Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis

CHRISTIAN DIETRICH, MARTIN HOFFMANN, and DANIEL LOHMANN,
Friedrich-Alexander-Universität Erlangen-Nürnberg

Cyber–physical systems typically target a dedicated purpose; their embedded real-time control system, such as an automotive control unit, is designed with a well-defined set of functionalities. On the software side, this results in a large amount of implicit and explicit static knowledge about the system and its behavior already at compile time. Compilers have become increasingly better at extracting and exploiting such static knowledge. For instance, many optimizations have been lifted up to the interprocedural or even to the whole-program level. However, whole-program optimizations generally stop at the application–kernel boundary: control-flow transitions between different threads are not yet analyzed.

In this article, we cross the application–kernel boundary by combining the semantics of a real-time operating system (RTOS) with deterministic fixed-priority scheduling (e.g., OSEK/AUTOSAR, ARINC 653, μITRON, POSIX.4) and the explicit application knowledge to enable system-wide, flow-sensitive compiler optimizations. We present two methods to extract a cross-kernel, control-flow–graph that provides a global view on all possible execution paths of a real-time system. Having this knowledge at hand, we tailor the operating system kernel more closely to the particular application scenario. For the example of a real-world safety-critical control system, we present three possible use cases. (1) Runtime optimizations, by means of specialized system calls for each call site, allow one speed up the kernel execution path by 28% in our benchmark scenario. Furthermore, we target transient hardware fault tolerance with two automated software-based countermeasures: (2) generation of OS state assertions on the expected system behavior, and (3) a system-wide dominator-region based control-flow error detection, both of which leverage significant robustness improvements.

CCS Concepts: • **Computer systems organization** → **Real-time operating systems**; *Reliability*; Embedded systems; • **Software and its engineering** → *Compilers;*

Additional Key Words and Phrases: Global control-flow graph, static analysis, OSEK, AUTOSAR, static system tailoring, whole-system optimization

ACM Reference Format:
Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. 2017. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. ACM Trans. Embed. Comput. Syst. 16, 2, Article 35 (January 2017), 25 pages.
DOI: http://dx.doi.org/10.1145/2950053

1. INTRODUCTION

Embedded real-time control systems are special-purpose systems: the built-in computers are dedicated to specific, predefined tasks [Marwedel 2006; Cooling 2003]. Thus, it

This work was partly supported by the German Research Foundation (DFG) under grant nos. LO 1719/1-3 (SPP 1500) and SCHR 603/9-1.
Authors' addresses: C. Dietrich and D. Lohmann, Leibniz University Hanover, Appelstrasse 4, 30167 Hanover; emails: {dietrich, lohmann}@sra.uni-hannover.de; M. Hoffmann, Friedrich–Alexander–Universität (FAU) Erlangen–Nürnberg, Martenstrasse 1, 91058 Erlangen, Germany; email: hoffmann@cs.fau.de.

DOI: http://dx.doi.org/10.1145/2950053

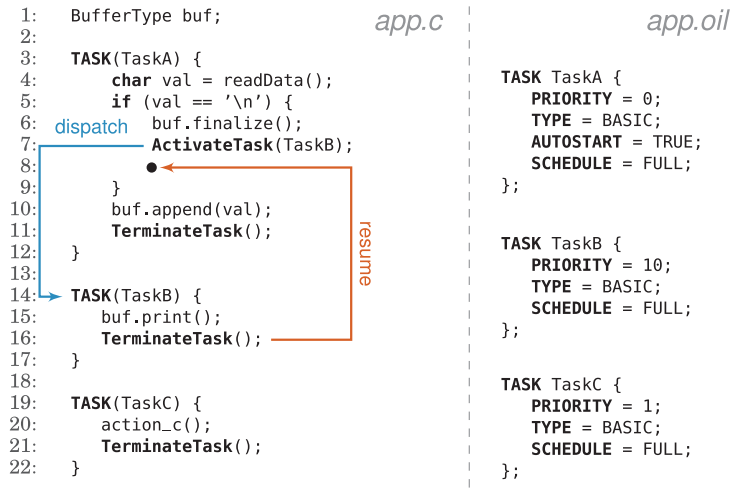


Fig. 1. A small OSEK system with three tasks. `TaskA` receives data and fills a buffer, which is processed by `TaskB`. `TaskC` is never activated. The right-hand side shows the static system configuration.

is possible (and common practice) to tailor both the hardware and system software of an embedded system to its specific needs in order to keep per-unit hardware costs as low as possible [Broy 2006].

In the software, the "special-purposeness" of an embedded system manifests in the large amount of implicit/explicit static knowledge that we have available already at compile time: the structure of the application code is typically static by nature. With respect to predictability, standards for embedded software development, such as MISRA-C [MISRA 2004], favor `static` data over stack-based or heap-based memory allocation, prohibit the use of function pointers, and suggest using constants whenever possible.

Thanks to *whole-program analyses* (WPAs), modern compilers can extract and exploit this static structure of the application to perform a great deal of interprocedural optimizations, such as constant folding, caller-site inlining, or elimination of dead code and data. Such optimizations become particularly effective if they are applied per thread [Shivers 1988], that is, if the compiler is made aware of the OS-managed control flows of the application and their respective entry points [Erhardt et al. 2011]. This is generally possible in embedded real-time systems, as the set of control-flows is also static: The *real-time operating system* (RTOS) itself is tailored toward the specific application; all HW/SW events are prioritized and mapped to a finite set of threads, *interrupt-service routines* (ISRs), semaphores, or other system objects. This makes it possible to allocate all system objects at compile time in static arrays and address them by their constant index value. The automotive OSEK/AUTOSAR [OSEK/VDX Group 2005; AUTOSAR 2013] RTOS standards, for instance, suggest this technique to keep the RAM overhead as low as possible. Figure 1 exemplifies such an OSEK-based system consisting of three tasks.

1.1. Problem Statement

Nevertheless, even a control-flow–sensitive WPA cannot provide a complete picture, as it does not cover control-flow transitions between *different* threads. These transitions are executed and managed by the OS scheduler on behalf of a syscall (e.g., posting a semaphore) and, thus, outside of the semantics of the programming language.

The common assumption is that the kernel might switch at any time to any other thread; thus, the compiler cannot derive constraints about interthread control-flow transitions.

However, while this pessimistic assumption is true for general-purpose operating systems that employ probabilistic scheduling of a dynamically changing set of threads (e.g., Linux, Windows), industry RTOS standards typically demand fixed-priority scheduling of a well-defined set of possible tasks, thus, are generally deterministic. In Figure 1, for example, the `AUTOSTART` thread `TaskA` sets thread `TaskB` ready (`ActivateTask(TaskB)`, line 7). As the static priority `10` of `TaskB` is higher than the static priority `0` of `TaskA`, we know *at compile time* that, at this point, the kernel of an event-driven RTOS will *always* dispatch to `TaskB`. Furthermore, as `TaskB` is activated only from `TaskA` (and itself does not activate any other task), we also know that, upon termination of `TaskB` (`TerminateTask()`, line 16), the execution will *always* continue in `TaskA`. A compiler that is aware of these facts could optimize the code to not invoke the kernel scheduler at these points. Even further, the compiler could inline the user-code of `TaskB` into `TaskA` and completely eradicate the respective system calls and the – then dead – `TaskB` object. Similarly, the compiler could detect that thread `TaskC` is also dead (the kernel will never dispatch to it); thus, it could also be eradicated. Eventually, the system from Figure 1 will be collapsed into a single thread, so that even the scheduler is no longer needed.

This basic example is, of course, simplistic. For instance, it does not contain any ISR that may activate a task at any time. Nevertheless, even with ISRs, it is typically possible in an event-triggered real-time system to derive *some* knowledge about inter-thread transitions in order to exploit this knowledge for truly global cross-kernel control-flow system optimizations.

1.2. About This Article

In this article, we describe two approaches to construct a *global control-flow graph* (GCFG) of a static fixed-priority, event-triggered real-time system. In addition to an ordinary (control-flow–sensitive) *control-flow graph* (CFG), the GCFG also incorporates the RTOS semantics, including interrupts and synchronization primitives, to model the control flow even across multiple threads and kernel invocations.

We show how, once obtained, the GCFG can be employed to speed up kernel activations in our benchmark system by 28%. It can also be employed to harden the kernel against transient hardware faults. Our software-based system-state assertions reduce the silent data corruption count by 45% for an already hardened system. A control-flow error-detection detection schema mechanism on the GCFG improves the silent data corruption (SDC) count by 14.

The presented optimizations are possible only by the whole-system view of the GCFG. Further measures only applicable on the system level, such as checking for specification conformity, are possible. In short, the GCFG information enables the lifting of optimizations, like dead code elimination, pointer-alias analysis, or constant propagation, from the (interprocedural) function level to the (interthread) system level.

This article is an extended version of our LCTES'15 conference paper [Dietrich et al. 2015b], in which we have already presented the *System-State Enumeration* approach to construct the GCFG (Section 3.2). This approach is precise, but can become intractable for larger systems due to its worst-case exponential complexity. In this article, we extend the previous work by the *System-State Flow Analysis* (Section 3.3), an alternative dataflow-based analysis to construct the GCFG that trades preciseness for a polynomial runtime. We also present an additional application scenario for the GCFG in Section 4.3 and new evaluation results in Section 5.

Table I. (Incomplete) List of System Services Provided by the OSEK API.
Not All Control Flows are Allowed to Invoke All System Services

System Service	Arguments	Brief Description
`ActivateTask`	`TaskID`	Task – `TaskID` – is activated. If the current task is preemptable, an immediate reschedule operation takes place.
`TerminateTask`	–	The current task terminates itself. An immediate reschedule operation takes place.
`ChainTask`	`TaskID`	The atomic combination of `ActivateTask(TaskID)` and `TerminateTask()`.
`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 preemptable tasks.
`DisableAllInterrupts`	–	Disables all interrupts.
`EnableAllInterrupts`	–	The inverse operation to `DisableAllInterrupts`.

