FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

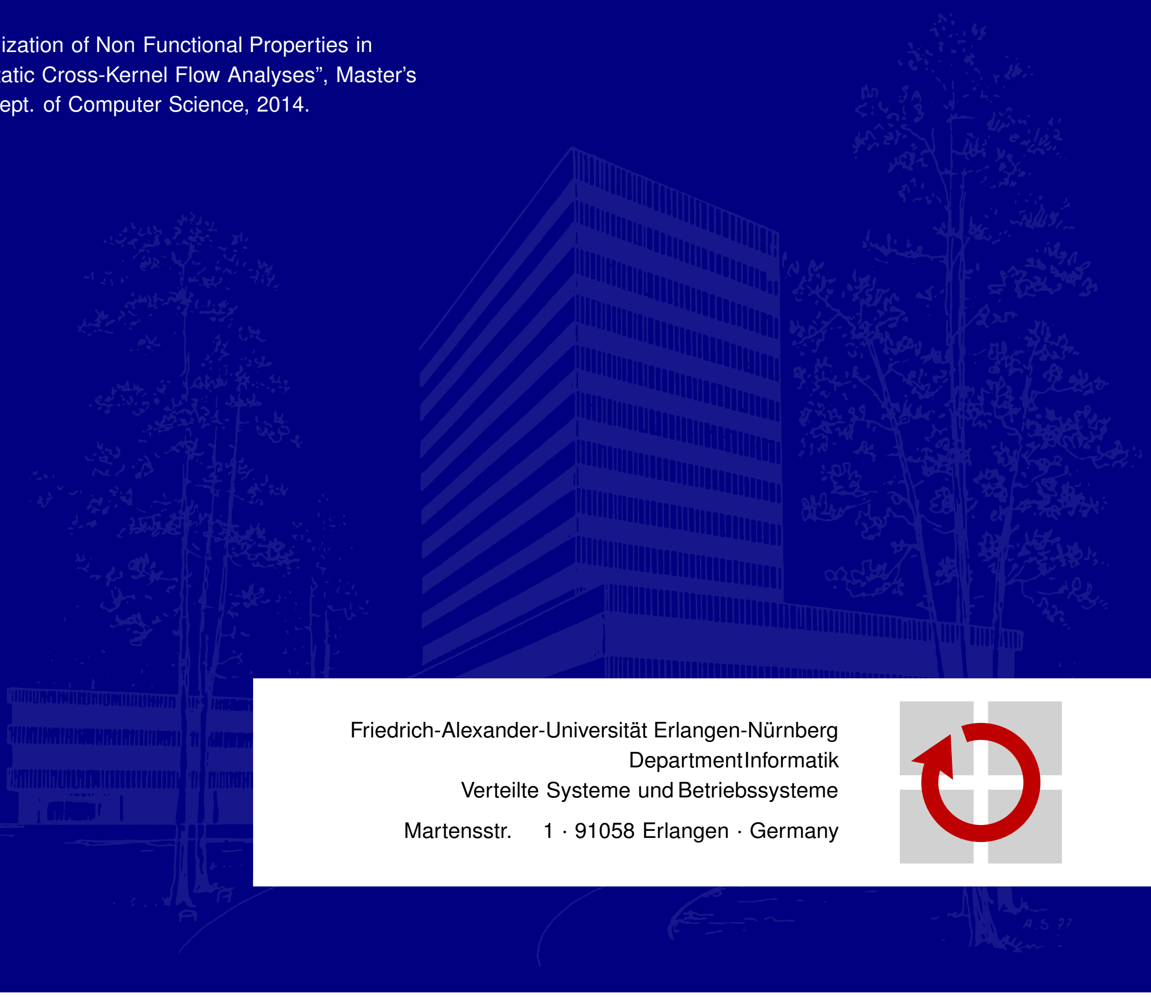
Christian Dietrich

Global Optimization of Non Functional Properties in OSEK Real-Time Systems by Static Cross-Kernel Flow Analyses

Masterarbeit im Fach Informatik

1. September 2014

Please cite as:
Christian Dietrich, “Global Optimization of Non Functional Properties in OSEK Real-Time Systems by Static Cross-Kernel Flow Analyses”, Master’s Thesis, University of Erlangen, Dept. of Computer Science, 2014.



Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

www4.informatik.uni-erlangen.de

Global Optimization of Non Functional Properties in OSEK Real-Time Systems by Static Cross-Kernel Flow Analyses

Masterarbeit im Fach Informatik

vorgelegt von

Christian Dietrich

geb. am 5. Dezember 1989
in Rothenburg o.d.T.

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dipl.-Ing. Martin Hoffmann**
Betreuender Hochschullehrer:	**Dr.-Ing. habil Daniel Lohmann**
Beginn der Arbeit:	**1. April 2014**
Abgabe der Arbeit:	**13. Oktober 2014**

Abstract

Computer systems are part of almost every aspect in our life. They are not only present in the form of the general-purpose desktop computing systems, but far more often they are hidden in the form of embedded real-time systems. These systems are often small and have very strict resource constraints. Therefore, application and the underlying real-time operating system are shipped as a single unit. Because of this tight coupling, the operating system can be tailored precisely to the needs of the application.

In the automotive industry, the OSEK standard has emerged to build embedded systems. This standard requires preconfiguration of the required system resources to allow a static tailoring process. Until now, tailoring an OSEK system utilizes only the knowledge about the operating-system objects, like tasks and alarms, but no knowledge about the inner structure of the application and its interaction with the operating system is used.

In this work, the approach of the global control-flow graph is introduced for OSEK-like operating systems. The global control-flow graph expresses all possible system transitions between the tasks the application consists of. It can be calculated by a data-flow analysis on the control-flow graph of the application or by enumerating all possible system states.

The in-depth application knowledge is used not only to tailor the operating system to the static requirements of the application, but also to generate the kernel code exactly towards the dynamic behavior of the application. Furthermore, the application knowledge allows the generation of application specific software-based measures against transient hardware-faults.

The approach is integrated into *d*OSEK, a dependable OSEK implementation, and the resulting real-time systems are evaluated in several dimensions: code size, kernel run time, and rate of silent data corruptions. As a result, a *d*OSEK instance was constructed that uses 49.03 percent less flash memory for code and has a 42.59 percent lower run time. The applied dependability measures reduced the silent data corruption rate by 34.84 percent compared to the best, until now, available *d*OSEK.

Kurzfassung

Computer Systeme berühren beinahe jeden Teil unserer alltäglichen Erfahrung. Sie sind nicht nur in der Form von Allzweckrechensystemen gegenwärtig, sondern viel häufiger in der Form von eingebetteten Echtzeitsystemen anzutreffen. Diese Systeme sind meistens klein und stark limitiert in ihrem Ressourcenverbrauch. Aus diesem Grund werden Anwendung und das darunterliegende Betriebssystem als eine Einheit ausgeliefert. Wegen dieser engen Kopplung ist es möglich das Betriebssystem präziese an die Anforderungen der Anwendung anzupassen.

In der Automobilindustrie hat sich der OSEK Betriebssystemstandard etabliert um eingebettete Systeme zu konstruieren. Dieser Standard verlangt vom Anwendungsentwickler eine statische Vorkonfiguration, in der alle benötigten Systemobjekte, wie Ausführungsstränge und Alarme, aufgeführt sind. Diese Vorkonfiguration erlaubt einen statischen Anpassungsprozess des Betriebssystems vor der Laufzeit. Bis jetzt wird in diesem Anpassungsprozess nur diese Objektinformationen verwendet um Datenstrukturen statisch zu erzeugen; Informationen über die innere Struktur der Anwendung und deren Interaktionsmuster mit dem Betriebssystem werden nicht ausgenützt.

In dieser Arbeit wird der Ansatz des Globalen Kontrollflussgraphen für OSEK-ähnliche Betriebssysteme einegeführt. Dieser Globale Kontrollflussgraph drückt alle möglichen Systemtransitionen zwischen den verschiedenen Ausführungssträngen aus. Berechnet wird der Globale Kontrollflussgraph entweder durch eine Datenflussanalyse auf den Kontrollflussgraphen der Ausführungsstränge oder durch eine vollständige Aufzählung der möglichen Systemzustände.

Das tiefgehende Anwendungswissen kann nicht nur für die Anpassung des Betriebssystems an die statischen Anforderungen verwendet werden, sondern auch um einen Betriebssystemkern zu konstruieren, der auf das *dynamische* Verhalten der Anwendung hin angepasst ist. Desweiteren erlaubt das Anwendungswissen die Erzeugung von spezialisierten Software-basierten Härtungsmaßnahmen gegen flüchtige Hardwarefehlfunktionen.

Der präsentierte Ansatz wurde in *d*OSEK, eine verlässliche OSEK Implementierung, eingebaut und die resultierenden Echtzeitsysteme auf verschiedene Aspekte hin untersucht: Programmgröße, Kernlaufzeit und die Rate der stillen Datenbeschädigungen. Als Resultat wurde eine *d*OSEK Instanz konstruiert, die 49.03 Prozent weniger Programmspeicher benötigt und eine 42.59 Prozent geringere Kernlaufzeit aufweist. Gegenüber dem bisher besten verfügbaren *d*OSEK, verbesserten die angewendeten Härtungsmaßnahmen die Rate der stillen Datenbeschädigungen um 34.84 Prozent.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Christian Dietrich)
Erlangen, 13. Oktober 2014

Contents

Abstract **ii**

Kurzfassung **iii**

1 Introduction **1**

2 Fundamentals **3**

2.1 The OSEK/VDX Operating System Standard 3
2.2 Atomic Basic Blocks: Abstracting Real-Time Systems 6
2.3 Summary 10

3 System Analysis **11**

3.1 The Global Control Flow Graph 11
3.2 Abstract Model of a RTOS 15
3.3 Construction of the *global control flow graph* (GCFG) 18
3.3.1 The CFG on the ABB level 18
3.3.2 The ICFG and its Passes 20
3.3.2.1 Interrupt Block State 23
3.3.2.2 Resource Occupation 24
3.3.2.3 Dynamic Priority 26
3.3.3 System-Semantic Simulation 26
3.3.3.1 Symbolic System Execution 31
3.3.3.2 System-State Flow Analysis 33
3.3.3.3 Combination of Results 35
3.4 Incorporating More Information 36
3.5 Summary 37

4 System Construction **39**

4.1 Tailoring System Calls 39
4.1.1 System-Call Specialization 40
4.1.2 Technical Implementation 42
4.2 Assertions on the System State 44
4.3 Control-Flow Monitoring 46
4.4 Summary 50

5 Evaluation **51**
- 5.1 Evaluation Scenarios 51
  - 5.1.1 *d*OSEK: A Generator for Dependable Operating Systems 51
  - 5.1.2 Applications 52
    - 5.1.2.1 *d*OSEK Test Suite 52
    - 5.1.2.2 *I4Copter*: A Realistic Work-Load Scenario 53
- 5.2 System Analysis 54
  - 5.2.1 Run Time of the System Analysis 54
  - 5.2.2 System-State Precision 57
  - 5.2.3 Comparing the GCFG 59
- 5.3 System Construction 60
  - 5.3.1 Code Size of the Kernel Fragments 61
  - 5.3.2 Run Time of the Kernel Fragments 63
- 5.4 Fault-Injection Campaign 67
  - 5.4.1 System and Fault Model 68
  - 5.4.2 Applying a Single Measure 69
  - 5.4.3 Composing Measures 72
- 5.5 Summary 76

6 Related Work **79**

7 Future Work **81**
- 7.1 Usage in Other Whole-System Analyzes 81
- 7.2 A Better Control-Flow Monitoring 81
- 7.3 Converting OSEK into a Finite State Automata 82
- 7.4 Limitations of the Approach 83

8 Conclusion **84**

Lists **85**
- List of Acronyms 85
- List of Figures 87
- List of Listings 88
- List of Tables 89
- References 94

Appendix A Fault-Injection Campaign **96**

Appendix B Reproducibility **99**

Chapter 1

Introduction

Embedding computer systems into almost every aspect of our life is one of the main characteristic of the technological advancement at the beginning of the 21st century. General-purpose computer systems are not only the main component of mobile systems, which get a lot of attention in the general public, but they also control most of our industrial production lines and our automotive vehicles. Here, special properties are required from the systems. They not only have to fulfill real-time requirements, but they should also fit onto small and cheap devices, while being resilient to unforeseen environmental influences. Therefore, we use specialized real-time operating systems to orchestrate different processes within a single real-time application.

These real-time operating systems are tightly coupled to the applications they control. The software for most real-time systems is shipped as a single software package that includes as well the operating system as the application. This tight coupling opens a possibility for the systems engineer: When the application on top of the operating system never changes, we can tailor the operation as closely to the application as possible. With this tailoring process, various non-functional properties as code size, run time, and memory consumption can be optimized to fit the bare minimum of the application's requirements. But for this tailoring process, detailed knowledge about the application has to be available.

In the automotive industry, a standard for operating systems emerged: the OSEK [37] standard. In the first place, OSEK is intended to ease the interaction between various component vendors in the automotive industry. But OSEK is also a standard for *static* operating systems. In static operating systems, all kernel objects (i.e., tasks, alarms, and interrupts) have to be declared beforehand in a system specification file. Therefore, the number and the static characteristics of these objects is known at compile time. The application knowledge described in the specification file can be used to drive an operating-system generator. It is best practice in the OSEK world to allocate and initialize the system objects statically into arrays and to link a generic OSEK implementation against it.

The application knowledge in the specification file is very coarse grained. Only the existence of tasks and some static configuration data, like static priority and preemptability of the task, are denoted there. Therefore, the tailoring process has to remain on that level of abstraction. Lohmann et al. [29] showed with their OSEK implementation CiAO the potential of static tailoring in the OSEK world due to manual configuration. They tailored the operating system as closely to the static requirements of the application as possible. The CiAO operating-system family exposes configuration switches for interrupt synchronization, memory protection, a highly modular IP stack [9], and many more. But all those switches turn on or off whole components or aspects of components for the whole application. So the tailoring is close to the application as a whole, but it is not close to distinct points within the application.

This thesis wants to extend the amount of in-depth application knowledge that can be used in the system-tailoring process. Not only the superficial application knowledge that is denoted in the specification file of an OSEK system should be used, but also the interaction between the application logic and the operating system is to be taken into account. This interaction is analyzed across kernel activation borders and the control-flow graph of all tasks in the system is combined in a *global control-flow graph*.

The in-depth application knowledge can be used to optimize the whole real-time system towards different non functional properties. One non functional property, which gained more and more attention in the recent years, is the resilience against soft errors. The developments in the fabrication of processors, like shrinking structure sizes and lower operating voltages, did not only enable the design of complex real-time systems with high computation requirements, but made the hardware more susceptible to transient hardware faults [13, 11]. Single bits in memory or the processor registers are more likely to be flipped by a cosmic particle or radiation. These soft errors can be mitigated by expensive hardware measures, like lock-step processors or ECC-checked memory [4, 55], but also software-based measures [10, 18] can be applied. With the in-depth application knowledge at hand, various software-based dependability aspects, which exploit the application's structure, can be introduced into the real-time system.

The focus of this thesis is the real-time operating system. Many software-based measures to mitigating the effect of soft-errors demand a reliable computing base [16] in form of an operating system that either detects a soft error or makes it a benign fault, but it should never fail silently. Since prior work showed that the static design of OSEK inherits a lower vulnerability towards soft-errors [20], the operating system standard is a good starting point for constructing this reliable computing base. With wise design decisions and arithmetic encoding, the *d*OSEK operating system [19, 31] already provides a four orders of magnitude lower silent-data-corruption rate than a off-the-shelve OSEK. But this resilience has a high cost in terms of code size and run-time overhead. But not only can the in-depth application knowledge be used to ease this overhead, but it can also be utilized to apply further software-based measures.

The rest of this thesis is structured as following: In Chapter 2 the fundamental methods and the prior work are presented; Chapter 3 explains the methods that are used for gathering the application knowledge in depth; Chapter 4 shows three different methods how the knowledge can be exploited to optimize different non functional properties of the system; The evaluation for different benchmarks scenarios is done in Chapter 5. The thesis closes with a wider overview of the related work (Chapter 6), a brief discussion of possible future work (Chapter 7), and the conclusion (Chapter 8).

Chapter 2

Fundamentals

In this chapter, I will present the essential concepts and standards for this thesis. They are well-known from the literature, and I will only discuss them as detailed as it is necessary to understand this work. For a more elaborated discussion on those topics please consult the referenced publications. First, the OSEK operating system standard is revisited. OSEK defines the construction rules for the real-time systems, that are analyzed in this work. Afterwards I will reexamine the concepts used by Scheler [44] to abstract and manipulate the real-time system architecture. Especially the notion of *control flow graphs* (CFGs) and *atomic basic blocks* (ABBs) are of special interest for this work.

2.1 The OSEK/VDX Operating System Standard

The OSEK/VDX[1] operating system specification was developed by the automotive industry, aiming for an “open-ended architecture” for control units in vehicles [37, p. 2]. This open-ended architecture allows the easy integration of components built by different vendors into one control unit. The OSEK specification defines not only an API (OSEK-OS) for a *single-core real-time operating system* (RTOS), but also a inter-process communication interface (OSEK-COM [39]) and network management for a distributed system (OSEK-NM [40]). In addition to the event-triggered OSEK-OS, also an additional specification for a time-triggered real-time operating-system, that runs on top of OSEK-OS is defined (OSEK-TIME [41]). For this work, only the basic operating system specification OSEK-OS is of interest, and I will refer to it from now on only as OSEK.

The main concept of OSEK is the notion of a task. A task is one flow of execution that is managed by the operating system. At various points the operating system can preempt the current task execution and start or resume another task flow. Since OSEK is a *real-time operating system* (RTOS), more stringent rules for tasks are applied to get deterministic and predictable behavior. Each task has a static priority and a defined entry point, which cannot be changed at run time. Tasks are in one of three states: `RUNNING`, `READY` or `SUSPENDED`.

The transition model for task states is depicted in Figure 2.1. A task is activated by an external event or by another task with an explicit system call. It is started and preempted by the system scheduler and cannot be terminated by any other entity than itself. The scheduling rules for starting and terminating tasks are very strict. The operating system ensures that the

---

[1] OSEK - “‘Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug”’ (open systems and their interfaces for electronic in automobiles)

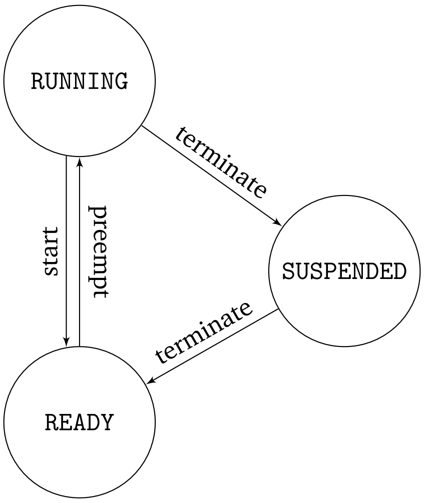


Figure 2.1 – OSEK task state diagram ([37, Fig. 4.3])

highest-priority task, which is `READY`, is always started or resumed. When another task with a higher priority becomes `READY`, the current the current (low-priority) task is preempted and will be resumed when the higher priority task terminates. In short, at any given time the task with the highest priority that can run, will run.

The external activation of tasks can be done by periodic or by non-periodic events; see Chapter 6 and 7 in Liu [27]. Periodic events are implemented by alarms and counters. Counters are represented by a counter value that is incremented by the hardware and wraps around at a preconfigured value. Each alarm is “wired” to one of these counters. Alarms are either in one-shot mode or in recurring mode. When the counter reaches the alarm’s activation value, the is triggers and activates the preconfigured alarm-action. Besides low-latency alarm-callbacks, the activation of a task is a possible action.

OSEK handles non-periodic events [27, p. 40] in terms of *interrupt requests* (IRQs) that trigger a user provided *interrupt service routine* (ISR). There are two different kinds of ISRs: ISR1s may not use system calls and are very low-latency, since they do not have to be synchronized with the operating system. ISR2s are synchronized with the kernel and can therefore use a limited set of system calls. For example, an ISR2 can activate another task by using a system call. It is up to the operating system developer, whether ISR2s can be preempted (like tasks) or be re-interrupted by another ISR. As ISR1s do not interact with the operating system, they are of no further interest for the system analysis undertaken here and the term ISR refers always to ISR2s.

Critical sections, which can be entered by different tasks, are managed with OSEK resources. An OSEK resource has two states: it is either taken or free. If a task wants to enter the critical section, it takes the resource. After the critical section, the task gives the resource back and the resource is free again. When using critical regions in real-time systems, uncontrolled priority inversions have to be prevented. Image a situation where a low-priority task is in a critical region that is also used by a high-priority task. If the low-priority task is now preempted by a mid-priority task, the resource cannot be released. If the high-priority task preempts the mid-priority task and requests the resource, the operating system cannot fulfill the request since the resource is already occupied by the preempted low-priority task. The described situation may lead to deadline-misses or dead-lock situations, like it happened on the Pathfinder Mars Mission [23, 53].

In order to prevent priority inversion, OSEK implements a stack-based priority ceiling protocol [45]: Each task has a dynamic priority, which is initialized with the static priority. When a task acquires a resource, the its dynamic priority is increased immediately to the ceiling priority of the resource. The ceiling priority of a resource is the highest static resource of all tasks that are allowed to acquire the resource. The priority ceiling protocol prevents a mid-priority task from preempting the task within the critical region, since the ceiling priority of the resource is the static priority of the high-priority task. When a task releases a resource, the dynamic priority drops to the highest ceiling priority of all resources occupied by the task or, if no resource is acquired, to the task's static priority. With the *priority ceiling protocol* (PCP), deadlocks and uncontrolled priority inversion are prevented, which is highly desirable for real-time systems.

Producer-Consumer synchronization is done with events. For events the additional task state `WAITING` is introduced. A task can wait for an event and transits into the waiting state. Waiting tasks are not considered during scheduling, even when they have the highest priority. Every other task (or an alarm) can release the event, which brings the waiting task back to the `READY` state. The rescheduling takes places immediately, after the event was released.

```
TASK TaskA {
  PRIORITY = 2;
  AUTOSTART = TRUE;
  RESOURCE = resource2;
}
```

Listing 1 – Example for an OIL declaration. TaskA is declared with static priority 2, it is automatically set to `READY` when the operating system boots and can obtain the `resource2`

OSEK is designed with resource efficiency in mind. Therefore, it has some limitations unknown from general-purpose operating systems. The system is preconfigured before the actual run-time. In a domain specific language, the *OSEK implementation language* (OIL) [38], the developer declares the system objects statically. The number of tasks and their static priorities are declared as well as alarms, the alarm-actions, and ISRs. This system description allows static allocation of system objects and their static configuration. It avoids dynamic memory management within the operating system, and therefore increases the predictability of the system. Listing 1 shows an excerpt of a system configuration written in OIL. `TaskA` is declared with a static priority of 2. The declared task starts automatically, when the system is booted and `TaskA` is allowed to acquire `resource2`. From this snippet we know that `resource2` has at least a ceiling priority of 2.

Table 2.1 is an incomplete overview of the OSEK system calls, which the specification also calls system services. System calls either get system objects, like task and alarm identifiers, or integers, like counter values, as arguments.

OSEK is used on a wide range of target systems of various sizes. Hence not all features are used by every application, OSEK defines four conformance classes. The basic conformance class 1 (BCC1) defines the most fundamental class. With BCC1 each priority can only contain one task, every task can be activated only once and no events may be used (no `WAITING` state). This conformance class, with some extensions regarding to resource usage, is the base for this thesis. I aim for this conformance class, since it already allows the design of complex systems and contains most of the problems to be discussed. At some point, hints are given how extend this work to more advanced conformance classes.

System Call	Arguments	Brief Description
`ActivateTask`	`TaskID`	The task `TaskID` is transferred from the `SUSPENDED` state into the `READY` state
`TerminateTask`	–	This service causes the termination of the calling task. It is transferred to the `SUSPENDED` state.
`ChainTask`	`TaskID`	Terminates the calling task, while it atomically activates the task `TaskID`.
`GetResource`	`ResID`	Acquires the resource `ResID`, so that it enters the associated critical region.
`ReleaseResource`	`ResID`	Leaves the critical region associated with the resource `ResID`. The resource is free afterwards.
`SetRelAlarm`	`AlarmID, inc, cyc`	Winds up an alarm. After `inc` ticks have elapsed the alarm action is triggered. If `cyc` is not zero, reoccurs afterwards every cycle ticks (recurring mode), otherwise the alarm is only in single-mode.
`CancelAlarm`	AlarmID	Cancels the activated alarm AlarmID.
`EnableAllInterrupts`	–	Disables all interrupts for which the hardware supports disabling.
`EnableAllInterrupts`	–	Restores the regcognition status of all interrupts that were disabled by `SuspendAllInterrupts`.

Table 2.1 – OSEK System Call Overview. The system calls (system services) are described in [37, Chapter 13].

2.2 Atomic Basic Blocks: Abstracting Real-Time Systems

In his dissertation, Scheler [44] describes real-time systems at an abstract level. A real-time system is closely connected to its environment. This connection is expressed in terms of measuring the physical world, computing values and using the calculated values to manipulate the physical world. The notion of time and a timely behavior is of distinguished importance for a real-time system. This interleaved connection of the real-time system and the physical world calls Scheler the “external-view of a real-time system” [44, p. 21]. This external view is expressed in terms of events, tasks and deadlines and has to be implemented by the real-time engineer. Scheler calls this technical implementation, which fits the external view, the *internal-view of a real-time system* [44, p. 23].

The internal-view is mainly defined by the *real-time system-architecture* (RTSA). In industry two main RTSAs developed [44, Section 3.4.1][27, p. 60f]: *event-triggered real-time systems* and *time-triggered real-time systems*. Event-triggered systems release jobs upon external events, which can fire at any given time. The real-time operating system’s job is to execute jobs in such a way, that they meet their deadlines. Whether the jobs meet their deadlines in any given situation, has to be proven by a schedulability analysis. The dynamic nature of event-triggered real-time systems is more intuitive to developers; therefore these systems are considered easier in their construction. But their timely behavior is harder to prove [27, p. 72].

The other RTSA is the time-triggered architecture. In a time-triggered system, every action is controlled by a timetable. The entries in the timetable are executed sequentially with a fixed time delay. A hardware timer is used to trigger the next action the timetable defines. If the last entry is finished, the execution cycle starts again at the first table entry. Therefore these systems are strictly periodic. This strict periodicality make time-triggered systems easier to verify, but the construction of timing tables is a difficult and error-prone task, when done manually.

Therefore, Scheler identifies common structures, used in both paradigms, and aims for an automatic translation of the RTSA. This allows the real-time engineer to write an event-triggered system and translate it automatically to a time-triggered system. The main abstraction Scheler uses is the *atomic basic block dependency graph*.

Scheler borrows concepts that are commonly used in compiler technology. Therefore it is useful to take first a look at these concepts and to investigate on their descriptive strengths for the RTSA afterwards. Compilers use the representation of a program as *control flow graph* (CFG) consisting of *basic blocks* (BBs). Aho, Sethi, and Ullman [1, p. 528] define a basic block like this:

Definition 1 (Basic Block)**.** *A* basic block *is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.*

Basic blocks are identified by their *leaders*. For a function, statements are marked as leaders, if they follow a branch statement or are target of a branch statement. The first statement of a function is always a leader. A BB consists of a leader and all following non-leader statements up to, but not including, the next leader or the end of the function [1, p. 529].

Within a basic block, the flow of control progresses linearly. The basic blocks can be connected in a directed graph. There, edges are drawn from a source block to a target block, iff the execution flow *can* proceed from the last instruction of the source block to the first instruction of the target block. This directed graph is called the *control flow graph* (CFG) and it subsumes all possible execution paths. Normally, the CFGs are constructed for single functions, therefore they have an *entry basic block* and an *exit basic block*. Whether function call statements, which transfer the control to the CFG of another function, are leader statements or not is implementation specific.

For global (whole-program) optimizations an *inter-procedural control flow graph* (ICFG) is constructed. The CFGs of all functions in a program are mangled into one directed graph. A basic block containing a function call is connected to the entry block of the called function. From the exit block of the called function an edge is drawn to the basic block containing the statement following the original call statement. This may be the same block as the calling basic blocks or one of its direct successors [46].

Scheler identifies the concept of *atomic basic blocks* (ABBs) as an abstraction for the RTSA. ABBs are similar to basic blocks, but have a whole-system view. In the internal-view, tasks are *dependent* on each other. Dependencies can be either directed or undirected. Directed dependencies model that some action has the be completed before another one can start, for example one task sets another task ready. Undirected dependencies express that two actions exclude each other on the timeline: *critical sections*. To gain a fine granularity for these dependencies, the system is divided into ABBs, which are connected in an ABB dependency graph. Scheler uses the following rules to identify ABBs [44, p. 45]:

1. ABBs include one or more basic blocks of a function, that are a connected component of the function's CFG. An ABB is a *control flow region*.
2. Every ABB has one definite entry basic block; the ABB can only be entered via this block. Analogues, every ABB has one definite exit basic block; the ABB can only be left via this block. An ABB is a *single-entry single-exit region*.
3. ABBs reach from the end of the last ABB to the next ABB endpoint. Endpoints are sources or targets of system dependencies, for example system calls.
4. If an ABB endpoint is contained within a basic block, then the block is split up at the ABB-endpoint.

The main drawback of the dependency graph, discussed by Scheler, is that its construction is flow-insensitive. The fact that the application logic may prohibit a dependency at one point is not taken into account. For a more detailed discussion of ABBs and the ABB dependency graph please refer to Scheler [44]. To give a practical example, I will briefly sketch the ABB and graph construction phases for an event-triggered real-time system, implemented with OSEK:

```
TASK(SerialByte) {
   unsigned char received = getByte();
   message_add(received);

   if (message_isComplete()) {
      ActivateTask(MsgHandler):
   }
   TerminateTask();
}

TASK(MsgHandler) {
   message_process();
   TerminateTask();
}
```

Listing 2 – Example OSEK application. The `SerialByte` task receives a byte and adds it to a message buffer. If the message is complete, the message processing task `MsgHandler` is activated.

Listing 2 is the source code listing of an OSEK system. It resembles a simple serial-message receiver. The `SerialByte` task is activated (not shown) when a new byte is received. The task adds the byte to a message handling subsystem. If this subsystem signals that a message is complete, another task (`MsgHandler`) is activated. The `MsgHandler` task processes the message and terminates itself afterwards.

The ABB dependency graph is constructed from the task's CFGs (Figure 2.2a). The `ActivateTask` system call, which is an ABB endpoint, is located within BB2. As the fourth rule of ABB construction implies, this basic block is split into two basic blocks. Each basic block is then wrapped into an ABB (Figure 2.2b). System calls insert inter-task dependencies: ABB4 can only be executed when `MsgHandler` is activated, therefore an dependency from ABB2a is drawn (Figure 2.2c). The dependency graph can be used to translate the event-triggered real-time system into a time-triggered one.

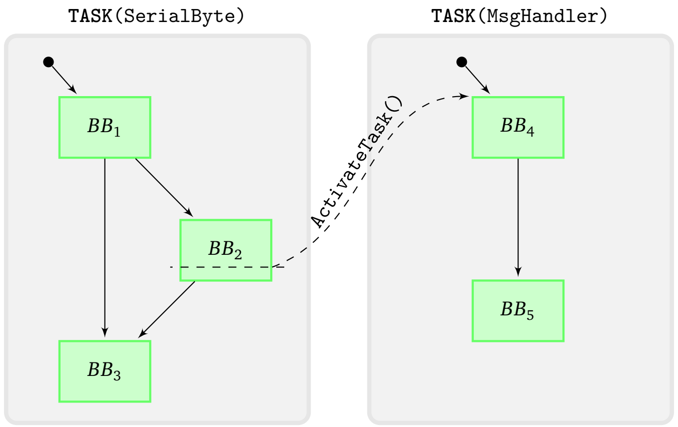
Another important part of this translation process is the system model introduced and used by Scheler. The system model is a description of the external-view of the real-time system. The real-time system engineer annotates events, deadlines and tasks. Scheler distinguishes between tasks, that are part of the external-view, and subtasks, which are the implementation detail of the internal-view.

Definition 2 (Task). *In a real-time system, tasks contain all forms of actions to be done, when an event occurs. Tasks are part of the external-view of a real-time system.*

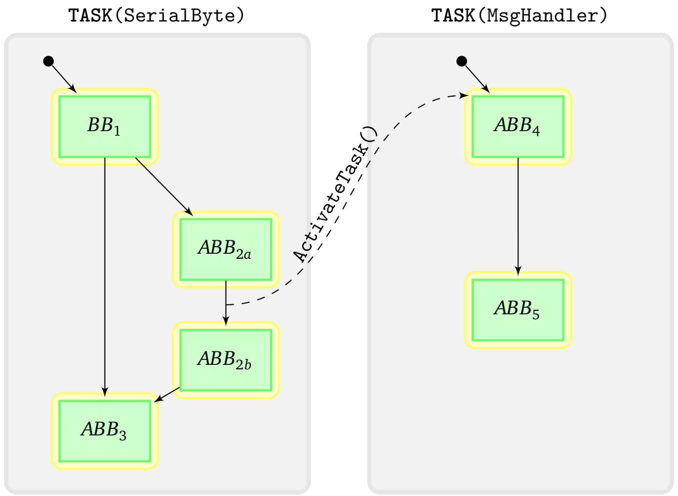
Each task is connected to an event, which releases it and issues a new job. The implementation then arranges subtasks to achieve the job in the given time.

Definition 3 (Subtask). *Subtasks incorporates the actual event handling. They are part of the internal-view of a real-time system. Each task contains one or more subtasks and each task has at least one root-subtask that is started when the event occurs.*

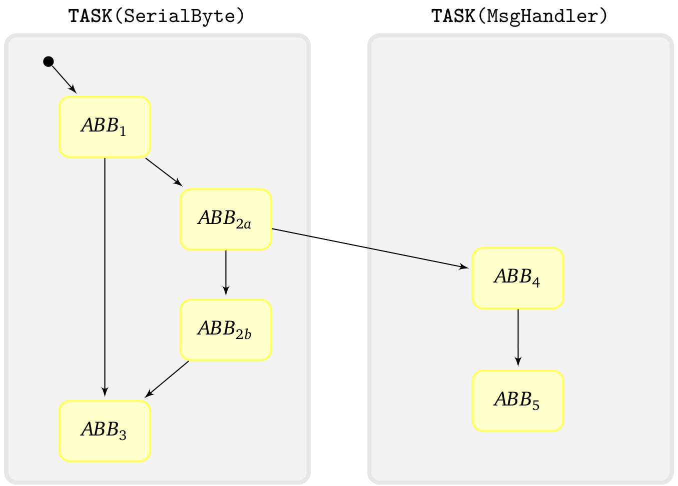
OSEK does not use this terminology, but a different one. The OSEK term "task" maps to the notion of a subtask, since it contains the actual event handling code. The notion of tasks is not



(a) Control flow graphs for the system from Listing 2. The dashed line is the logical result of the `ActivateTask` system call.



(b) The BB with the system call is splitted into two ABBs. All other BBs are converted only to one ABB.



(c) Directed edges in the resulting dependency graph express the ABB execution order.

Figure 2.2 – Construction of the ABB dependency graph.

present in the OSEK specification. In order to not confuse you, I will use Scheler's terminology consistently for the rest of this work.

One important constraint in Scheler's work, which is not explicitly mentioned, is that all subtasks within a task must finish their execution, before the task's event triggers again. This limitation is tightly coupled to the system model, since there each event is assigned a minimal inter-arrival time and each task is annotated with a deadline. Scheler's dissertation also contains a more detailed discussion of the system model [44, p. 50].

2.3 Summary

The OSEK operating system standard is used to build static real-time operating systems. Many different system objects are described in the standard. Subtask have a static priority, but can change their dynamic priority at run-time by acquiring a resource. Using PCP, resources implement critical sections without uncontrolled priority inversion. The scheduling is strictly priority driven and rescheduling takes place immediately. Subtasks can be activated by other subtasks or *interrupt service routines* (ISRs) with a system call. Additionally, alarms can be winded up and activate a subtask on expiration. A consumer-producer situation can be implemented with events, which are sent by a subtask, while another subtask waits on the event. All system objects have to be declared in the OIL description at compile time, and can therefore be allocated statically by an operating system generator.

