# Generische und betriebssystemgewahre statische Analysen von Echtzeitapplikationen auf Ein- und Mehrkernsystemen zur Optimierung nichtfunktionaler Eigenschaften

Der Fakultät für Elektrotechnik und Informatik

der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR

(abgekürzt: Dr.-Ing.)

vorgelegte Dissertation

von Herrn

Gerion Entrup, M. Sc.

geboren am 23. Mai 1991

in Osterncappeln, Deutschland

2024

| | |
|---|---|
| 1. Referent | **Prof. Dr.-Ing. habil. Daniel Lohmann** |
| 2. Referent | **Prof. Dr.-Ing. Peter Ulbrich** |
| Tag der Promotion | **ausstehend** |

# Kurzfassung

Eingebettete Systeme sind in ein sie umgebendes System integrierte Rechensysteme. Im Gegensatz zu generischen Rechensystemen erfüllen sie damit nur eine, vor ihrer Inbetriebnahme bereits festgelegte Funktion. Oft sind an diese Funktion auch noch zeitlich kritische Bedingungen geknüpft, die das System zu einem Echtzeitsystem machen. Beides eröffnet ein Potential für statische Maßschneiderung, das das der generischen Systeme übersteigt. Statische Maßschneiderung benötigt immer eine vorhergehende statische Analyse, die der anschließenden Optimierung die notwendigen Informationen zuführt. In dieser Arbeit bewege ich mich in diesem Feld und entwerfe dazu verschiedene betriebsgewahre statische Analysen, die eine Echtzeitanwendung um die Verwendung des Echtzeitbetriebssystems analysieren. Sie haben dabei das primäre Ziel der anschließenden Maßschneiderung und Optimierung des Echtzeitbetriebssystems auf die Anwendung und die damit einhergehende Verbesserung der nichtfunktionalen Eigenschaften des Gesamtsystems.

Neben der Grundlagenvorstellung bettet der erste Teil der Arbeit dazu erstmals die verwandten Vorarbeiten in einen theoretischen Kontext, der deren Schwächen aufzeigt: Sie sind in verschiedener Kombination nur auf statische Systeme anwendbar, arbeiten nur mit genau einer Echtzeitbetriebssystemschnittstelle, können nur mit Systemen mit einem Kern umgehen oder liefern keine Informationen bezüglich möglicher Optimierung.

Mit ARA habe ich ein statisches Analyse-Framework geschaffen, das diese Probleme angeht: Es kombiniert verschiedene bestehende und neuentwickelte statische Analysen mit verschiedenen Echtzeitbetriebssystemmodellen und ist so in der Lage, eine Vielzahl von Echtzeitanwendungen automatisiert zu analysieren und passende Informationen für eine nachgelagerte Optimierung zu liefern. Konkret beinhaltet ARA die neu geschaffene statische Instanz- und Interaktionsanalyse, die Anwendungen untersuchen können, die gegen eine dynamische Echtzeitsystemschnittstelle geschrieben wurden. Sie ermitteln dort die Menge der dynamisch erstellten Betriebssystemobjekte und deren Interaktionen. Überdies ermöglicht die neu entwickelte MultiSSE eine vollständige abstrakte Analyse eines Mehrkernsystems. Sie bildet den dabei notwendigen abstrakten Betriebssystemzustand über alle Kerne nur an den Stellen, an denen dieser durch Kontrollflussinformationen oder eine Zeitanalyse notwendig ist. Durch die Trennung der Analysen in betriebssystemspezifische und generische Teile, sowie dem Entwurf einer gemeinsamen Schnittstelle zwischen verschiedenen Echtzeitbetriebssystemen konnte ich anschließend die Analysen betriebssystemagnostisch mit Anwendungen für AUTOSAR OS, FreeRTOS, Zephyr und POSIX durchführen und evaluieren.

Meine Forschungsgruppe hat verschiedene Synthesen entwickelt, die auf Basis der Analyseergebnisse ein optimiertes System erzeugt haben. Wir konnten dabei die Systemstartzeit bei dynamischen Systemen um bis zu 44% senken und eine Verbesserung des Schlupfes um bis zu 35% für Mehrkernsysteme erreichen. Ich konnte überdies die Wirksamkeit der Schnittstelle mit der Analyse von 8 Echtweltanwendungen zeigen. In der Domäne der eingebetteten Systeme zeigt diese Arbeit als Ergebnis somit die erfolgreiche Erweiterung von betriebssystemgewahren Analysen auf dynamische Systeme und Mehrkernsysteme sowie deren Generalisierung in einem gemeinsamen Framework, das den Vergleich und das Ausführen verschiedener Analysen mit Anwendungen verschiedener Echtzeitbetriebssysteme ermöglicht.

***Schlüsselwörter —*** betriebssystemgewahre Analyse, Mehrkernanalyse, Echtzeitsysteme, dynamische Systeme

# Abstract

Embedded systems are computing systems integrated into a surrounding system. In contrast to generic computing systems, they only fulfill a function that is strictly predefined before runtime. Often, it additionally contains time-critical constraints that make the system a real-time system. Both properties give a potential for static optimization that exceeds that of generic systems. Static tailoring always requires a prior static analysis that provides the necessary information for the subsequent optimization. In this work, I develop multiple such static analyses. They all analyze how an application uses the real-time operating system for a following tailoring of the operating system to the application resulting in an improvement of the non-functional properties of the overall system.

Besides foundational work, the first part of this thesis embeds the related existing analyses in a theoretical context for the first time, showing their weaknesses: In various combinations, these analyses only work with static systems, are only able to understand exactly one real-time–operating-system interface, can only deal with systems with one core or do not provide any information regarding possible optimization.

With ARA, I have created a static analysis framework that addresses these problems: It combines various existing and newly developed static analyses with different real-time–operating-system models and is thus able to automatically analyze a variety of applications providing information for a following optimization. Specifically, ARA includes the newly created static instance and interaction analysis, two analyses that can analyze applications that are written against a dynamic real-time–operating-system interface. They determine the number of dynamically created operating-system objects and their interactions. With the MultiSSE, I have also developed an analysis that allows a complete abstract analysis of a multi-core system. It creates the necessary abstract operating-system state across all cores only at those points where it is necessary due to control flow information or a time analysis. By separating the analyses into an operating-system–specific and a generic part, as well as the design of a common interface between different real-time operating systems, I was additionally able to apply the analyses in an operating-system–agnostic way with applications for AUTOSAR OS, FreeRTOS, Zephyr and POSIX.

My research group has developed various syntheses for creating an optimized system based on the analyses results. We were able to lower the system-start time for dynamic system up to 44% and an get an improvement of slack of up to 35% for multi-core systems. Furthermore, I was also able to demonstrate the effectiveness of the interface by analyzing 8 real-world applications. In the embedded-systems domain, this thesis thus demonstrates the successful extension of operating-system–aware analyses to dynamic systems and multi-core systems as well as their generalization in a common framework that enables the comparison and execution of different analyses with applications of different real-time operating systems.

***keywords —*** operating-system aware analysis, multi-core analysis, real-time systems, dynamic systems

# Inhalt

# 1

# Einleitung

## Motivation und Kontext

*“To be is to do” – Socrates.*
*“To do is to be” – Jean-Paul Sartre.*
*“Do be do be do” – Frank Sinatra.*

*Kurt Vonnegut aus „Deadeye Dick“ (1982), basierend auf einem Graffiti in Richardson, Texas*

## 1.1 Motivation

Als John Bardeen, William Shockley und Walter Brattain 1947 den ersten baubaren Bipolartransistor erfanden, schufen sie damit die Grundlagen für eine rapide Verkleinerung der in den folgenden Jahrzehnten entstandenen Rechenmaschinen [BB48,Nob23]. Mithilfe moderner Mikroelektronik ist es inzwischen möglich, alle möglichen Geräte mit programmierbaren Bausteinen auszurüsten, was zu einem Blumenstrauß an programmierbarer Kleinstelektronik geführt hat (z.B. Glühlampen, Kühlschränke, Heizungsthermostate aber auch Assistenzsysteme in Fahrzeugen, Schleusensteuerungen, Signalsysteme), deren Arten alle zu eigen haben, dass das enthaltene Rechensystem keinem Selbstzweck dient, sondern in ein größeres Gerät eingebettet ist. Diese Systeme werden darum *eingebettete Systeme* genannt [Mar21A1.1]. Eingebettete Systeme stellen einen wachsenden Markt dar, der von einer Marktgröße in 2021 von 39,4 Mrd. USD bis 2027 auf 63,7 Mrd. USD anwachsen soll [Gro22].

*Echtzeitsysteme*

Eng verwandt mit eingebetteten Systemen sind die *Echtzeitsysteme*, die dadurch definiert sind, dass sie in einer festgesetzten Zeitspanne („*Frist (Deadline)*“) reagieren müssen, da andernfalls fatale Folgen eintreten [Liu00A2.3, Kop11A1.1, Die19A2.1]. Diese Systeme kontrollieren beispielsweise Steuerungen wie die Fluglageregelung eines Quadrokopters, der nicht rechtzeitig genug reagieren und abstürzen könnte, wenn er seine Frist nicht einhält. Da Echtzeitsysteme üblicherweise in ein größeres System eingebettet sind, bilden sie eine Unterkategorie innerhalb der eingebetteten Systeme [Kop11A1.6.1,Mar21A1.3].

Eingebettete Systeme werden von Software gesteuert, der *Anwendung*, die oftmals ein Betriebssystem als Hilfsbibliothek benutzt. Um auch Echtzeitsystemen gerecht zu werden, sind dies oftmals *Echtzeitbetriebssysteme*.

*(nicht-) funktionale Eigenschaften*

Zur Erfüllung ihrer Aufgabe müssen eingebettete Systeme eine spezifizierte Menge an Eigenschaften erfüllen, die *funktionalen Eigenschaften*. Ebenso große Bedeutung haben aber auch die *nichtfunktionalen Eigenschaften*, eine Menge aller Eigenschaften, die für die korrekte Erfüllung der Aufgabe irrelevant sind, wie z. B. der Speicherverbrauch, der Energieverbrauch oder die Anwendungsgröße [Loh14A6.1,Die19A2.1]. Diese werden maßgeblich durch die Software beeinflusst, beschränken aber entweder die verwendbare Hardware („dieser Microchip ist nicht stark genug“) oder beeinflussen maßgeblich das Benutzererlebnis. So finden oftmals Firmwareupdates ein größeres Presseecho, die nichtfunktionale Eigenschaften wie die Akkulaufzeit verbessern [Gar22,App22].

*Maßschneiderung*

Verbesserungen der nichtfunktionalen Eigenschaften werden oftmals durch *Maßschneiderung* erreicht: Die Software wird konkret auf die Hardware angepasst, verwendet deren Spezialinstruktionen (z. B. gesonderte Vektoroperation), vermeidet überflüssige Anweisungen (z. B. das Initialisieren überflüssiger Hardware) oder inkludiert bereits vorausberechnete Ergebnisse (z. B. ausgerechnete Konstanten) [ALS+08A1.2.5+9+11,Loh14A1].

Ein großer Teil dieses Prozesses ist mühsam, immer gleich und eignet sich hervorragend zum Automatisieren. Dies wurde schon in den Sechzigern erkannt, als mit dem IBM Fortran H einer der ersten optimierenden Compiler vorgestellt wurde (nur sechs Jahre nach der Vorstellung der Sprache Fortran selbst) [SK80].

Eingebettete Systeme haben meistens durch ihre Einbettung eine vor ihrer Inbetriebnahme festgelegte Aufgabe [Mar07A1.1]. Um die Aufgabenerfüllung gewährleisten zu können, muss auch die gesamte Software, die auf dem Gerät laufen wird, bereits zu dieser Zeit bekannt sein. Dementsprechend gut eignet sich Software für eingebettete Systeme für eine *Whole-Program-Optimization*, also eine Optimierung, die das System im Gesamten berücksichtigt [BEG+07]. Ein spezieller Teilbereich dieser Optimierung beschäftigt sich dabei mit der Maßschneiderung des Betriebssystems auf die Anwendung. Mit dOSEK [HLD+15b,DHL17] oder dem RTSC [SS10] existieren z. B. zwei Whole-System-Compiler, die durch Entfernung von Systemaufrufen, Vorberechnung der Scheduling-Entscheidungen oder Aufteilen auf mehrere Kerne das Gesamtsystem beschleunigen und verkleinern können.

*statische Analyse*

Die grundlegende Architektur ist dabei bei allen Übersetzern gleich: Einer jeden Optimierung geht ein entsprechender Analyseschritt voraus, der die notwendigen Informationen automatisch extrahiert und damit die Optimierung überhaupt ermöglicht. Da diese Analyse das zu übersetzende Programm nicht ausführt, sondern Rückschlüsse dadurch zieht, den Programmtext zu betrachten, wird sie *statische Analyse* genannt [RY20A1.3.2].

Die oben genannten Übersetzer sind auf den OSEK-Echtzeitbetriebssystemstandard zugeschnitten, der von der Anwendung verlangt, alle Betriebssystemobjekte statisch (also vor der Laufzeit) festzulegen, nur auf Einkern-Systemen läuft und genau ein Scheduling-Modell unterstützt [OSE05]. All dies kommt der Analyse in den Übersetzern zugute, die dadurch auf einer Betriebssystemschnittstelle operieren können, bei der bereits große Teile des Systemverhaltens zur Übersetzungszeit festgeschrieben sind.

*Dynamische Betriebssysteme*

Die Verfahren verlieren aber dadurch an Allgemeingültigkeit: Im Gegensatz zu OSEK ist beispielsweise im Echtzeitbetriebssystem Zephyr sowohl eine statische als auch eine dynamische Erstellung von Betriebssystemobjekten möglich [Pro23b]. Das Echtzeitbetriebssystem FreeRTOS wiederum lässt nur die dynamische Konfiguration zu [Ama23]. Die Analyse nur auf statische Betriebssysteme auszulegen, verhindert damit bereits konzeptuell die Untersuchung von Anwendungen für andere Betriebssystemschnittstellen, die aber gerade in der eingebetteten Domäne ebenso verwendet werden[1].

*Mehrkernsysteme*

OSEK ist 2003 im AUTOSAR-Standard aufgegangen, der das Betriebssystem um partitionierte Mehrkernfähigkeit erweitert [AUT13]. Mehrkernunterstützung ist in anderen Bereichen (Server, HPC, Desktop [TOP23, Val23]) längst angekommen und wird auch

[1] Nach [EA19] ist Linux (POSIX) das am häufigsten eingesetze Betriebssystem im eingebetteten Bereich gefolgt von FreeRTOS, beide vollständig dynamisch.

für eingebettete Systeme immer wichtiger. So haben von den 133 ARM basierten SoCs von Texas Instruments (für eingebettete Systeme der beliebteste Prozessorhersteller [EA19]) 86 der Hauptprozessoren mehrere Kerne oder zumindest einen oder mehrere Coprozessoren [Inc23]. Gerade im Bereich der Echtzeitsysteme fehlen aber passende Analysen. Das Hauptproblem dabei ist der fehlende Determinismus der Kerne zueinander. Konkret ist unklar, an welcher Position ein Kern im Vergleich zu den anderen ist. Die naive Lösung dieses Problems ist die Berechnung des Kreuzproduktes aller Kernpositionen, die jedoch durch ihr exponentielles Wachstum praktisch ungültig ist [Min12]. In bisherigen Ansätzen wurde darum die Flusssensitivität für den Interaktionsbereich aufgegeben [Min12] oder modellbasiert gearbeitet [HBR21,HBR22,MB21]. Diese Ansätze verwerfen allerdings Informationen, um skalierbar zu sein.

*Betriebssystem-spezifische Analyse*

Die Übersetzer RTSC und dOSEK, aber auch andere betriebssystemgewahre Analyseprogramme wie GOBLINT [VV09] unterstützen nur eine Betriebssystemschnittstelle. Dies beschränkt die implementierten Algorithmen zwar konzeptuell nicht zwangsläufig auf genau dieses Betriebssystem, praktisch aber oft schon. Es birgt außerdem die Gefahr, den Algorithmus auf bestimmte Konzepte des Betriebssystems anzupassen, ihn dadurch aber auch davon abhängen zu lassen. Die später vorgestellte SSE-Analyse [DL17], die von dOSEK implementiert wird, ist beispielsweise inhärent nur auf statische Systeme anwendbar.

## 1.2 Forschungskontext

*Methodik der Spezialisierung*

Diese Arbeit findet im Forschungskontext des DFG-Projektes „Automated Hardware Abstraction in Operating-System Engineering“ (LO 1719/4-1) statt. Ziel des Projektes ist die Grenzen möglichst automatischer Maßschneiderung bzw. Spezialisierung des Betriebssystems und ggf. auch der darunterliegenden Hardware auf eine vorgegebene Anwendung eines eingebetteten Systems auszuloten. Dabei geht es zum einen darum, einen möglichst hohen Grad der Maßschneiderung zu erreichen, diesen aber anschließend auch variieren zu können. Der grundlegende Gedanke lässt sich in folgender Formel zusammenfassen [›FED+18][2]:

$$RTS(\vec{I}) = \vec{O} \overset{RTS}{=} RTS^{A_{RTOS}}_{HW}(\vec{I})$$

Wir gehen dabei von einer Spezifikation für ein *Echtzeitsystem (Real-Time System, RTS)* aus, das eine Menge von Eingabeereignissen $\vec{I}$ erhält und auf diese mit einer definierten Ausgabe $\vec{O}$ reagieren soll. Die Spezifikation kann nun von einer konkreten Implementierung $RTS^{A_{RTOS}}_{HW}(\vec{I})$ erfüllt werden. In diesem Fall gilt der spezielle Gleichheitsoperator $\overset{RTS}{=}$, der eine Implementierung mit der Spezifikation vergleicht. Die konkrete Implementierung besteht

[2] Ich gehe hier nur kurz auf die Formel ein. Mein Kollege Björn Fiedler beschreibt sie deutlich detaillierter in seiner Dissertation [Fie23A2.2].

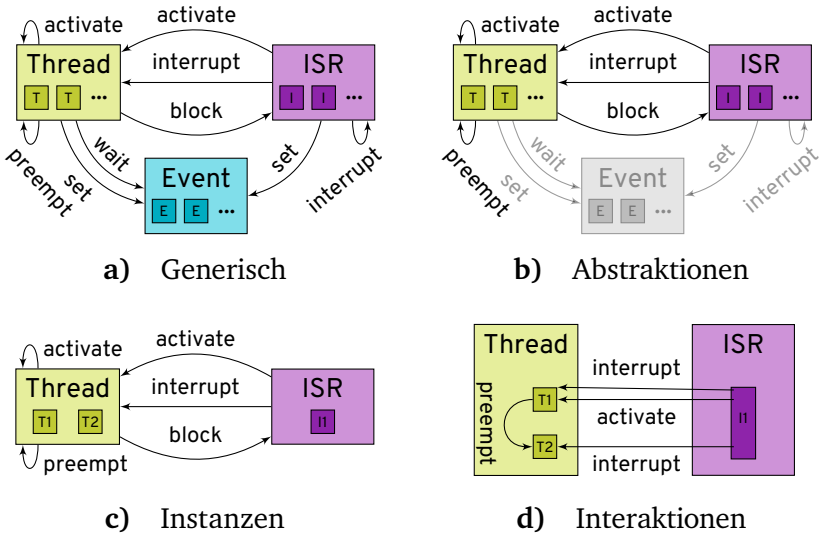



**Abbildung 1.1** Die verschiedenen Optimierungslevel für eingebettete Systemsoftware. Das generische Level beschreibt die vollständige unoptimierte Betriebssystemschnittstelle. Bei b) kann auf der Ebene ganzer Betriebssystemabstraktionen spezialisiert werden. c) beschreibt die Spezialisierung einzelner Instanziierung von Betriebssystemobjekten, während im Level d) die genauen Interaktionen zwischen Instanzen flusssensitiv bekannt sind und spezialisiert werden können (adaptiert aus [›FED+18]).

aus einer *Anwendung (A)*, einem *Echtzeitbetriebssystem (Real-Time Operating System, RTOS)* und einer unterliegenden *Hardware (HW)*. Für alle diese Komponenten gilt: Sie können beliebig angepasst (maßgeschneidert) werden, solange sie der Spezifikation entsprechen, also die Gleichheit erfüllt bleibt.

*automatische Spezialisierung*

Ziel des Forschungsprojektes ist die *automatische* Maßschneiderung. Konkret haben wir dazu einen Whole-System-Compiler entwickelt, der als Eingabe eine Implementierung einer Anwendung in einer Hochsprache als Spezifikation annimmt und daraus ein maßgeschneidertes konkretes System $RTS^{\frac{A}{RTOS\ HW}}$ erzeugt. Auch hier gliedert sich der Mechanismus in eine Synthese und eine vorhergehende Analyse, bei der grundsätzlich gilt, dass sie die für die Synthese notwendigen Informationen liefern soll. In dieser Dissertation werde ich auf die Analyse eingehen. Die verschiedenen Synthesen hat zum einen Björn Fiedler in seiner Dissertation erarbeitet [Fie23] und zum anderen Andreas Kässens für seine in Arbeit befindliche Dissertation.

*Grade der Spezialisierung*

Generell gilt für die Analyse, dass sie alle Informationen liefern muss, die die Synthese benötigt, also insbesondere auch eine Übermenge an Informationen liefern kann. Wir haben daher als Teil des Projektes ebenfalls eine Klassifikation an Spezialisierungsgraden (und damit auch Analysegraden) erstellt, die in Abbildung 1.1 dargestellt ist:

**Grad der Abstraktionen:** Jedes Betriebssystem bietet eine Fülle von Abstraktionen, wie Fäden, Semaphoren oder Signale. Je nach Anwendung werden diese Abstraktionen aber nie verwendet, was zur Optimierung genutzt werden kann. Wenn eine automatisierte Analyse feststellt, dass eine Anwendung eine Abstraktion nie verwendet, kann diese für die Anwendung vollständig aus dem Betriebssystem ausgebaut werden [TKH+12, RHL14].

**Grad der Instanzen:** Viele Betriebssystemabstraktionen können instanziiert werden. Die Optimierung hier gilt dann für diese konkreten Instanzen oder ihre (flussinsensitiven) Interaktionsmuster. Stellt eine Analyse dabei z. B. fest, dass genau zwei instanziierte Fäden einmal ausschließlich lesend und einmal ausschließlich schreibend auf eine bestimmte instanziierte geteilte Liste zugreifen, kann die dahinterliegende Datenstruktur entsprechend optimiert werden [Fie23A5.1]. Es gibt weitere Optimierungen, die Speicher einsparen oder die Robustheit gegen Angriffe erhöhen [HBD+14,HLS+09,HDM+12].

**Grad der Interaktionen:** Die Instanzen, z. B. zwei Fäden, interagieren miteinander an fest definierten Stellen im Programmkontrollfluss: den Systemaufrufen. Eine Analyse kann hier beispielsweise feststellen, dass das Senden eines Signals an einer definierten Stelle im Kontrollfluss *immer* dazu führt, dass ein ganz bestimmter Faden aufgeweckt und zudem eingeplant wird. In diesem Fall kann diese Planungsentscheidung in das System fest eincodiert werden und so die Zeit senken, die das Betriebssystem zur Berechnung braucht [DHL17].

Der hier vorgestellte Grad ist ansteigend, sodass eine Analyse, die Interaktionsinformationen liefert, automatisch auch Synthesen ermöglicht, die Abstraktions- oder Instanzinformationen benötigen. Bisherige Werkzeuge lassen sich in die oben beschriebenen Kategorien einsortieren, sind aber entweder auf ein spezielles Betriebssystem zugeschnitten oder konzentrieren sich auf spezielle Optimierungen. Zentraler Punkt des Forschungsprojektes ist im Gegensatz dazu eine Vergleichbarkeit verschiedener Methoden zu schaffen, die es ermöglicht, *mit dem gleichen Werkzeug* Systeme verschiedener Betriebssysteme auf gleiche Art zu optimieren bzw. ein System in verschiedenen Arten bzw. Graden zu optimieren. Ein wichtiger Baustein ist dabei die optimale Ausnutzung von Mehrkernsystemen, insbesondere bei der Optimierung auf Basis von Interaktionen, da diese kein Bestandteil vorheriger Arbeiten sind. Ich habe in dieser Dissertation die Grundlagen für eben diese Vergleichbarkeit gelegt, indem ich ein Analyse-Framework geschaffen habe, das betriebssystemagnostisch Systeme in verschieden Graden und auch Mehrkernsysteme analysieren kann.

## 1.3 Zweck dieser Arbeit

*Anwendung zur Demonstration*

Um die Problematik etwas plastischer darzustellen, will ich etwas vorgreifen und ein Beispielsystem mit möglichen Optimierungen zeigen, die in verwandten Arbeiten nicht

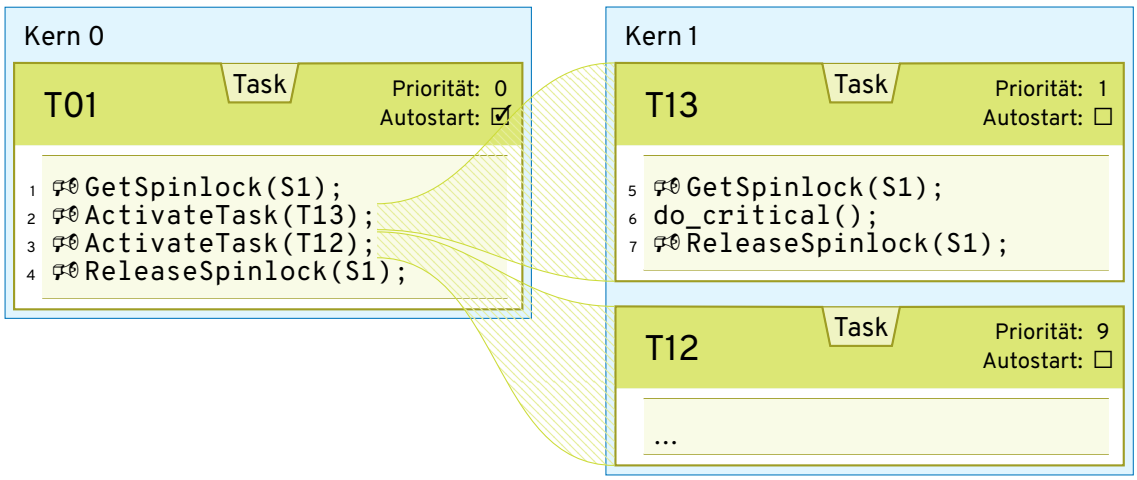



**Abbildung 1.2** Eine Echtzeitanwendung, die mithilfe der Echtzeitbetriebssystemschnittstelle AUTOSAR geschrieben wurde. Zu sehen sind drei AUTOSAR-Tasks (Fäden) verschiedener Priorität (eine größere Zahl ist höherprior), die auf zwei Kernen laufen. *T01* startet beim Systemstart und aktiviert („setzt lauffähig“) anschließend die beiden anderen. `ActivateTask(T12)` wird in dieser Anwendung niemals ein Umplanen („Reschedule“) auslösen und braucht deswegen keinen Inter-Prozessor-Interrupt.

| Eigenschaft | dOSEK | RTSC | GOBLINT | Astrée | ARA |
|---|---|---|---|---|---|
| Mehrkernunterstützung | ✗ | ✗ | ✓ | ✓ | ✓ |
| Betriebssystemagnostisch | ✗ | ✗ | ✗ | ✓ | ✓ |
| Dynamisches System | ✗ | ✗ | ✓ | ✓ | ✓ |
| Quelloffen | ✓ | ✓ | ✓ | ✗ | ✓ |
| Für Maßschneiderung (Compiler) | ✓ | ✓ | ✗ | ✗ | ✓ |

**Tabelle 1.1** Vergleich verschiedener betriebssystemgewahrer statischer Analysen

existieren. Abbildung 1.2 zeigt dazu eine Echtzeitanwendung, die für ein Zielsystem mit zwei Kernen und auf Basis der AUTOSAR-Echtzeitbetriebssystemschnittstelle geschrieben wurde. Konkret aktiviert der Task *T01* auf Kern 0 hier den Task *T13*, der daraufhin vom ersten Kern parallel ausgeführt wird. Dadurch, dass anschließend die Sperre von Task *T01* gehalten wird, ist Task *T13* in jedem Fall noch am Laufen, wenn Task *T12* aktiviert wird. Der Betriebssystemstandard schreibt in diesem Fall vor, dass – wenn nötig – ein Neuplanen auf allen Kernen ausgeführt werden muss. Dazu muss der entsprechende Kern benachrichtigt werden, was in einem teuren Inter-Prozessor-Interrupt resultiert [HCK+03, ›EKF+24]. Mit der geeigneten Analyse kann gezeigt werden, dass dieser Interrupt überflüssig ist und entsprechend optimiert werden kann. Die Analyse, die dieses Wissen extrahiert, muss einerseits die Systemaufrufe der Betriebssystemschnittstelle interpretieren, andererseits

korrekte Rückschlüsse über das Scheduling-Verhalten ziehen und zuletzt auch noch über die Interaktionen zwischen den Kernen schlussfolgern können. Sie muss außerdem den Kontrollfluss innerhalb des Kerns und die Systemaufrufwerte korrekt erkennen.

*Forschungsfragen*

Mit dieser Dissertation habe ich ein Framework – ARA – geschaffen, dass alle diese Analyseaufgaben erfüllt, verschiedene Analysegrade zulässt und alle Analysen überdies betriebssystemagnostisch ausführt. Mit diesem Framework will ich die oben genannten Probleme angehen: Mehrkernfähigkeit für Optimierung nicht gegeben, Einschränkung der Analyse auf statische Betriebssysteme, Vergleichbarkeit von mehreren Betriebssystemen nicht möglich. Konkret führen mich diese Probleme zu den folgenden Forschungsfragen.

**Forschungsfrage 1: Welche Herausforderungen entstehen auf eingebetteten dynamischen Systemen und können betriebssystemgewahre Analysen diese überwinden?**

**Forschungsergebnis:** Ich habe zur Beantwortung dieser Frage die *statische Instanzanalyse (Static Instance Analysis, SIA)* entwickelt, die eine betriebssystemgewahre Analyse für dynamische Systeme darstellt und dort konkret Instanzen von Betriebssystemobjekten und deren Interaktionen (in einer flussinsensitiven Darstellung) extrahiert. Die SIA ihrerseits gliedert sich in ein flusssensitives und flussinsensitives Verfahren auf, die sich hinsichtlich Präzision und Komplexität unterscheiden, zusammen mit einem kombinierten Verfahren, das die Vorteile beider vereint. Sie liefert dabei Ergebnisse auf Ebene der Instanzspezialisierung, analysiert den Code aber wahlweise auf Ebene der Interaktionen. Die Resultate der SIA konnten Björn Fiedler und ich zusammen mit einer passenden Synthese evaluieren und eine Verringerung der Startgeschwindigkeit von 44% erreichen.

Relevante Veröffentlichung: [›FED+21]

**Forschungsfrage 2: Ist eine betriebssystemgewahre abstrakte Interpretation auf Mehrkernsystemen ohne Potenzmengenbildung durchführbar?**

**Forschungsergebnis:** Diese Forschungsfrage konnte ich durch die Entwicklung der MultiSSE bejahen. Die MultiSSE setzt die abstrakte Interpretation für Mehrkernsysteme um, umgeht die Potenzmengenbildung aber durch die Zustandstrennung in Zustände, die von anderen Kernen unabhängig auf einem singulären Kern auftreten und expliziten Synchronisationspunkten, die zwei oder mehrere Kerne in ihrer zeitlichen Relation zueinander determinieren. Mithilfe der MultiSSE konnte ich zusätzliche Optimierungen finden, die nur im Kontext von Mehrkernsystemen Sinn ergeben, aber auf der Ebene von Interaktionen spezialisieren, also auch eine solche Analyse voraussetzen.

Relevante Veröffentlichung: [›EFL23,›EKF+24]

**Forschungsfrage 3: Welche Gemeinsamkeiten und Unterschiede haben Echtzeitbetriebssystemschnittstellen und wie kann man Analysen davon unabhängig machen?**

**Forschungsergebnis:** Um Gemeinsamkeiten und Unterschiede der bestehenden und neu geschaffenen Analysen aufzuzeigen, habe ich diese (erstmals) in einen theoretischen

Kontext gebettet. Mit einer zusätzlichen Klassifizierung der Echtzeitbetriebssystemschnittstellen in analyserelevante Faktoren konnte ich eine *Analyse-RTOS-Schnittstelle* entwickeln, die Analysen und Betriebssystemschnittstellen trennt. All das konnte ich so in ein gemeinsames Framework – ARA – einbetten, das damit betriebssystemagnostisch statische betriebssystemgewahre Analysen durchführen kann. Innerhalb des Frameworks schaffen konkrete Implementierungen der Schnittstelle für AUTOSAR, FreeRTOS, Zephyr und POSIX die Unterstützung für Anwendungen auf Basis dieser Systeme, was ich mit der Analyse von 8 Echtweltanwendungen nachweisen konnte.

Relevante Veröffentlichungen: [›ESD19,›ENL22]

## 1.4 Aufbau der Arbeit

Die Arbeit besteht aus zwei Teilen: Im Teil I: „Grundlagen“ stelle ich bestehende Techniken vor und ordne diese in einen theoretischen Kontext ein. Der Teil gliedert sich dazu in vier Kapitel:

Kapitel 2: „Eingebettete und Echtzeitsysteme“ gibt einen Einblick in meine Forschungsdomäne. Dieses Kapitel stellt die wichtigsten Konzepte aus dem Bereich und auch die konkreten Echtzeitbetriebssystemschnittstellen vor, die in dieser Arbeit Verwendung finden.

Kapitel 3: „Statische Analyse“ gibt einen Einblick in meine Forschungsmethode. Es wird die theoretischen Grundlagen der in dieser Arbeit verwendeten Analysen vorstellen und allgemeine Vorteile und Herausforderungen herausarbeiten.

Kapitel 4: „Betriebssystemgewahre Analyse“ verbindet Methode und Domäne zum Mechanismus der betriebssystemgewahren Analyse und stellt den aktuellen Stand der Kunst auf diesem Bereich vor. In diesem Kapitel werde ich weiterhin einige vorhandene Analysen theoretisch einordnen und damit einen Baustein zur Beantwortung der 3. Forschungsfrage liefern.

Kapitel 5: „ARA“ stellt alle Teile des neu geschaffenen ARA-Frameworks vor, die für die eigentlichen betriebssystemgewahren Analysen notwendig sind, aber bereits bestehende Verfahren verwenden.

Anschließend geht der Teil II: „Konzepte“ detailliert mit drei Kapiteln auf meine Forschungsfragen und die dafür neu geschaffenen Analysen ein:

Kapitel 6: „Statische Analyse dynamischer Systeme“ stellt die *statische Instanzanalyse (Static Instance Analysis, SIA)* vor, eine Analyse, die ich zur Beantwortung meiner 1. Forschungsfrage geschaffen habe und zeigt deren Evaluation an zwei Echtweltsystemen.

Kapitel 7: „Die Analyse von Mehrkernsystemen“ geht auf die MultiSSE ein, eine Analyse, die die 2. Forschungsfrage beantwortet, zeigt deren Funktionsweise zeigt, stellt eine Synthese auf ihrer Basis vor und evaluiert und diskutiert die Analyse anschließend.

Kapitel 8: „RTOS-Agnostische Analyse“ gliedert die SIA und MultiSSE anschließend in ARA ein und macht diese unabhängig vom konkreten RTOS. Es beantwortet dabei meine 3. Forschungsfrage.

Im Kapitel 9: „Zusammenfassung“ fasse ich abschließend meine Forschungsergebnisse zusammen und gebe einen kleinen Ausblick.

## 1.5 Typographische Konventionen

Um Patterson and Hennessy zu zitieren [PH21A1.1]:

> *Computer Designers (including the authors)* love *using acronyms, which are* easy *to understand once you know what the letters stand for!*

Aus diesem Grund ist selbstverständlich auch diese Arbeit gespickt von Akronymen. Diese sind aber oft eine Abkürzung für die englischen (Original-)Begriffe. Ich werde daher bei Akronymen immer die englische Version verwenden, aber die deutsche Variante für die Langversion[3]. Bei der Einführung eines Begriffs werde ich sowohl die deutsche als auch englische Version nennen.