2. SYSTEM MODEL

To achieve sound GCFG analyses, the underlying RTOS has to provide four basic properties: first, a deterministic scheduling policy, as, for example, fixed-priority preemptive scheduling. Second, all system objects must be declared before runtime, either in some dedicated configuration file or unambiguously in the application code itself. Third, system-service calls must be explicit, that is, indirect invocations via function pointers are not allowed. Finally, system objects must be referenced with compile-time constant identifiers or link-time constant addresses.

In practice, these requirements are already fulfilled or easy to achieve for event-triggered, hard real-time control systems – they are basically a technical consequence of predictability thus, already mandated by the dominant coding and RTOS standards of the domain. Examples include ARINC 653 (avionics), which prescribes fixed-priority scheduling within its time partitions [AEEC 2003]; μITRON (automation control, automotive) [Takada and Sakamura 1995]; and OSEK/AUTOSAR (automotive) [OSEK/VDX Group 2005; AUTOSAR 2013], but also the POSIX.4 real-time extensions (with SCHED_FIFO). Without loss of generality, we therefore describe our approach in the following on the example of the system model mandated by the OSEK-OS standard.

In our current implementation, the GCFG is constructed for a single core (OSEK-OS specifies a single core-system). However, our approach would also work for multicore systems with strictly partitioned scheduling, such as AUTOSAR 4.0 [AUTOSAR 2013].

2.1. Overview of OSEK-OS

OSEK-OS [OSEK/VDX Group 2005] has been the dominant industry standard for automotive RTOS for the last two decades. Originally intended for single-core, single-application systems, it has been extended for spatial and temporal isolation and multicore support in AUTOSAR-OS [AUTOSAR 2013], but the core API and concepts remained unchanged. Thus, all of the following equally hold for AUTOSAR-OS–based systems.

OSEK specifies terminology and the API for a completely statically configured event-triggered RTOS. For a specific automotive application, all system objects and their configuration have to be declared at compile-time in a compile time language, the *OSEK Implementation Language* (OIL) [OSEK/VDX Group 2004]. From this specification, the concrete RTOS instance is typically derived by a generator.

At runtime, the application manipulates the OS state by invoking *system services*, which influences the system behavior (Table I gives a short overview).

Control-Flow Abstractions: ISRs and Tasks. OSEK offers two main control-flow abstractions: ISRs and *tasks* (traditionally called 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 fixed-priority preemptive scheduling policy.

On each new activation, tasks start from the very beginning until their (self-)termination. Each task is configured to be either nonpreemptive (enforcing run-to-completion semantics) or fully preemptive (see `SCHEDULE = FULL` in Figure 1). Preemption points can be either *synchronous*, for example, caused by an explicit activation of a higher-priority task (e.g., `ActivateTask(TaskB)`, line 7), or *asynchronous*, if a higher-priority task is activated inside an ISR. Recurring, periodic, or aperiodic task activations can be triggered with the help of statically configured *Alarms*.

Synchronization Primitives: Global Lock and Resources. Inter task synchronization can be realized either by a coarse-grained global interrupt lock or more fine-grained *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.

The OSEK specification defines four conformance classes (BCC1, BCC2, ECC1, ECC2) describing the minimum requirements of the features provided by the system. BCC1/2 allows only *basic* tasks, which have a strict run-to-completion semantic. ECC1/2 introduces *extended* tasks, which are, furthermore, allowed to sleep on *Event* objects. In this work, we achieved different conformance classes for our two analysis methods: For our *system-state enumeration* (SSE) analysis, we target the conformance class ECC1 (i.e., extended tasks, but only with one task per priority). For the *system-state flow* (SSF) analysis, we target the BCC1 conformance class (i.e., basic tasks with one task per priority) plus the aforementioned resource concept. In principle, the other conformance classes could be supported as well; however, we have not yet implemented this.

Sources of Information. With these system objects at hand, we can construct real-time systems, composed of ISRs and tasks, which are activated by external or software signals and coordinated using interrupt blocks or resources. The OIL file statically defines and configures all objects, providing *coarse-grained* application knowledge. To achieve more *fine-grained* knowledge on the overall system behavior, a detailed analysis of the kernel–task interaction is necessary.

3. FINE-GRAINED INTERACTION KNOWLEDGE

One piece of fine-grained information about the application is interaction knowledge: how the application interacts with the kernel and the kernel's reaction. This section describes the (GCFG and presents two methods to extract it from the application's structure and system configuration: first, the SSE analysis, an expensive, but more precise method; second, the SSF) analysis, being of polynomial time, but resulting in more actually infeasible GCFG paths.

3.1. Global Control-Flow Graph

In many modern compilers, CFGs [Allen 1970] are the vehicle to capture the program logic of single functions. CFGs are directed graphs with basic blocks (BBs) as nodes and a single *entry node*. The functions' code is partitioned into BBs, in which the code in one BB can only be executed linearly. From a high-level perspective, an edge in the CFG between two BBs has an execution-order semantic; in every execution trace, two BBs can only follow each other, if and only if there is an edge in the CFG.

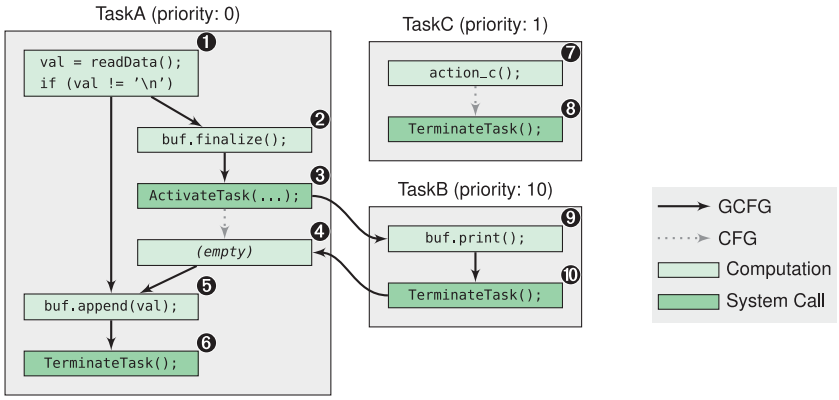


Fig. 2. The GCFG representation of the system shown in Figure 1. Dotted lines are part of the local CFG, but not part of the GCFG. The "dead" `TaskC` is not part of the GCFG at all.

We develop the GCFG semantic from the observation that the CFG expresses the BB execution order *within* a function. With a function call, control is transferred from the caller's to the callee's CFG. The *interprocedural control-flow graph* (ICFG) is formed by connecting all BBs in a program; it captures the execution order on the program level and respects control transfers caused by function calls. From the OS perspective, the ICFG expresses the execution order on the task (or thread) level. By rescheduling, the OS switches control between two tasks, and therefore between their ICFGs. The GCFG is one level higher; it expresses the execution order on the system level. If and only if there is an edge in the GCFG between two basic blocks, they may be executed directly after each other on the real hardware. Nevertheless, like regular CFGs, the GCFG can include infeasible paths.

In Figure 2, an example GCFG is shown for the application from Figure 1. `TaskA` has been assigned a low priority, while `TaskB` has a high priority. When the application does not interact with the kernel, the GCFG corresponds to the CFG execution order (e.g., ❶ → ❷, ❶ → ❺). Any system-service invocation (❸, ❻, ❽, ❿) requires the kernel to react. In the case of `ActivateTask` (❸), a task with higher priority is activated. According to the OSEK specification, the kernel scheduler chooses `TaskB` and dispatches to its entry block (❸ → ❾). After `TaskB` has terminated itself with `TerminateTask` (❿), the execution of `TaskA` is resumed (❿ → ❹). Here, we can observe that edges present in the CFG (❸ → ❹) are not necessarily part of the GCFG: block ❹ cannot execute directly after block ❸ on the system level.

For the construction of the GCFG, we have to answer two questions: (1) How do we partition the application code into blocks? (2) What edges have to be drawn between these blocks?

For the code partitioning, we use an adaptation of the *atomic basic block* (ABB) concept introduced by Scheler and Schröder-Preikschat [2010]. An ABB is a control-flow super structure that subsumes one or more BBs, forming a *single-entry–single-exit* (SE-SE) region; it has *exactly one* distinguished entry BB and one exit BB. Besides these two blocks, no BB has a preceding or succeeding block outside of the ABB region. Every BB is member of exactly one ABB. As an adaptation of the original ABB concept, we construct and connect the ABBs differently for the whole application at once:

(1) A function that contains a system call is a *system-relevant function*. Each function that calls a system-relevant function is a system-relevant function itself.

(2) We iterate over all basic blocks of all functions in the application: each basic block that contains system calls and/or calls to system-relevant functions is split directly before and after those locations into subsequent parts (see ABBs ❷, ❸, and ❹ in Figure 2).
(3) Depending on their content, we assign a type to each basic block: system-call block, function-call block, or computation block.
(4) We collect adjacent *computation* blocks into SE-SE regions.
(5) Each system-call block is an ABB, which contains a single system-call; each function-call block is an ABB, which contains a single function call; and each SE-SE region of computation blocks is an ABB.
(6) Within a function, the ABBs are connected into a local CFG corresponding to the connections of their entry and exit BBs.

After this construction, we have a *local* ABB-graph for each function within the application code. By the distinction of system-relevant functions, calls to system-*irrelevant* functions and subsystems are fully subsumed into computation ABBs. This subsumption not only reduces the number of blocks that we have to consider, but also sharpens the focus on the application logic that is visible to the OS. Interaction with the kernel is only possible in system-call blocks. Each system-call block has only computation blocks as direct neighbors.

The identification of SE-SE regions of computation blocks is possible by matching of control-flow patterns [Scheler and Schröder-Preikschat 2010], intersection of dominance and post-dominance regions, or from the program-structure tree [Johnson et al. 1994].