Atomic basic blocks, presented by Scheler [44], abstract the real-time system architecture and are used to translate an event-triggered real-time system into an time-triggered one. ABBs are defined similar to basic blocks, which are well known from compiler technology. ABBs are a fine-grained division of the subtask's *control flow graph* (CFG) and contain one or more basic blocks. The ABBs can be connected in a dependency graph, that expresses forced execution order and critical sections. The external-view of the real-time system is annotated, in terms of tasks, events and deadlines, by the real-time engineer in the *system model*.

Chapter 3

System Analysis

In Chapter 2, the OSEK operating system standard and the notion of ABBs were discussed. In this chapter, I will give the methods used to analyze a real-time system, which consists of the real-time operating system and the real-time application. The methods will be applied to applications that conform to the OSEK standard. The OSEK standard is well-defined, easy to comprehend, and has a strict scheduling semantic. Additionally is operating system interface quite small compared to other standardized operating systems, like POSIX [42]. Therefore, it is a good choice for analyzing the interaction between operating system and application.

The cross-kernel analysis takes the application behavior, in terms of application logic, as well as the operating-system semantic into account. As a result, we get the *global control flow graph* (GCFG) that describes the interaction at the operating-system interface in a flow-sensitive fashion. First, I will present the semantic of the GCFG and what considerations led to its definition. Afterwards, a model for describing the data structures and operations to represent the inner state of an OSEK system are discussed. With this model at hand, I will present a set of transformations that gather static application knowledge that is coupled to the ICFG. This static knowledge is used by two alternative algorithms, which differ in complexity and preciseness, to construct the GCFG. As a closing, additional information supplied by the application developer is integrated into the analysis, so that the graph becomes more dense and more precise.

3.1 The Global Control Flow Graph

First I want differentiate the dependency graph that is used by Scheler [44, p. 55], and the venture that is carried out in this thesis: The ABB dependency graph is explicitly flow-insensitive. In its construction process no information about the context of an ABB is used. The flow information is not utilized to avoid unnecessary dependencies and its integration is marked as a topic of further research. For example, the dependency graph contains two edges when an ABB activates another subtask. As well the other subtask's entry ABB as the successor of the activating ABB are dependent in their execution upon the calling ABB. In the GCFG, the execution can only proceed in one of those blocks, since priority-driven scheduling is applied. The dependency graph also contains no information where the execution should proceed after a subtask has terminated; it does not include the interaction of tasks with ISRs. The GCFG, since it is a *flow graph*, does explicitly include this information. Nevertheless, I do not aim for an better dependency graph, but for a flow-sensitive description of the application's interaction with the operating system.

Nevertheless, the abstraction in terms of ABBs has proven to be the granularity of choice for reasoning about a the inner structure of real-time systems.

The usage of control flow graphs in optimizing compilers is due to Frances E. Allen [3][2]. On a semantic level, control flow graphs express an execution order on the BBs. If a BB is the predecessor of another BB, it is executed first and the exection flow continues in a successor. We can abstract this observation in order to define the term *control flow graph*:

Definition 4 (Control flow graph). *A control flow graph is a directed graph of* work packages *as nodes with an unique root node. A execution path is a sequence of work packages starting at the root node. The graph is the combination of all possible work-package execution paths.*

These “work packages” are executed on a *virtual machine* (VM). I will use the term VM not in the sense of full- or para-virtualized machines, but in a more abstract manner. Tanenbaum [48] uses the term of a *multi-level machine* to describe a complex programming environment. This multi-level machine is a hierarchy of stacked VMs that increasingly boost the abstraction level. Each VM has a set of instructions as programming interface. It utilizes the programming interface of the underlying VM to provide its own interface. Each VM is allowed to propagate parts of the underlying interface as well as construct complex operations, which are made up from lower level instructions. If an application developer codes a program for a high-level VM, the abstraction is continuously lowered throughout the VM stack, while the program semantic is kept until a “real” VM can execute the program.

The C programming language provides such a VM. Statements and expressions are its programming interface. This programming language VM is lowered by a compiler to the underlying VM, which is provided by the operating system. This OS-VM executes a machine program enriched by OS-specific instructions: the system calls. One example of an OS-VM is “Linux/IA-32”. Most of the instructions are directly interpreted by the underlying hardware (IA-32). But when the execution comes to a special machine instruction (`int 80`), the OS (Linux) is activated and partially interprets the system call. Linux itself runs on the bare-metal VM, defined by the IA-32 instruction set architecture and lowers the abstraction of the executed program on-the-fly to the real hardware, in terms of pipeline operations and switching of transistors.

For the control flow graph, the executing VM decides, based on its internal state, after each work package, which successor of the work package should be executed next. With this in mind, I will describe a hierarchy of control-flow graphs and their VMs. These VMs are used to describe the possible dynamic behavior of the real-time system on several levels:

The “function VM” and the CFG A function, like it is defined in the C programming language and preserved during the compilation process, executes in its own VM. Each incarnation of the function starts a new VM with new “virtual registers” (local variables) and executes the instructions of the function in one of many strictly defined sequences. The control flow graph of this VM subsumes all sequences of executed instructions. Therefore we can use single instructions, which are also called *minimal basic blocks*, as work packages. Nevertheless, it is more common and more appropriate to use the more coarse-grained normal/maximal basic blocks as work packages. This function-local graph of basic blocks is the most common control-flow graph and is therefore always abbreviated with CFG.

---

[2] Frances E. Allen was also the first woman awarded with the Turing Award in 2006.

The "subtask VM" and the ICFG Each subtask is executed in its own VM. A time-sharing OS virtualizes the real processor and provides each subtask with an "virtual processor". Each subtask has its own set of hardware registers and its own instruction pointer. This subtask VM is located above the function VMs in the hierarchy. Most of the instructions, which are executed within the subtask VM, are propagated to the currently running function VM. Nevertheless, when the subtask VM executes a `call` or `return` instruction, the current function VM is exchanged. The flow of control is transferred from one function VM to another one. The execution sequences of the subtask VM is the combination of many interleaved function VM sequences. The subtask VM is the execution engine for the *inter-procedural control flow graph* (ICFG).

The "system VM" and the GCFG Above the subtask VMs, the system VM is located. It is defined by the operating system semantic and switches back and forth between the different subtask VMs. The system VM distinguishes between normal basic blocks that only contain computations and basic blocks that contain (only) system calls. The normal basic blocks (computation blocks) are propagated to the current subtask VM and the system VM does not interfere. But when a system call block should be executed, the system VM interprets it by itself. It manipulates its internal state according to the system-call semantic and may switch the current subtask VM in favor of another one. Therefore the execution sequence of the system VM is the combination of the execution sequences of many subtask VMs. The *global control flow graph* (GCFG) connects all basic blocks of the whole system to express all possible execution paths of the system VM.

Since ABBs enclose many basic blocks, but are a single-entry/single-exit region of the control flow graph, we can use them as a drop in replacement for basic blocks in the definition. Even whole function call hierarchies that contain no system call can be subsumed by a single ABB. This partition of the flow graph reduces the number of nodes. Not only the analyses' run time benefit from this reduction of nodes, but it also sharpens the focus on the system behavior, since unrelated application structures are hidden. From the ABB definition, we know that each ABB contains at most one endpoint, in our case system calls. Therefore ABBs can be used for all three control flow graph types as work package units. The trace of the system VM is a sequence of ABBs.

Definition 5 (Global control-flow graph). *The global control-flow graph of a system is a control-flow graph. The work packages are atomic basic blocks that either contain normal computations or a single system call. The execution paths subsumed in the graph express all possible execution sequences of ABBs on the system VM.*

For the example presented earlier, Figure 3.1 shows the execution sequences on different hierarchy levels. Each row represents either a function VM, a subtask VM, or the system VM. The time progresses with each column and an ABB execution can span more than one time step in different VMs, since an ABB may subsume a whole function-call hierarchy.

There are a few interesting points in this sequence: At point ①, an interrupt occurs and activates the system VM. The interrupt releases a `SerialByte` job and the corresponding subtask VM starts execution with ABB_1. The subtask VM partial interprets ABB_1 by handing the control over to the function VM of the main subtask handler. Subsequently the subtask VM executes different function VMs to achieve the work defined in ABB_1. At point ②, the `Handler()` function VM issues a `TerminateTask()` system call in order finish its execution. The subtask VM cannot handle this "instruction" and gives the control back to the system VM.

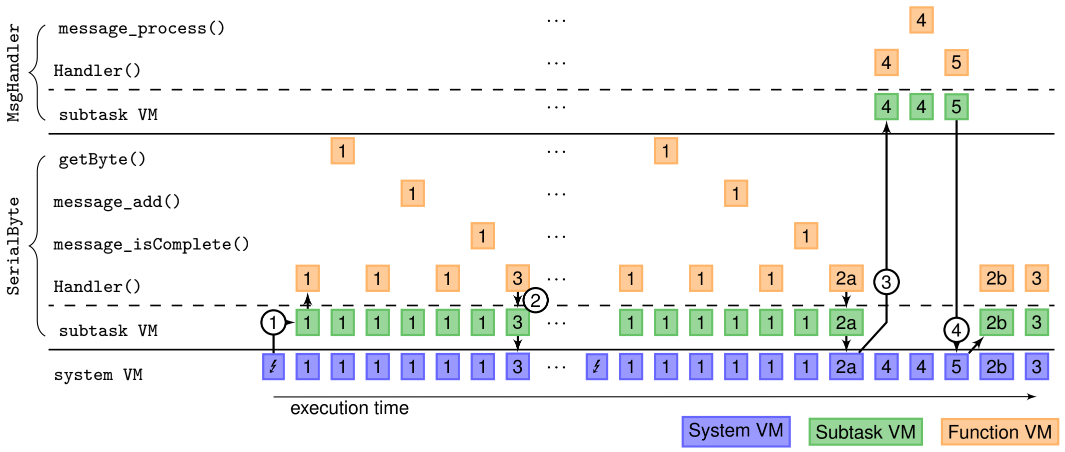
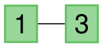
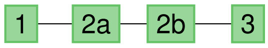


Figure 3.1 – Execution sequence for Listing 2. The different ABBs and their execution in the different layers of virtual machines.

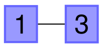
This sequence repeats itself for every received byte until the message is complete. The sequence for a completed message is shown in the right half of the diagram. Until message is complete, the execution sequence is the same for each received byte. When the branch indicates a completed message, the function VM decides to execute ABB2a (③). The `ActivateTask()` in ABB_{2a} cannot be handled by the function VM and the subtask VM, and is therefore executed on the system VM. As a result of the system call, `MsgHandler` is activated and immediately scheduled, since it has a higher priority. At point ④, the `MsgHandler` subtask VM terminates and the `SerialByte` VM is resumed.

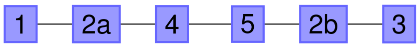
For a received byte, we observe two different possible execution sequences in the subtask level for the `SerialByte` subtask VM:





On the system level, we get these execution sequences, while the second one is interleaved with the exection of the `MsgHandler` subtask:





From these execution sequences, we can construct the GCFG for this system. In Figure 3.2 both the ICFG and the GCFG are depicted. At ABB_{2a}, the ICFG proceeds to ABB_{2b}, since a system call *looks* synchronous to the subtask VM. But, as the GCFG reveals, the system call results in a preemption in favor of `MsgHandler`.

This GCFG is only a partial one, since the activation of `SerialByte` is not depicted. In a whole system GCFG, the root node is the starting point of the operating system. In case of OSEK this is the call to the special system call `StartOS()`.

Another important aspect for the GCFG semantic is the handling of interrupts. Interrupts can occur at all times and cause a system-state manipulation through alarms and ISRs. They result in edges within the GCFG, that cannot be extracted from the application's CFG and ICFG structure. In essence, a asynchronous activation results in a jump to the entry ABB of an interrupt handler. These interrupt ABBs can either be fully included the GCFG or treated with a special semantic. For this thesis, these blocks will be treated specially and excluded from the GCFG (but not from the analysis). As system calls within ISRs manipulate the OS state synchronously,

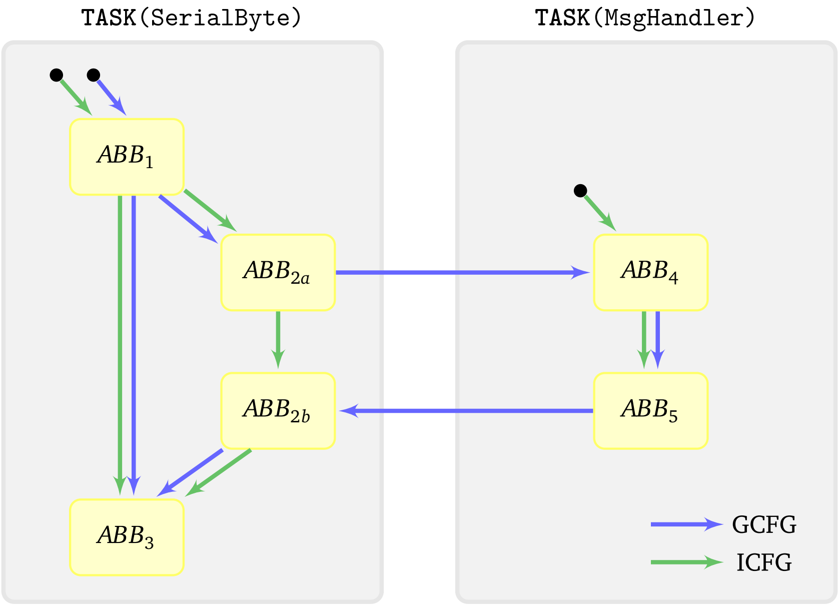


Figure 3.2 – ICFG/GCFG for OSEK Example Application. The flow graphs for Listing 2 express the execution order of all ABBs in the system on different levels.

asynchronous events can be summarized as transactions on the system state. This leads me to the first limitation to put on the system semantic:

Limitation 1. *System calls, interrupt service routines, and alarm activations cannot be interrupted. They are transactions on the OS state and execute atomically.*

The handling of ISRs and their interruptability is not defined by the OSEK standard and is therefore implementation specific. For the OS, this limitation does not force necessarily a global interrupt block, but it demands an atomic system call semantic. For ISRs this limitation forces an interrupt lock during the ISR execution, which will increase the interrupt-service latency.

As a result of this limitation, interrupts can only occur in computation ABBs that are not executed within an interrupt context. Since each external event is treated as an asynchronous system call, this event results in an additional edge to an activated or resumed subtask.

Without this limitation, all computation blocks would have an edge to all ISR entry blocks, and all ISR exit block would have an edge to all computation blocks. For doing useful data-flow analyses on the whole system, the resulting GCFG would be useless.

3.2 Abstract Model of a RTOS

After each ABB, the system VM decides upon its internal state where the execution should continue. The successors in the GCFG of an ABB are candidates for this decision. The system VM decides which successor is to be selected. Therefore we have to determine the structure of the internal state that is used for selecting the next ABB. In the implementation, this internal state resides in the OS memory regions: the OS state. The application may not manipulate the OS state directly, but only through explicit system calls. This can either be enforced by convention or, like in desktop systems, by memory protection. The implementation's OS state is represented in this thesis by an abstract system state. This system state is not dependent on the implementation but on the system specification.

`Function :: Arguments → Results`

In order to give describe the system state, I will give you the function signatures of accessor methods that operate on the state. I will use a notation that is similar to the one used by the Haskell programming language. Words with capitial letters are scalar data types. Square brackets indicate a list of a certain subtype. For example, `[Subtask]` indicates a list of subtask objects. For the OSEK specification, the following system state accessors are used to describe the dynamic state of an operating system:

- running_subtask `:: State → Subtask`
  In every system state, one subtask is the currently running subtask. All interrupt service routines and alarms are treated as subtasks for the system state.
- continuations `:: (State, Subtask) → ABB`
  Each subtask has exactly one ABB that indicates which ABB is to be executed next. When a subtask is suspended, the entry ABB of the subtask handler is to be executed next. For preempted subtasks, this is the ABB to be executed after the subtask is resumed again.
- subtask_state `:: (State, Subtask) → READY || SUSPENDED`
  Each subtask is either `READY` or `SUSPENDED`. The `RUNNING` state is omitted, since running_subtask indicates the currently running subtask (VM), which must also be `READY`. In the implementation, this part of the system state is usually implemented as the ready list.
- priority `:: (State, Subtask) → Integer`
  Each subtask has a dynamic priority in each state. The scheduler selects from all `READY` subtasks the one with the highest priority. The static priorities, which are the base for the dynamic priorities, are defined in the OIL file, but can be remapped by the system generator as long as the original priority order is preserved.
- isr_block `:: State → Boolean`
  The application may block interrupts in OSEK, but there are many restrictions on the usage of explicit interrupt control. In a later section, I will examine the interrupt block semantic more closely.
- resource_taken `:: (State, Resource) → Boolean`
  The application coordinates critical regions with resources. The occupation state of a resource is part of the system state.
- running_abb `:: State → ABB`
  This system state accessor is only derived from using running_subtask and continuations, and is only a convenient shortcut. Since, for each system state, it is unambiguously which ABB is currently executed, this accessor is well-defined:

  running_abb(State) = continuations`(State,` running_subtask`(State))`

Each system state is coupled to a point in time. As a convention, the system states in this thesis always represent the situation *before* an ABB is executed (see Figure 3.3). Each ABB does a transformation (`ABB_transformation()`) on the system state and the system VM decides afterwards upon the output state, which ABBs can be executed next. The ABB transformation is determined by the ABB type. There are several classes of ABB types:

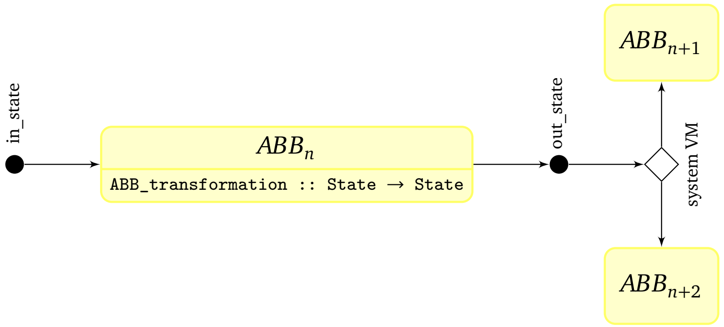


Figure 3.3 – ABB_transform on the system state. A system state represents always the OS state *before* an ABB is executed. The ABB transforms the input state and the system VM decides upon the output which ABBs is executed next.

computation Most of the application is hidden within computation blocks. Computation blocks can subsume complex algorithms, whole function-call hierarchies or only simple calculations, as long as the enclosed functionality does not interfere with the system state. A computation-block transformation does only change the continuations for the currently running subtask. In computation blocks interrupts can occur, if not blocked explicitly.

system calls This ABB type class contains several types that are defined by the system specification. System calls are explicitly called by the application developer. The state-transformation semantic of system calls is defined by the system specification. Examples for system calls in OSEK are `ActivateTask()` or `TerminateTask()`

artificial system calls For modeling implicit system behavior some artificial system calls are introduced. They can be part of the GCFG, but do not necessarily have a corresponding piece of code in the application. Examples for artificial system calls are the `IdleLoop`, the `iret` at the end of interrupt handlers, or a subtask's `kickoff` call.

Artificial system calls are one part of the strategy to express system semantic within the GCFG. Another is the unification of alarms and ISRs. Since alarms can trigger asynchronously like ISRs, artificial ISRs are used to model the effect of an expired alarm. Alarms that activate a subtask are modeled as an ISR with an enclosed `ActivateTask()` system call. Since ISRs also are activated at some point, execute a few ABBs and terminate with an `iret`, they are also unified with subtasks. ISRs (and alarms) are treated as subtasks with a high priority that cannot be preempted by other subtasks (Limitation 1). With these semantic-maintaining unifications, we have a system where all execution flows are expressed as single subtask VMs.

Until now, the described system state is a *precise* one. But during the analyses of a system, it is not possible to have a precise picture of the system state at all points. For example when a subtask is activated only in one branch of a conditional path, we do not know whether the subtask is `READY` or `SUSPENDED` afterwards.

Definition 6 (Fuzzy System State). *A system state is fuzzy, if one or more system-state accessors cannot return an unambiguous answer.*

Fuzzy system states are the key to express partial knowledge about the system at one point. For most applications, we will not have perfect knowledge for all states at all points. But fuzzy system states allow us to express the partial knowledge we have. Fuzziness is especially useful for the ready list and for next ABB to be executed. Therefore, I extend the state-accessor methods to reflect this fuzziness:

- subtask_state `:: (State, Subtask)` → `READY` || `SUSPENDED` || `FUZZY`
  If the activation state of a subtask is not known for sure, it is `FUZZY`. So when it comes to scheduling, we have to assume, that the subtask *may* be `READY`.

- continuations `:: (State, Subtask)` → `[ABB]`
  The continuations accessor does not only return a single ABB, but a list of possible successors. The subtask may continue at any of the returned ABBs.

The system state, as the second central data structure besides the GCFG, may be extended for various analyses with different data fields, depending on the analysis' focus. One extension that is used later on, is an artificial call stack that collects the function call history.

The target of the system analysis is not only the construction of the GCFG, but also the extraction of a system state for every ABB. Both results are tightly coupled and have an overlapping semantic. Edges in the GCFG express decisions the system VM might make, while the continuations are the possible decisions to choose from. In general, the less a system state is fuzzy, the less edges are present in the GCFG.

3.3 Construction of the GCFG

Modern optimizing compilers are structured into passes to cope with the complexity of the compilation process. Each pass transforms the program or provides information. A pass depends on other passes and may destroy information, which might be recalculated, if needed at some later point. The following sections describe the passes on the ABBs and the CFG that are needed in order to construct the GCFG and to gather the system states for each ABB.

We are starting with the function-local CFGs of the application and connect them to an ICFG. On the ICFG, we use different data-flow analyses to obtain static application knowledge. This static knowledge is used to construct the GCFG.

3.3.1 The CFG on the ABB level

The system analysis is carried out for a specific application. Therefore, the application has to be available at the moment of the analysis. The source code is compiled to the immediate representation of the used compiler or directly to machine code. From the compiled application the CFG consisting of normal basic blocks is extracted for every function.

All functions and system calls are individualized by splitting basic blocks before and after the call (see Figure 3.4). Afterwards, a basic block does contain either only a call instruction or only other instructions. This ensures that every basic block has one distinct type; it is either a call block or a non-call block. All system calls are contained in their own call-block. For those system call blocks, we have to ensure, that they are surrounded by non-system-call blocks. This is crucial to model the system behavior correctly: With a computation block in place, the application can be interrupted between two directly-following system calls. If a system call block does not meet this requirement, it is surrounded by empty artificial computation blocks.

Additionally, artificial system calls are injected into the CFG to model the implicit system behavior. The entry block of each subtask and ISR is "prefixed" with an `kickoff()` system call. This system call represents the OS transition of bringing a subtask from the `READY` to `RUNNING` state for the first time since the last termination. The exit block of ISRs is "postfixed" by an `iret` system call. A `StartOS` function with a `StartOS` system call is introduced to have an anchor for the system bootup. The idle subtask and its handler, which is executed when no other subtask is executable, is also introduced.

These artificial system call blocks are similar to the *explicit join points* used in aspect-oriented programming. An explicit join point is an call to an empty function. Its mere purpose is to install an explicit name for a location in the source code. During the aspect-oriented compilation process, the aspect-weaver can match on those locations and apply different code-snippets (aspects) to them. Explicit join points were already used before for designing and implementing real-time operating systems [30, 28].

The ABB detection is carried out on each function CFG like described by Scheler [44, p. 65f]. Basic blocks are only merged into one ABB, if they have a different type (call/non-call). The result is a CFG consisting of ABBs for each function. ABBs may be simple computation blocks, function call blocks or (artificial) system call blocks. As an example the CFG of an ISR handler is shown in Figure 3.4. The single basic block is unstiched at the system call (`call ActivateTask`). Artificial system call blocks are introduced at the entry and the exit of the handler to model the system behavior.

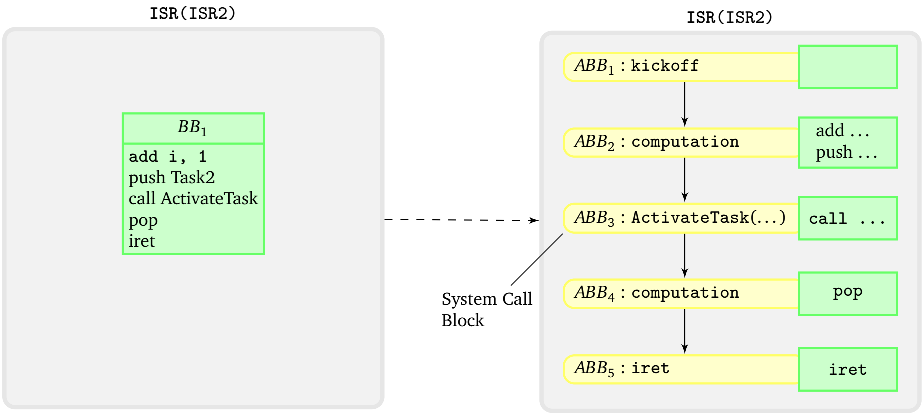


Figure 3.4 – CFG with artificial syscall blocks. Into the CFG of an ISR artificial system call blocks were introduced and system calls are individualized into blocks.

The CFG-construction and the uplifting from basic blocks to ABBs is also the phase that has to cope with legacy systems. In many embedded environments the source code of the application is not available for the OS vendor. Often “application blobs” are delivered with an accompanying system description (e.g., written in OIL). But for the system analysis, we need the CFG and information about system call and function call locations. This information could be obtained by forcing the vendor to deliver a CFG. This CFG does not have to contain all information about the basic blocks, but only the ABB structure, which hides most of the application knowledge. But the CFG can also be obtained by binary analysis. Reconstruction of the CFG from machine code is also done for flow-sensitive worst-case execution time analysis and already available for many architectures [49, p.103].

The normal OSEK semantic of system calls is quite permissive when it comes to system-call arguments. Since most OSEK-compliant operating systems are linked as a library against the application, they must handle variable arguments anyway. When using those systems the application developer can store a subtask id in a global variable and use it as a parameter for the `ActivateTask` system call. Listing 3 shows an example of an utterly complex subtask arrangement that is only possible due to variable system call parameters.

```c
int timer;
TaskID to_activate;

ISR(watchdog) {
  timer++;
  ActivateTask(to_activate);
  if (timer % 2 == 0) {
    to_activate = TaskA;
  } else {
    to_activate = TaskB;
  }
}

Task(TaskA) {
  to_activate = TaskC;
  TerminateTask();
}
```

Listing 3 – Example of variable System Call Argument. With variable arguments, hard to analyze subtask arrangements can be constructed.

But often this flexibility is not needed, since the application developer knows in every situation for sure which subtask to activate. Additionally, variable arguments make the implementation and the identification of interacting components harder and less clear. Especially in safety-critical real-time systems clarity is a key concern.

Limitation 2. *System calls must be explicit. Arguments to system calls that reference system objects (e.g. subtasks or alarms) have to be constant.*

Therefore, I demand the application to use constant arguments for certain system-call parameters. Every time when a system object (e.g., a subtask or an alarm) is required as a parameter, it has to be a compile-time constant. Thereby the system objects that are referenced by the system call are known beforehand. Especially, it ensures a static subtask arrangement: It is always know which subtask influences another one.

This limitation does really limit the expressiveness of the OSEK standard. Constructs, like the one presented in Listing 3, are not possible anymore. It must also be ensured that the decision which system call to execute is known for a system-call site. From a real-time engineers perspective the limitation increases the explicitness of the system. If such dynamic behavior is desired, it can still be achieved with an if-elif-else cascade containing an explicit system call for each possible case.

3.3.2 The ICFG and its Passes

As described earlier, the *inter-procedural control flow graph* (ICFG) expresses all possible execution paths that can be executed by the subtask VM of a certain subtask. A subtask's ICFG is a combination of the functions that can be called within the subtask's context. We base the ICFGs construction on the single function CFG's system:

1. All ABBs in the system are the nodes of the ICFG.
2. All edges from all function CFGs are copied into the ICFG.

3. ABBs calling a function are connected to the entry ABB of the called function.
4. The exit ABBs of the function are connected with the successor of the calling ABB.
5. The edge from the calling ABB to its successor is removed from the ICFG, but kept in the CFG.

Figure 3.5 shows both the CFGs and the ICFG for two functions, where `Function1()` calls `Function2()`. `Function1()` calls `Function2()` in two different ABBs, which both get an edge to ABB_4. Both successors of a calling ABB get an edge from the exit block of `Function2()` (ABB_5). The edges (ABB_1, ABB_2) and (ABB_2, ABB_3) are not present in the ICFG. This absence of edges expresses that those blocks cannot be executed in sequence within the subtask VM. Nevertheless, within the function VM of `Function1()` they must be executed directly after each other. A function call is an "instruction" of the subtask VM that is interpreted by the subtask VM and not propagated to the currently running function VM.

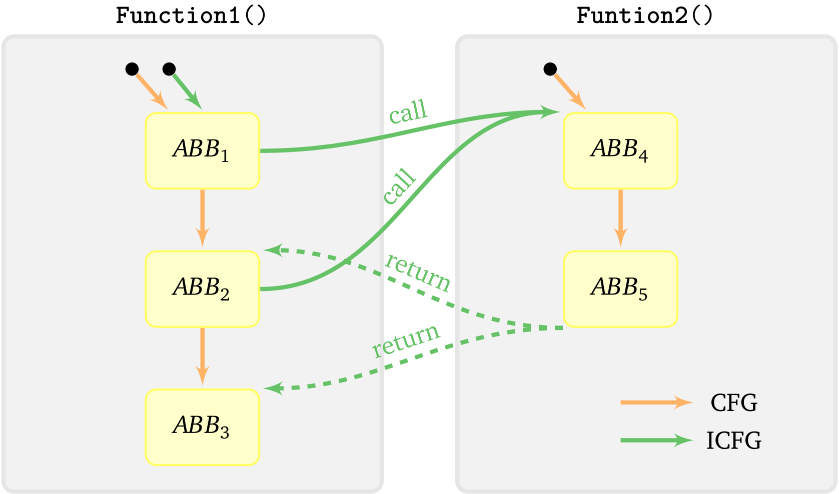


Figure 3.5 – CFG and ICFG for two functions. The ICFG contains the same nodes as all CFGs combined. The ICFG is an "overlay" graph for all CFGs.

After all call–return edges are encoded within the ICFG, the ICFG for a certain subtask is the subgraph that can be reached from the entry ABB of the subtask handler. Return edges in this reachability analysis are treated special: They may only be considered if the corresponding call edge is already part of the subtask's ICFG. This expresses the system semantic "no return without prior call". With this reachability analysis, we calculate the subset of functions that can be reached from the subtask's entry ABB. Since the ABBs of the whole system are the initial nodes of the ICFG, an ABB could be reachable from two subtask entry ABBs. This ambiguity would impede my cross-kernel analysis, therefore I forbid this ambiguity:

Limitation 3. *Each subtask has a set of "system-relevant" functions that are reachable from the entry ABB on the subtask's ICFG. These sets have to be pairwise distinct.*

This limitation would have severe effects on the expressiveness of the application code, if all functions are taken into account. But we restrict this limitation to *system-relevant* functions. System-relevant functions that interact, directly or indirectly, with the operating system. In order to determine the system-relevant functions, we analyze the call relationship of all function in the system. Each function that calls a system call by itself is marked as system-relevant. All functions that call system-relevant functions are marked as system-relevant themselves. This marking of functions can be achieved by a depth-first search on the call graph.

If we only take system-relevant functions into account, we decrease the size of the ICFG significantly. Additionally, the restriction of system-relevant functions makes Limitation 3 more tolerable, since less functions have to be taken into account when comparing the function sets. Virtually the effect of the functions, which are not system-relevant, is subsumed by the ABB containing the function call. This subsumption of whole function call hierarchies is already described by Scheler in the term of the *global ABB graph* [44, p. 69]. With the notion of system-relevant functions the usage of libraries that do not interact with the OS is not problematic. Virtually, they become part of computation blocks.

The result of the ICFG construction is a directed graph of ABBs for each subtask. Limitation 3 ensures that all ICFGs have a pairwise distinct node set. A counter example for a system that does not fulfill this limitation is shown in Listing 4. Therefore, the executing subtask VM is clear without ambiguity for each ABB in a system-relevant function. For the rest of this thesis, ICFG refers always to the ICFG consisting of system-relevant functions.

```
TASK(TaskA) {
    bar();
    TerminateTask();
}
TASK(TaskB) {
    bar();
    TerminateTask();
}

void
bar(void)
{
    ActivateTask(TaskC);
    return;
}
```

Listing 4 – Forbidden activation of system-relevant function. The system-relevant function `bar()` can only be called from a single subtask.

On the ICFG, we do several analysis passes in order to gather information about the system state that is associated with the subtask VM. This information is switched with the change of the subtask VM. The first information that is trivial to obtain for each ABB is the currently running subtask. It expresses the affiliation of an ABB to a subgraph's ICFG.

$$\text{running\_subtask\_in} :: \texttt{ABB} \rightarrow \texttt{Subtask}$$

More information about the system state is obtained by data-flow analysis on the ICFG. Data-flow analysis is a standard compiler technique [34, S. 217–266] using the control-flow graph to simulate the flow of information within the program. For easier understanding, I will outline shortly the general data-flow approach on control flow graphs.

In Figure 3.6, the basic principle of a forward data-flow analysis is given. "Forward" denotes here the flow direction of information; it flows forward through the control-flow graph. An initial data-flow information vector is created for each block. The data-flow operation for a block merges all vectors from its predecessors. The merged vector is transformed according to the transformation function of the basic block and the result is propagated to the successors of the block. The data-flow operation is repeated for all blocks, whose inputs have changed, until the information vectors converge. A data-flow analysis must define the initial vector, the merge function and the manipulation function. It must also always reach a fixpoint in order to terminate. The result of the data-flow analysis is the set of vectors after a fixpoint is reached. Typical data-flow analyses in compilers calculate *reachable definitions* and do *constant propagation*.

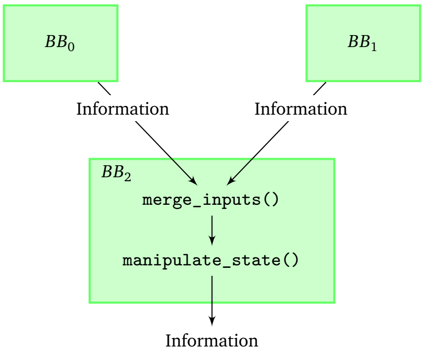


Figure 3.6 – Data-flow analysis schema. Each basic block merges the incoming data-flow information from its predecessors, transforms the information and propagates the result to its successors.

Since each ICFG is associated with a subtask, we can use a ICFG data-flow analysis to obtain information that stays within the subtask VM context. As an example, we can consider the `running_subtask_in` information as a data-flow problem. We initialize the currently running subtask for all ABBs to an empty set; for the entry ABB of the subtask handler we add the subtask to the set. The merge function calculates the union of the incoming subtask sets and the transform function propagates the result. Of course, the result is unsurprising, since we constructed the ICFG to contain only ABBs of one subtask. But other parts of the system state can be obtained in a similar manner.

3.3.2.1 Interrupt Block State

An OSEK application is allowed to block interrupts for protecting critical regions. The specification defines several system calls for interrupt block control (see Table 3.1). The different variants differ in their nestability and the class of interrupts they block. Since each of those system calls block ISR2s, which are the only interrupts we are interested in, we treat all pairs of block/unblock operations the same. In the same manner, we only over-aproximate the operations, when we assume the block/unblock operations can be nested.

System Call	Nestability	ISR Types
`EnableAllInterrupts()`	no	ISR1, ISR2
`DisableAllInterrupts()`	no	ISR1, ISR2
`SuspendAllInterrupts()`	yes	ISR1, ISR2
`ResumeAllInterrupts()`	yes	ISR1, ISR2
`SuspendOSInterrupts()`	yes	ISR2
`ResumeOSInterrupts()`	yes	ISR2

Table 3.1 – Interupt Block System Calls. The interupt block system calls differ in their nestability and in the class of blocked interrupts.