Literaturreferenzen sind in eckigen Klammern gesetzt. Sollten diese auf Bücher, Dissertationen und Habilitationen verweisen, sind diese gesondert mit einer Abschnittsmarkierung versehen, wo in dem entsprechenden Werk die gewünschte Information aufzufinden ist, z. B. befindet sich bei [Mar21A1.1] die Referenz in dem Buch „*Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*“ von Peter Marwedel in Abschnitt 1.1. Referenzen zu bereits veröffentlichten Arbeiten, bei denen ich Coautor bin, sind zusätzlich mit „›“ gekennzeichnet (z. B. [›ENL22]).

Einige Konzepte sind gesondert gesetzt:

- „system_aufruf()“ ist ein Systemaufruf (Definition 4).
- „*T01*“ und „*GPSQueue*“ kennzeichnen Instanzen (Definition 2).
- „CALL“ ist ein Typ eines Atomaren Basisblocks (Definition 18).
- „main“ ist eine Funktion.

[3] Manche deutschen Fachwörter erscheinen ungewohnt. Ich habe mich hier an Schröder-Preikschat orientiert [Sch19].

# Teil I
# Grundlagen

Diese Arbeit kombiniert die Methode der statischen Analyse mit der Domäne der eingebetteten Echtzeitsysteme. Abbildung 3 stellt die Inhalte des Grundlagenteils in seiner Gesamtheit dar. Die statische Analyse ist eine Methode, um Informationen aus Systemen zu extrahieren, *ohne* diese auszuführen [RY20A1.3.2]. Je nach gewünschten Informationen gibt es eine Vielzahl von Analysen von unterschiedlichen Laufzeiten und Komplexitäten. In dieser Arbeit werde ich mich mit dem Teilgebiet der *abstrakten Interpretation* beschäftigen [CC77]. Diese verwende ich dazu, Anwendungen eingebetteter Systeme hinsichtlich der Verwendung des Betriebssystems zu untersuchen.

Eingebettete Echtzeitsysteme sind Rechensysteme, die in ein sie umgebendes System eingebettet sind. Sie haben darum einen festen Einsatzzweck und einen Fokus auf Zuverlässigkeit und Sicherheit [Mar21A1.1]. Ersterer gestattet und letztere erfordern ein möglichst deterministisches Design, das diese Systeme analysierbar macht.

Ich werde in diesem Teil zuerst eingebettete Systeme (Kapitel 2) und die allgemeine Technik der statischen Analyse vorstellen (Kapitel 3). Anschließend werde ich auf den Stand der Kunst eingehen, beide zu kombinieren (Kapitel 4). In Kapitel 5 geht es schlussendlich um das ARA-Framework – die konkrete Anwendung und Adaptierung der davor angesprochenen Konzepte, die die Grundlage für die im nächsten Teil vorgestellten Analysen bildet.

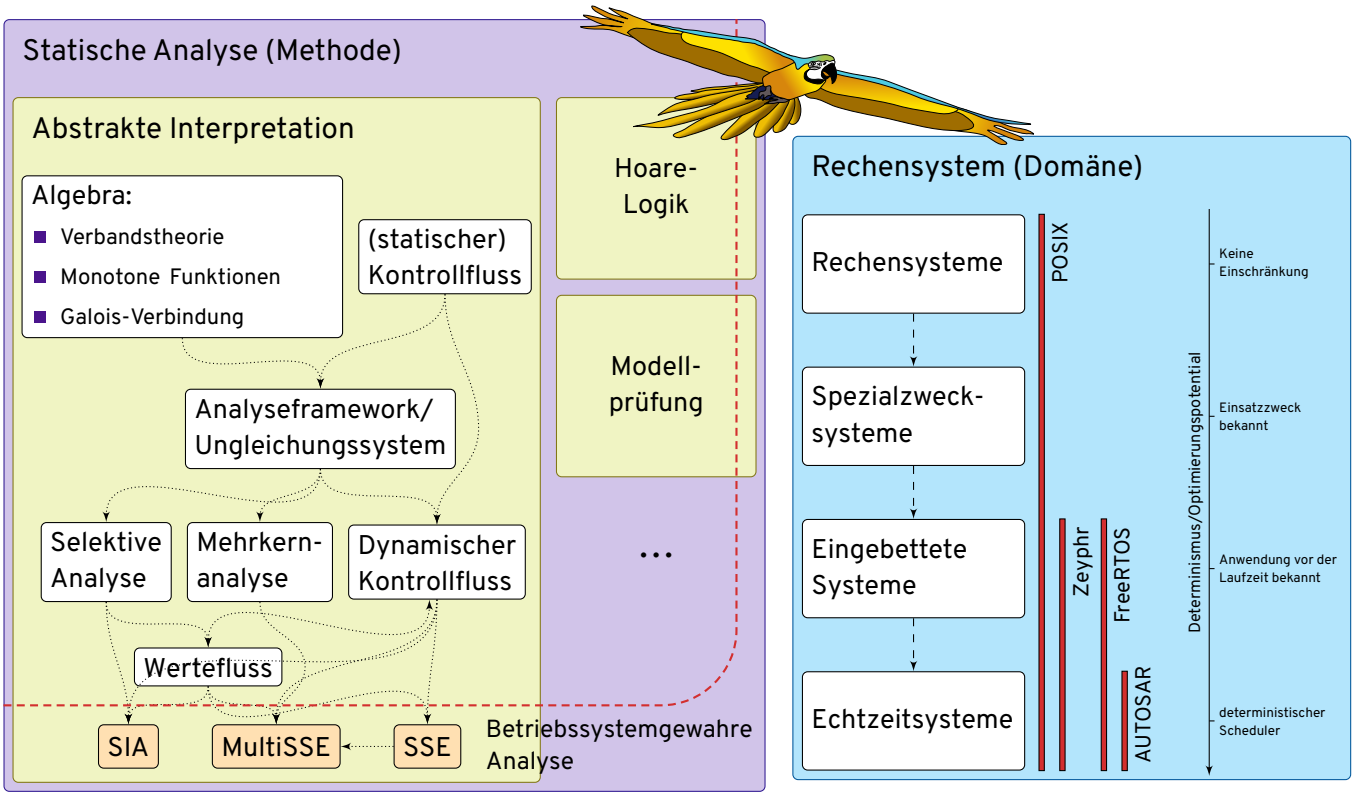



**Abbildung 3** Einordnung abstrakter Interpretationen und eingebetteter Systeme. Ich arbeite mit der Methode der statischen Analyse auf der Domäne der eingebetteten Systeme. Beide Bereiche werden durch das ARA-Analyse-Framework verbunden.
Die statische Analyse gliedert sich in mehrere verschiedene Verfahren auf, von denen ich mit der abstrakten Interpretation gearbeitet habe [CC77]. Beachtet man die Domäne als Eigenschaft der statischen Analyse, erhält man das (querschneidende) Teilgebiet der betriebssystemgewahren Analyse, bei der die schon existierende konkrete Analyse SSE [DL17] und die mit dieser Arbeit neu entwickelten Analysen SIA [›FED+21] und MultiSSE [›EFL23] in den Bereich der abstrakten Interpretation fallen. Innerhalb der abstrakten Interpretation gibt es umfangreiche (theoretische und praktische) Vorarbeit, die sich in verschiedenen aufeinander aufbauenden Analyseverfahren manifestieren, bei denen die betriebssystemgewahren Analysen letztlich an der Spitze stehen. In der Domäne der Rechensysteme kann eine Hierarchie gefunden werden, bei der die Echtzeitsysteme die am meisten beschränkte, aber dadurch am besten optimierbare Variante darstellen. Für Rechensysteme existieren eine Vielzahl von Betriebssystemen bzw. Betriebssystemschnittstellen, die jeweils einer entsprechenden Untermenge dieser Varianten zugeordnet werden können. In dieser Arbeit werde ich mich auf vier dieser Betriebssystemschnittstellen konzentrieren, von denen POSIX die allgemeinste und AUTOSAR die spezialisierteste Variante darstellt.

# 2

# Eingebettete und Echtzeitsysteme

## Konzepte und ihre Ausprägung

*Die Latenz in definierten Grenzen zu sichern, kennzeichnet die Echtzeitfähigkeit eines Systems. Es wäre also besser von einem „Rechtzeitsystem“ zu sprechen.*

*Dr.-Ing. Claus Kühnel [Küh17]*

Die in dieser Arbeit vorgestellten statischen Analysen sind für eingebettete Systeme ausgelegt. Eine Unterkategorie dieser Systeme sind Echtzeitsysteme. Zur Anwendungsentwicklung wird für beide oft ein Echtzeitbetriebssystem verwendet.

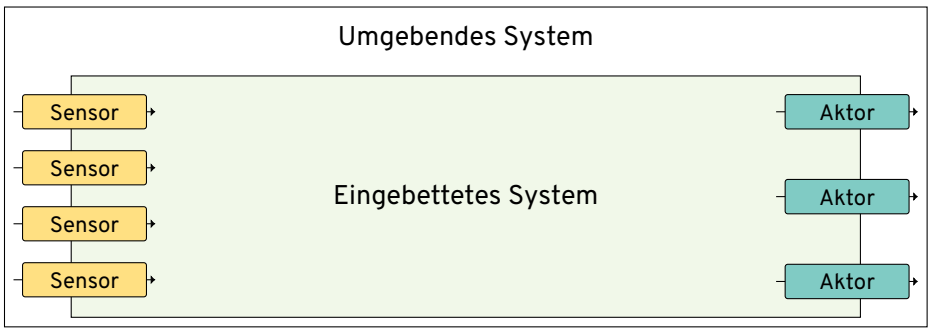



**Abbildung 2.1** Ein eingebettetes System (Schema). Das eingebettete System ist Teil eines es umgebenden Systems und mit diesem über Sensoren und Aktoren verbunden (nach [Die19A2.1]).

## 2.1 Eingebettete Systeme

Bei Rechnersystemen denken wir oft an *Personal Computers (PCs)*, Laptops, Handys und ggf. auch noch Server. Wir benutzen und bedienen diese Systeme direkt. Wir führen dort Programme aus oder wir installieren neue. Wir benutzen das System, um etwas mit diesem System zu machen.

Der deutlich größere Teil an Rechensystemen wird allerdings in andere Geräte eingebettet, um diese in ihrer Funktionalität zu erweitern. Sie sind dadurch für gewöhnlich nicht direkt sichtbar (manchmal auch mit dem Begriff der „verschwindenden Computer“ bezeichnet [Mar07A1.1]) und werden indirekt und nur noch als Teil des größeren Systems benutzt. Diese Systeme heißen *Eingebettete Systeme*.

> **Definition 1: Eingebettete Systeme.**
> Eingebettete Systeme sind Systeme, die Informationen verarbeiten und in sie umgebende Produkte eingebettet sind (aus dem Englischen übersetzt) [Mar21A1.1].

Sie sind dabei aber vollwertige Rechner, gekennzeichnet durch eine Ausführungseinheit (CPU), Arbeitsspeicher (RAM) und Festspeicher (Flash).

Sinn der Rechensysteme ist oft die Steuerung der sie umgebenden Systeme. Dafür sind sie mithilfe von Sensoren und Aktoren mit diesen verknüpft. Abbildung 2.1 zeigt ein eingebettetes System schematisch[4].

*Verbreitung* Eingebettete Systeme finden sich in nahezu allen größeren technischen Geräten, z. B. in Automobilen, Flugzeugen, smarten Glühbirnen und Industrieanlagen. Sie bilden bei weitem den größten Anteil an Rechensystemen. So kam ein Acatech-Report[5] von 2011 zu dem

[4] Manchmal lesen eingebettete Systeme auch nur Daten, ohne zu steuern. In diesem Fall entfallen die Aktoren.
[5] Deutsche Akademie der Technikwissenschaften

Schluss, dass eingebettete Mikroprozessoren zu diesem Zeitpunkt einen Marktanteil von 98% ausmachen [Deu11].

Obwohl die Hardware eingebetteter Systeme alle Komponenten eines vollwertigen Computers besitzt, sind die Anforderungen an Hard- und Software ganz andere, da zum einen der Zweck des Systems im Vorhinein feststeht und das System zum anderen möglichst lange unverändert laufen soll.

*Anforderungen*

Vor allem folgende Anforderungen können identifiziert werden [Mar21A1.3,Kop11A1.4]:

**Zuverlässigkeit:** Eingebettete Systeme sollen sicher sein (vor Ausfällen, aber auch vor Angriffen), gut reparierbar und hoch verfügbar. Ein Ausfall eines eingebetteten Systems kann das umgebende System zu einer Fehlfunktion verleiten und damit fatale Folgen provozieren. Ein tragisches Beispiel dafür sind die beiden Flugzeugabstürze der Boing 737 Max, die mehrere Hundert Menschenleben kosteten und durch eine Fehlfunktion der Software der Flugzeugsteuerung verursacht wurden [Tra19,Flo19].

**Digitalisierung von analogen Daten:** Diese Anforderung bestimmt vor allem die Frequenz des eingebetteten Systems, also die Geschwindigkeit, in der es Daten verarbeiten muss, die damit direkt von dem es umgebenden System abhängt.

**Ressourceneffizienz:** Eingebettete Systeme sollen möglichst effizient sein. Das betrifft vor allem folgende Bereiche:

- Energieverbrauch: Viele der Systeme laufen mit Batterien oder Akkumulatoren, die umso länger halten, je energiesparender das System ist.
- Programmgröße: Der Speicher auf eingebetteten Systemen ist üblicherweise stark limitiert, sodass die Anwendung nur begrenzten Platz zur Verfügung hat.
- (Bauteil-)Kosten: Eingebettete Systeme werden oftmals in sehr großen Stückzahlen produziert. Schon kleinste Einsparungen am Hardwareverbrauch kann zu enormen Einsparungen führen.

Da bei eingebetteten Systemen der Zweck bekannt ist (manifestiert durch die bereits vor der Laufzeit bekannte Anwendung), bilden sie eine Unterkategorie der *Spezialzwecksysteme*, die wiederum Systeme sind. In dieser Arbeit sind nur eingebettete Systeme im engeren Sinn relevant.

### 2.1.1 Maßgeschneiderte Systeme durch Optimierung

Alle eben beschriebenen Designziele führen zu der grundlegenden Eigenschaft, dass eingebettete Systeme auf ihre Aufgabe zugeschnitten werden. Konkret bedeutet das, dass weder überflüssige Hardware verbaut wird noch unbenutzte Software installiert ist [Mar21A1.4].

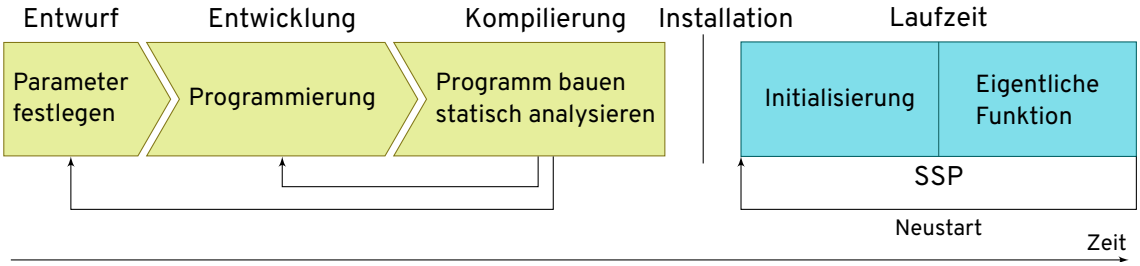



**Abbildung 2.2** Verschiedene Lebensphasen von eingebetteten Systemen. Die Phasen vor der Laufzeit sind nicht zwangsläufig streng folgend, sondern können bei Bedarf (wenn eine spätere Phase Probleme aufdeckt) wiederholt werden. Die Laufzeit gliedert sich in Initialisierung und eigentlichem Betrieb, die durch den *System Setup Point (SSP)* getrennt werden.

Überflüssige Hard- und Software senkt die Zuverlässigkeit. Eine Fehlfunktion eines unnötigen Bauteils oder einer unnötigen Anweisung stellt eine überflüssige Fehlerquelle dar. Unbenutzte Bauteile verbrauchen unnötige Energie und kosten zusätzliches Geld. Überflüssige Funktionen in Software benötigen Prozessorzyklen oder Speicher, der sonst kleiner dimensioniert werden könnte.

*Spezialisierung vs. Flexibilität*

Die Maßschneiderung wird allerdings mit einem Verlust von Flexibilität erkauft. Das System kann abschließend nur noch für den vorgesehenen Zweck benutzt werden. Um die Software trotzdem aus generischen Bausteinen aufzubauen, läuft die Maßschneiderung (teil)automatisiert ab. Das System wird erst mithilfe von generischen Bausteinen erstellt und getestet und anschließend programmunterstützt optimiert [Gre22].

Die Systemeigenschaften werden dafür in funktionale und nichtfunktionale Eigenschaften aufgeteilt. Funktionale Eigenschaften sind für das ordnungsgemäße Funktionieren des Systems zwangsläufig zu erfüllen. Nichtfunktionale Eigenschaften haben auf die Korrektheit des Systems keinen Einfluss, eine Verbesserung kann aber aus anderen Gründen wünschenswert sein [Die19A8.3,SWH10A1.1]. Der Energieverbrauch, eine oft nichtfunktionale Eigenschaft, wird z. B. bei Softwareupdates gerne verbessert [Gar22,App22].

Optimierungen sind Verbesserungen der nichtfunktionalen Eigenschaften des Systems, ohne die funktionalen Eigenschaften zu verändern [DAM08]. Voraussetzung für die Optimierung ist die im letzten Abschnitt beschriebene statische Analyse, die bei eingebetteten Systemen konkret ausnutzen kann, dass die Anwendung dieser Systeme im Vorhinein bekannt und somit eine Analyse des gesamten Systems möglich ist.

### 2.1.2 Lebensphasen

Eine Anwendung für ein eingebettetes System durchläuft verschiedene Phasen. Abbildung 2.2 visualisiert die für diese Arbeit relevanten Phasen schematisch. Begonnen wird

mit der Entwurfsphase, bei der wichtige Systemparameter festgelegt werden. Es folgt die Entwicklungsphase, bei der der konkrete Anwendungscode geschrieben wird. Danach wird das System in der Kompilierphase gebaut, d. h. ein Image für die Zielmaschine erzeugt. Hier ist der Zeitpunkt, zu dem statische Analysen zumeist stattfinden[6]. Alle in dieser Arbeit vorgestellten Analysen finden in dieser Phase statt. Mit dem fertigen Image kann das System ausgerollt („deployed“) bzw. in der Zielmaschine installiert werden. Abschließend kann die Anwendung gestartet werden. Alles in dieser Phase findet zur Laufzeit der Anwendung statt. Innerhalb dieser kann weiterhin der *System Setup Point (SSP)* definiert werden, der den Zeitpunkt angibt, an dem das System vollständig initialisiert ist [›FED+21].

*System Setup Point*

## 2.2 Echtzeitsysteme

*Echtzeitsysteme (Real-Time Systems, RTSs)* sind Systeme, bei der eine Aufgabe nur dann korrekt erfüllt ist, wenn sie zusätzlich in einer vorher festgelegten Frist („Deadline“) erfolgt ist [Liu00A2, Kop11A1.1]. Die Korrektheit des Systems hängt damit von der physikalischen Größe Zeit ab – der „echten Zeit“ –, die somit zu einer funktionalen Eigenschaft wird[7] [Die19A2.1]. In der Literatur wird weiterhin zwischen Echtzeitsystemen und *Echtzeitrechensystemen (Real-Time Computing Systems, RTCSs)* unterschieden [Die19A2.1]. Letztere beinhalten zwangsläufig einen Computer, während erstere auch rechnerlose, z. B. rein mechanische Maschinen sein können[8]. Ich werde in dieser Arbeit beide Begriffe synonym verwenden und beziehe mich immer auf RTS mit Computer. Auf dem RTCS läuft eine *Echtzeitanwendung (Real-Time Application, RTA)*, die die konkrete Kontrolllogik implementiert, dabei aber für die Aufgabenverwaltung und Synchronisation oft auf ein *Echtzeitbetriebssystem (Real-Time Operating System, RTOS)* zurückgreift.

*Begriffe: RTS, RTCS, RTA, RTOS*

RTSs steuern oftmals größere Geräte, in die sie eingebettet sind und zählen folglich zu den eingebetteten Systemen [Kop11A1.6.1]. Abbildung 2.3 erweitert Abbildung 2.1 um den internen Aufbau eines RTSs. Die Sensordaten werden hierbei in der Echtzeitanwendung verarbeitet, was in einem neuen Satz an Aktordaten resultiert.

[6] Prinzipiell können diese auch schon verschränkt zur Entwicklungsphase stattfinden. Die Analyse basiert dann aber natürlich auf unvollständigem Code.

[7] Die Aufgabe, meistens eine Reaktion auf ein Ereignis, muss dabei keinesfalls schnellstmöglich passieren, was im Multimediabereich oft mit dem Begriff „Echtzeit“ verknüpft wird, sondern zwangsläufig innerhalb der Frist.

[8] Praktisch sind diese Systeme schwer zu finden. In den allermeisten RTCSs ist die Frist nur ein Hilfsmittel, um negative Folgen auf eine andere physikalische Größe zu vermeiden (z. B. die „rechtzeitige“ Abschaltung eines Kernreaktors bei Überhitzung, oder das fristgerechte Öffnen eines Überdruckventils). Rein mechanisch sind diese Systeme oft über einen direkten Rückkopplungsmechnismus realisiert (z. B. ein Bimetall, das den Wasserkocher abschaltet, oder ein Schwimmerventil, das den Wasserzufluss bei passendem Füllstand schließt). Echte „zeitliche“ Fristen, die rein mechnisch realisiert sind, finden sich z. B. beim Aktivieren einer Eieruhr oder codiert über die Länge einer Zündschnur für Sprengstoff.

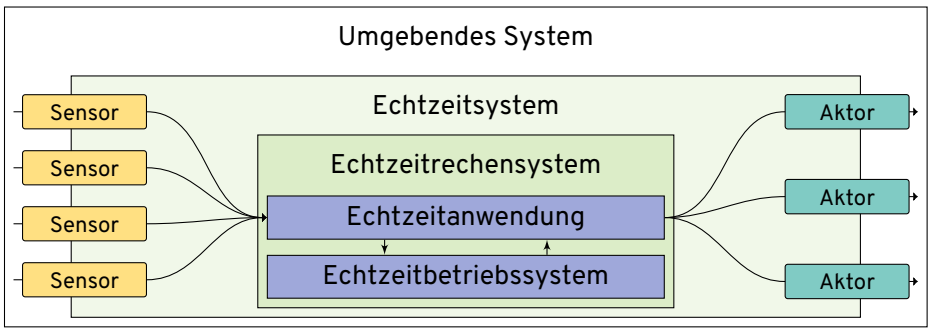



**Abbildung 2.3** Ein Echtzeitsystem. Innerhalb des Systems befindet sich ein Echtzeitrechensystem, das über Sensoren über Aktoren mit der Umgebung verbunden ist. Die Daten werden mithilfe einer Echtzeitanwendung verarbeitet, die wiederum ein Echtzeitbetriebssystem für die Jobkontrolle benutzt (nach [Die19A2.1]).

*Eigenschaften*

Um die Einhaltung der Fristen garantieren zu können, sind Echtzeitsysteme so deterministisch wie möglich angelegt. Insbesondere ist die Prozessplanung üblicherweise prioritätenbasiert und damit zu einem gewissen Grad statisch vorhersagbar. Dadurch kann idealerweise mithilfe einer Echtzeitanalyse nachgewiesen werden, dass alle Fristen unter allen Umständen eingehalten werden. Praktisch verbreitet ist aber oft eine messbasierte Validierung [Liu00A6.1].

Für Echtzeitsysteme ist die Nomenklatur nicht einheitlich. Ich werde mich in dieser Arbeit konzeptuell hauptsächlich an Liu [Liu00] und Kopetz [Kop11] orientieren und als Nomenklatur die „Burns Standard Notation“ [Dav13] verwenden. Eine Übersicht an wichtigen Begriffen findet sich in Tabelle 2.1.

Im Folgenden stelle ich ausgewählte Konzepte von Echtzeitsystemen und damit auch eingebetteten Systemen vor. Da die Konzepte von allen dort vorherrschenden Betriebssystemen implementiert werden, sind sie essentieller Bestandteil einer betriebssystemgewahren statischen Analyse.

### 2.2.1 Jobs und Tasks

*Jobs und Tasks*

Der Entwurf von Echtzeitsystemen folgt meistens einer strikten Modellierung [Liu00A3]. Die im Vorhinein bekannte Anwendung, die das Echtzeitsystem schlussendlich ausführen soll, lässt sich üblicherweise in Arbeitspakete teilen, sogenannte *Tasks*. Tasks sind die Entität, die mit einer Frist versehen werden. Sie haben eine Startzeit, wann sie frühestens ausgeführt werden können und sie haben Abhängigkeiten untereinander. Eine konkrete Instanziierung eines Tasks wird *Job* genannt [Dav13].

*Prozessoren*

Alle Jobs werden auf *Prozessoren* ausgeführt. Das können „echte“ Prozessoren im Sinne von physischen CPU-Kernen sein, das können aber auch Speicher- oder Netzwerkcontroller sein. Sie zeichnen sich dadurch aus, dass sie aktiv einen oder mehrere Jobs ausführen.

| Notation | Terminologie | Erklärung |
|---|---|---|
| $\tau_i$ | Task | Der Task mit Index $i$. |
| $\tau$ | Tasks | Menge aller Tasks. |
| $C_i$ | WCET | Eine obere Grenze der Zeit, die der Task $\tau_i$ maximal läuft. |
| $D_i$ | relative Frist | Die längstmögliche Zeit, die der Task $\tau_i$ laufen darf. |
| $P_i$ | Priorität | Die Priorität des Tasks $\tau_i$. |
| $T_i$ | Periode | Die minimale Zwischenankunftszeit zwischen Jobs des Tasks $\tau_i$. |

**Tabelle 2.1** Nomenklatur von Echtzeitsystemen (Auswahl, in der „Burns Standard Notation“ [Dav13])

Jobs interagieren mithilfe von *Ressourcen*. Sie stellen Entitäten dar, mit denen Daten ausgetauscht werden können und die in einer festgelegten Menge vorkommen. Beispielsweise lässt sich ein Sensor als Ressource modellieren, die exklusiven Zugriff bietet (sodass Daten nicht gleichzeitig gelesen werden können). Ressourcen können dabei beansprucht und wieder freigegeben werden (semantisch ähnlich zu einem Mutex oder Lock). *Ressourcen*

Die Modellierung in Jobs/Tasks, Prozessoren und Ressourcen ist entwicklerabhängig. Beispielsweise kann ein Speicher als Prozessor modelliert werden (der mit einer gewissen Latenz aktiv Daten bereitstellt oder wegschreibt) als auch als Ressource (die exklusiv angefordert werden kann).

Eine wichtige Größe beim Systementwurf ist die *schlechtmöglichste Ausführungszeit (Worst-Case Execution Time, WCET)*, die eine obere Grenze der Ausführungsdauer eines Task vorgibt. Die Bestimmung der WCET ist dabei nicht trivial und Gegenstand aktueller Forschung [Mar21A5.2]. Probleme hierbei sind vor allem fehlende Hardwaremodelle („Wie lange braucht die Hardware für diesen Befehl?“) [AGV+22] und ein nicht trivialer Kontrollfluss der Anwendung (z. B. verhindern Programmschleifen ohne obere Grenze ihrer Durchläufe jegliche WCET-Analyse). Der WCET steht die *bestmöglichste Ausführungszeit (Best-Case Execution Time, BCET)* gegenüber, die eine untere Grenze der Ausführungszeit angibt. Die nicht verbrauchte Zeit zwischen BCET und WCET heißt Schlupf („slack“). *WCET*

## 2.2.2 Fristen („Deadlines“)

Bei Fristen wird oft zwischen harten, festen und weichen Fristen unterschieden [Liu00A2, Kop11A1.5,SR94] (Abbildung 2.4). Harte Fristen sind Fristen, deren Überschreiten fatale Folgen hätte. Der Wert des Ergebnisses ist in diesem Fall negativ. Beispielsweise kann das nicht rechtzeitige Reagieren eines Hilfssystems im Automobil (wie eines Airbags) tödliche Folgen haben. Bei weichen Fristen hingegen ist ein Überschreiten nicht wünschenswert, aber

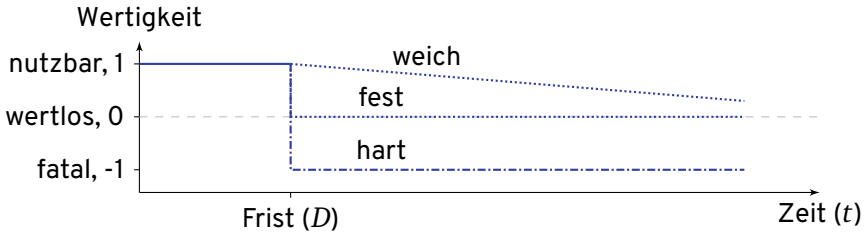



**Abbildung 2.4** Weiche, feste und harte Fristen. Das Ergebnis einer Berechnung hat je nach Typ eine andere Wertigkeit. Die abfallende Wertigkeit bei weichen Fristen ist hier nur beispielhaft eine lineare Funktion.

verkraftbar: Der Wert des Ergebnisses ist reduziert, aber noch vorhanden. Eine Verzögerung bei der Dekodierung von Videobildern z. B. führt zu einem Ruckeln des Videos oder zur Artefaktbildung, die zwar unschön sind, aber keinen Personenschaden anrichten. Wird eine feste Frist überschritten, treten keine fatalen Folgen ein, aber das Ergebnis verliert unmittelbar jeglichen Wert. Beispielsweise stellt das rechtzeitige Erreichen eines Zuges eine feste Frist dar: Wird der Zug verpasst, ist das Ziel unabhängig von der Dauer der Verspätung nicht erreicht.

Obige Definitionen ziehen die Frage nach sich, wie genau der Wert bemessen wird. Dieser muss situationsbezogen entschieden werden und ist stark abhängig vom Systemdesigner. Liu [Liu00A2.3.2] zieht z. B. die Grenze zwischen harten und weichen Fristen aus diesem Grund wie folgt: Wann immer ein Benutzer vom Entwickler verlangt, die Einhaltung einer Frist zu validieren oder zu verifizieren, handelt es sich um eine harte Frist (im Vergleich zur weichen Frist).

### 2.2.3 Ablaufplanung („Scheduling“)

Kernaufgabe eines Echtzeitsystems ist die korrekte Einhaltung der Fristen, die von mehreren Jobs mit unterschiedlichen Ausführungszeiten und Abhängigkeiten untereinander vorgegeben werden. Ein Problem entsteht dabei dadurch, dass Jobs auf Prozessoren ausgeführt werden müssen und üblicherweise weniger Prozessoren als Jobs existieren. Normalerweise wird der Prozessor dazu über die Zeit gemultiplext, d. h. verschiedenen Jobs exklusiv zugeteilt. Im einfachsten Fall ist dies eine bloße Hintereinanderausführung. Soll nun ein Job mit einer langen Frist, aber kurzen Laufzeit starten, kann bei guter Planung problemlos ein weiterer kurz laufender Job „zwischengeschoben“ werden, ohne Fristen zu verletzen. Diese Art Planung wird vom *Ablaufplaner (Scheduler)* erledigt. Können Jobs unterbrochen und verdrängt werden, heißt dies *präemptive Ablaufplanung*[9]. Die eigentliche Prozessorzuteilung

[9] Präemption wurde bereits 1962 verwendet, um *Timesharing* zu implementieren, bei dem Jobs periodisch den Prozessor erhalten, um gleichzeitige Ausführung zu simulieren [CMD62].

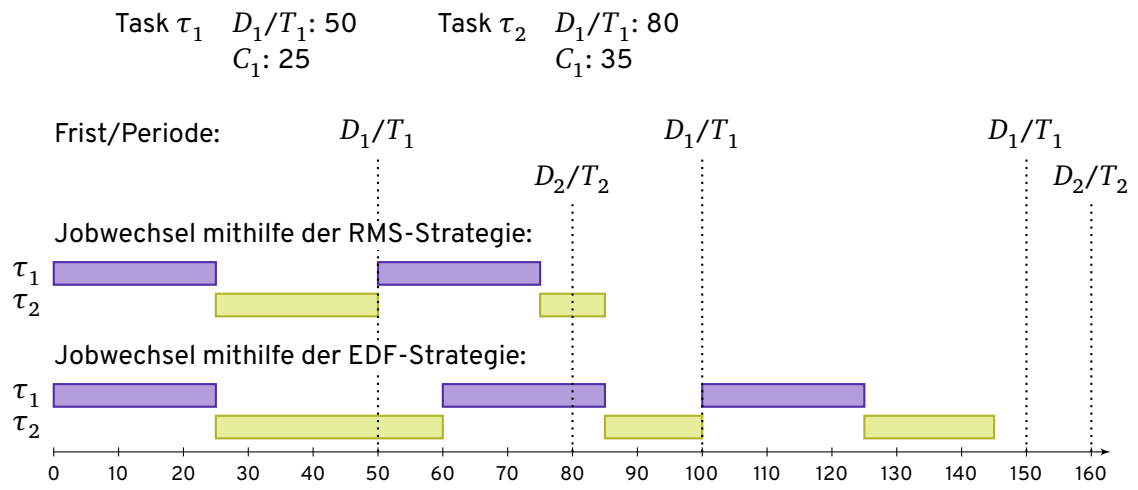



**Abbildung 2.5** Ein Set an Tasks, das einmal mit der RMS-Strategie und einmal mit der EDT-Strategie geplant wird. RMS findet in diesem Fall im Gegensatz zu EDF keine gültige Strategie. Die Auslastung des Systems ist mit $\frac{25}{50} + \frac{35}{80} = 0.94$ über den etwa 69%, die RMS fehlerlos planen kann (Beispiel nach [SGG05A19.5])

läuft prioritätenbasiert [SGG05A19.4]. Für die technische Umsetzung sind zwei Komponenten zuständig: Der Ablaufplaner trifft nach einer *Scheduling-Strategie* die Entscheidung, welcher Job laufen soll. Die eigentliche Durchsetzung der Entscheidung übernimmt der *Dispatcher* [Die19A2.1.2].

*ereignisgetriebene und zeitgetriebene RTS*

RTSs unterscheiden sich hier in zwei grundlegende Konzepte: Ereignisgetriebene (event-driven) und zeitgetriebene (time-driven) RTS [Liu00A4.4,Kop11A1.5.5]. Erstere reagieren dynamisch auf Ereignisse (die für gewöhnlich über Interrupts signalisiert werden). Letztere arbeiten nach einem festen Zeitplan periodisch alle Ereignisse ab. Zeitgetriebene Systeme haben den Vorteil, dass sämtliches Scheduling offline, d. h. *vor* der Laufzeit erledigt werden kann. Ereignisgetriebene Systeme sind dafür insbesondere bei Ereignissen, die in unterschiedlicher Frequenz auftreten, ressourceneffizienter, da sie keine feste Periode setzen müssen, die entweder die schnellen Frequenzen nicht abdeckt oder sehr klein ist, aber dafür oft überflüssige Systemlast erzeugt. In dieser Arbeit werden wir uns ausschließlich mit ereignisgetriebenen Echtzeitsystemen beschäftigen.

Die Ablaufplanung teilt die (begrenzte) Ressource „Prozessorzeit“ verschiedenen konkurrierenden Jobs zu. Es dient aber überdies zur Abhängigkeitsmodellierung der Jobs untereinander. Hängt ein Job von einem anderen ab, kann dies im System so modelliert werden, dass der letztere den ersten (mithilfe des Planers) aktiviert und damit lauffähig macht. Diese Abhängigkeiten sind bei ereignisgetriebenen Systemen explizit durch Systemaufrufe, bei zeitgetriebenen Systemen aber nur implizit modelliert, da sie im vorher berechneten Plan „einfach“ hintereinander eingeplant werden (ein wesentlicher Mechanismus in der Analyse des RTSC, siehe Abschnitt 4.4).

