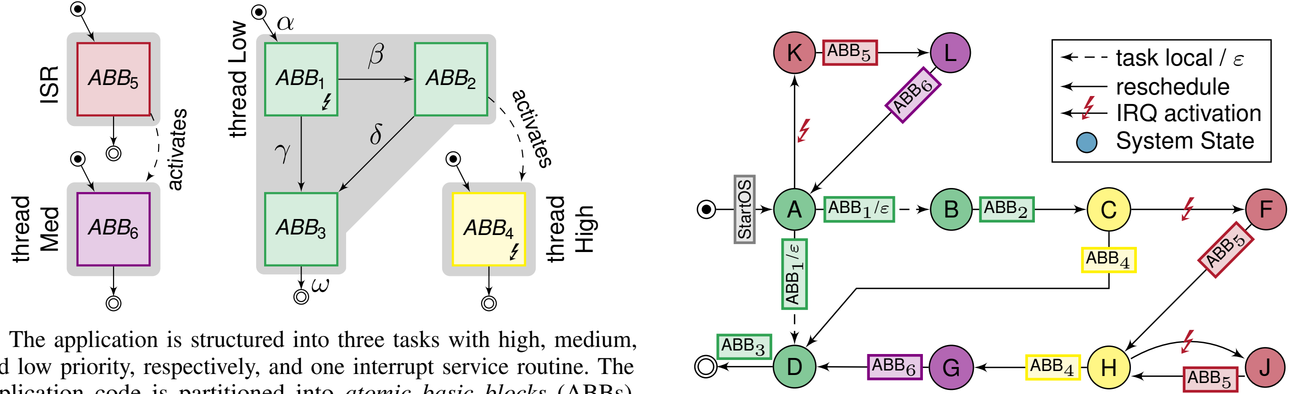
Inter-task synchronization is realized by *resource* objects. Based on a *stack-based priority-ceiling protocol*, OSEK resources ensure mutual exclusion while preventing deadlocks and priority inversion. Through the acquisition of a resource, a task raises its *dynamic* priority to the *ceiling* priority of the resource – the highest static priority of all tasks that can obtain the resource, according to the OIL file.

Figure 2 shows a small example system that consists of three tasks and one ISR. These coordinate their execution with the help of the OS, which is activated through system service invocations (syscalls). Figure 3a depicts the same system as read in by the dOSEK generator.

2.2. Generating a System. A dOSEK kernel instance is generated from two inputs (see also Figure 1): The application's OIL file specifies the employed RTOS objects (i.e., the tasks *Low*, *Med*, *High* and the ISR *ISR* in our example). The application's source code (Figure 2) specifies how these system objects interact according to the OSEK semantics.

Internally, we structure the application code into a set of *control-flow graphs* (CFGs) consisting of *atomic basic blocks* (ABBs) (Figure 3a). An ABB [7], [8] is a control-flow superstructure that subsumes one or more traditional *basic blocks* (BBs) forming a single-entry single-exit region; it has *exactly one* distinguished entry BB and one exit BB. The construction (see [7] for a detailed description) results in one ABB-graph for each task within the application code. By construction, every ABB either contains a single syscall or only computation code that does not interact with the OS (no syscalls). From the kernel's point of view, an ABB executes atomically, but can be interrupted by ISRs.

At build time, the dOSEK generator computes a *state transition graph* (STG) from the ABB graphs of the individual tasks and the system configuration (OIL). (We formally define STGs in Section 2.3.) This STG is the *static state-transition graph* (SSTG) already mentioned in the introduction. Starting with an initial global system state, derived from the OIL file, the generator enumerates all reachable system states explicitly (Figure 3b). Every state carries the currently running task, a block that is executed in this state (e.g. state A executes ABB_1), and other relevant scheduling data, like the list of activated tasks. A state transition is caused by either a computation block, a system call block, or an interrupt request. For the associated post-state, the currently running task and the currently executed block are calculated according to the OSEK scheduling semantics. The SSTG subsumes the interwoven application–kernel behaviour and includes all possible scheduling sequences an OSEK kernel exposes for the given application. Since *computation* blocks do not perform syscalls, their execution does not influence future scheduling decisions, and is therefore represented



(a) The application is structured into three tasks with high, medium, and low priority, respectively, and one interrupt service routine. The application code is partitioned into *atomic basic blocks* (ABBs). In ABB_2, the low-priority task activates the high-priority task via `ActivateTask`. The ISR activates the medium-priority task. (To simplify the example, the IRQ may happen only in ABB_1 and ABB_4, and the tasks issue `TerminateTask()` implicitly.)