OSEK implies strict rules for the usage of interrupt blocks usage: if interrupts are blocked no system call, besides the unblock operation, may be executed [37, S. 26]. On top of this strict rules, I put another limitation on the usage of these system calls in order to ease the system analysis:

Limitation 4. *The interrupt-block nesting count must be unambiguous for each ABB.*

From an developers point of view, this limitation enforces a very disciplined usage of interrupt blocks. For each block operation, the unblock operation should be located in the same source code block. Functions that block interrupts must not be called from different interrupt-block nesting levels. These guidelines are sane for each real-time system. Critical sections protected by interrupt blocks should be kept small, in order to keep the worst-case latency low. Therefore, it is acceptable to imply very strict limitations here.

The information vector for the interrupt-block data-flow analysis is a plain integer for each ABB: the nesting count. Initially, all counts are set to zero. The merge functions calculates the maximum of the incoming nesting counts. An interrupt block ABB increases the nesting count; an interrupt unblock decreases it; all other block types just propagate the nesting count to their successors. We can enforce Limitation 4 by checking, that all incoming edges for a certain ABB propagate the same nesting count. The analysis result is the information whether the interrupt-nesting count is higher than zero or not:

$$\mathsf{isr\_block\_in} :: \texttt{ABB} \rightarrow \texttt{Bool}$$

With this analysis, the isr_block system-state information described in Section 3.2 is derived from other parts of the system state, namely the currently executed ABB. isr_block_in gives the state of the interrupt block for the currently executed ABB. This principle of reducing parts of the system state to information that can be calculated statically beforehand is one key to improve the analyzability.

3.3.2.2 Resource Occupation

The principle of reducing the dynamic system state can also be applied to the usage of OSEK resources. The OSEK specification implies here also strict rules for the acquisition of resources. The set of participating subtasks must be noted in the OIL file beforehand. When a subtask holds a resource, the subtask is not allowed to terminate or to wait for an event. Nested resource acquisition is only allowed when the critical sections are strictly stacked and resources may not be taken more than once [37, S. 58]. Similar to the interrupt block, I give on top of that another limitation on the usage of resource, so that the system gets more analyzable:

Limitation 5. *In the ICFG of a certain subtask, the occupation state must be unambiguous for the resources the subtask is allowed to acquire.*

```
if (cond) {
    GetResource(Res1);
}

computation();

if (cond) {
    ReleaseResource(Res1);
}
```

Listing 5 – Conditional Acquisition of a Resource. When a subtask takes a resource only under a certain condition, the acqusition state for `computation()` is ambiguous.

Like in interrupt case, this limitation ensures a very disciplined usage of resources. The limitation prohibits, that a resource is taken only conditional. Listing 5 shows such a conditional acquisition. From the standpoint of clarity, it is desirable to put strong restrictions on the usage of resources, since they influence the subtask inter-leavings significantly. Therefore a real-time system developer should handle critical sections with care; or to say it in the words of the OSEK specification: "Generally speaking, critical sections should be short" [37, S. 59].

As a result of the OSEK priority-ceiling protocol, the acquisition request issued by a subtask succeeds always. This is due to the fact that the acquisition of an resource boosts the subtask's priority *immediately* to the ceiling priority. Therefore all other subtasks than can acquire the resource, as it is denoted in the OIL file, cannot be scheduled, since they have a lower priority. Therefore an analysis on the ICFG is sufficient. When a subtask can acquire resource A, there are two possible cases within the subtask VM: either the resource is not occupied, then the subtask VM can execute and can acquire the resource by itself; or the resource is occupied by somebody else, then the subtask VM *cannot* execute. So when taking the perspective of the executing subtask VM, the information about the resource acquisition state stays on the ICFG.

We calculate the occupation state for each subtask with a data-flow analysis on the ICFG. The information vector consists of two bits for every resource that can be obtained by the subtask. The first bit indicates that the resource is taken by the subtask. The second one indicates that it is *not* taken. Therefore only the two bit-patterns `10` and `01` are valid. All other patterns signal that Limitation 5 was not met or that the resource protocol, as described in the specification, was violated.

```
Require: vectors: Incoming occupation vectors
Require: abb: Currently executed ABB
Require: res: Currently examined resource
outgoing_vector = '00'
for vec in vectors do
  outgoing_vector = outgoing_vector or vec
end for
if abb.type = GetResource and abb.argument == res then
  outgoing_vector = '10'
else if abb.type = ReleaseResource and abb.argument == res then
  outgoing_vector = '01'
end if
return outgoing_vector
```

Algorithm 3.1 – Data-Flow Operator for Resource Occupation State. The data-flow analysis is executed for each ICFG and for each resource taken by the subtask. This operator combines the merge and the transformation operation

The merge function does a bit-wise or operation on the incoming state vectors. A `GetResource` block sets the bit-pattern for the given resource to `10`. `ReleaseResource` sets it to `01`. This can only be done since resources are system objects and Limitation 2 enforces those system call arguments to be constant. The arguments are accessible for the analysis. Algorithm 3.1 shows the combined merge and transformation function for this data-flow analysis.

The analysis reaches always a fixpoint, since bit-wise or operation is a strictly monotonic merge function and the transformation function does not change the number of 1's. After the fixpoint is reached, we know whether a resource is occupied or free in an ABB. The analysis is carried out for every subtask's ICFG.

$$\text{resource\_taken\_in} :: (\texttt{ABB, Subtask, Resource}) \to \texttt{Boolean}$$

For resources that can be occupied by a subtask, the resource_taken information is equivalent to the resource_taken_in information. For resources not taken by the subtask it does not provide any information. But the general resource occupation state, which is defined for all resources at all points, cannot be obtained from the system anyway.

3.3.2.3 Dynamic Priority

OSEK uses the resource occupation state for exactly one thing: determining the dynamic priority of a subtask. With the resource_taken_in information at hand, we can calculate the dynamic priority for a subtask at a certain point *statically*. For an ABB, the maximal ceiling priority of all resources that are taken determines the dynamic priority.

$$\text{priority\_in} :: \texttt{(ABB, Subtask)} \rightarrow \texttt{Integer}$$

$$\texttt{priority\_in(a, s)} = max \left\{ \texttt{res.ceiling\_prio} \;\middle|\; \begin{array}{l} \forall \texttt{res} \in \texttt{s.resources} \\ \quad \wedge \texttt{resource\_taken\_in(a, s, res)} \end{array} \right\}$$

Normally resources have not assigned a static priority, but their ceiling priority is depended on the resource group. OSEK allows the system generator to remap the static priorities of the subtasks, as long as the priority structure is not changed. Therefore, I create a uniform priority space by assigning a static priority to each subtask, ISR and resource. These system objects take part in the scheduling process. The idle subtask, which is artificial, gets the lowest priority. The system priority structure is preserved, but it is spreaded to make room for the resources. A resource gets a higher static priority than all subtasks that may obtain the resource. All ISRs have an higher priority than all subtasks and resources. The result is a priority structure that may look like the one in Table 3.2.

Higher Priority

System object	Static priority
Idle subtask	0
Subtask A	1
Subtask B	2
Resource A	3
Subtask C	4
ISR A	5
ISR B	6

Priority Group

Table 3.2 – Example of Priority Spreading. The priority relation between Subtask A, B, and C is preserved. The dashed box indicates a priority group: both Subtask A and Subtask B can obtain Resource A. The ISRs have a higher priority than all other system objects.

A possible ICFG for the priority distribution from Table 3.2 is shown in Figure 3.7. The corresponding priority_in and isr_block_in are annotated for each ABB. Subtask A acquires the resource; during the holding period its dynamic priority is increased. Subtask B issues a interrupt blocked. The dynamic priority and the interrupt block state is not reset until the successor of the releasing system call for technical reasons.

3.3.3 System-Semantic Simulation

The ICFG passes provide static system information for each ABB block, which is derived from the applications flow graph and the system description. This static information is used to construct

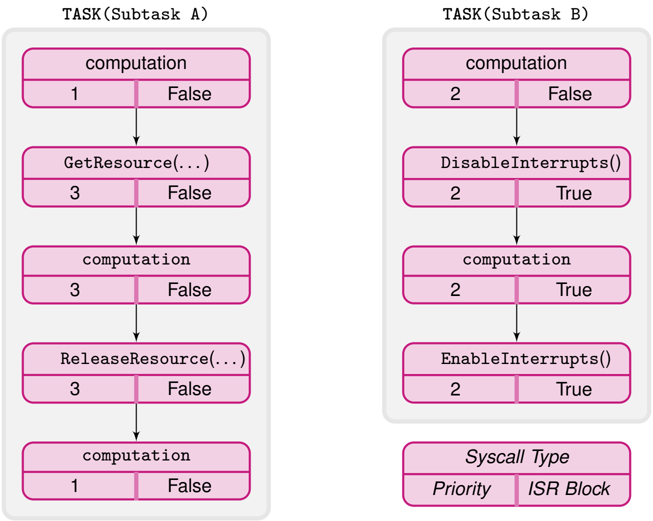


Figure 3.7 – ICFG analyses result for Table 3.2. Subtask A acquires Resource A and changes therefore its dynamic priority. Subtask B has a critical section protected by an interrupt block.

the GCFG by simulating the OSEK system semantic. I will present two approaches for the system simulation, which differ in precision and complexity. Both methods encapsulate the system semantic in a transformation function that maps one system state into many possible output states. Since each output state is a possible outcome, this function encapsulates the indeterminism of an event-triggered real-time system. When revisiting our concept of the multi-level machine, this simulation corresponds to the operation the system VM executes when it interprets a system call or an interrupt request.

$$\texttt{system\_semantic :: State} \rightarrow \texttt{[State]}$$

This function combines the ABB type and the OSEK scheduling semantic. It includes the `ABB_transformation()` (see Figure 3.3) to reflect the ABB semantic. On the result state, a virtual scheduling that resembles the OSEK semantic is carried out. Exemplary, I want to show for two block types the `ABB_transformation()`. Afterwards the "block scheduler" and the "virtual dispatch" is explained.

```
State ABB_transformation_ChainTask(State state, Subtask target) {
    source = running_subtask(state);
    State ret = copy_state(state);
    continuations(ret, source) = [source.entry_abb];
    subtask_state(ret, source) = SUSPENDED;
    subtask_state(ret, target) = READY;
    return [ret];
}
```

Listing 6 – `ABB_transformation()` for `ChainTask`. The transformation copies the incoming system state and manipulates the copy. The `ChainTask` returns only one followup state.

The `ChainTask()` system call is used in OSEK to atomically terminate the current subtask, while activating another one. The execution is "chained" to the subtask that is given as an argument. The transformation, which is depicted in Listing 6, overwrites the subtask_state for the currently running subtask to `SUSPENDED` and for the argument subtask to `READY`. The continuations set for the current subtask is set to the entry ABB of the subtask's handler function; for the target subtask they remain untouched.

Since parts of the system state are overwritten by the `ABB_transformation()`, they are precise afterwards, even if they were fuzzy before. This observation is true for each synchronous system call; system calls within non-ISR code increase the precision of the system state.

The most complex ABB transformation is carried out for computation blocks (see also Figure 3.8). Due to branches, computation blocks, unlike system call blocks, may have more than one successor in CFG and ICFG. Here the subtask VM can decide to execute one of the successors. Therefore, for each successor in the ICFG, a output state is created. In order to reflect the increased instruction pointer effect, the running_abb information is set to the successor.

Additionally to the synchronous control flow, interrupts may occur within computation blocks. For a currently executed ABB the external event occurs only when interrupts are not blocked (isr_block_in). For every configured interrupt, an additional output state is emitted. These output states reflect the effect of the system VM, when an interrupt-request "instruction" is executed by the system VM. In each followup state, the subtask corresponding to the ISR is set to `READY`. For modelling the preemption semantic for interrupts, nothing has to be changed for the currently running subtask. Its continuations remain the same, since the execution returns to the same place after the interrupt returns. This return-to-self semantic allows the execution of several ISRs in sequence during the execution of a single ABB.

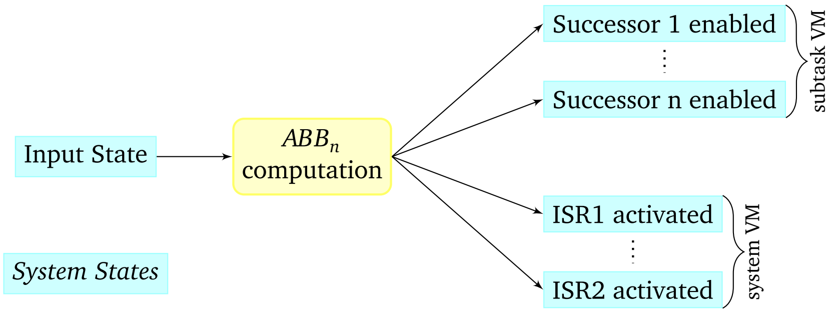


Figure 3.8 – Output System States for Computation Block. Computation blocks are the most complex ABB type and produce several output states for the successors in the ICFG (branches and function calls). Computation blocks also model interrupt events.

On the resulting states of the ABB transformation a virtual scheduling is executed. This virtual scheduling is done to determine which subtask is the currently running subtask. Precise system states are easy to schedule: always the highest-priority `READY` subtask is chosen. The problem gets more complex with a fuzzy system state. If a high-priority subtask is only `FUZZY`, it may be the scheduled subtask, but we do not know for sure. The most simple solution is to create a followup state where each `READY` or `FUZZY` subtask is the dispatching target, regardless of its priority. But this would not take the system semantic of scheduling the highest-priority task into account. The *block scheduler* combines both ideas to generate a more precise scheduling result.

The *block scheduler* is named after the fact, that it decides upon a fuzzy state which ABB to execute next. It is depicted in Algorithm 3.2 as pseudocode. The result for calling it on an input state is a list of ABBs that are possible dispatch targets. The scheduler first collects a list of candidate blocks. The continuations of `FUZZY` and `READY` subtasks are candidates. For `FUZZY` subtasks, all continuations are candiates, but they are marked as *not* surely ready. For `READY` subtasks, all continuations are added, but only those with the lowest dynamic priority are marked

```
Require: in_state: System state to schedule upon
possible_blocks = [ ]                    // List of tuples of the form (ABB, priority, surely_ready)
// Phase 1: Collect possible blocks
for subtask in all_subtasks do
    if subtask_state(in_state, subtask) is FUZZY then
        for abb in continuations(in_state, subtask) do
            possible_blocks append (abb, priority_in(abb, subtask), false)
        end for
    else if subtask_state(in_state, subtask) is READY then
        min_prio = min({priority_in(abb, subtask) | abb ∈ continuations(in_state, subtask)})
        for cont in continuations(in_state, subtask) do
            // Only the block with the lowest dynamic priority is surely running.
            surely_ready = (priority_in(abb, subtask) == min_prio)
            possible_blocks append (abb, priority_in(abb, subtask), surely_ready)
        end for
    end if
end for
// Phase 2: Sort blocks by priority; Highest priority to the front.
possible_blocks = sort(possible_blocks, sort_by = priority)
// Phase 3: Determine the minimum system priority
min_system_prio = 0
for (abb, priority, surely_ready) in possible_blocks do
    if surely_ready then
        min_system_prio = priority
        break for loop
    end if
end for
// Phase 4: Determine the return set of blocks
return_blocks = [ ]
for (abb, priority, surely_ready) in possible_blocks do
    if min_system_prio ≤ priority then
        return_blocks append block
    end if
end for
return return_blocks
```

Algorithm 3.2 – The block scheduler

as surely running. Later on, I will give an example where this behavior is mandatory. In Phase 2, the possible blocks are sorted by their dynamic priority. In Phase 3, a minimum system priority is determined. The minimum system priority is the absolute minimum priority of a subtask that is running after the schedule/dispatch process is executed. No subtask with a priority below the minimum system priority can be the result of the scheduling process. In Phase 4, exactly those blocks with an equal or higher priority are collected as the return set.

The special case in Phase 1 of the algorithm is needed for situations like the one shown in Table 3.3 and Listing 7. Let's assume that TaskC is currently running. It can either be in ❸ or ❹. In both places it has two different dynamic priorities. Then the interrupt ISRA fires. ISRA may activate TaskB, but it is not for sure. Therefore TaskB's subtask_state is `FUZZY` when ❶ is reached. When the block scheduler has to decide the correct result would be, that it can dispatch as well to ❷, ❸, and to ❹. But when we look only at the priority table and go from right to left and take blocks as long as they are `FUZZY` or `READY` and stop at the first `READY` block, we will forget ABB_1 and ABB_2.

	ABB_1	ABB_2	ABB_3	ABB_4
Subtask	TaskC	TaskB	TaskC	ISRA
ABB type	`computation`	`computation`	`computation`	`iret`
Running State	`READY`	`FUZZY`	`READY`	`SUSPENDED`
Dynamic Priority	4	5	6	7
Surely Ready?	True	False	False	–

Table 3.3 – Corner case for the block scheduler. When Task A terminates the block scheduler may not decide that only ABB_1 is the only outcome of the scheduling decision, but ABB_1, ABB_2, and ABB_3 are possible.

```
ISR2(ISRA) {
  if (cond1) {
    ActivateTask(TaskB);
  }
  // ❶: ABB_4
}

TASK(TaskB) {
  // Priority 5
  // ❷: ABB_2
}

TASK(TaskC) {
  if (cond2) {
    // Static Priority 4
    // ❸: ABB_1
  } else {
    GetResource(Res1);
    // Ceiling Priority of 6
    // ❹: ABB_3
    ReleaseResource(Res1);
  }
  ...
}
```

Listing 7 – Code for the Special Case in Table 3.3

In phase 4 of the algorithm, all blocks are collected, from top-down, until a surely running block comes along. Therefore, we have to mark ABB_3 as not surely ready, although TaskC is `READY`. This ambiguity stems from the fact that the subtask_state is stored per subtask and not per ABB.

The block scheduler is called on every possible output state the block semantic produces (see Figure 3.8) to get a list of possible blocks. A virtual dispatcher is called on the result. The virtual dispatcher fixes up the system state by asking "What would happen to the system state if we dispatch to that block?". For this, it also uses the list of possible blocks sorted by their dynamic priority. If a specific block is the dispatch target, all blocks with a higher dynamic priority are wiped out of the continuation sets, since they cannot be ready when the lower-priority ABB is dispatched. If a subtask has no continuations left, we can say for sure, that it is `SUSPENDED`. In the example from Table 3.3, a virtual dispatching to ABB_1 results in removing ABB_2 and ABB_3 from the continuations. The virtual dispatcher marks `TaskB` as `SUSPENDED` in this schedule-dispatch result.

The whole `system_semantic()` operation is shown in Figure 3.9: The ABB state transformation incorporates the influence of the ABB type on the system state. The block scheduler selects the possible dispatching targets for each followup state, and the virtual dispatcher takes the influence of the scheduling decision on the system state into account. A few more corner cases, like not scheduling away from non-preemptive subtasks or ISRs, have to be taken into account when implementing the OSEK semantic, but those details are only technical and of no further interest.

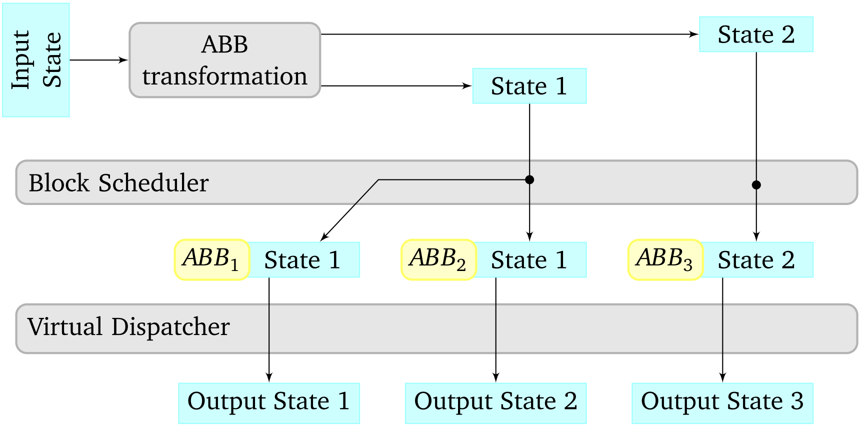


Figure 3.9 – Operation schematic of `system_semantic()`. Upon the multiple output states of the ABB transformation the block scheduler and the virtual dispatcher calculate the followup states.

3.3.3.1 Symbolic System Execution

After the system semantic is hidden within the `system_semantic()` function, the first method to construct the GCFG is rather straightforward. The fundamental idea is to define an initial state (e.g., after the OS is started with `StartOS`) and to use `system_semantic()` recursively until all possible system states are discovered. The result of this recursion is a graph with system states as nodes, which are connected with a directed edge if the target node is a possible followup state of the source node. Duplicate edges are not inserted into the *symbolic system execution* (SSE) graph.

To improve the precision of the SSE, the abstract system state can be enriched by a call stack. When a calling edge is taken on the ICFG the source block's successor on the CFG is pushed onto the subtask's call stack. If the function returns, not all return edges are taken, but only the one to the block on top of the call stack. The returning ABB is popped off the stack.

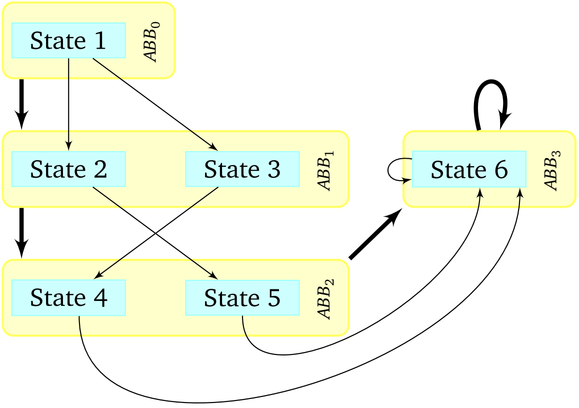


Figure 3.10 – SSE graph and the ABB grouping. The system states calculated by the SSE are grouped on the currently executed ABB and GCFG edges are inserted.

The GCFG is constructed by grouping the possible states in the SSE graph by the currently executed ABB (like in Figure 3.10). Since the currently executed ABB is part of the state information, this grouping is unambiguous. For each edge in the SSE graph, the GCFG gets an edge from the ABB group of the source state and to the ABB group of the target state. The result is a graph of ABBs, where an edge exists if a state transition exists, that brings the execution from one ABB to the other one. This graph has the semantic of the GCFG, since all system state transitions are included.

ISRs can be treated specially in the construction of the GCFG: Limitation 1 forbids that ISRs are interrupted while running. Therefore, they have a strict run-to-completion, non-preemptable semantic. They form regions of ABBs, which are a single-entry/single-exit regions; the entry ABB is the entry node, and the `iret` block is the exit node. To gain a GCFG that only models the normal subtask control flow, these ISR regions are "cut out". This cutting out is done on the system state graph to not loose the discriminatory power of the SSE. As the result of this cutting-out operation, the edge that pointed to the entry ABB of the ISR points now to the followup ABBs of the `iret` block.

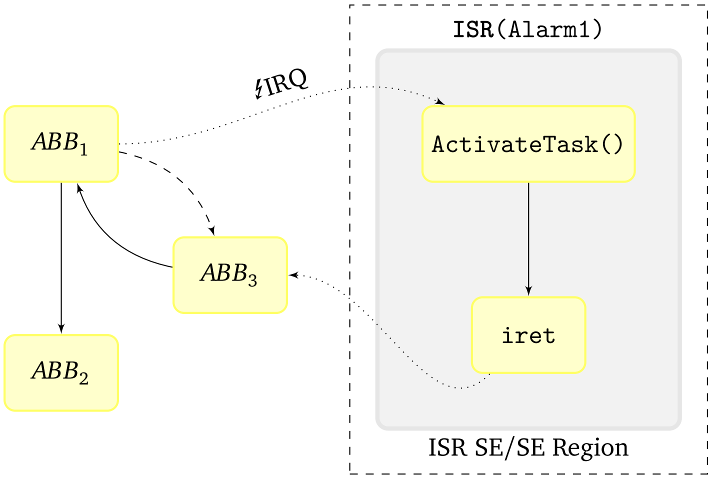


Figure 3.11 – ISR regions within the GCFG can be cut out to get an flow graph that only contains synchronous system calls and non-ISR code. The dashed line is the replacement edge for the SE/SE ISR region.

Figure 3.11 shows such an ISR region. ABB_1 can either directly continue to ABB_2 or an IRQ can occur that enables the Alarm1 handler. The ISR handler issues an operation that makes ABB_3 runnable and terminates itself with the artificial `iret` system call. ABB_3 executes and gives back the control to ABB_1. The time that is executed within the ISR handler can be cut out as an ISR region and we insert direct edge from ABB_1 to ABB_3 (dashed). Since the ISR region replacement results in a loop between ABB_1 and ABB_3, the IRQ can occur more than once before the execution of ABB_1 is finished. Without timing information about ISR execution and the ABB execution, this semantic is the most conservative one.

The SSE analysis is a very precise, but costly method to calculate the GCFG, since it has to enumerate all possible system states. This is only practical because of the very range-bounded abstract system state and the aggressive pruning done in the block scheduler and the virtual dispatcher. The run time is additionally bounded, since most parts of the application are hidden within ABB blocks. These irrelevant substructures are unnecessary for the SSE. Nevertheless, for large systems it might be too expensive to enumerate all system states.

3.3.3.2 System-State Flow Analysis

The SSE, which was discussed in the last section, is an expensive analysis, since it has to enumerate all system states. Therefore, a faster GCFG construction method, which may be more imprecise, is desirable. The *system-state flow* (SSF) method is a data-flow analysis as discussed already in Section 3.3.2. But the SSF is a kind of special data-flow analysis. The graph on which the data "flows" on is the GCFG, which is of course not known beforehand. Edges within the GCFG are constructed "on-the-go" until no new edge is found and the system states converge.

```
Require: initial_state :: system state          // Initial system state after StartOS
// System states are stored for blocks and for edges
state_before = empty map of type (ABB $\rightarrow$ system state)
edge_states = empty map of type ((ABB, ABB) $\rightarrow$ system state)
working_stack = empty stack
//  Set up the working stack and fake the inputs for the initial block
initial_abb = running_abb(initial_state)
state_before[initial_abb] = initial_state
edge_states[(initial_abb, initial_abb)] = initial_state
push(working_stack, initial_abb)
// Run the fixpoint iteration until the working stack is empty
while not isEmpty(working_stack) do
    abb = pop(working_stack)
    state_before[abb] = merge_states(edge_states[(*, abb)])
    followup_states = system_semantic(new_before_state)
    for next_state in followup_states do
        next_abb = running_abb(next_state)
        if (abb, next_abb) $\notin$ gcfg_edges then
            new_gcfg_edge(abb, next_abb)
            edge_states[(abb, next_abb)] = next_state
            push(working_stack, next_abb)
        else if next_state $\neq$ edge_states[(abb, next_abb)] then
            edge_states[abb, next_abb] = next_state
            push(working_stack, next_abb)
        end if
    end for
end while
```

Algorithm 3.3 – The System-State Flow Analysis. The SSF is a data-flow analysis that adds the edges used for the analysis during its traversal of the graph. It results in the GCFG.

The SSF analysis starts with an graph that only consists of all system ABBs without any edges. The `manipulate_state()` function for the data-flow analysis is the `system_semantic()` function. It encloses, as discussed earlier, the semantic of the current ABB as well as the scheduling semantic. So we need only a function that merges the inputs, which are propagated by the predecessors. This merge function combines many system states into one system state, while the result might be a fuzzy system state.

If the `merge_states()` function merges two system states with different subtask_state for a single subtask, the subtask_state is `FUZZY` afterwards. If one subtask_state was `FUZZY` before, the combined state is also `FUZZY`. The continuations set for each subtask, is the union of the corresponding fields in the inputs. All other parts of the system state were statically computed by the ICFG passes as discussed in Section 3.3.2, therefore they are not touched during the merge process.

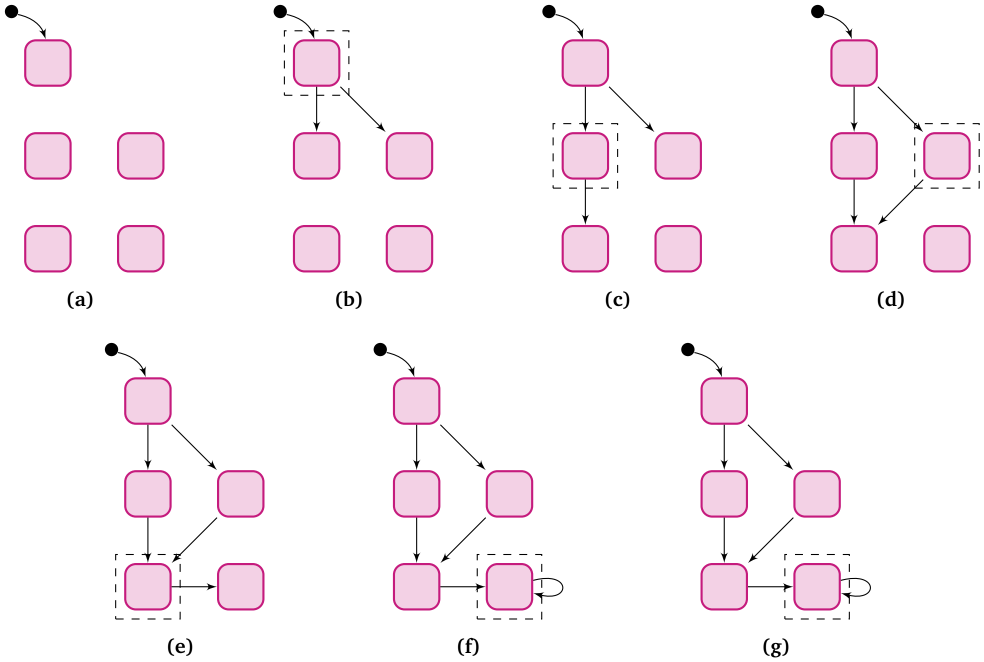


Figure 3.12 – Progressive GCFG construction with *system-state flow* (SSF). During the analysis, new GCFG edges between the ABBs are discovered and the system states are propagated on

The data-flow analysis is depicted in Algorithm 3.3. It resembles a fixpoint iteration implemented as a working stack algorithm. The state of the fixpoint iteration is stored in two maps (line 2f): The `state_before` map stores the system state before each ABB, it is also part of the result. The `edge_states` map stores, for each edge within the GCFG ($ABB \times ABB$), the propagated system state. Since the virtual dispatcher discussed earlier fixes up the system states according to the scheduling decision, not all outgoing edges of an ABB propagate the same system state.

The algorithm is initialized in line 6f with the system state after the OS is booted. Until the working stack is empty, an ABB is popped off. The states propagated by the predecessors of the ABB merged and a set of followup states is derived (line 13f). For each followup state a new GCFG edge is created if necessary (line 18). If a new edge was introduced or the propagated system state changed, the target ABB is pushed onto the working stack and the propagated state is updated (line 19 and line 22). Since the merge function is monotonic in the sense that states only get more fuzzy, all GCFG edges are found, the `state_before` map converges, and the fixpoint iteration terminates. Figure 3.12 shows the progress of this edge construction process. The current ABB is always marked with a dashed box.

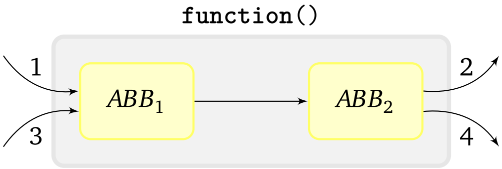


Figure 3.13 – SSF merge at Function Calls. Function calls are handled in SSF by using the call–return edges from the ICFG. This causes the states to merge in the entry ABB of each function.

The first aspect that is of special interest with the SSF is the handling of function calls. The solution to simulate a call stack, like it was done with the SSE, is not applicable here. When two different call stacks flow together into one ABB, the merge result is not defined meaningful. A fuzzy call stack is not useful to determine where to return when the function has finished. Therefore the call–return edges from the ICFG are used. The entry block of an function has all call sites as predecessors and the exit blocks have all returning blocks as successors. By this, function entry blocks are a major source of merged, and therefore fuzzy, system states. This problem is illustrated in Figure 3.13: In the real system the OS state that enters the function via the edge 1 can only leave it with the return edge 2 (same with edge 3 and 4). The SSF merges the system states from edge 1 and 3 in the ABB_1 block and propagates it to both successors of ABB_2.

The other important aspect of SSF is the handling of ISRs. Conceptional the invocation of an ISR through an IRQ is like a asynchronous subtask activation, which can happen anytime in the execution of an computation block; it is an asynchronous system call. So the first idea would be to make every computation, besides its propagation on the ICFG, a subtask activation. But then, the entry block of the ISR has the same problem as the entry block of functions, it merges all incoming states. But it is worse here, since an interrupt can happy nearly at all times and therefore each computation block has an edge to the entry ABB. All system states are merged in the entry ABB, and therefore redistributed into the “normal” application flow. This would render the analysis nearly useless.

To prevent this merge problem, Limitation 1 (uninteruptible, non-preemptable interrupts) is exploited. For every possible interrupt invocation another SSF analysis is spawned. The current system state is used as the initial state. The ISR, which is modeled as a subtask, is activated and the analysis starts with the ISR entry block. In this secondary SSF analysis no interrupts can occur and the subtask cannot be preempted. Therefore the ISR activation must end at the `iret` block. We use the system state at that point to summarize the influence of the ISR on the initial interrupt state and we use it as one result of the `ABB_transformation()` function. The ISR activation forms an “ISR transaction”.

Additionally, this ISR transaction semantic results in the same GCFG semantic as it is given by cutting out the ISR regions from the SSE graph. Therefore, both GCFGs are compatible, but the SSE is more precise than the SSF. This imprecision afflicts itself in additional flow edges that are not possible in the real system. They stem from the merge problematic at function entries.

3.3.3.3 Combination of Results

With two GCFG constructing methods at hand, multiple operation combinations are possible. Of course, either one of the methods can be picked to suite the specific requirements on the algorithm. But they can also be combined. For handling small system precisely and large systems fast, the SSE can be executed for a given timeout. If the calculation takes to long, SSF is used as a fallback option. A fallback mechanism can also be used if other features should be considered, which are not trivial to implement in the SSF. For example, the usage of events and wait states is future work for the SSF analysis, but it could be quite easy integrated into the SSE. Therefore SSF might be used for systems without wait states and SSE only for systems where subtasks wait for events.

Whether both algorithms are not only combineable, but also composable is a topic of further research. Perhaps it is fruitful to use the SSF to get an over-approximation of the GCFG and to check with SSE whether the edge is really possible at that point.

For validation purposes, both methods can be used in parallel and the results are compared. Since the SSE is the more precise method all edges in its GCFG must be also present in the SSF's GCFG. And the other way around: no edge found by SSF may be missing in the SSE result. By this, both methods were validated against each other during the development phase. When adapting the approach for another OS, this validation may help the developer to find bugs.

3.4 Incorporating More Information

In the GCFG, all possible decisions of the system VM must be present to gain a valid result; no possible system transition may be missed. On the other hand, present edges that are in the real world impossible decrease only the quality, but do not harm the validity. Therefore the GCFG is an over-approximation. As a thought experiment, we can say that all ABB transitions are possible, but most of them are guarded with contradictory conditions. So, unnecessary edges are no problem; missing edges undermine the validity. Therefore, we can only remove edges if we can promise that an edge can never be taken.

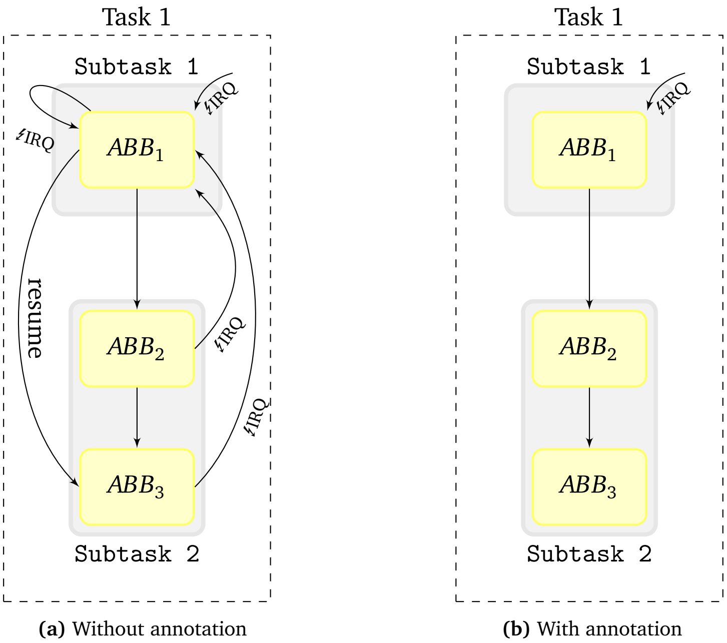
The main source of edges in the GCFG analysis are interrupts. Interrupts can enable the ISR handler in every computation block. The activation of an ISR manipulates the system state with the ISR transaction. This introduced indeterminism also causes the SSE analysis' run-time to explode, since it "forks" the system state space in every computation block. So the limitation of ISR activation points in the system is not only worthwhile for getting a more dense GCFG, but also decreases the run-time of the system analysis.