Bei ereignisgetriebener Planung gibt es mehrere Strategien, von denen *Rate Monotonic Scheduling (RMS)* und *Earliest Deadline First (EDF)* am populärsten sind [SGG05A19.5]. Beide arbeiten prioritätenbasiert. Jedem Task wird dafür eine solche zugeordnet. Die Ablaufplanung folgt dann der Regel, dass der lauffähige Task mit der höchsten Priorität den Prozessor erhält.

*RMS* RMS arbeitet nach dem Prinzip, dass die Prioritätenzuordnung statisch erfolgt: Je kürzer die Periode des Tasks ist, desto höher ist die Priorität. Dieses Scheduling-Verfahren ist einfach (und effizient) zu implementieren, aber nicht optimal. Es garantiert nicht, dass ein Ausführungsplan gefunden wird, wenn einer existiert. Konkret garantiert RMS, dass ein möglicher Ausführungsplan gefunden wird, wenn die Systemauslastung unter $\ln 2$ (etwa 70%) liegt [Liu00A6.7].

*EDF* EDF ordnet die Prioritäten dynamisch zu: Der Task mit der kürzesten relativen Deadline bekommt die momentan höchste Priorität. Dieses Verfahren ist unter der Voraussetzung, dass jeder Task unterbrechbar ist, und ausschließlich für Einkernsysteme optimal[10] [Liu00A4.6]. Wird EDF auf einem System implementiert, das initial mit statischen Prioritäten geplant hat, müssen bei jedem Kontextwechsel die Prioritäten neu berechnet werden. EDF braucht außerdem einen Zähler für absolute Zeit (auf unter 32-Bit-Systemen praktisch kompliziert) und trifft keine vorhersagbare Aussage darüber, was bei Überlastsituationen passiert, da im Gegensatz zu RMS nicht immer derselbe Task bevorzugt wird [But05,BG06]. Aus diesen Gründen wird in der Praxis oft RMS bevorzugt. In Abbildung 2.5 werden beispielhaft die Ergebnisse beider Algorithmen auf einem Set an Tasks dargestellt. RMS findet in diesem Beispiel im Gegensatz zu EDF keinen funktionierenden Ablaufplan.

In dieser Arbeit spielt die Scheduling-Strategie technisch eine Rolle, da sie vom Betriebssystemmodell interpretiert werden muss. Gerade Scheduling-Algorithmen für Echtzeitsysteme erleichtern aber die statische Analyse derselben, da sie wohldefiniert und deterministisch sind und daher die Scheduling-Entscheidung oftmals vorausberechnet werden kann.

## 2.3 Echtzeitbetriebssysteme

*Echtzeitbetriebssysteme (Real-Time Operating Systems, RTOSs)* sind Programme, die unterstützenden Code für die eben beschriebenen Konzepte implementieren. Der Scheduler und der zugehörige Kontextwechselmechanismus sind z. B. Kernkomponenten eines RTOSs [SGG05A19.4]. Die betriebssystemgewahren Analysen in dieser Arbeit setzen genau an der Schnittstelle zwischen Anwendung und RTOS an, den Systemaufrufen, und operieren auf einem Betriebssystemmodell, das die Betriebssystemkonzepte abstrakt repräsentiert.

[10] Dies gilt genaugenommen nur in der Theorie, da auch der Kontextwechsel selbst Zeit in Anspruch nimmt [SGG05A19.5.2].

*Aktivitäten: Fäden und ISRs*

Das zentralste Konzept eines RTOSs ist das der *Aktivität*: Diese beschreibt zusammenhängende Code-Teile, die vom RTOS bzw. der Hardware selbst zeitlich auf dem Prozessor multiplext werden und untergliedert sich in *Fäden (Threads)* und *Interrupt Service Routines (ISRs)* [Die19A2.1.2]. Multiplexing von beschränkten Ressourcen, von denen der Prozessor die wichtigste darstellt, ist eine der Hauptaufgaben von Betriebssystemen. Um den Prozessor zeitlich zu multiplexen, verwendet es Fäden, die sich gegenseitig abhängig von der Scheduling-Strategie unterbrechen können. Diese sind für die ausgeführte Applikation gleichbedeutend mit einem virtuellen Prozessor, insbesondere ist es für einen Faden nicht ohne weiteres feststellbar, dass er unterbrochen wurde. Aus diesem Grund müssen Betriebssysteme bei Fäden (äquivalent zu einem echten Prozessor) einen Program-Counter mitführen und in den meisten Fällen auch noch zusätzliche Register, die zum Kontext des Prozessors gehören[11]. In RTOSs werden Jobs in vielen Fällen auf Fäden abgebildet [Liu00A12.1]. Es ist aber auch möglich, mehrere Jobs innerhalb eines Fadens auszuführen. Konkret werde ich darauf in Abschnitt 2.4 eingehen.

Neben Fäden kann der Prozessor auch noch ISRs ausführen. Im Gegensatz zu Fäden, die durch Software ausgelöst werden, werden ISRs durch die Hardware, konkret durch eine *Unterbrechung (Interrupt)*, ausgelöst, können Fäden unterbrechen, werden aber (für gewöhnlich) nicht von diesen unterbrochen. Liegt eine Unterbrechung an (*Interrupt Request (IRQ)*), springt der Prozessor zu einer Interrupt-spezifischen Behandlungsroutine. In den im Folgenden vorgestellten RTOSs sind ISRs zusätzlich zu Fäden explizit modelliert, in der Literatur wurde aber gezeigt, dass Fäden bzw. sämtliche Kontrollflüsse als ISRs modelliert werden können [Hof14,RSK03] und daher auch hier nach Dietrich gemeinsam unter dem Begriff „Aktiviät“ fungieren [Die19A2.1.2].

*Instanzen*

RTOSs definieren außerhalb von Aktivitäten zumeist noch weitere Konzepte. Üblicherweise sind das Konzepte zur Synchronisation zwischen Fäden (z. B. Semaphoren, Mutexe, Ressourcen), der Signalisierung (z. B. Events) oder der Kommunikation inkl. Nutzdaten (z. B. Messagequeues, Queues). Manche Systeme definieren weiterhin Abstraktionen für die Hardware oder Netzwerke (z. B. Sockets). Für die später in dieser Arbeit beschriebenen Analysen ist in diesem Zusammenhang der Begriff der *Instanz* (eines virtuellen Betriebsmittels) von entscheidender Bedeutung. Ich will ihn daher hier definieren:

**Definition 2: (Betriebsmittel-)Instanz.**
Eine Instanz ist eine konkrete Ausprägung eines (virtuellen) Betriebsmittels im laufenden System. Virtuelle Betriebsmittel wiederum sind Abstraktionen des Betriebssystems, die den vorgesehenen Betrieb ermöglichen[12]. Der Begriff „Betriebssystemobjekt“ wird oft synonym mit einer Instanz verwendet.

[11] Auch in dem RTOS-Modell, das ich für diese Arbeit verwende, finden diese Felder Verwendung

[12] Genaugenommen ist der Begriff „Instanz“ eine Fehlübersetzung des englischen „instance“, die sich aber in der Informatik etabliert hat und die ich daher hier übernehme. Besser wäre „Exemplar“.

Beispiele für Instanzen sind Instanziierungen von virtuellen Betriebsmitteln wie Fäden, Semaphoren, Messagequeues usw., also z. B. der konkrete „Motor-Control-Thread“ oder der „Modem-Mutex“. Instanzen sind damit die zentralen Bausteine einer RTOS-Schnittstelle, da sie die Objekte darstellen, mit denen die Anwendung moderiert über das RTOS Betriebsmittel benutzen kann.

Über die Zeit sind viele RTOS-Implementierungen entstanden. Diese Arbeit wird sich vor allem auf vier Implementierungen konzentrieren, die so gewählt sind, dass sie zusammen ein breites Spektrum an Konzepten abdecken. Vor allem fällt die verschiedenartige Instanziierung auf. Die Instanzen können dabei entweder statisch in der Kompilierphase, oder dynamisch, d. h. zur Laufzeit, angelegt werden. Tabelle 2.2 fasst die konzeptuellen Unterschiede zusammen. Eine Tabelle konkreter Systemaufrufe findet sich in Anhang A.1. Ich werde die einzelnen RTOSs im Folgenden kurz vorstellen.

| **Konzept** | **AUTOSAR** | **FreeRTOS** | **Zephyr** | **POSIX** |
|---|---|---|---|---|
| Art | Standard | Implementierung | Implementierung | Standard |
| Einsatzzweck | RTS | RTS/<br>Eingebettete Systeme | RTS/<br>Eingebettete Systeme | Universell |
| Instanziierung | Statisch | Dynamisch | Statisch/Dynamisch | Dynamisch |
| Konfiguration | DSL (OIL) | C-Makros | DSL (KConfig) | kA |
| Scheduler | RMS | RMS | konfigurierbar | konfigurierbar |
| Virtuelle Betriebsmittel (u. a.) | Tasks<br>Spinlocks<br>Ressourcen<br>Events | Tasks<br>Semaphoren<br>Notifications | Threads<br>Semaphoren<br>Mutexe<br>Bedingungsvariablen | Threads<br>Mutexe<br>Signale |

**Tabelle 2.2** Konzeptueller Vergleich der verschiedenen RTOSs. Die Art beschreibt, ob es sich um einen Standard (ggf. mit mehreren Implementierungen) oder eine Implementierung (ohne Standard) handelt. Einige RTOSs werden über eine *domänenspezifische Sprache (Domain Specific Language, DSL)* konfiguriert.

### 2.3.1 AUTOSAR

AUTOSAR (**AUT**omotive **O**pen **S**ystem **AR**chitecture) ist ein Zusammenschluss großer Automobilhersteller oder -zulieferer, mit dem Ziel, eine gemeinsame Softwareplattform für den Einsatz in Automobilen zu schaffen. Er wurde 2002 gegründet [AUT24].

Ein Bestandteil der Gesamtplattform ist die „AUTOSAR Classic Platform“, die Standards für „Embedded Control Units“ definiert. Innerhalb dieser Standards ist für diese Arbeit AUTOSAR OS relevant, das einen Standard für ein RTOS darstellt [AUT20]. AUTOSAR

OS basiert auf dem älteren OSEK-OS[13] [OSE05], erweitert dieses aber hauptsächlich um Mehrkernunterstützung. Wenn ich in dieser Arbeit von AUTOSAR spreche, dann meine ich damit immer den Betriebssystemstandard.

Mehrkernunterstützung in AUTOSAR ist über einen partitionierten Scheduler umgesetzt: Der Scheduling-Algorithmus läuft auf jedem Kern separat. Die meisten Instanzen müssen dazu fest einem Kern zugeordnet werden (in AUTOSAR heißen diese dann „Locatable Entities“). Insbesondere sind Tasks fest einem Kern zugeordnet und wechseln niemals. Systemaufrufe betreffen im Standardfall nur den eigenen Kern. Einige Systemaufrufe haben einen fest definierten Effekt auf einen oder mehrere spezifizierte andere Kerne.

*Instanziierung und Konfiguration*

AUTOSAR arbeitet ausschließlich mit statischer Instanziierung. Instanzen müssen bereits in der Entwicklungsphase definiert werden, sodass die Instanziierung in der Kompilierphase geschieht. AUTOSAR definiert hierfür eine *domänenspezifische Sprache (Domain Specific Language, DSL)*, die *OSEK Implementation Language (OIL)*, in der alle wichtigen Instanzparameter festgelegt werden können. Der Standard hat ein natives Verständnis von *Tasks*, die der Scheduler direkt verwaltet. Tasks können sich gegenseitig aktivieren oder verketten. Dies ist auch über Kerngrenzen hinweg möglich[14].

*PCP*

AUTOSAR plant auf fest zugewiesenen Prioritäten. Um Prioritätsinversion in kritischen Bereichen und Deadlocks zu vermeiden, implementiert es dabei das *Priority Ceiling Protocol (PCP)*, das Tasks, die einen kritischen Bereich betreten, temporär eine ggf. höhere dynamische Priorität zuordnet [Liu00A8.6,AUT20]. Dieser entspricht der höchsten Priorität aller Tasks, die potentiell im kritischen Bereich konkurrieren („ceiling priority“). Der Standard spezifiziert dazu *Ressourcen*, die ein Task zu Beginn eines kritischen Bereichs anfordert und anschließend wieder freigibt. Die Berechnung der passenden Priorität kann dabei nur statisch erfolgen, somit ist auch die Ressourcenzuordnung zu Tasks zwingend statisch. Ressourcen synchronisieren ausschließlich Tasks auf dem gleichen Kern.

*Mehrkern: Sperren*

Für die Synchronisation zwischen Kernen definiert AUTOSAR *Spinlocks*. Diese sind allerdings nicht Deadlock-frei und Tasks können verhungern. Der Standard empfiehlt daher, Sperren nicht zu verschachteln. Prinzipiell wurde bereits 1990 eine Erweiterung des PCP für mehrere Kerne entworfen [Raj90]. Dieser Mechanismus ist allerdings nicht durch den Standard berücksichtigt [LBR11]. Weiterhin haben Wieder und Brandenburg eine umfassende Analyse verschiedener Locktypen in AUTOSAR vorgenommen, in der sie zu dem Schluss kommen, dass der Locking-Mechanismus, so wie er aktuell ist, die Planbarkeit

[13] OSEK steht kurz für „**O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug“ und war ein Vorgängerkonsortium aus Automobilunternehmen.

[14] AUTOSAR unterscheidet hier weiterhin zwischen „Basic Tasks“ und „Extended Tasks“, wobei erstere nicht warten dürfen und somit von einer höherprioren Aktivität unterbrochen werden können. Der technische Vorteil davon ist, dass sich alle Basic Tasks einen Stack teilen können. Für die Analysen in dieser Arbeit spielt die Unterscheidung allerdings keine Rolle.

deutlich verringert und eine feinere Spezifikation, insbesondere mit geordneten Spinlocks, sinnvoll wäre [WB13].

Zur Signalisierung definiert AUTOSAR weiterhin *Events*. Tasks können auf Events warten oder auch welche senden. Diese werden in einer Maske gespeichert und sind über Kerngrenzen hinweg verfügbar.

### 2.3.2 POSIX

POSIX (**P**ortable **O**perating **S**ystem **I**nterface) ist eine Familie von Standards [Var22]. Diese wurde zuerst als Versuchsversion 1986 veröffentlicht [Isa90], hat seitdem mehrere Neuauflagen erhalten, zuletzt 2018, und wird von der Austin-Group entwickelt [ISO18].

Der Sinn von POSIX ist es, Schnittstellen zu definieren, die benutzt werden können, um portable Anwendungen zu schaffen. Der Standard definiert dazu eine Schnittstelle zum Betriebssystem (auf Basis von UNIX) inkl. Hilfsfunktionen der C-Standardbibliothek, eine Shell-Laufzeitumgebung und Hilfsprogramme.

Für die Systemschnittstelle sind etwa 1200 Funktionen definiert, die Konzepte wie Threads, Dateien und Pipes, Signale und Sockets festlegen. Sie beinhaltet aber auch Funktionen wie `strcpy()`, also eine Klasse an Hilfsfunktionen für gängige Programmierprobleme. POSIX beinhaltet zudem eine Echtzeit-Erweiterung, die striktere Vorgaben an Scheduling und Speicherverwaltung stellt. Die POSIX-Systemschnittstelle wird von der libc implementiert, die dann wiederum für einige Funktionen auf Systemaufrufe zurückgreift, die Teil der jeweiligen Betriebssystemschnittstelle sind [The23,Fre23].

In dieser Arbeit beschäftige ich mich ebenfalls mit POSIX, werde dort aber nur ein Subset beachten, der für eingebettete Systeme eine Rolle spielt, insbesondere werde ich ausschließlich Systemaufrufe nach Definition 4 beachten.

Im Gegensatz zu AUTOSAR als Beispiel für eine Schnittstelle mit statischen Instanzen und sehr genauer Spezifikation aller Mechanismen ist POSIX ein Beispiel für eine Schnittstelle, die ausschließlich dynamische Instanzen beinhaltet und auch sonst versucht, ein Maximum an Einsatzzwecken zu erfüllen, dadurch aber oftmals unspezifischer ist oder mehrere Möglichkeiten anbietet.

### 2.3.3 FreeRTOS

FreeRTOS ist ein Open-Source-RTOS, dessen Entwicklung 2003 gestartet wurde und inzwischen maßgeblich von Amazon vorangetrieben und betreut wird [Ric20,Git22].

Es ist wie POSIX ein vollständig dynamisches System, stellt aber insgesamt (vergleichbar mit AUTOSAR) weniger Funktionen zur Verfügung. Sämtliche Instanzen müssen zur

Laufzeit erstellt werden[15]. *Tasks* bei FreeRTOS sind als Fäden im klassischen Sinne implementiert [Ama23]. Insbesondere werden sie zu einem beliebigen Zeitpunkt erstellt, dabei automatisch lauffähig und dann einmalig gestartet. Um dennoch periodisch arbeiten zu können, implementieren sie oftmals eine Endlosschleife.

Andere Betriebssystemobjekte in FreeRTOS sind *Queues* zum synchronisierten, gepufferten Austausch von Daten und *Semaphoren*, *Mutexe* und *Streambuffer*, die synchronisierten ungepufferten Datenaustausch zwischen Threads ermöglichen. Semaphoren und Mutexe sind bei FreeRTOS intern vollständig auf die bereits synchronisierte Queue-Implementierung abgebildet.

In einer FreeRTOS-Applikation werden oft zu Beginn der Laufzeit in der `main`-Funktion alle Instanzen erstellt, dann der Scheduler gestartet und anschließend die eigentliche Aufgabe des Systems verrichtet. Der explizite Startzeitpunkt des Schedulers kann als SSP angesehen werden (wenn kein Task Initialisierungsaufgaben verrichtet). Ab diesem Zeitpunkt ist das System bereit, seine eigentliche Funktion wahrzunehmen.

### 2.3.4 Zephyr

Zephyr ist wie FreeRTOS eine Open-Source RTOS-Implementierung, die allerdings unter der Schirmherrschaft der Linux-Foundation entwickelt wird. Es ist eine Weiterentwicklung des Virtuoso DSP Betriebssystems und wird seit 2016 als Community-Projekt entwickelt [Pro22]. Im Gegensatz zu AUTOSAR und FreeRTOS ist Zephyr ein deutlich umfangreicheres RTOS. Es enthält nicht nur den eigentlichen Systemkern, der Thread-Multiplexing und Synchronisation bereitstellt, sondern auch fertige Dateisystemtreiber, einen Netzwerkstack und Gerätetreiber.

Die meisten Funktionen von Zephyr sind mithilfe eines KConfig-basierten[16] Systems an- und abwählbar, wodurch der (kompilierte) Kern sehr schlank gehalten werden kann. Zephyr verbindet durch seine Schnittstelle ein wenig die Eigenheiten von FreeRTOS und AUTOSAR. So unterstützt es sowohl die statische als auch die dynamische Instanziierung von Betriebssystemobjekten.

Als Aktivitäten implementiert Zephyr Fäden und kann diese mit mehreren auswählbaren Algorithmen planen (u. a. RMS), die alle prioritätenbasiert arbeiten [Pro23b]. Als weitere Kernkonzepte implementiert Zephyr zur Synchronisation Mutexe, Semaphoren und Condition-Variables (vergleichbar mit AUTOSAR-Events). Zum Datenaustausch gibt es überdies noch Pipes, Queues (in verschiedenen Varianten), Stacks, und einige mehr. Zusätzlich

[15] FreeRTOS hat Systemaufrufe mit dem Suffix „Static“ wie `xTaskCreateStatic`, die suggerieren, dass eine statische Instanz erstellt werden soll. Diese Aufrufe erstellen bis auf die Verwendung statischen Speichers die Instanz aber weiterhin dynamisch.

[16] KConfig ist das Konfigurationssystem des Linux-Kerns.

zu den Kernkonzepten gibt es dann noch definierte Schnittstellen für Netzwerk, Bluetooth, USB, Dateisysteme und weitere.

Zephyr hat Unterstützung für Mehrkernbetrieb [Pro23a]. Dabei verwaltet das Betriebssystem alle Kerne gleichberechtigt. Fäden können zwischen Kernen wechseln (wenn sie nicht explizit auf einen Kern beschränkt sind). Zephyr unterstützt zur Synchronisation zwischen mehreren Kernen und Interrupts ein globales Lock, empfiehlt aber die Verwendung von feingranularen Spinlocks.

### 2.3.5 Die Systemaufruf-Schnittstelle

Ich habe in den letzten Abschnitten vier verschiedene RTOS-Schnittstellen beschrieben, die erst einmal verschieden klingende Konzepte bereitstellen, deren Implementierungen und Instanziierungsmechanismen sich unterscheiden. Diese Arbeit wird sich damit beschäftigen, alle diese Systeme mit den gleichen Methoden zu analysieren und muss dazu ihre Gemeinsamkeiten und Unterschiede herausstellen.

Wir können dazu an dieser Stelle bereits feststellen:

- Alle beschriebenen Echtzeitbetriebssysteme werden von den Applikationen über definierte Funktionsaufrufe verwendet, den Systemaufrufen.
- Alle beschriebenen Systeme haben ein Konzept von Instanzen mit dem Spezialfall der Aktivitäten, die aktiv weitere Aktionen auslösen.

*Systemaufrufe*

Meine Arbeit beschäftigt sich mit der Interaktion zwischen Instanzen und deren Auswirkung auf das Systemverhalten. Manche Schnittstellen, insbesondere POSIX und teilweise auch Zephyr, definieren sehr viele Funktionen, von denen einige keinen Einfluss auf die Instanzinteraktion haben und eher als Hilfsfunktionen gelten sollten (wie das schon erwähnte `strcpy()`). Alle diese Funktionen als Teil der Betriebssystemschnittstelle aufzufassen, bietet darum keinen Mehrwert für meine Analysen. Ich will hier eine Definition für einen Systemaufruf geben, die eine Trennung der Schnittstelle in betriebssystemrelevante und betriebssystemirrelevante Funktionen erlaubt. Wir brauchen dazu den Begriff des *betriebssystemrelevanten Systemzustands*. Dieser beinhaltet alle für das jeweilige RTOS relevanten Daten und ist für dieses spezifisch. In allen hier untersuchten RTOSs gehört beispielsweise die Priorität eines Fadens zu diesem Zustand, während der Inhalt seines Benutzerstapels nicht dazugehört. Anschließend können wir eine Systemfunktion definieren:

**Definition 3: Systemfunktion.**
Eine Systemfunktion ist eine Funktion, innerhalb derer eine Instanz erzeugt oder der betriebssystemrelevante Systemzustand einer Instanz berührt wird (gelesen wird oder sich ändert).

*Korollar*: Nur in einer Systemfunktion sind Instanzänderungen möglich.

**Definition 4: Systemaufruf.**
Ein Systemaufruf ist der Aufruf einer Systemfunktion.

Leider wird oftmals nicht zwischen Funktion und konkretem Aufruf unterschieden. In den Fällen, in denen eine Unterscheidung notwendig ist, werde ich diese daher gesondert kenntlich machen.

Es gibt Systemfunktionen, die den Instanzzustand nur lesen (beispielsweise `getpid()`). In meinen Analysen sind allerdings ausschließlich Änderungen des Systemzustands relevant, weswegen diese derartige Systemfunktionen aus Performanzgründen übergehen.

## 2.4 Vom Modell zur Anwendung

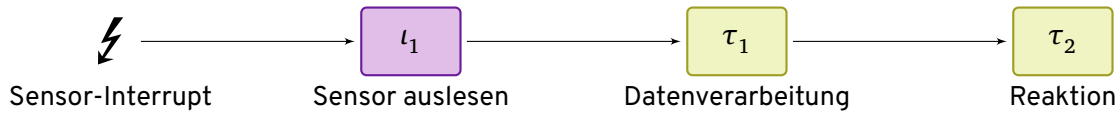
Speziell für Echtzeitsysteme gelten sehr strikte funktionale Anforderungen. Diese werden für gewöhnlich vor Entwurf und Implementierung festgelegt. Anschließend gehen sie über eine Entwurfsphase in ein Modell über Tasks, Prozessoren und Ressourcen ein. Dieses wird anschließend implementiert, d. h. in Quellcode transformiert. Das RTOS stellt dabei Implementierungen gängiger Konzepte des Modells bereit. Sowohl die Zuordnung zu Konzepten innerhalb des Modells als auch die anschließende Überführung in eine Implementierung sind nicht eindeutig [Die19A2.2].

*uneindeutige Abbildung*

Abbildung 2.6 zeigt z. B. ein Task-Modell mit einer möglichen Implementierung mithilfe des AUTOSAR-Standards und zwei weiteren mithilfe der Zephyr-Schnittstelle. Es fällt auf, dass die AUTOSAR-Implementierung in diesem Fall am eindeutigsten die Vorgaben aus dem Modell abbildet, aber die beiden Tasks beispielsweise in der Zephyr-Implementierung II ohne Probleme auf nur einen Faden abgebildet werden können.

Auf dem Weg zur Implementierung muss sichergestellt werden, dass die funktionalen Anforderungen weiterhin erfüllt bleiben. Jede Stufe bringt dabei aber neue Ungenauigkeiten ins System ein. Die Implementierung der Systemaufrufe ist entscheidend (ein Austausch der RTOS-Implementierung führt zwangsläufig zu einem anderen Zeitverhalten). Auch die eingesetzte Hardware wirkt sich signifikant auf das Zeitverhalten aus. Es steht aber erst mit der konkreten Implementierung fest, wie genau die Hardware benutzt wird.

Aus diesen Gründen ist eine (messbasierte oder formale) Verifizierung des Systems erst ganz am Ende, wenn der gesamte Code vorliegt, überhaupt möglich. Genau aus diesem Grund setzen die Analysen dieser Arbeit an dieser Stelle an, um alles notwendige Wissen aus einer Anwendung zu extrahieren, das für Optimierungen nicht funktionaler Eigenschaften notwendig ist: Nur an dieser Stelle liegt dieses Wissen vor.




**a)** Eine Anwendung. Auf den Interrupt folgt die ISR $\iota_1$ sowie die Tasks $\tau_1$ und $\tau_2$.

```
ISR SENSOR:(ι1)                    .oil
  CATEGORY = 2;
  PRIORITY = 100;
  DEVICE = 37;

TASK T1:(τ1)
  PRIORITY = 2;
  SCHEDULE = FULL;

TASK T2:(τ2)
  PRIORITY = 2;
  SCHEDULE = FULL;
```

```
ISR(SENSOR) {(ι1)                  .c
  save_sensor_data();
  ActivateTask(T1); }

TASK(T1) {(τ1)
  process_sensor_data();
  ChainTask(T2); }

TASK(T2) {(τ2)
  send_reaction();
  TerminateTask(); }
```

**b)** AUTOSAR-Implementierung

```
K_SEM_DEFINE(sensor_sem, 0, 1);    .c

void isr_sensor(void) {(ι1)
  save_sensor_data();
  k_sem_give(&sensor_sem);
}
IRQ_CONNECT(37, 100, isr_sensor,
            (ι1) NULL, NULL);

K_CONDVAR_DEFINE(processed);

t1_action() {(τ1)
  process_sensor_data();
  k_condvar_signal(&processed);
}
t2_action() {(τ2)
  k_condvar_wait(&processed, NULL,
                 K_FOREVER);
  send_reaction();
}

main_t() {
  while(true) {
    k_sem_take(&sensor_sem);
    k_thread_create(t1,
                    (τ1) t1_action,
                    2);
    k_thread_create(t2,
                    (τ2) t2_action,
                    1);
  }
}
K_THREAD_DEFINE(main, main_t, 1);
```

**c)** Zephyr-Implementierung I

```
K_SEM_DEFINE(sensor_sem, 0, 1);    .c

void isr_sensor(void) {(ι1)
  save_sensor_data();
  k_sem_give(&sensor_sem);
}
IRQ_CONNECT(37, 100, isr_sensor,
            (ι1) NULL, NULL);

t1_action() {
  k_sem_take(&sensor_sem);
  process_sensor_data();(τ1)
  send_reaction();(τ2)
}
K_THREAD_DEFINE(t1, t1_action, 1);
```

**d)** Zephyr-Implementierung II

**Abbildung 2.6** Drei verschiedene Implementierungen der gleichen Applikation. Die Anwendung schreibt zwei Tasks vor, die auf die ISR folgen sollen (a)). In AUTOSAR kann man diese direkt implementieren (b)). In Zephyr braucht es dazu Threads. In c) wird die Applikation auf drei Threads abgebildet: einen Verwaltungsthread (*main*), der zur eigentlichen Behandlung dynamisch zwei weitere Threads erzeugt. In d) werden die zwei Tasks auf nur einen Thread abgebildet. Bei beiden Zephyr-Implementierungen kommunizieren die Threads explizit.

## 2.5 Die Analyse von eingebetteten Systemen

Harte Echtzeitsysteme sind oftmals so deterministisch wie möglich angelegt: Schleifen innerhalb des Quellcodes sind zwangsläufig begrenzt, Rekursion kommt nicht vor, Fäden werden (bis auf eine Initialisierungsphase) nicht dynamisch erzeugt und haben verschiedene Prioritäten, der Scheduler plant streng prioritätenbasiert. All dies macht die Reaktion eines RTS deterministisch und vereinfacht bzw. ermöglicht überhaupt erst eine statische Echtzeitanalyse, die die Einhaltung von Fristen (unter gewissen Bedingungen) garantiert [KKZ12, DFK+21]. AUTOSAR ist hierbei das einzige RTOS, das einen Großteil dieses Determinismus im Entwurf erzwingt. Zephyr ermöglicht ihn, wohingegen FreeRTOS und POSIX ihn durch die steigende Dynamik sogar eher erschweren. POSIX im Speziellen ist dabei nicht ausschließlich auf RTS ausgelegt, sondern beinhaltet diese als Erweiterung. Abbildung 3 stellt diese Unterschiede schematisch dar.

Ein RTOS muss aber nicht zwangsläufig für harte RTSs verwendet werden. Es kann auch (bedingt durch den kleinen System-Overhead) als Basis für ein weiches RTS oder ein allgemeines eingebettetes System dienen. So sind die meisten der in dieser Arbeit analysierten Anwendungen keine harten Echtzeitsysteme: Einige sind weiche Echtzeitsysteme. Bei anderen spielen Fristen keine oder nur eine sehr geringe Rolle. Entsprechend folgt der Entwurf dieser Systeme auch nicht dem klassischen Muster für RTS, so sind die Fadenprioritäten beispielsweise eher willkürlich gesetzt. Die Prioritäten in der Entwicklung liegen eher woanders: Eine komfortable Schnittstelle für weiteren Code bereitstellen, eine saubere Trennung zwischen verschiedenen Aufgaben schaffen. Das RTOS wird für diese Systeme hauptsächlich als Hilfsbibliothek genutzt, die bereits Scheduling, Synchronisation und Kommunikation implementiert.

Für meine Arbeit sind aber auch diese Systeme ein gleichwertiges Evaluationsziel, da meine Analysen die Verwendung des RTOS zwecks Verbesserung nichtfunktionaler Eigenschaften des Gesamtsystems zum Ziel haben. Auch eingebettete Systeme mit weichen oder keinen Echtzeitanforderungen haben starke Anforderungen nach effizienter Ressourcennutzung, die durch die Verbesserung der nichtfunktionalen Eigenschaften erreicht wird.

## 2.6 Zusammenfassung

Der zweite Teil von Abbildung 3 stellt eine gute graphische Zusammenfassung dieses Kapitels dar: Es hat sich mit der Hierarchie der Rechensysteme beschäftigt und ist über Spezialzwecksysteme im Speziellen auf eingebettete Systeme eingegangen, deren Einsatzzweck bereits vor der Laufzeit bekannt ist. Eine Unterkategorie der eingebetteten Systeme stellen dabei die Echtzeitsysteme dar, die überdies die Ausführungszeit von Aufgaben zu einer funktionalen Eigenschaft erheben.

Eingebettete Systeme sind oft in eine Echtzeitanwendung und ein Echtzeitbetriebssystem aufgeteilt, wobei erstere letzteres benutzt, um Konzepte wie Fäden, Synchronisations- oder Kommunikationsprimitiven bereitgestellt zu bekommen. Ich habe konkret die RTOSs AUTOSAR, FreeRTOS, Zephyr, und POSIX vorgestellt, die durch ihren Entwurf verschieden stark auf die Domäne zugeschnitten sind: AUTOSAR stellt dabei das minimalste und statischste System dar, währen POSIX für Echtzeitanwendungen nur durch eine seiner Erweiterungen geeignet ist und vollständig dynamisch arbeitet.

Im nächsten Kapitel wird es unabhängig von der Domäne der eingebetteten Systeme um die Methode der statischen Analyse gehen, die mit dem darauffolgenden Kapitel zur betriebssystemgewahren Analyse vereint werden.

# 3

# Statische Analyse

## Die Methode der Abstrakten Interpretation

Statische Analysen analysieren ein Programm, ohne es auszuführen. Eine Möglichkeit, dies zu erreichen, ist das Berechnen der Auswirkung jeder Anweisung auf einen mitgeführten abstrakten Zustand, die Methode der abstrakten Interpretation. Dieses Kapitel stellt beide Bereiche näher vor und geht zudem auf die theoretischen Grundlagen der abstrakten Interpretation ein, die die Korrektheit des Verfahrens formal beweisbar machen.