(b) This *static state-transition graph* (SSTG) captures all possible state transitions from the start to the termination of the low-priority task according to the OSEK semantics. Every state carries and executes exactly one ABB.

Figure 3: Example system

TABLE 1: Excerpt of system services provided by the OSEK-OS API.

System Service	Arguments	Brief Description
`ActivateTask`	`TaskID`	Task `TaskID` is activated. If the current task is preemptible, immediate rescheduling takes place.
`TerminateTask`	–	The current task terminates itself and immediate rescheduling takes place.
`GetResource`	`ResID`	Acquires the resource identified by `ResID`.
`ReleaseResource`	`ResID`	Leaves the critical region associated with the resource `ResID`. The dynamic priority of the calling task is changed and a reschedule takes place for preemptible tasks.

by ε-transitions. For instance, ABB_1 is computational and, therefore, the transition between states A and D becomes an ε-transition.

Thus computed, the SSTG has node identities representing the current global system state, while the edges of the SSTG are either ε-transitions or labelled with the system calls triggering the respective state transitions. System call labels can be either system calls in the proper sense, interrupts (which are triggered by the hardware outside the system), interrupt returns (irets), or the idle system call. The idle system call is executed from an idle state when no other task is ready for execution. This ensures that all maximal paths in the SSTG are infinite. The latter three types of system call labels are artificially added by the generator and only model the implicit state transitions of an OSEK system; they are not explicitly named in the OSEK specification.

From the SSTG information, the dOSEK system generator (Figure 1) produces a kernel binary that is optimized for the actual application usage patterns. Optimizations include, for example, the avoidance of scheduler invocations for system call sites that have a known scheduling outcome.

2.3. State Transition Graphs. Our verification method is concerned with properties of and relations between STGs, which in process-theoretic parlance are essentially deterministic labelled transition systems. That is, given a set $\mathcal{A}$ of *labels* a state transition graph consists of a set S of *states* and a set $T \subseteq S \times \mathcal{A} \times S$ of *labelled transitions*. We write $s \xrightarrow{a} t$ for $(s, a, t) \in T$. The transition relation is required to be *deterministic* (but may be, and typically is, *partial*), that is, whenever $s \xrightarrow{a} t$ and $s \xrightarrow{a} t'$ then $t = t'$. We occasionally consider state transition graphs with *ε-transitions*; these additionally admit transitions of the form $s \xrightarrow{\varepsilon} t$ where ε is a special label not contained in $\mathcal{A}$. For ε-transitions, we do not require determinism. For example, the graph from Figure 3b is nondeterministic at state A w.r.t. ABB_1-transitions, which are replaced by ε-transitions as explained in Section 2.2.

To STGs with ε-transitions, we apply the usual process of *ε-elimination*, i.e. we insert an a-transition from state s to state t whenever t is reachable from s by first performing any number of ε-transitions (possibly none) and then an a-transition. We then remove all unlabelled transitions, and all states that become unreachable as a result. In general, this will produce a nondeterministic STG; we will explain in Section 3.2 why the particular STGs that appear in our verification framework do remain deterministic after ε-elimination.

3. The Formal Verification Method

Our formal verification methodology comprises a *vertical* and a *horizontal* verification process (cf. Figure 1). The central data structure for our verification is the SSTG (Section 2.2). In the *vertical verification*, we check that the SSTG adheres to key aspects of the behaviour specified by the OSEK standard, in particular regarding prioritized scheduling and interrupt handling. To this end, we formalize the corresponding parts of the OSEK standard in CTL and model check the SSTG against the arising temporal specification (Section 3.1). In the horizontal verification, we then ensure that the actual

generated code conforms to the originally projected system-wide control flow; this is achieved by graph isomorphism checking of the SSTG against a judicious abstraction of the code, viz. another STG called the *dynamic state-transition graph* (DSTG) (Section 3.2).

3.1. Vertical Verification. We next describe how we formally verify that the static state transition graph (SSTG) complies with the OSEK specification. To this end, we generate a NuSMV-model and CTL formulas that formalize (parts of) the OSEK specification. Our formalization currently covers the standard roughly up to conformance class ECC1, with the exception of alarms and resource management. As input for our generator we use the (ε-eliminated) SSTG and the OIL specification, which specifies all tasks and interrupts with their respective priorities, as well as all events and resources of the system.