The first approach to limit the ISR activation points uses the isr_block_in information, which is statically derived for each ABB in the system in Section 3.3.2.1. In a protected ABB, no interrupts can occur. Technically the `ABB_transformation()` of computation blocks with such a protection ignore the interrupt sources declared for the system. This measure relies only on the OSEK specification and exploits information the developer has written down already.

A real-time engineer has to cope a lot with timeliness, a property we have not exploited yet. It would be possible to combine the GCFG construction with an *worst-case execution time* (WCET) analysis of the application. In combination with the minimal inter-arrival time for interrupts, we could eliminate re-activations of interrupts at a given time. But the usage of WCET analysis is out of the scope of this thesis. Therefore, I will take a much simpler approach: The application developer promises that interrupts cannot occur at a given time.

The external-view of a real-time system. which was defined by Scheler [44, p. 21], will be exploited. In the external view, a task is activated by some event. In the technical implementation, the concept of tasks does not exists, but it is an abstract concept for containers of subtasks. The developer can promise that the task-activating event does not occur again until the task execution has finished. The re-activation does not overhaul the task execution. A task has finished its execution, if all subtasks within have finished. This promise, the *no-reactivation* annotation, can be given for every task.

In Figure 3.14, the influence of the no-reactivation annotation is exemplified. Subtask 1 and 2 are contained within the same task. The task is virtually activated by an external interrupt event, technically this event activates Subtask 1, which chains its execution to Subtask 2. For



(a) Without annotation

(b) With annotation

Figure 3.14 – Task Containers as IRQ-block annotation. Subtasks are contained conceptually in task containers. The developer can annotate a task, such that the activating event (IRQ) may not reoccur, if the task execution has not finished yet.

demonstrative and density reasons all ABBs in the shown systems are computation blocks. Therefore the activating interrupt can occur at any time and activate Subtask 1. When the interrupt occurs within ABB_1 the system state does not change, therefore an self-loop is inserted. In ABB_2 and ABB_3, subtask 1 is normally not ready, therefore Subtask 2 is preempted in favor of Subtask 1, and additional resume edges have to be inserted.

When the no-reactivation annotation in place, the system state transition for computation blocks checks if any subtask in Task 1 is ready or currently running. If this is the case no interrupt is released. Only the event that initially activates task 1 remains within the GCFG.

The model of forbidding the task reactivation, while it is still running, is similar to the concept of “deadline equals period”, which is often used for real-time systems [27, p. 41]. The no-reactivation annotation is a qualitative statement about the tasks maximal response time (WCET + preemption time): the whole task’s maximal response time is *always* less than the inter-arrival time of the activating event.

The no-reactivation annotation expresses only the bond between the activating event and the corresponding task, but it does not cover the relation to other events or tasks in the system. Additional annotations, similar to the described one are thinkable, but have not been considered yet.

3.5 Summary

To close this chapter, I want to give a quick overview what was reached and what information can be gathered about a concrete application. In the next chapter, this information will be used to tailor the OS closely to the specific application.

The *global control flow graph* (GCFG) was motivated by a model of stacked virtual machines. The GCFG contains all execution paths the system VM can execute. The branching decisions in the GCFG are done upon the *system state* by this system VM. The system state is manipulated by the application through explicit system calls or external events.

The construction of the GCFG is done in several passes with an increasing incorporation of the OS specification: From the compiled application, a CFG is extracted and the basic blocks are subsumed into *atomic basic blocks* (ABBs). Function calls are inserted to gain the *inter-procedural control flow graphs* (ICFGs) from the CFG. On the ICFG several data-flow analyses, which are a standard compiler technique, are used to deduce parts of the system state statically. This includes the state of the interrupt block, resource occupation and the dynamic priority for each ABB.

The *symbolic system execution* (SSE) and *system-state flow* (SSF) are two methods to calculate the GCFG from CFG, ICFG and the static information. The SSE enumerates all possible system states and is therefore expensive, but accurate. SSF does a data-flow analysis, while simultaneously constructing the GCFG edges, and produces an over-approximation of the GCFG. Both methods can be combined to validate the results. Their composability is a topic of further research.

The main source of edges in the GCFG are interrupts. Therefore, an annotation by the developer at which point an interrupt cannot occur improves not only analysis run-time, but also the density of the GCFG.

The result of this chapter is a fully constructed GCFG, which may be an over-approximation. For each ABB a (maybe) fuzzy system state is given by both GCFG construction methods.

Chapter 4

System Construction

In Chapter 4, I introduced the *global control flow graph* (GCFG) and methods for its construction. For each node (ABB) in this graph a (fuzzy) system state was calculated. The analysis is independent of a concrete OSEK implementation and relies only on the OSEK specification. This chapter will investigate how to use the gathered information to manipulate the operating system for a given application to achieve better non-functional properties. This manipulation is done through an operating system generator, which first analyses the application and applies the construction methods afterwards. Because of its static design, OSEK is well suited for this generation process. The application is statically linked against the operating system and is therefore not intended for replacement; operating system and application form a firm bundle.

First, I will give describe a method that tailors the operating system very closely, on the level of single system call sites, to the application. The resulting system image has shorter paths through the kernel and generality is only employed where it is necessary. Afterwards two software-base dependability methods are presented. They are intended to improve the resilience against transient hardware faults. All presented methods will utilize the GCFG and the calculated system states to incorporate deep application knowledge.

4.1 Tailoring System Calls

The operating system is the internal interpreter of the system VM. All VM instructions, which are not propagated to the currently-running subtask VM but executed directly by the system VM, are interpreted by the operating system. The presented method for tailoring this interpreting engine will change the internal structure, but it will not change the behavior on the interfaces.

Normally, modern operating systems are decomposed into components that interact with each other, like it is shown in Figure 4.1. Whether this decomposition is done on a function or on an class/object level is not important here. I only assume that a composition of functionality is done, such that components are activated in sequence to achieve the functionality. Basic components can thereby be shared among the higher-level components. The functional programming paradigm is one method to decompose a system into interacting components. Therefore I assume functional decomposition here.

Several functions within the operating system act as gateways for the application into the kernel. These gateway functions, which are the system VM instructions, are the system calls (e.g., `ActivateTask`). All system-call invocations of the same kind result in a function call to the same function within the kernel. The ABBs calling `ActivateTask` use the same technical

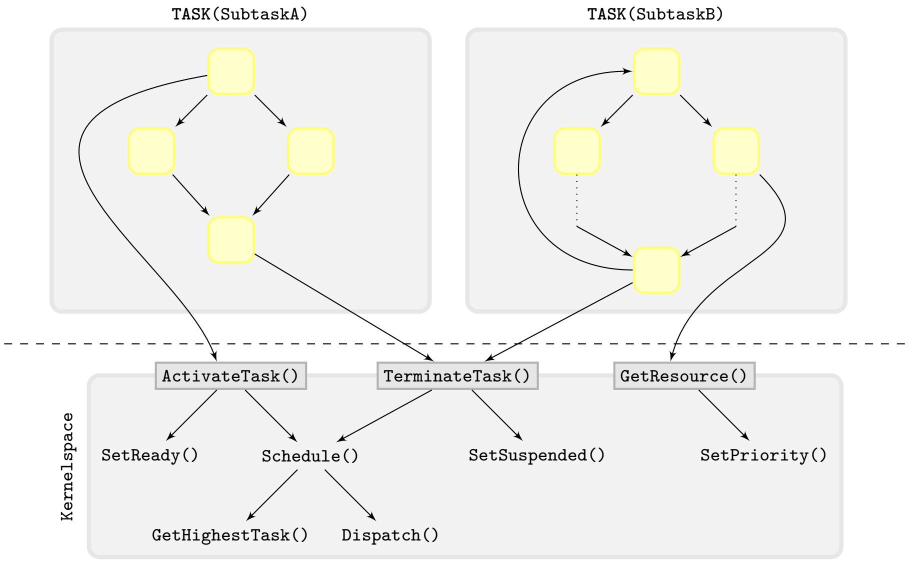


Figure 4.1 – Classical OS component activation graph. In classical operating systems, different subtasks invoke the same system calls with different parameters. The code for one type of system call resides in one distinct function, which activates other system functionality.

implementation for this system call. Each *call site*, for a given system call type, results in the same function activation. The system-call function itself uses various other system components to achieve the correct functionality in every possible situation. The system call and its component usage is implemented in a *generic* manner and decides upon the dynamic parameters and the dynamic system state, which is stored in memory, what actions have to be done to achieve the functionality.

This decomposition into components is done on two levels: The data that is used by the various components is separated into objects or structures, while the code is separated into functions or classes. The separation and reuse of code reduces the used resources for program memory and is often associated to maintainability of the source code. On the other side the decomposition and reuse of the data is necessary, since a shared state between all participants is essential for operation. The components communicate through these data structures.

The code decomposition is bought at the price of the *dictate of generality*. Since a component can be called by several parties in various situations, additional run-time checks have to be in place to dispatch dynamically upon the system state which action is appropriate. This additional code costs run time during a system call and influences non-functional properties, like the tolerance against transient hardware-faults. It also disallows to exploit static information of the system at a certain point. For example, we do not need to call the scheduler, when we activate a lower-priority subtask, since the system call will always return to the calling subtask.

4.1.1 System-Call Specialization

I will present the method of *system call specialization* to allow optimization of the operating system in the case where the quality of application knowledge differs from system-call site to system-call site. For this specialization the decomposition of the operating system is done in a different way: The kernel is no longer a set of components that share code, but it is a set of components that only share the state. This principle is depicted in Figure 4.2. We no longer build

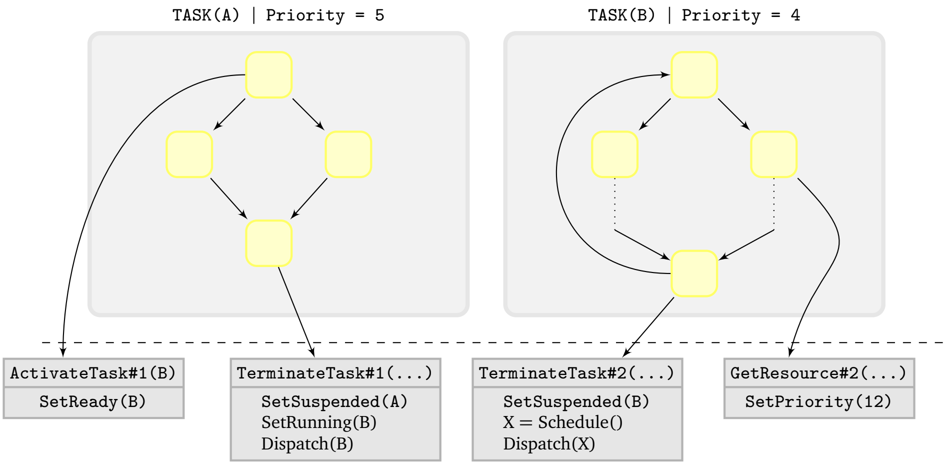


Figure 4.2 – System Call Decoupling and Specialization. Application knowledge enables specialized system calls, which are instantiated for each call site. The kernel itself dissolves and the functionality is unrolled and inlined.

components, which are connected through function calls, but we instantiate a function for *each* system call site.

In Figure 4.2, four system call sites are in place. For each call site, a component is instantiated, which is only called from one place in the application. Since this component is no longer forced under the dictate of generality, the statically derived system state from Chapter 3 can be used to tailor the system call. In case of the `ActivateTask#1` function, it is obvious that no rescheduling has to be done, since subtask A activates the lower priority subtask B. It has only to be marked that subtask B is now `READY`.

In the case of `TerminateTask#1`, it is for sure that subtask B is `READY`, since every control flow in subtask A calls `ActivateTask#1` beforehand. Therefore, no scheduling has to be done. But again, the mutable system state has to be adapted. Additionally, subtask B can be dispatched directly. In the `TerminateTask#2` system call the situation is not so clear and a general instantiation has to be in place. In the case of `GetResource` the dynamic priority of the subtask can be updated with a constant, since Limitation 5 allows the resource occupation and the dynamic priority to be computed beforehand for each ABB.

The specialized system calls in Figure 4.2 can share component code, but we can also force the compiler to inline all functions directly into the system call. This is a trade-off between decomposition and increasing code size. In the extreme case of inlining everything, some very interesting properties emerge:

1. Most arguments can be written down as constants in the specialized system calls. Therefore the compiler can use *constant folding* to remove unreachable code (e.g. unreachable branches) in this instantiation for these concrete arguments. This reduces memory accesses, executed instructions and run time.

2. By inlining all needed system functionality in the specialized system call, no call–ret machine cycle is needed, which may affect the dependability, since it avoids indirect jumps on memory when returning from a function.

3. Since the system call code is located in one coherent memory region, a *memory protection unit* (MPU) can be used to isolate the system calls from each other. One protection domain for each system call is formed.

4. The specialized and constant-folded system call might be small enough for inlining it into the application.

```
;; 0x10       = (1 << TaskID(B))
;; SetReady(B);
orl     ready_list, 0x10
```

Listing 8 – Machine Code example for `ActivateTask#1`. The `SetReady` in Figure 4.2 is condensed by constant folding and inlining to a single machine instruction that updates the systems ready list/bitmap.

For `ActivateTask#1`, the resulting system call can be compressed to a single machine instruction, that sets a `READY` bit in a bitmask (see Listing 8). This single instruction is functionally equivalent to the full-blown generic implementation that sets the subtask ready and does a full reschedule, including the following dispatch.

When we examine this system tailoring process on a more abstract level, the single system call site looses unneeded flexibility. In the real-time world, two main paradigms are used to construct real-time systems: the event-triggered and the time-triggered paradigm. In his PHD thesis, Scheler [44] used the ABB abstraction to translate from the first paradigm into to the second one. This is a full shift from a the dynamic event-triggered paradigm to a static time-triggered one. The system call specialization uses similar techniques, but takes the half way to a static system. Only situations where the flexibility is not needed are made static. The expressiveness and flexibility that the event-trigger paradigm provides remains.

4.1.2 Technical Implementation

To tailor the system calls, the call sites within the application have to be decoupled from each other. This decoupling is done by rewriting the system call sites in the compiled application. The compiled application is an object file or a compiler immediate representation. Within this pre-compiled application, the system calls are function calls to dangling symbols. In other OSEK systems, these dangling symbols are provided by the operating-system library, which is linked to get the system image.

In the ABB identification phase, we split up the basic block such that each system call resides in its own ABB. The generator rewrites those system call blocks to call a unique function, whose name is derived from the original system call name and from the ABB identifier. Thus, the application fragment has not the conventional system call names as dangling undefined symbols (like `ActivateTask()`) but one dangling reference for each system call site (e.g., `ActivateTask__ABB42()`).

For each dangling reference, the generator will instantiate a specialized system call. The generator uses a decision tree to select the components needed to fulfill the system specification. The generator decides at each edge upon the system state which subtree provides the needed functionality for a certain call site.

In Figure 4.3 a decision tree for the `ActivateTask` system call class is given. The tailoring process starts at the top. If no information is available for a system call site, the generic implementation is used; otherwise code for the `SetReady` and `Schedule` functionality is generated in sequence. Every time we do not have enough information to specialize, a generic implementation is used. Sometimes the system information allows us to omit no code at all, when the system

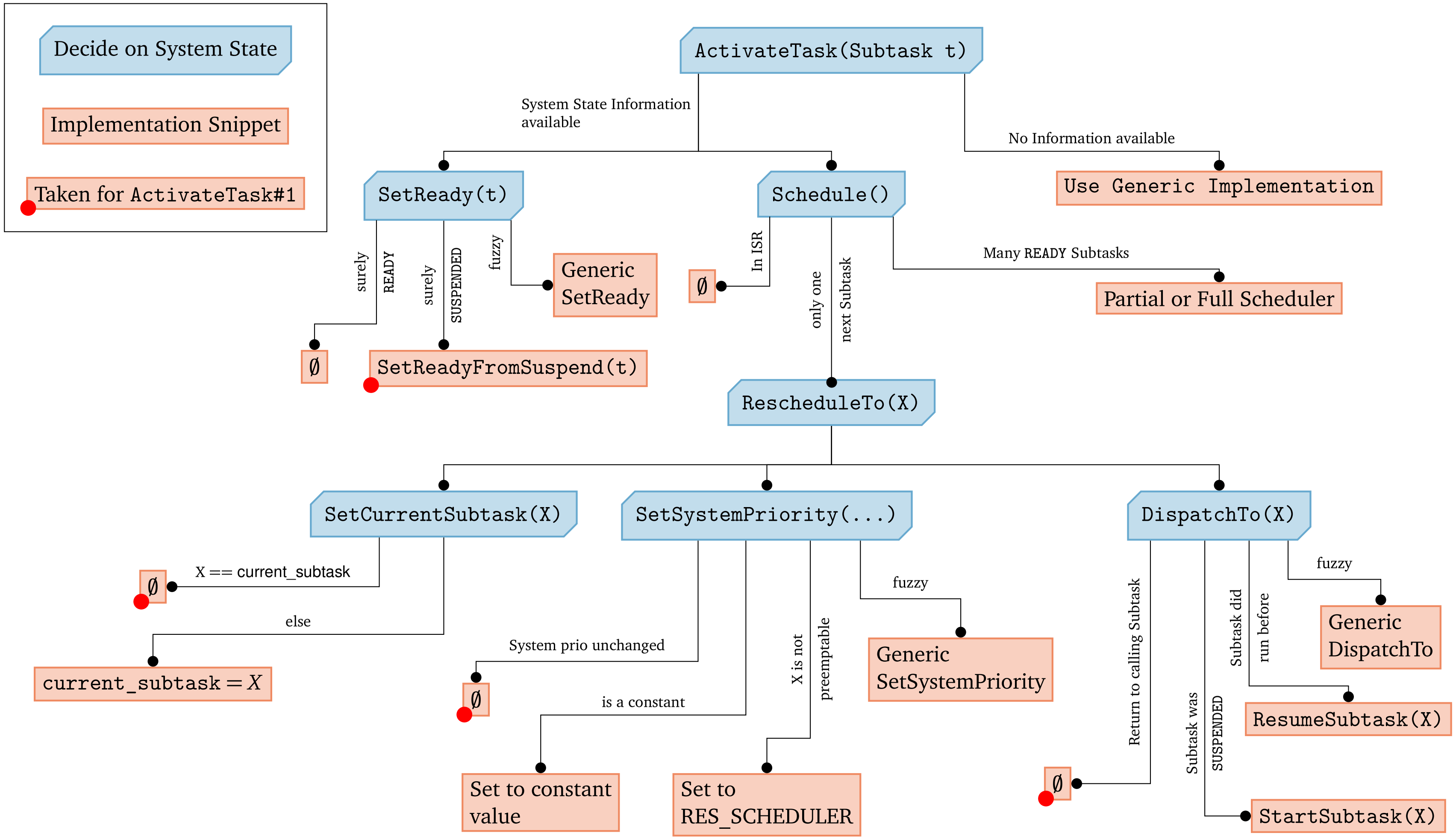


Figure 4.3 – Only one exit edge is taken for each node and all child nodes under horizontal bars are generated in sequence if the decision was taken. The implementation snippets, marked with red dots, are used in the generation process of `ActivateTask#1` from Figure 4.2.

state would not change. The implementation snippets that are collected for the `ActivateTask#1` example from Figure 4.2 are marked with red dots. Combined, they result in a single machine instruction (Listing 8).

The decision tree is optimized to emit dense code, if it is possible to determine the next running subtask exactly. But also if more than one subtask are possible scheduling outcomes, a *partial scheduler* is instantiated. While a full scheduler checks every subtask for its subtask_state, a partial scheduler is endowed with the information that some subtasks are *surely* not ready. In those cases, the scheduler does not have to check the surely-`SUSPENDED` subtasks. Not all scheduler types can benefit from this information. For example, a bitmap scheduler often uses a `get_highest_bit_in_word()` operation, which has constant run-time for a certain number of subtasks. Other scheduler types, like a multi-level queue scheduler, will benefit indeed from this information.

Another use-case for optimizing system calls is the `Schedule()` operation that has to be done when an interrupt changed the ready list and returns. Normally a generic scheduler has to be called, since we never know which task is running at the interrupt activation time. But from the GCFG we can deduce the ABBs which are successors of `iret` blocks. Since every ABB belongs exactly to one subtask we determine the set of possible next subtasks and instantiate a partial scheduler.

If the same system call is generated for multiple system call sites, code size could be saved by discarding identical copies. This kind of optimization is already done for C++ template instantiations and could also be applied here. But this micro optimization to save code size was not done for this thesis.

4.2 Assertions on the System State

Since all OSEK implementations have to fulfill the same specification, they are functional equivalent to a large extent. Therefore, the non-functional properties, like run time and code size, are target of optimization and research. Another non-functional property that gets more and more attention in the embedded world, is the resilience towards transient hardware faults. This resilience may obtained by specialized hardware, like expensive lock-step processors [4, 55], or with software-based hardware-fault–tolerance measures, like checksums over data structures [10]. Many software-based approaches, like *triple modular redundancy* (TMR), rely on the operating system as a minimal *reliable computing base* (RCB) [16].

The operating system is a program that is called only exceptionally. Therefore, its state resides passive in the main memory for long periods of time. It is used only in a bursty manner during system calls, which execute in a short time. Therefore, the protection of the OS state is a fruitful target for implementing dependability measures. One possibility is a checksum over the operating-system state to protect its integrity during the time the OS is not running. This can be done without using static application knowledge. I want to present a method for checking the constant part of the operating-system state before entering the kernel that is less complex in terms of code in the final product and also hardens the system call code.

The GCFG is accompanied by a set of system states. This information is exploited to generate assertions on the OS state when the kernel is entered or left. For this, the generator installs two hooks within each system call (see Figure 4.4): before the actual system call, a *SystemEnterHook* is installed; the *SystemLeaveHook* is installed before the system resumes to the subtask. Both hooks are part of the same critical region as the system call. These hooks are inserted per system call site. When we use system call tailoring, such hooks can easily be inserted, but they can also be instantiated by augmenting the system call sites. Again, these hooks are similar to explicit join points known from aspect-oriented programming [28].

For both hooks a system state is determined that summarizes our knowledge about the system at that point. Since parts of the system state are precise, the generator writes out asserts that check the OS state for having the correct value. For the entering hook, this state is directly extracted from the GCFG, since it was a direct result of the construction process. For the leave-hook we have to merge all before-states of all successors in the GCFG of the system call ABB.

Since the state transformation, done by a system call, generates precise information, the side-effect of the system call is checked by the assertions in the leave hook. The assertions have also a flow control monitoring semantic: When the application issues an unexpected system call as a result of a fault, the asserts may catch this fault, since the system is not in the expected state for the system call.

The state when a system call is invoked and before it returns may be totally different. This originates from the fact that system calls are preemption points. Figure 4.4 shows a system with two system calls, which are placed within different subtasks (`T1` and `T2`). Both system calls have an own enter and leave hook and the GCFG execution flow visits them in a sequence that is not directly obvious: `SystemEnterHook#1` (❶) and `Syscall#1` (❷) are visited after each other. The executing task T1 is preempted, since T2 got ready. At some point after the preemption, T2 issues the `TerminateTask` (ABB_2) system call; `SystemEnterHook#2` (❸) and `Syscall#2` (❹) are executed. T2 terminates and therefore the OS dispatches back to the first subtask and the remaining `SystemLeaveHook#1` (❺) is executed.

As we see, the enter hook and the leave hook of one "physical" system call are executed at some very distinct points in time and totally different system states can be in place there. This is due to the fact, that these hooks are executed as part of the system call and therefore on the system VM level. When calculating the system states, which are base for the checking, we have to take the system VM execution trace into account.

The observation that leave-hooks are related to enter-hooks of different system calls can be used to reduce the number of assertions without loosing much of their benefits. System calls execute for a very short time, compared to the long time spent in the application. Therefore, we assume that each static fact has to be checked only once during a single kernel activation. In the Figure 4.4 example, the points ❸, ❹, and ❺ are within one kernel activation. But ❶ and ❷ belong to another kernel region. We remove those assertions from the leave hook that are surely

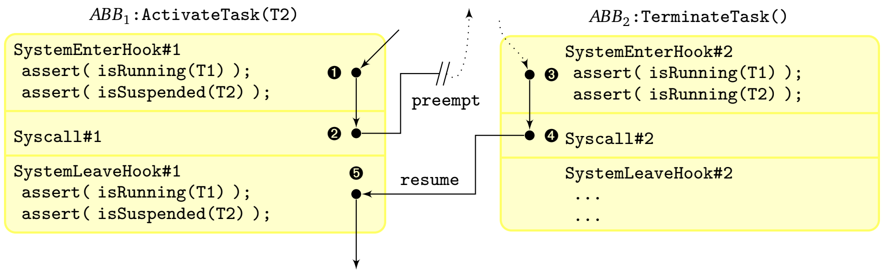


Figure 4.4 – For constant assertions upon the system state, all system calls are augmented with an enter and a leave hook. When a system call results in a preemption, the enter and the leave hook are executed independently on the timing axis.

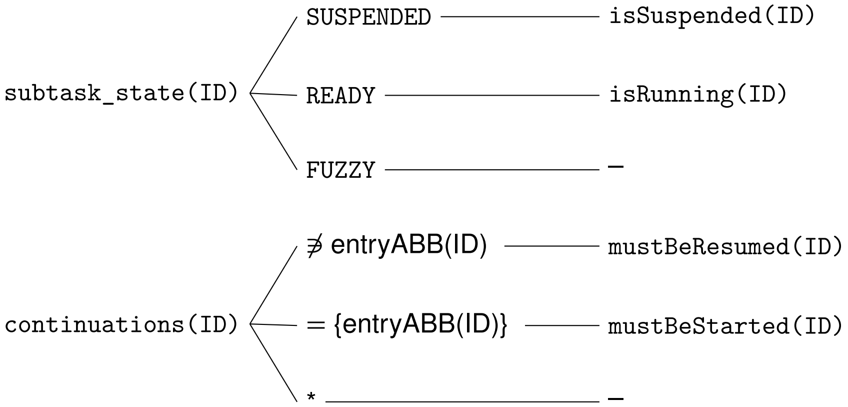
done within the same kernel activation. For this we do the intersection of all assertions done in the enter-hooks that may lead to the current leave-hook.

In the example, `SystemLeaveHook#1` has only one system enter hook that proceeds to it (`SyscallEnterHook#2`). The surely-done assertion set for this leave hook is therefore:

`{isRunning(T1), isRunning(T2)}`

Therefore, we remove the `isRunning(T1)` assertion from the leave hook. This optimization does not prevent the assertions from protecting the OS state during long phases of a dormant OS. And it does not remove the check whether the system call had the correct side-effect, since data the system call touched does not have the same value as before. In the example, we can guess only from looking at the pre- and post-conditions that `Syscall#2` involves a subtask termination.

Since the assertion code has constant arguments (e.g., the constant subtask id) the compiler generates very efficient code for the implemented assertion types. Four types of static assertions are generated from the system state:



The `isRunning(ID)` and `isSuspended(ID)` assertions are directly derived from the corresponding subtask_state system state. Whether a subtask was already started and is currently preempted or whether it is `READY`, but was not exectued before, is deduced from continuations. If the set consists only of the subtask's entry ABB, the subtask was surely not running since its last termination. If it does not contain this block, the subtask is surely to be resumed. If it contains more than one ABB, but also the entry ABB, we have not enough knowledge for inserting an assertion.

The static assertions on the system state utilize the state information that is a result of the GCFG construction process to detect transient hardware-faults. They exploit the fact that the application invokes the system calls in a certain pattern and the system state is therefore well-known at certain points.

4.3 Control-Flow Monitoring

The system state assertions have already a control flow monitoring semantic by accident. The constant parts of the system state may be different for different places within the GCFG. Therefore, a faulty jump to a location where the constant system state part differs results in a detected error. With the GCFG information at hand, we can do even further control flow monitoring with a small overhead.

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

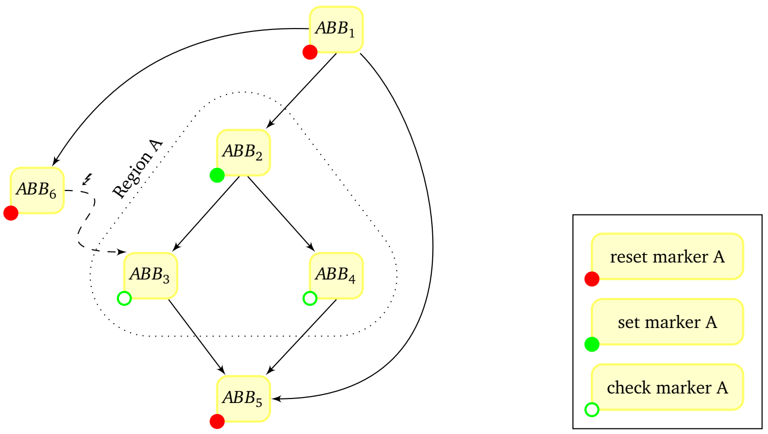


Figure 4.5 – Control Flow Regions Used for Fault Detection. Region A sets a marker A, which must be present in all blocks of that region. The marker is reset in all blocks outside the region. When a faulty control flow jumps into the region, the marker is not present and the fault is detected

ABBs that belong to this region. All blocks outside the region reset the marker again. If a marker is not present within the region, it is sure that the region was not entered through the root ABB; a fault must have occurred. Figure 4.5 shows an example of a control-flow region that sets a marker A in ABB_2, which is checked within the rest of the region. If a hardware fault leads to a jump from ABB_6 directly into the region, the marker is not present and the mechanism detects 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 *dominator analysis* [1, S. 602] provides exactly that semantic.

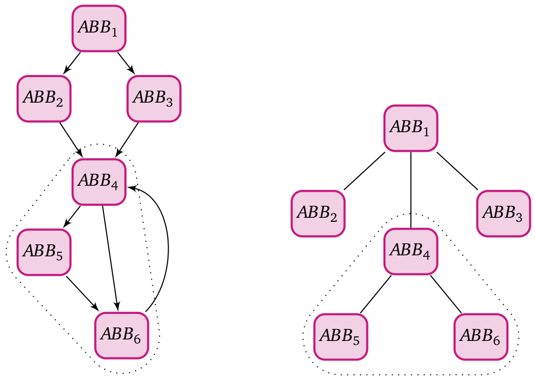
Definition 7 (Dominator and dominates-relation)**.** *In a flow graph, one block A dominates block B, iff all possible paths through the graph originiating from the initial node visit block A before block B. A is a dominator of B.*

More colloquial, there is no "sneak-around" block A when executing B. Every block dominates itself and only a *strict* dominance relation implies that A and B must be distinct nodes. Additionally the concept of *immediate dominators* is widely used to express the dominance relationship as a *dominator tree* [1, p. 602].

Definition 8 (Immediate dominator and dominator tree)**.** *Each block B has an unique immediate dominator A that is the last dominator in any path from the initial node to B. In the dominator tree, B is a direct child of A.*

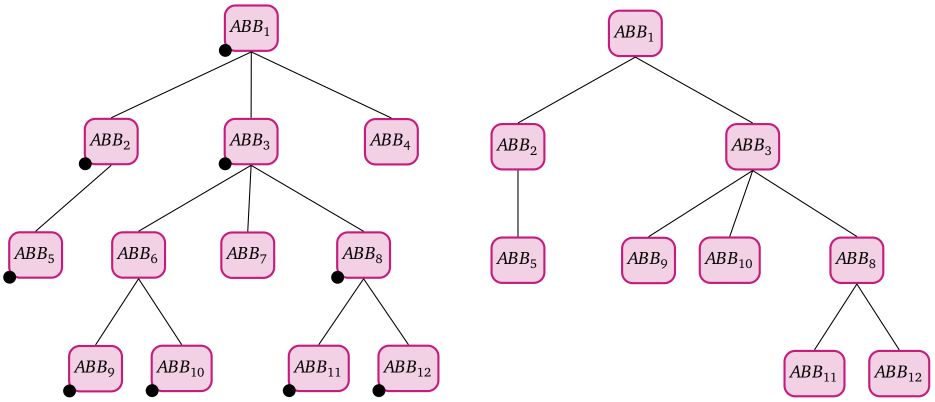
Figure 4.6 shows an example of a control-flow graph and its dominator tree. In the dominator tree each node is the immediate dominator of its direct children and it dominates the whole subtree strictly. There are several algorithms with different complexities that are used for dominator analysis. Allen formulated the dominance computation as a global data-flow analysis with $O(N^2)$ complexity [3]. In university courses, often the Lengauer-Tarjan [26] algorithm with complexity $O(E * \log N)$ is taught, although it is slower on real flow graphs than an iterative scheme [14].

Since the GCFG is a flow graph, a dominator tree that contains the system's ABBs can be constructed. Some nodes are computation blocks and some contain system calls. But since we are specializing only the operating system, the generator can only influence the system call blocks.

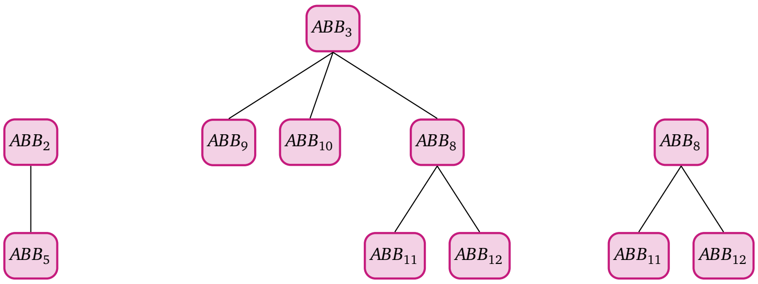


(a) Control Flow Graph **(b)** Dominator Tree

Figure 4.6 – Control Flow Graph and Dominator Tree. Each node in the tree is the immediate dominator of its direct children and it dominates the whole subtree. The marked area is a dominator region with ABB_4 as the dominator.



(a) Dominator Tree with System Calls (●) **(b)** System Call Dominator Tree



(c) Subregion (ABB_8) **(d)** Subregion (ABB_3) **(e)** Subregion (ABB_8)

Figure 4.7 – System Call Dominator Tree. All subtrees of the SCDT with a size greater than one and smaller than the whole system are interesting regions for control-flow monitoring.

Therefore, we condense the dominator tree to the *system call dominator tree* (SCDT), such that it only contains the system call blocks. The SCDT expresses the dominance relationship among all system call sites in the application. The SCDT has the `StartOS` block as a root node and expresses the dominance relationship in the system-call invocation trace.

From the SCDT, the candidates for control flow regions, monitored by markers, are calculated. All subtrees in the SCDT that are not equal to the SCDT and are non-trivial (contain more than one node) are candidates (see Figure 4.7). The subtrees' root node is called the *region leader* of the CFG region.

```
// Enter regions by setting markers
markers = markers | 8;
// Check the presence of region markers
if ((markers & 0xf8) != 0xf8 )
  signal_fault();
// Reset markers
markers = markers & (~(4 | 2));
```

(1) Emitted C code

```
;; Enter regions by setting markers
  mov    eax, markers
  or     eax, 8
;; Check the presence of region markers
  mov    ecx,eax
  and    ecx, 0xF8
  cmp    ecx, 0xF8
  je     L1
  ud2    ;; signal_fault()
L1:
  ;; Reset markers
  and    eax, 0xfffffff8
  mov    markers, eax
```

(2) Compiled IA32 instructionse

Listing 9 – The OS generator emits efficient code within each system call site that sets, checks, and clears markers. The bitmask values are known at compile-time.

For the candidate subtrees, the OS generator emits setting-, checking- and marker-resetting operations into each system call site. They can, for example, be placed in the enter hook. The operations are implemented as bit operations on a marker word and the number of monitored regions is limited to the bit width of the marker word. For a fine-grained monitoring the regions are sorted by size and the smallest regions are selected. Each region is assigned one bit in the marker word. When implementing the marker operations as bit operations, the overhead in terms of RAM is the marker word and in terms of code is a constant-length code block for each system call site.