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int BUF_SIZE = 128;

int main() {
    char* buf = malloc(BUF_SIZE);
    if (buf == NULL)
        goto error;

    FILE* fp = fopen("data.txt", "rb");
    if (fp == NULL)
        goto error;

    fread(buf, 1, BUF_SIZE-1, fp);
    buf[BUF_SIZE-1] = '\0';

    printf("%s\n", buf);
    free(buf);
    fclose(fp);
    return 0;

error:
    printf("An error occurred.\n");
    return 1;
}
```

**Codeblock 3.1** Ein einfaches Programm (für die POSIX-Schnittstelle), das Speicher alloziert und im Fehlerfall nicht wieder freigibt.

*statische vs. dynamische Analysen*

Programmanalysen werden prinzipiell in statische und dynamische Analysen unterschieden. Erstere finden vor der Laufzeit des Programms statt (typischerweise in der Kompilierphase). Letztere finden zur Laufzeit des Programms statt. Statische Analysen arbeiten mit der Interpretation des Programmcodes, *ohne* ihn auszuführen [RY20A1.3]. Dynamische Analysen beobachten das Programm während einer Ausführung.

Soll das Resultat der Analyse für den Kompilierprozess verwendet werden, bieten sich statische Analysen an. Es ist aber durchaus auch möglich, das Ergebnis von dynamischen Analysen für Optimierungen zu verwenden[17]. Die beiden Hauptvorteile statischer Analyse liegen in den folgenden Punkten:

- Eine statische Analyse kann auch Programmpfade analysieren, die in der konkreten Ausführung nicht oder nur unter bestimmten Bedingungen auftreten [Loc21A5.1.2].

[17] Klassisches „Debugging“ ist z. B. ein Art der dynamischen Analyse, deren Resultat fast immer verwendet wird, um den Originalcode zu verbessern.

Quellcode 3.1 zeigt z. B. ein Programm, das nur im Fehlerfall den zuvor allozierten Speicher nicht freigibt. Eine dynamische Analyse, bei der das Programm fehlerlos läuft, findet den Speicherfehler nicht. Eine statische Analyse hingegen wird in diesem Fall beide Pfade analysieren.

- Eine statische Analyse kann selektiv arbeiten und uninteressante Programmteile überspringen. Dadurch kann sie schneller werden als eine Programmausführung (z.B. kann eine Schleife nur einmal abstrakt durchlaufen werden).

In dieser Arbeit werde ich mich nur mit statischen Analysen beschäftigen. Dieser Bereich ist erneut in mehrere Unterbereiche unterteilbar von denen in dieser Arbeit vor allem die *abstrakte Interpretation* relevant sein wird (vgl. Abbildung 3). Für die abstrakte Interpretation gibt es umfangreiche Vorarbeit, um mathematisch die Terminierung, Korrektheit und Vollständigkeit der Analyse nachweisen zu können [CC77]. Darauf aufbauend wurde ein Ansatz für ein generisches Analyse-Framework geschaffen, das für zusätzliche Funktionen oder Anforderungen erweitert werden kann [RY20A3,SWH10A1.12].

Dieses Framework liefert die formalen Grundlagen, um die bereits existierende *System-State Enumeration (SSE)* [DHL17] und die in dieser Arbeit neu geschaffenen Analysen *statische Instanzanalyse (Static Instance Analysis, SIA)* [›FED+21] und MultiSSE [›EFL23] in einem gemeinsamen Kontext einordnen und deren Korrektheit nachzuweisen.

Ich werde darum in diesem Unterkapitel zuerst Korrektheit und Vollständigkeit definieren und anschließend das Analyse-Framework zusätzlich zu den relevanten Erweiterungen und mathematischen Grundlagen vorstellen.

# 3.1 Korrektheit und Vollständigkeit

*Abbildbarkeit auf das Halteproblem*

Grundsätzlich wird bei einer statischen Analyse das Ziel verfolgt, die Existenz oder Absenz von bestimmten Programmeigenschaften nachzuweisen. Die ideale statische Analyse wäre demnach ein Programm, das für das zu untersuchende Programm für jede Eingangskonfiguration bestimmen kann, ob die zu untersuchende Eigenschaft bei dieser Konfiguration existiert oder nicht. Diese resultiert in einer Äquivalenz der Analyseergebnisse und einer Programmausführung:

Das Programm erfüllt bei einer Konfiguration eine bestimmte Eigenschaft $\leftrightarrow$

Die Analyse weist diese Eigenschaft nach.

Mit der Annahme, dass die Analyse automatisch abläuft, ist das Problem, eine solche Analyse zu konstruieren, auf das Halteproblem reduzierbar und damit nicht allgemein lösbar [RY20A1.3,MS18A1.2][18]

Um dennoch sinnvolle Analysen zu konstruieren, muss die obige Äquivalenz oder der Automatismus aufgegeben werden [RY20A1.3]:

**Automatisierbarkeit:** Die Analyse läuft nicht mehr automatisch. An Stellen, an denen der Analysemechanismus versagt, muss der Anwender aushelfen. Ein typisches Beispiel dafür sind Schleifengrenzen (also die numerische Grenze, wie oft eine Schleife maximal durchlaufen wird). Dieses Verfahren ist neben einem größeren Aufwand des Anwenders auch fehlerträchtig, da das eingegebene Wissen Fehler beinhalten kann. Es gibt außerdem korrekte und vollständige Analysen, die allerdings modellbasiert (also auf einer reduzierten nicht mehr Turing-vollständigen beschränkten Eingabe) arbeiten. Das entsprechende Modell muss auch hier im Vorhinein manuell (oder mit einer unvollständigen oder inkorrekten Analyse) erzeugt werden.

**Korrektheit:** Ein positives Ergebnis der Analyse impliziert, dass das Programm diese Eigenschaft auch mit jeder Eingabe besitzt. Eine Analyse dieser Art heißt *korrekt („sound“)*. Gerne wird auch von „pessimistischer“ Analyse gesprochen, da die Analyse bei Entscheidungen im Zweifel verneint, um keine Fehler zu machen. Dementsprechend wäre die triviale Analyse dieser Art eine Analyse, die alles verneint.

**Vollständigkeit:** Wenn ein Programm eine Eigenschaft besitzt, impliziert dies, dass auch die Analyse zu einem positiven Ergebnis kommt. Eine Analyse dieser Art heißt *vollständig („complete“)*. Diese Analyse wird analog auch „optimistisch“ genannt, da sie bei zweifelhaften Entscheidungen stehts bejaht. Wieder wäre die triviale Analyse dieser Art eine alles bejahende Analyse.

Die Analysen, die in dieser Arbeit behandelt werden, liefern Informationen für Optimierungen und sollten daher automatisch und korrekt sein. Sie können es sich allerdings leisten, unvollständig zu sein, da eine im Zweifel verneinte Entscheidung zwar eine Optimierung verhindert, aber auf die funktionalen Eigenschaften des Systems keine Auswirkung hat. In dieser Arbeit werden allerdings auch vollständige, aber inkorrekte Analysen behandelt, die als Voraussetzung für die dann eigentliche korrekte aber unvollständige Analyse dienen. Beispielsweise sucht eine Analyse nach singulär erstellten Betriebssystemobjekten und kann diese korrekt/pessimistisch nur dann als solche nachweisen, wenn sie zuvor in einer

[18] Beweisskizze: Gegeben ist ein Programm A und es soll berechnet werden, ob A anhält. Unter der Annahme, dass ein automatischer Analysator B im obigen Sinne existiert, wird ein Programm C entworfen, das A ausführt und eine Eigenschaft ausführt genau dann, wenn A anhält. Wenn C nun B analysieren kann, kann es auch das Halteproblem lösen.

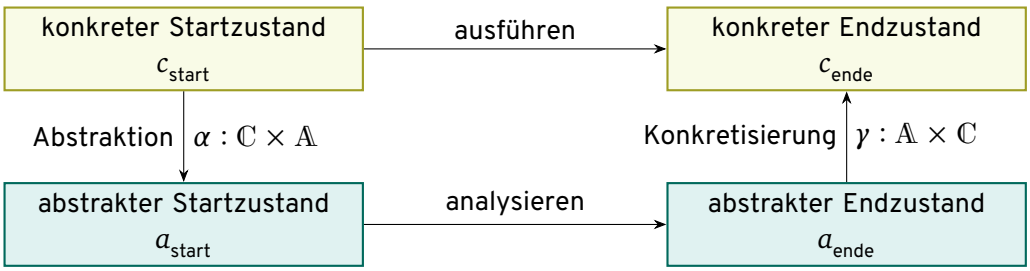



**Abbildung 3.1** Der schematische Ablauf einer abstrakten Interpretation.

Überapproximation der erreichbaren Instruktionsmenge (also einer vollständigen Menge) nach allen Stellen, an denen diese konstruiert werden könnten, gesucht hat.

## 3.2 Die Methode der Abstrakten Interpretation

Eine abstrakte Interpretation soll statisch die Existenz oder Absenz einer Programmeigenschaft nachweisen. Typischerweise weist sie dazu nach, ob das Programm dabei je nach Ziel entweder einen Zustand erreichen kann, der diese Eigenschaft erfüllt oder nie einen solchen Zustand erreichen kann [RY20A2.3].

*abstrakter und konkreter Zustand*

Eine abstrakte Interpretation arbeitet, indem sie abstrakte Zustandsübergänge auf abstrakten Zuständen ausführt, die aber derart mit den konkreten Zuständen und Zustandsübergängen zusammenhängen, dass sie entsprechende Rückschlüsse auf das Programmresultat bei einer realen Ausführung erlauben [RY20A2.3]. Dazu soll erst informell beschrieben werden, was ein Zustand ist: Ein Rechensystem arbeitet, indem es schrittweise Befehle ausführt, die dabei den aktuellen Maschinenzustand ändern. Dieser beschreibt dabei die Gesamtheit der Maschine (Arbeitsspeicher, Register, Caches). Im Namensschema der abstrakten Interpretation heißt dieser Zustand *konkreter Zustand* ($c$) aus der Menge aller konkreter Zustände $\mathbb{C}$ (Abbildung 3.1). Dem gegenüber steht ein *abstrakter Zustand* ($a$) aus der Menge aller abstrakten Zustände $\mathbb{A}$, der aus dem konkreten Zustand durch *Abstraktion* gebildet werden kann. Dies geschieht mithilfe der Abstraktionsfunktion $\alpha : \mathbb{C} \to \mathbb{A}$.

*Konkretisierung*

Auf einem konkreten Zustand ($c_n$) kann das Rechensystem nun einen Befehl oder eine Menge dieser Befehle (ein Programm p) ausführen („ausführen“), die in einem neuen konkreten Zustand resultieren ($c_{n+1}$). Die abstrakte Interpretation verhält sich analog mit einer abstrakten Operation „analysieren“ auf dem abstrakten Zustand. Für die Abstraktion gibt es eine Rückabbildung, die *Konkretisierung*, die mit der Funktion $\gamma : \mathbb{A} \to \mathbb{C}$ beschrieben wird.

*Über-/ Unterabschätzung*

Die sukzessive Anwendung der drei Funktionen $\alpha$, „analysieren“ und $\gamma$ auf einen konkreten Zustand erzeugt nun – genau wie die Anwendung der Funktion „ausführen“ einen Zustand

$c_{\text{direkt}}$ erzeugt – einen neuen konkreten Zustand $c_{\text{Analyse}}$, der das Analyseergebnis beschreibt und gewisse Eigenschaften aufweisen muss:

- Der Zustand $c_{\text{Analyse}}$ lässt einen Rückschluss auf die erforderliche Eigenschaft zu.
- Der Zustand $c_{\text{Analyse}}$ muss eine Über- bzw. Unterapproximierung des Zustandes $c_{\text{direkt}}$ sein, damit der Abstraktionsalgorithmus korrekt ist. Konkret soll der Zustand $c_{\text{Analyse}}$ den Zustand $c_{\text{direkt}}$ unterapproximieren, wenn die Existenz einer Eigenschaft nachgewiesen werden soll und überapproximieren, wenn die Absenz gezeigt werden soll.

*selektive Analyse*

Da die Abstraktion $\alpha$ zumeist Informationen des konkreten Zustandes verwirft, verändern viele Schritte, die in der Funktion „analysieren“ gemacht werden, den abstrakten Zustand nicht. Oftmals kann eine (Teil-)Menge der Schritte, die keinen Effekt auf den Zustand haben, bereits im Vorhinein erkannt und bei der Analyse entsprechend übersprungen werden. Diese Art der Analyse nennt man *selektiv („sparse“)*.

*formale Korrektheit*

Cousot & Cousot [CC77] haben gezeigt, dass, um formal nachzuweisen, dass der Analysealgorithmus korrekt ist und terminiert, es möglich ist, den Zuständen und Funktion bestimmte algebraische Eigenschaften nachzuweisen:

- $\alpha$, $\gamma$, $\mathbb{A}$ und $\mathbb{C}$ bilden einen Galois-Körper.
- Dies inkludiert: $\alpha$, $\gamma$, „ausführen“ und „analysieren“ sind monoton.
- $\mathbb{A}$ und $\mathbb{C}$ sind vollständige Verbände.

**Exkurs: Mathematische Grundlagen [SWH10A1.5,MS18A4.2]**

Um den Begriff *Verband* definieren zu können, müssen wir zuerst definieren, was eine *partielle Ordnung* und eine *obere Schranke* sind:

**Definition 5: Partielle Ordnung (Halbordnung, „partial order“).**
Sei $M$ eine Menge mit einer Relation $\sqsubseteq \subseteq M \times M$. Elemente dieser Menge sind partiell geordnet, wenn folgendes gilt:

$$x \sqsubseteq x \qquad \text{(Reflexivität)}$$
$$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y \qquad \text{(Antisymmetrie)}$$
$$x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z \qquad \text{(Transitivität)}$$

für $x, y \in M$. Ein solche Menge wird auch partielle Ordnung oder Halbordnung genannt[19].

[19] Beachte: Im Vergleich zur totalen Ordnung muss eine Ordnungsrelation nicht für jedes Paar definiert sein.

**Definition 6: Schranken.**
Sei $M$ eine partielle Ordnung. Ein Element $s \in M$ heißt *obere Schranke* für eine Teilmenge $M_{\text{Teil}} \subseteq M$, falls

$$m \sqsubseteq s \qquad \text{für alle } m \in M_{\text{Teil}}$$

Ein Element $s \in M$ heißt *kleinste obere Schranke („least upper bound")* für eine Teilmenge $M_{\text{Teil}} \subseteq M$, falls gilt:

1. $s$ ist obere Schranke.
2. $s \sqsubseteq y$ für jede obere Schranke $y$ von $M_{\text{Teil}}$.

Diese kleinste obere Schranke wird auch $\bigsqcup M_{\text{Teil}}$ bezeichnet.

Für die *(größte) untere Schranke („greatest lower bound")* $\bigsqcap M_{\text{Teil}}$ gilt eine analoge Definition.

Damit ist ein vollständiger Verband definierbar:

**Definition 7: Vollständiger Verband („complete lattice").**
Eine partielle Ordnung $V$ ist ein *vollständiger Verband*, falls jede Teilmenge $V_{\text{Teil}} \subseteq V$ eine kleinste obere Schranke $\bigsqcup V_{\text{Teil}}$ und eine größte untere Schranke $\bigsqcap V_{\text{Teil}}$ besitzt.

Im Gegensatz zum (unvollständigen) Verband besitzen bei einem vollständigen Verband auch die leere Menge eine größte untere Schranke und die Menge aller Elemente eine kleinste obere Schranke. Daraus folgt die Existenz zweier zusätzlicher Elemente *Top* ($\top$) und *Bottom* ($\bot$), die jeweils größer oder gleich bzw. kleiner oder gleich jedem anderen Element des vollständigen Verbandes sind.

Auf Basis von Halbordnungen ist es möglich, einen Galois-Körper zu definieren.

**Definition 8: Galois-Körper.**
Ein Galois-Körper ist eine Abbildung zwischen zwei Halbordnungen $C$ und $A$:

$$C \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} A$$

$\alpha : C \times A$ und $\gamma : A \times C$ – die Abbildungsfunktionen – müssen dabei monoton sein. Für eine monotone Funktion $f$ gilt:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Unter dieser Voraussetzung gilt:

$$c \sqsubseteq \gamma(\alpha(c)) \qquad \text{mit } c \in C$$

In anderen Worten: Wenn wir bei einer statischen Analyse den Systemzustand als Halbordnung definieren und die Abstraktionsfunktion $\alpha$ und die Konkretisierungsfunktion $\gamma$ monoton gestalten, ist eine Konkretisierung der Abstraktion eines Zustandes nachweisbar größer oder gleich dem Zustand. Wenn wir dazu noch die Ordnungsrelation so gestalten, dass „größere“ Zustände in Bezug auf den Nachweis einer Eigenschaft „kleinere“ Zustände überapproximieren, dann ist damit die Korrektheit des Algorithmus mathematisch nachweisbar.

## 3.3 Terminierung der Analyse

Mithilfe der obigen mathematischen Formalisierung ist nachweisbar, dass eine abstrakte Interpretation terminiert [SWH10A1.5,MS18A4.4]. Der grundsätzliche Ansatz hierfür ist die Analyse so zu gestalten, dass sie einen Fixpunkt findet.

**Definition 9: Fixpunkt.**
Wenn für eine Funktion $f$ gilt: $f(x) = x$, so heißt $x$ *Fixpunkt*.

Die folgenden beide Sätze garantieren unter bestimmten Bedingungen die Existenz eines Fixpunktes.

**Definition 10: Satz von Kleene [Rog69A11.2].**
In einem vollständigen Verband $V$ mit finiter Höhe hat jede monotone Funktion $f : L \times L$ einen eindeutigen kleinsten Fixpunkt $lfp(f)$, der definiert wird als:

$$lfp(f) = \bigsqcup_{i \leq 0} f^i(\bot)$$

Dieser Satz garantiert die Terminierung, wenn der vollständige Verband, den die abstrakten Zustände beschreiben, finiter Höhe ist. Da dies nicht allgemein garantiert ist, ist weiterhin folgender Satz anwendbar:

**Definition 11: Satz von Knaster – Tarski [Tar55].**
In einem vollständigen Verband $V$ hat jede Funktion $f : V \to V$ einen kleinsten Fixpunkt $v_0$, welcher auch die kleinste Lösung der Ungleichung $x \sqsupseteq f(x)$ ist.

*Widening*

Die Gleichsetzung des Supremums über eine Mehrfachausführung der Funktion $f$ mit dem Fixpunkt im Satz von Kleene beschreibt bereits einen (ineffizienten) Algorithmus zum Auffinden des Fixpunktes: Die Funktion $f$ muss nur solange auf das Resultat ihrer selbst angewendet werden, bis ein Fixpunkt erreicht ist. Der Satz von Knaster – Tarski beschreibt keinen Weg, um den Fixpunkt zu finden und setzt durch das Zulassen von vollständigen

Verbänden infiniter Höhe eine neue Technik voraus: Das *Widening*. Die in dieser Arbeit entworfenen Algorithmen arbeiten sowohl auf Zuständen finiter Menge, sodass der Satz von Kleene eine Terminierung garantiert, als auch auf Zuständen infiniter Menge, sodass in diesem Fall ein Widening-Operator benötigt wird. Ich werde diese Technik darum im Folgenden beschreiben.

Beide Sätze setzen außerdem die Existenz einer (mathematischen) Funktion voraus, die aber nicht unmittelbar gegeben ist: Die abstrakte Interpretation folgt erst einmal nur einem Programmstrom. Diese Funktion kann aber generisch aus dem Kontrollflussgraph gewonnen werden, was ich ebenso im Folgenden beschreiben werde.

## 3.4 Kontrollflussgraph und Gleichungssystem

*Kontrollflussgraph*

Ein Programm wird vom Prozessor Instruktion für Instruktion abgearbeitet. Für gewöhnlich bilden die Instruktionen dabei eine lineare Kette: Sie werden hintereinander abgearbeitet. Einige Instruktionen können aber diese Kette durchbrechen („Sprünge“ auf Maschinensprachebene oder Schleifen, Verzweigungen und Prozeduren auf Hochsprachebene) und machen aus der Kette einen Graphen. Ein Pfad durch diesen Graphen wird Kontrollfluss genannt, der Graph heißt dementsprechend *Kontrollflussgraph (Control-Flow Graph, CFG)* [ALS+08A8.4].

*lokaler und interprozeduraler Kontrollfluss*

Generell wird zwischen dem *lokalem (LCFG)* und *interprozeduralem (ICFG)* Kontrollflussgraphen unterschieden. Ersterer bildet ausschließlich den Kontrollfluss innerhalb von Funktionen bzw. Prozeduren ab, letzter erweitert den Kontrollfluss um Kanten zwischen Prozeduren. Während der lokale Kontrollfluss bei den meisten Programmiersprachen rein syntaktisch bestimmbar ist, ist die Extraktion des interprozeduralen Kontrollflusses von der Bestimmung der Prozeduraufrufziele abhängig. Während dies bei einigen Aufrufen ebenfalls unmittelbar im Quellcode steht (direkter Aufruf, statisch), sind bei Funktionszeigern die Ziele nur dynamisch bestimmbar (indirekter Aufruf, dynamischer Kontrollfluss). In diesem Fall ist eine vorhergehende statische Analyse notwendig, um die Zeigerziele korrekt zu approximieren.

*Basisblock*

Für Analysen und Transformationen sind im Kontrollfluss oftmals Verzweigungen oder Schleifen interessant (also Knoten mit einem Ein- oder Ausgangsgrad größer als eins). Im Compilerbau existiert darum eine Technik, Ketten von Instruktionen zusammenzufassen:

> **Definition 12: Basisblock (manchmal auch Grundblock) [ALS+08A8.4,HP19A3.1].**
> Ein *Basisblock (Basic Block, BB)* ist eine Folge von Anweisungen, bei der nur das erste und das letzte Element einen Ein- oder Ausgangsgrad größer als eins haben dürfen.

Abbildung 3.2 zeigt den lokalen und interprozeduralen Kontrollflussgraph eines Beispielprogramms.

```
1  unsigned x = user_input();
2  int y = 0;
3  while (x != 0) {
4      x -= 1;
5      y += 2;
6  }
```

**Codeblock 3.2** Eine Schleife, die eine Variable hochzählt.

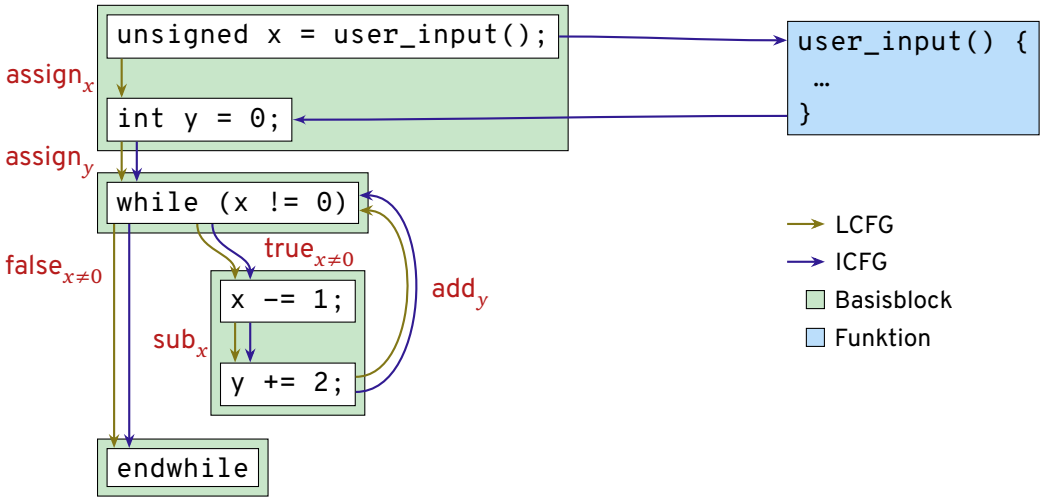



**Abbildung 3.2** Der Kontrollfluss des Codes aus Quellcode 3.2.

*Abbildung auf Gleichungssystem*

Mithilfe des Kontrollflusses kann nun der Algorithmus der abstrakten Interpretation in ein Ungleichungssystem überführt werden. Sämtliche Kanteneffekte – also die konkreten Anwendungen der Funktion „analysieren“ – werden dazu als Funktion über die eingehenden abstrakten Zustandsmengen aufgefasst und mit den tatsächlichen abstrakten Zuständen verglichen [SWH10A1.3]. Sei $A_i$ die abstrakte Zustandsmenge zum Zeitpunkt der Anweisung $i$ (also vor deren Ausführung) und $[\![k]\!]^\#$ die Funktion, die den Kanteneffekt der Kante $k$ auf die abstrakte Zustandsmenge bei Ausführung der Anweisung beschreibt, dann ergibt sich folgendes Ungleichungssystem für den Code in Quellcode 3.2 (vergleiche die entsprechenden Kanteneffekte aus Abbildung 3.2):

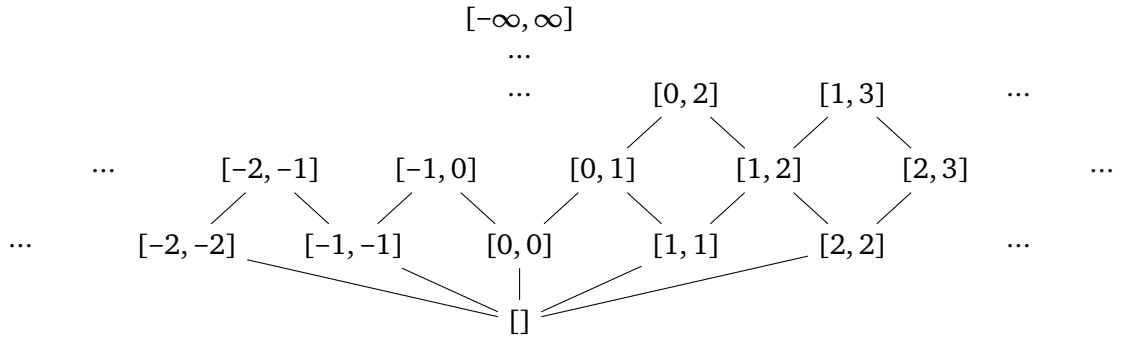



**Abbildung 3.3** Die abstrakte Intervalldomäne. Sie bildet einen vollständigen Verband infiniter Höhe (nach [MS18A6.1]).

$$
\begin{aligned}
A_0 &\subseteq \emptyset && = f_0(A_0, \dots, A_5) \\
A_1 &\subseteq [[\text{assign}_x]]^\#(A_0) && = f_1(A_0, \dots, A_5) \\
A_2 &\subseteq [[\text{assign}_y]]^\#(A_1) && = f_{2a}(A_0, \dots, A_5) \\
A_2 &\subseteq [[\text{add}_y]]^\#(A_4) && = f_{2b}(A_0, \dots, A_5) \\
A_3 &\subseteq [[\text{true}_{x\neq 0}]]^\#(A_2) && = f_3(A_0, \dots, A_5) \\
A_4 &\subseteq [[\text{sub}_x]]^\#(A_3) && = f_4(A_0, \dots, A_5) \\
A_5 &\subseteq [[\text{false}_{x\neq 0}]]^\#(A_2) && = f_5(A_0, \dots, A_5)
\end{aligned}
$$

Wird jede Funktion nun als Funktion über alle Instruktionen betrachtet und das gesamte Ungleichungssystem als mehrdimensionale Funktion, ergibt sich die für den Satz von Kleene und Knaster – Tarski geforderte Funktion [SWH10A1.4].

## 3.5 Widening

| Schritt | Intervalle ohne Widening | Intervalle mit Widening |
|---|---|---|
| 0 | $x = [0, \infty], y = [0, 0]$ | $x = [0, \infty], y = [0, 0\ \ ]$ |
| 1 | $x = [0, \infty], y = [0, 2]$ | $x = [0, \infty], y = [0, 2\ \ ]$ |
| 2 | $x = [0, \infty], y = [0, 4]$ | $x = [0, \infty], y = [0, \infty]$ |
| 3 | $x = [0, \infty], y = [0, 6]$ | |
| 4 | ... | |

**Tabelle 3.1** Iterationsschritte bei einer abstrakten Interpretation des Codes aus Quellcode 3.2 auf der abstrakten Intervalldomäne.

Das Widening ist eine Methode, eine korrekte Überapproximation einer abstrakten Domäne zu berechnen [RY20A3.3,CC77]. Sowohl das Problem als auch die Lösung lassen sich gut am Beispiel aus Quellcode 3.2 verdeutlichen. Dort ist eine Schleife zu sehen, die eine Integer-Variable `y` verändert. Wir sind in diesem Fall daran interessiert, in welchem Intervall der Wert dieser Variable sein kann und wählen darum die abstrakte Intervalldomäne. Diese Domäne bildet einen vollständigen Verband infiniter Höhe (Abbildung 3.3). Die Anwendung des naiven Fixpunkt-Algorithmus aus Abschnitt 3.3 führt in den ersten Iterationsschritten zu den Werten, die in Tabelle 3.1 dargestellt sind und terminiert offensichtlich nicht.

*Widening-Operator*

In diesem Fall setzt nun das Widening ein, indem es eine Überabschätzung in genau dem Punkt vornimmt, in dem mehrere Intervalle verschmolzen werden müssen, im Beispielfall im Schleifenkopf. Der Operator kann dazu verschieden ausgelegt sein. Eine Möglichkeit ist es, bei jedem Durchlauf die Anzahl variabler Werte zu verringern. Für die abstrakte Intervalldomäne gibt es generell zwei variable Werte: die untere und die obere Grenze. Im Beispiel ist die untere Grenze von Iterationsschritt zu Iterationsschritt stabil. Ein möglicher Widening-Operator kann also in diesem Fall unterschiedliche Werte bei der oberen Grenze detektieren und diese dann im entsprechenden Verband auf den größtmöglichen Wert setzen: $\infty$. Dadurch ergibt sich beim Anwenden des Operators bereits nach 2 Schritten das Interval $[0, \infty]$. Der Widening-Operator senkt die Präzision der Analyse u. U. enorm, weswegen der Entwurf eines guten Operators essentiell ist und verschiedene Techniken existieren, die Präzision wieder zu erhöhen [SWH10A1.10,RY20A5.2.2]

Mit dem Widening habe ich jetzt alle theoretischen Schritte beschrieben, um praktisch eine abstrakte Interpretation durchzuführen, die aber theoretisch formalisierbar ist. Ich will im Folgenden aber noch auf zwei weitere Themen der statischen Analyse eingehen, die für meine Forschungsfragen relevant sind: Die Mehrkernanalyse, die kein Teil der klassischen abstrakten Interpretation ist und die Konstruktion eines Werteflussgraphen.

## 3.6 Mehrkernanalyse

*Kreuzprodukt bei Mehrkern-analysen*

Die abstrakte Interpretation geht in ihrer Grundform immer von einem Faden, bzw. einer aktiven Ausführungseinheit aus. Während dies für viele statische Analysen ausreichend ist (z. B. Konstantenpropagierung oder das Finden gemeinsamer Ausdrücke) und auch unzählige verschiedene abstrakte Interpretationen entstanden sind ([CC14] liefert einen guten Überblick), sind erst vermehrt in den letzten Jahren Analysen für Mehrkernsysteme entstanden. Das Hauptproblem bei dieser Art von Analyse ist die fehlende Abschätzung darüber, in welcher zeitlicher Relation („Interleavings“) die Fäden zueinander sind. Der naive Ansatz dafür ist die Berechnung *aller* möglichen Zustände, wie die Kerne zueinander stehen können. Dies ist damit gleichbedeutend mit dem Kreuzprodukt aller Einzelzustände auf dem jeweiligen Kern und damit für praktische Belange zu kompliziert [Min12].

*Reduktion der Domäne*

Es gibt einige modellbasierte Ansätze, um mit Interaktionen mehrerer Kerne umgehen zu können: So benutzen Mittermayr et al. die Kronecker-Algebra, eine Methode, um parallellaufende Automaten in einer Matrix zu kodieren [MB16, MB21]. Damit sind sie in der Lage, eine dünn besetzte Matrix aufzustellen, die das Wissen von Barrieren (wie Semaphoren) abbilden kann und in der tatsächlich gangbare Wege erst bei Bedarf („lazy“) berechnet werden können. Die Modellierung ist flusssensitiv, erfordert aber die Überführung in Automaten und ist damit eher modellbasiert. Die Analyse dient in diesem Fall der Verklemmungserkennung und zur Analyse der WCET. Haur et al. wählen ebenfalls einen modellbasierten Ansatz und modellieren die Fadenrelationen untereinander mit Hilfe von speziellen Petrinetzen, den „high-level colored time Petri nets“ [HBR21,HBR22]. Auch sie suchen nach inkorrekten Mustern von Sperren. Die Modellbildung ist in diesem Fall manuell und damit aufwendig. Der vergleichsweise alte Ansatz von Callahan et al. geht in eine ähnliche Richtung, indem er zwar einen korrekten Kontrollflussgraphen über Kerngrenzen hinweg aufstellt, aber die Domäne stark reduziert (nur parallele Schleifen, keine verschachtelten Schleifen) [CS88].

*Sperrenmuster-analysen*

Weiterhin gibt es Ansätze, im Speziellen nach Verletzung von Sperrenmustern zu suchen: Engler et al. machen dies ebenfalls mit einer interprozeduralen und flusssensitiven Analyse, die allerdings nicht zwangsläufig korrekt ist und sich daher an die Anwendungsentwicklung richtet, bei der die entstehenden falsch-positiven Resultate manuell überprüft werden können [EA03]. Kroening et al. erkennen mit einer korrekten Analyse Verklemmungen (für POSIX/pthreads) [KPS+16]. Sie bauen dafür einen speziellen Sperrengraphen. Beide Ansätze konzentrieren sich ausschließlich auf Sperren und lassen andere Betriebssystemkonzepte außer acht. Andere Analysen sind automatisch, verzichten aber an bestimmten Stellen auf Genauigkeit. Dabei wären vor allem die Analyse-Programme Astrée und GOBLINT zu nennen, die ich im nächsten Kapitel 4 detaillierter behandele.

Im Gegensatz zu meiner eigenentwickelten Analyse, die ich im Kapitel 7 vorstelle, verfolgen alle diese Analysen nicht den Zweck der Optimierung, sondern den der Erkennung von Wettlaufsituationen und sind darum darauf ausgelegt, vor allem die Teile des Systems zu detektieren, die mit Sperren arbeiten. Insbesondere interpretieren die Analysen nicht das Scheduling-Verhalten auf dem einzelnen Kern zusätzlich zu Mehrkerninteraktionen.

## 3.7 Wertanalyse

Bei der abstrakten Interpretation von Anweisungen muss der abstrakte Eingangszustand hinreichend genau sein, um den Effekt der unterliegenden Anweisung berechnen zu können. Insbesondere arbeitet die Anweisung auf Daten, deren Wert oder Wertebereich feststehen muss. Genau diese Werte zu bestimmen, ist Aufgabe einer Wertanalyse [SKR90].

*Werteflussgraph*

Bei der klassischen abstrakten Interpretation (wie z. B. Astrée eine implementiert [CCF+05]), ist diese Analyse inklusive: Der abstrakte Zustand besteht letztlich aus einer Sammlung von