In Figure 2, the green blocks are ABBs. In the example, `readData()` is no system-relevant function; therefore, block ❶ is not split before and after the function call, subsuming the internal logic of `readData()`. The empty ABB ❷ is the result of the split operation that was performed because of the `ActivateTask` system call. Inserting empty ABBs ensures that each function-call and system-call block has only computation blocks as neighbours.

3.2. System-State Enumeration

To construct the GCFG, we combine three sources of information: (1) the *system semantics*, as defined by the OSEK specification; (2) the static *system configuration*, which is specified in the OIL file; and (3) the *application logic*, described by the local ABB graphs.

As the first combination method, we present the SSE. Briefly explained, the SSE computes all possible system states ahead of time and creates a *state-transition graph*. The resulting states are partitioned into groups depending on the ABB that they are currently executing. A GCFG edge between two ABBs exists, if and only if at least one state in the state group of the source ABB has an edge to a state in the state group of the target ABB.

Abstract System State Representation. The basis of the SSE analysis is the *Abstract System State* (AbSS) representation, which subsumes all relevant behavioral information of the system for single points within the control flow. Figure 3 depicts a single system state for the system from Figure 2. Each task declared in the OIL file is assigned a record with fields capturing its current task state and dynamic priority. For OSEK, each task can also hold zero or more resources, which are used to calculate the dynamic priority. The resume-point field contains the ABB to be executed next in the context of the task. Preempted tasks will continue their execution at this point. The resume point of the currently running task is the next block to be executed in the system context;

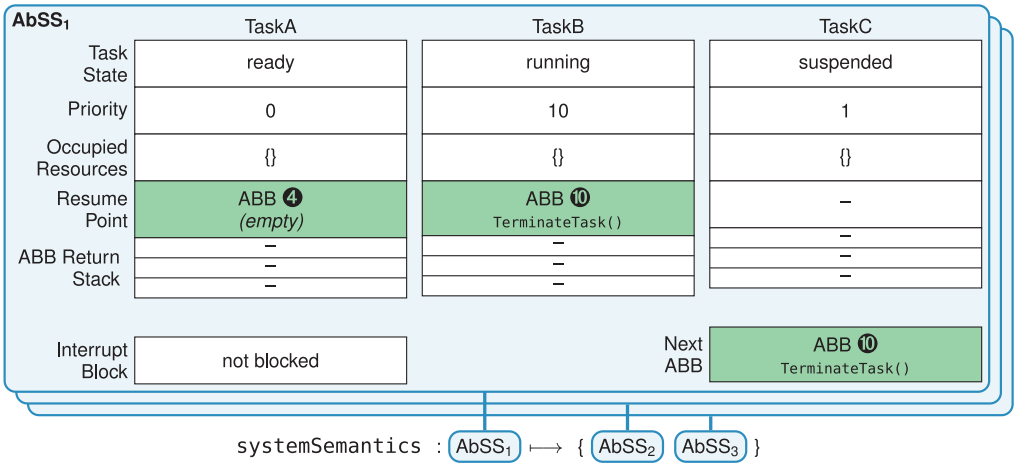


Fig. 3. A detailed view of an *Abstract System State* (AbSS). The `systemSemantics` function maps an input AbSS to a set of follow-up states.

the *next ABB*. The *return stacks* store return ABBs, which are pushed on function calls. *Interrupt block* indicates whether interrupts (ISR2s) are currently enabled, which is system-wide information.

SSE Algorithm: Basic Concept. The enumeration of all system states is achieved by the repetition of a step function until a fix point is reached (no new AbSSs appear). The step function pops one state from a working stack, calculates all subsequent states, inserts edges into the AbSS graph, and pushes the follow-up states onto the working stack if they were newly discovered. The calculation of follow-up states is based on three functions:

$$\texttt{systemSemantics} : \mathit{State} \longmapsto \{\mathit{State}\}$$
$$\texttt{schedule} : \mathit{State} \longmapsto \mathit{State}$$
$$\texttt{execute} : \mathit{State} \longmapsto \{\mathit{State}\}$$

The `systemSemantics` function maps the input state to a set of follow-up states, and is composed of the `execute` and the `schedule` function:

$$\texttt{systemSemantics}(x) \rightarrow \{\texttt{schedule}(y) \mid y \leftarrow \texttt{execute}(x)\}$$

The `schedule` function updates the dynamic priorities according to the resource occupation states, and chooses the next running task according to the scheduling rules mandated by OSEK. The `execute` function captures the influence of executing the next ABB. For each of the three block types, different rules apply.

For system-call blocks, all nonterminating system calls set the resume point of the currently running task to the computation block following in the local CFG. In Figure 3, the resume point of `TaskA` was set by the `ActivateTask` in ABB ❸. Then, `execute` transforms the input state according to the system-call type and arguments. This is enabled by the constraint that all system calls must be constant in location, type, and their arguments. In the example, `execute` evaluates the `TerminateTask` system call and returns a single state: `TaskB` is *suspended* and its resume point is reset. The `schedule` operation marks `TaskA` as the running task, and ABB ❹ will be executed next.

Function-call blocks push their single CFG-successor block onto the ABB return stack. When the execution of the called function reaches a computation block without CFG successors (exit node), an ABB is popped from the stack as the resume point.

Although computation blocks seem harmless, their `execute` semantic is the most complex. While all system-call and function-call blocks have a single successor, due to the ABB split operation, computation blocks may have *several* successors. For every CFG successor, `execute` emits a single follow-up state in which the next ABB is set to the successor block.

Furthermore, interrupts (alarms and ISR2s) occur only in computation blocks. With function-call and system-call blocks, we capture only the uninterruptible, atomic moment of control transfer between system-relevant functions or tasks. Therefore, all asynchronous signals are handled within computation blocks.

Interrupt Handling. In order to support interrupts in the system analysis, we create a virtual task for every ISR defined in the OIL description. These tasks are configured as nonpreemptable tasks, with fully disabled interrupts, and with the highest possible priority in the system. Therefore, our ISRs cannot be nested, which is one possible implementation according to the OSEK specification. For each declared alarm, we create an ISR containing a single `ActivateTask()` call.

The activation of an interrupt that is synchronized with the kernel can be treated like an asynchronous system call made by the hardware. If interrupts are enabled in the input state, `execute` emits one follow-up state for each ISR and each alarm, for which the virtual ISR task is set to ready. The resume point of the interrupted task is not changed; the interrupt will return to the exact same computation block. Then, `schedule` will always jump to the entry of the handler function, which is executed in a run-to-completion semantic.

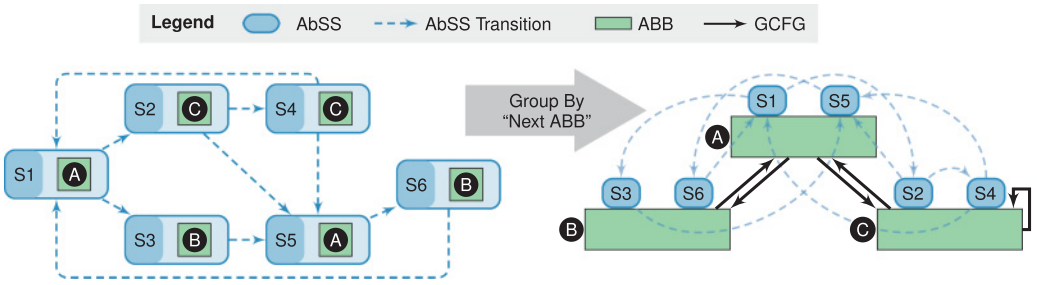
With firing every interrupt source in every computation block, we are on the safe side if no information is available about timing, minimal interarrival times, and execution times of computation blocks or interrupt handlers. By leaving the resumption point untouched, we capture multiple activations of a single interrupt and activations of multiple interrupts. Nevertheless, this approach has the drawback of a significant state explosion.

To ease this shortcoming, we provide the possibility to give additional coarse-grained information about the system configuration. The developer can declare groups of tasks; each group handles a single physical event. The interrupt that activates a group cannot fire again until all tasks in the task group have finished their execution. Providing this information is a qualitative statement about the execution time; the deadline of the task group's execution is shorter than the minimal interarrival time of the activating interrupt. As future work, quantitative timing information, such as the block *worst-case execution times* (WCETs) and precise interrupt timings, could be used to rule out some interrupt activations from the analysis.

Final GCFG Construction. Once the stepping function has reached a fix point, we have enumerated all possible system states. Based on the resulting state graph, we can construct the GCFG by partitioning all system states into state groups depending on their *next ABB* field. In each group, all states will execute the same ABB next. We add GCFG edges between the ABBs Ⓐ and Ⓑ if any state in group Ⓐ has at least one successor in the state group of ABB Ⓑ. The GCFG edge expresses: after Ⓐ has executed, it is possible to execute Ⓑ next.

Figure 4(a) shows a state-transition graph for a system consisting of three ABBs. This graph is the direct result of the SSE stepping function. In Figure 4(b), the state groups are drawn next to their ABBs; S3 and S6 belong to the state group of ABB Ⓑ. Since a state–state transition exists between S1 and S3, we insert a GCFG edge between ABB Ⓐ and ABB Ⓑ. ABB Ⓒ has a self-loop, since S2 can be followed directly by S4.

The fusion of all states within an ABB state group represents the expected system behavior at the entry of the respective ABB. The resulting predictive ABSS is a union



(a) State-Transition Graph (b) Derived Global Control-Flow Graph

Fig. 4. The abstract system states in the state-transition graph are partitioned into state groups according to their "Next ABB" field. A GCFG edge links two ABBs if any state Sn in the ABB's state group has at least one successor in the other's group.

of the individual (task) information fields of each involved AbSS. If a field provides different values in different states, we insert a "no information" marker leading to an imprecise state for this task field; otherwise, the task's state is unambiguous at this point. To give an example: if a task is marked as ready in S3, but suspended in S6, the predictive AbSS cannot provide information about this task; the task's state is not predictable at ABB B. On the other hand, if a task is denoted as suspended in each AbSS of a group, it is surely known to be suspended on entry of the corresponding ABB.