The compiler emits very efficient machine code, since the region information is constant for each system call ABB. The generated C code and a compiled IA32 assembler program is shown in Listing 9. The enter operation sets at most a single bit, since each ABB is at most the leader of one region. In all blocks within the region, the markers for the region must be present. As third step, the region markers are resetted, when the block is not member of the region.

The described approach is a positive test. The check can only trigger, if the execution is actually within the region, but the marker is not present. The case that the control flow jumps outside the region cannot be caught. One approach, which was not implemented by me but is worth mentioning, uses the *dominance frontier* [15]. The theoretical approach will be presented in Section 7.2.

The control flow monitoring presented in this chapter tracks the execution irrespectively of system call and subtask borders. It utilizes the control flow semantic of the GCFG and integrates massively application knowledge.

4.4 Summary

In this chapter, I presented three different approaches that are used in an operating-system generator. All three methods utilize the in-depth analysis of the application behavior from Chapter 3 to influence non-function system properties on a fine-grained level. This influence changes only the implementation of the system VM, but not its external behavior.

In the system call tailoring process, the system call sites are decoupled by instantiating a kernel fragment for each call site. Since this system call instance is not forced to work in all situations, but only for a single call site, a high potential for tailoring system calls is released. The kernel code for the system call instance can be fully inlined to get a loop-free and call-free code region.

Besides the code size and run time, also resilience against transient hardware faults becomes more and more important. Therefore, I presented two software-based reliability measures that utilize the GCFG information. The first method inserts assertions on constant parts of the system states before and after each system call sites. Additional to memory corruptions, they also detect certain types of control-flow errors.

The third approach incorporates even more control-flow monitoring into the system. I used dominance regions within the GCFG to detect even more control-flow error classes by setting markers at the entry of a dominator region. The marker must be present in every node of the control-flow region.

All three methods have in common that they utilize application knowledge, which was not given through the system description file (OIL), but take also the application's inner structure into account.

Chapter 5

Evaluation

In the previous chapters, we have developed techniques for analyzing the interaction between the application and the underlying operating system, as well as strategies for utilizing the gathered information for optimizing non functional properties of the real-time system. In this chapter, I will investigate on the characteristics of the GCFG construction, as well as the influence of the system construction on the non-functional system properties.

After the presentation of the evaluation scenarios, the two core techniques for constructing the GCFG are examined. The *symbolic system execution* (SSE) and the *system-state flow* (SSF) analysis are compared in terms of run time and the quality of the resulting GCFG. For this comparison, I will define several metrics to quantify these dimensions of the approach.

The system-construction techniques in Chapter 4 were mainly developed for increasing the system dependability against transient hardware faults. Therefore, an extensive fault-injection campaign covering the whole effective fault space is undertaken on the resulting system image. But also the influence other non functional system properties, like code size and run time, are measured.

5.1 Evaluation Scenarios

The presented analyzes and techniques are all part of the *d*OSEK operating-system generator. *d*OSEK is an OSEK-like system generator that was constructed with dependability in mind and is a result of the DanceOS research project[3]. *d*OSEK is configurable in regard to the techniques that are applied during the system generation and is therefore well-suited for measuring the impact of the analyzes. As applications, the extensive *d*OSEK test suite and the *I4Copter* benchmark are used.

5.1.1 *d*OSEK: A Generator for Dependable Operating Systems

The *d*OSEK [19, 32] system generator aims to be a RCB for real-time applications that is more resilient against hardware-faults than other OSEK implementations. *d*OSEK's design is based upon two pillars: strict fault avoidance, which is done mostly by indirection avoidance, and reliable fault-detection techniques. Therefore, the techniques presented in Chapter 4 are a perfect match for both categories. The system-call specialization reduces the number of useless instructions

---

[3] `http://www.danceos.org`

that are executed during a system call, and therefore reduces “indirection”. The system-state assertions and the control-flow monitoring incorporates additional fault-detection mechanisms.

Besides the standard technique of spatial isolation, which is nowerdays state of the art when designing a real-time system, *d*OSEK also incorporates the possibility to enrich the kernel execution with additional redundancy. For this, parity bits are used, as well as an ANB-encoded scheduler [32, p. 38]. This scheduler exposes a very low number of *silent data corruptions* (SDCs) at the cost of a high overhead in terms of code size and run time.

*d*OSEK includes the technique of decoupling different system call sites, like discussed in Section 4.1, but the spatial isolation and the encoded operation are configurable. As a result of this generation technique, *d*OSEK cannot be shipped as a library. Other real-time operating systems can be configured with the rough system parameters (e.g., number of subtasks) and compiled into a kernel library, which can be linked against a manifold of application that fits the preconfigured system parameters. *d*OSEK tailors the kernel exactly to the behavior of the application, they are inseparable afterwards.

For the evaluation, I will use *d*OSEK with spatial isolation. In *d*OSEK, this spatial isolation is implemented on the IA-32 architecture by utilizing the memory-protection unit. Lukas [31] describes in his master thesis, how *d*OSEK uses static page tables to mimic the behavior of a MPU. *d*OSEK does this mimicry, since MPUs are more common in embedded system than MMUs. Table 5.1 gives a quick overview about the used *d*OSEK features.

5.1.2 Applications

For the evaluation, several applications that run on top of *d*OSEK were chosen. These applications have different properties and were either developed for testing the operating system or were designed as an evaluation scenario.

5.1.2.1 *d*OSEK Test Suite

*d*OSEK comes with an extensive test-suite consisting of 52 test applications. The developers use the test-suite actively in the development process of *d*OSEK and each test case is a complete OSEK application. When a developer submits a change into the code reviewing process, which is done with the gerrit [4] tool, a continuous integration process is started: Each test application is

	unencoded *d*OSEK	encoded *d*OSEK
Decoupling System Calls	●	●
Fully Inlined System Calls	●	●
System-Call Argument Inlining	●	●
Memory Protection	●	●
ANB-protected Scheduler		●
ANB-protected Counters		●
Protected Dispatcher		●
Parity for saved Stackpointers		●
Parity for saved Return-addresses		●

Table 5.1 – *d*OSEK Variant Overview. For the evaluation, two variants of *d*OSEK are used, an unencoded and unprotected version and the fully enriched version.

---

[4] `https://code.google.com/p/gerrit`

processed by the generator and the resulting system image is executed automatically within a virtual machine, where it must produce a specific activation trace.

The *d*OSEK developers built the test suite with the goal of having test cases that are not overly complicated, while covering as many corner cases of the OSEK specification as possible. Because of this, the test-suite is not usable for evaluating the general precision and the general applicability of the approach. But it is indeed, well-suited for comparing the two approaches to GCFG construction relative to each other. And it is well-suited to observe properties of the methods, when keeping in mind that the test-suite consists of precise and not overly complicated applications.

ISRS Tasks	0	1	2
1		alarm1a(22), isr2a(22),	
2	sse1a(27), resource1a(31), resource1c(33),	alarm1c(22), alarm1b(24), alarm1f(24), isr2b(24), sysmodel-1alarm-a(30), sysmodel-1alarm-aa(30), alarm3a(34), resource2a(34), alarm3b(38), alarm3d(38), alarm3e(39), alarm3c(41),	complex1a(27), complex1b(27), complex1d(27),
3	task1a(21), task1a-sse(21), task2a(21), task2c(21), task1f(22), task1e(23), task1c(24), task2b(24), task1b(25), task1g(26), task1d(28), resource1b(33), resource1k(35), resource1g(37), resource1d(41),	isr2d(22), alarm1d(24), isr2c(27), alarm1e(30), complex2a(32), resource2b(38),	complex1c(27), alarm2a(31), alarm2b(34), alarm2c(35),
4	sse1b(36), resource1h(38), sse1c(40), resource1j(42),	lukas-alarmstress(29),	
5	resource1e(37), resource1f(39),		

Table 5.2 – Matrix of *d*OSEK Test-Cases. The 52 test-cases developed along with *d*OSEK are grouped by the subtask and ISR count. The number of ABBs is given in parenthesis.

The test cases in *d*OSEK have different complexities (see Table 5.2) and are grouped by the number of subtasks that are involved and the number of ISRs that can fire. The more subtasks (1-5 in *d*OSEK) and ISRs (0-2 in *d*OSEK) are declared, the more complex is the application's outer structure. Additionally, the more ABBs are needed to partition the ICFGs the more complex is application's inner structure. Therefore, when test-cases are referenced in the future, the format `Name(Subtasks, ISRs, ABBs)` is used, like resource2b(3, 1, 38). If a short form is desired, the format `Name(ABBs)` is used, like resource2b(38).

5.1.2.2 *I4Copter*: A Realistic Work-Load Scenario

The *I4Copter* [50] was already used in previous work by Hoffmann et al. [20] and Lukas [32] as a realistic workload scenario for fault-injection experiments. Since the *I4Copter* is a flying quad-rotor helicopter, it is a real-world safety-critical application.

As illustrated in Figure 5.1, the *I4Copter* consists of several interacting components. A sampling task is activated by an periodic event every 30 milliseconds. The sampling task, which

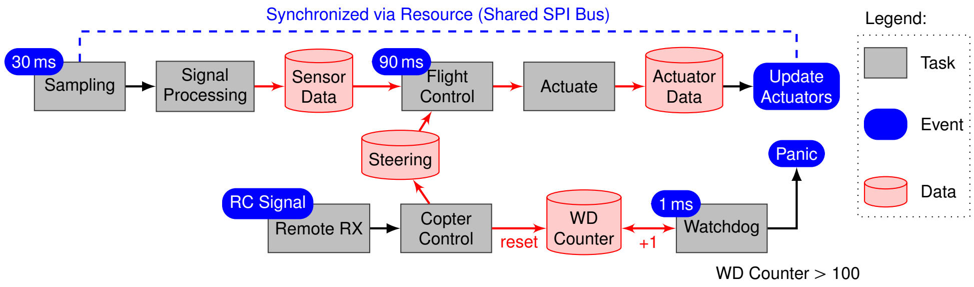


Figure 5.1 – *I4Copter* Subtask Constellation. Simplified representation of the *I4Copter* task and resource constellation, resembling a real-world safety-critical embedded system. The system acts a common workload scenario for all generated *d*OSEK variants. (taken from Hoffmann et al. [20]; modified)

includes the signal processing, is implemented with an OSEK alarm and five interacting subtasks. Every 90 milliseconds, another periodic event is activated to schedule the flight-control task, which calculates the new actuator values. The flight-control task is implemented with 4 subtasks and one activating OSEK alarm. In order to measure sensor values with as little jitter as possible, all subtasks of the sampling task have a higher priority, than the other subtasks.

A remote control signal activates the interaction with the outside world. It releases a copter-control task, which provides data for the flight-control task and resets a watchdog timer. The copter-control task is implemented with a single subtask and a single ISR2. The watchdog task is a single subtask that would be activated if a OSEK alarm expires, which is regularly reseted by the copter-control task.

In the end, the *I4Copter* system is implemented with eleven subtasks, three alarms, one ISR, and one OSEK resource. Since I am interested in the interaction of the application with the kernel, only the subtask setup is used as a benchmark. The application code is replaced with a checkpoint function. It is the OS' job to ensure the same checkpoint call sequence under all circumstances. To limit the benchmark's run time, it executes for three hyperperiods, which results in 172 checkpoint activations.

Since the *I4Copter* benchmark was not intended to give a small and easy to comprehend GCFG, but to resemble a real-world scenario, I will use it to perceive general properties of the system analysis. Because of it's long run time, it also has more timer interrupts than the test-suite benchmarks.

5.2 System Analysis

Since the system analysis is done at compile time, its run time is not as critical as the quality of the resulting GCFG. Nevertheless, the SSE has an inherent exponential run time, caused by the exponential number of possible system states. Therefore, I will do an qualitative evaluation of the analysis' run-time, as well as a quantitative measurement of the analysis' result.

5.2.1 Run Time of the System Analysis

When doing an run time evaluation, it is necessary to find a meaningful metric. Of course, the most plausible choice is to measure actual time elapsed during the analysis. But this metric has some caveats. First of all, the actual run time is highly dependent on the technical implementation of the algorithms, but not necessarily on the quality and limits of the underlying concepts. Secondly, the actual run time is hard to measure on modern multi-core processors operated with preemptive general-purpose operating systems. This is especially true for benchmarks, that run only for a very limited amount of time.

Therefore, I think it is useful to find a metric that is independent of those technical issues, but nevertheless reflects the quality of the concepts in regard to computation requirements. For the system analysis, I identified the number of *copied system states* as such a metric. Using this metric is plausible, because of the following reasons: The core of both analyses is the `system_semantic()` function, which hides all the specification details of the OSEK standard and maps one input state to multiple output states. It consists of the system call semantic, which only manipulates a few bits in the system state. For a specific application, the system state is a constant-sized data-structure, and therefore the actual run-time of the `ABB_transformation()` is over-approximated by a small constant. Additionally, copy operations are easy to measure by counting the calls to the copy function.

The fixed size of system state and the ICFG graphs permits us to over-approximate the state manipulations of the block scheduler and the virtual dispatcher with a constant. Nevertheless, the `system_semantic()` function has to create several new system states as an output. Therefore, I measure the number of copied system states for each `system_semantic()` invocation. Each output state has to be emitted by the block scheduler and to be fixed up by the virtual dispatcher. In the next step, it has also to manipulated by the `ABB_transformation()`. Therefore, the number of copied system states represents a quantitative metric of the run time of both analysis' concepts.

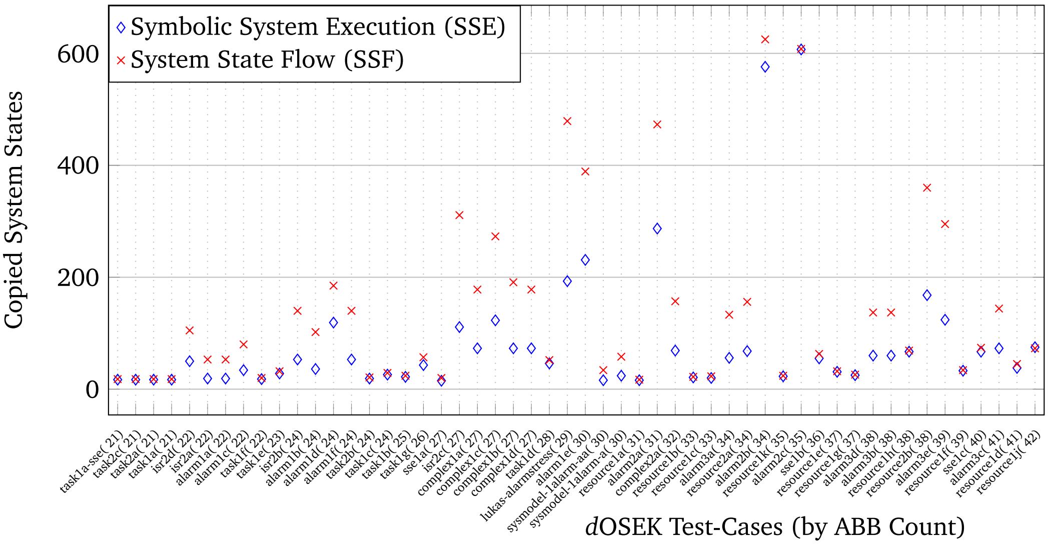


Figure 5.2 – Copied System States for SSE/SSF. The number of system state copy operations for SSE is smaller than for SSF for the small applications from the *d*OSEK test-suite.

For the *d*OSEK test-suite, the number of copied system states is given in Figure 5.2. The test-cases are sorted by the number of ABBs in system-relevant functions. It is surprising that the SSF needs an overall higher number of copy operations. But a closer inspection makes this result plausible: When annotating the number of external events in the test case, like it is done in Figure 5.3, we observe that the number of copied system states is especially higher for SSF, when external events have to be handled by the data-flow analysis.

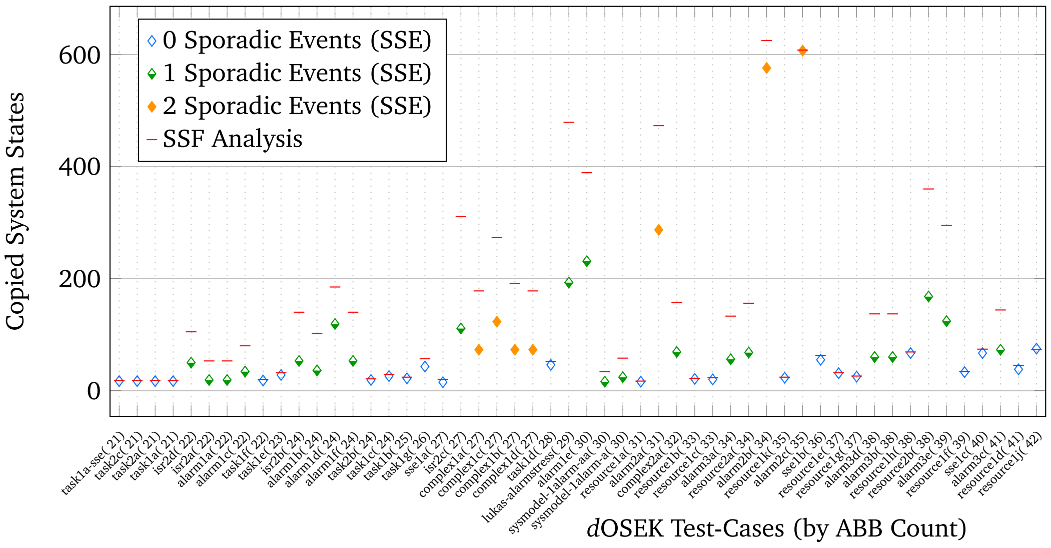


Figure 5.3 – Copied System States by ISR count. For SSE, the number of copy operations can increase dramatically by adding more ISRs.

The SSF analysis handles external events (ISRs and alarms) by spawning a new fix-point analysis for the ISR function. This is done for each computation block and for each external event. For small applications, this seems to be more expensive than enumerating the whole system-state space. But additional external events inflict also a high penalty on the SSE, which has to calculate the whole system-state sub graph that is caused by the external event.

For the *I4Copter* benchmark, two scenarios are in place, when evaluating the analyses: We can start the analysis without any additional information about task containers. Or we can apply the no-reactivation–annotation principle from Section 3.4. The no-reactivation annotation will cut down the number of external event activations that can occur. For a large system, like the *I4Copter*, we expect a higher impact of the annotation on the SSE.

		Copied States	Possible States	Run-Time (in s)
SSE	(w/o ann.)	1,907,313	1,579,971	431.8
	(w/ ann.)	15,329	13,473	1.85
SSF	(w/o ann.)	6,547	n/a	1.47
	(w/ ann.)	4,208	n/a	0.87

Table 5.3 – Run-Time Evaluation for analyzing the *I4Copter*. The copied states column gives the number state copy operations that were needed for the analysis. The possible states column gives the size of the resulting system state graph and the run-time for the analysis is given in seconds.

The results for analyzing the *I4Copter* benchmark are summarized in Table 5.3. Without the annotation, the SSE must copy over 1.9 million system states, while it is running for 431.8 seconds. Compared to this, the SSF is much faster in terms of copied system states as well as in terms of actually elapsed time. It must copy only 6,547 states and has only 0.34 percent of the SSE's run time.

When incorporating additional information, like it was described for the *I4Copter* in Section 5.1.2.2, the SSE analysis improves significantly. The number of copied system states drops down to 15,329, which is only 0.8 percent of the original value. The impact on the SSF is with −35.73 percent also measurable, but not as impressive as for SSE. In this large system, the limiting of external events mitigates the IRQ problematic. This mitigation has less impact on the SSE, since spawning a new SSF analysis for each external event activation takes only a constant amount of time and does not open a new branch in the system-state graph. Here, the exponentially of the SSE analysis surfaces.

Since the *I4Copter* has four external events (3 ISRs and 1 alarm), it is clear why the annotation has such a huge impact on the SSE. Additionally, the *I4Copter* application consists of 105 ABBs, which adds up to the high run-time of the SSE, since the currently running ABB is a distinguishing part of the system state.

For the *I4Copter*, I presented the run-time in seconds only to give an impression of the current implementation and not to give an absolute metric of the problem. The current implementation is done in a scripting language[5] and was not optimized for speed, but only the number of copy operations was kept as low as possible to give none of the implementations a penalty caused by the implementation. This claim, that the number of system-state copies is kept low, can be shown for the SSE, when we compare the number of copied system states with the number of system states in the resulting state-graph, which is a lower bound for the copy operations.

In Table 5.3, the possible system states column gives this lower bound for the number of copy operations. The number of unnecessary generated system states is with 1.21× (1.14× with annotation) very close to the lower bound for the SSE. For the SSF, I cannot give such a precise lower bound.

5.2.2 System-State Precision

The more we know for sure about a system, the more optimization and additional checks we can apply to our system. But how can we measure the degree of knowledge we have about a system? We have to find a objective measure that enables us to compare the quality of the analyses results, regardless of the size of the application. This objective measure has to reflect the precision or fuzziness of the system states, which are generated by SSE or SSF. Therefore, I introduce the metric of "system-state precision".

Definition 9 (System-state precision). *The precision of a system state is a number in the interval* $[0, 1]$*, that gives the amount of knowledge we have about a system state. A fully precise system state has a precision of* 1.0*. If we do not know anything about the system state, it has a precision of* 0.0*.*

The two main parts of the system state, we are interested in are the subtask_state and the continuations set for each subtask. For the subtask_state, we score every subtask that is surely `READY` or surely `SUSPENDED`:

$$\texttt{st\_score(state, subtask)} = \begin{cases} 1 & if\, \texttt{subtask\_state(state, subtask)} = \texttt{READY} \\ 1 & if\, \texttt{subtask\_state(state, subtask)} = \texttt{SUSPENDED} \\ 0 & if\, \texttt{subtask\_state(state, subtask)} = \texttt{FUZZY} \end{cases}$$

For the continuations, we have no knowledge if the subtask can be resumed in every ABB of the ICFG. Therefore, the size of the subtask's ICFG ($ICFG_T$) is an upper bound. If we have perfect knowledge, there is exactly one continuation point for each subtask. Therefore we use a linear function that compares both numbers and has $[0, 1]$ as an image:

$$\texttt{ct\_score(state, subtask)} = \frac{|ICFG_{\texttt{subtask}}| - |\texttt{continuations(state, subtask)}|}{|ICFG_{\texttt{subtask}}| - 1}$$

---

[5] Python 3.4.0

We weight both scores equally important and normalize it by the number of subtasks that are declared within the system. This normalization ensures that we can compare systems of different size. The result is a precision metric, within $[0, 1]$, for each system state. The precision of a whole system is the average over all system states that are calculated by SSE or SSF.

$$\texttt{state\_precision(S)} = \frac{0.5}{\texttt{\#Subtasks}} \cdot \sum_{T \in Subtasks} (\texttt{st\_score(S, T)} + \texttt{ct\_score(S, T)})$$

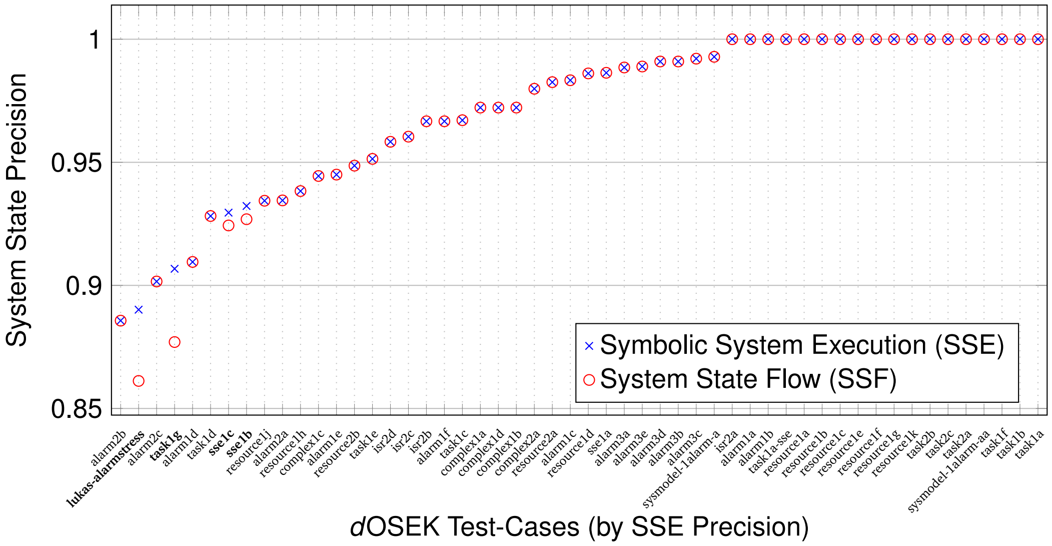


Figure 5.4 – System State Precision for SSE/SSF. The precision of the SSE state precision is *always* greater or equal to the SSF state precision.

For the *d*OSEK test-suite, Figure 5.4 summarizes the precision results for both SSE and SSF. Since the test-suite is designed for clarity, it has an overall high precision and SSE and SSF have an equal precision most of the time. In some cases, the SSF analysis reaches a smaller precision than the SSE analysis. A closer look reveals that these test-cases included merging of system states at function call and branches. Since those merge operations are done in the SSE analysis after the whole system state space has been calculated, they have no negative effect during the analysis. On the other hand, the SSF analysis merges states after each step, and therefore propagates a fuzzy system state immediately in the data-flow analysis.

A closer look on the precision results reveals the high influence of external events (ISRs and alarms) on the system state precision. In Figure 5.5 the precision results of the SSE analysis are shown again, but the number of external events is additionally annotated. It is remarkable that almost all test-cases with external events do not reach full precision. Also a higher number of external events results in a lower precision. The only exception to this is the sysmodel-1alarm-aa test case. This test case has a no-reactivation annotation. This annotation assists the analysis in calculating a perfectly precise result, although external events are in place.

Table 5.4 shows the system state precision for the *I4Copter* benchmark. Again, the benchmark is evaluated with both analysis methods and with/without no-reactivation annotation. We observe that the annotation boosts the precision of SSE more than the precision of SSF. The higher benefit of the SSE analysis is in accordance with the results of the artificial *d*OSEK test-suite. But it is remarkable, that SSF with annotation reaches a higher precision than SSE without annotation. This exemplifies the impact of giving the application developer more control about possible interrupt occurrences.

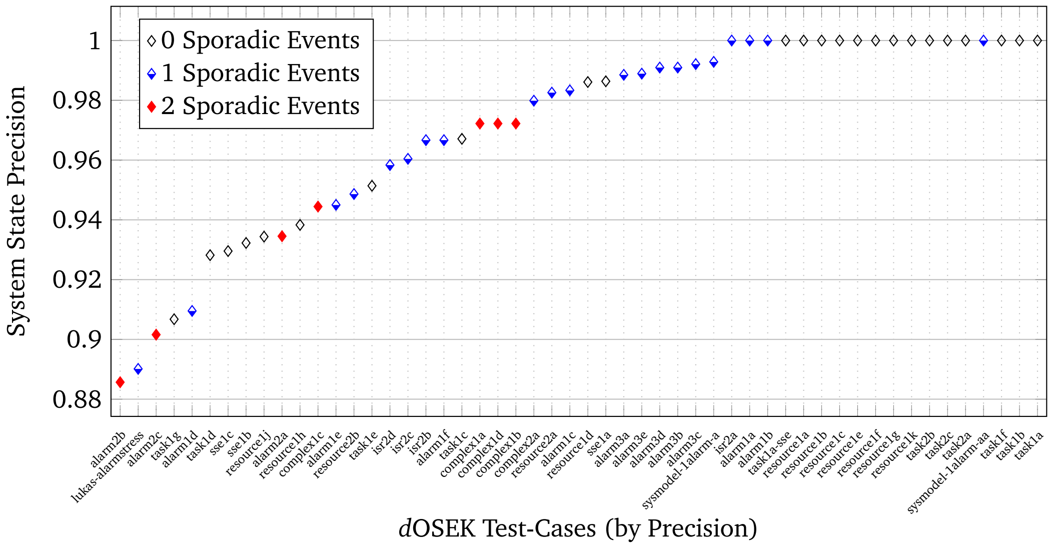


Figure 5.5 – System State Precision by ISR count. A high amount of external events has an high impact on the system state precision.

		System State Precision
SSE	(w/o ann.)	0.83
	(w/ ann.)	0.87
SSF	(w/o ann.)	0.82
	(w/ ann.)	0.84

Table 5.4 – System State Precision for *I4Copter*

5.2.3 Comparing the GCFG

The precision of system states for different analysis methods describes how good the different approaches work out. But it does not set the general GCFG approach into context to other ways of incorporating static application knowledge. I will outline this topic only briefly and I do not claim any form of completeness here. But I want to give a hint what metrics are useful, when we compare the quality of static application knowledge.

As discussed in Section 3.1, the main task of an OSEK system is to execute work-packages form different subtasks on a single processor. The system VM is active only “in between” two work packages. Therefore, the degree of application knowledge can be measured in terms of possible transitions. Since these transitions are the external manifestation of the internal operation, they can used to compare different approaches for describing the knowledge about the observable behavior of an operating system. For the GCFG, the number of possible transitions is the number of edges within the GCFG. The less edges the GCFG contains, the more we know about the external behavior of the operating system.

The most trivial case to compare against the GCFG is the approach that ignores the application knowledge completely. Each ABB (work package) can transition to every other ABB, even to itself. Therefore the number of transitions is quadratic in the number of ABBs: $(\#ABBs)^2$. Although, this way of utilizing the application knowledge seems too naïve, it is state-of-the-art, when a library OS is linked against the application. Figure 5.6 compares the size of the GCFG relative against the complete graph. The lower the mark is on the y-axis, the more we know about the system compared to the full graph. In the worst case, the GCFG has 5 percent of the size of the

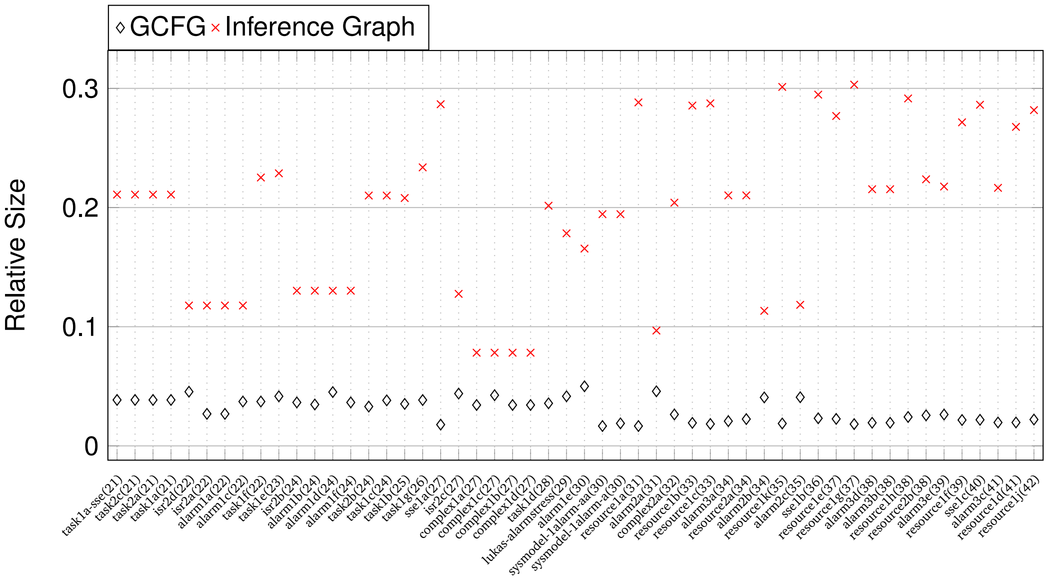


Figure 5.6 – Relative Graph Sizes. All numbers compare to the complete graph, which must be assumed when no application knowledge is incorporated. When using the static priority information, only those transitions to a ABB with higher or equal priority have to be considered in the inference-graph. For the GCFG only the edges that are present are counted.

complete graph; in the best case the GCFG has 1.66 percent of the complete graph's size. So, the GCFG eliminates more than 95 percent of the possible transitions.

Barthelmann [5] used an interference graph to bring additional application knowledge into the compiler. The inference graph contains an directed edge to every ABB that can preempt the currently running ABB. These transitions point always from low-priority ABBs to high-priority ABBs. Barthelmann used the inference graph to allocate the processor registers smartly among the subtasks. By this, he optimized the number of registers that have to be saved during the context switch. Along with the normal ICFG edges, these upwards-edges over-approximate all ABB–ABB transitions that can be drawn when using only information from the OIL file.

In the best case, the inference graph has only 7.82 percent of the complete graph. In the worst case, the inference graph has 30.31 percent of the complete graph's size. But nevertheless, for these test-cases the inference graph has always more possible transitions than the GCFG. Even in the best case the inference graph has 1.56 times the size the GCFG has.

In the *I4Copter* benchmark, the GCFG has 1.67 (1.51 with annotation) percent of the size of the complete graph. Compared to that, the inference graph has a relative size of 26.45 percent. This amount of additional knowledge is not used in currently employed OSEK generators.

5.3 System Construction

In Chapter 4, three approaches for optimizing the non functional properties of the system images were given. Of course, these methods have not only an impact on the desired non functional properties, but also on others. In this section, I will quantify the influence on the non-functional properties code size and run time.

5.3.1 Code Size of the Kernel Fragments

In *d*OSEK, each system call site is decoupled from the other call sites of the same system call type. Therefore, the size of the kernel increases with the number of system-call sites. Since each system-call instance is one contingent code region, we can measure the needed code size for each instance exactly.

In order to get a higher number of system call instances, I gathered all system calls from the *d*OSEK test-suite and from the *I4Copter* benchmark. For the comparison, I selected the three system calls `ActivateTask`, `TerminateTask`, and `ChainTask`. Since these system calls normally call the scheduler, and they produce the largest instances in terms of code size when doing aggressive inlining.

For the comparison, *d*OSEK is configured to produce an IA-32 binary that runs on the bare machine and spatial isolation is turned off. It is built once as a robust variant with the encoded operations, and once as a normal variant *without* the encoded operations.

Along with the baseline (either encoded or unencoded), four combinations of applied methods are evaluated:

Baseline This variant is the normal *d*OSEK variant that does *not* utilize application knowledge related to the GCFG.

+opt Like the baseline variant, but the system calls are specialized according to the GCFG and the calculated system states, like it is described in Section 4.1.1.

+ass Like the baseline variant, but system state asserts are in place for each system call site, like it is described in Section 4.2.

+flow Like the baseline variant, but control-flow monitoring based on dominator regions is added to the system, like it is described in Section 4.3.

+all Like the baseline variant, but the generator applied all three of the above methods to the system.

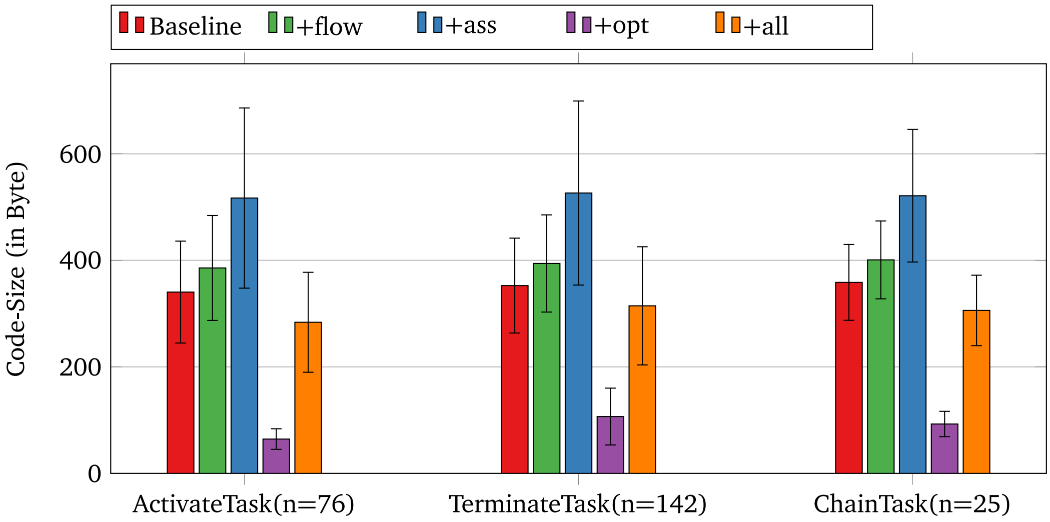


Figure 5.7 – System Call Codesize for an *unencoded* *d*OSEK. For each evaluated optimization, the influence on the code-size differs from system call to system call.