```
1 int increment(int* arg) {
2   return *arg + 1;
3 }
4
5 int main() {
6   int a = 5;
7   int* ptr = &a;
8   return increment(ptr);
9 }
```

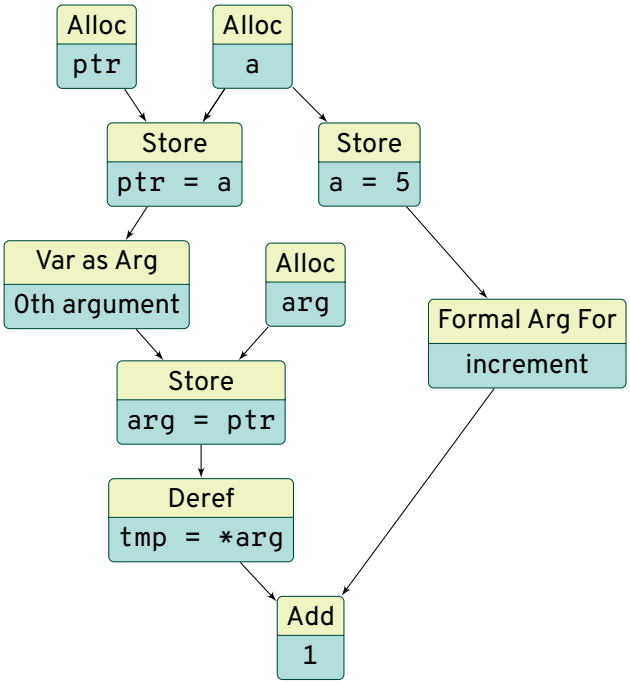



**Abbildung 3.4** Ausschnitt aus dem SVFG (vereinfacht). Das Beispiel zeigt den Fluss des Wertes für die lokale Variable `a` über den Zeiger `ptr` als erstes Argument in die Funktion `increment`. Die SVF [SX16] verknüpft initial dabei nicht die Zuweisung von 5 mit dem weiteren Fluss (flussinsensitiv), erkennt aber die Möglichkeit eines solchen (abgebildet durch die Zusatzkante über den „Formal Arg For“-Knoten).

Werten oder Wertebereichen. Es gibt aber auch Analysen, die damit arbeiten, eine Zusatzdarstellung zu schaffen, die die Suche nach Werten deutlich vereinfacht: Den *Wertefluss-graphen*. Diese Darstellung wandelt die vormals ausschließlich semantisch erkennbaren Informationen über den Wertefluss in einen expliziten (syntaktisch darstellbaren) Graphen um und ermöglicht, Datenflussanalysen als Graphproblem zu formulieren [SKR90].

*SSA* Im Gegensatz zum Kontrollflussgraphen, der die Reihenfolge beschreibt, in der eine CPU Anweisungen abarbeitet, beschreibt ein Wertflussgraph die logischen Abhängigkeiten von Werten zueinander. Der Werteflussgraph wird aus dem Kontrollfluss gewonnen und besteht aus verschiedenen Ebenen. Eine Darstellung, die am dichtesten dem Kontrollfluss folgt, ist die Transformation in die *statische Einzelzuweisungsform (Static-Single-Assignment, SSA)*. Die SSA ist ebenfalls ein Kontrollflussgraph, allerdings sind Datenabhängigkeiten explizit dargestellt, indem jede Zuweisung einer Variablen immer singulär ist (praktisch wird dies durch das Hinzufügen von Hilfsvariablen erreicht) [ALS+08A6.2]. Wird eine Variable immer nur singulär zugewiesen und damit definiert, liegt es nahe, die Benutzungen dieser direkt mit der Definition zu verbinden. Das ergibt „Use-Define-Chains“ (die insgesamt einen Graphen darstellen) und kann für diverse Analysen und Optimierungen im Übersetzer

genutzt werden [ALS+08A9.2]. Beide Techniken werden von LLVM, einem Übersetzer-Framework, auf dem ich aufbaue, implementiert [LA04]. Sie alleine sind ausreichend, um Argumente zu finden, die konstante globale Daten sind, wie z. B. von dOSEK genutzt (Abschnitt 4.5). Sie bilden aber keinen indirekten Datenfluss ab, der seinen Weg über per Zeigern adressierten Speicher nimmt. Dazu braucht es komplexere *Werteflussanalysen*, u. a. eine Alias-Analyse („Welche Zeiger zeigen auf den gleichen Wert?“) [MS18A11].

*Wertefluss-analysen*

Da ich eine solche komplexere Analyse benötige, aber diese ein eigenes Forschungsgebiet darstellt und bereits zahlreiche Implementierungen und gesonderte Algorithmen existieren (außer der SVF u. a. [RHS95,LBL+11,SHB19,HHF22]), habe ich mich bei der Konstruktion eines Werteflussgraphen für den *Sparse Value Flow (SVF)* entschieden [SX16]. Dieser Algorithmus berechnet zuerst Zeigerziele, um daraus einen interprozeduralen Kontrollfluss zu konstruieren und erzeugt anschließend flussinsensitiv den Fluss von Werten. Er interpretiert dazu sämtliche Anweisungen auf der LLVM-Zwischensprache (mehr dazu in Kapitel 5), die einen Datenfluss darstellen können (z. B. eine Zuweisung an ein Register, eine Dereferenzierung eines Zeigers, ein Speichern in den Hauptspeicher usw.), und verbindet jeden Fluss in einem gemeinsamen Graphen, dem *Sparse Value-Flow Graph (SVFG)*. Insbesondere passiert diese Analyse auch über Funktionsgrenzen hinweg, sodass auch die Werte von Argumenten, die in eine Funktion fließen, also an sie übergeben werden, mit den entsprechenden Parametern innerhalb der Funktion verknüpft sind. Abbildung 3.4 zeigt eine vereinfachte Darstellung des SVFGs. Eine entsprechende Wertanalyse kann dann auf dem SVFG basieren, wie ich in Kapitel 5 detaillierter erläutere.

## 3.8 Zusammenfassung

Wir können für die Zusammenfassung wieder auf Abbildung 3 zurückkommen, deren erster Teil durch dieses Kapitel abgedeckt wird. Aus dem großen Bereich der statischen Analyse konzentriert sich diese Arbeit auf die abstrakte Interpretation. Dort haben Cousot und Cousot [CC77] gezeigt, dass die mathematische Verbandstheorie mit dem Übersetzermodel des Kontrollflusses kombinierbar ist, um korrekte statische Analysen entwickeln zu können, die darauf basieren, dass der Effekt auf einen abstrakten Zustand Anweisung für Anweisung abstrakt interpretiert wird.

Die grundlegende Analyse kann in mehreren Richtungen erweitert werden: Werden nur Teile des Kontrollflusses interpretiert, handelt es sich um eine selektive Analyse. Besitzt das analysierte System mehrere Kerne, benötigt es eine Mehrkernanalyse, die (bislang) Informationen verwirft, um skalierbar zu sein. Soll interprozedural analysiert werden, ist eine vorhergehende Zeigeranalyse nötig. Damit verschränkt existiert die Spezialdarstellung des Werteflusses, der selektiv auf Basis des dynamischen Kontrollflusses den Fluss von Werten über Variablengrenzen hinweg darstellt (und seinerseits mithilfe einer Fixpunktanalyse den dynamischen Kontrollfluss verbessern kann).

Die bisher vorgestellten Analysen sind sich nur rudimentär des Betriebssystems bewusst (z. B. um Sperren erkennen zu können). Im nächsten Kapitel werde ich auf betriebssystemgewahren Analysen zu sprechen kommen und auch ihre Unzulänglichkeiten deutlich machen. Anschließend werde ich auf die Verbindung der Einzelteile eingehen. Die im folgenden Kapitel beschriebenen betriebssystemgewahren Analysen lassen sich auf die eben vorgestellten Prinzipien der abstrakten Interpretation abbilden, was einerseits ihre Terminierung zeigt, anderseits aber auch Unzulänglichkeiten aufdeckt.

# 4

# Betriebssystemgewahre Analyse

## Stand der Kunst und theoretische Einordnung

In den letzten beiden Kapiteln habe ich allgemein die Methode der statischen Analyse bzw. abstrakten Interpretation sowie die Domäne der eingebetteten Systeme im Speziellen der Echtzeitsysteme vorgestellt. Dabei bin ich auf verwandte Arbeiten eingegangen, die sich zwar in dem Sinne des Betriebssystems gewahr sind, als dass sie bestimmte seiner Aufrufe abstrakt interpretieren können (z. B. um kritische Bereiche zu erkennen), dieses jedoch nicht als hauptsächliches Ziel haben. In diesem Kapitel werde ich auf *betriebssystemgewahre Analysen* und das es umspannende Forschungsfeld eingehen. Abschließend werde ich die SSE und SSF, zwei existierende betriebssystemgewahre Analysen theoretisch einordnen.

Betriebssystemgewahre Analysen untersuchen Systeme ob der Verwendung ihres Betriebssystems. Diese Arbeit behandelt ausschließlich eingebettete Systeme, sodass in diesem Kapitel die Methode der statischen Analyse mit der Domäne der eingebetteten Systeme kombiniert wird. Zunächst ist eine genauere Definition der betriebssystemgewahren Analyse wichtig.

**Definition 13: Betriebssystemgewahre Analyse.**
Eine statische Analyse ist dann betriebssystemgewahr, wenn sie eine Anwendung dahingehend untersucht, welche und wie sie die virtuellen Betriebsmittel des RTOS – die Instanzen – benutzt. Je nach Analyse ist das die Menge der Instanzen, deren genaue Ausprägung, deren Interaktion oder deren (exaktes) Verhalten auf den Kontrollfluss.

Es gibt eine Vielzahl von betriebssystemgewahren Analysen. Ich gehe hier zuerst auf das Forschungsfeld im Gesamten ein, um anschließend vier Analysen detaillierter vorzustellen, die mit meiner Arbeit am verwandtesten sind.

## 4.1 Allgemeines Forschungsfeld

*Formale Verifizierung*

Ein großes Forschungsgebiet ist es, die Betriebssystemimplementierung oder Teile davon formal zu verifizieren. Viele Arbeiten zielen auf OSEK ab [CA11, VCY+16, DFW+21, ZCO14, HZZ+11, ZAC15, FKD+12], aber auch FreeRTOS [CJ21, SYY+15, DGM09] oder Zephyr [FYD+19] wurden dahingehend untersucht. Eine weitere Methode ist die Systemmodellierung mithilfe von „timed automata“ (einer Erweiterung klassischer endlicher Automaten um die Modellierung von Zeitverhalten), um darauf die formale Korrektheit des Systems festzustellen bzw. ein formal korrektes System generieren zu können [BHM08, BLN+23]. Die Ansätze sind aber nicht automatisch, zielen zumeist nur auf einen kleinen Teil des Betriebssystems ab und wollen insbesondere nicht das Verhalten einer konkreten Anwendung erfassen. Sie treffen daher nur bedingt auf die obige Definition zu.

*WCET-Analyse*

Weiterhin gibt es im Bereich der WCET-Analyse Arbeiten, die sich im Speziellen auf Betriebssystemsemantik konzentrieren. Schuster et al. stellt mit SWAN einen Analysator für das ganze System vor, der zur WCET-Analyse nicht nur den Anwendungscode, sondern auch den Betriebssystemcode analysiert [SWU+19]. Wieder et al. stellen eine WCET-Analyse für verschiedene Sperrenmechanismen von AUTOSAR vor und schlagen überdies unterbrechbare Sperren für AUTOSAR vor, die WCET-Analysen deutlich vereinfachen würden [WB13].

*Entfernung von totem Code*

Eine weitere Klasse von Arbeiten beschäftigt sich mit der Eliminierung von totem Code in der Betriebssystemimplementierung. Während dies bei eingebetteten Systemen, in denen Systemaufrufe zumeist als normale Funktionen implementiert sind und damit bereits mit normalen Mitteln des Übersetzers im Rahmen einer Linkzeitoptimierung entfernt

werden können, unproblematisch ist, stellt die Verfolgung des Kontrollflusses aus unprivilegiertem Anwendungscode in privilegierten Kerncode ein Problem dar. Bertran et al. konstruieren darum flussinsensitiv einen globalen Kontrollfluss für Linux, der den Instruktionsstrom durch das Betriebssystem abbildet [BGC+06]. Rajagopalon et al. [RDH+05] und Chanet et al. [CDD+05] benutzen beide „Binary Rewriting“ (also das Modifizieren des bereits übersetzten Systemabbildes), um unbenutzte Teile des Kerns zu entfernen. Lee et al. extrahieren statisch den Aufrufgraphen der Anwendung für Linux für den gleichen Zweck [LLH+04]. Bei all diesen Ansätzen geht es um die Entfernung von totem Code und nicht um Optimierungen, die ein flusssensitives oder kontextbasiertes Verständnis über die genaue Benutzung des Betriebssystems seitens der Anwendung voraussetzen.

*Finden von Protokollverletzungen*

Mit SLAM [BR02], SDV [BBL+06] (für Windows) und deren Erweiterung auf Linux [PK07] existiert eine Familie von statischen Analysen, die einen Treiber nach Protokollverletzungen hinsichtlich der Betriebssystemkonzepte untersuchen. Obwohl ein Treiber normal der Betriebssystemimplementierung zugerechnet wird, fungiert er hier doch als Anwendung, die korrekt auf die Schnittstellen des (Kern-)Betriebssystems zugreifen muss, sodass die Analyse betriebssystemgewahr ist. Es handelt sich hier aber um die Domäne der generischen Systeme und um interne Betriebssystemschnittstellen.

*Ideale Konfiguration*

Es gibt außerdem noch den Ansatz, einen Betriebssystemkern zu erzeugen, der für die Anwendung ideal konfiguriert ist (speziell bei Linux mit mehreren tausend Konfigurationsoptionen lohnenswert). Ruprecht et al. lösen dies über eine kombinierte Form der dynamischen und statischen Analyse. Eine dynamische Analyse stellt die benötigten Funktionen des Kerns fest und eine statische Analyse erzeugt dafür die passende Kernkonfiguration [RHL14,DTS+12]. Schirmeier et al. berechnen eine möglichst ideale Konfiguration für eCos (ein RTOS), indem sie rein statisch den Kontrollflussgraphen in Kripke-Strukturen überführen, auf denen dann per Modellprüfung die benötigten Konfigurationsoptionen extrahierbar sind [SBS+10]. Während die Analyse betriebssystemgewahr ist, extrahiert sie kein feingranulares Wissen oder zielt auf mehrere Betriebssysteme ab.

*OS-Prozessorregister*

Wie diese Arbeit schlägt Berthelmann eine Analyse vor, die speziell Maßschneiderung von Anwendung und Betriebssystem zum Ziel hat. Er will dazu konkret Übersetzer und Betriebssystemimplementierungen stark integrieren, um den Speicherverbrauch zu reduzieren, indem effizient verwendete Prozessorregister den Task-Zustand zwischenspeichern [Bar02]. Er geht aber im Gegensatz zu dieser Arbeit z. B. nicht weiter auf Kontrollflussoptimierungen oder andere Optimierungen ein.

## 4.2 Astrée

Astrée ist ein proprietäres Programm zur statischen Analyse, das initial 2005 von Patrick und Radhia Cousot (die auch die Theorie der abstrakten Interpretation ausarbeiteten) entwickelt

wurde und aktuell von der Firma AbsInt vertrieben wird [CCF+05]. Anwendungszweck ist der Nachweis von Laufzeitfehlern der Anwendung auf Basis des C99-Standards. Astrée ist korrekt: Wenn es keine Laufzeitfehler findet, sind im Programm auch keine vorhanden. Es wurde initial für Anwendungen des RTOS-Standards ARINC 653 entwickelt, der vor allem von der Flugzeugindustrie verwendet wird [AEE03]. Seither ist es aber um die Unterstützung für POSIX und OSEK/AUTOSAR erweitert worden [KMS+17]. Um 2012 ist es zudem vor allem von Antoine Miné um Unterstützung für Mehrkernanwendungen erweitert worden [Min12].

*Mehrkernanalyse*

Astrée implementiert die klassische abstrakte Interpretation. Es unterstützt dafür mehrere abstrakte Domänen, die zusammenspielen können. Relevant für diese Arbeit ist vor allem die Mehrkernanalyse, die aufgrund ihrer Skalierbarkeit als Goldstandard unter ihresgleichen gilt [SSS+21]. Astrée unterteilt dabei sämtliche Daten in fadenlokale und globale Daten. Anschließend wird jeder Faden getrennt mithilfe einer abstrakten Interpretation analysiert. Fadenlokale Daten werden klassischerweise behandelt. Wann immer die Analyse allerdings auf globale Daten trifft, werden diese fadenübergreifend behandelt: Da die Analyse nicht in der Lage ist, die Beziehung der Fäden zueinander zu determinieren, muss sie die Reihenfolge sämtlicher Schreibzugriffe auf globale Daten als beliebig annehmen. Astrée gruppiert darum alle globalen Daten und behandelt diese flussinsensitiv. Da zusätzlich lokale Daten von globalen Daten abhängen können, kann die einmalige Analyse aller Fäden dabei nicht ausreichen: Die abstrakte Darstellung von globalen Daten kann bei der späteren Analyse anderer Fäden derart geändert werden, dass sie die der lokalen Daten beeinflusst. Astrée löst dies durch die mehrfache Analyse aller Fäden, bis sich die abstrakte Repräsentation sämtlicher lokalen und globalen Daten stabilisiert hat. Um die Präzision zu erhöhen, unterstützt Astrée die Analyse von Lock-Mustern: Greift ein Faden innerhalb eines Locks mehrfach schreibend auf globale Daten zu, wird nur der letzte Schreibzugriff in die globale Sammlung aufgenommen. Astrée wurde 2017 von Miné außerdem um das Scheduling-Verfahren von OSEK/AUTOSAR erweitert [Min17]. Miné behandelt in diesem Fall die verschiedenen Aktivitäten von OSEK auf einem Einkernsystem wie Fäden auf einem Mehrkernsystem und kann damit die gleichen Analysen bezüglich Wettlaufsituationen anwenden wie bei der Mehrkernanalyse. Konkret behandelt er OSEK-Ressourcen wie Sperren und führt eine zusätzliche Operation „yield“ ein, die signalisiert, dass eine Aktivität an dieser Stelle von einer anderen unterbrochen werden kann. Durch die Berücksichtigung des Scheduling-Verfahrens von OSEK (insbesondere durch die Implementierung des PCP), kann Miné die Menge an Interfadenunterbrechungen deutlich reduzieren. Das Verfahren ist also quasi eine Rückabbildung der flussinsensitiven Mehrkernanalyse auf ein präemptives Einkernsystem bei zusätzlicher Ausnutzung des Schedulers, bringt dabei aber auch die gleichen Nachteile mit sich.

Die vorgestellte Analyse hat den großen Vorteil, dass sie gut skaliert, da sie die Fäden getrennt analysieren kann. Der Preis dafür ist aber die flussinsensitive Abbildung globaler Daten und damit auch die flussinsensitive Darstellung von Fadeninteraktionen.

## 4.3 GOBLINT

*Mehrkernanalyse über Einschränkungen mit Seiteneffekten*

GOBLINT ist ein Programm zur statischen Analyse von C-Programmen mit mehreren Fäden [SSS+21]. Wie Astrée will GOBLINT die Abwesenheit von Laufzeitfehlern nachweisen und legt damit den Fokus im Vergleich zu dieser Arbeit nicht auf die Verwendung des Betriebssystems und dessen Optimierung im Speziellen. Es analysiert dazu eine Untermenge von C (es verbietet beispielsweise Zeigerarithmetik) und C++ [VV09,HA12]. GOBLINT ist aber in der Lage, kritische Abschnitte zu erkennen, ein Konzept, das durch das Betriebssystem realisiert wird. Konkret arbeitet GOBLINT auf POSIX, wurde aber ebenfalls auf OSEK portiert [SSV+11]. Im Vergleich zu Astrée ist GOBLINT keine klassische abstrakte Interpretation, sondern arbeitet mit *Einschränkungen („Constraints“)*. Mit einer klassischen abstrakten Interpretation werden dazu aus dem Kontrollflussgraph Einschränkungen extrahiert: der Effekt, den die jeweilige Anweisung auf eine abstrakte Domäne hat. Als Besonderheit zu klassischen Einschränkungssystemen benutzt GOBLINT *Einschränkungen mit Seiteneffekten*, die Effekte auf mehrere abstrakte Domänen gleichzeitig ausdrücken können und dazu genutzt werden, die Effekte auf eine globale Domäne separat auszudrücken [SVM03]. GOBLINT setzt damit u. a. die Analyse von Mehrfädigkeit um: Alle Auswirkungen auf andere Fäden werden in der globalen Domäne gesammelt.

*Einordnung*

Auf dem extrahierten (möglicherweise infiniten) Einschränkungssystem kann GOBLINT anschließend bedarfsgerecht mithilfe eines „Constraint Solvers“ mögliche Variablenwerte bestimmen, was letztlich einer Rückwärtssuche durch das System an Einschränkungen entspricht. Schwarz selbst bezeichnet das System als nicht vergleichbar mit denjenigen in Astrée (das eine kann nicht mithilfe des anderen abgebildet werden) [SSS+21]. Wie bei Astrée werden hier in der Domäne der Seiteneffekte globale Werte miteinander verschmolzen Die Ordnung, die Sperren vorgeben, kann aber modelliert werden. Auch GOBLINT ist in der Lage, das Scheduling-Verfahren von OSEK zu interpretieren und benutzt ebenso wie Astrée die Interpretation des PCP, um bessere Rückschlüsse auf Laufzeitfehler zu finden [SSV+11]. Es konstruiert damit auch für den Einkernfall ein Abhängigkeitssystem an Aktivitäten für OSEK, ähnlich wie es der gleich vorgestellte RTSC tut.

## 4.4 RTSC

*Umwandlung von ereignis- in zeitgetriebene RTSs*

Der *Real-Time Systems Compiler (RTSC)* wurde initial von Fabian Scheler geschrieben, um ein ereignisgetriebenes RTS automatisch zu einem zeitgetriebenen RTS umzuformen [Sch11A5]. Der RTSC hat dabei initial Anwendungen für einen eigenen Echtzeitstandard analysiert, ist dann aber zur Unterstützung des ereignisgesteuerten OSEK und einer entsprechenden Konvertierung auf OSEKTime (die zugehörige zeitgesteuerte Variante) erweitert worden. Die Idee hierbei ist, dass bestimmte (vorher bekannte) Systemaufrufe Quellen und Senken für zeitliche funktionsübergreifende Abhängigkeiten darstellen. Der Systemaufruf

ActivateTask z. B. stellt eine Quelle einer zeitlichen Abhängigkeit zum aktivierten Task mit der entsprechenden Senke als dessen erster Anweisung dar. Die Analyse ist dabei nicht kontextsensitiv, versucht also im Speziellen auch keine Scheduling-Entscheidung vorauszuberechnen, die auf dem aktuellen Kontext basiert. Eine Kante sagt dementsprechend nur aus, dass eine Senke der Quelle *irgendwann* und nicht zeitlich unmittelbar folgt.

Der Ansatz geht damit in eine ähnliche Richtung wie der einschränkungsbasierte Ansatz von GOBLINT, nur dass er statt serialisierten kritischen Regionen auf Mehrkernsystemen versucht, seriell ausgeführte Kontrollflussregionen zu sortieren, die von der Betriebssystemlogik vorgegeben werden. Auch GOBLINT ist in der Lage, diese Sortierung in gewissem Maß herauszufinden, verzichtet dabei aber im Gegensatz zum RTSC auf jegliche Konzentration auf die reale Zeit.

*Atomare Basisblöcke*

Kern der Analyse des RTSC ist die Transformation der Eingangsanwendung in *Atomare Basisblöcke (Atomic Basic Blocks, ABBs)*, die – genau wie die schon angesprochenen Basisblöcke Anweisungen auf Basis von Sprüngen im Binärcode gruppieren – nun versuchen, Basisblöcke auf Basis von Sprüngen in der Ablaufplanung des Betriebssystems zu gruppieren[Sch11A4]. ABBs machen somit den betriebssystemspezifischen Kontrollfluss sichtbar und stellen in Bezug auf die Methoden der abstrakten Interpretation eine selektive Analyse dar, da (irrelevante) Einzelanweisungen damit übersprungen werden können. Dietrich hat dieses Konzept für seine Dissertation in adaptierter Form übernommen [Die19A3.3] (siehe den nachfolgenden Abschnitt 4.5)[20]. Ich baue meine Analysen ebenfalls (in erneut adaptierter Form) darauf auf.

Die folgende Eigenschaft bildet dabei die Basis eines ABBs und wurde auch von Dietrich und mir so übernommen:

> **Definition 14: Atomarer Basisblock – Kerneigenschaften.**
> Ein ABB besteht aus einer Menge an Basisblöcken und wird dabei immer durch genau einen Basisblock betreten und durch genau einen Basisblock verlassen (*single-entry–single-exit (SESE)* Region). Die Kanten zwischen ABBs entsprechen dabei denjenigen zwischen Ausgangsbasisblöcken der Vorgänger-ABBs und Eingangsbasisblock des Nachfolger-ABBs.

*ABBs im RTSC*

Scheler verknüpft in der Definition zusätzlich noch ABBs mit *ABB-Endpunkten*, das sind die schon beschriebenen Quellen und Senken. Diese stellen dabei jeweils die erste (im Fall einer Senke) oder letzte Instruktion des ABB dar (im Fall einer Quelle). Sollte diese Instruktion innerhalb eines Basisblocks liegen, wird dieser aufgeteilt. Die Erstellung der ABBs erfolgt dabei musterbasiert, so werden zuerst Quellen und Senken im aus Basisblöcken bestehenden Kontrollflussgraphen identifiziert. Zwischen diesen werden anschließend eine definierte Anzahl von Mustern erkannt und zusammengefasst. Dieser Schritt wird solange

[20] Batista Ribeiro nutzt für seine entwickelten COFIE-Ausdrücke ein ebenfalls sehr ähnliches Konzept [BB19].

iteriert, bis die vordefinierten Muster nicht mehr auf den Graphen passen. Dieser Ansatz ist allerdings nicht vollständig, insbesondere kann er keine Basisblöcke mit irreduzibler Struktur (z. B. Schleifen mit uneindeutigem Kopf) zusammenfassen [Die19A3.3.2].

*Systemmodell des RTSC*

Für die eigentliche Transformation definiert Scheler außerdem ein Modell für Echtzeitbetriebssysteme, auf denen generisch Regeln für die Umformung definiert sind, mit einer anschließenden Implementierung am Beispiel von OSEK/AUTOSAR [Sch11A4.2]. Er setzt dabei (zusammengefasst) drei Konzepte voraus:

1. Ereignisse: Diese beschreiben Momente in der realen Zeit, die eine Aktion nach sich ziehen. An Ereignisse ist das Zeitverhalten des Systems gekettet (sie können z. B. eine Periode haben).
2. Aufgaben: Diese kapseln die Aktionen, die das RTS nach einem Ereignis ausführt.
3. Termine: Diese entsprechen den bereits vorgestellten Terminen, also der Zeitspanne, die Aufgaben zur Ausführung benötigen dürfen.

*Einordnung*

Auch ich werde in dieser Arbeit ein Systemmodell beschreiben. Das im RTSC beschriebene Systemmodell ist allerdings stark darauf ausgelegt, die zeitlichen Abhängigkeiten innerhalb eines Echtzeitsystems zu modellieren, während für die Analysen dieser Arbeit die detaillierten logischen Abhängigkeiten im Vordergrund stehen. Insbesondere fehlen dem Modell des RTSC Informationen über Konzepte, die keine Aufgaben sind, wie Ressourcen, Queues und Semaphoren. Diese werden ausschließlich über Kanten zwischen ABBs abgebildet, was aber für einige Optimierungen unzureichend ist. Mein Modell ist daher nicht auf dem Modell des RTSC aufgebaut, sondern erweitert das im nächsten Abschnitt beschriebene Modell innerhalb von dOSEK.

*Erweiterung um Mehrkernunterstützung*

2016 wurde der RTSC von Franzmann et al. um Mehrkernunterstützung erweitert [FKU+16]. Er analysiert dabei aber nicht Systeme auf Mehrkernbasis, sondern generiert basierend auf der (vorhandenen) Systembeschreibung ein möglichst ideales System für eine Zielmehrkernsystem. Raffeck et al. hat die Arbeit noch weitergeführt und hat den RTSC um Task-Migrationen erweitert, geeignete Zeitpunkte, um Tasks von einem auf den anderen Kern zu umgehen [RUS19]. Passend gesetzte Migrationspunkte können dabei ein vormals unplanbares System doch noch planbar machen. Die mit diesen Ansätzen gelösten Probleme bewegen sich allerdings trotz der Verwendung von Mehrkernsystemen in einer anderen Domäne, da für meine Analysen bereits eine fertige Zuordnung auf mehrere Kerne besteht, die bei der Analyse berücksichtigt werden muss.

## 4.5 dOSEK

dOSEK ist ein Framework für betriebssystemgewahre Analyse und Generierung, das für das „dependable OSEK“-Projekt geschrieben wurde [HLD+15a] Es wurde von Christian

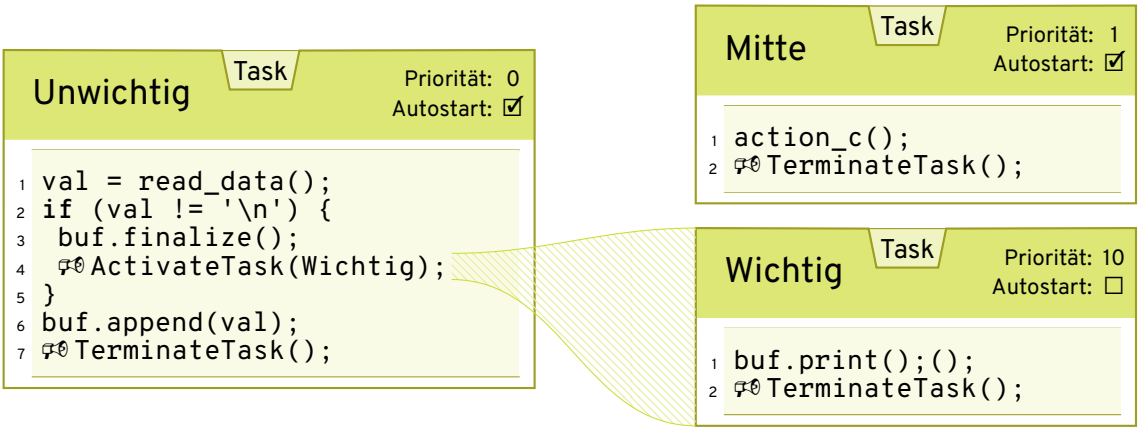



**Abbildung 4.1** Ein RTS mit Optimierungspotenzial: Der ActivateTask wird immer dazu führen, dass der Task *Wichtig* eingeplant wird. (adaptiert aus [Die19A3.4])

Dietrich um feingranulare Interaktionsanalysen erweitert, die ich in meiner Forschung benutze und erweitere und deswegen hier detailliert vorstelle [Die19].

*Interaktionsanalyse*

Ziel dieser Analysen ist es, Wissen für systemaufrufspezifische Optimierung bereitzustellen, d. h. zu ermöglichen, dass der konkrete Systemaufruf (nicht nur die Systemfunktion) optimiert wird. Ein Beispiel dafür zeigt Abbildung 4.1. In der dargestellten Anwendung kann bereits statisch gezeigt werden, dass der Systemaufruf zu ActivateTask *immer* den Task *Wichtig* aktiviert und dieser Task auch nur an dieser Stelle lauffähig wird. Diese Scheduling-Entscheidung kann somit fest codiert werden, sodass der Systemaufruf zu einem normalen Funktionsaufruf wird oder der ganze Task *Wichtig* sogar in den Task *Unwichtig* eingebettet wird. Ich werde auf diesen Analysen aufbauen und sie in einen theoretischen Kontext betten, weswegen ich sie hier detaillierter vorstellen will.

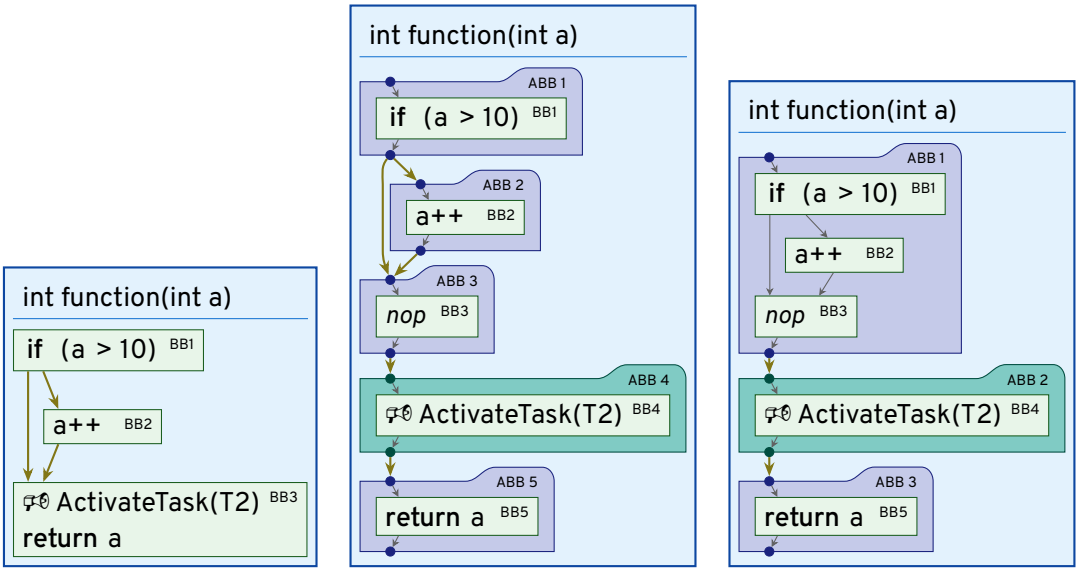
*globaler Kontrollfluss*

Dietrich hat für die betriebssystemgewahre Generierung das Konzept des *globalen Kontrollflusses* entwickelt. Dieser bildet im Gegensatz zum lokalen oder interprozeduralen Kontrollfluss zusätzlich die möglichen Fädenwechsel ab. Eine Kante des globalen Kontrollflusses trifft im Gegensatz zu den Kanten zwischen ABBs bei Scheler die Aussage, dass Quelle und Ziel der Kante *direkt hintereinander* ausgeführt werden können und weiterhin, dass genau *alle möglichen* Nachfolger durch Kanten mit dem Vorgänger verbunden sind.

*Atomare Basisblöcke*

Um den globalen Kontrollfluss zu erzeugen, hat Dietrich zwei Analysen entwickelt, die *System-State Enumeration (SSE)* und den *System-State Flow (SSF)*. Erstere ist langsamer, aber präziser. Beide operieren auf den schon vorgestellten ABBs, die aber adaptiert sind. Konkret kommt zu der bereits bestehenden Definition eine weitere Eigenschaft hinzu [Die19]:

> **Definition 15: Atomarer Basisblock – Typen.**
> ABBs sind entweder vom Typ COMPUTATION oder führen, direkt oder indirekt, zu einem einzelnen Systemaufruf (aus dem Englischen übersetzt).




**a)** Basisblöcke **b)** Aufgeteilt nach Typ **c)** Zusammengefasst

**Abbildung 4.2** Die ABB-Konstruktion nach Dietrich [Die19]. a) zeigt die Basisblöcke wie vom Übersetzer gegeben. b) zeigt den ersten Transformationsschritt, in der die Basisblöcke bei systemrelevanten Funktionen aufgeteilt werden und eins zu eins auf ABBs abgebildet werden (der ABB-Typ ist farblich markiert: COMPUTATION und SYSCALL). c) zeigt die mit dem Dominator-Verfahren zusammengefassten ABBs.

Dietrich hat dafür ein weiteres Konzept eingeführt:

**Definition 16: Systemrelevante Funktion.**
Eine Funktion ist dann systemrelevant, wenn sie, direkt oder indirekt, eine Systemfunktion aufruft.

Falls der ABB einen direkten Systemaufruf beinhaltet, ist er vom Typ SYSCALL. Falls der ABB eine systemrelevante Funktion aufruft, ist er vom Typ CALL. Weiterhin befindet sich zwischen zwei ABBs des Typs SYSCALL oder CALL mindestens ein ABB des Typs COMPUTATION [DHL15].

*ABB-Konstruktion mit Dominatoren*

Die Konstruktion der ABBs basiert im Unterschied zu Scheler auf Dominatoren, die Methode, die ich auch in dieser Arbeit verwende. Abbildung 4.2 zeigt alle Schritte an einem Beispiel. Zuerst extrahiert Dietrich den interprozeduralen Kontrollfluss in Form von Basisblöcken (implementiert mithilfe des Compiler-Frameworks LLVM, siehe Abbildung 4.2a). Damit dies möglich ist, setzt er einen statisch extrahierbaren interprozeduralen Kontrollfluss voraus, also einen Kontrollfluss, bei dem sämtliche Funktionsaufrufe statisch bestimmbar sind. Er erreicht dies durch ein Verbot von Funktionszeigern. In diesem markiert er systemrelevante Funktionen und teilt anschließend die Basisblöcke derart auf, dass, wenn immer sie einen Systemaufruf oder einen Aufruf zu einer systemrelevanten Funktion enthalten, sie an genau dieser Funktion geteilt werden. Basisblöcke, die einen Systemaufruf oder Aufruf

zu einer systemrelevanten Funktion enthalten, werden anschließend direkt in einen ABB eingebettet, womit die zweite Hälfte der obigen Definition bereits erfüllt ist (4.2b). Die bereits genannten Dominatoren (einem Konzept aus der Übersetzertechnik) verwendet Dietrich, um die ABBs vom Typ COMPUTATION zu konstruieren (4.2c):

> **Definition 17: Dominator, Postdominator [Pro59].**
> Ein Knoten $n$ in einem Graph ist ein *Dominator* von $m$ in Bezug auf einen Startknoten $s$, wenn gilt, dass alle Pfade von $s$ zu $m$ durch $n$ laufen. Ein Knoten $n$ in einem Graph ist ein *Postdominator* von $m$ in Bezug auf einen Endknoten $e$, wenn gilt, dass alle Pfade von $m$ zu $e$ durch $n$ laufen.

Ein COMPUTATION-ABB ist dann eine Menge an Basisblöcken mit folgenden Eigenschaften (nach [Sha80], der dies für allgemeine SESE-Regionen definiert hat):

1. Der Eintrittsknoten muss den Austrittsknoten dominieren.
2. Der Austrittsknoten muss den Eintrittsknoten postdominieren.
3. Es darf in der Menge bis auf die Kanten von Eintritts- und Austrittsknoten keine Kanten zu Knoten außerhalb der Menge geben.

Zur eigentlichen ABB-Konstruktion benutzt Dietrich ein iteratives Verfahren, das mit einem einzelnen Basisblock anfängt und die Mengen der Basisblöcke in einem ABB solange vergrößert, bis die Bedingungen nicht mehr erfüllt sind. Dieses Verfahren findet – im Gegensatz zu Schelers Ansatz – auch Basisblöcke mit irreduzibler Struktur.

Dietrich verweist weiterhin noch auf ein Verfahren zur Berechnung von SESE-Regionen mit konstanter Laufzeit vor [JPP94]. Dies wurde von Vanhatalo et al. weitergeführt und auf dreifach zusammenhängende Graphen abgebildet [VVK08]. Der Kontrollfluss in ABB-Form bildet die Voraussetzung für die folgenden Analysen.

### 4.5.1 Die *System-State Enumeration (SSE)*

*Abstract System State*

Die *System-State Enumeration (SSE)* ist eine abstrakte Interpretation, die als abstrakten Zustand aber ausschließlich die für das unterliegende Betriebssystem relevanten Daten mitführt, den *Abstract System State (AbSS)* [Die19A3.4.2,DHL15]. Abbildung 4.3 zeigt einen solchen Zustand. Dort wird der aktuelle ABB gespeichert (als abstrakte Repräsentation des „Program Counter“), ob Unterbrechungen angestellt sind (als Abstraktion des für das RTOS relevanten Hardware-Zustands), sowie eine Liste aller Tasks zusammen mit für deren abstrakte Ausführung wichtigen Daten. Die Analyse ist dabei auf das OSEK-Betriebssystem zugeschnitten, geht aber allgemein von Betriebssystemen mit statisch konfigurierten Instanzen aus.

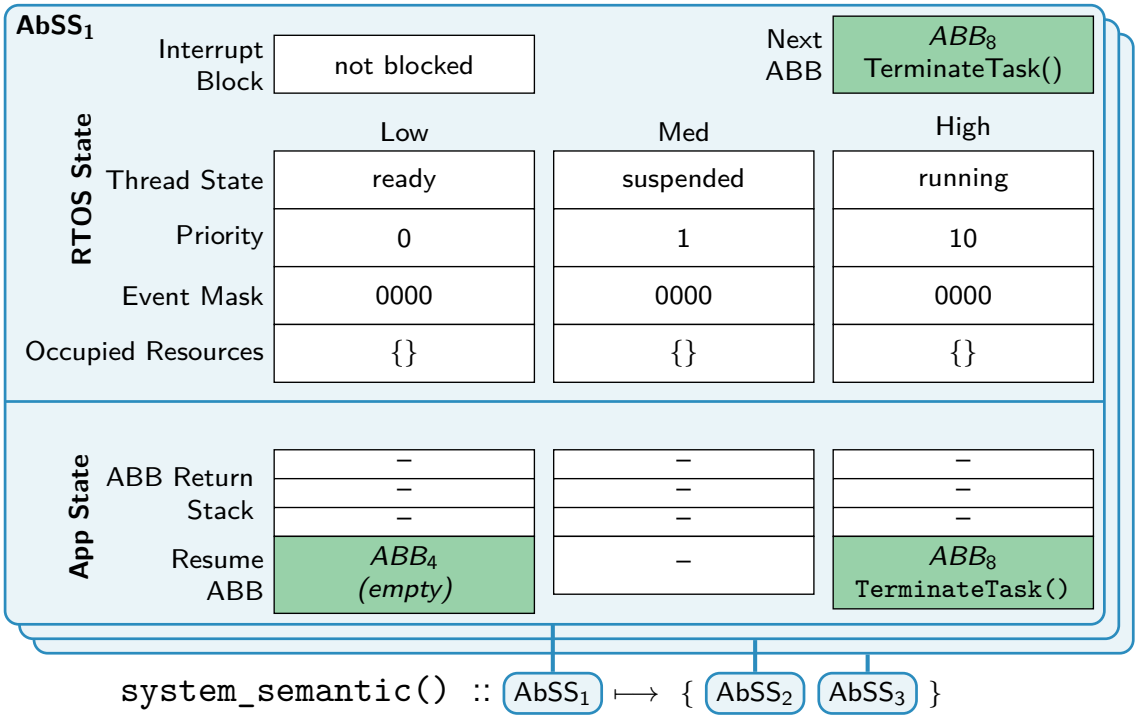



**Abbildung 4.3** Der *Abstract System State (AbSS)*: Eine abstrakte Darstellung der betriebssystemrelevanten Daten (aus [Die19A3.4])

Auf den AbSSs ist eine abstrakte Transition möglich, die einen AbSSs auf eine Menge an Folgezuständen überführt: *Transitionsfunktion*

$$(\text{AbSS}_{g+1,0}, \ldots, \text{AbSS}_{g+1,n}) = \text{system\_semantic}(\text{AbSS}_g)$$

$$g = \text{Generation}$$

Die Transitionsfunktion arbeitet dabei auf dem im AbSSs gegebenen ABB:

- Bei einem ABB vom Typ COMPUTATION ist das Betriebssystem nicht involviert, dementsprechend werden als Folgezustände Zustände erzeugt, die alle jeweiligen Nachfolge-ABBs beinhalten. Die SSE geht allerdings davon aus, dass logisch alle externen Unterbrechungen (nur) in COMPUTATION-ABBs stattfinden können und löst deswegen zusätzlich virtuell alle im aktuellen Zustand möglichen Unterbrechungen aus.
- Bei einem ABB vom Typ CALL wird dem Aufruf gefolgt.
- Bei einem ABB vom Typ SYSCALL wird der Aufruf abstrakt interpretiert: Die SSE berechnet dabei den neuen virtuellen Zustand durch die Abbildung der realen Auswirkungen

des Systemaufrufs auf die abstrakte Domäne. Konkret beinhaltet das die Interpretation des Systemaufrufs selbst (z. B. wird bei einem ☞ ActivateTask der abstrakte Laufzeit-Zustand des jeweiligen zu aktivierenden Tasks auf „Ready“ gesetzt), sowie einen anschließenden Aufruf eines abstrakten Schedulers.

*Static State-Transition Graph*

Die AbSSs bilden anschließend die Knoten eines Graphen, des *Static State-Transition Graphs (SSTGs)*, dessen Kanten durch deren Transition zustande kommen. Als Besonderheit der SSE werden gleiche Zustände wieder verschmolzen. Wird also z. B. innerhalb einer Schleife ☞ ActivateTask aufgerufen, ist der abstrakte Zustand beim zweiten Durchlauf identisch, wird mit dem bereits vorhandenen verschmolzen und die Analyse an dieser Stelle nicht mehr fortgesetzt. Um aus dem SSTG anschließend den globalen Kontrollflussgraphen zu konstruieren, gruppiert Dietrich die abstrakten Zustände anhand des beinhaltenden ABBs und kann so alle möglichen Folge-ABBs bestimmen. Damit die abstrakte Interpretation der Systemaufrufe gelingt, müssen deren Parameter statisch erkennbar sein, was eine vorhergehende Wertanalyse erfordert. Bei der Implementierung in dOSEK passiert dies über die Einschränkung der Parameter auf globale Konstanten.

Der Hauptnachteil der Analyse ist ihre exponentielle Laufzeit. Durch den mehrmaligen Durchlauf durch sämtliche Kontrollstrukturen, solange bis alle möglichen abstrakten Zustände gefunden sind, wächst die Laufzeit exponentiell in Bezug auf die Menge an Anweisungen. Sie wird durch die Verwendung von ABBs verbessert, die zwar irrelevanten Kontrollfluss vor der Analyse verschmelzen, aber das grundsätzliche Problem von Schleifen oder Verzweigungen in analyserelevantem Kontrollfluss nicht verhindern. Zur Umgehung dieses Nachteils hat Dietrich eine weitere Analyse konzipiert, den SSF, den ich jetzt vorstelle.

### 4.5.2 Der *System-State Flow (SSF)*

Der *System-State Flow (SSF)* konstruiert wie die SSE einen globalen Kontrollfluss, überspringt dabei aber die Konstruktion des SSTG. Dies bewirkt eine zur Anzahl von ABBs polynomielle Laufzeit, verringert aber die Präzision. Konkret klassifiziert der SSF mehr Transitionen im globalen Kontrollfluss als die SSE [Die19A3.5.2].

*Widening: Unpräzise Zustände*

Der SSF arbeitet dazu auf unpräzisen Zuständen. In diesen ist es möglich, dass Felder statt eines konkreten Wertes den Wert „unbekannt“ erhalten. Kann z. B. ein Systemaufruf über zwei Pfade erreicht werden, bei denen ein aktuell nicht laufender Task einmal lauffähig und einmal inaktiv ist, dann bekommt dieser den Wert „unbekannt“ für seinen Laufzeitzustand. Könnte der Task einige Zustände später potentiell abstrakt eingeplant werden, berücksichtigt der Scheduler sowohl die Möglichkeit, dass er nicht lauffähig sein könnte und emittiert einen entsprechenden Zustand, als auch die Möglichkeit, dass er nicht lauffähig ist mit einem zweiten dementsprechenden Zustand. Die Analyse ist damit nicht mehr pfadsensitiv, da sie die Information verliert, woher der Laufzeitzustand kam (ein klassisches Problem der statischen Analyse [VV09]).

Der Vorteil dieser Methode ist die Möglichkeit, den globalen Kontrollflussgraph direkt aufzubauen, indem an jedem ABB die Transition auf einer Menge an möglichen Zuständen stattfindet. Dietrich setzt hier aber stärkere Einschränkungen voraus:

- Der ABB muss eindeutig einem Task und einem Aufrufpfad zuordenbar sein (insbesondere darf die gleiche Funktion nicht von zwei Tasks aufgerufen werden). Dies ist nachträglich erreichbar, indem man alle betroffenen Funktionen vervielfältigt und in ihre aufrufende Funktion einbettet.
- Die dynamische Priorität eines ABB muss unabhängig vom Kontext sein (auch dies ist durch Vervielfachen der ABBs beim nicht eindeutigen Benutzen von Ressourcen möglich).

Die verbesserte Laufzeit des SSF im Vergleich zur SSE wird erkauft durch eine geringere Präzision und stärkere Einschränkungen an die Anwendung.

## 4.6 Diskussion

Mit Astrée und GOBLINT existieren zwei betriebssystemgewahre Analyseprogramme, die aber beide nicht geeignet sind, meine Forschungsfragen zu beantworten. Beide Analysen unterstützen zwar Mehrkernsysteme und mehrere Betriebssysteme, legen den Fokus aber auf den Nachweis der Absenz von Laufzeitfehlern. Konkret analysieren beide Programme z. B. OSEK-Programme wie ein Mehrkernsystem mit stärkeren Einschränkungen, detektieren dabei aber nicht den konkret möglichen Ablaufplan (wie die SSE) oder vergleichen Betriebssystemschnittstellen miteinander. Insbesondere haben beide Programme nicht zum Ziel, eine spätere Optimierung zu ermöglichen, untersuchen darum grundsätzlich andere Programmeigenschaften und führen einen anderen Zustand (andere Daten) mit.

Der RTSC und dOSEK gehen von der Zielsetzung in eine ganz andere Richtung (sie sind darauf ausgelegt, ein System zu optimieren bzw. zu transformieren), beantworten aber ebenso meine Forschungsfragen nicht. Beide Systeme arbeiten ausschließlich auf dem OSEK-Betriebssystemstandard und sind speziell darauf zugeschnitten. Weiterhin sind beide Analysen dadurch implizit auf Einkernsysteme beschränkt.

Speziell bei der SSE ist die Erweiterung auf dynamische Systeme konzeptuell nicht ohne weiteres möglich, da ansonsten ihre Terminierung nicht mehr gewährleistet ist. Dies ist die direkte Folge einer theoretischen Einordnung der Analysen, die ich im Folgenden vornehmen werde.

### 4.6.1 Theoretische Einordnung der SSE und SSF

Ich werde in diesem Abschnitt erstmals die SSE und SSF mit den Techniken der abstrakten Interpretation verbinden, was der Analyse einen theoretischen Unterbau gibt und damit

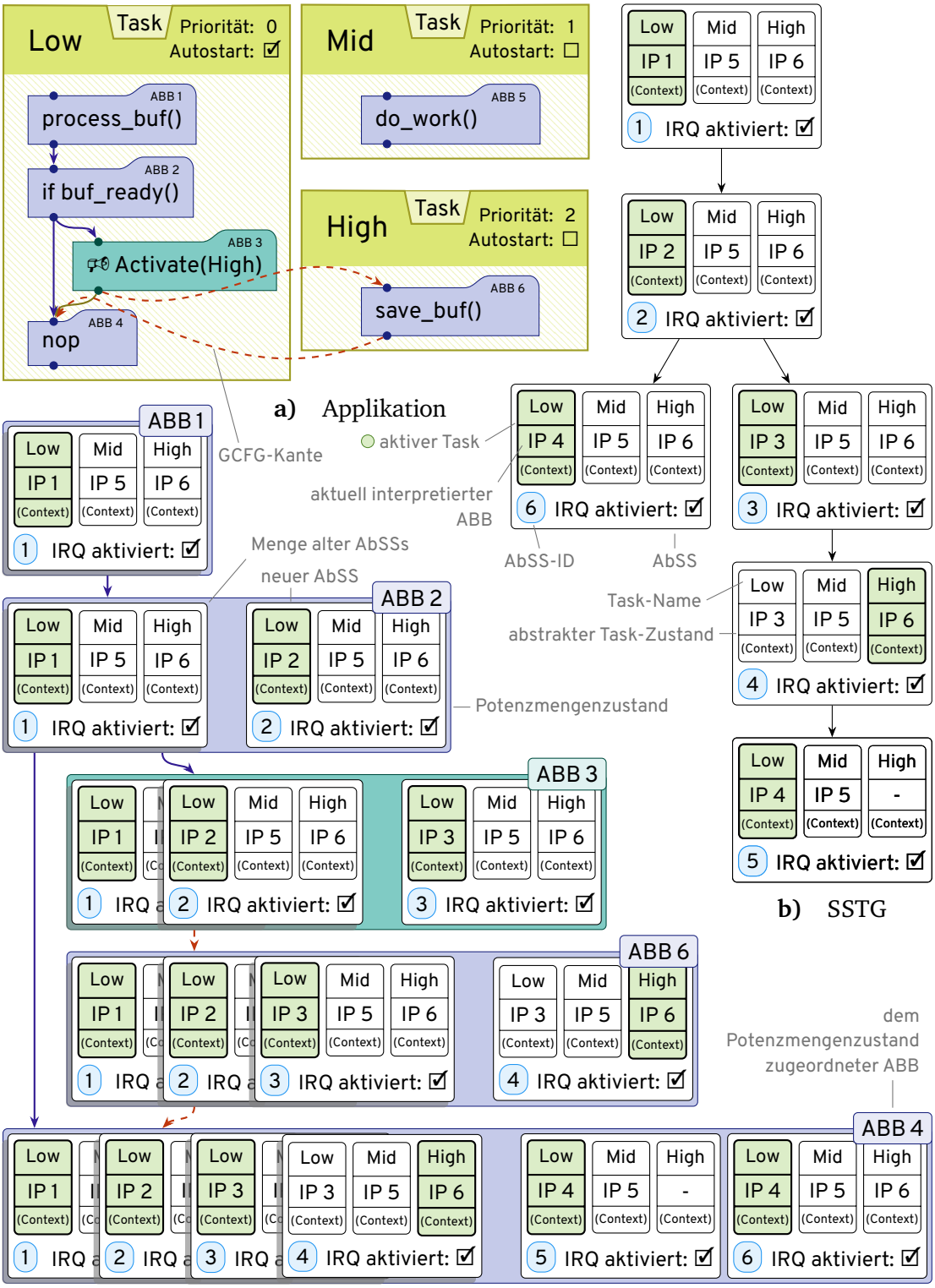



**Abbildung 4.4** Die Abbildung des SSTG auf eine Menge aus Teilmengen. a) zeigt die Anwendungs-ABBs. b) zeigt den SSTG. c) zeigt dessen Rückabbildung auf die ABBs.

ihre Stärken und Schwächen zeigt. Dietrich hat in seiner Dissertation zwar den Zusammenhang zur abstrakten Interpretation hergestellt [Die19A3.4], aber dabei keine theoretische Einordnung vorgenommen oder eine detaillierte Betrachtung angestellt. Ich werde dies hier nachholen, auch um damit die Grundlagen für meine 3. Forschungsfrage zu legen. Die SSE ähnelt einer klassischen abstrakten Interpretation ohne Widening-Operator. Sie durchläuft deshalb Schleifen mehrfach. Die AbSSs sind außerdem keine vollständigen Verbände, insbesondere ist auf ihnen nur ein Gleichheits-, aber kein Vergleichsoperator definiert. Für eine formale Abbildung auf der SSE auf die Theorie der abstrakten Interpretation müssen folgende Bedingungen erfüllt sein:

1. Der abstrakte Zustand muss ein vollständiger Verband sein.
2. Dieser muss überdies entweder endlich oder ein Widening-Operator definiert sein.
3. Die Transitionsfunktion muss monoton sein.

*Potenzmengenverband*

Die erste Bedingung ist erfüllbar, wenn man nicht den bereits definierten AbSS als Zustand ansieht, den die SSE direkt verarbeitet, sondern die SSE so auffasst, dass sie eine Menge an AbSSs als abstrakten Systemzustand verarbeitet (ich nenne diese Menge „Potenzmengenzustand“). Damit ergibt sich automatisch ein vollständiger Verband, da die Menge der Teilmengen von beliebigen Entitäten mit der Teilmengenrelation als Ordnung einen vollständigen Verband darstellt (einen Potenzmengen-Verband) [SWH10,MS18]. Der Transitionsoperator muss dann so angepasst werden, dass er die alten Zustände „mitschleift“:

*Transitionsfunktion*