3.3. System-State Flow Analysis

The SSE, presented in the previous section is of exponential time complexity since it enumerates all possible system states. Therefore, a faster GCFG construction method, which may be more imprecise, is desirable. The SSF analysis is a regular dataflow analysis that uses already discovered GCFG edges to propagate the system state through the system.

For the SSF analysis, we put some further restrictions on the use of system services and the placement of system calls. Each ABB within a system-relevant function has to be reachable only from a single task; the task's CFGs have to be disjoint. Furthermore, the dynamic priority of the running task, which is influenced by OSEK's resource protocol, has to be constant for each ABB in the task's ICFG. Combined, we must know the currently running task for every ABB (`abb.task`) and its priority (`abb.prio`). Relaxing these restrictions and support for higher conformance classes than BCC1 is a topic of further research.

The imprecise states, which are used for the SSE to capture our knowledge at the beginning of each ABB, is the basic data structure for the SSF analysis. If we do not know the task state of a task, we mark it as **unknown**. Furthermore, the resume point of a task becomes a set of ABBs: the task will execute one of the following blocks in the next step. The next ABB field (resume point of the currently running task) has a set with exactly one entry; surely suspended tasks have an empty set.

Similar to the SSE, the `systemSemantics` function captures the OS behavior. The `execute` function, which modifies the AbSS according to the system call, is almost the same as for the SSE analysis; only minor adaptations were necessary. For example, the `ActivateTask()` system call adds the entry ABB of the task to the resume-point set if the task is suspended or unknown. When executing computation blocks, we use the ICFG successor blocks instead of the function-local successor blocks, which eliminates the need for an abstract call stack, but decreases the preciseness of the analysis. The interrupt activation within computation blocks is handled differently, and will be explained later on.

ALGORITHM 1 (a) Adapted `schedule` Operation for the SSF Analysis That Schedules on Imprecise System States. `schedule` : $State \longmapsto \{State\}$

```
Require: in_abss: imprecise AbSS to schedule upon
 1: possible_blocks ← []                    // List of tuples of the form (ABB, surely_ready)
 2: // Phase 1: Collect possible blocks
 3: for task in all_tasks do
 4:   if task_state(in_abss, task) is unknown then
 5:     for abb in resume_points(in_abss, task) do
 6:       possible_blocks append (abb, false)
 7:     end for
 8:   else if task_state(in_abss, task) is (ready ∧ running) then
 9:     min_prio ← min({abb.prio | abb ∈ resume_points(in_abss, task)})
10:     for abb in resume_points(in_abss, task) do
11:       // Only the block with the lowest dynamic priority is surely ready.
12:       surely_ready ← (abb.prio == min_prio)
13:       possible_blocks append (abb, surely_ready)
14:     end for
15:   end if
16: end for
17: // Phase 2: Sort blocks by priority; highest priority to the front.
18: possible_blocks ← sort(possible_blocks, sort_by = abb.prio)
19: // Phase 3: Dispatch to each ABB until the first surely running block
20: return_abss ← []
21: used_blocks ← []          // Already dispatched ABBs
22: for (abb, surely_ready) in possible_blocks do
23:   next_abss = copy(in_abss)
24:   // Dispatch virtually to each possible ABB
25:   task_state(next_abss, abb.task) ← running
26:   resume_points(next_abss, abb.task) ← {abb}
27:   remove_resume_points(next_abss, used_blocks)
28:   return_abss append next_abss
29:   used_blocks append abb
30:   if surely_ready then
31:     return return_abss
32:   end if
33: end for
```

The `schedule` operation has to be modified to work on imprecise system states. In Algorithm 1(a), the used scheduler is given in pseudo-code and operates on a single imprecise system state and emits several possible followup states. The scheduler works in three phases.First, a list of possible candidate blocks is generated. For each block, we store the information regarding whether the belonging task is surely ready. All tasks with unknown task states are not surely running (line 6). For ready and running tasks, only the block with the lowest dynamic priority is surely ready (line 12). In phase 2, the possible blocks are sorted by their dynamic priority. In phase 3, we "virtually" dispatch, by emitting an AbSS, to all candidates until we find the first surely running ABB. The emitted AbSS is modified to reflect the influence of the dispatching operation. The target task must be surely running (line 25) and has only the target ABB as resume point. Furthermore, we have to remove all resume points from the AbSS that were already considered in other follow-up states (line 27). If `remove_resume_points()` eliminates the last resume point of an task, it is known to be surely suspended. In the end, we have a list of possible follow-up AbSSs.

The special case in phase 1 for ready and running tasks solves a problem involving two tasks: we stop considering blocks after we have encountered the first surely ready block (line 30). If we have a surely ready task with two resume points at a high priority and low priority and a second unknown task with a block at a medium priority, we have to emit three follow-up states for each involved block since it is not sure that the first task will resume in the high-priority block.

ALGORITHM 1 (b) The System-State Flow Analysis. The SSF is a Data-flow Analysis That Adds the GCFG Edges Used for the Analysis During Its Traversal of the Graph. It Results in the GCFG.

```
Require: initial_state : : system state                    // Initial system state after StartOS()
 1: // System states are stored for blocks and for edges
 2: state_before = empty map of type (ABB → system state)
 3: edge_states = empty map of type ((ABB, ABB) → system state)
 4: working_stack = empty stack
 5: // Set up the working stack and fake the inputs for the initial block
 6: initial_abb = running_abb(initial_state)
 7: state_before[initial_abb] = initial_state
 8: edge_states[(initial_abb, initial_abb)] = initial_state
 9: push(working_stack, initial_abb)
10: // Run the fixpoint iteration until the working stack is empty
11: while not isEmpty(working_stack) do
12:    abb = pop(working_stack)
13:    state_before[abb] = merge_states(edge_states[(*, abb)])
14:    followup_states = system_semantic(state_before[abb])
15:    for next_state in followup_states do
16:       next_abb = running_abb(next_state)
17:       if (abb, next_abb) ∉ gcfg_edges then
18:          new_gcfg_edge(abb, next_abb)
19:       end if
20:       if next_state ≠ edge_states[(abb, next_abb)] then
21:          edge_states[abb, next_abb] = next_state
22:          push(working_stack, next_abb)
23:       end if
24:    end for
25: end while
```

The `systemSemantics` function is embedded into the SSF analysis, which is depicted in Algorithm 1(b). The analysis is a working-stack algorithm with two additional data structures. `state_before` stores the imprecise system state describing the system just before the execution of the ABB. `edge_states` stores the imprecise system state that "flows" from one ABB to the next. The before state is the combination of all incoming edge states.

After pushing the initial ABB (line 6) onto the stack, we pop ABBs from the stack until it is empty. For the examined ABB, we merge all incoming edge states (line 13) and apply the `systemSemantics` function to the result. For each possible follow-up state, we add a GCFG edge to the resume point of the running task if it is not already present (line 18). If there is no edge state for this GCFG edge or it differs from the stored one, we update (line 21) the edge state and push the follow-up ABB onto the working stack. During the algorithm, we gradually discover that more GCFG edges and the system states become more imprecise until we have reached the fix point.

In our flow-graph, we have at most $\#ABBs + \#ABBs^2$ system states, and each system state has $\mathcal{O}(\#tasks)$ variables with a fixed domain and $\#tasks$ resume-point sets, each

set having a maximum of #*ABBs* items. In each step, we change, in the worst case, only a single fixed-domain variable to unknown or add only a single ABB to a resume-point set. Therefore, we need at most $\mathcal{O}((\#ABBs \cdot \#tasks) \cdot \#ABBs^2)$ iteration steps with polynomial complexity itself. In total, our SSF analysis has polynomial runtime.

The presented approach would also work if interrupts are modeled the same way as in the SSE analysis. However, this approach would render the analysis nearly useless, since ISR entry and exit points are connected to all computation blocks. On these merge points within the GCFG, the system state would flow and mix until the states lost all precise information. We ease this problem by spawning a separate SSF analysis for each ISR activation. The subordinate SSF analysis starts with the before state of the computation block and stops with the follow-up state of the ISR exit block. The nonpreemptable ISR activation becomes a complex system call. Even with the spawning of subordinate SSF analyses, the overall analysis remains polynomial. After the (main) work stack is empty, we add all visited ISR activation and termination edges to obtain a complete GCFG.

3.4. GCFG Construction Summary

The result of both methods, SSE and SSF analysis, is a GCFG and a set of before-system states. Like in the SSE analysis, further information about interrupt activations can be supplied to the SSF by the developer to assist the GCFG construction. Nevertheless, the SSF-GCFG will most likely have more edges as a result of the system-state merging, but all SSE-GCFG edges will be included in the SSF-GCFG; the SSF-GCFG is a more imprecise overapproximation of the actual GCFG.

4. APPLICATION SCENARIOS

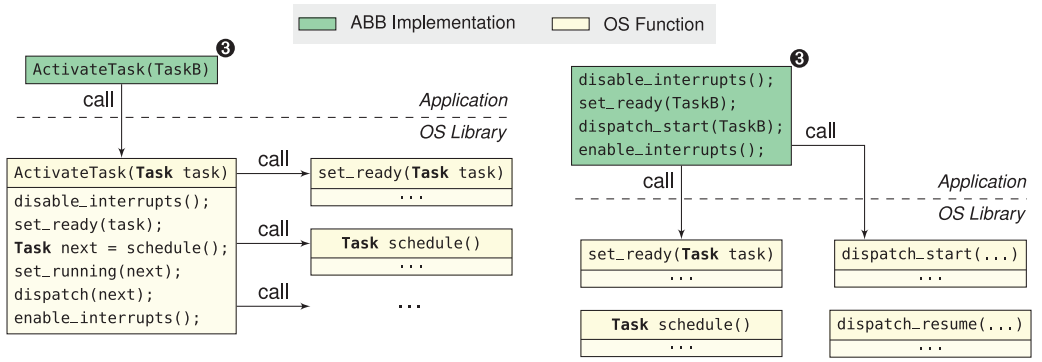
With the system analysis, we have gained two pieces of fine-grained knowledge about the interaction between application and kernel. First, GCFG edges, with system-call blocks as sources, represent all possible scheduling decisions after returning from the system call. Second, the predictive system states, computed for every ABB, describe the system before the block is executed. With this fine-grained information, we can optimize the whole system to different nonfunctional properties. In the following, we present three different optimizations that are enabled by the fine-grained information. The measures are discussed in detail next; the evaluation follows in Section 5.