Figure 5.7 shows the average system-call size in bytes for the three selected system-call types in case of an unencoded *d*OSEK. Without any applied methods, the baseline variant needs a similar amount of code for the three system calls. Over all (not only the three selected) system calls, the code size is 191±179 bytes. The high standard deviation originates from the fact that some system calls include a scheduler, and are therefore larger.

The control-flow monitoring (+flow) adds up the least overhead to the system call (AT: +45.28 b, TT: +41.54 b, CT: +42.28 b)[6]. A closer evaluation shows that the code-size overhead for this measure is constant and the deviation is only caused by the compiler's code-alignment strategy.

The system-state–assert method adds a higher code overhead to the kernel per system call (AT: +176.51 b, TT: +173.76 b, CT: +162.72 b). Surprisingly, the overhead is quite homogeneous among the different system-call types, although the number of asserts is depending on the location of the system call in the GCFG.

The influence of the system-call specialization is of special interest. It decreases the size of the kernel fragments significantly (AT: −81.06 %, TT: −69.72 %, CT: −74.13 %). Particularly noteworthy is the standard deviation of the `ActivateTask` system calls. With ±19 bytes, they show a low standard deviation. This is caused by the fact that an `ActivateTask` never includes a scheduler! Since it does neither change the priority of the calling subtask nor of the activated subtask, the priorities of both subtasks are fixed. Therefore the scheduling decision can always be pre-calculated and only a dispatcher is necessary. This dispatcher dispatches either to the calling or the activated subtask.

Relative to the specialized system, the other two measures applied show a code-size increase (AT: +219.22 b, TT: +207.85 b, CT: +213.2 b) that is the composition of the single measures. The resulting system (+all) is in average (over all system calls) 13.8 percent smaller than the baseline system. So we accomplished to construct kernels with additional dependability measures, which are smaller in terms of code size given a system with call-site decoupling..

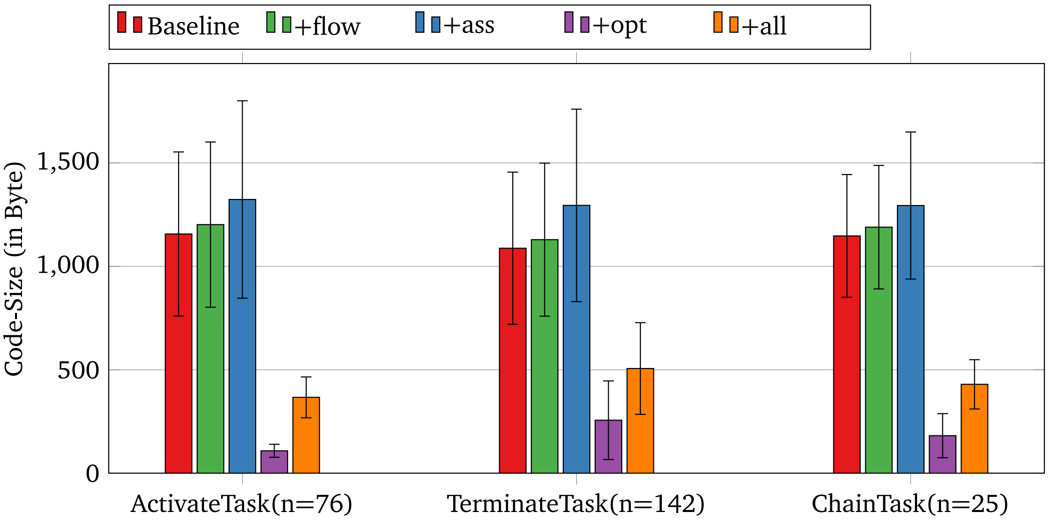


Figure 5.8 – System Call Codesize for an *encoded* *d*OSEK. For the encoded variant, the influence of specialized system calls (+opt) is higher than for the unencoded variant.

For the *encoded* variant of *d*OSEK, Figure 5.8 reveals similar results. Caused by the encoded operations, the baseline variant has a higher code size per system call from the beginning.

---

[6] AT = `ActivateTask`, TT = `TerminateTask`, CT = `ChainTask`

Therefore, the relative overhead of the control-flow monitoring is quite small (AT: +3.92 %, TT: +3.82 %, CT: +3.69 %). The absolute overhead (AT: +45.38 b, TT: +41.56 b, CT: +42.28 b) is nearly equal to the unencoded *d*OSEK. The system-state asserts have an absolute overhead (AT: +166.82 b, TT: +207.24 b, CT: +146.92 b) that is in the same range as for the unencoded *d*OSEK.

For the encoded variant, the system-call specialization has an immense effect (AT: −90.63 %, TT: −76.44 %, CT: −84.18 %). This is caused by the fact that the specialization cuts out parts of the scheduler, which is especially costly in the encoded case. Over all system calls, the specialization decreases the size of system calls by 82.53 percent. Here, the specialized systems with the additional dependability measures applied have only 50.97 percent of the original system-call sizes.

Since the quality of the system-call specialization is dependent on the quality of the system-state information, it is fruitful to investigate on the relation between code size and the precision of the system-state information for a specific system-call instance. Figure 5.9 shows this detailed examination for the unencoded *d*OSEK, Figure 5.10 depicts it for the encoded *d*OSEK.

Since the system calls originate from different systems with different subtask counts, the scattering is quite high. Therefore, a sliding average with a window-size of 0.02 is drawn additionally into the graphs to get a feeling for a general trend.

It is remarkable, that the specialized `ActivateTask` (Figure 5.9a, +opt) shows a nearly straight line. This is the effect of having always a precise scheduling outcome for this type of system call. So, even when the system state has a low precision the scheduling outcome is sure, and therefore no scheduler but only a dispatching operation is included. Since in *d*OSEK, the size of the dispatcher is not dependent on the number of subtasks in the system, the sliding average shows a nearly plain line. This plain line appears also in the encoded *d*OSEK (Figure 5.10a, +opt).

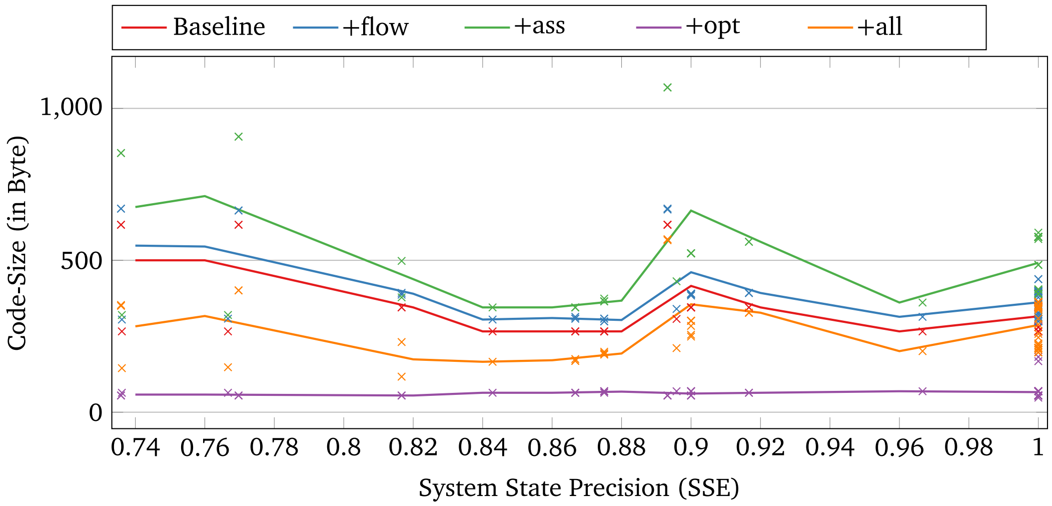
For `TerminateTask`, the specialized system calls show a general downwards trend for both the unencoded and the encoded *d*OSEK. So, a more precise system state results in smaller kernel fragments. This is caused by the partial scheduler strategy the encoded scheduler can benefit from.

For the two dependability measures that add code (+ass, +flow), the average follows the baseline trend. Nevertheless, it is remarkable, that the gap between the baseline and the system-state assert average increases with a higher precision. This is caused by the fact, that more precise system states result in more system-state asserts, since more parts of the system state are invariant.

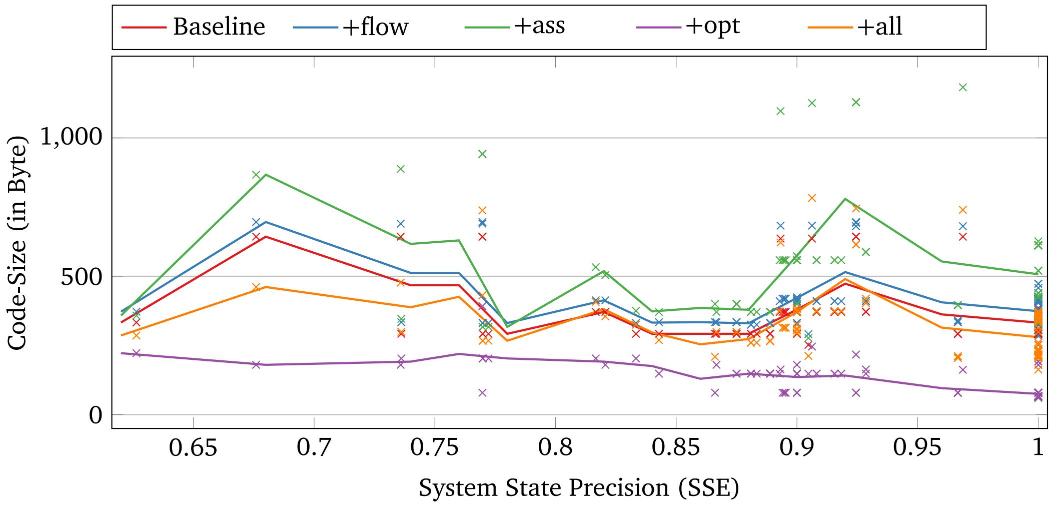
Again, we can observe that the system-call specialization has a higher influence on the encoded system than on the unencoded system. This originates from the higher code-size overhead the single components have; the implementation leafs in the specialization decision tree (see Figure 4.3) have a higher cost.

5.3.2 Run Time of the Kernel Fragments

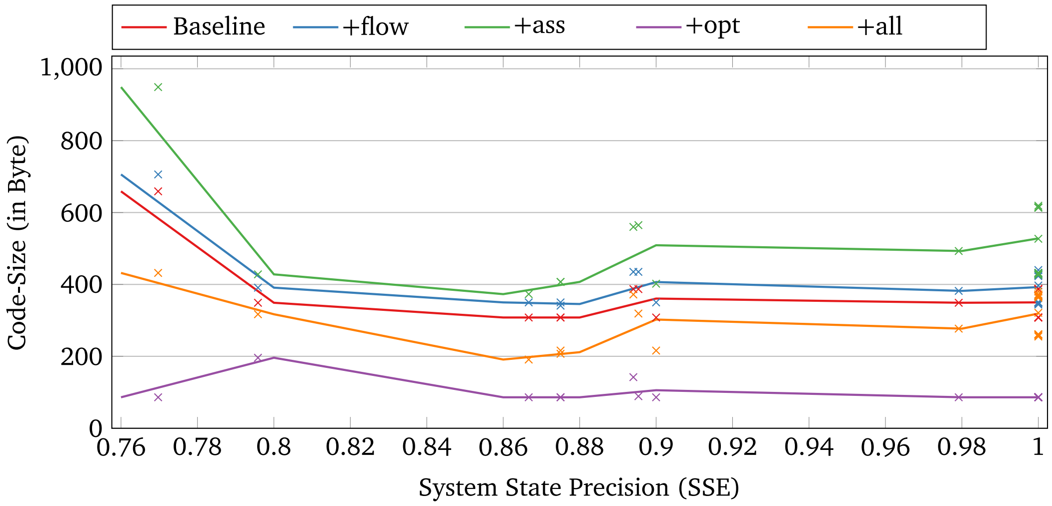
The applied measures have not only an impact on the code size of the kernel fragments, but also on the time that a system call needs to execute. In order to quantify the impact of the applied measures, I executed the different systems (test-suite and *I4Copter*) in an IA-32 emulator. The emulator is instructed to record an execution trace of the whole application. Within the



(a) `ActivateTask()` System Calls

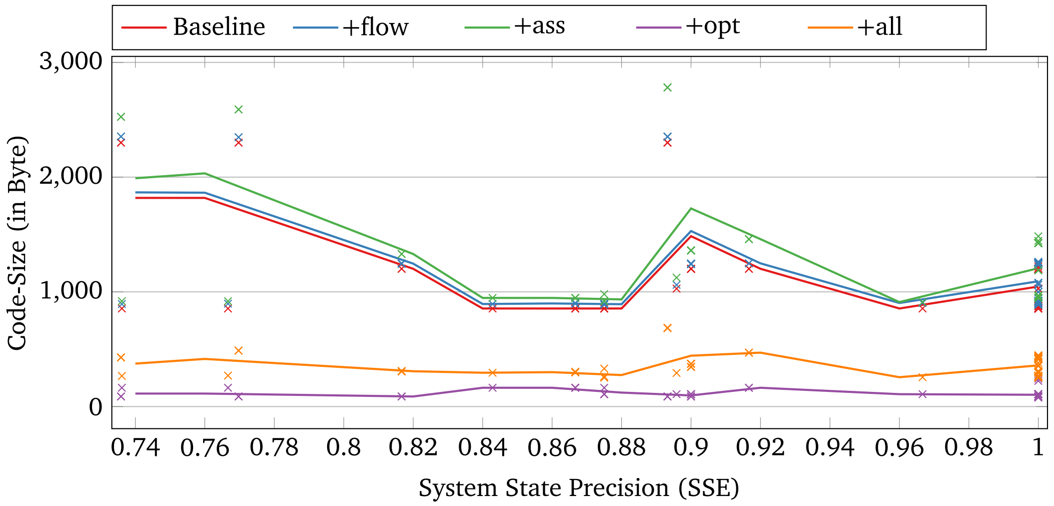


(b) `TerminateTask()` System Calls

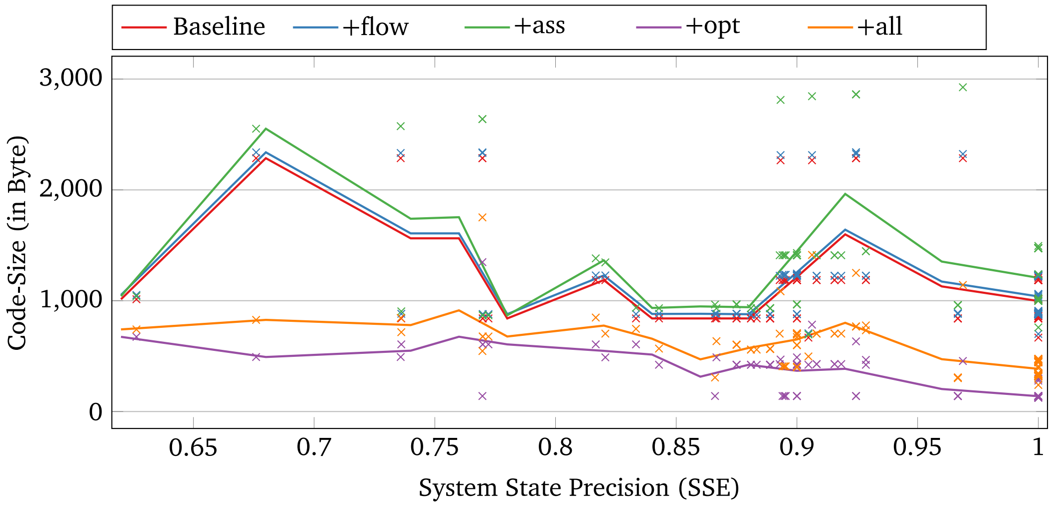


(c) `ChainTask()` System Calls

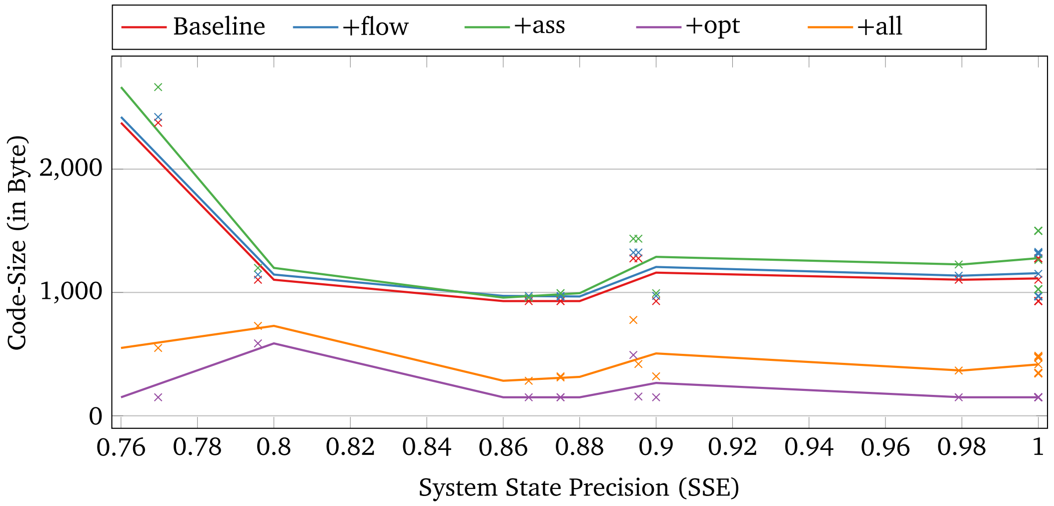
Figure 5.9 – System-Call Codesize for an *unencoded* dOSEK (Detail). In dOSEK, a system call is instantiated for each call site. The size of those kernel fragments is given in relation to the system state precision of the system call site. The straight line is a sliding average with window-size 0.02.



(a) `ActivateTask()` System Calls



(b) `TerminateTask()` System Calls



(c) `ChainTask()` System Calls

Figure 5.10 – System-Call Codesize for an *encoded* dOSEK (Detail). In dOSEK, a system call is instantiated for each call site. The size of those kernel fragments is given in relation to the system state precision of the system call site. The straight line is a sliding average with window-size 0.02.

execution trace, all system call activations are identified automatically and connected to a specific system-call instance. Therefore, we can connect each kernel activation to a system-call type.

For the run-time quantification of a system call activation, I count the processor instructions. I took this simple approach, although the number of executed instructions is not necessarily linearly correlated to the amount of time the actual execution needs on the real hardware. Vendor-specific hardware optimization, like caching, branch prediction, and out-of-order execution, cause a divergence here. Because of this, the real execution times are highly dependent on the specific hardware version.

On the other hand, counting instructions depends only on the instruction-set architecture and can easily be compared. Also in our case, the influence of the micro-architecture should be negligible. The kernel is activated only for a short amount of time and in *d*OSEK it is always entered by a system trap. Therefore caching and pipelining effects have a very limited effect on the actual run time. Additionally, many advanced processor features, like out-of-order execution, are not yet available for the embedded platforms, which are mainly used for real-time systems.

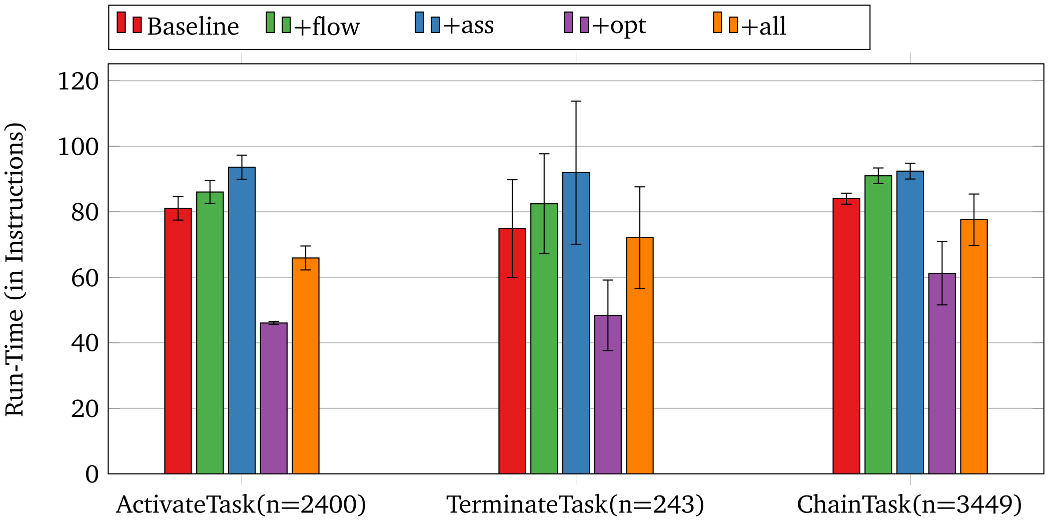


Figure 5.11 – System Call Runtime for an *unencoded d*OSEK. In all *d*OSEK test-cases and the *I4Copter* benchmark system call activations were recorded and their run-time in executed instructions was measured.

For the unencoded *d*OSEK variant, the average instruction counts for all activations in the *d*OSEK test-suite and the *I4Copter* benchmark are given in Figure 5.11. Figure 5.12 depicts the encoded *d*OSEK variant.

For the unencoded case, the control-flow monitoring adds about the same amount of instructions (AT: +5, TT: +7.57, CT: +6.99) as for the encoded variant (AT: +5.68, TT: +7.55, CT: +6.99). Over all system calls, the run time increases in average by 6.2 instructions. Consequently, control-flow monitoring with dominator regions can be implemented with less than 10 instructions executed per system-call activation.

When only the system-state assert (+ass) measure is applied, the overhead for the unencoded variant (AT: +12.58, TT: +17.07, CT: +8.4) is in the same range as for the encoded case (AT: +10.74, TT: +27.27, CT: +9.88). Of course, no real upper bound for the run time can be given, when the number of inserted asserts is unbounded. In average, the system-state asserts cause 10.57 instructions to be executed additionally in each kernel activation.

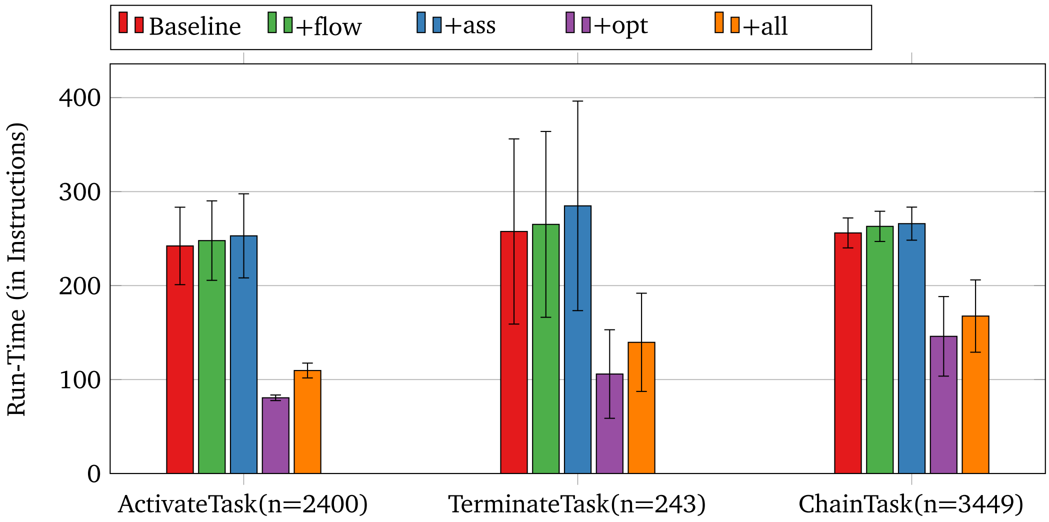


Figure 5.12 – System Call Runtime for an *encoded* *d*OSEK. Compared to the unencoded variant, the system-call specialization has a much higher impact on the encoded *d*OSEK.

In all variants, the impact of the system-call specialization (+opt) is significant. Over all system calls, the unencoded system gets 33.38 percent faster, while the encoded variant event gets 52.74 percent faster. Especially noteworthy is the impact on the `ActivateTask` system call: Because of the very linear execution, caused by having only a single dispatching call, the standard deviation is quite small (unencoded: ±0.42 instructions, encoded: ±2.98 instructions).

Of special interest is the comparison between baseline and the variant with all three measures activated (+all). In all cases, we can push the run time under the original run time, although additional code was introduced into the kernel. The impact on the run time is not as high as for the code size, but this can easily be explained: the specialization cuts out code regions that have no influence on the system state, and code pieces that never will be executed in a specific place. When a code region is never executed, erasing it influences the code size but not the run time.

When considering all system calls, system-call specialization (+opt) pushes down the run-time overhead of the encoded operations to +43.23 percent compared to the unencoded baseline variant. Without specializtion, the overhead of the encoded variant is +203.06 percent. The system-call specialization massively eases the overhead of the encoded operation.

Comparing the unencoded baseline kernel to the encoded fully-modified (+all) kernel, the run-time overhead of the systems (AT: +35.26 %, TT: +86.42 %, CT: +99.44 %) is much smaller than for the unmodified encoded variant (AT: +198.87 %, TT: +243.97 %, CT: +204.76 %). Over all system calls, the encoded, fully-modifed kernel remains with an overhead of 73.98 percent compared to the totally unprotected and unmodified *d*OSEK.

5.4 Fault-Injection Campaign

The third and most interesting category, which I want to evaluate the presented methods in, is the robustness against soft errors. Soft errors become more and more a topic of interest in the embedded community, since shrinking structure sizes and high clock frequencies increase the probability of transient hardware faults [11].

For the robustness analysis, I choose the realistic *I4Copter* benchmark as the evaluation scenario. It was augmented with combinations of the three methods. With the resulting system image I undertook an extensive fault-injection campaign. For the fault-injection campaign, the system image is run in a test environment. The test-environment executes the system for a given amount of time; injects a bit flip and resumes the execution until the scenario is completed or the system crashed. This process of injecting single-bit flips is repeated until all possible faults were injected.

5.4.1 System and Fault Model

For the fault-injection campaign it is essential to define the system and fault model precisely. I will use a single-event, single-bit fault model, which can for example be caused by radiation or voltage fluctuations. These effects, which can have an impact on all important parts of a system, are assumed on the instruction-set architecture level. So, I assume single-bit flips to occur in the general-purpose registers, the flags register, the instruction pointer, and the main memory [6].

As a system model, I assume the presence of reliable *read-only memory* (ROM). In this ROM all code and constant data is stored. As recent research shows the reliability of ROM is much higher than for volatile main memory and increases even with the technological progress [22]. Additionally, I demand an MPU-like hardware protection, which is implemented on the IA-32 architecture in *d*OSEK by using an MMU.

Now, that we have defined what faults can happen and on which hardware they occur, I will describe what tools were for the fault-injection. For the fault-injection campaign, I used the FAIL* [2] framework. FAIL* provides various test-platforms, including the BOCHS [25] emulator for the IA-32 architecture.

FAIL* works in multiple phases: First, the system is run without any disturbance and a golden run is recorded. For this golden run, the simulator is stopped after each instruction and the instruction pointer is stored as well as the addresses of all accessed memory locations. Secondly, FAIL* analyzes the golden run and selects the fault that has to be simulated. Each fault is described by the time the fault occurs, the fault location, and which fault pattern should be applied. In the third and last phase, FAIL* starts, for each possible fault event, the simulator again and stops the simulation at the time of the desired fault. In the halted simulator, FAIL* injects the fault pattern into the fault location and proceeds the system until the normal execution finishes, an error occurred, or a timeout expires.

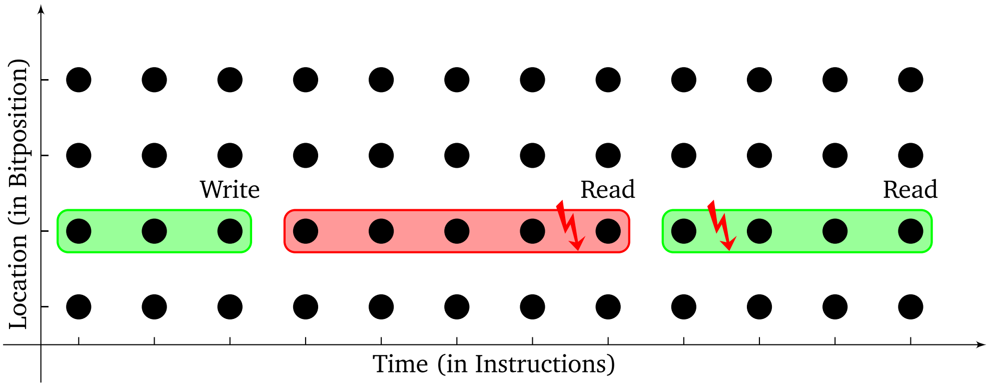


Figure 5.13 – FAIL*'s Fault Space Model. Each dot is a possible fault. In every time step a fault can occur in every bit. To cut down the complexity, the fault space is partitioned into equivalence classes. For each fault within an equivalence class, the system will show the same behavior.

FAIL* provides also a good model how to quantify the vulnerability of a given system-under-test. In Figure 5.13, this model is visualized. FAIL* assumes a uniformly distributed occurrence of faults. So, for every bit and for every time step a fault can occur. For the *I4Copter* benchmark, this fault space consists of $4.45 \cdot 10^{11}$ faults. Of course, this fault space is too huge to simply execute every possible fault. Therefore, FAIL* partitions the fault space into equivalence classes. Within each equivalence class that consists of multiple possible faults each fault leads to the same experiment outcome.

In Figure 5.13, three equivalence classes are show. Before each write operation to a bit at a given time, a fault will be benign, since the applied fault pattern is overwritten. The second equivalence class, which consists of five possible faults, ends with a read operation. A fault within this equivalence class results in an observable error. The third equivalence class extends from the instruction after the last read operation to the next read operation. A fault within this equivalence class results in no observable error. So, for these three classes, FAIL* orders two experiments for the read operations and will report 5 errors and 7 faults without any effect. For all *I4Copter* variants, FAIL* executed $2.55 \cdot 10^7$ experiments.

The fault-injection outcomes can be partitioned into three coarse categories: the fault that showed no effect; the fault that resulted in a trap, a timeout or an error-hook; and the *silent data corruptions* (SDCs). The error-hook category is less dangerous, since the operating system can detect those conditions and react on them, for example, with a *fail-stop* semantic. For silent data corruptions, as the name implies, such a detection did not trigger. For the *I4Copter* scenario, I define two situations as a silent data corruption: When the operating system could not achieve the correct checkpoint-activation sequence, which was described in Section 5.1.2.2, or when the application data is corrupted by the operating system. In both cases, the operating system did not fulfill its purpose and could not detect the fault. So, the main purpose of any dependability measure is to push down the absolute SDC rate.

5.4.2 Applying a Single Measure

In order to quantify the influence of each described measure (control-flow monitoring, system-state asserts, and system-call specialization) on the base system, I compare the SDC rate of the base system to the variant that has exactly one measure applied.

Figure 5.14 shows the SDC rates for the unencoded *d*OSEK, while Figure 5.15 shows the result for the encoded *d*OSEK. For the unencoded *d*OSEK I use a logarithmic y-axis, since the memory fault location is far higher than the other fault locations. For both cases, the baseline numbers are comparable to the numbers described by Lukas [32]. For the unencoded baseline the number improved slightly from $1,794.01 \cdot 10^6$ to $1{,}404.79 \cdot 10^6$. This decrease is due to the fact that the *d*OSEK developers improved the unencoded variant. Nevertheless, the numbers stay in the same order of magnitude.

All three measures improve the baseline (unencoded and encoded), in all cases. The least improvement is gained by the system-call specialization. There, only a −1.62 percent improvement can be measured in the unencoded *d*OSEK; −3.25 percent for the encoded *d*OSEK. The low improvement rate for the specialization was surprising, but can be explained by the fact that mainly code regions, which are never executed, are removed by the specialization. The system-call specialization was designed particularly for decreasing code size and run time.

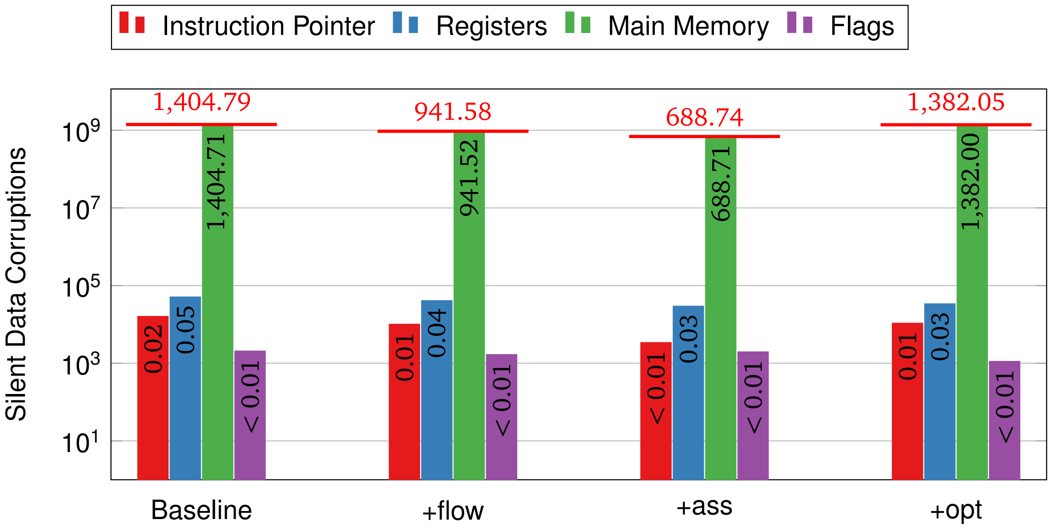


Figure 5.14 – SDC Rates for the *unencoded* *d*OSEK. The majority of the SDCs stems from fault that were injected into the main memory. The horizontal bar gives the sum of all SDCs for each variant. Because of the high variance in the SDC rates a logarithmic axis is used.

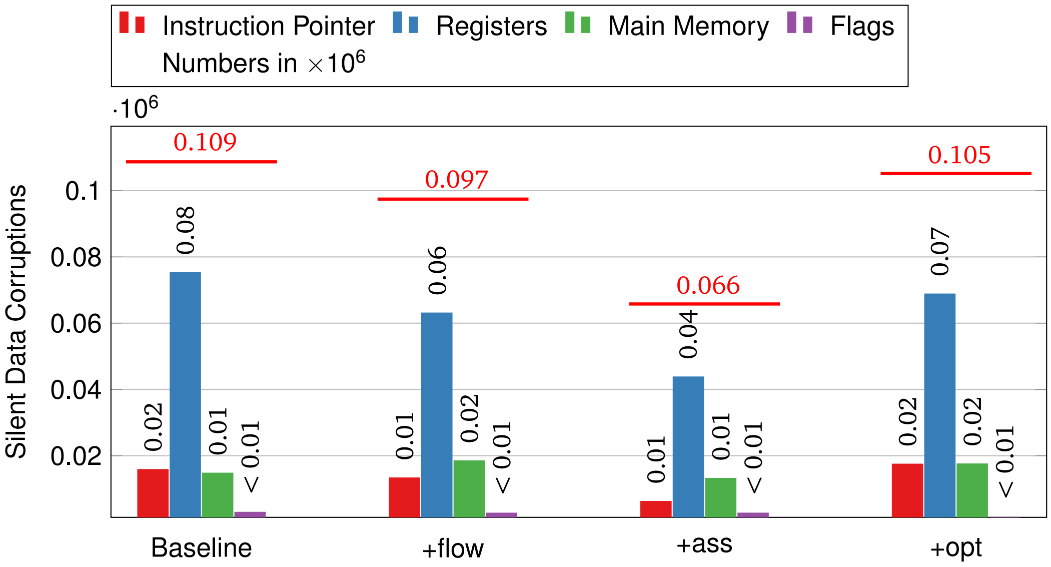


Figure 5.15 – SDC Rates for the *encoded* *d*OSEK. Compared to the unencoded *d*OSEK, the SDC rate is reduced especially for memory locations. All described measures reduce the number of SDCs even further.