```
1  def transition_new(state)
2      new_state = state.copy()
3      for abss in state:
4          new_state.append(transition_old(abss))
```

Konkret wird nun jeder AbSS aus der Menge des Potenzmengenzustands mit der bereits definierten Transitionsfunktion transformiert. Die Menge dieser Ausgangszustände definiert den neuen Potenzmengenzustand. Abbildung 4.4 verdeutlicht dies für die (vereinfachte) RTA aus Abbildung 4.1. In Teil b) ist der SSTG für den ABB-Graphen aus a) dargestellt. Bei diesem Graphen repräsentieren die Knoten AbSSs und die Kanten Transitionen zwischen den einzelnen AbSSs. In Teil c) sind die AbSSs (wie bei dem SSF und der Konstruktion des globalen Kontrollflussgraphens) rückabgebildet auf den jeweiligen ABB, den sie verarbeiten, und bilden in ihrer Gesamtmenge den abstrakten Zustand, der für die abstrakte Interpretation verwendet wird. Durch diese Technik ist auch Bedingung 3 erfüllt, da jede Transition ausschließlich Zustände hinzufügt, sodass der kombinierte Endzustand alle AbSSs des SSTG enthält.

*Statische Betriebssysteme*

Die zweite Bedingung erfüllt die SSE durch ihre Einschränkung auf statische Betriebssysteme. Da bei diesen Systemen alle Instanzen schon zu Analysebeginn bekannt sind und – im Falle von OSEK OS – die Zustände dieser auch noch abzählbar sind (es gibt z. B. genau

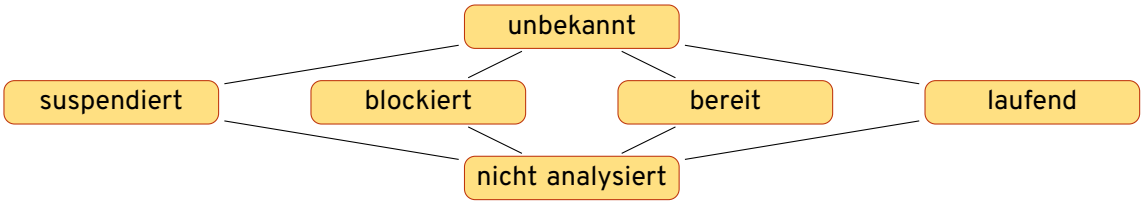



**Abbildung 4.5** Der vollständige Verband des abstrakten Taskzustands.

vier Taskzustände: laufend, bereit, wartend, beendet)[21], sind die Potenzmengenzustände endlich und damit auch die Höhe des vollständigen Verbands finit.

*Widening im SSF*

Der SSF ordnet im Gegensatz zur SSE interessanterweise die AbSSs konzeptuell schon direkt den ABBs zu, definiert aber zusätzlich noch einen Widening-Operator. Dieser verschmilzt die Liste an präzisen Zuständen, die bei der SSE pro ABB existieren, zu einem unpräzisen Zustand. Durch den Wegfall der Potenzmengenzustände müssen wir den vollständigen Verband in diesem Fall direkt auf den AbSS definieren. Dazu machen wir uns den folgenden Satz zunutze:

> **Satz 1: Produktverband [MS18].**
> Das Produkt aus vollständigen Verbänden $V_1, V_2, \ldots, V_n$ bildet wieder einen vollständigen Verband $P = V_1 \times V_2 \times \ldots \times V_n$ mit einer komponentenweisen Ordnungsrelation.

Können wir also jeden Teil (aufgeteilt nach der jeweiligen Instanz) des AbSS als vollständigen Verband auffassen, bildet der gesamte AbSS ebenso einen. Durch den Gleichheits-Operator ist eine „Sortierung“ auf gleicher Ebene bereits vorgegeben. Durch das Widening in den Wert „unbekannt“ existiert zudem ein Top-Element. Das Bottom-Element wird durch ein künstliches Element „nicht analysiert“ vorgegeben. Beispielsweise stellt Abbildung 4.5 das Hasse-Diagramm des vollständigen Verbands für die Task-Zustände dar.

*Betriebssystemsemantik*

Für den Beweis der Korrektheit der SSE und des SSF muss weiterhin die Semantik des Betriebssystems, hier OSEK, formal definiert werden, sodass die Korrektheit der abstrakten Abbildung für jede einzelne Instruktion gezeigt werden kann, was in diesem Rahmen aber zu weit führen würde. Zusammengefasst sind also sowohl die SSE als auch der SSF abstrakte Interpretationen, sodass die Terminierung und Korrektheit mit der bereits vorhanden Theorie formal bewiesen werden kann.

### 4.6.2 Atomare Basisblöcke

Sowohl der RTSC als auch die SSE und SSF setzen Atomare Basisblöcke voraus. Im RTSC sind sie die zentrale Datenstruktur, mithilfe derer die Interaktionen zwischen Instanzen von

[21] Genaugenommen sind in der realen Welt durch Beschränkungen der Hardware alle Zustände abzählbar, z. B. gibt es für eine 32-Bit-Integer „nur“ 4 294 967 295 mögliche Werte. Diese führen damit zwar letztlich auch zu einer Terminierung des Algorithmus, aber zum Preis einer inpraktikablen Laufzeit.

Betriebssystemobjekten gespeichert werden und ohne die die Analyse nicht funktionieren würde. Bei der SSE sind die ABBs hingegen nur ein Mittel, um die Analyse auf die Teile zu beschränken, in denen sich der abstrakte Zustand auf die gleiche Art und Weise ändert. Letzteres ist ein bereits bekanntes Mittel bei abstrakter Interpretation, um die Analysezeit zu verringern und heißt *selektive Analyse* („sparse analysis“). Die SSE würde aber auch auf einzelnen Instruktionen oder auf Basisblöcken, bei denen System- und Funktionsaufrufe singulär sind, funktionieren. Essentiell für die Analyse ist die korrekte Interpretation aller Systemaufrufe (Analyse der SYSCALL-ABBs) unter Beibehaltung des korrekten Aufrufkontextes (Analyse der CALL-ABBs) und Kontrollstruktur (Analyse der COMPUTATION-ABBs).

### 4.6.3 Analyse-Restriktionen

```
1  class MutexLocker {
2      SemaphoreHandle_t mtx;
3
4    public:
5      MutexLocker(SemaphoreHandle_t mtx) mtx(mtx) {☞xSemaphoreTake(mtx);}
6      ~MutexLocker() {☞xSemaphoreGive(mtx);}
7  };
8
9  class Wrapper {
10     SemaphoreHandle_t xGPSDataMutex;
11
12   public:
13     Wrapper() {xGPSDataMutex = ☞xSemaphoreCreateMutex();}
14     int do_critical(int a) {MutexLocker g(xGPSDataMutex); [...] }
15 };
16
17 int main() {
18     Wrapper w;
19     w.do_critical();
20 }
```

**Codeblock 4.1** Die Verwendung des Semaphors *xGPSDataMutex* im GPSLogger (siehe Abschnitt 6.4.4, schematisch). Der Semaphor (verwendet als Mutex) wird in einigen Funktionen, die kritische Regionen implementieren, automatisch genommen und freigegeben (in der Grafik `do_critical()`). Dazu wird eine Klasse `MutexLocker` eingeführt, die den Code per *Resource acquisition is initialization (RAII)* schützt [Str13A13.3]. Eine Wertanalyse müsste den Speicherort des Semaphors bis in die Klasse `MutexLocker` nachvollziehen.

*Vergleichbarkeit des RTSC*

Der RTSC ist durch die andere Zielsetzung deutlich weniger präzise als die SSE. Insbesondere bildet er Interaktionen zwischen Instanzen nur soweit ab, wie sie für die Abhängigkeitsmodellierung bezüglich des Schedulings notwendig sind und schließt damit feingranulare

Optimierungen auf Interaktionsebene aus. Auch das Modell des RTSC ist darauf zugeschnitten, die zeitlichen Eigenschaften des Systems zu fassen (z. B. wären Optimierungen des Speicherbedarfs damit nicht möglich). Wie dOSEK geht auch der RTSC von statisch konfigurierten RTOSs aus. Ich werde den RTSC für die in dieser Arbeit beschriebenen Methoden außer Acht lassen.

*Einschränkung auf Einkernsysteme*

Sowohl der RTSC und dOSEK sind auf Einkernsysteme beschränkt. Dieses Problem hat auch Dietrich aufgegriffen und macht den Vorschlag, die SSE mehrfach pro Kern auszuführen. Das hat allerdings den Nachtteil, dass sämtliche Interaktionen zwischen den Kernen verloren gehen. Er sieht aber generell in den Bereich weiteren Forschungsbedarf [Die19A3.6]. Genau dieses Punktes habe ich mich angenommen und stelle im Kapitel 7 mit der MultiSSE eine Analyse vor, die in der Lage ist, mit gleichbleibender Präzision auch Mehrkernsysteme und deren Interaktion zu analysieren.

*Einschränkung auf statische Systeme*

Bei der Abbildung der SSE und des SSF auf die Theorie der abstrakten Interpretation wird eine weitere Restriktion der Verfahren unmittelbar deutlich: Sie funktionieren nur auf statisch konfigurierten Betriebssystemen. Um dynamische RTOSs zu unterstützen, wäre eine Art Widening-Operator über die Instanzmenge notwendig. Ich werde mich auch dieses Problems annehmen und stelle erst eine Transformation des AbSS und der SSE vor, damit der Algorithmus betriebssystemspezifischen Code vom Algorithmus entkoppelt. Dieser wird dann Teil eines echtzeitsystemübergreifenden Betriebssystemmodells (Kapitel 8). Ich gehe weiterhin auf Probleme auf dynamischen Systemen ein und stelle einen Algorithmus vor, der diese angeht (Kapitel 6).

*Einschränkungen von SSE und SSF*

Dietrich spricht für die SSE und den SSF einige weitere Restriktionen an [Die19A3.3,3.4]:

**Der Kontrollfluss (der Aufrufgraph) muss im Vorhinein bestimmbar sein:** Er verbietet dazu in der Implementierung Funktionszeiger. Das hat sich für diese Arbeit für einige Echtweltsysteme (wie den GPSLogger oder den LibrePilot) als unrealistisch herausgestellt. Der GPSLogger verwendet z. B. C++-Vererbung, bei der der dynamische Dispatch über Funktionszeiger realisiert wird (automatisch durch den Compiler).

**Alle Argumente von Systemaufrufen müssen statisch bestimmbar sein:** Im einfachsten Fall heißt das, dass für Systemaufrufargumente nur globale Konstanten verwendet werden dürfen. Auch diese Annahme ist bei einigen Echtweltsystemen unhaltbar. Quellcode 4.1 zeigt vereinfacht, wie der eben schon besprochene GPSLogger einen gemeinsamen Semaphor nutzt. Die Instanz *mtx* ist in diesem Fall zwar eindeutig statisch bestimmbar, allerdings nur mit einer umfangreichen statischen Wertanalyse.

Die beiden letztgenannten Nachteile werde ich in meiner Implementierung, die ich im nächsten Kapitel beschreibe, angehen und beseitigen.

# 5

# ARA

## Ein Fundament

Für meine Analysen habe ich größtenteils den Ansatz von Dietrich erweitert. Konkret habe ich dafür den *Automatic Real-Time–System Analyzer (ARA)* geschrieben. In dieser Implementierung habe ich unter anderem die meisten Restriktionen von dOSEK beseitigt. Die Teile, die bestehende Techniken adaptieren und die das Fundament meiner fortführenden Analysen bilden, sind Thema dieses Kapitels.

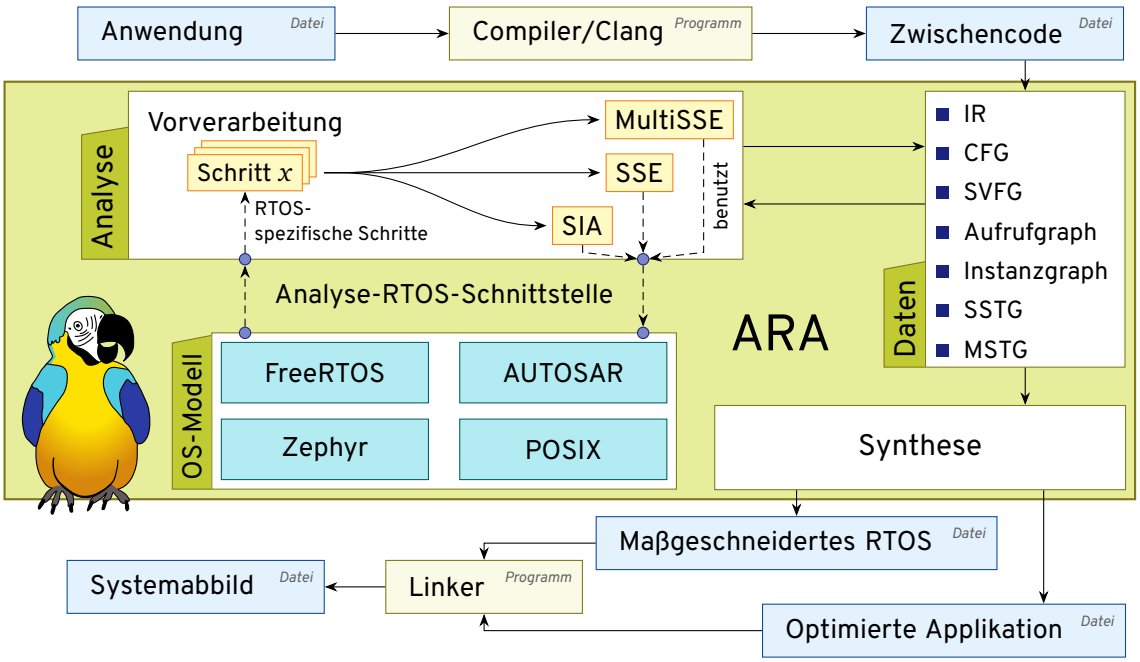



**Abbildung 5.1** Überblick über ARA. ARA bekommt die gesamte Applikation als LLVM-Zwischencode, analysiert diese und synthetisiert anschließend eine optimierte Applikation mit einem maßgeschneidertem RTOS. Die Analysen und die Synthese benutzen dabei verschiedene Graphen als geteilte Daten. Alle Analysen werden im Verlauf der Arbeit erklärt. Um betriebssystemagnostisch zu sein, verwenden die Analysen über eine Analyse-RTOS-Schnittstelle verschiedene Betriebssystemmodelle (Kapitel 8). Damit die betriebssystemgewahren Analysen SIA (Kapitel 6), SSE und MultiSSE (Kapitel 7) arbeiten können, brauchen sie eine Reihe von Vorverarbeitungsschritten, um die es in diesem Kapitel geht.

# 5.1 Überblick

*ARA & LLVM*

ARA basiert, wie auch dOSEK, auf der LLVM-Compilerinfrastruktur. LLVM wurde initial von Chris Lattner geschrieben und ist inzwischen, neben GCC, einer der bekanntesten Open-Source-Compiler [LA04]. Das Hauptmerkmal von LLVM ist die markante Aufteilung in Front-, Middle- und Backend, bei der das Frontend die eigentliche Übersetzung einer Hochsprache in eine Zwischensprache, der LLVM-*Intermediate Representation (IR)*, erledigt (für C und C++ ist dies das Programm Clang), das Middleend Optimierungen vornimmt und das Backend die Zwischensprache in den plattformspezifischen Binärcode übersetzt. Sämtliche Optimierungen werden dabei vom Middleend auf der *gleichen* Zwischensprache erledigt, die folglich alle notwendigen Informationen enthält[22].

[22] Andere Übersetzer haben hier mehrere Zwischensprachen bzw. Zugriff zwischen den Ebenen.

Die LLVM-IR ähnelt dabei einem typisierten Assembler mit beliebiger Anzahl an Registern. Ein paar Eigenschaften sind für diese Arbeit relevant: *LLVM-IR*

- Es gibt primitive Datentypen (i16, i32, float, ...) und Verbunddatentypen (ähnlich wie C-Structs), was die Arbeit der Wertanalyse erleichtert.
- Der Code liegt vollständig codiert in Basisblöcken vor, die vom IR-Parser auch direkt zu einem funktionslokalen Kontrollfluss verbunden werden.

ARA agiert anschließend als Übersetzer der ganzen Anwendung („whole-system compiler“, siehe Abbildung 5.1). Es erwartet Anwendungen in Form von LLVM-IR und extrahiert dort mittels LLVM den funktionslokalen Kontrollfluss. Anschließend führt es nach verschiedenen Vorverarbeitungsschritten eine der Hauptanalysen aus (das Thema der nächsten Kapitel). Mit dem dort extrahierten Wissen kann ein Syntheseschritt dann ein optimiertes RTOS erstellen, das zusammen mit der RTA mittels LLVM-Linker zu einem RTS zusammengebaut wird. ARA ist sowohl in C++ (Schnittstelle zu LLVM und SVF, gute Performance) als auch in Python (geringe Entwicklungskosten) geschrieben mit der Graphbibliothek „graph-tool“ [Pei14] als Daten-Schnittstelle, die beide Sprachen beherrscht. In diesem Kapitel soll es um die Vorverarbeitungsschritte gehen.

## 5.2 Interprozeduraler Kontrollfluss

ARA konstruiert zu Beginn einen *interprozeduralen Kontrollflussgraphen (interprocedural Control-Flow Graph, ICFG)*, der im Gegensatz zu dOSEK Funktionszeiger erlaubt. Vor allem die Analyse von Applikationen auf C++-Basis erfordert dies, da die Sprache Vererbung bzw. die dynamische Bindung intern über Funktionszeiger umsetzt. Für die Korrektheit der anderen Analysen muss der ICFG vollständig sein: Existiert keine Kante zwischen Aufruf und Funktion, wird diese garantiert *nicht* aufgerufen. Die Analyse geschieht mehrstufig: *vollständiger ICFG*

1. Die Interpretation von Funktionszeigern erfordert eine Analyse ihrer Werte[23]. ARA verwendet zunächst den SVF für deren Extraktion, der allerdings nur eindeutig bestimmbare Werte liefert, also eine korrekte aber nicht vollständige Analyse darstellt [SX16].
2. Für alle noch unklaren Funktionszeigerwerte extrahiert ARA anschließend aus dem Code die Menge aller Funktionen, von denen überhaupt ein Zeiger genommen wird (die gültige Domäne, die einzigen Funktionen, die als Zeigerwert in Frage kommen).
3. Diese Menge reduziert ARA anschließend auf typkompatible Zeiger. Es vergleicht dazu die Signatur der Funktion mit dem Zeigertyp. Zwei Signaturen müssen dabei

[23] Im Gegensatz zur Aliasanalyse fragt diese explizit nach dem Datum des Zeigers.

sowohl in der Parameteranzahl als auch in deren Typen übereinstimmen.[24] Dieser Ansatz kombiniert die Bildung eines „Points-To“ Sets (die Menge aller möglichen Funktionszeiger) mit einem Filtern der Ergebnisse durch die Funktionssignatur [Atk04].

4. Die so überapproximierten Funktionszeigerziele übergibt ARA wieder zurück an den SVF, um auch dessen abgeleiteten Werteflussgraph auf einem vollständigen Aufrufgraphen zu erhalten.

Da diese Analyse die Funktionszeigerziele überabschätzt, ist sie vollständig aber nicht korrekt. Für die späteren Analysen ist aber nur wichtig, jeden möglichen Kontrollfluss zu traversieren, was so gewährleistet ist. ARA speichert den interprozeduralen Kontrollfluss außerdem in kondensierter Form als Aufrufgraph ab.

## 5.3 Wertanalyse

Eine weitere Einschränkung von dOSEK ist die Einschränkung auf globale Konstanten für die Systemaufrufargumente. Diese Annahme hat sich für bestimmte Echtweltapplikationen als nicht haltbar herausgestellt, die zwar statisch extrahierbare Systemaufrufargumente besitzen, diese aber in Form von funktionslokalen Zeigern oder als Teil von globalen Verbundtypen herumreichen (siehe Abschnitt 4.6.3). Um das Problem anzugehen, wird eine Wertanalyse benötigt. Grundsätzlich gibt es dazu zwei Ansätze:

1. Die Wertanalyse als Teil der abstrakten Interpretation ausführen. Dazu müssen die relevanten Werte zusätzlich in den abstrakten Zustand eingebettet werden und bei der Transitionsfunktion entsprechend berücksichtigt werden. Astrée z. B. wählt diesen Ansatz [CCF+05]. Für betriebssystemgewahre Analysen hat dies aber den signifikanten Nachteil, dass – statt nur den Systemaufrufen – jede Instruktion berücksichtigt werden muss, die Werte manipuliert, die *potentiell* ein Argument für Systemaufrufe sind oder diese beeinflussen können [VV09]. Da diese Menge im Vorhinein unbekannt ist, betrifft dies alle Werte. Ich habe mich daher gegen diesen Ansatz entschieden.

2. Die Wertanalyse als separate Rückwärtsanalyse auf einem Werteflussgraphen zu realisieren. Der Ansatz ist hier wie folgt: Wann immer der konkrete Wert eines Systemaufrufes benötigt wird, startet ausgehend vom aktuellen Systemaufruf und ausgestattet mit

[24] Besonderes Augenmerk muss dabei auf Verbunddatentypen gelegt werden. Vererbte Typen in C++ (bei denen die Kindklasse als Spezialisierung der Elternklasse behandelt wird) werden in LLVM derart übersetzt, dass die Felder der Elternklasse in die Kindklasse eingebettet werden (bei Mehrfachvererbung an beliebigen Positionen). Somit behandelt ARA auch die Verbunddatentypen als kompatibel zum geforderten Typ, die den geforderten Typen enthalten.