The SSTG already resembles a NuSMV-model, except that NuSMV-models do not have edge labels. Thus, we need to convert the edge labels into state labels, and our generator does that by pushing labels along the arrows into the next state. This leads to a multiplication of states by the number of different labels of incoming edges. Of course, this conversion produces a nondeterministic model, however with a unique edge between two states as every syscall has a unique effect on the current state. In more detail, our NuSMV-model has as global state variables (a) the variable syscall that contains the name of the system call (i.e. the label of the edge) that took the system into the current state, (b) a variable state carrying the current state in form of the node id from the SSTG, and (c) variables for all other parts of the global system state. Note that the values of the latter variables are already determined uniquely by the node id; we included them merely to make the CTL formulas and counterexample traces more readable.

All OS objects, i.e. tasks, ISRs, events, and resources are realized by means of NuSMV-modules, which are instantiated as specified in the OIL specification of the system. They do not carry internal state but are merely used to group related variables for readability in formulas and error traces. For example, the state and (dynamic) priority of a task t are referred to by t.state and t.priority. The actual state is fed into the instances through global variables that form parameters of the modules. The next value of state at any state is chosen nondeterministically as one of the successor nodes of the current SSTG node (given by the current value of state). The next(syscall) is then uniquely determined by the current value of state and the value of next(state).

Some variables, for example the resource priorities, are not explicitly contained in the OIL file and thus have to be calculated according to the OSEK specification. The latter demands that resource priorities are at least the maximum priority of all tasks using the resource but are lower than the priority of every task not using the resource but having higher priority than the ones using the resource. To avoid priority collisions between tasks due to resource occupation, we scale all priorities by a factor of two and calculate the resource priorities as the maximum of all the priorities of

```
MODULE main()
VAR
  ...
  syscall : { Start, ..., TerminateTask, ...,
              ActivateTask_High, ..., interrupt_37, ... };
  running : { Idle, Low, Med, High, ISR };
  state : { ABB_67_0, ..., ABB_4_0, ..., ABB_23_0, ABB_24_0,
            ABB_25_0, ..., ABB_63_0, ... };
  ...
ASSIGN
  init(syscall) := Start;
  next(syscall) := case
    ((state = ABB_67_0) & next((state = ABB_4_0))) :
      ActivateTask_High;
    ((state = ABB_67_0) & next((state = ABB_23_0))) :
      interrupt_37;
    ((state = ABB_67_0) & next((state = ABB_24_0))) :
      interrupt_37;
    ((state = ABB_67_0) & next((state = ABB_25_0))) :
      interrupt_37;
    ((state = ABB_67_0) & next((state = ABB_63_0))) :
      TerminateTask;
    ...
    next(TRUE) : syscall;
  esac;
  init(running) := Low;
  next(running) := case
    next((... | (state = ABB_4_0))) : High;
    next((... | (state = ABB_23_0) | (state = ABB_24_0) |
          (state = ABB_25_0) | ...)) : ISR;
    next((state = ABB_63_0)) : Idle;
    ...
    next(TRUE) : running;
  esac;
  init(state) := ABB_67_0;
  next(state) := case
    ...
    (state = ABB_67_0) : {ABB_4_0, ABB_23_0, ABB_24_0,
                          ABB_25_0, ABB_63_0};
    ...
  esac;
  ...
CTLSPEC ...
```

Figure 4: Slice of the NuSMV-model for the graph in Fig. 3

the tasks using the resource plus one. This does not affect the scheduling behaviour and ensures that there can never be a situation where multiple tasks of the same dynamic priority are ready to run at the same time.

Figure 4 shows a slice of the NuSMV-model generated from the SSTG for the example system from Figure 2 (depicted in simplified form in Figure 3(b)). We include only the variables `state`, `syscall` and `running` and only the transitions from the starting state `ABB_67_0`, which corresponds to state A. The transition to `ABB_4_0` corresponds to the transition $\mathsf{A} \xrightarrow{\varepsilon} \mathsf{B} \xrightarrow{\mathsf{ABB_2}} \mathsf{C}$ and the one to `ABB_63_0` corresponds to the transition to the final state with label $\mathsf{ABB_3}$. The remaining three transitions essentially correspond to $\mathsf{A} \to \mathsf{K}$ but take into account that in reality an interrupt in the program of Figure 2 may happen before `test()` is executed or at the two points after the branch (i.e. just before `ActivateTask(High)` or just before `TerminateTask()`).

We verify that the SSTG adheres to the OSEK specification by model checking the NuSMV-model generated from the SSTG against the CTL formulas generated from the OIL file according to the OSEK specification. The latter arise by instantiating formula patterns that are parametric in the OIL configuration. Figure 5 shows an example formula,

to be discussed shortly. Additional formulas specify that transitions of event states, resource states and interrupt states are correct, that ISRs may only perform their allowed system calls etc.; see the full version [9] for details. We frequently need to quantify over finite sets that are obtained from the OIL specification or are specified by OSEK (Table 2); such quantifiers are just expanded into finite conjunctions or disjunctions.