On contrary to the system-call specialization, the control-flow monitoring and system-state assert were designed as *additive* dependability measures. Additive in the sense that they add code to system; the system-call specialization is *substractive*, since it removes code from the system.

The control-flow monitoring decreases the SDC rate for the unencoded *d*OSEK by −32.97 percent and for the encoded *d*OSEK by −10.36 percent. The measure mainly decreases the number of SDCs that stem from memory faults, which is surprising in the first moment, since the operating-system memory is not checked directly by this measure. Nevertheless, faults in the system state can lead to scheduling decisions that were not denoted in the GCFG. So a system-state fault can result in an illegal control flow, which can be detected by the control-flow monitoring. This observation explains also, why the influence on the encoded *d*OSEK is not as high as for the unencoded variant. In the encoded variant, the system state is especially hardened against faults, so the control-flow monitoring cannot have such a high impact here.

The system-state asserts decrease the SDC rate for the unencoded *d*OSEK by −50.97 per cent and for the encoded *d*OSEK by −39.47 percent. So again, the measure has a higher impact on the unencoded *d*OSEK. Again this higher impact of the unencoded variant stems from the memory faults (−50.97 %), which are highly reduced by the encoding. For the register faults, the improvement is in the unencoded case is with −42.06 percent almost equal to the encoded variant (−41.82 %). In all cases, the SDCs that stem from flag-register faults are quite low.

Another interesting question is, whether the additive dependability measures add SDCs to the system. Since these measures add code to the system, we are in danger that this additional code leads to additional SDCs. For the overall SDC, we have seen that the number of SDCs decreased in all cases. Nevertheless, for some fault locations the SDC rate increases. For example, the SDC rate for memory faults increases by 25.17 percent (encoded) when control-flow monitoring is applied. This behavior is quite surprising, since the additional code loads and stores memory only through absolute addresses. The only possible explanation I could give for this behavior is the fact that the memory placement of the other system objects changes by introducing the control-flow–region marker word. Here a further investigation is needed. Nevertheless, a system engineer should always look for such unexpected SDC increases and that the SDC decreases weight out the SDC increase that is caused by additional code.

in 10^6	unencoded		encoded	
	SDCs	Improvement	SDCs	Improvement
Baseline	1,404.86	–	0.21	–
+flow	946.62	−32.62 %	0.18	−15.25 %
+ass	690.52	−50.85 %	0.16	−24.01 %
+opt	1,387.07	−1.27 %	0.19	−10.06 %

Table 5.5 – SDC Rate Improvement without spatial isolation. The SDC rates of *d*OSEK without MPU/MMU protection resemble the results for the spatial-isolated *d*OSEK. The improvement is always given to the corresponding Baseline.

For a *d*OSEK without spatial isolation, the numbers are quite the same. The absolute numbers and the improvements are given in Table 5.5. More detailed graphs can be found in the appendix on page 96.

But not only the rates of silent data corruptions is of interest, but also the distribution of the detected errors. So, how does the applied measures influence the experiment outcomes besides reducing the SDC rate; where were the faults detected. Figure 5.16 gives this overview for the unencoded *d*OSEK with spatial isolation. The error classes besides the SDC, needs some explanation: If a fault injection leads to an hardware *trap* (e.g., illegal opcode, division by zero), the operating system can react to this events. The *timeout* class catches all endless loops in the system. These kind of errors can be detected easily with a watchdog unit. The asserts are split up in three categories: *General Asserts* are those asserts that are present in *d*OSEK anyway and trigger on invalid system states. The *flow-region assert* and the *system-state assert* category are only present in the variants with the corresponding dependability measure.

First of all, some categories are nearly unaffected by the applied measures. The timeout, trap, and general assert numbers are quite stable. The only exception is the system-state assert method

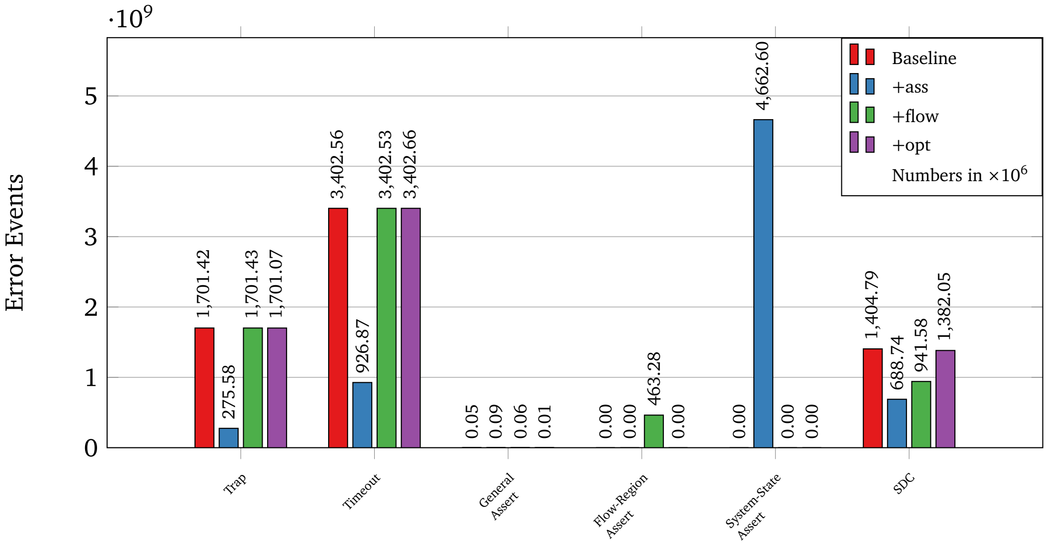


Figure 5.16 – All Error Events for the Fault Injection of an unencoded *d*OSEK. Each experiment produces an reaction of the system that can be categorized in benign faults, detected errors and silent data corruption. Benign faults are not shown in this figure.

(+ass). Here, most faults that would lead to traps or timeouts are converted into system-state–assert events. The method does not only decrease the SDC rate, but does, as a side effect, trigger on system states that would lead to traps.

For the control-flow monitoring, the situation is quite different. Here, the number of traps does not decrease, but additional flow-region asserts occur. One interesting observation is that the absolute number of SDCs, which are avoided by the control-flow monitoring measure ($-463.21 \cdot 10^6$), matches the number flow-region asserts quite precisely ($463.28 \cdot 10^6$). From this very close matching, I conclude that the control-flow monitoring is, although it does not decrease the SDCs as effectively as the system-state asserts, highly effective for a class of faults that are not covered at all by the base system. This observation on the unencoded system does not hold in the same magnitude for the encoded system (SDCs ↓: $-0.01 \cdot 10^6$; Flow-Region Assertions ↑: $0.16 \cdot 10^6$), since there other dependability measures are already in place. So, in the encoded case, the control-flow monitoring catches faults that would have been also catched by the encoded *d*OSEK.

5.4.3 Composing Measures

Until now, we have only applied a single measure. But how do the presented measures interact, when multiple ones are applied at once. Does their influence on the SDC simply add up, or annihilate their combined influence itself. For this, the *I4Copter* scenario was augmented with all combinations of the three measures. For resulting systems, a complete fault-injection campaign was undertaken. The results are shown for the unencoded *d*OSEK in Figure 5.17 and for the encoded *d*OSEK in Figure 5.18.

In the two figures, the relationship of the variants is given as a tree. All variants in one horizontal layer have in common the fact that they have the same number of methods applied. The first two layers were already discussed in Section 5.4.2. The edges between layer two and layer three indicate how the measures are combined. A node's color indicates the SDC rate,

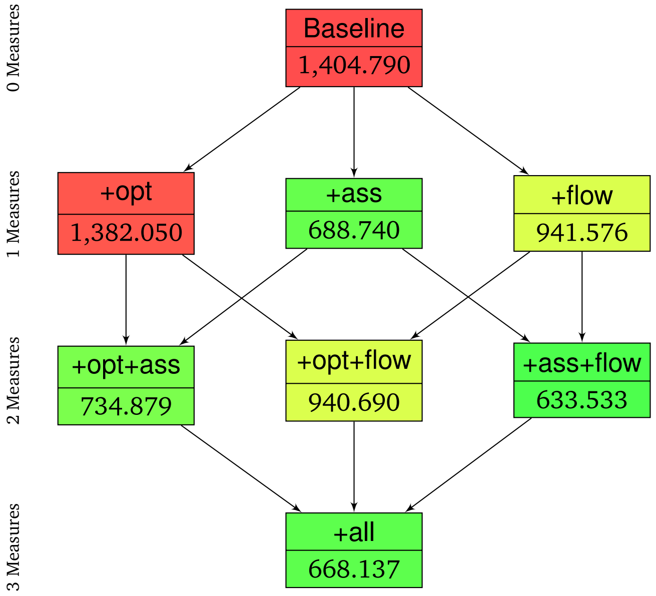


Figure 5.17 – SDC Distribution for Multiple Dependability Measures in *unencoded d*OSEK. All SDC rates are given in 10^6.

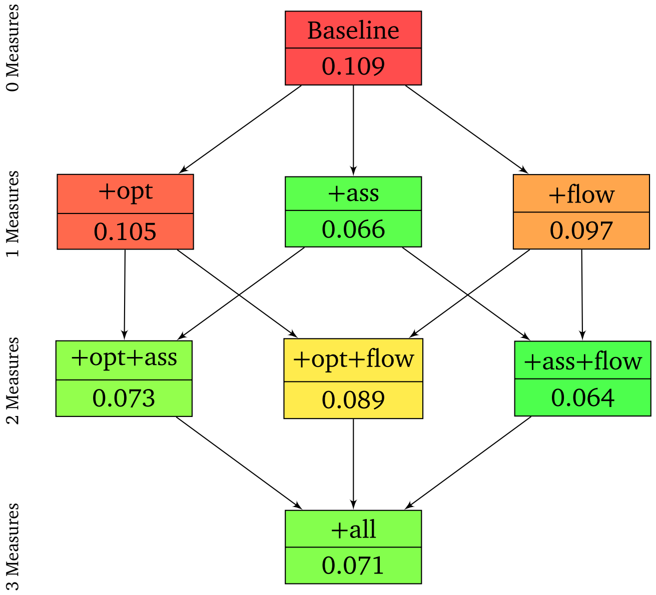


Figure 5.18 – SDC Distribution for Multiple Dependability Measures in a *encoded d*OSEK. All SDC rates are given in 10^6.

which as also given as an absolute number in 10^6. Here, the best value is filled in green and the worst (highest) SDC rate is filled in red.

The most surprising observation is the fact that not the variant with all measures applied (+all) inhibits the lowest SDC rate, but the variant with only two measures applied (+ass; +flow). This observation holds not only for the encoded, but also for the unencoded *d*OSEK. It is surprising, since each measure applied by itself decreases the number of SDCs. But the influence of the SDC decrease does not simply add up when more than one measure is applied.

When looking at the SDC rates, two situations arise: First, the combination of two methods does not show the full sum of SDC savings the single measures showed. The combination of system-state assert and control-flow monitoring is such a case. For the unencoded system, system-state asserts shows an absolute SDC saving of $-716.04 \cdot 10^6$, while the control-flow montoring has an absolute SDC decrease of $-463.21 \cdot 10^6$. When both methods are applied at once, the SDC decrease is $-771.25 \cdot 10^6$. This effect stems from the fact that different dependability measures catch the same faults. So when both methods are applied, the catched faults overlap and the SDC decrease is not the full sum.

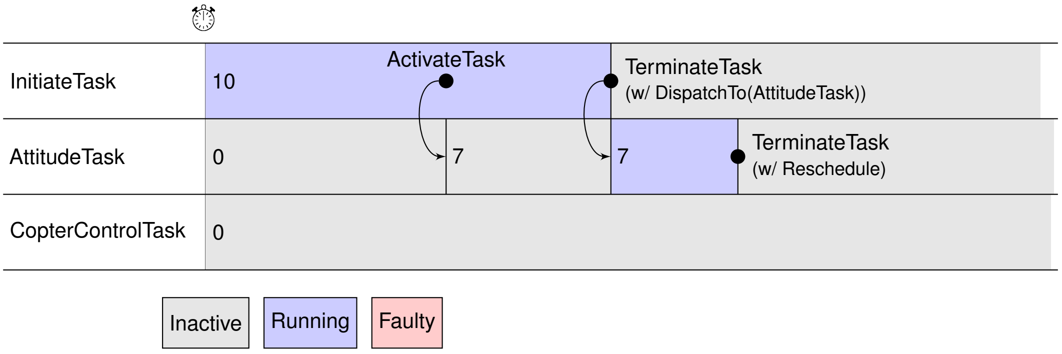
The second and more severe, combination anomaly is when the application of two (or more) methods increase the SDC rate. This is the case for the combination of system-state asserts and system-call specialization. For the encoded system, the additional application of specialization to a system, which is already protected by system-state assert, the SDC rate increases by 10.53 percent. Also, the encoded system with all three methods applied shows a 10.93 percent higher SDC rate than the system without system-call specialization. For the unencoded system, the situation is very similar. So, how can this effect emerge?

For the unencoded *d*OSEK, the anomaly is with $+46.14 \cdot 10^6$ quite high. Almost all of those additional faults stem from faults that were injected into the main memory. Furthermore, only 39, very large, equivalence classes cause this anomaly. A detailed investigation showed the following picutre: Faults that were injected into the third lowest bit into the ready list, which causes an priority increase for a single subtask by 8 priorities, were detected when only system-state asserts were enabled. When the system calls are specialized, the detected errors transform to silent data corruption.

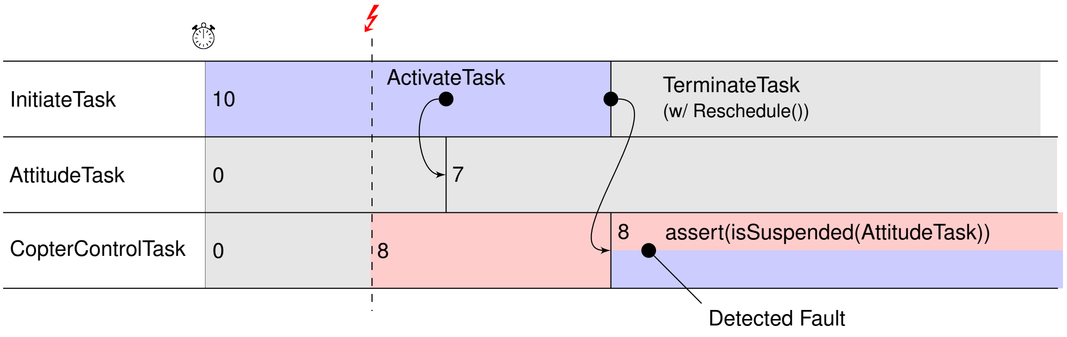
Another closer look revealed the problem. Figure 5.19 shows the execution traces of both systems. At the beginning of each hyper period, after the system has halted for a long time, an alarm activates the InitateTask. Within this alarm interrupt, a fault occurs and flips the third bit of the dynamic priority of the CopterControlTask. Because of this flip the CopterControlTask increases its priority to 8, although it is the subtask with the lowest priority in the system and can never have this priority level. The faulty ready word endured long passively in memory, which causes very long equivalence classes. During its operation, the InitiateTask activates another subtask (AttitudeTask) and hoists its priority to 7. In the system without system-call specialization, a scheduler is invoked when the InitiateTask substak terminates. Since there, the CopterControlTask is the subtask with the highest dynamic priority, it is scheduled. But, the CopterControlTask subtask is enriched by the system-state assert approach with a check that no higher-priority subtask, especially the AttitudeTask substask in this case, can be `READY` (priority > 0). Therefore, the system-state assert catches the fault and no silent data corruption occurs.

In the system with system-call specialization enabled, the picture at the termination of the InitiateTask subtask is a little bit different. Here, the system generator did not call the scheduler, since it is for sure that the AttitudeTask subtask must be dispatched. Therefore, AttitudeTask is dispatched, although it has not the highest priority in that very moment. So, the influence of the fault manifests itself only after AttitudeTask terminates. At this point, the CopterControlTask subtask is scheduled by a regular scheduler instance, but the system-state assertion cannot catch the fault anymore, since no higher-priority subtask is ready. A silent data corruption, in form of a wrongly visited checkpoint in CopterControlTask, appears.

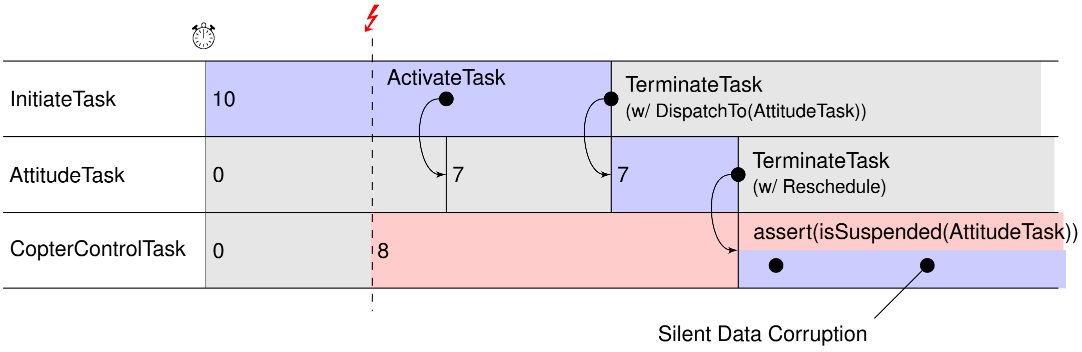
So here the “anomaly” is no real anomaly. It shows an effect that emerges when system-call specialization and system-state asserts are used within the same system. The assertions were inserted to check whether the system state is as assumed when the execution hits the system-state assert. So, as already mentioned, the system-state asserts have an additional flow-control



(a) Golden Run



(b) Without System-Call Specialization



(c) with System-Call Specialization

Figure 5.19 – SDC Anomaly in the unencoded *d*OSEK. Numbers denote the dynamic priorities. In the specialized system, the effect of the fault reveals at a later point, where the system-state asserts cannot catch it anymore.

monitoring semantic; they check whether the system has reacted correctly on the system state. The system-call specialization makes the execution path through the system more static; more decisions are made at the code generation time. In the case of the described problem, the execution path to the check was made static, the scheduler is not queried anymore. Therefore, the influence of the fault is delayed for so long that the system-state assert cannot catch the fault anymore.

For the encoded *d*OSEK, the cause of the anomaly differs from the unencoded *d*OSEK. Figure 5.20 shows the SDC rates for the anomaly. The number of SDCs that originate from register faults decreases, while the influence of instruction-pointer faults increases significantly. For the first anomaly (+ass → +opt+ass), the increase for the program counter is +171.22 %; for the second observed anomaly (+ass+flow → +all) it is 172.94 percent. When we do not take the

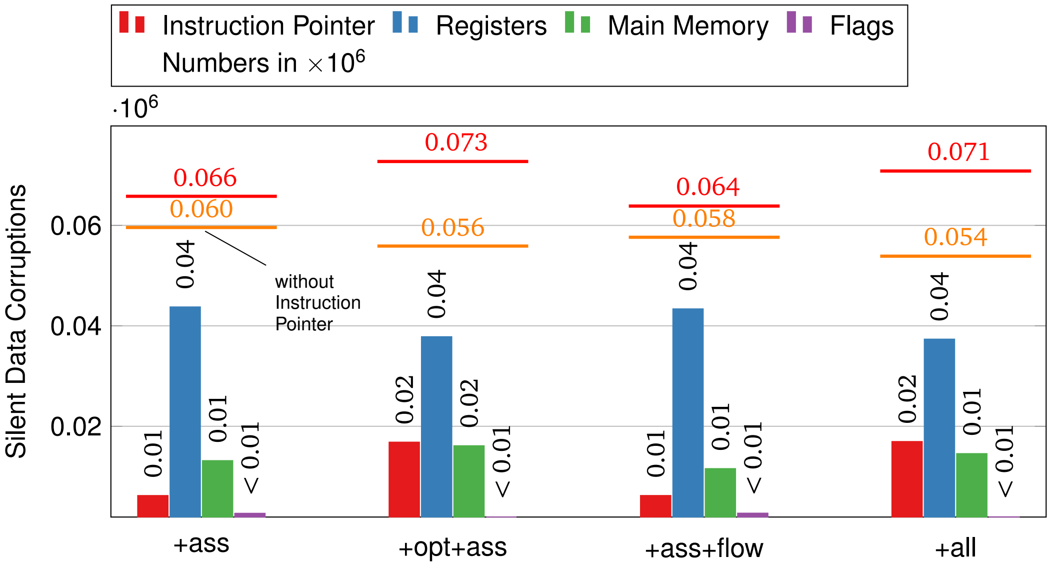


Figure 5.20 – Anomaly in SDC Rates for Multiple Measures (encoded *d*OSEK). When applying the system-call specialization method to a system, which is already protected by system-state asserts, the SDC rate increases.

instruction pointer into account, the SDC rate decreases as we would expect it (orange bar). I assume, that this increase stems from the fact that each system call decreases in code size. By this decrease the single system-call instances move close together. So, a fault in the instruction pointer has a higher chance to jump to another system call, which has a different effect.

The fault, which caused the higher SDC rate in the unencoded *d*OSEK, cannot lead to an error in the encoded *d*OSEK, since all priorities stored encoded and a single bit flip causes an invalid value, which is catched by the next full scheduling. In the encoded case, the anomaly is a real anomaly and only caused by the code placement strategy of compiler and the functional density of the resulting code.

5.5 Summary

In this chapter, I evaluated the characteristics of the GCFG construction methods, as well as the system images that were generated and enriched by the gathered in-depth application knowledge. The evaluation was done on three different levels: the characteristics of the analyses, the metrics of the resulting system images, and the dynamic behavior of the system images towards soft errors. For the evaluation, I selected two different benchmark applications: the *d*OSEK [19] test-suite and the subtask setup of the *I4Copter* benchmark.

The first evaluation direction is the characteristic of the system analysis passes. Especially, the two GCFG construction methods SSE (Section 3.3.3.1) and SSF (Section 3.3.3.2) determine the complexity and the quality of the system analysis. In terms of run time, the SSE analysis shows a lower constant overhead for small systems, while the overall complexity reveals an exponential characteristic. In opposite to this, the SSF analysis exhibits a high constant overhead due to the handling of external events (i.e., interrupts and alarms), while the run-time requirements remains low, even for large systems. For both methods, the handling of external events is the main factor for analysis run time and the quality of the resulting application knowledge. The no-reactivation annotation, which was presented in Section 3.4 and limits the activation of external events, supports the system analysis significantly, while the SSE benefits in a much higher extend than the SSF analysis.

For measuring the quality of the gathered information, I introduced the system-state precision, which denotes the quantity of sure knowledge about the system at a given point in the application. The more we can say for sure about the system at one point, the higher is the precision of the calculated system state. All benchmarks point out the higher precision of the SSE analysis compared to the SSF analysis. The comparison also revealed that the number and the possible activation points of external events gravely influence the overall precision of the system states. Therefore, the usage of no-reactivation annotations assists both analyses in producing precise system states.

When we compare the GCFG approach to the normal approach of only utilizing the information from the static system description (OIL) during the system generation time, a much higher potential of application knowledge is expressed and can be provided to later phases of the system generation.

The second level of evaluation are the code size and run-time characteristics of the resulting system images. For this, the *d*OSEK operating-system generator applied three optimization measures to the benchmark applications. The three measures, which optimized different non-functional properties, are the system-call specialization (Section 4.1.1), the system-state assertions (Section 4.2), and the control-flow monitoring with dominator regions (Section 4.3). For the kernel fragments that are the result of the system-call decoupling the code size can be measured exactly. The system-call specialization reached an average code-size decrease of 82.53 percent for the encoded *d*OSEK. So, for the encoded *d*OSEK the cost of the encoding could be decreased significantly, while the observable functionality stayed the same. For the two dependability aspects, the code size overhead is independent of the technique that is used to implement the system-call logic. The system-state asserts add in average 138 bytes to each system-call site, while the control-flow monitoring adds about 31 bytes. In total, when choosing from the repertoire of *d*OSEK and the presented measures, we can now either choose to build a encoded and specialized system at half of the code-size price of an unprotected *d*OSEK (0.55×). Or we can build an protected and specialized *d*OSEK with both dependability measures applied at 60 percent overhead of the cost of an unprotected *d*OSEK (+118.07 bytes per system call).

Like the code size, the influence on the run time of the presented measures is important to the real-time–systems engineer. The run-time overhead for both dependability measures remain about the same for the encoded and for the unencoded *d*OSEK (system-state asserts: +10.57 instructions; control-flow monitoring: +6.2 instructions). Both methods can be applied to any OSEK-like operating system with a very low number of additionally executed instructions.

The most impact on the run time of the system has the system-call specialization. Here the optimization cuts away 33.38 (unencoded) respectively 52.74 (encoded) percent of the run time that is spent during the kernel execution. When we, again, combine all three presented measures, a system with encoded operations and specialized system-calls can be build with 43.23 percent run-time overhead. With the two additional dependability measures, the fully protected system shows a run-time overhead of 73.98 percent compared to the fully unprotected and unmodified *d*OSEK.

The most interesting evaluation aspect is the influence of the three measures on the rate of silent data corruptions. When applying only a single measure, the influence of the system-call specialization had the smallest influence on the SDC rate. In the best case, −3.25 percent could be gained compared to the base systems. The ystem-state assert is the best of the presented measures and cuts the SDC rate in half (−50.97 %). The control-flow monitoring showed a slightly lesser effect (maximal −32.97 %).

The SDC rate measurement for systems were multiple measures were applied showed some interesting effects. Although, the SDC rate could be reduced by −54.90 per cent for the unencoded *d*OSEK, the additional application of system-call specialization *increased* the number of SDCs again. The detailed investigation revealed that the system-state assert measure and the specialization interact in an unforeseen way: The specialization delayed the influence of faults until a point were the assertions could not detect the fault anymore. So, although the three presented measures are composable, their effects on the SDC rate does not simply add up.

As a baseline, I could construct a encoded *d*OSEK instance that uses 49.03 percent less flash memory for code, has a 42.59 percent lower run time and incorporates both system-state asserts and control-flow monitoring. These applied measures reduce the SDC rate by 34.84 percent compared to the best, until now, available *d*OSEK.

Chapter 6

Related Work

In this chapter, I want to present previous work that was not the fundamental of this thesis, but is related to the various topic discussed in this work.

A global view on the interaction between operating system and application was already proposed by Bertran et al. [8]. There a global control-flow graph for a complex embedded system, which was built on top of Linux, was constructed in a flow-insensitive manner. 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 the scheduling and system-call semantic was not taken as closely into account as I did it. This stems mainly from the fact that the Linux kernel has not such a strict and simple semantic as OSEK.

Barthelmann [5] used the system description file of an OSEK system and the included static priorities to determine basic-block transitions that are impossible. With this application knowledge, the number of registers that has to be saved in the preemption of a task is reduced by the compiler. Here, also application knowledge is used to influence a non functional property of an OSEK kernel. But in contrast to my thesis, the scheduling semantic of OSEK was used in a flow-insensitive manner.

In the area of formal methods and verification, the OSEK semantic got some attention: Waszniowski and Hanzálek [51] designed a model for the OSEK standard for the model checker UPPAAL[7]. They modeled all components as timed automata and took also inter-process communication (OSEK events) into account. Their main focus was verifying application properties, like schedulability analysis. Huang et al. [21] modeled OSEK as communicating sequential processes (CSP). The application subtasks were modeled without considering the internal application structure and interrupts were excluded. With this model, they could verify different properties of their OSEK system, like dead-lock freedom and freedom of priority inversion. For my approach, these models could provide a more formal definition of `system_semantic()`.

System specialization was already discussed by the operating-system community for general-purpose operating systems. Pu, Massalin, and Ioannidis [43] developed the Synthesis kernel, which included a code synthesizer that produced optimized code paths at run time for often invoked system calls, like `read()`. Due to manual implementation of code templates, which are then filled by the synthesizer, huge performance benefits arouse from shorter kernel execution paths. In comparison to the dynamic Synthesis system, my approach toward system-call specialization took also the in-depth application knowledge into account and is executed off-line. Pu et

---

[7] http://www.uppaal.org/

al. also mention the problem of code-size explosion. McNamee et al. [33] used Tempo, a partial evaluator for C programs, and a set of specialization predicates to identify functions automatically for specialization within the kernel. Here, the specialization was done also dynamically at run time and the specialization does not include detailed application knowledge.

Several approaches towards control-flow monitoring were developed for application logic. Benso et al. [7] uses regular-expression automata to check whether the executed basic block sequence of an application is correct. Oh, Shirvani, and McCluskey [36] presented an approach with *software signatures*. Each basic block is assigned an unique number; when the basic block is entered or left, a global variable is xored with the unique number. Without control-flow errors, the global variable contains always exactly the unique number of the currently executed basic block. Yau and Chen [54] divided the control-flow graph into loop-free regions. For each region a database of possible paths is encoded and checked during the execution. All mentioned approaches do only consider the control-flow graph of a single function or subtask, but could be extended to catch control-flow errors on the ABB and GCFG level.

Chapter 7

Future Work

In this chapter, I want to present a few ideas that came to my mind during the development and writing phase of this thesis. These ideas are the result of many discussions with my supervisors Martin Hoffmann and Daniel Lohmann.

7.1 Usage in Other Whole-System Analyzes

The GCFG provides a global view upon the interaction between the application and the real-time system. This knowledge could be exploited also in other areas of the real-time world.

One example of such an area is the analysis of the worst-case execution time. When doing flow-sensitive WCET analysis, the micro-architecture and the cache state are important factors to the precision of the WCET over-approximation. When not taking the operating system into account, the WCET analyzer has to assume after each system call that the cache does not contain any data of the subtask's working set, since all cache lines could be extruded by another subtask. Chong et al. [12] proposed an approach to incorporate the influence of the operating system to ease the effect on the system state during the WCET analysis. They annotated each system call (type) with a maximal effect that can occur onto the system state. By this flow-insensitive approach, the influence of system calls is limited. With the GCFG analysis at hand, the WCET analysis could be provided by even better annotations on the influence of the operating system.

Similar to the WCET analysis, could a whole-system application compiler, like KESO [17, 52], benefit from the global view upon the system's control flow. With the information which basic block is executed on the machine directly after a system call, constant propagation across kernel barriers is achievable. Also could the global view and a detailed static analysis of the application code reveal protocol violations in the application code. Static code checkers could be made operating-system aware in a flow-sensitive manner.

7.2 A Better Control-Flow Monitoring

In Section 4.3, I presented an approach to insert control-flow monitoring checks into the system calls for a positive test. The positive test can check whether the execution flow jumps into the flow region without entering it via the region leader. Here I want to present an approach, which was not implemented by me because of technical difficulties. It uses the concept of the *dominance frontier* [15] to determine all resetting points for the marker when the region is left.

Definition 10 (Dominance frontier)**.** *The dominance frontier of a CFG node X is the set of all nodes Y, such that X dominates a predecessor of Y, but does not strictly dominate Y.*

So the dominance frontier of a region leader are those ABBs that are executed just after the region is left. If we reset the region marker there, the resetted marker also indicates an information: We have left the region on a valid path. Therefore, we can add checks outside the region that the regions markers are not present. Resetting could also be done in the "last" dominated block within the region, which are the predecessors of the dominance frontier (see Figure 7.1).

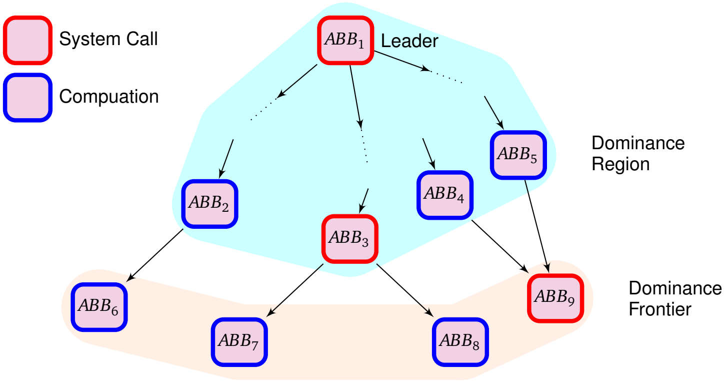


Figure 7.1 – A control flow region with its leader and possible resetting points for the region marker, which can be located in the dominance frontier or directly "before".

But when only the system-call blocks are manipulated by the OS generator a problem arises. For having a sound check for being outside the region, we must ensure that no control flow "sneaks" around the resetting nodes. If as well the node within the dominance frontier and its predecessor(s) in the region are computation blocks, the OS generator has no chance to insert the resetting operation. So if the control flow leaves the region through that path, the marker is not reset and the next negative check would cause an false positive. ABB_2 and ABB_6 from Figure 7.1 are an example of such a bypass. This could be solved by manipulating computation blocks, but because of technical difficulties this was not implemented for this work. Nevertheless, with a solution here, a negative check could be implemented as cheaply as the positive check.

7.3 Converting OSEK into a Finite State Automata

The construction of the GCFG with the symbolic system execution showed that the system-calls, the application logic, and the OSEK semantic can be expressed as a directed graph of all possible system states. So, the operating system transitions from one state into another one, when an external event, like a system-call or an external event occurs. Upon the result state an action, in our case a scheduling decision and a dispatch operation, is done. Therefore, it should be possible to express the OSEK semantic for exactly this application as a finite state automata.

State automata based approaches to design applications are already widely used in the real-time systems community. The state-machine compiler [47] provides an domain specific language to express application logic in terms of states and transitions and is able to produce code in various languages. State machines have one main benefit for verification of systems: they are known to be turing *in*complete. Therefore, some work was already done to design an operating system as a state machine [24, 35].

With the global control-flow graph at hand, a state machine for exactly one application could be generated that shows exactly the scheduling decisions the OSEK specification demands. By this the system state could be reduced to architecture specific memory variables and a machine word that indicates the current system state. For this venture, some tasks have to be tackled: Do all states of the symbolic system execution need a representation in the operating system state machine? How can we keep the code size of the state transition table low? What do we have to do in order to cope with the indeterminism introduced by alarms and events?

7.4 Limitations of the Approach

The presented analysis approach is not only to be extended in the already directions, but it is also a topic of further research to ease the limitations the current approach reveals. During Chapter 3, I enlisted already the limitation I demanded from the application developer. These limitations touched mainly the usage of system calls in various parts of the applications and the system-call arguments. Besides these internal limitations, the current approach has several other limitations that narrow the scope of the presented methods.

First of all, not all concepts that are enlisted in the OSEK standard are covered by the `system_semantic()` transformation. Since I restricted the scope of this thesis to OSEK BCC1, I could omit event support, multiple activations per subtask, and multiple subtasks per priority. Especially the missing event support is a severe limitation, since it narrows down the expressiveness of the application code, while the other missing features limit the coarse-grained structure.

Another limitation of the approach is the complexity of the analysis. As depicted in Section 5.2.1, does the SSE analysis exhibit an exponential analysis run time. The number of system states that have to be enumerated explodes with large applications. This explosion can either be limited by the presented no-reactivation annotation or by a certain application design. The closer a system call is to the root of the call hierarchy of a subtask, the larger can the ABBs be constructed. Therefore, it is a limitation of the approach that it forces a certain application design, when a short analysis time is required.

Among the exploitation methods, the system-call specialization has the most limited scope. The general overhead of the system-call decoupling cannot be compensated by the savings the system-call specialization carries out. Here an hybrid approach might be applicable: Specialize only those system-calls with very precise system states and use generic implementations for situations with fuzzy system states.

Chapter 8

Conclusion

In this work, the global control-flow graph is presented as a approach for expressing in-depth application knowledge about real-time systems. Through the tight coupling of application and the real-time operating system, much of the interaction between those two components can be statically deduced from the application logic and the operating-system semantic. The OSEK operating-system standard provides such a semantic and is widely used in the automotive industry.

The global control-flow graph connects the atomic basic blocks, which are a superstructure of the application's basic blocks, of different subtasks to get a global view on the possible scheduling and preemption decisions in the system. I provided two approaches, which both operate on an abstract system state, to calculate the global control-flow graph. The symbolic system execution enumerates all possible system states, while the system-state flow analysis propagates an fuzzy system state through the application logic.

The application knowledge can be used to optimize the non functional properties of the operating system in several ways: The system-call specialization decouples the system-call sites from each other and exploits the gathered information to remove unnecessary code paths from the kernel fragments. As a dependability measure, the system-state–assert method inserts checks to verify that constant parts of the system state have the correct value at certain points in the application. The control-flow monitoring uses dominator regions within the global control-flow graph to monitor the execution flow of operating system and application.