```
Eingabe:
  • llvm_value:  Zu suchender Wert, z.B ein (Systemaufruf)-Argument
  • type:        Typ des Arguments: Konstante oder Symbol
  • callpath:    Aufrufpfad/Kontext, in dem der Wert gefunden werden soll
Ausgabe:
  • value_node:  Entsprechender Knoten des SVFG

def get_value(llvm_value, type, callpath):
    vfg = SVF.getVFG(filter_by=callpath)
    value_node = vfg.get(llvm_value)

    while value_node.hasUniquePredecessor():
        value_node = value_node.predecessor

    if value_node.isConstant and ptype == CONSTANT:
        return value_node
    else if value_node.isSymbol and ptype == SYMBOL:
        return value_node

    raise NOT_FOUND

Eingabe:
  • pointer:   Zeiger, dessen Ziel gesetzt werden soll
  • obj:       Instanz/Objekt, auf das der Zeiger gesetzt werden soll
  • callpath:  Aufrufpfad/Kontext, in dem der Wert gefunden werden soll

def set_pointer(pointer, obj, callpath):
    value = get_value(pointer, SYMBOL, callpath)
    value.setValue(obj)
```

**Codeblock 5.1** Suche nach Werten auf dem vorliegenden SVFG (vereinfacht). Die Funktion get_value filtert zuerst den SVFG anhand des übergebenen Kontextes, sodass nur Parameterübergaben von gültigen Funktionen beachtet werden. Auf diesem Pfad läuft die Funktion anschließend ausgehend vom initial übergebenen Argument den Graph aufwärts, bis sie den gewünschten Wert findet. Falls sie nichts findet, liefert sie eine passende Fehlermeldung. Die Funktion set_value benutzt intern get_value, um die passende Speicherstelle zu finden.

dem aktuellen Aufrufkontext eine Rückwärtssuche auf dem gerichteten Werteflussgraphen. Für die Konstruktion des Werteflussgraphen verwendet ARA den bereits vorgestellten SVFG.

*Verwendung: Werteflussgraph*

Der zweite Ansatz hat sich als einerseits sehr performant und andererseits als praktikabel genug erwiesen, auch komplizierte Muster finden zu können. Der Wert des Semaphors aus dem Beispielcode aus dem letzten Kapitel (Quellcode 4.1) kann beispielsweise mithilfe des

SVFG gefunden werden. Wie dOSEK hat ARA weiterhin die Einschränkung, dass Systemaufrufargumente statisch extrahierbar, also konstant (oder berechenbar) sein müssen. Im Gegensatz zu dOSEK kann ARA aber durch die Wertanalyse deutlich komplexere Muster erkennen und erhöht damit die Menge der Anwendungen, die analysierbar sind.

*Wertanalyse*

Die SVF liefert zuerst einmal nur einen SVFG. Um darauf einen Wert zu finden, muss ARA noch eine konkrete Analyse implementieren. Diese ist vereinfacht in Quellcode 5.1 dargestellt. ARA stellt dabei grundsätzlich zwei Methoden zur Verfügung: `get_value` und `set_value`. Die erstere dient der Suche nach einem Wert. Der Typ spezifiziert dabei, ob eine Konstante (also ein konkreter Wert) oder ein Symbol (also eine Speicherstelle) gefunden werden soll. Letzteres ist beispielsweise bei Zeigern vonnöten, denen der Systemaufruf etwas zuweist. Weiterhin haben Systemaufrufe Rückgabewerte, weisen also Symbolen einen neuen Wert zu (z.B. einem Zeiger eine Instanz). Dieser Rückgabewert kann mit `set_value` der Wertanalyse bekannt gemacht werden, sodass diese beim nächsten Aufruf von `get_value` den zuvor zugewiesenen Rückgabewert liefert.

## 5.4 Atomare Basisblöcke

Auch für meine Analysen haben sich ABBs als gute Abstraktion für eine selektive Analyse herausgestellt. Ich habe diese dazu wie von Dietrich implementiert übernommen, will aber ihre Definition expliziter gestalten:

**Definition 18: Atomarer Basisblock.**
Ein *Atomarer Basisblock (Atomic Basic Block, ABB)* ist eine Menge von BBs, die die folgenden Bedingungen erfüllt:

1. Jeder ABB hat einen Typen: SYSCALL, CALL oder COMPUTATION.
2. Wenn eine Instruktion ein Systemaufruf ist, formt diese alleine einen ABB mit dem Typen SYSCALL. Wenn eine Instruktion ein Aufruf ist, dessen Zielfunktion als Teil ihrer Bearbeitung einen Systemaufruf macht, formt diese Instruktion einen ABB mit dem Typen CALL.
3. Jeder ABB hat genau einen Eintritts-BB und genau einen Austritts-BB, der ihn mit anderen ABBs verbindet (SESE-Region).
4. Zwei aufeinanderfolgende ABBs gehören niemals beide der Typmenge SYSCALL und CALL an. Zwischen ihnen befindet sich also immer mindestens ein ABB vom Typ COMPUTATION (der auch aus der leeren Instruktion bestehen kann).

Damit ergibt sich ein triviales Verfahren zur Konstruktion von ABBs, die bereits für alle Analysen ausreichend sind:

1. Enthält ein Basisblock einen Systemaufruf oder Aufruf, trenne den Basisblock davor oder dahinter auf (Bedingung 2).

2. Folgen zwei Basisblöcke aufeinander, die einen (System-)Aufruf beinhalten, füge einen neuen BB dazwischen ein, der die leere Anweisung enthält (Bedingung 4).

3. Bilde jeweils einen ABB auf genau einen BB ab (Bedingung 3).

4. Ordne jedem ABB seinen entsprechenden Typen zu (Bedingung 1).

*Maximale ABBs*

Dieses Verfahren erzeugt allerdings (noch) nicht die von Dietrich verwendeten ABBs, da dieser maximale ABBs benutzt:

**Definition 19: Ordnungsrelation auf ABBs.**
Ein ABB $a$ ist größer als ein ABB $b$, wenn $a$ eine echte Übermenge an BBs in Bezug auf $b$ enthält.

**Definition 20: Maximaler ABB.**
Ein ABB ist maximal, wenn es keinen größeren ABB gibt.

*Endschleifen*

Ich erzeuge nun ebenfalls maximale ABBs mit dem bereits angesprochen Verfahren über Dominatoren, behandle aber Endschleifen verschieden:

**Definition 21: Endschleife.**
Eine Endschleife ist eine Endlosschleife, die bereits statisch erkennbar keinen Ausgang hat und darum für das Programm selbst das Ende der weiteren Ausführung bedeutet.

Endschleifen sind z. B. beliebt zur Implementierung von Assertions (im Fehlerfall lösen sie eine Fehlermeldung und Endlosschleife aus) oder vor Code, der nie erreicht werden soll (z. B. das Ende der main-Funkion in einem eingebetteten System) und haben daher eine hohe praktische Relevanz.

Da die Erkennung von Endschleifen für meine Analysen die Präzision erhöht, will ich sie im ABB-Graphen erhalten. Die standardmäßige Konstruktion von maximalen ABBs über Dominatoren verschmilzt Endschleifen aber zwangsläufig zu einem singulären ABB. Um dies zu verhindern, erkennt die Analyse zusätzlich den Schleifenkopf von Endschleifen und lässt diesen gleichwertig wie Systemaufrufe oder Aufrufe zu systemrelevanten Funktionen behandeln, die bereits als Barrieren für den Verschmelzungsprozess dienen. Minimale ABBs, hauptsächlich eine eins-zu-eins-Abbildung auf Basisblöcke, sind bereits ausreichend für alle folgenden Analysen. Maximale ABBs mit der Erkennung von Endschleifen erhöhen die Analysegeschwindigkeit bei gleichbleibender Präzision.

Es ist außerdem anzumerken, dass ABBs nur für bestimmte Analysen Vorteile bringen, konkret für alle, die auf der Betriebssystemdomäne operieren. Die bereits angesprochene

Wertanalyse benutzt aber z. B. eine andere abstrakte Domäne und muss daher auf einer anderen selektiven Menge operieren: Der Menge an Anweisungen, die Daten laden, speichern oder kopieren. Genaugenommen ist die Verwendung von ABBs auch nur möglich, weil die Wertanalyse vorgelagert ist und die Systemaufrufinterpretation durch die Wertanalyse indirekt auf die andere Domäne zurückgreifen kann.

## 5.5 Zusammenfassung

ARA ist ein Programm zur betriebssystemgewahren Optimierung von RTSs. Es verbindet dazu die Domäne der eingebetteten Systeme mit der Methode der abstrakten Analyse (Abbildung 3). Für die folgenden Synthesen hat dieses Kapitel die notwendigen Vorverarbeitungsschritte vorgestellt.

Dies ist zum einen die Wertanalyse, die den (konstanten) Wert von Variablen statisch bestimmen kann. Zum anderen führt ARA eine Kontrollflusskonstruktion durch, die durch die Verfolgung von Funktionsaufrufen den ICFG und Aufrufgraph erzeugt und zusätzlich Basisblöcke in ABBs zusammenfasst.

# Teil II
# Konzepte

Teil I hat sich mit den Grundlagen von eingebetteten Systemen beschäftigt, hat statische Analysen vorgestellt und ist dabei insbesondere auf betriebssystemgewahre Analysen eingegangen – statische Analysen, die eingebettete Systeme analysieren. Abgeschlossen hat der Teil mit der Vorstellung von ARA, einem Werkzeug, das betriebssystemgewahre Analysen durchführen kann.

In diesem Teil soll es um genau diese betriebssystemgewahren Analysen und deren Verbindung gehen, die ich im Rahmen dieser Arbeit neu entwickelt habe. Die schon vorgestellte Abbildung 5.1 kann dabei als Leitfaden dienen: Ich werde in Kapitel 6 die *statische Instanzanalyse (Static Instance Analysis, SIA)* vorstellen und damit meine 1. Forschungsfrage beantworten, anschließend in Kapitel 7 die *MultiSSE* vorstellen, die die 2. Forschungsfrage beantwortet und zur Beantwortung der 3. Forschungsfrage abschließen mit der Integration beider Analysen in ein Framework mit gemeinsamer *Analyse-RTOS-Schnittstelle*, um sie betriebssystemagnostisch zu machen.

# 6

# Statische Analyse dynamischer Systeme

## Die Suche nach Instanzen und deren Interaktionen

Einige RTOS-Schnittstellen sind dynamisch gestaltet, was betriebssystemgewahre Analysen, die auf der abstrakten Interpretation basieren, vor eine erhöhte Herausforderung stellt: Im Allgemeinen ist die genaue Menge der Instanzen erst zur Laufzeit bekannt. Analysen, wie die SSE, die die Interaktionen zwischen Instanzen in jedem beliebigen Systemzustand untersuchen, können daher nicht ohne weiteres arbeiten. Dieses Kapitel stellt darum eine betriebssystemgewahre Analyse vor, die gezielt nach der Menge der Instanzen und deren Interaktionen sucht. Diese kann einerseits von weiteren Analysen weiterverwendet werden. Andererseits kann aber auch eine nachgeschaltete Synthese (wie in diesem Kapitel vorgestellt) auf dieser Basis dynamisch Instanzen, die prinzipiell statisch angelegt werden könnten, automatisch zu solchen statischen Instanzen umformen.

Dynamik in eingebetteten Systemen und deren Vorhersagbarkeit stehen in Widerspruch: Hardwareeigenschaften wie der Speicherverbrauch sind bei dynamischen Instanzen nicht vorhersagbar. Das Zeitverhalten des Systems ändert sich in Abhängigkeit davon, ob eine Instanz präsent ist oder nicht. Wenn sich das System verschiedenen äußeren Gegebenheiten mit einer vollständig anderen Konfiguration anpasst, ist sogar die wesentliche Eigenschaft eingebetteter Systeme verletzt, das System mit einem ganz bestimmten Anwendungszweck zu verknüpfen.

*dynamische RTOS-Schnittstellen*

Nichtsdestotrotz sind viele RTOS-Schnittstellen dynamisch gestaltet. Der genaue Grund ist unklar. Dynamische Schnittstellen suggerieren zuerst einmal höhere Flexibilität beim Entwickeln, die aber später vielleicht gar nicht mehr gebraucht wird. Weiterhin sind viele Entwickler ein dynamisches Programmiermodell von Anwendungen außerhalb der eingebetteten Domäne gewohnt. Fiedler geht in seiner Dissertation genauer auf die Frage ein, welche Probleme bei welcher Schnittstellenwahl auftreten [Fie23A6/2.2.1]. Von den hier verwendeten RTOSs schreibt einzig AUTOSAR statische Instanzen vor, während Zephyr immerhin solche ermöglicht. FreeRTOS und POSIX unterstützen ausschließlich dynamische Instanzen. Dies führt mich zu meiner 1. Forschungsfrage: Welche Herausforderungen entstehen auf eingebetteten dynamischen Systemen und können betriebssystemgewahre Analysen diese überwinden?

*pseudostatische Instanzen*

Ich will mich bei der Beantwortung vor allem auf zwei Aspekte beziehen: Dynamische Instanzen und deren Interaktionen. Bei der Untersuchung der Echtweltprogramme GPSLogger und LibrePilot (beide auf FreeRTOS basierend) war das Konzept der pseudostatischen Instanziierung zu erkennen:

> **Definition 22: Pseudostatische Instanz.**
> Eine pseudostatische Instanz ist eine Instanz, deren Parameter bereits statisch bekannt sind, die aber trotzdem dynamisch erzeugt wird. Bekannt sein müssen im Speziellen die Anzahl der Erzeugungen und die vollständige Kenntnis der Aufrufkontexte.

Anwendungsentwickler intendieren eine statische Instanz, die in der ganzen Anwendung verwendet werden kann, haben aber nicht die Mittel, dies im RTOS auszudrücken. Der offensichtlichste Ort für pseudostatische Instanzen ist die Initialisierungsphase des Systems (die Phase bis zum System Setup Point, vgl. Abbildung 2.2). Instanzen werden in dieser Phase (im Normalfall) genau einmal angelegt, da diese Phase insgesamt nur einmal durchlaufen wird, und arbeiten zudem mit globalen Parametern, um im restlichen Programm bekannt zu sein. Pseudostatische Instanzen haben im Vergleich zu echt statischen Instanzen einige Nachteile:

- Sie schränken Echtzeitanalysen ein: Die Analyse muss um die Existenz der Instanz wissen, um das Gesamtverhalten berechnen zu können. Dieses Wissen liegt aber ohne weiteres nicht (statisch) vor.

- Der maximale Speicherverbrauch ist statisch nicht bestimmbar. Damit ist die korrekte Dimensionierung des Arbeitsspeichers nicht mehr möglich.

- Sie führen zu sinnlosen Berechnungen: Die Initialisierungsphase bis zum Erreichen des SSP ist die erste Frist im System. Erst danach ist das System in der Lage, geeignet auf externe Ereignisse zu reagieren. Aus diesem Grund hat diese Phase sogar Einzug in offizielle Standards gefunden, so muss z. B. die Rückfahrkamera in einem US-Fahrzeug nach der FMVSS111 innerhalb von einer Sekunde nach Einlegen des Rückwärtsgangs ein Bild liefern können [Tra18]. Die Initialisierungsphase beeinflusst weiterhin maßgeblich die Garantien zur Fehlertoleranz bei Systemen, die bei transienten Fehlern neustarten [FRS+16,FRR+17,ANN+20]. Pseudostatische Instanzen werden in dieser Phase ein jedes Mal gleich angelegt und verlängern dabei unnötig die Initialisierungsphase, sodass die Wahrscheinlichkeit für transiente Fehler zusätzlich steigt.

*generische RTOS-Interaktionen*

Eine Limitierung durch die RTOS-Semantik gibt es auch noch an einer weiteren Stelle: Bei Interaktionen zwischen Instanzen. Bestimmte Interaktionsmuster erlauben effiziente Datenstrukturen, so gibt es z. B. Spezialimplementierungen von Single-Reader-Single-Writer-Warteschlangen, die durch weiche Synchronisierung im Vergleich zu generischen Warteschlagen deutlich performanter sind [LGC+13,GMV08,MS96]. Oftmals bietet aber die RTOS-Semantik nur generische Systemaufrufe, die solche Interaktionsmuster nicht beachten und dementsprechend auch mithilfe einer generischen Implementierung arbeiten müssen. Statische RTOSs wie AUTOSAR erlauben bzw. erzwingen die statische Erklärung von Interaktionsmustern. Beispielsweise muss statisch klar sein, welche Ressourcen ein Task verwenden darf, um daraus bereits zur Kompilierzeit die geeignete dynamische Priorität für das PCP berechnen zu können. Dynamische RTOS vereiteln diesen Versuch im Ansatz: Der Anwendungsentwickler hat keine Möglichkeit statisch auszudrücken, wie Instanzen auf andere zugreifen, da schon die Instanzen statisch nicht bekannt sind.

*Lösung durch Optimierung*

Sowohl die Nachteile pseudostatischer Instanziierung als auch die folgende Fehlbeschreibung von Interaktionsmustern sind durch geeignete Optimierung lösbar: Die Anwendung muss in einem ersten Schritt bezüglich der Verwendung von Instanzen und Interaktionen untersucht werden. Aus Basis dieser Informationen kann anschließend eine Optimierung pseudostatische in echt statische Instanzen umwandeln und zusätzlich die Implementierung für erkannte aber nicht implementierte Interaktionsmuster gegen eine performantere Implementierung austauschen.

Björn Fiedler und ich sind diese Probleme als Teil des ARA-Frameworks angegangen. Ich habe dabei die betriebssystemgewahren Analysen übernommen, um die es im Folgenden

```
template <class DT> class GuardedData {
    SemaphoreHandle_t data_mutex;
    DT internal_data;
  public:
    void get(DT* d) {
        xSemaphoreTake(data_mutex, BLOCK_FOREVER);
        *d = internal_data;
        xSemaphoreGive(data_mutex); }
    void set(DT* d) {
        xSemaphoreTake(data_mutex, BLOCK_FOREVER);
        internal_data = *d;
        xSemaphoreGive(data_mutex); }
    GuardedData() { data_mutex = xSemaphoreCreateMutex(); } };

QueueHandle_t storage_queue;
TaskHandle_t fusion_task;
TaskHandle_t storage_task;

int main() {
    SensorInfo_t *si = detect_sensors();
    xTaskCreate(fusion_entry, "fusion", 512, si, 1, &fusion_task);
    for (int i=0; i < si->length; i++) {
        xTaskCreate(sensors_entry, "sensor", 512,
                    &si->sensors[i], 2, &(si->sensors[i]->task));
        si->sensors[i]->data = new GuardedData<SensorData_t>(); }
    vTaskStartScheduler(); }

void fusion_entry(void *param) {
    SensorInfo_t *si = (SensorInfo_t*) param;
    storage_queue = xQueueCreate(sizeof(message_t), 4);
    xTaskCreate(storage_entry, "storage", 512, NULL, 3, &storage_task);
    while (1) {
        SensorData_t result = do_fusion(si);
        xQueueSend(storage_queue, &result, BLOCK_FOREVER); } }

void storage_entry(void *param) {
    init_storage();
    while (1) {
        SensorData_t data;
        xQueueReceive(storage_queue, &data, BLOCK_FOREVER);
        storage_write_data(&data); } }

void sensor_entry(void *param) {
    Sensor_t *sensor = (Sensor_t*) param;
    while (1) {
        SensorData_t data = sensor->function(sensor);
        sensor->data->set(& data); } }
```

**Codeblock 6.1** Beispielanwendung für FreeRTOS (adaptiert aus [Fie23A3]). Markiert sind Systemaufrufe, die Instanzen erstellen oder Instanzen interagieren lassen.

geht. Der naive Ansatz liegt dabei in der Wiederverwendung bzw. Erweiterung der bereits existierenden Analysen, im Speziellen der SSE. Diese scheint aufgrund der bereits vorhandenen Interpretation aller Systemaufrufe gut geeignet, auch solche Systemaufrufe zu interpretieren, die Instanzen erzeugen und als Nebenprodukt pseudostatische Instanzen und deren Interaktionen zu erkennen. Allerdings zeigt die theoretische Betrachtung der SSE in Abschnitt 4.6.1, dass sie nur für statische Systeme geeignet ist, insbesondere eine endliche Menge an Potenzmengenzuständen aufweisen muss. Dynamische RTOSs erfüllen diese Eigenschaft aber inhärent nicht, da Instanzen immer in Schleifen angelegt werden können und ihre Anzahl damit potentiell unendlich ist[25]. Der typische Mechanismus, unbeschränkten Größen innerhalb einer abstrakten Interpretation zu begegnen, ist das Widening. Die SSE erzeugt darüber hinaus noch viel zu viele Informationen für den geforderten Anwendungszweck. Es ist darum sinnvoll, den abstrakten Zustand anzupassen, auf diesem Widening zu ermöglichen und die Analyse zudem selektiver zu gestalten und damit zu beschleunigen.

*Instanzgraph und SIA*

Ich werde darum in diesem Kapitel zuerst den Instanzgraphen vorstellen, der ein geeignetes Resultat einer betriebssystemgewahren statischen Analyse ist, um von einer anschließenden Optimierung verwendet zu werden. Dieser stellt außerdem einen im Vergleich zu den Zuständen der SSE deutlich verkleinerten (obgleich an einigen Stellen erweiterten) Zustand dar. Anschließend werde ich die *statische Instanzanalyse (Static Instance Analysis, SIA)* vorstellen, eine Analyse, die in der Lage ist, mit dynamischen Instanzen umzugehen. Zuletzt werde ich die Analyse evaluieren und diskutieren, insbesondere im Vergleich zur SSE.

*laufendes Beispiel*

Des besseren Verständnisses willen verwende ich für die hier vorgestellten Analysen ein laufendes Beispiel: Quellcode 6.1. Diese Anwendung basiert auf FreeRTOS und wurde initial von Björn Fiedler für seine Dissertation entworfen [Fie23A3]. Sie besteht aus mindestens drei Fäden, die mittels verschiedener Synchronisationsobjekte miteinander kommunizieren. In diesem System ermitteln verschieden viele Fäden Sensorwerte (ein Faden pro Sensor) und übermitteln diese an den Faden „*fusion*“, der die Werte zusammenführt und anschließend weiter überträgt an einen Faden „*storage*“, der die zusammengeführten Sensorwerte wegspeichert. Hier ist bereits mit geübtem Blick erkennbar, dass der *fusion*-Faden eine pseudostatische Instanz ist, die singulär in der main-Funktion erstellt wird. Auch der *storage*-Faden ist eine solche, die allerdings bereits hinter dem SSP erzeugt wird. Hier garantiert die Betriebssystemsemantik allerdings das singuläre Durchlaufen des entsprechenden Systemaufrufs, der die Instanz erstellt. Die Menge der *sensor*-Fäden hingegen ist keine Menge pseudostatischer Instanzen, da deren Anzahl erst zur Laufzeit bekannt ist.

[25] Eine unendliche Anzahl Instanzen ist praktisch unsinnig, allerdings ist die Erzeugung von Instanzen in begrenzten Schleifen durchaus denkbar und setzt daher von der Analyse voraus, die Grenze entweder selbst herauszufinden oder mitgeteilt zu bekommen, was den Analyserahmen um ein Vielfaches vergrößert.

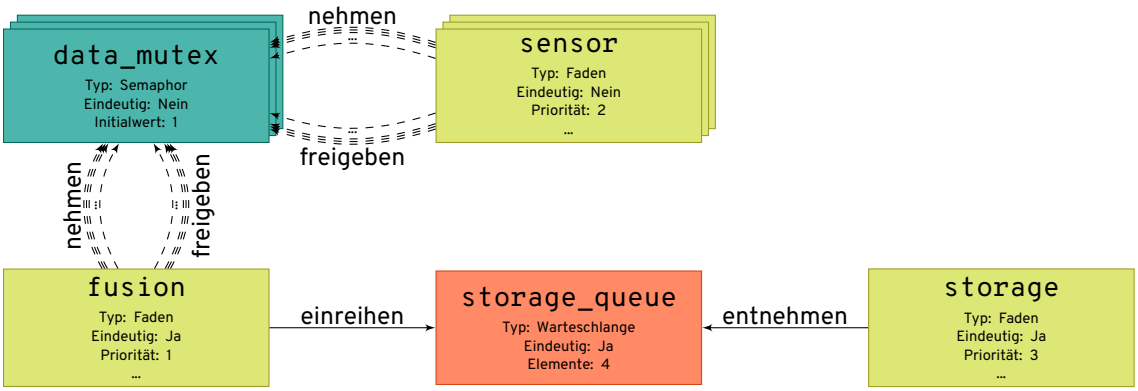



**Abbildung 6.1** Instanzgraph der Beispielanwendung. Erkennbar sind drei Fäden, eine Warteschlange und ein Semaphor. Der *sensor*-Faden und der Semphor *data_mutex* ist dabei eine Fadenvorlage bzw. Semaphorvorlage, deren Anzahlen nicht eindeutig sind. Grafik adaptiert aus [Fie23A3].

## 6.1 Der Instanzgraph

*Instanzgraph*

Die hauptsächliche Datenstruktur, um Wissen über Instanzen und Interaktionen auszutauschen, ist der *Instanzgraph*. Abbildung 6.1 zeigt den Instanzgraphen der Beispielanwendung. Der Graph beinhaltet dabei:

- Eine Menge an Knoten: Diese beschreiben alle Instanzen der Anwendung mithilfe verschiedener Parameter. Dazu gehören insbesondere der Typ der Instanz und deren Häufigkeit (ist die Instanz eindeutig, innerhalb einer Abzweigung oder innerhalb einer Schleife erzeugt) sowie instanzspezifische Parameter. In Abbildung 6.1 ist das z. B. die Fadenpriorität. In der Abbildung ist die Häufigkeit der Instanz zusätzlich grafisch dargestellt (z. B. bei *sensor*), wird aber in der unterliegenden Datenstruktur mit nur einem Knoten repräsentiert.

- Eine Menge an gerichteten Kanten: Jede Kante beschreibt dabei eine Interaktion zwischen einer Instanz und einer anderen (startend bei der auslösenden Instanz). Die Interaktion ist an einen Systemaufruf gekoppelt, der aber in mehreren Kontexten aufgerufen werden kann, was über ein entsprechendes „Anzahl“-Feld markiert ist. In Abbildung 6.1 ist auch hier die Anzahl grafisch repräsentiert, besteht aber in der unterliegenden Datenstruktur nur aus genau einer Kante.

*Bedeutung des Instanzgraphen*

Mit diesen Informationen stellt der Instanzgraph eine flussinsensitive Beschreibung des Programmaufbaus aus Sicht des Betriebssystems dar. Instanzen, die im Instanzgraphen als eindeutig, also genau einmal erzeugt, markiert sind, sind pseudostatische Instanzen (in Abbildung 6.1 z. B. *fusion* und *storage*, aber nicht *sensor*). Die Interaktionsmuster ergeben sich

aus den Kanten: Existiert beispielsweise nur eine eingehende und eine ausgehende Kante in eine Warteschlange, handelt es sich um eine Single-Reader-Single-Writer-Warteschlange (in Abbildung 6.1 z. B. die *storage_queue*). Der Interaktionsgraph stellt die Schnittstelle zwischen Analyse und Optimierung dar.

## 6.2 Die Statische Instanzanalyse (SIA)

Die SIA ist eine Analyse, die aus dem Quellcode der Anwendung einen Instanzgraphen erzeugt. Die Analyse folgt prinzipiell dem Muster einer abstrakten Interpretation. Für den Algorithmus gibt es dabei einige Dinge zu beachten: *Anforderungen*

1. Die Analysen müssen pro RTOS verschieden tief sein. RTOSs, bei denen Instanzen dynamisch erstellt werden, brauchen eine Analyse, um die Menge der Instanzen (die Graphknoten) zu bestimmen. RTOSs, bei denen Instanzen statisch erstellt werden (wie AUTOSAR), reicht eine Analyse der Interaktionen (die Graphkanten). Bei AUTOSAR und Zephyr braucht ARA allerdings für die Bestimmung der statischen Instanzmenge eine spezifische Analyse der Konfigurationsdatei bzw. des Quellcodes.

2. Abhängig vom RTOS gibt es verschiedene Programmstartpunkte. Dieser muss also parametrisierbar sein. Beispielsweise startet FreeRTOS in einer `main`-Funktion, während Zephyr zwar optional eine `main`-Funktion unterstützt (intern als eigener Faden mit höchster Priorität realisiert, der andere Fäden starten kann), aber auch direkt mit der Ausführung der Tasks beginnen kann.

3. Die SIA beeinflusst ihre eigene Ausführung: Eine klassische abstrakte Interpretation traversiert den erreichbaren Programmcode. Erkennt die SIA nun aber eine neue Aktivität, so ist diese mit (eventuell) vormals unerreichbarem Programmcode verknüpft, der als Folge ebenfalls traversiert werden muss. Ein Beispiel dafür sind Fäden, die ihrerseits wieder Fäden erzeugen (wie *fusion*).

4. Da der Instanzgraph flussinsensitiv ist, wäre ebenfalls eine flussinsensitive Analyse wünschenswert, die diesen erzeugt. Dem steht die fundamentale Abhängigkeit gegenüber, dass Instanzen erzeugt werden müssen, bevor sie interagieren können. Ich habe dazu die Instanzgrapherstellung in zwei baugleiche Analysen aufgesplittet, die SIA, die nach Instanzen sucht und die *Interaktionsanalyse (Interaction Analysis, INA)*, die *anschließend* nach Interaktionen zwischen diesen sucht.

Der Algorithmus selbst funktioniert diesen Anforderungen entsprechend nach dem Grundgerüst, das in Quellcode 6.2 dargestellt ist. Sowohl das Kriterium 2, als auch das Kriterium 3 werden durch ein Mehrfachausführung des Algorithmus gelöst, die jeweils einen bestehenden Instanzgraphen erweitert. *SIA im Überblick*

*Aufrufreihenfolge der SIA, die durch ARA vorgegeben wird:*

```
Eingabe:
  • instance_graph: Leerer Instanzgraph
  • entry_points:   „main“ oder vom Nutzer vorgegebene Funktion
  • flow_graph:     Interprozeduraler CFG oder Aufrufgraph (modusabhängig)
Ausgabe:
  • instance_graph: Vollständiger Instanzgraph

def call_sia(instance_graph, entry_points, flow_graph):
    while entry_points are not empty:
        instance_graph, new_entry_points = sia(instance_graph,
                                               entry_points.pop(),
                                               flow_graph)
        entry_points += new_entry_points
    return instance_graph
```

*Der eigentliche SIA-Algorithmus:*

```
Eingabe:
  • instance_graph:    (Unvollständiger) Instanzgraph
  • entry_point:       Zu analysierende Funktion
  • flow_graph:        Interprozeduraler CFG oder Aufrufgraph (modusabhängig)
Ausgabe:
  • instance_graph:    Teilvollständiger Instanzgraph
  • new_entry_points:  Liste an neu zu analysierenden Funktionen

def sia(instance_graph, entry_point, flow_graph):
    new_entry_points = []

    # Systemaufrufe iterieren
    for syscall, call_context in traverse_cfg(flow_graph, entry_point):

        # Parameter aus dem Kontext bestimmen (Wertanalyse)
        args = os_model.get_arguments(syscall)
        params = vfg.resolve(args, call_context)

        # Systemaufruf interpretieren
        new_entity = os_model.add(instancegraph, syscall, params)

        # Aufruf in Schleife, Instanz eindeutig, ...
        add_metadata(new_entity, syscall, call_context)

        # Neuen Kontrollfluss beachten
        if new_entity.is_activity():
            new_entry_points.add(new_entity.entry)
    return instance_graph, new_entry_points
```

**Codeblock 6.2** Grundgerüst der SIA (vereinfacht). call_sia ruft dabei sia so oft wie notwendig auf. Diese iteriert alle Systemaufrufe, findet deren Argumente und interpretiert diese anschließend abstrakt. Durch geeignete Metadaten wird der Widening-Operator umgesetzt.

*Funktionsweise*

Der überliegende Algorithmus startet mit einem leeren Instanzgraphen, dem interprozeduralen Kontrollflussgraphen bzw. Aufrufgraphen, wie in Abschnitt 5.2 beschrieben, und ruft solange die eigentliche SIA auf, bis alle Startpunkte erschöpft sind. Die SIA traversiert den Kontrollfluss und interpretiert anschließend abstrakt jeden Systemaufruf auf seine Auswirkungen auf den Instanzgraphen. Drei Operationen in der SIA sind im Quellcode 6.2 bewusst unspezifisch:

1. Die Traversierung des Kontrollfluss- oder Aufrufgraphen (Zeile 13): Hierzu gibt es zwei Methoden, die ich Abschnitt 6.2.2 genauer beschreiben will. Beide Methoden liefern eine Liste an Systemaufrufen mit dem zugehörigen Aufrufkontext, der für den Aufruf von `add` notwendig ist.

2. Das eigentliche Modifizieren des Instanzgraphen (die Operation `add` in Zeile 20): Da diese Operation sehr spezifisch für jeden Systemaufruf und damit spezifisch für das jeweilige RTOS ist, wird die Operation an ein RTOS-Modell ausgelagert, das ich im Kapitel 8 genauer vorstelle. Insbesondere wird durch dieses Vorgehen der Algorithmus selbst RTOS-agnostisch. An dieser Stelle ist relevant, dass das Modell dabei den entsprechenden Knoten oder die entsprechende Kante im Graphen erzeugt. In Zeile 16 muss das Modell weiterhin die Systemaufrufargumente liefern, die anschließend die Wertanalyse nach dem Algorithmus, der in Abschnitt 5.3 beschrieben ist, auflösen kann.

3. Das Hinzufügen von Metadaten (Zeile 23): Diese beschreiben den Aufrufkontext des Systemaufrufs und damit seine Häufigkeit. Ich werde diese in Abschnitt 6.2.3 vorstellen.

### 6.2.1 Klassifizierung von Systemaufrufen

*SOCs und Kommunikationsaufrufe*

Im Quellcode 6.2 wird nicht weiter spezifiziert, welche Systemaufrufe untersucht werden. Es hat sich aber als sinnvoll erwiesen, die Systemaufrufe zu filtern, wozu es einer Klassifizierung bedarf. Der Instanzgraph besteht aus Knoten (den Instanzen) und Kanten (deren Interaktionen), sodass ich die Systemaufrufe ebenfalls an genau dieser Stelle trennen will: Instanzen werden mithilfe von *Systemobjektkonstruktoren (System Object Creators, SOCs)* erzeugt. Interaktionen zwischen Instanzen passieren mithilfe von *Kommunikationsaufrufen*. In der Beispielanwendung (Quellcode 6.1) ist diese Trennung bereits farblich markiert. Einige Systemaufrufe wirken eher als eine direkte Anweisung an das Betriebssystem als eine Anweisung an eine andere Instanz, z. B. `DisableAllInterrupts`. Diese können aber abstrahiert als eine Kommunikation zwischen dem aktuellen Faden und einer speziellen „RTOS“-Instanz gesehen werden.

*SIA und INA*

Diese Aufteilung ermöglicht die bereits erwähnte Trennung in die SIA und INA, die die Flusssensitivität beider Analysen prinzipiell überflüssig macht. Außerdem ist damit zusätzlich eine feingranularere Analyse möglich. So ist für eine Optimierung von pseudostatischen Instanzen nur die SIA notwendig. Interaktionen, die von der INA gefunden werden,

werden nicht benötigt. Zu guter Letzt realisiert die Trennung die korrekte Behandlung der verschiedenen RTOS-Anforderungen. So benötigt beispielsweise AUTOSAR keine SIA aber eine INA. Die beiden Analysen unterscheiden sich dabei ausschließlich in der Art der interpretierten Systemaufrufe und nicht im Algorithmus.

1. In einem ersten Schritt extrahiert die SIA eine Liste der Instanzen (also ausschließlich die Knoten), indem sie alle SOCs auswertet und die Effekte aller anderen Systemaufrufe ignoriert.

2. In einem zweiten Schritt extrahiert die *Interaktionsanalyse (Interaction Analysis, INA)* die Interaktionen zwischen den Instanzen und vervollständigt somit die Instanzliste in einen Instanzgraphen durch die abstrakte Interpretation der Kommunikationsaufrufe.

### 6.2.2 Traversierung des Kontrollflussgraph

*flusssensitive Iteration*

Der Instanzgraph beschreibt die Verwendung des RTOS seitens der Anwendung in einer *flussinsensitiven* Weise. Wird nun die klassische abstrakte Interpretation auf die SIA übertragen, so erhält man einen flusssensitiven Algorithmus, der den Instanzgraphen erzeugen kann. Dieser Algorithmus ist im Quellcode 6.3 dargestellt[26]. Er beginnt bei einem festgelegten Programmeintrittspunkt und traversiert den ICFG in ABB-Form. Dazu führt er den aktuellen ABB als Befehlszähler im abstrakten Zustand mit. Trifft der Algorithmus auf einen CALL-ABB, folgt er diesem und führt überdies einen abstrakten Aufrufpfad mit, der einerseits für die korrekte Interpretation der Systemaufrufargumente, anderseits für den korrekten Rücksprung notwendig ist (in Analogie zu einem Aufrufstapel). Im Gegensatz zur SSE durchläuft der Algorithmus keine Schleifen und nach einer Verzweigung wieder verschmolzene Pfade mehrfach. An dieser Stelle tritt ein Widening-Operator in Kraft, der die entsprechende Instanz als „mehrfach erzeugt“ markiert (der Verständlichkeit halber im Code nicht dargestellt). Durch den viel kleineren Zustand (den singulären Instanzgraphen, anstatt vieler AbSSs) und dem singulären Durchlauf von Verzweigungen und Schleifen, ist diese Version bereits deutlich effizienter als die SSE.

*flussinsensitive Iteration*

Es gibt für die Traversierung eine noch effizientere flussinsensitive Variante, die in Quellcode 6.4 dargestellt ist. Hier wird nicht auf dem ICFG gearbeitet, sondern auf der bereits deutlich kondensierteren Form des Aufrufgraphen. ARA erzeugt diesen als Teil der Wertanalyse bereits, sodass keine Mehrkosten entstehen. Im Algorithmus werden nun alle Systemaufrufe im Aufrufgraphen durchlaufen (Knoten mit einer Kardinalität von eins), gefiltert nach SOCs oder Kommunikationsaufrufen. Anschließend werden Bottom-Up von diesem Systemaufruf ebenfalls über den Aufrufgraphen alle möglichen Aufrufpfade ermittelt, die

[26] In der tatsächlichen Implementierung verwendet ARA aus Effizienzgründen eine iterative Variante dieses Algorithmus, der zudem mit der SSE und MultiSSE geteilt wird.