The formula in Figure 5 expresses that the scheduling for a task t is correct, i.e. each transition made in the SSTG is legal, and every transition required by the specification is in fact made by the system. Note that the transitions from Waiting and Suspended directly to Running are not technically legal according the OSEK specification; but the required intermediate state where the task in question would be Ready is not observable to the application, and we therefore opt to allow the direct transitions. The formula is instantiated for every task, and, in a slightly modified version, for every ISR, as interrupt scheduling is to some extent left up to the implementation. The formula is structured into four subformulas $\psi_1, \dots, \psi_4$, each handling the allowed state transitions from one of the four possible starting states. To make this rather large formula readable we employ the following abbreviations for subformulas and properties:

- t.isHighestPriority specifies that control flow t *wants* to run, e.g. has state Ready or Running, and has higher priority than every other control flow that currently wants to run.
- t.isWaitingFor(e) states that the task t is waiting for the event e to occur.
- allOthersPreemptible(t) states that all tasks currently ready, other than t, are preemptible. Note that this formula needs to be used together with t.isHighestPriority to ensure that no interrupt is currently running and is needed altogether because when checking the starting transition of a non-preemptible task t, knowing t.isHighestPriority alone is not enough. For example, before the transition, t.isHighestPriority would always be false, since t was suspended, and after the transition, t would have already received the ceiling priority of the internal scheduler resource due to being non-preemptible.
- othersWillPreempt(t) denotes that task t will be preempted by another control flow due to witnessing a scheduling system call. This happens only if another control flow s currently has the highest priority, and if s is not an ISR, then t also has to be preemptible.
- waitSc(t) formalizes that task t is waiting for at least one event that is not set.

Note that the parameter e in t.isWaitingFor(e) is realized in our NuSMV-model as follows: For every task, an array containing the states of all its events is generated. The parameter e then simply is an index into this array.

Formula ψ_1 handles the case where t was previously suspended. Here, the only way for t to start up again is by a system call from the set Act (with a transition to Ready or Running depending on what else is currently running). In the case where t was previously Running, ψ_2 captures the

TABLE 2: Finite sets used in the OSEK formalization.

Set	Name/Description
SC	Scheduling calls, i.e. system calls that lead to a (re-)scheduling of tasks
Act(r)	System calls that activate the control flow (i.e. task or ISR) r
E(t)	Events of task t
TSC(r)	Terminating system calls, i.e. system calls that would lead to the termination of r
NPTasks	Non-preemptible tasks, i.e. tasks that cannot be preempted by higher priority tasks
RPreempt(r)	Control flows that could preempt r

possible transitions:

(1) There is another control flow that will preempt t (othersWillPreempt(t)) due to a system call that causes rescheduling, and t will become Ready.
(2) t issues a system call from the set TCS(s) signalling its termination, and becomes Suspended.
(3) t waits for one of its events. If the event is not set, t becomes Waiting.

In the case where t was previously Ready, ψ_3 formulates that t will either remain Ready, or become Running if it has the highest priority. Finally, in the case where t was previously Waiting, ψ_4 expresses that t keeps Waiting until one of the events it was waiting for is set and then becomes either Ready or Running, depending on whether it currently has the highest priority.

3.2. Horizontal Verification. To increase trust in the correctness of the actual generated code, we complement the verification that the SSTG complies with the OSEK specification (Section 3.1) with a verification procedure ensuring that the the generated code agrees with the SSTG. To this end, we extract a normalized STG, the *dynamic state transition graph (DSTG)*, from the actual binary. Interestingly, while one might expect the notion of agreement of the DSTG with the SSTG to be based on classical process-algebraic notions of equivalence such as bisimilarity [10], it turns out that the normalization process in fact guarantees agreement of the two STGs up to isomorphism. We therefore base our verification procedure on isomorphism checking, not only because we thus obtain stronger correctness guarantees but also because isomorphism checking of deterministic systems is computationally cheap, and in fact can be performed on-the-fly.