4.1. System-Call Specialization

Most OSEK implementations are shipped as library operating systems. The library, which might be tailored with the coarse-grained knowledge from the OIL file, is linked against the application to generate the system image. System services are implemented as library functions and a system call boils down to a function call (see Figure 5(a)). The system-service implementation has to be generic, since the OS developer has no control over from which context the function is called.

With fine-grained interaction knowledge at hand, our system generator *automatically* tailors the system calls more specifically to the application behavior in order to speed up the kernel execution paths. Instead of calling the generic system service, we insert a specialized service at the call site. This decoupling enables us to use the interaction knowledge for selecting the minimum necessary functionality at that point.

In Figure 5(b), we specialize the `ActivateTask` system call from the running example. Since ABB ❸ has only one direct successor at `TaskB`, we do not have to call the scheduler; we already know the result and, therefore, directly dispatch `TaskB`. Furthermore, from the fact that all GCFG edges lead to the entry block of `TaskB`, we know that we will never resume but always start `TaskB` from the beginning. As the references to system objects are inserted as constants, the compiler might even inline the dispatching mechanism



(a) Common OSEK Library OS (b) Specialized System Calls

Fig. 5. System-call specialization uses the GCFG to extract possible scheduling decisions and generate tailored system service instances for each call site. In our example, `TaskB` is known to be the highest-priority task at ABB ❸, and can be dispatched without invoking the scheduler.

here. In cases in which the generator does not have enough information to insert a specialized version of the system service, it falls back to a default implementation.

Not only scheduler invocations can be avoided. Other complex system operations are substituted by mechanisms that only update the system state with a precalculated result of the operation. For example, if the dynamic priority of a task is unambiguous after a `ReleaseResource` system call, we do not have to determine it at runtime, but can update the OS state with a constant value. Depending on the OS implementetion, this might boil down to a single memory write at a constant address.

However, even if the result of a scheduling operation cannot be calculated completely at compile time, we allow a *partial specialization* of the schedule operation. From the GCFG edges, we can tell all possible scheduling outcomes. This information is used to tailor a scheduling operation that checks only for potentially runnable tasks. As their number is typically much lower than the total number of tasks, this particularly pays off if the scheduler's computational complexity depends on the number of tasks (e.g., $O(n)$). The downside of this tailoring is an increased code memory use, due to the extensive specialization.

4.2. Assertions on the Predicted System State

In addition to the kernel execution time, the resilience against transient hardware faults is a nonfunctional property of the operating system: can the kernel detect, or even recover, from a bit flip in its data structures? Caused by shrinking hardware structure sizes and lower operating voltages, the problem of transient hardware faults becomes increasingly important for automotive and other safety-critical control applications.

Software-based dependability measures allow for selective and resource-efficient robustness improvements. Generally, such measures–for example, triple modular redundancy and checksumming–are dynamic by nature; they check integrity by comparison of dynamically computed values. With fine-grained interaction knowledge at hand, we have compile-time information about the dynamic behavior of the application. Therefore, we can derive constraints that must hold for all possible execution paths. We enforce these constraints with runtime assertions at each system call. The predictive system states express the knowledge that we have about the system before an ABB is entered. Checkable pieces of the system state are, for example, the task state or the resumption points of the preempted task.

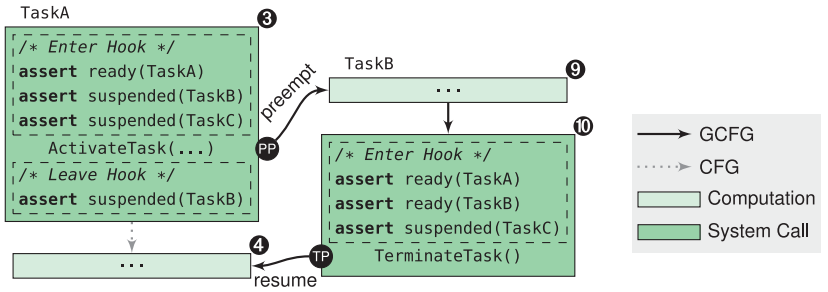


Fig. 6. Kernel enter and leave hooks, which are executed atomically with the system-call, provide assertions on the system state.

These assertions not only allow detection of corruptions in the kernel memory, but also errors in the control flow. Undesired, faulty jumps beyond the next expected system call, or even into another task's control flow, are detectable if the predicted system state does not match the current kernel state. This allows collection of different types of constraints for each system-call block, and generation of code that is executed *atomically* with the system call. We achieve atomicity by substituting the invocation at system-call site with a code sequence that is enriched by *kernel enter* and *kernel leave hooks*.

Figure 6 depicts such hooks, as well as the assertions on the tasks' states for ABB ❸ and ABB ❿ of our running example. We extract the constraints for a system call by inspecting different system states for enter and leave hook. The enter hook is filled with the predictions of the system-call block itself. Since the leave hook is executed only after the preemption point (PP), we use the predictive system state of the system call's local CFG successor. In the example (Figure 6), the enter hook is filled with constraints from ABB ❸, while the leave hook uses the predictions from ABB ❹.

The independent collection of constraints for enter and leave hooks leads to duplicate assertions. We can avoid double checks during a kernel activation to save run-time and code size. Each synchronous kernel activation consists of the system call's enter hook, the system call itself, and a leave hook of the resumed task. In the example, we return control from the termination point (TP) to the PP of `TaskA`. Therefore, the `TerminateTask()` activation consists of enter hook ❿, `TerminateTask()`, and leave hook ❸. We eradicate all assertions from leave hooks that are surely checked in all resuming enter hooks. In the example, we do not check `TaskA`'s and `TaskC`'s task state in leave hook ❸ since it is already checked in enter hook ❿.

4.3. Control-Flow Monitoring with Dominator Regions

In addition to the system-state assertions, we have developed another measure to harden a system against transient hardware faults, which exploits the GCFG information for a system to detect *control-flow errors*.

For the control-flow monitoring, we identify regions within the GCFG that must be entered through a single ABB. In this "root ABB," a marker is set. The marker must be present in all other ABBs that belong to this region. All blocks outside the region reset the marker again. If a marker is not present while executing a block within the region, the region was not entered through the root ABB; a fault must have occurred. Figure 7 shows an example of a control-flow region that sets a marker A in ABB_B, which is checked within the rest of the region. If a transient hardware fault leads to a jump from ABB_F directly into the region, the marker is not present and we detect the fault.

First, we need to identify those control-flow regions in the GCFG that can only be entered through a single block. Luckily, a well-known compiler technique called

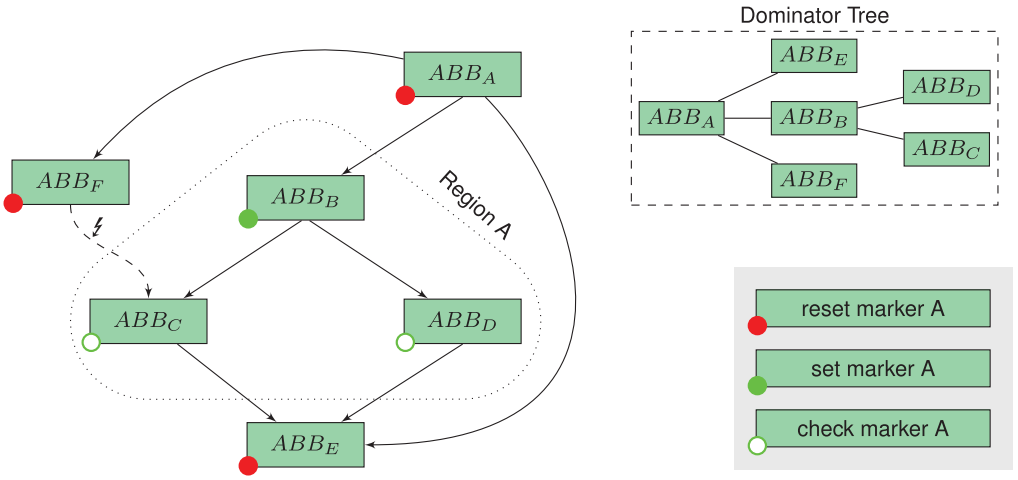


Fig. 7. Control Flow Region used for Fault Detection. Region A sets a marker A, which must be present in all system-call blocks of that region. The marker is reset in all blocks outside the region. If a faulty control flow jumps into the region, the marker is not present and the fault is detected.

dominator analysis [Prosser 1959] provides exactly the desired semantic, and fast algorithms for its calculation are available [Lengauer and Tarjan 1979].

Within a CFG, a block A *dominates* a block B if and only iff all possible execution sequences from the entry block visit block A before visiting block B. A block always dominates itself. In Figure 7, ABB_B dominates $\{ABB_B, ABB_C, ABB_D\}$. ABB_E is not dominated by ABB_B, since a execution path from ABB_A to ABB_E can bypass ABB_B. From the dominance relationship, we can calculate the *immediate dominator*. The immediate dominator of block A is its first predecessor that dominates block A. ABB_A is the immediate dominator of ABB_B, while ABB_B is the immediate dominator of ABB_D.

We calculate the dominance relationship and the immediate dominators for the system's GCFG. The immediate-dominator information is expressed as the *dominator tree*. In the dominator tree, each ABB is the child of its immediate dominator. Therefore, an ABB dominates all of its children transitively. For the example CFG, Figure 7 also includes the dominator tree.