In the evaluation, I could show the impact of the presented measures upon the *I4Copter* benchmark. I could construct a encoded *d*OSEK instance that uses 49.03 percent less flash memory for code, has a 42.59 percent lower run time and incorporates both system-state asserts and control-flow monitoring. These applied measures reduce the silent data corruption rate by 34.84 percent compared to the best, until now, available *d*OSEK.

The usage of in-depth application knowledge is one of the keys to software-based measures against transient hardware faults. The awareness of application behavior opens up the possibility to very fine-grained dependability measures. But this application knowledge could also be used in other areas of analyzing and tailoring real-time systems. For example, could a worst-case execution time analysis benefit strongly from the knowledge about all scheduling decisions. Also, a whole-system compiler could do constant propagation across kernel activation borders. And the total knowledge of all system states could provide the possibility to express the whole operating-system logic as a finite state automata.

List of Acronyms

ABB	atomic basic block
BB	basic block
CFG	control flow graph
GCFG	global control flow graph
ICFG	inter-procedural control flow graph
IRQ	interrupt request
ISR	interrupt service routine
MPU	memory protection unit
OIL	OSEK implementation language
OSEK	“Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug” (open systems and their interfaces for electronic in automobiles)
PCP	priority ceiling protocol
RCB	reliable computing base
ROM	read-only memory
RTOS	real-time operating system
RTSA	real-time system-architecture
SCDT	system call dominator tree
SDC	silent data corruption
SSE	symbolic system execution
SSF	system-state flow
TMR	triple modular redundancy
VM	virtual machine
WCET	worst-case execution time

List of Figures

2.1 OSEK task state diagram ([37, Fig. 4.3]) 4
2.2 ABB Dependency Graph Construction 9

3.1 Execution Sequence of OSEK Application 14
3.2 ICFG/GCFG for OSEK Example Application 15
3.3 ABB_transform on the system state 17
3.4 CFG with artificial syscall blocks 19
3.5 CFG and ICFG for two functions 21
3.6 Data-flow analysis schema 23
3.7 ICFG analyses result for Table 3.2 27
3.8 Output System States for Computation Block 28
3.9 Operation schematic of `system_semantic()` 31
3.10 SSE graph and the ABB grouping 31
3.11 ISR regions within the GCFG 32
3.12 Progressive GCFG construction with *system-state flow* (SSF) 34
3.13 SSF merge at Function Calls 34
3.14 Task Containers as IRQ-block annotation 37

4.1 Classical OS component activation graph 40
4.2 System Call Decoupling and Specialization 41
4.3 Decision Tree for Generating an ActivateTask 43
4.4 SystemEnterHook and SystemLeaveHook 45
4.5 Control Flow Regions Used for Fault Detection 47
4.6 Control Flow Graph and Dominator Tree 48
4.7 System Call Dominator Tree 48

5.1 *I4Copter* Subtask Constellation 54
5.2 Copied System States for SSE/SSF 55
5.3 Copied System States by ISR count 56
5.4 System State Precision for SSE/SSF 58
5.5 System State Precision by ISR count 59
5.6 Relative Graph Sizes 60
5.7 System Call Codesize for an *unencoded d*OSEK 61
5.8 System Call Codesize for an *encoded d*OSEK 62
5.9 System-Call Codesize for an *unencoded d*OSEK (Detail) 64
5.10 System-Call Codesize for an *encoded d*OSEK (Detail) 65

5.11 System Call Runtime for an *unencoded d*OSEK 66
5.12 System Call Runtime for an *encoded d*OSEK 67
5.13 FAIL*'s Fault Space Model 68
5.14 SDC Rates for the *unencoded d*OSEK 70
5.15 SDC Rates for the *encoded d*OSEK 70
5.16 All Error Events for the Fault Injection of an unencoded *d*OSEK 72
5.17 SDC Distribution for Multiple Dependability Measures in *unencoded d*OSEK 73
5.18 SDC Distribution for Multiple Dependability Measures in a *encoded d*OSEK 73
5.19 SDC Anomaly in the unencoded *d*OSEK 75
5.20 Anomaly in SDC Rates for Multiple Measures (encoded *d*OSEK) 76

7.1 Dominance Borders and Resetting points 82

A.1 SDC Rates for the *unencoded d*OSEK without Spatial Isolation 96
A.2 SDC Rates for the *encoded d*OSEK without Spatial Isolation 97
A.3 SDC Distribution for the Instruction Pointer for the unencoded *d*OSEK 97
A.4 SDC Distribution for the Instruction Pointer for the encoded *d*OSEK 98

B.1 Experiment Flow Chart 101

List of Listings

1 Example for an OIL declaration 5
2 Example OSEK application 8
3 Example of variable System Call Argument 20
4 Forbidden activation of system-relevant function 22
5 Conditional Acquisition of a Resource 24
6 `ABB_transformation()` for `ChainTask` 27
7 Code for the Special Case in Table 3.3 30
8 Machine Code example for `ActivateTask#1` 42
9 Generated Code for CFG Region Monitoring 49

List of Tables

2.1 OSEK System Call Overview 6

3.1 Interupt Block System Calls 23
3.2 Example of Priority Spreading 26
3.3 Sandwiching priorities as a special case for the block scheduler 30

5.1 *d*OSEK Variant Overview 52
5.2 Matrix of *d*OSEK Test-Cases 53
5.3 Run-Time Evaluation for analyzing the *I4Copter* 56
5.4 System State Precision for *I4Copter* 59
5.5 SDC Rate Improvement without spatial isolation 71

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[2] Horst Schirmeier et al. "FAIL*: Towards a Versatile Fault-Injection Experiment Framework." In: *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*. (Munich, Germany). Ed. by Gero Mühl, Jan Richling, and Andreas Herkersdorf. Vol. 200. Lecture Notes in Informatics. Gesellschaft für Informatik, Mar. 2012, pp. 201–210. ISBN: 978-3-88579-294-9.

[3] Frances E. Allen. "Control Flow Analysis." In: *SIGPLAN Not.* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: `10.1145/390013.808479`. URL: `http://doi.acm.org/10.1145/390013.808479`.

[4] M. Baleani et al. "Fault-tolerant Platforms for Automotive Safety-critical Applications." In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '03. San Jose, California, USA: ACM, 2003, pp. 170–177. ISBN: 1-58113-676-5. DOI: `10.1145/951710.951734`. URL: `http://doi.acm.org/10.1145/951710.951734`.

[5] Volker Barthelmann. "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)*. (Berlin, Germany). New York, NY, USA: ACM Press, 2002, pp. 149–154. ISBN: 1-58113-527-0. DOI: `10.1145/513829.513855`.

[6] Robert C Baumann. "Radiation-induced soft errors in advanced semiconductor technologies." In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316.

[7] A Benso et al. "Control-flow checking via regular expressions." In: *Test Symposium, 2001. Proceedings. 10th Asian*. 2001, pp. 299–303. DOI: `10.1109/ATS.2001.990300`.

[8] Ramon Bertran et al. "Building a Global System View for Optimization Purposes." In: *Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture (SCA-WIOSCA '06)*. Boston, USA, 2006.

[9] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. "CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack." In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. (Low Wood Bay, Lake District, UK). New York, NY, USA: ACM Press, June 2012, pp. 435–448. ISBN: 978-1-4503-1301-8. DOI: `10.1145/2307636.2307676`.

[10] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. "Generative Software-based Memory Error Detection and Correction for Operating System Data Structures." In: *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN '13)*. (Budapest, Hungary). Washington, DC, USA: IEEE Computer Society Press, June 2013. DOI: `10.1109/DSN.2013.6575308`.

[11] Shekhar Y. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation." In: *IEEE Micro* 25.6 (2005), pp. 10–16. ISSN: 0272-1732.

[12] Lee Kee Chong et al. "Integrated Timing Analysis of Application and Operating Systems Code." In: *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. 2013, pp. 128–139. DOI: `10.1109/RTSS.2013.21`.

[13] C. Constantinescu. "Trends and challenges in VLSI circuit reliability." In: *Micro, IEEE* 23.4 (2003), pp. 14–19. ISSN: 0272-1732. DOI: `10.1109/MM.2003.1225959`.

[14] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. "A simple, fast dominance algorithm." In: *Software Practice & Experience* 4 (2001), pp. 1–10.

[15] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (Oct. 1991), pp. 451–490. DOI: `10.1145/115372.115320`.

[16] Michael Engel and Björn Döbel. "The Reliable Computing Base: A Paradigm for Software-Based Reliability." In: *Proceedings of the 1st International Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*. (Braunschweig, Germany). Lecture Notes in Computer Science. Gesellschaft für Informatik, Sept. 2012.

[17] Christoph Erhardt et al. "Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study." In: *JTRES '11: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. (York, UK). New York, NY, USA: ACM Press, Sept. 2011, pp. 96–105. ISBN: 978-1-4503-0731-4. DOI: `10.1145/2043910.2043927`.

[18] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware." In: *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '09)*. (Hamburg, Germany). Ed. by B. Buth, G. Rabe, and T. Seyfarht. Heidelberg, Germany: Springer-Verlag, 2009, pp. 283–296. ISBN: 978-3-642-04467-0. DOI: `10.1007/978-3-642-04468-7_23`.

[19] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. "dOSEK: A Dependable RTOS for Automotive Applications." In: *Proceedings of the 19th International Symposium on Dependable Computing (PRDC '13)*. (Vancouver, British Columbia, Canada). Fast abstract. Washington, DC, USA: IEEE Computer Society Press, Dec. 2013. URL: `http://www.danceos.org/publications/PRDC-FAST-2013-Hoffmann.pdf`.

[20] Martin Hoffmann et al. "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs." In: *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*. Reno, Nevada, USA: IEEE Computer Society Press, 2014. URL: `http://www4.cs.fau.de/Publications/2014/hoffmann_14_isorc.pdf`.

[21] Yanhong Huang et al. “Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP.” In: *Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on*. 2011, pp. 142–149. DOI: `10.1109/TASE.2011.11`.

[22] F. Irom and D.N. Nguyen. “Single Event Effect Characterization of High Density Commercial NAND and NOR Nonvolatile Flash Memories.” In: *IEEE Transactions on Nuclear Science* 54.6 (2007), pp. 2547–2553. ISSN: 0018-9499. DOI: `10.1109/TNS.2007.909984`.

[23] Michael B. Jones. *What really happend on Mars?* `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/`, visited 4.7.2014. 1997.

[24] Tae-Hyung Kim and Seongsoo Hong. “State Machine Based Operating System Architecture for Wireless Sensor Networks.” English. In: *Parallel and Distributed Computing: Applications and Technologies*. Ed. by Kim-Meow Liew et al. Vol. 3320. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 803–806. ISBN: 978-3-540-24013-6. DOI: `10.1007/978-3-540-30501-9_158`. URL: `http://dx.doi.org/10.1007/978-3-540-30501-9_158`.

[25] Kevin P. Lawton. “Bochs: A Portable PC Emulator for Unix/X.” In: *Linux Journal* 1996.29es (1996), p. 7.

[26] Thomas Lengauer and Robert Endre Tarjan. “A fast algorithm for finding dominators in a flowgraph.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141. ISSN: 0164-0925. DOI: `10.1145/357062.357071`.

[27] Jane W. S. Liu. *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0-13-099651-3.

[28] Daniel Lohmann et al. “Aspect-Aware Operating-System Development.” In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD ’11)*. (Porto de Galinhas, Brazil). Ed. by Shigeru Chiba. New York, NY, USA: ACM Press, 2011, pp. 69–80. ISBN: 978-1-4503-0605-8. DOI: `10.1145/1960275.1960285`.

[29] Daniel Lohmann et al. “CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems.” In: *Proceedings of the 2009 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2009, pp. 215–228. ISBN: 978-1-931971-68-3. URL: `http://www.usenix.org/event/usenix09/tech/full_papers/lohmann/lohmann.pdf`.

[30] Daniel Lohmann et al. “Interrupt Synchronization in the CiAO Operating System.” In: *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’07)*. (Vancouver, British Columbia, Canada). New York, NY, USA: ACM Press, 2007. ISBN: 1-59593-657-8. DOI: `10.1145/1233901.1233907`.

[31] Florian Lukas. “Design and Implementation of a Soft-error Resilient OSEK Real-time Operating System.” MA thesis. Friedrich-Alexander University Erlangen-Nuremberg, 2014.

[32] Florian Lukas. “Design and Implementation of a Soft-error Resilient OSEK Real-time Operating System.” (Work in Progress). Master Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2014.

[33] Dylan McNamee et al. “Specialization Tools and Techniques for Systematic Optimization of System Software.” In: *ACM Trans. Comput. Syst.* 19.2 (May 2001), pp. 217–251. ISSN: 0734-2071. DOI: `10.1145/377769.377778`. URL: `http://doi.acm.org/10.1145/377769.377778`.

[34] Steven S. Muchnick. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[35] *nesos – a finite state machine operating system*. `http://www.nilsenelektronikk.no/nenesos.html`, visited 29.7.2014.

[36] N. Oh, P.P. Shirvani, and E.J. McCluskey. “Control-flow checking by software signatures.” In: *Reliability, IEEE Transactions on* 51.1 (2002), pp. 111–122. ISSN: 0018-9529. DOI: `10.1109/24.994926`.

[37] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2011-08-17. OSEK/VDX Group, Feb. 2005.

[38] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`, visited 2009-09-09. OSEK/VDX Group, 2004.

[39] OSEK/VDX Group. *OSEK/VDX Communication 3.0.3*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf`. OSEK/VDX Group, July 2004.

[40] OSEK/VDX Group. *OSEK/VDX Network Management 2.5.3*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/nm253.pdf`. OSEK/VDX Group, July 2004.

[41] OSEK/VDX Group. *Time-Triggered Operating System Specification 1.0*. Tech. rep. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`. OSEK/VDX Group, July 2001.

[42] *Portable Operating System Interfaces (POSIX®) Base Specifications*. 2008.

[43] Calton Pu, Henry Massalin, and John Ioannidis. “The Synthesis Kernel.” In: *Computing Systems* 1.1 (1988), pp. 11–32.

[44] Fabian Scheler. “Atomic Basic Blocks - Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur.” PhD thesis. University of Erlangen-Nuremberg, 2011.

[45] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. “Priority Inheritance Protocols: An Approach to Real-Time Synchronization.” In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. DOI: `10.1109/12.57058`.

[46] Micha Sharir and Amir Pnueli. “Two approaches to interprocedural data flow analysis.” In: *Program Flow Analysis: Theory and Applications*. Ed. by Steven S. Muchnick and Neil D. Jones. Englewood Cliffs, NJ: Prentice-Hall, 1981. Chap. 7, pp. 189–234.

[47] *State-Machine Compiler*. `http://smc.sourceforge.net`, visited 29.7.2014.

[48] Andrew S. Tanenbaum. *Structured Computer Organization*. Fifth. Prentice Hall PTR, 2006. ISBN: 978-0131485211.

[49] Henrik Theiling. “Control flow graphs for real-time systems analysis: reconstruction from binary executables and usage in ILP-based path analysis.” http://d-nb.info/97232237X. PhD thesis. Saarland University, 2004. URL: `http://scidok.sulb.uni-saarland.de/volltexte/2004/297/index.html`.

[50] Peter Ulbrich et al. “I4Copter: An Adaptable and Modular Quadrotor Platform.” In: *Proceedings of the 26th ACM Symposium on Applied Computing (SAC ’11)*. (TaiChung, Taiwan). New York, NY, USA: ACM Press, 2011, pp. 380–396. ISBN: 978-1-4503-0113-8.

[51] Libor Waszniowski and Zdeněk Hanzálek. “Formal Verification of Multitasking Applications Based on Timed Automata Model.” In: *Real-Time Systems* 38.1 (Jan. 2008), pp. 39–65. ISSN: 0922-6443. DOI: `10.1007/s11241-007-9036-z`.

[52] Christian Wawersich, Michael Stilkerich, and Wolfgang Schröder-Preikschat. “An OSEK/VDX-Based Multi-JVM for Automotive Appliances.” In: *Embedded System Design: Topics, Techniques and Trends*. (Irvine, CA , USA). IFIP International Federation for Information Processing. Boston: Springer-Verlag, 2007, pp. 85–96. ISBN: 978-0-387-72257-3.

[53] David Wilner. *Vx-Files: What really happened on Mars?* Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS ’97). San Francisco, CA, USA, Dec. 1997.

[54] S.S. Yau and Fu-Chung Chen. “An Approach to Concurrent Control Flow Checking.” In: *Software Engineering, IEEE Transactions on* SE-6.2 (1980), pp. 126–137. ISSN: 0098-5589. DOI: `10.1109/TSE.1980.234478`.

[55] Y.C. Yeh. “Triple-triple redundant 777 primary flight computer.” In: *Proceedings of the 1996 IEEE Aerospace Applications Conference*. (Aspen, CO, USA). Washington, DC, USA: IEEE Computer Society Press, Feb. 1996, pp. 293–307. ISBN: 978-0780331969. DOI: `10.1109/AERO.1996.495891`.

Curriculum vitae

Christian Dietrich was born in Dezember 5th, 1989 in Rothenburg o.d.T., Germany. After finishing his Abitur[8] at the *Reichsstadt-Gymnasium Rothenburg o.d.T.* in 2009, he started his studies in computer science in the same year at the *University of Erlangen-Nuremberg*.

He received the Bachelor of Science as his first university degree in 2012 after six semesters of studying. In the bachelor thesis "A Robust and Portable Approach for Extracting Build-System Variability" he focused on the build system of the Linux kernel and its expression of static variability. Early during his studies, he began to work as student volunteer assistant at the *Department of Computer Science 4 – Distributed Systems and Operating Systems*. After working in the VAMOS project, which was focused on static variability in system software, he switched to the DanceOS working group. The main focus of DanceOS project is software-based fault tolerance in system software. In 2012, he was granted the *Max Weber Programm* scholarship of the Bavarian state.

His current research topics include the analysis of static operating systems, dependability measures and reproducible research. Luckily all of those topics could be included in this thesis.

---

[8] Certificate of General University Maturity

Appendix A

Fault-Injection Campaign

Here, I want to present you some of the graphs that were removed from the main corpus of this thesis, not because they are not interesting or informative, but because most of their information is partly redundant with other graphs and does not provide new insights.

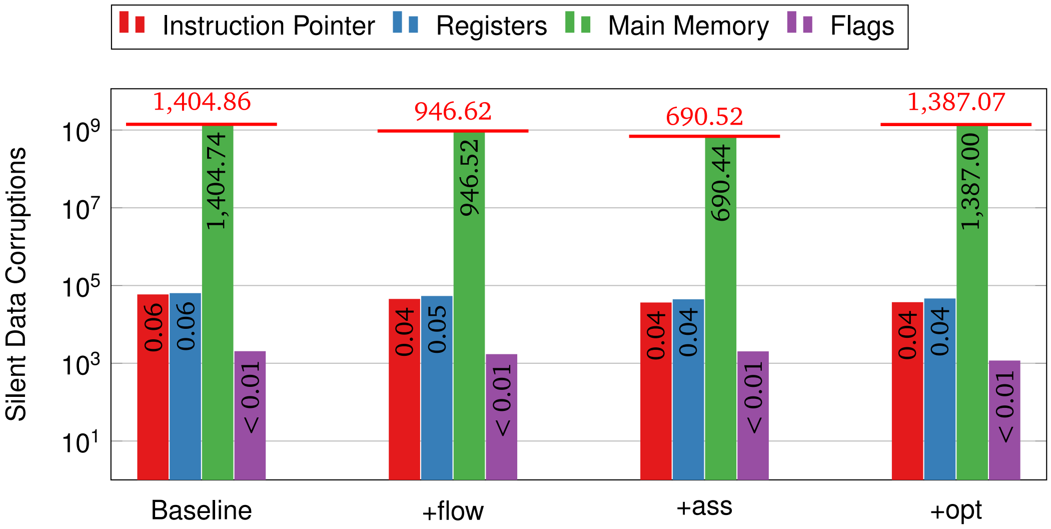


Figure A.1 – SDC Rates for the *unencoded* *d*OSEK without Spatial Isolation.

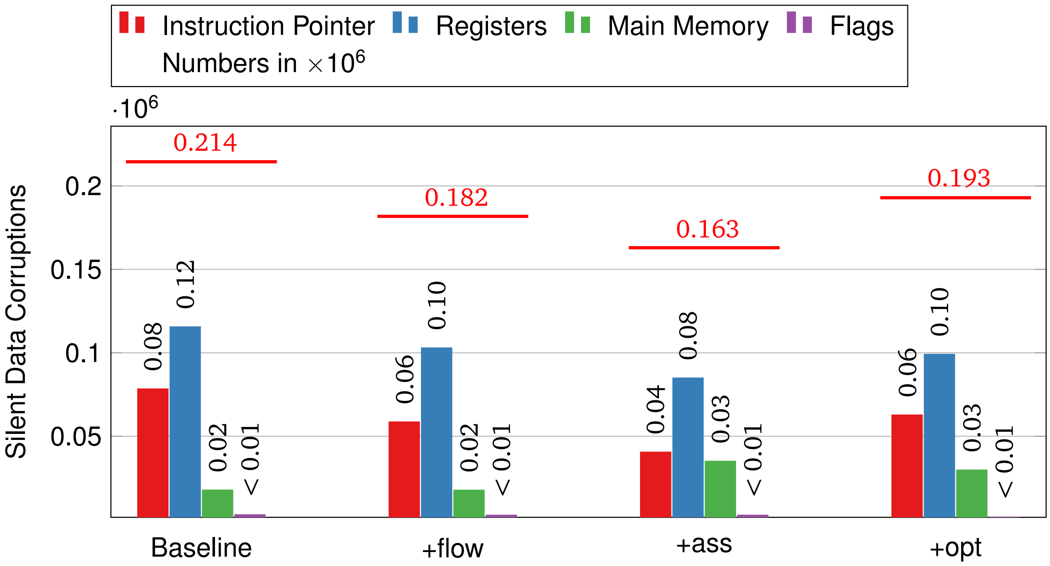


Figure A.2 – SDC Rates for the *encoded* *d*OSEK without Spatial Isolation.

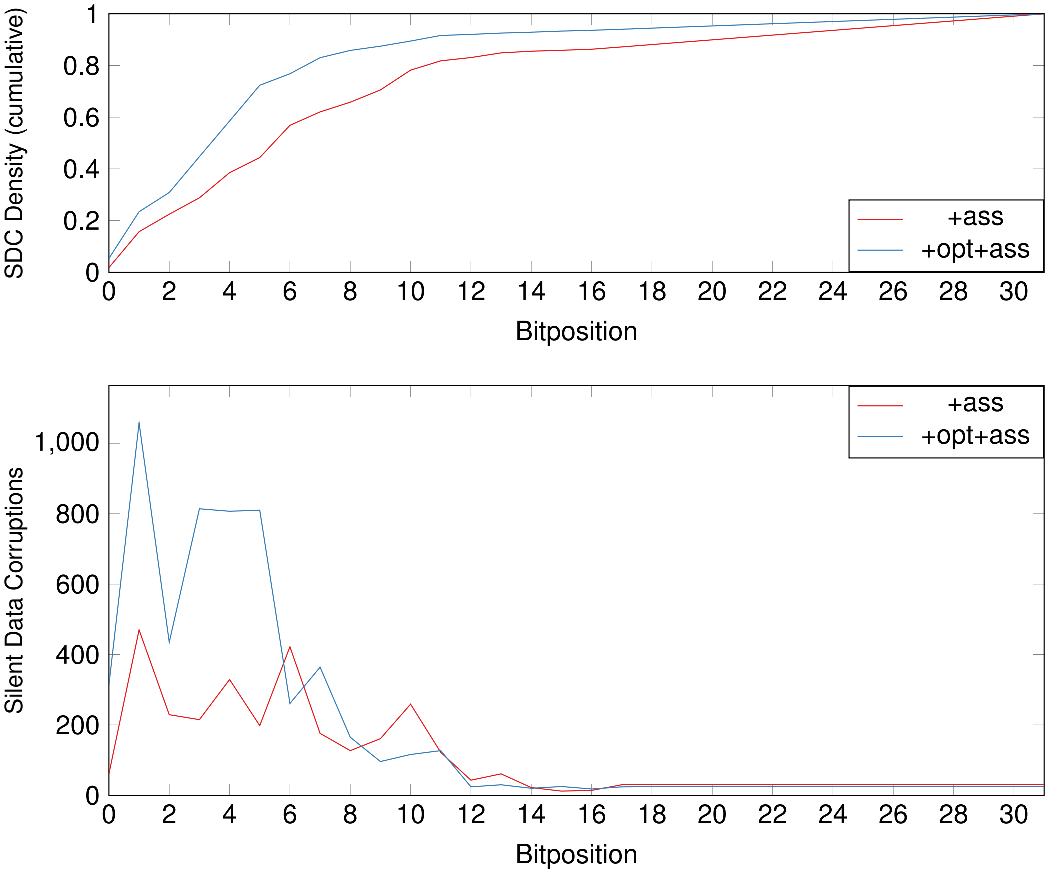


Figure A.3 – SDC Distribution for the Instruction Pointer for the unencoded *d*OSEK. When injecting faults into the instruction pointer, the system-call tailored variant shows a higher SDC rate for low bit offsets. Low bit offsets equal short jumps by a power of two.

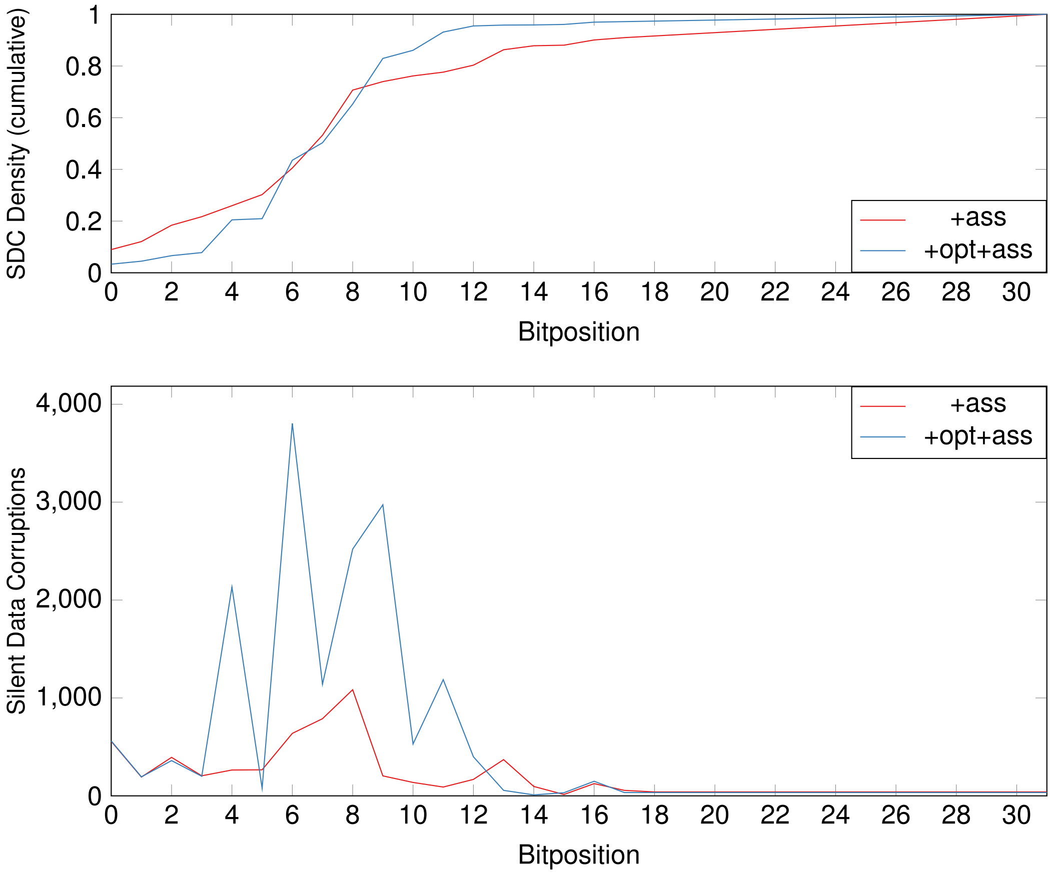


Figure A.4 – SDC Distribution for the Instruction Pointer for the encoded *d*OSEK. When injecting faults into the instruction pointer, the system-call tailored variant shows a higher SDC rate for low bit offsets. Low bit offsets equal short jumps by a power of two.

Appendix B

Reproducibility

The results presented in Chapter 5 were generated almost entirely by an automatic process. This automation allows not only an easy access to the raw data, which was used in the evaluation, but makes the results also more easy to reproduce. In this appendix, I want to describe the raw data, which can be obtained together with the LaTeX source of this document, and the formal description of the experiment scripts.

The experiment descriptions were written in the versuchung[9] framework. This framework allows the definition of formal experiment descriptions as Python code. Each experiment takes well-defined input parameters, instrument *d*OSEK and FAIL* to run with the given arguments, and output well-defined outputs. These outputs can be post-processed by other versuchung experiments. Figure B.1 show the data flow within those versuchung experiments for each of the three evaluation sections.

For each evaluation section, the experiments output a `data.tex` file. These files contain dataref[10] keys. dataref is a LaTeX package that allows the user to define symbolic data points and to reference them from within the document. A data point's value can be printed directly, it can be used within a calculation, or it can be used as an input parameter to a PGFPlot[11]. By using these techniques, all numbers, percentages and graphs are always up-to-date with the latest run of the experiment results.

The `GatherStatsDict` (see Figure B.1) experiment checks out a *d*OSEK source tree. It configures *d*OSEK with the given configuration vector and builds the *d*OSEK test suite and the *I4Copter* benchmark. For the Section 5.3, the experiment is furthermore instructed to record a golden run of the application.

The `CombineStatsDict` calculates the dataref keys, related to the code size used by the application. Furthermore the system state precision and other metrics for the analyses are extracted from the reports that are generated during the construction of *d*OSEK.

The `DOSEKStatistic` experiment analyses the golden-run traces for each variant and calculates average activation times and other metrics related to the dynamic behavior of the *d*OSEK applications.

The `FailTrace` experiment builds the *I4Copter* benchmark for various configurations. As a result, 16 golden-run traces and the corresponding system images are stored. These traces are imported into a MySQL database by the `FailImport` experiment.

---

[9] `https://www.github.com/stettberger/versuchung`

[10] `https://www.github.com/stettberger/dataref`

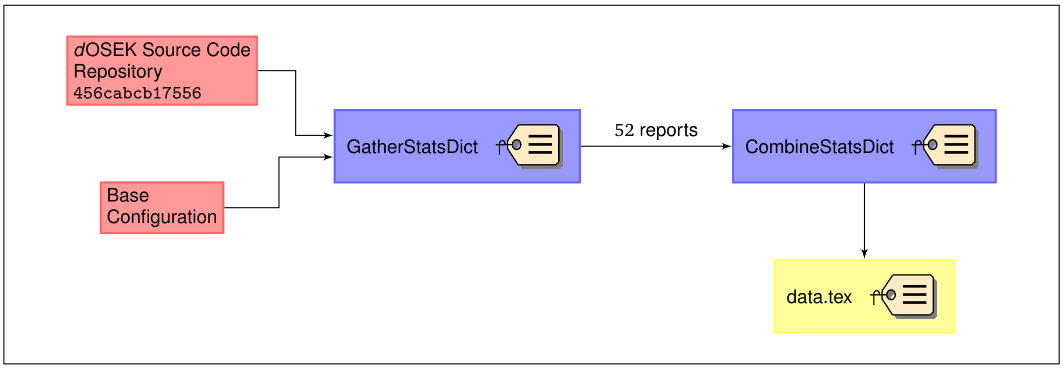
[11] `https://www.ctan.org/pkg/pgfplots`

The activation of FAIL* is the only step of the evaluation that was not codified with versuchung, since the computer cluster of the University of Erlangen-Nuremberg[12] was utilized. Operating this cluster required some manual worksteps that were not worth automating. FAIL* writes back the result of the fault-injection campaign into the database. The results are queried by a SQL script and the output is only transformed into dataref keys, but not further modified.

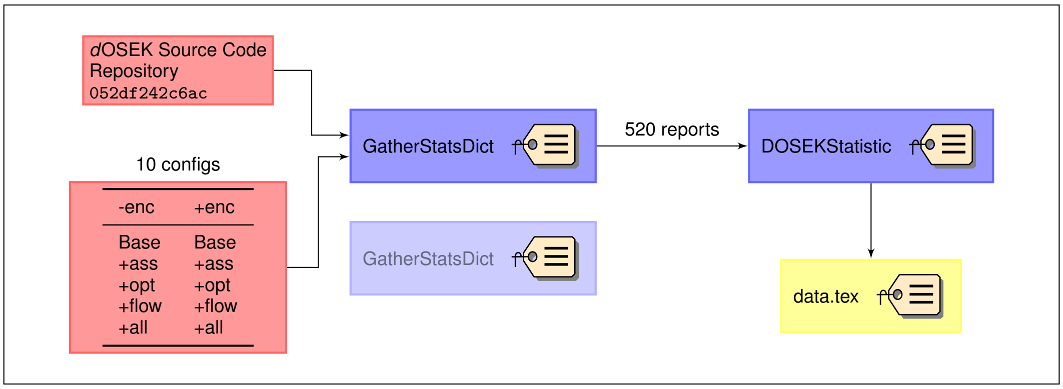
---

[12] `https://www.rrze.uni-erlangen.de/dienste/arbeiten-rechnen/hpc/systeme/`

Section 5.2 — `rawdata/analysis-result/`

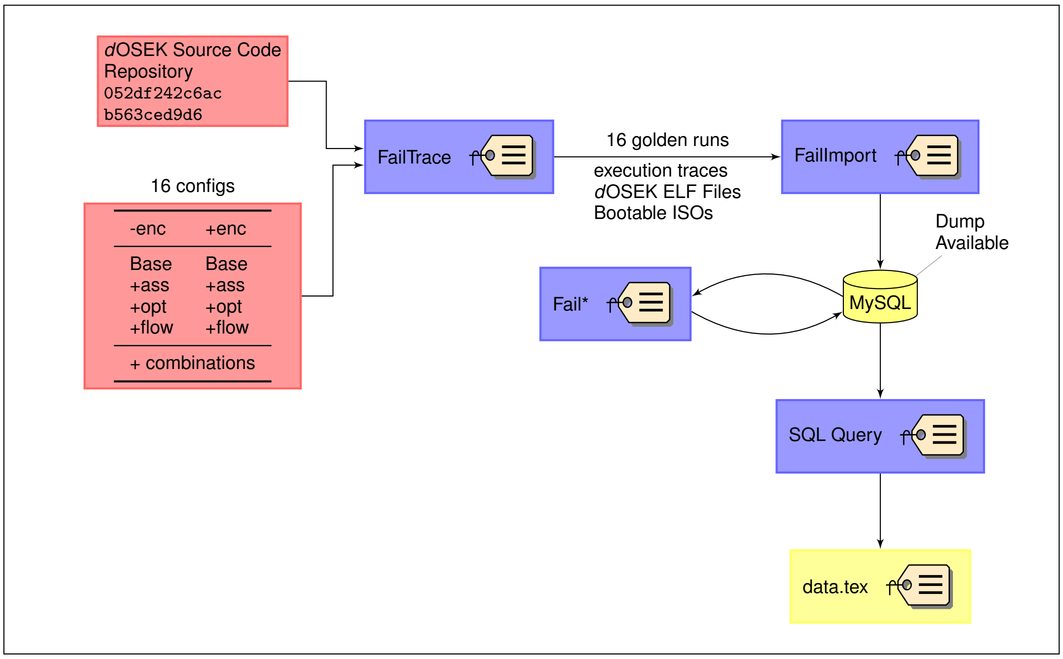


Section 5.3 — `rawdata/dosek-statistic/`



-enc	+enc
Base	Base
+ass	+ass
+opt	+opt
+flow	+flow
+all	+all

Section 5.4 — `rawdata/fault-injection/`



-enc	+enc
Base	Base
+ass	+ass
+opt	+opt
+flow	+flow
+ combinations	

Figure B.1 – Experiment Flow Chart. For each evaluation chapter, a separate experiment workflow was used. In the PDF version the Icons link to the experiment descriptions