In more detail, we extract the DSTG by executing and probing the generated system binary with all possible syscall sequences that can originate from the given application. In order to facilitate exploration of the state space, we transform the tasks' ABB graphs (see Section 2.2) into a *mock-up*, a C-program that omits the processing logic of the application and retains only the control flow, and then run an external search procedure on the mock-up that traverses the state space depth-first. We generate the mock-up (see Figure 6 for a partial mock-up of the running example) in two steps: (1) We use tools from the dOSEK framework to generate function-local ABB graphs from the generated LLVM code. (2) From these ABB graphs, we generate C-code that emulates the control

$$
\begin{aligned}
&\mathsf{AG}((t.\mathsf{state} = \mathsf{Suspended} \rightarrow \psi_1) \wedge (t.\mathsf{state} = \mathsf{Running} \rightarrow \psi_2) \wedge (t.\mathsf{state} = \mathsf{Ready} \rightarrow \psi_3) \wedge (t.\mathsf{state} = \mathsf{Waiting} \rightarrow \psi_4)), \text{ where}\\
&\psi_1 \equiv (\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(\mathsf{syscall} \in \mathsf{Act}(t) \rightarrow (t.\mathsf{state} = \mathsf{Running} \leftrightarrow t.\mathsf{isHighestPriority})))\\
&\qquad \wedge (\neg\mathsf{allOthersPreemptible} \rightarrow \mathsf{AX}\,(\mathsf{syscall} \in \mathsf{Act}(t) \leftrightarrow t.\mathsf{state} = \mathsf{Ready}))\\
&\qquad \wedge \mathsf{AX}((t.\mathsf{state} = \mathsf{Suspended} \rightarrow \neg\mathsf{syscall} \in \mathsf{Act}(t))\\
&\qquad \wedge \mathsf{AX}((t.\mathsf{state} = \mathsf{Ready} \vee t.\mathsf{state} = \mathsf{Running}) \rightarrow \mathsf{syscall} \in \mathsf{Act}(t)))\\
&\psi_2 \equiv \mathsf{AX}((t.\mathsf{state} = \mathsf{Ready} \leftrightarrow \mathsf{othersWillPreempt}(t)) \wedge (t.\mathsf{state} = \mathsf{Suspended} \leftrightarrow \mathsf{syscall} \in \mathsf{TSC}(t)) \wedge (t.\mathsf{state} = \mathsf{Waiting} \leftrightarrow \mathsf{waitSc}(t)))\\
&\psi_3 \equiv (\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(\mathsf{syscall} \in \mathsf{SC} \rightarrow (t.\mathsf{state} = \mathsf{Running} \leftrightarrow t.\mathsf{isHighestPriority}))) \wedge \mathsf{AX}\,(t.\mathsf{state} = \mathsf{Running} \vee t.\mathsf{state} = \mathsf{Ready})\\
&\psi_4 \equiv \bigwedge_{e \in \mathsf{E}(t)} (t.\mathsf{isWaitingFor}(e) \rightarrow ((\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(e.\mathsf{set} \rightarrow (t.\mathsf{state} = \mathsf{Running} \leftrightarrow t.\mathsf{isHighestPriority})))\\
&\qquad \wedge (\neg\mathsf{allOthersPreemptible}(t) \rightarrow \mathsf{AX}\,(e.\mathsf{set} \rightarrow t.\mathsf{state} = \mathsf{Ready}))\\
&\qquad \wedge \mathsf{AX}\,((t.\mathsf{state} = \mathsf{Waiting} \wedge \neg e.\mathsf{set}) \vee t.\mathsf{state} = \mathsf{Running} \vee t.\mathsf{state} = \mathsf{Ready})))\\
&\mathsf{allOthersPreemptible}(t) \equiv \textstyle\bigwedge_{ot \in \mathsf{NPTasks} \setminus \{t\}} ot.\mathsf{state} \neq \mathsf{Running}\\
&\mathsf{othersWillPreempt}(t) \equiv \mathsf{syscall} \in \mathsf{SC} \setminus (\mathsf{TSC}(t) \cup \mathsf{WaitCalls}) \wedge \textstyle\bigvee_{or \in \mathsf{RPreempt}(t) \setminus \{t\}} or.\mathsf{isHighestPriority}\\
&\mathsf{waitSc}(t) \equiv \textstyle\bigvee_{e \in \mathsf{E}(t)} \neg e.\mathsf{set} \wedge t.\mathsf{isWaitingFor}(e)
\end{aligned}
$$

Figure 5: Example CTL formula

```
TASK(Low) {
  print_state_hash(at: ABB1, next: interrupt);
  trigger_interrupt();
  if (read_decision(0)) {
    print_state_hash(at: ABB2, next: ActivateTask);
    ActivateTask(High);
  }
  print_state_hash(at: ABB3, next: TerminateTask);
  TerminateTask();
}
```

Figure 6: Example system – generated mockup for task Low

flow, i.e. performs function and system calls as specified. Additionally, the mock-up:

- outputs node identifiers containing the identifier of the current ABB, as well as a hash of the actual current operating system state, where the latter includes the program counter;
- outputs identifiers for system calls containing the name of the system routine as well as the call site;
- optionally triggers any enabled interrupts;
- reads decisions on branching (including whether to trigger an interrupt) from standard input.