In the dominator tree, we identify subtrees that have a system-call block as root node and include more than one system-call block. For each subtree, we allocate a marker. In our implementation, markers are single bits in a machine-word-sized variable. In the region entry, we set the marker bit; outside the region, we clear it. We check the presence of 32 markers at once with word-sized bit operations; if we are inside a region and a marker is missing, we detected an error. Bit flips in the marker word might result in a false-positive detected error, but will never propagate to a failure.

With this approach, we can only detect control-flow errors that jump *into* a region. Control-flow errors that leave a region due to a fault cannot be detected since we clear the marker bit in every outside block. As future work, we could clear the markers only when the region is left to detect faulty region exits.

4.4. Further Use Cases

In addition to system-state asserts and control-flow monitoring, we condensed the state-transition graph into a finite state machine that captures the exact scheduling behavior of the RTOS for a particular application. This state machine facilitates instance validation or can be transformed into a hardware component [Dietrich et al.

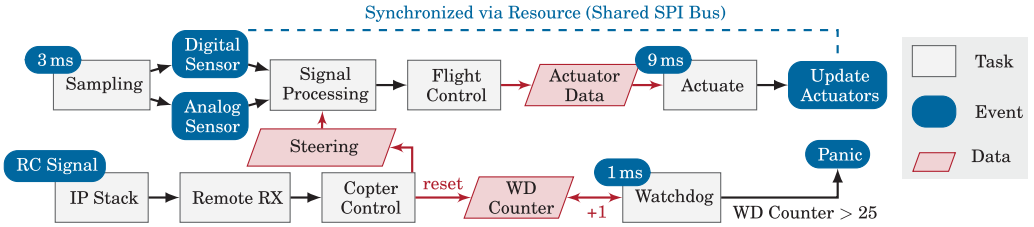


Fig. 8. The flight-control application of the *I4Copter* quadrotor helicopter.

2015a]. Another direction of future research is the improvement of WCET analysis by fine-grained interaction knowledge. Global flow information could assist cache analysis and would allow us to analyze OS code in a flow-sensitive manner.

5. EVALUATION

As an evaluation platform, we use the existing *d*OSEK OS generator. It is designed as a dependable OS that is resilient against transient hardware faults in memory and registers [Hoffmann et al. 2015]. The generative approach of *d*OSEK is a perfect fit for the presented analysis and optimizations. *d*OSEK is available in two basic configurations: *unprotected* and *protected*. Only the protected variant includes dependability measures against transient hardware faults. The presented analyses and optimizations were integrated into *d*OSEK.

5.1. Scenario

For our evaluation scenario, we use the *I4Copter* [Ulbrich et al. 2011], a safety-critical embedded control system (quadrotor helicopter) developed in an industry project with Siemens Corporate Technology.

We analyze the task setup of the *I4Copter* control application (Figure 8): tasks are activated both periodically and sporadically by interrupts. Intertask synchronization is realized with OSEK resources; a watchdog task observes the remote control communication. In total, the scenario consists of 11 tasks, three periodic interrupts (alarms), one sporadic interrupt, and one resource.

We replaced the application logic with checkpoint markers since we are only interested in the *interaction* between application and OS. The substitution does not change the GCFG or the analysis, but touches only the contents of the computation blocks. The ABB construction emits 86 ABBs and 52 system-call blocks.

5.2. Analysis Quality and Runtime

With our system analysis, we gather fine-grained knowledge about the interaction between applications and the OS. We will quantify the amount of knowledge that we can gather with our methods by comparing their predictive power regarding all system execution sequences. The higher the predictive power of a method, the more *impossible* execution sequences are sorted out.

The result of both analyses is a GCFG with ABBs as nodes. The lesser amount of edges this graph has, the lesser amount of execution sequences are possible and the higher is the predictive power of the GCFG. For example, the least informative GCFG, with the worst predictive power, is the fully connected graph. It proposes that every block–block transition is possible; every ABB can be followed by any other ABB. For our evaluation scenario, the fully connected graph has 86 nodes and 7,396 edges, including self loops, and does not reveal information about the system's behavior.

Table II shows the results for our evaluation scenario. We implemented both methods within the *d*OSEK generator with the Python 3 scripting language. We measured the

Table II. Analysis Quality and Runtime for the Benchmark (86 System-call ABBs)

	System-State Enumeration		System-State Flow	
	w/o Ann.	w/ Ann.	w/o Ann.	w/ Ann.
GCFG Edges	266	243	320	313
Analysis Runtime [s]	470.2	2.77	0.23	0.24
Memory Consumption [MiB]	3,539.91	43.21	0.65	0.65

Note: With more detailed application knowledge, we can construct sharper GCFGs.

analysis runtime on an i7-2600 with 3.4GHz, with one thread of execution. The memory consumption is the increased space requirement for the Python objects that are created during the analysis without garbage collection.

Without the qualitative information about the interarrival time of interrupts, which was discussed in Section 3.2, the SSE constructs a GCFG with 266 edges, but uses over 3.5GB of memory and took over 7 minutes to complete. During the analysis, a state-transition graph with 1,563,169 states and 2,098,236 transitions is constructed. With the user-supplied annotations, the analysis consumes only a fraction of time and memory, while the number of GCFG edges is reduced by 8.65%. The state-transition graph has 20,063 states and 23,876 transitions. This reduction results from the prohibition of interrupt activations.

In contrast to the SSE analysis, the system-state flow analysis takes less than a second for the evaluation scenario, regardless of the user-supplied annotation. Also, the memory consumption stays below 1MiB. On the downside, the information value of the resulting GCFG is lower. Since the SSE analysis is an overapproximation, we know that all additional edges within the SSF-GCFG are superfluous. Without the annotations, the SSF-GCFG is 20.3% larger than the SSE-GCFG; with the annotation, at least 28.81% of the SSF edges are superfluous.

In addition to our *I4Copter* application, we analyzed a synthetic application with a pathological structure as a stress test for our analyses. This generated OSEK-conformant application consists of 250 tasks that are organized in 125 pairs. The lower-priority thread of a pair is activated by a hardware-driven OSEK alarm (125 sporadic events) and drives a software counter that activates the pair's high-priority task. Between the pairs, we introduced 42 cross-pair dependencies, such as cross-pair task activation and execution chaining.[1] For this synthetic application, the SSE was unable to derive a GCFG. The SSF extracted the GCFG, which consists of 2,844 ABBs and 61,016 edges, in 564 seconds.

5.3. Runtime and Code Size of Specialized Kernel Fragments

The presented methods increase the amount of application knowledge significantly. But what influence can we take on the nonfunctional properties of the system? First, we quantify the time the *d*OSEK kernel takes to execute. Therefore, the system executes for three hyperperiods on an IA-32 emulator, while, at the same time, an execution trace is recorded. During the benchmark, 411 system calls are issued.

Table III shows the average time the system remains in the kernel for a system call. We measure the time in instructions, although this number is not linearly correlated to execution time on modern pipelined, out-of-order processors. Nevertheless, instruction counts are more comparable over different hardware versions. Further, many of the mentioned CPU features are not yet available for embedded platforms, which are mainly used for real-time systems.

[1]Note that these numbers would be extremly high for a real-world system. OSEK conformance mandates support for 16 tasks and even leading proprietary implementations support not more than 255.

Table III. Code Size of the *d*OSEK Kernel and Runtime for Three Hyperperiods

	Unprotected *d*OSEK (SSE)		Protected *d*OSEK (SSE)		Protected *d*OSEK (SSF)	
	Code Size (Bytes)	Runtime (Instructions)	Code Size (Bytes)	Runtime (Instructions)	Code Size (Bytes)	Runtime (Instructions)
Baseline	33,263	31,347	67,966	114,308	67,966	114,308
+ Specialize	10,859	23,688	24,458	81,962	31,135	87,872
+ Dom. Regions	34,369	32,481	69,200	115,743	69,088	115,393
+ Assertions	46,331	38,842	81,884	124,372	79,099	122,272

Note: System analysis was done with user supplied annotations and the specified analysis method.

The unprotected *d*OSEK needs 31,347 instructions for the whole benchmark. If we enable the system-call specialization (SSE) with support for instantiating a partial scheduler, we gain 24%.

The improvement for the protected *d*OSEK is even larger, since the protected kernel operations are much more expensive compared to the unprotected system. It starts on average with 114,308 instructions for the unmodified, but protected, *d*OSEK system. Specialization (SSE) reduces the kernel time by 28%.

Inlining the system service into the system-call sites has an impact on the code size. *d*OSEK always creates an inlined instance of the system service for each call site. This inlining increases the resilience against hardware faults by avoiding function calls within the kernel execution. For the benchmark, baseline *d*OSEK requires 33,263 bytes of program memory for the unprotected, and 67,966 bytes for the protected variant. Compared to this, the system-call specialization reduces the code size for each system call to 10,859 bytes (unprotected, SSE), respectively, 24,458 bytes (protected, SSE). Using the more imprecise SSF analysis results in a smaller performance benefit in the specialized system. Conversely, the overhead on state assertions and dominator regions decreases, since less knowledge is available with the SSF-GCFG.

Of course, when integrated into a commonly developed library-based OSEK, the system-call specialization will in general *not* reduce the code size. Nevertheless, the code overhead per system-call site will be in the same range as for *d*OSEK.

5.4. SDC Count Decrease

The enrichment of the system with system-state assertions is a measure against transient hardware faults. Therefore, we used the FAIL* [Schirmeier et al. 2015] fault injection framework for an extensive injection campaign on the presented evaluation scenario. We used a single-event, single-bit fault model, which emulates transient hardware faults caused by radiation or voltage fluctuations. According to our fault model, a single-bit flip occurs at one point in time in one location that is visible on the instruction set architecture. As locations, we not only examine the memory, but also general-purpose registers, the flag register, and the instruction pointer.

The benchmark scenario, which was augmented with checkpoint markers, runs for three hyperperiods, and simultaneously, visits 172 checkpoints. It is the task of the OS to adhere to this checkpoint sequence, even in the presence of hardware faults. If the kernel cannot provide the correct activation order or corrupts the application data, and is not able to detect the fault, we record an SDC. If the fault was not benign, but the kernel detected it, we hand over control to the application; the fault is *not* counted as an SDC.

With FAIL*, we executed the system with the deterministic BOCHS [Lawton 1996] IA-32 emulator. Into each execution, we injected one single-bit flip into the operating system's memory or registers and observed the system's reaction. Due to the focus on the OS execution, the replacement of the application logic does not influence the

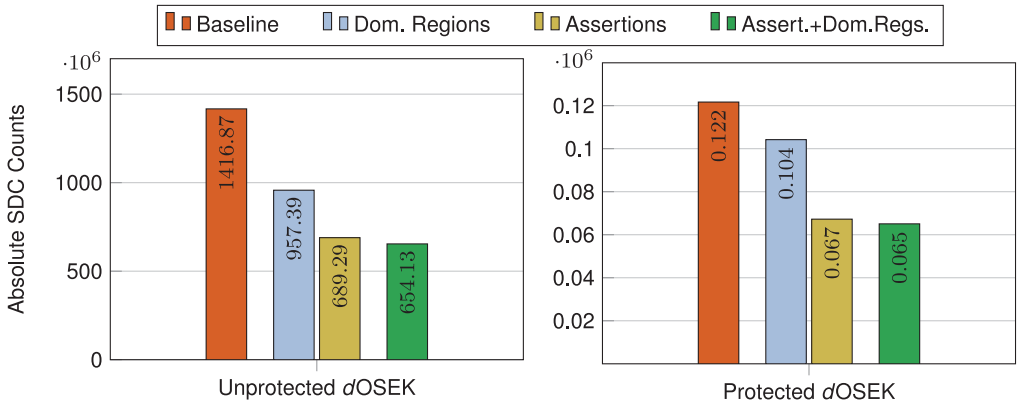


Fig. 9. Absolute SDC Counts. Influence of inserting state assertions and dominator-region-based control-flow monitoring on the unprotected *d*OSEK (left) and the protected *d*OSEK (right). SSE analysis was used.

results. For the benchmark, the injected faults cover the *entire* effective fault space. For the fault injection, we always use the SSE-GCFG.

In Figure 9, we present the results of the injection campaign in absolute SDC counts for different configurations. We start with the unprotected baseline *d*OSEK with $1416.87 \cdot 10^6$ SDCs for the scenario. Nearly all of these SDCs stem from faults in the main memory, since this variant does not protect the OS state at all.

The system-state assertion optimization inserts 748 assertions into 48 system-call sites; 639 assertions were introduced into the enter hooks and 109 assertions into the leave hooks. The insertions add 13,068 bytes of code to the system and increase its runtime by 24% (see Table III). By spending these overheads, we reduce the SDC count by 51%. These improvements originate mostly from the protective nature of the measure on the OS state in memory ($-728 \cdot 10^6$). In total, faults in registers, instruction pointer, and flags are negligible for the unprotected *d*OSEK.

The control-flow monitoring identified 25 dominator regions and inserted region checks into all 52 system-call sites, which resulted in a code-size increase of 1106B bytes of code. The runtime increased by 3.62%, while the absolute SDC count dropped by 32%.

The protected *d*OSEK has several magnitudes better starting conditions. For the baseline, the system reveals $0.12 \cdot 10^6$ SDCs for the benchmark. Due to encoded operations, memory SDCs dropped significantly, while the other fault locations remained on the same level as in the unprotected system.

If we insert the same number of assertions into the protected *d*OSEK, we can further reduce the number of SDCs by 45%. Here, the improvements stem from reducing register and instruction-pointer faults (-39%, -77%) that occur during the kernel operation. The system-state assertions are able to detect incorrect or incomplete results directly after they have been written back to memory in the kernel leave hooks.

The insertions of control-flow-region checks into the encoded *d*OSEK revealed an SDC decrease of 14%.

A combination of control-flow monitoring and system-state assertions did not reveal a significant increase in robustness. For the unencoded *d*OSEK, adding control-flow monitoring to the system-state assertion variant showed another decrease of SDCs by 5% (encoded: -3%).

6. DISCUSSION

One of the main challenges in analyzing OSEK systems with the SSE approach is the size explosion of the state-transition graph. In theory, a system could have an

exponentially higher amount of states compared to the number of tasks and basic blocks. Therefore, measures to ease this exponential burden are crucial.

6.1. Design Recommendations to Tackle the State Explosion

With the ABB abstraction, we reduce the number of blocks in the system significantly. Subsuming blocks that do not interact with the kernel sharpens the focus on the application–kernel interaction and abstracts from the application's microstructure. Whole library hierarchies and algorithms can be hidden within a single ABB. As a general design principle, we further recommend avoiding system calls that modify the kernel state, deep in the call hierarchy. Although deeply buried system calls do not impede our analysis, they still result in many split basic blocks and thus complex and confusing GCFGs. Often, such hidden system calls reveal surprising side effects and activation sequences. In order to reduce the complexity of the analysis – thus of the entire system—we suggest planning the real-time application in large computation blocks, which are not sliced by synchronous syscalls.

6.2. Dynamic Task Preemption and Interrupts

Our approach does support indeterminism caused by interrupts and subsequent task activations, which may also lead to dynamic preemption of the running task. Our goal is to extract and exploit as much static knowledge *as possible* to constrain the real indeterminism. By analyzing the interrupt handler code, our system generator knows the tasks that are potentially activated dynamically and, thus, also the tasks that potentially are preempted at runtime (i.e., have a lower priority and may be running when the interrupt arrives).

Nevertheless, as interrupt requests fork the state-transition graph in every computation block, they are the main driver of state explosion. However, in real-time systems, interrupts are rarely *totally* unpredictable, which opens many optimization possibilities for further state reduction. In order to maintain analyzability, developers already have to determine minimal and maximal interarrival times. Furthermore, parts of the application are synchronized with these signals. With the simple task-group annotation, we let the developer already express this knowledge about signal–signal handling causality, which in the case of safety-critical systems has to be determined anyway.

Further knowledge of the peripheral-device behavior regarding interrupts could be provided by IO protocols. Here, a *logic of actions* on application level could be derived. For example, it could be defined that a "send buffer empty" interrupt could only occur within a specific time after the associated `SendMessage()` function had been invoked.

6.3. Scalability of the SSE and SSF Analysis

As a rough estimation, we can assume that the SSE will scale linear in runtime to the number of system states. We cannot deduce how large a system may be, to remain manageable. Nevertheless, we already can handle real-world scenarios without having further assistance by the developer. As a topic of further research, methods from the symbolic-execution community could be applied to cut down on the analysis runtime, and other methods could be developed to construct the GCFG.

In Section 5.2, we have seen that SSE and SSF analyses have different objectives. While the first is slow but results in a more precise GCFG, the second works in polynomial time but results in more infeasible paths; the resulting GCFG is not as informative as in the SSE case. Furthermore, the SSF analysis currently imposes more strict limitations onto the system and supports only conformance class BCC1 (no events and waiting states). Nevertheless, the performance benefit for specialized system calls is comparable for SSF analysis (23.13%) and SSE analysis (28.3%).

6.4. Threats to Validity and Applicability

The major threat to validity of the experimental findings is that they are based on single case study only. However, the flight control of our quadrotor flying vehicle is a real-world safety-critical system that we have developed in close cooperation with Siemens Corporate Technology [Ulbrich et al. 2011]. Thus, we consider it to be reasonably representative for these kinds of applications in size and RTOS usage.

The major threats to the general applicability of our approach are the restrictions we put on (1) the RTOS semantics and (2) the application structure. In essence, our approach extracts and exploits *inherent determinism* that is available at compile time due to the RTOS semantics and its utilization by the application. While this works reasonably well for fixed-priority scheduling, the usefulness is limited on systems that offer significantly less determinism, such as an RTOS with an *earliest deadline first* (EDF) scheduler or any kind of scheduler that performs online acceptance tests. On the application side, all interactions with the RTOS have to be detectable at compile time. This forbids any sort of dynamic code loading, the invocation of syscalls via nontrivial function pointers, and syscall arguments that are not computable at compile time.

For the domain of safety-critical embedded control systems, however, these restrictions impose little impact in practice—they are prescribed and demanded by the relevant industry and safety standards anyway EDF scheduling, for instance, is barely used in embedded control systems; the relevant industry standards (such as OSEK/AUTOSAR, ARINC 653, μITRON, but also POSIX.4) all employ fixed-priority scheduling; the usage of function pointers and any sort of dynamic code modifications is discouraged by the relevant coding and safety standards (e.g., MISRA [2004] and ISO 26262-4 [2011]). In summary, most of our requirements have to be fulfilled anyway by embedded control systems that need to pass certification authorities. Here, our approach can even support the development process: Incorrect API usage within the application code, violating the specified RTOS semantic, is immediately uncovered by the GCFG analysis.

Other current restrictions, such as multiple tasks per priority as well as multiple task activations, are no conceptual problem for the SSE analysis. Multiple task per priority can be mapped to the existing model by letting all tasks of one priority share a common resource to serialize their execution. Multiple task activations can be modeled by an activation queue within the AbSS, finally leading to a larger state graph.

7. RELATED WORK

Bertran et al. [2006] proposed a global view on the interaction between operating system and application. They constructed a GCFG for a complex embedded system, which was built on top of Linux. System-call entry points and library entry points were connected to the corresponding call sites. On this GCFG, dead-code elimination in terms of removing uncalled system calls and unreferenced library functions resulted in a reduced code size of the system image. In contrast to this work, their analysis was flow-insensitive and did not take the semantic of system calls into account. Basically, they extended the CFG into the kernel, but not out of it.