The mock-up is linked with the specialized kernel produced for the actual application by the dOSEK generator. It is then used by the *Dynamic State Explorer (DSE)*, a search procedure that generates the space of reachable states depth-first, steering the mock-up through the state space by feeding input to it in order to determine branching. The result of the search procedure is an STG. More precisely, some transitions in the STG are labelled with system calls as indicated above, and some are unlabelled, i.e. the STG includes ε-transitions; the latter correspond to internal transitions between computational ABBs. The ε-transitions are nondeterministic, since we omit the processing logic in the mock-up, so any form of conditional branching in the original application turns into nondeterminism; also, we cannot foresee external input. This STG is then subjected to ε-elimination as described in Section 2.3. We thus generate an STG with only labelled transitions; it is this STG that we refer to as the dynamic state transition graph (DSTG). The DSTG, as well as the SSTG, is deterministic, since every non-ε label is a tuple $(\textit{call site}, \textit{syscall type})$ and as such deterministically changes the system state. The same holds for interrupts, even though these are nondeterministic with regard to the activation time: Every interrupt transition label contains the interrupt number and, therefore, exactly describes its influence on the OS state, just like every other syscall-induced transition.

We check the DSTG for isomorphism with the SSTG. This is computationally unproblematic: since both LTS are deterministic after ε-elimination, we only need to check that both sides allow for the same transition labels, and then propagate this property to the states reached by the corresponding transitions. The reason that both graphs are isomorphic is that the states of the DSTG are identified by the hash value over the OS state, which is also reflected in the fields of every SSTG node.

4. Experiments

4.1. Positive Tests. To evaluate our verification method, we have run experiments on a number of OSEK systems. These systems stem from a test suite originally designed for the dOSEK implementation. In total, we have fully verified 58 test systems. We have selected eight systems that highlight key properties targeted in the verification (Table 3). The test cases “copter” and “copter-small” are the control software of a quadrotor copter and a simplified version thereof that arises by removing one asynchronous signal. For each of the systems, we list the number of system objects of the relevant types, i.e. interrupts, tasks (including the idle task), events, and resources specified in the OIL file.

Table 3 shows key parameters and the performance of the model checking tool on those systems. To qualify the generated NuSMV-model, we give the number of reachable states as well as its diameter (i.e. the length of the longest loop-free path). On a 2.4 GHz Intel Core i7-5500U machine with 8 GB of memory, the smaller experiments were completed within a fraction of a second. Only the verification of the two copter examples took longer, but finished in under three minutes.

For the horizontal verification, we probed the same 58 OSEK systems in two generator configurations (with and without system call site specialization) and established

TABLE 3: Performance of vertical verification.

Name	ISRs/Tasks/ Events/Res	Reachable states	Diameter	User time (sec)	Memory (MB)
bcc1-resource1j	0/6/0/4	26	16	0.10	27
bcc1-sse1c	0/6/0/4	24	14	0.08	28
ecc1-bt1g	0/6/2/2	10	10	0.05	24
ecc1-event1e	0/4/4/2	13	13	0.11	29
bcc1-isr2d	1/4/0/2	21	10	0.07	23
timing-abcomp	1/3/2/2	77	13	0.14	27
copter-small	3/11/0/3	1366	29	19.44	250
copter	4/12/0/3	4458	32	147.15	829

isomorphism with the respective SSTG in all cases. For most systems, the horizontal verification took less than 1 second. Only for the copter, the probing took 2.13s (0.81s for copter-small) and the isomorphism checking 0.17s (0.04s). When developing the hash function, we have probed examples with over 400.000 states in under 4 minutes.

4.2. Negative Tests and Fault Injection. For the vertical verification, we have performed *negative tests* by introducing faults into systems to check that these are correctly identified by our verification tool chain. To this end we have implemented a modified dOSEK generator that injects various types of faults into the input for our verification method. One can select between (a) mutations in the SSTG and (b) mutations in the input OIL specification. For (a), there is a random choice of either adding an edge or merging two states, and for (b), there is a random choice of either exchanging the priorities of two tasks or toggling the preemptability or the auto-start flag.

It should be noted here that not all faults introduced in this random way will necessarily lead to actual errors, e.g. when the change to the configuration does not influence the actual interaction between application and OS, or when additional transitions are actually valid. In our experiments, additional edges mostly did lead to errors and were detected by the formal verification; in some cases, additional edges produced legal transitions or violated parts of the specification not currently reflected in our formalization, e.g. that a task is not allowed to release resources it has not currently reserved. Merging graph nodes almost always produced errors caught by the verification, except in cases where the net effect would have been produced by ε-elimination anyway.