Barthelmann [2002] makes use of the static semantics of an OSEK system to minimize the task contexts to be saved at specific preemption points. A static analysis reveals an *interference graph* describing mutual preemptions of basic blocks, based on the tasks' static priorities. With this knowledge, an optimized *intertask* register-allocation is performed, including context-switch code generation. Similar to our approach, application knowledge is used to influence a nonfunctional property of an OSEK kernel. In contrast to our work, the scheduling semantic of OSEK was used in a flow-*in*sensitive manner. This means that, the interference graph includes superfluous preemptions

that are actually impossible according to a GCFG analysis. This approach corresponds to the "System Configuration" case from Table II. Nevertheless, the paper describes a first approach of a generative whole-system optimization taking both the application and the OS into account.

The OSEK semantic also found attention in the area of formal methods and verification: Waszniowski and Hanzálek [2008] designed a model of the OSEK standard targeting the UPPAAL model checker. They modeled all components as timed automata, also taking interprocess communication (OSEK events) into account. Their main focus was the verification of different application properties and schedulability analyses. Huang et al. [2011] modeled OSEK as communicating sequential processes (CSP). The application subtasks were modeled without considering the internal application structure; interrupts were excluded entirely. With this model, they could verify different properties of their OSEK system, like dead-lock freedom and freedom of priority inversion. Regarding our approach, these models could provide a more formal definition of the `systemSemantics` function.

System specialization was already discussed by the OS community for general-purpose systems. Pu et al. [1988] developed the Synthesis kernel, which included a code synthesizer that produced optimized code paths at runtime for often invoked system calls, for example, `read` or `write`. Due to manual implementation of code templates, which are then filled by the synthesizer, huge performance benefits result from shorter kernel execution paths. In comparison to the dynamic Synthesis system, our approach of the system-call specialization also takes the in-depth application knowledge into account, but is executed offline. Pu et al. [1988] also mention the problem of code-size explosion. McNamee et al. [2001] used *Tempo*, a partial evaluator for C programs, and a set of specialization predicates to identify functions automatically for specialization within the kernel. In their approach, the specialization was also done dynamically at runtime, omitting detailed static application knowledge.

Several approaches to control-flow monitoring were developed for application logic. Benso et al. [2001] employ regular-expression automata to check the correctness of executed BB sequences of an application. Yau and Chen [1980] divided the CFG into loop-free regions. For each region a database of possible paths is encoded and checked during the execution. Several *software signature* mechanisms for single basic blocks were proposed, for example, ECCA [Alkhalifa et al. 1999], CFCSS [Oh et al. 2002], or YACCA [Goloubeva et al. 2003]. These methods maintain one or more runtime signature registers, which are continuously updated and checked at the beginning and the end of each basic block. CEDA [Vemu and Abraham 2008] uses dominance information to lift signature-based schemes to larger SE-SE regions. Nevertheless, it does not use dominator regions with multiple exits. All mentioned approaches only consider the CFG of a single function or task, but could be extended to catch control-flow errors on the ABB and GCFG level.

8. CONCLUSION

Real-time systems include a large amount of concealed static knowledge about their dynamic behavior. With the presented methods, we make this knowledge accessible for OSEK-like systems and exploit it to optimize nonfunctional system properties. We presented two methods to calculate the *global control-flow graph*, which covers all possible system execution paths. The exponential *system-state enumeration* as well as the polynomial *system-state flow* analysis combine application logic, the system configuration, and the OS specification. We employ the fine-grained interaction knowledge to inline *specialized system services* into the system-call sites. *System-state asserts* check statically derived constraints at runtime, and control-flow checks based on dominator regions improve the resilience against transient flow errors. With these applications

of our fine-grained knowledge, we could speed up the kernel execution by 28% and decrease the soft-error vulnerability by 47%.

Source Code and Raw Data

The analysis source code, which is published as free software under GPLv3+, is available at github.[2] The results that lead to the numbers in this article were calculated by an automated experiment workflow. The *d*OSEK version used for all results in this article have the git commit hash 54e0aa4. Raw results are available at our website.[3]

REFERENCES

Airlines Electronic Engineering Committee (AEEC). 2003. Avionics Application Software Standard Interface (ARINC Specification 653-1).

Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. 1999. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.* 10, 6 (June 1999), 627–641. DOI:http://dx.doi.org/10.1109/71.774911

Frances E. Allen. 1970. Control flow analysis. *SIGPLAN Not.* 5, 7 (July 1970), 1–19. DOI:http://dx.doi.org/10.1145/390013.808479

AUTOSAR. 2013. *Specification of Operating System (Version 5.1.0)*. Technical Report. Automotive Open System Architecture GbR.

Volker Barthelmann. 2002. Inter-task register-allocation for static operating systems. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES/SCOPES'02)*. ACM, New York, 149–154. DOI:http://dx.doi.org/10.1145/513829.513855

A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. 2001. Control-flow checking via regular expressions. In *Proceedings of the 10th Asian Test Symposium 2001 (ATS'01)*. IEEE, Washington, DC, 299–303. DOI:http://dx.doi.org/10.1109/ATS.2001.990300

Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. 2006. Building a global system view for optimization purposes. In *2nd W'shop on the Interaction between Operating Systems and Computer Architecture (WIOSCA'06)*. IEEE, Washington, DC.

Manfred Broy. 2006. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*. ACM, New York. 33–42. DOI:http://dx.doi.org/10.1145/1134285.1134292

Jim Cooling. 2003. *Software Engineering for Real-Time Systems*. Addison Wesley.

Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. 2015a. Back to the roots: Implementing the RTOS as a specialized state machine. In *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'15)*. 7–12.

Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. 2015b. Cross-kernel control-flow-graph analysis for event-driven real-time systems. In *Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'15)*. ACM, New York. DOI:http://dx.doi.org/10.1145/2670529.2754963

Christoph Erhardt, Michael Stilkerich, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2011. Exploiting static application knowledge in a Java compiler for embedded systems: A case study. In *Proceedings of the 9th International Workshop on Java Technologies for Real-time & Embedded Systems*. ACM, New York, 96–105. DOI:http://dx.doi.org/10.1145/2043910.2043927

O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. 2003. Soft-error detection using control flow assertions. In *Proceedings. of the18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. 581–588. DOI:http://dx.doi.org/10.1109/DFTVS.2003.1250158

Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. 2015. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS'15)*. IEEE, Washington, DC, 259–270. DOI:http://dx.doi.org/10.1109/RTAS.2015.7108449

Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. 2011. Modeling and verifying the code-level OSEK/VDX operating system with CSP. In *Proceedings of the 5th International Symposium on Theoretical Aspects of Software Engineering (TASE'11)*. IEEE, Washington, DC, 142–149. DOI:http://dx.doi.org/10.1109/TASE.2011.11

[2]https://www.github.com/danceos/dosek.

[3]https://www4.cs.fau.de/Research/dOSEK/data/tecs16/.

ISO 26262-4. 2011. *ISO 26262-4:2011: Road vehicles – Functional safety – Part 4: Product Development at the System Level*. ISO, Geneva, Switzerland.

Richard Johnson, David Pearson, and Keshav Pingali. 1994. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, New York, 171–185. DOI:http://dx.doi.org/10.1145/178243.178258

Kevin P. Lawton. 1996. Bochs: A portable PC emulator for Unix/X. *Linux Journal* 1996, 29es (1996), 7.

Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141. DOI:http://dx.doi.org/10.1145/357062.357071

Peter Marwedel. 2006. *Embedded System Design*. Springer, Heidelberg, Germany.

Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. 2001. Specialization tools and techniques for systematic optimization of system software. *ACM* 19, 2 (May 2001), 217–251. DOI:http://dx.doi.org/10.1145/377769.377778

MISRA. 2004. Guidelines for the Use of the C Language in Critical Systems. ISBN 0 9524156 2 3.

N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability* 51, 1 (2002), 111–122. DOI:http://dx.doi.org/10.1109/24.994926

OSEK/VDX Group. 2004. *OSEK Implementation Language Specification 2.5*. Technical Report. OSEK/VDX Group. Retrieved from http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf.

OSEK/VDX Group. 2005. *Operating System Specification 2.2.3*. Technical Report. OSEK/VDX Group. Retrieved from http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

Reese T. Prosser. 1959. Applications of Boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference (IRE-AIEE-ACM'59 (Eastern))*. ACM, New York, 133–138. DOI:http://dx.doi.org/10.1145/1460299.1460314

Calton Pu, Henry Massalin, and John Ioannidis. 1988. The synthesis kernel. *Computing Systems* 1, 1 (1988), 11–32.

Fabian Scheler and Wolfgang Schröder-Preikschat. 2010. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *EDCC IEEE International Symposium on OO Real-Time Distributed Computing (ISORC'10)*. IEEE, Washington, DC, 34–41. DOI:http://dx.doi.org/10.1109/ISORC.2010.11

Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. 2015. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC'15)*, Pierre Sens (Ed.). 245–255.

O. Shivers. 1988. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*. ACM, New York, 164–174. DOI:http://dx.doi.org/10.1145/53990.54007

Hiroaki Takada and Ken Sakamura. 1995. μITRON for small-scale embedded systems. *IEEE Micro* 15, 6 (1995), 46–54. DOI:http://dx.doi.org/10.1109/40.476258

Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. 2011. I4Copter: An adaptable and modular quadrotor platform. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC'11)*. ACM, New York, 380–396.

R. Vemu and J. A. Abraham. 2008. Budget-dependent control-flow error detection. In *Proceedings of the 14th IEEE International On-Line Testing Symposium (OLTS'08)*. 73–78. DOI:http://dx.doi.org/10.1109/IOLTS.2008.52

Libor Waszniowski and Zdeněk Hanzálek. 2008. Formal verification of multitasking applications based on timed automata model. *Real-Time Systems* 38, 1 (Jan. 2008), 39–65. DOI:http://dx.doi.org/10.1007/s11241-007-9036-z

S. S. Yau and Fu-Chung Chen. 1980. An approach to concurrent control flow checking. *IEEE TOSE* SE-6, 2 (Mar 1980), 126–137. DOI:http://dx.doi.org/10.1109/TSE.1980.234478

Received August 2015; revised January 2016; accepted May 2016