For the vertical verification, we injected 188 different faults into the test cases; this did not change the performance of the formal verification significantly compared to the unmodified test cases (Table 3). 177 faults lead to errors and were detected by the vertical verification. The other 11 faults were manually verified to be benign in the given usage pattern (a more detailed discussion can be found in the full version [9]). For the horizontal verification, we inserted 81 OIL-level faults: 61 lead to detected errors, while the other faults were manually checked to be benign.

4.3. Lessons Learned. Summing up the experience gained, it seems possible to achieve a fair degree of coverage in the verification of key aspects of the OSEK specification in tailored systems. (Unbounded) CTL model checking is feasible as the reachable part of the abstracted state space that we use remains within tractable range even for fairly large systems such as the quadrotor copter controller (with fewer than 4.5k states after abstraction); as stated above we attribute this fact to OS tailoring. Without wishing to get involved in the long-lasting linear-vs-branching-time war (e.g. [11]), we note that at the scale of our examples, LTL model checking does appear to reach the frontier of feasibility. For example, while in CTL, the copter-small example (Table 3) was discharged in under twenty seconds using less than 300 MB of memory, on the LTL correspondent of essentially the same specification we stopped the model checker after 30 minutes at 3.5 GB of allocated memory. This may be due to the higher formula complexity of LTL model checking (PSPACE instead of PTIME). A clear disadvantage of CTL, on the other hand, is that counterexamples are less informative, and typically stop at the first nested path quantifier.

Another somewhat surprising aspect is the fact that the notion of correspondence between the SSTG and the DSTG has turned out to be isomorphism of STGs. Actually getting this insight to work in full has required a somewhat laborious tuning process regarding the hashing of the OS state in the exploration of the DSTG, and in fact maintaining the tool chain in the future might be easier if one replaces isomorphism with strong bisimilarity.

5. Related Work

Our work is set in the highly active area of software model checking; see [12] for an (admittedly dated) overview. Our method exploits the high degree of predictability of scheduling afforded by OSEK, and in particular avoids the state space explosion caused by thread interleaving [13]. The static generation of state transition graphs from code in a somewhat similar style as featured in our approach has been used, in combination with LTL model checking, in the verification of event-condition-action systems [14].

The OSEK standard has been the subject of formal verification efforts to some degree. Waszniowski [15] modelled OSEK using timed automata within the UPPAAL model checker, and performed schedulability analyses. Huang et al. [16] modelled OSEK in CSP to verify various properties such as deadlock freedom. In this model, the internal application structure is not considered, and interrupts are excluded entirely. Vu et al. [17], [18] formalize the OSEK standard in Event-B and then verify designs of full OSs against the formalization. Where applications are considered [19], [20], these are verified in connection with an OS model rather than the actual OS implementation. In contrast, our approach avoids the verification gap between OS model and OS implementation by verifying the entire system composed of application and OS. This is made possible by focusing on the part of the OS behaviour actually relevant for the application at hand, instead of attempting to verify the full OS. We are thus able to a) work on the actual implementation, and b) fully model check the entire application/OS system including interrupts (expressly not covered in cited work on verifying OSEK applications).

Zhang et al. [21] formalize the OSEK standard in the K framework, along with the OIL and the programming language for applications. This is then used for test case generation and to verify applications by symbolic execution within the model. Interrupts are not considered.

Tigori et al. [22] use reachability checking of *extended finite automata* to remove dead code in tailored OSEK systems; the automata models involved are produced manually, while we generate STGs during code generation and from the actual code, respectively. Also, verification of OSEK adherence in [22] is by standardized testing, while we model-check the formalized standard.

On an entirely different scale, Klein et al. [23] formally verified the seL4 microkernel for functional correctness, in a project of 25 person years, and Sewell et al. [24] extended this verification from the C-Code level to the binary.

6. Conclusions

We have presented a framework for the fully automatic lightweight verification of tailored embedded systems following the OSEK industrial standard. Specifically, we have introduced a *vertical* verification process whereby the statically generated control flow of the tailored system is checked for conformance with the standard, and a *horizontal* verification method that ensures agreement between the static control and the actual generated code. Initial experiments run on a benchmark suite and on the control software of a quadrotor copter show promising results regarding the feasibility of full verification of key aspects of task interaction in OSEK systems, and in particular show that even substantial examples generate moderate-sized control flow graphs that allow fully-fledged model checking. The key to keeping state spaces small was to exploit OS tailoring as well as the particularities of scheduling in the OSEK standard. While our experiments indicate that dynamic exploration of control flow graphs scales up well, static methods that reconstruct the control flow from the compiled binary [24] may serve as a complementary approach in the future.

In further work, we plan to build more comprehensive coverage of the OSEK standard and to develop methods for validating our formalization of the standard against the informal specification, possibly building on previous work in this direction [17]. Also, we will apply our approach of model checking compiler-generated control flow graphs to application-specific verification goals beyond standard conformance.

Acknowledgments and Data. The authors thank the anonymous reviewers for their feedback. This work has been supported by the German Research Foundation (DFG) under the grants no. LO 1719/3-1, LO 1719/4-1, and SCHR 1118/5-3.

Source code and data are available at
https://gitlab.cs.fau.de/dosek-verification

References

[1] P. Marwedel, *Embedded System Design*. Springer, 2006.

[2] J. Cooling, *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.

[3] M. Broy, "Challenges in automotive software engineering," in *Proc. ICSE'06*. ACM Press, 2006, pp. 33–42.

[4] OSEK/VDX Group, "Operating system specification 2.2.3," Tech. Rep., Feb. 2005.

[5] AUTOSAR, "Specification of operating system (version 5.1.0)," Automotive Open System Architecture GbR, Tech. Rep., Feb. 2013.

[6] OSEK/VDX Group, "OSEK implementation language specification 2.5," Tech. Rep., 2004.

[7] C. Dietrich, M. Hoffmann, and D. Lohmann, "Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis," *ACM Trans. Embed. Comp. Sys.*, vol. 16, pp. 35:1–35:25, 2017.

[8] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *Proc. ISORC'10*. IEEE Computer Society Press, 2010, pp. 34–41.

[9] H.-P. Deifel, C. Dietrich, M. Göttlinger, D. Lohmann, S. Milius, and L. Schröder, "Automatic verification of application-tailored osek kernels," full version; available at ???

[10] R. Milner, *A Calculus of Communicating Systems*. Springer, 1980.

[11] M. Vardi, "Branching vs. linear time: Final showdown," in *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, ser. LNCS, vol. 2031. Springer, 2001, pp. 1–22.

[12] R. Jhala and R. Majumdar, "Software model checking," *ACM Comput. Surv.*, vol. 41, pp. 21:1–21:54, 2009.

[13] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," in *Proc. ICSE'11*. ACM Press, 2011, pp. 331–340.

[14] M. Schordan and A. Prantl, "Combining static analysis and state transition graphs for verification of event-condition-action systems in the RERS 2012 and 2013 challenges," *STTT*, vol. 16, pp. 493–505, 2014.

[15] L. Waszniowski and Z. Hanzálek, "Formal verification of multitasking applications based on timed automata model," *Real-Time Systems*, vol. 38, no. 1, pp. 39–65, Jan. 2008.

[16] Y. Huang, Y. Zhao, L. Zhu, Q. Li, H. Zhu, and J. Shi, "Modeling and verifying the code-level OSEK/VDX operating system with CSP," in *Proc. TASE'11*. IEEE Computer Society Press, 2011, pp. 142–149.

[17] D. Vu and T. Aoki, "Faithfully formalizing OSEK/VDX operating system specification," in *Proc. SoICT'12*. ACM, 2012, pp. 13–20.

[18] D. Vu, Y. Chiba, K. Yatake, and T. Aoki, "Verifying OSEK/VDX OS design using its formal specification," in *Proc. TASE'16*. IEEE Computer Society, 2016, pp. 81–88.

[19] H. Zhang, T. Aoki, and Y. Chiba, "Verifying OSEK/VDX applications: A sequentialization-based model checking approach," *IEICE Transactions*, vol. 98-D, no. 10, pp. 1765–1776, 2015.

[20] H. Zhang, T. Aoki, H. Lin, M. Zhang, Y. Chiba, and K. Yatake, "SMT-based bounded model checking for OSEK/VDX applications," in *Proc. APSEC'13*. IEEE Computer Society, 2013, pp. 307–314.

[21] M. Zhang, Y. Choi, and K. Ogata, "A formal semantics of the OSEK/VDX standard in K framework and its applications," in *Proc. WRLA'14*. Springer, 2014, pp. 280–296.

[22] K. Tigori, J.-L. Béchennec, S. Faucou, and O. Roux, "Formal model-based synthesis of application-specific static RTOS," *ACM Trans. Embed. Comp. Sys.*, vol. 16, pp. 97:1–97:25, 2017.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *Proc. SOSP'09*. ACM, 2009, pp. 207–220.

[24] T. Sewell, M. Myreen, and G. Klein, "Translation validation for a verified OS kernel," in *Proc. PLDI'13*. ACM, 2013, pp. 471–482.