```
Eingabe:
  • icfg:        Interprozeduraler CFG
  • entry_point: Zu analysierende Funktion
Ausgabe:
  • syscall:     Der nächste Systemaufruf
  • callpath:    Der zugehörige Aufrufpfad (Kontext)

def traverse_cfg_sensitive(icfg, entry_point):
    entry_abb = icfg.get_entry_abb(entry_point)
    for syscall, callpath in find_soc(icfg, entry_abb, [], false):
        yield syscall, callpath
```

```
Eingabe:
  • icfg:       Interprozeduraler CFG
  • abb:        Zu analysierender (Einstiegs)-ABB
  • callpath:   Der zugehörige Aufrufpfad (Kontext)
Ausgabe:
  • abb:        Der nächste Systemaufruf
  • callpath:   Der zugehörige Aufrufpfad (Kontext)

def find_soc(icfg, abb, callpath):
    next_abbs = icfg.get_icfg_successors(abb)

    # Behandlung von „call“ und „return“
    if abb.isCallSite:
        callpath.push(abb)
    else if abb.isReturn and callpath.empty() \
            or os_model.isSchedulerStart(abb):
        next_abbs = []
    else if abb.isReturn:
        caller_abb = callpath.pop()
        next_abbs  = icfg.get_lcfg_successors(caller_abb) or []

    # Behandlung der Systemaufrufe
    # Der Filter ist hier exemplarisch für die SIA gesetzt.
    if os_model.is_syscall(abb, filter_by=SOC):
        yield abb, callpath

    # Standardfall
    for next_abb in next_abbs:
        find_soc(icfg, next_abb, callpath)
```

**Codeblock 6.3** Kontrollflusssensitive Variante der SIA (angelehnt an [›FED+21]).

zur Parameterbestimmung notwendig sind. Diese können anschließend von der SIA weiterverwendet werden.

```
Eingabe:
  • icfg:         Interprozeduraler CFG
  • entry_point:  Zu analysierende Funktion
Ausgabe:
  • syscall:      Der nächste Systemaufruf
  • path:         Der zugehörige Aufrufpfad (Kontext)

def traverse_cfg_insensitive(callgraph, entry_point):
    # Der Filter ist hier exemplarisch für die SIA gesetzt.
    for syscall in callgraph.get_syscalls(filter_by=SOC):
        for all paths between syscall and entry:
            yield syscall, path
```

**Codeblock 6.4** Kontrollflussinsensitive Variante der SIA

Abbildung 6.2 zeigt die verschiedenen Traversierungen des Beispiels (Quellcode 6.1). Es ist zu erkennen, dass die flussinsensitive Variante hier deutlich weniger Schritte benötigt, andererseits aber die Systemaufrufe in keiner vorgegebenen Reihenfolge auswertet.

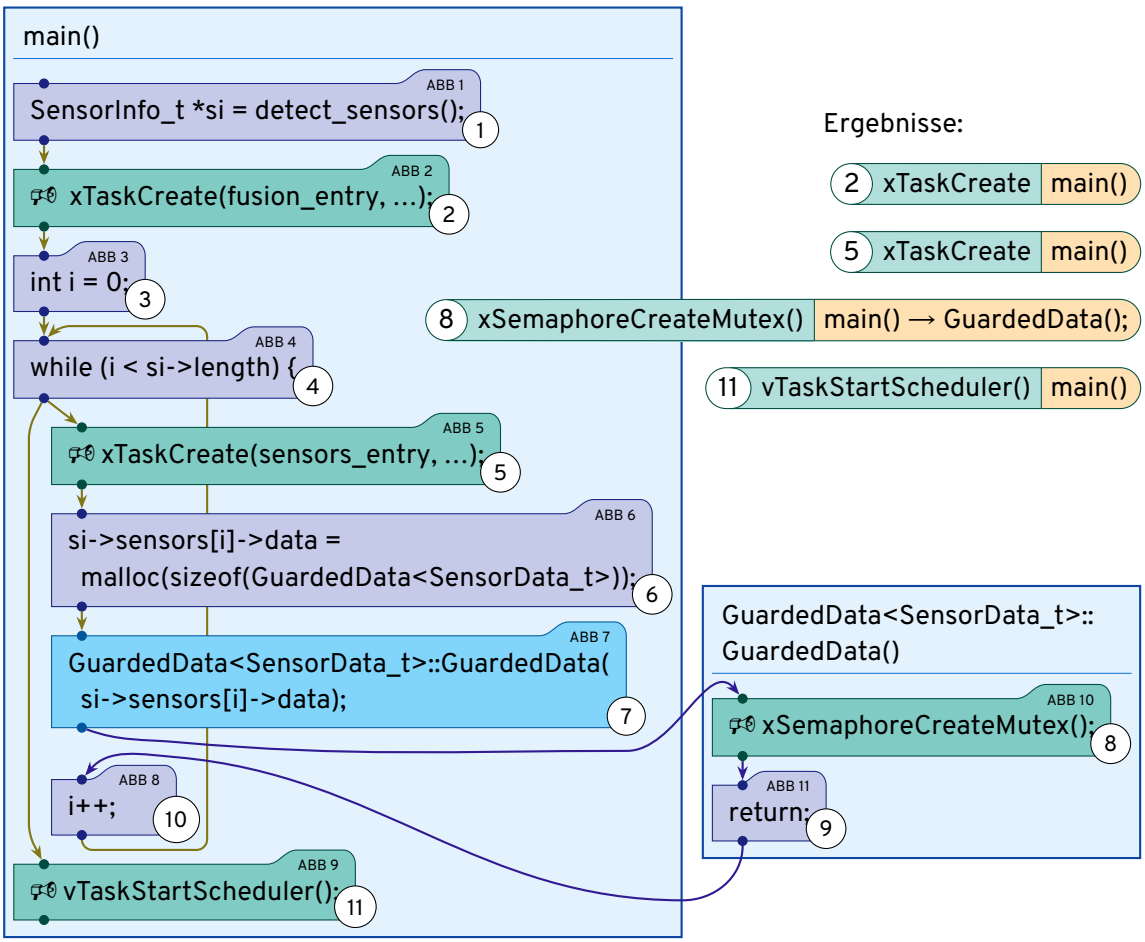
*Laufzeit beider Algorithmen*

Für die Berechnung der korrekten Systemaufrufargumente ist der Aufrufkontext essentiell. Damit ist in diesem Zusammenhang der Aufrufpfad gemeint (also die Liste an aufgerufenen Funktionen), den die Wertanalyse benutzen kann, um die Herkunft von Aufrufparametern zu verfolgen. Er ist außerdem notwendig, um zu bestimmen, ob der Systemaufruf im Kontext einer Schleife, Verzweigung oder Rekursion aufgerufen wird. Da beide Traversierungen zum Ziel haben, alle Systemaufrufe in allen Kontexten zu durchlaufen, unterscheiden sie sich in dieser Hinsicht nicht in der Laufzeit, die im schlimmsten Fall bei $O(n \cdot n!)$ liegt, wobei $n$ die Anzahl der in der Anwendung vorhandenen Funktionen darstellt. Vor allem die Anzahl an verschiedenen Aufrufpfaden im Aufrufgraphen ist hier mit $n!$ der entscheidende Faktor[27]. In eingebetteten System kann es aufgrund der Verwendung von (generischen) Bibliotheken teilweise sehr viele Funktionen geben. Die Komplexität des Aufrufgraphen liegt aber weit darunter, da sehr viele Funktionen unbenutzt sind und nicht in die Suche einbezogen werden. Der flusssensitive Algorithmus durchläuft *zusätzlich* zum flussinsensitiven Algorithmus auch noch einen jeden ABB, auch wenn dieser eine irrelevante Berechnung darstellt, und das ein jedes Mal, wenn diese Funktion durchlaufen wird. Der konkrete Mehraufwand korreliert mit der Menge der COMPUTATION-ABBs für jede Funktion, die zum Aufrufkontext beiträgt.

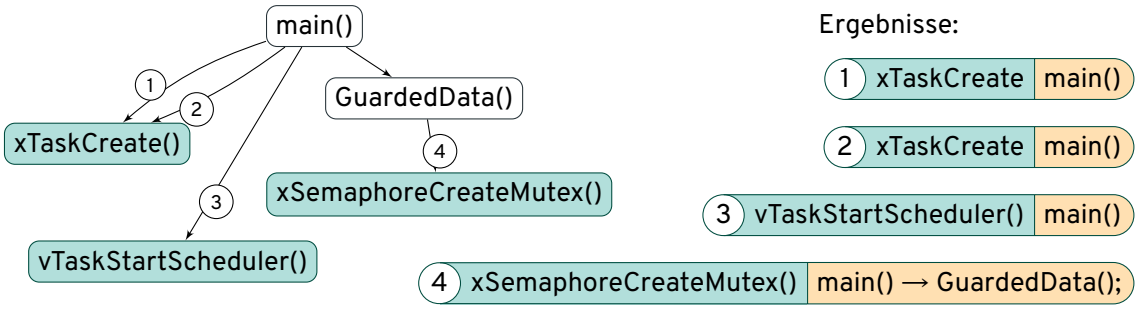
*Optimierungspotenzial*

Generell hat aber auch die flussinsensitive Variante den Nachteil, alle Aufrufpfade zu jedem Systemaufruf durchlaufen zu müssen. Bei größeren Anwendungen benötigt dieser Schritt enorm viel Zeit, weswegen es wünschenswert wäre, auch dies zu reduzieren. Die Aufrufpfade sind für zwei Informationen wichtig:

[27] Ich verwende die `all_paths`-Funktion aus der „graph-tool“-Bibliothek [Pei14] mit entsprechender Komplexität.




**a)** Flusssensitive Variante auf dem ICFG, (angelehnt an [›FED+21]).




**b)** Flussinsensitive Variante auf dem Aufrufgraphen.

**Abbildung 6.2** Visualisierung der zwei Verfahren, um den Instanzgraphen zu erzeugen. Die Abbildung zeigt beispielhaft die Traversierung der main-Funktion des Beispiels aus Quellcode 6.1. Ziel ist die Instanzbestimmung, darum werden nur SOCs betrachtet (hier alle Systemaufrufe). Die ABB-Typen sind farblich markiert: CALL, SYSCALL und COMPUTATION. Als Ergebnis werden dem überliegenden Algorithmus aus Quellcode 6.2 die Tupel aus Systemaufruf und dessen Aufrufkontext zurückgeliefert. Die Verfahrensschritte sind mit der entsprechenden Zahl (n) gekennzeichnet.

1. Die Bestimmung der Systemaufrufparameter: Die Wertanalyse ist in der Lage, Systemaufrufparametern über Funktionsgrenzen hinweg zu folgen. Dazu muss aber der Aufrufpfad bekannt sein. Da die Wertanalyse Bottom-Up funktioniert, wäre für diese aber nur so viel des Endes des Aufrufpfades (die innersten aufgerufenen Funktionen) nötig, wie Argumente über Funktionsgrenzen hinweggereicht werden, die den Systemaufruf beeinflussen. Dies macht eine enge Kopplung von Wertanalyse und Pfad-Traversierung nötig, die möglich wäre, aber in der aktuellen Implementierung nicht besteht.

2. Die Bestimmung der Eindeutigkeit: Über den Kontext wird bestimmt, ob eine Instanz eindeutig ist. Weiterhin arbeitet die SIA derart, dass jeder Kontext zu einer *neuen* Instanz führt, selbst, wenn diese mit dem gleichen Systemaufruf erzeugt wird. Dies entspricht dem realen Programmverhalten, beispielsweise, wenn Systemaufrufe mit Hilfsfunktionen umhüllt werden (z. B. per `GuardedData::GuardedData()` im laufenden Beispiel). Denkbar wäre hier aber eine Erweiterung des Widening-Operators, der als Folge eben auch noch Instanzen, die mit dem gleichen Systemaufruf erzeugt werden, kombinieren würde. Dies wäre insbesondere dann denkbar, wenn diese Instanzen bereits uneindeutig sind (also z. B. in einer Schleife erzeugt) und daher für die weitere Optimierung nicht geeignet sind. Nichtsdestotrotz schränkt ein solches Widening die Aussagekraft des Instanzgraphen weiter ein und war für die meisten der untersuchten Applikationen, die domänengemäß eher klein sind, unnötig, weswegen auf eine solche Implementierung verzichtet wurde.

```
1  int main() {
2      [...]
3      pthread_attr_t attr;
4      pthread_attr_init(&attr);
5      pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
6      struct sched_param sched_prio;
7      sched_prio.sched_priority = 5;
8      pthread_attr_setschedparam(&attr, &sched_prio);
9      pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
10     pthread_attr_setname_np(&attr, "A new Thread!");
11
12     pthread_t new_thread;
13     pthread_create(&new_thread, &attr, new_thread_routine, "some arg");
14     pthread_attr_destroy(&attr);
15     [...]
16 }
```

**Codeblock 6.5** Die Fadenerstellung unter POSIX ist nicht atomar, da Fäden über ein gesondertes Attribut spezifiziert werden.

*Nicht atomare SOCs*

Die flussinsensitive Variante ist trotz des schnelleren Durchlaufs leider nicht für jeden Fall geeignet. So gibt es SOCs, die die Instanz nicht atomar erzeugen, sondern in einem ersten

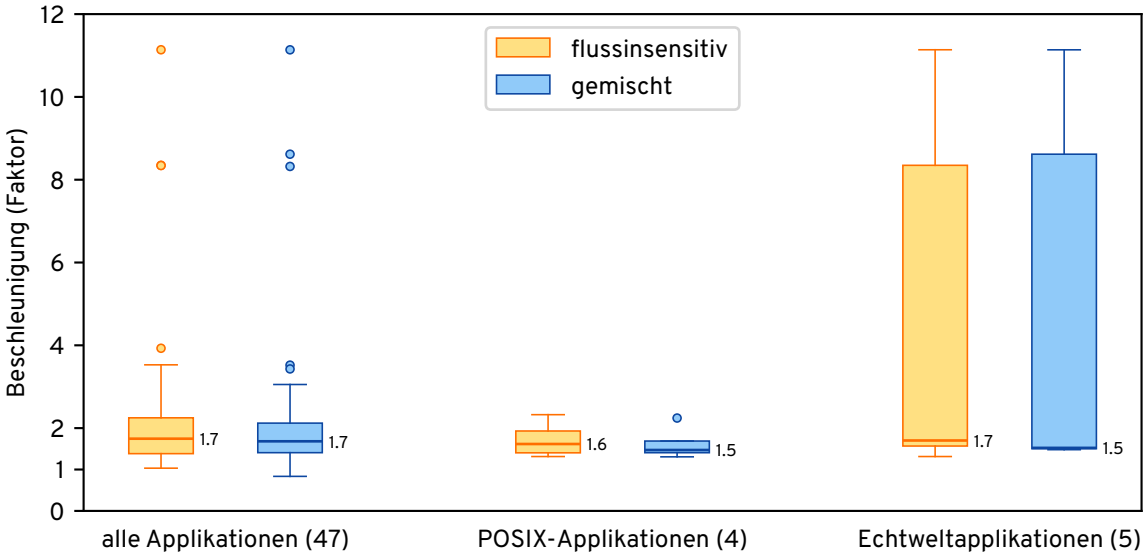



**Abbildung 6.3** Messung der Laufzeitbeschleunigung der verschieden Graphtraversierungen der SIA gegenüber dem flusssensitiven Modus. Die Testmenge besteht aus Test- und Echtweltapplikationen für ARA mit mindestens 5 Systemaufrufen. Die gemessenen Applikationen sind in drei Kategorien zusammen mit ihrer jeweiligen Anzahl sichtbar: Alle vermessenen Applikationen, nur diejenigen, die POSIX verwenden und alle Echtweltapplikationen (mit insbesondere deutlich umfangreicherem Code). Für jede Kategorie ist weiterhin die Beschleunigung des flussinsensitiven und gemischten Modus dargestellt und der Median der Beschleunigung explizit genannt. Die Analyselaufzeit für jede Applikation wurde zehnmal gemessen und gemittelt, durch eine parallele Ausführung der Messungen auf verschiedenen Kernen sind aber Scheduling-Effekte nicht auszuschließen.

Systemaufruf die Instanz allozieren, aber darauffolgend mit weiteren Systemaufrufen die Instanz konfigurieren. Dies ist abhängig vom jeweiligen RTOS und von den betrachteten Betriebssystemen besitzt nur POSIX nicht atomare SOCs. Beispielsweise werden Fadenattribute in POSIX mithilfe eines gesonderten Objektes und mehreren Systemaufrufen spezifiziert (Quellcode 6.5). In diesem Falle kann nur die flusssensitive Variante die Instanz korrekt erkennen.

*ISRs in FreeRTOS*

Die flusssensitive Variante ist wiederum nicht in der Lage, bei FreeRTOS ISRs zu erkennen. Bei diesem RTOS sind diese nicht gesondert spezifiziert, sondern müssen passend zur Hardware ohne Beteiligung des Betriebssystems erstellt werden. FreeRTOS besitzt allerdings gesonderte Systemaufrufe, die nur in ISRs aufgerufen werden können. Mit einer leichten Modifikation der flusssensitiven Variante sind damit ISRs auffindbar: Wann immer ein unterbrechungsspezifischer Systemaufruf gefunden wird, sucht der Algorithmus auf dem Aufrufgraphen selbstständig nach möglichen Einstiegspunkten und erstellt dafür im Instanzgraphen eine ISR und fährt fort wie gehabt.

*Mischform*

Um die Nachteile beider Varianten bei Beibehaltung der Vorteile zu umgehen, habe ich zusätzlich eine Mischform implementiert, bei der die Traversierung grundsätzlich flussinsensitiv läuft, aber im Falle eines nicht atomaren SOCs auf die flusssensitive Variante umschaltet[28]. Die Laufzeitverbesserungen der schnellen Varianten gegenüber der flusssensitiven Variante auf einer Vielzahl von Applikationen sind in Abbildung 6.3 zu finden.

*Laufzeitauswertung*

Es ist zu erkennen, dass die Mischform, obgleich sie alle Vorteile beinhaltet, keinen Laufzeitnachteil auf Anwendungen hat, die ausschließlich atomare SOC beinhalten. In der Gruppe der POSIX-Applikationen, bei denen die gemischte Variante in den flusssensitiven Modus umschaltet, ist eine leichte Verschlechterung der Laufzeit gegenüber der flussinsensitiven Variante erkennbar. Interessanterweise zeigt die Messung über alle Applikationen im Minimum sogar eine leichte Verschlechterung der Laufzeit im Vergleich zur flusssensitiven Variante. Das liegt am (künstlichen) Aufbau der zugehörigen Testapplikation, die hauptsächlich in einer einzigen Methode sehr viele Systemaufrufe direkt hintereinander ausführt und dadurch vom mitgeführten Kontext der flusssensitiven Variante profitiert. Bei der Vermessung von Echtweltapplikationen, die im Vergleich zum sonstigen Code deutlich weniger Systemaufrufe enthalten als die Testapplikationen, zeigt sich eine deutliche Beschleunigung der flussinsensitiven und gemischten Variante bis zu Faktor 11.1.

### 6.2.3 Metadaten: Schleifen, Verzweigungen & Rekursion

*Metadaten für das Widening*

Im SIA-Algorithmus (Quellcode 6.2) existiert eine Funktion `add_metadata`, auf die ich bislang nicht weiter eingegangen bin. Diese setzt den Widening-Operator der SIA um, der die Analyse von dynamischen Systemen erst möglich macht. Die SIA bestimmt damit die Häufigkeit des Systemaufrufs und damit speziell bei Instanzen, ob diese eindeutig und damit pseudostatisch sind. Sie erhebt dazu folgende Daten:

- Wird der fragliche Systemaufruf innerhalb einer Schleife aufgerufen? Die Anzahl der resultierenden Instanzen oder Interaktionen ist damit vollständig unklar. Der Systemaufruf kann keinmal bis beliebig oft aufgerufen werden. Die Information, ob der Systemaufruf Teil einer Schleife ist, erhält ARA vom LLVM-Framework, das diese Information für Basisblöcke bereits extrahiert. ARA überträgt die Basisblock-Informationen auf die ABB-Ebene, wobei es ABBs, die Schleifen innerhalb von BBs vollständig subsumieren, nicht als Teil einer Schleife markiert.

- Wird der fragliche Systemaufruf in einem Ast einer Verzweigung aufgerufen („nach einem ‚if"")? Die Anzahl der resultierenden Instanzen oder Interaktionen ist damit nicht

[28] Um die Laufzeitkomplexität gering zu halten, wird die Flusssensitivität ausschließlich für den Aufrufrahmen des initialen SOCs, der das Umschalten auf die flusssensitive Traversierung auslöst, aufrecht erhalten. Falls zusätzliche Systemaufrufe gefunden werden, die einen nicht atomaren SOC voraussetzen, aber in einem flussinsensitiven Kontext analysiert werden, wird entsprechend gewarnt.

```
1  #define assert(cond, msg) do { \
2    if (!cond) {                 \
3      log(ERROR, msg);           \
4      while(true) {}             \
5    }                            \
6  } while (0);
7
8  int main() {
9    bool succ = do_something();
10   assert(succ, "FAIL");
11   syscall();
12 }
```

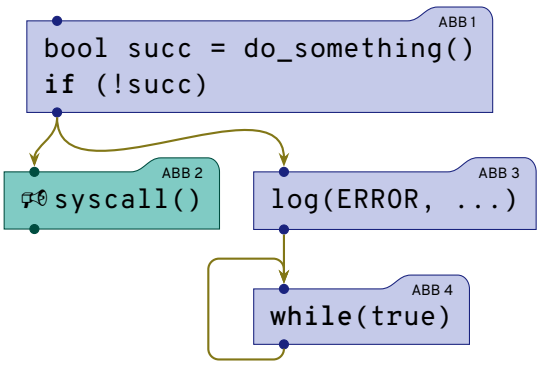



**Abbildung 6.4** Ein üblicherweise genommener Systemaufruf. Er befindet sich zwar in einem Ast einer Verzweigung, alle anderen Äste enden aber in einer Endschleife.

eindeutig, aber liegt bei null oder eins. Damit ist eine Instanz nicht pseudostatisch, aber die Synthese könnte sie trotzdem statisch erstellen zu dem Preis, dass sie u. U. zur Laufzeit überflüssig wäre. Wir haben uns bei der Synthese dagegen entschieden und nur echte pseudostatische Instanzen spezialisiert.

- Wird der fragliche Systemaufruf üblicherweise aufgerufen? Bei der Untersuchung der Echtweltprogramme haben wir als Muster im Speziellen bei Assertions festgestellt, dass diese in Endschleifen enden. Die Anwendung verzweigt also in einen Zustand, in dem sie gefangen ist. Abbildung 6.4 zeigt ein Beispiel eines solchen Code. Dies heißt aber auch zwangsläufig für alle nachfolgenden Systemaufrufe, dass sie sich in einem Ast einer Verzweigung befinden und damit nicht mehr eindeutig sind. Mit der Unterstellung, dass Anwendungsentwickler den Programmfluss nur in sehr seltenen Fällen und auf jeden Fall bewusst in Endschleifen treiben, ignoriert ARA in diesem Fall die Verzweigung und nimmt die Instanz oder die Interaktion trotzdem als „eindeutig“ an.

- Wird der fragliche Systemaufruf rekursiv aufgerufen? In diesem Fall ist die Anzahl der Aufrufe ohne weiteres nicht bestimmbar und die Instanz oder Interaktion uneindeutig und ähnelt damit einer Schleife. Rekursion gilt in Echtzeit- und eingebetteten Systemen für gewöhnlich zu vermeiden [KKZ12], da die Größe des Stapels von der Tiefe der Rekursion abhängt und damit dynamisch wird (abgesehen von Endrekursionen, die gut optimierbar sind [Ste77]). Analyseprogramme verbieten Rekursion wie dOSEK oder unterstützen diese erst seit kurzen wie Astreé (seit 2020). ARA kann explizit mit Rekursion umgehen, markiert die entsprechenden Stellen aber als nicht optimierbar. Die Bestimmung, ob eine Funktion rekursiv ist, erfolgt bei ARA über den Aufrufgraphen, da der entsprechende Knoten dort in diesem Fall Teil einer Schleife ist.

*Widening: Methodik*

Sowohl die Erkennung davon, ob Systemaufrufe üblicherweise aufgerufen werden, als auch die von Verzweigungen erfolgt bei ARA über Dominatoren: Ein ABB, der die Funktion

verlässt, heißt *Ausgang*. Eine Endschleife ist ebenfalls ein Ausgang. Ein Systemaufruf ist *bedingt*, wenn er sich innerhalb eines Astes einer Verzweigung befindet. Dies ist der Fall, wenn es mindestens einen Ausgang gibt, den er nicht dominiert. Ein Systemaufruf wird *üblicherweise* aufgerufen, wenn er dann bedingt ist, wenn Endschleifen als Ausgänge interpretiert werden und nicht bedingt ist, wenn Endschleifen keine Ausgänge darstellen.

ARA markiert anschließend einen Systemaufruf bei folgenden Bedingungen als eindeutig: Er in nicht Teil einer Schleife. Er wird nicht in einem rekursiven Kontext aufgerufen. Er wird nicht innerhalb eines Astes einer Bedingung aufgerufen, aber nur, wenn dieser Aufruf nicht üblicherweise ist. Diese Markierung erfolgt nicht nur über den Systemaufruf, sondern ebenso über alle Aufrufe innerhalb des Aufrufkontexts. Sobald einer dieser Aufrufe Teil einer Schleife oder Verzweigung ist, ist der Systemaufruf uneindeutig.

Es gibt noch einen weiteren Faktor zur Eindeutigkeit: Die Instanzerstellung vor oder nach dem Planerstart (also in einem Fadenkontext). Instanzen, die in Fäden angelegt werden, könnten uneindeutig sein, wenn der Faden mehrfach gestartet wird. Dies hängt von der konkreten RTOS-Semantik ab (FreeRTOS startet Fäden beispielsweise nur einmal, AUTOSAR kann Tasks mehrfach starten). Die SIA extrahiert diese Information darum zusätzlich, indem sie anhand des Einstiegspunkt entscheidet, ob sie sich in einem Fadenkontext befindet.

## 6.3 Synthese

Björn Fiedler hat eine auf der SIA aufbauende Synthese entwickelt, die die Optimierung umsetzt. Ich werde sie hier dennoch kurz erläutern, da sie essentiell für die folgende Evaluation des Gesamtverfahrens ist, die die Analyse validiert. Die Details sind in seiner Dissertation zu finden [Fie23A4.3]. Die Synthese konzentriert sich speziell auf den Extremfall des sehr dynamischen FreeRTOS und implementiert sowohl das Umwandeln von pseudostatischen in echt statische Instanzen als auch die Ersetzung von Single-Reader-Single-Writer-Warteschlangen mit einer effizienten Implementierung. Warteschlangen sind für FreeRTOS besonders relevant, da auch Semaphoren und Mutexe in FreeRTOS intern auf (synchronisierten) Warteschlangen basieren, was das Konzept dort zum zentralen Synchronisationsobjekt macht. Ich werde zuerst die Umwandlung von pseudostatischen in echt statische Instanzen und anschließend die Interaktionsspezialisierung erläutern.

### 6.3.1 Instanzsynthese

Für die Umwandlung der Instanzen genügen die Knoten des Instanzgraphen, sodass die SIA als Analyse ausreicht. Auf Basis des Instanzgraphen wird anschließend zuerst die folgende Kategorisierung der Instanzen vorgenommen, die deren statischen Grad bestimmt:

- Die Instanz ist vollständig in eine statische Instanz umwandelbar. Die Instanz ist damit eine pseudostatische Instanz.
- Die Instanz ist teilstatisch erstellbar. Insbesondere die Reservierung des Speichers kann statisch erfolgend, aber einige Parameter müssen dynamisch eingestellt werden.
- Die Instanz ist nur dynamisch erstellbar. Insbesondere die Häufigkeit der Instanz bzw. deren Existenz ist (statisch) unbekannt.

*Phasen der Spezialisierung*

Die Klassifizierung lässt sich auf verschiedene Phasen des Intialisierungsprozesses abbilden, die von der Synthese für statische Instanzen entsprechend modifiziert werden müssen:

1. **Speicherallokation:** In dieser Phase wird der Speicher für die Instanzen reserviert. Diese Maßnahme verringert insbesondere die Benutzung einer dynamischen Speicherverwaltung, die aufwändig in der Verwaltung ist und zudem die Wahrscheinlichkeit von resilienten Fehlern erhöht. Sind die pseudostatischen Instanzen sogar ausschließliche Nutzer der Halde, kann diese durch eine anschließende Optimierung vollständig entfallen. Sowohl der Speicher von vollständig statisch als auch der von teilstatisch erstellbar Instanzen muss in dieser Phase alloziert werden.

2. **Initialisierung:** In dieser Phase wird der allozierte Speicher mit sinnvollen Werten befüllt. Dies können alle Parameter als auch eine Untermenge sein, wobei es sich dann um eine teilstatisch erstellbare Instanz handelt. Bei Aktivitäten gehört in diese Phase z. B. das derartige initiale Schreiben des Aufrufrahmens, dass die Aktivität bei Einlastung mit der gewünschten Routine startet. Bei vollständig statisch erstellbaren Instanzen findet in dieser Phase nach der Optimierung keine Aktion mehr statt. Teilstatisch erstellbare Instanzen führen hier den noch notwendigen Teil der Optimierung durch.

3. **Registrierung im Betriebssystem:** Die fertig initialisierten Instanzen müssen anschließend dem Betriebssystem bekannt gemacht werden. Bei FreeRTOS beinhaltet dies z. B. das korrekte Verketten in den Bereitlisten, sodass vollständig statisch erstellbare Instanzen statisch im Betriebssystem registriert sind. Hierfür muss auch der Ablaufplaner derart modifiziert werden, dass er mit vorinitialisierten Datenstrukturen beginnt.

4. **Seiteneffekte und Rückgabewert:** Systemaufrufe können direkte Rückgabewerte und überdies einen Seiteneffekt auf andere Speicherbereiche haben. In FreeRTOS wird der Fadenkontrollblock z. B. optional einem dem Systemaufruf übergebenen Parameter zugewiesen. Bei vormals dynamischen Instanzen werden diese Effekte durch eine geeignete Ersetzung des ursprünglichen Systemaufrufs erzeugt. Dieser liefert beispielsweise das erfolgreiche Erstellen der Instanz zurück.

```
- TaskHandle_t fusion_task;
+ TaskHandle_t fusion_task = &fusion_task_tcb;
+ char fusion_task_stack[512];
+ TCB_t fusion_task_tcb = {.prio=1, .stack=&fusion_task_stack, [...]};

- xTaskCreate(fusion_entry, "fusion", 512, si, 1, &fusion_task);
+ set_initial_parameter(&fusion_task_stack, si);
```

**Codeblock 6.6** Spezialisierung des Fadens *fusion* (schematisch in äquivalentem C-Code), aus [Fie23A4.2.2]. Gut erkennbar ist die Aussparung des SOCs.

Die vorgenommenen Änderungen erfolgen durch einen Generator innerhalb des ARA-Frameworks. Dieser erzeugt zum einen eine zusätzliche Datei für den zusätzlichen Initialisierungscode, die beispielsweise bei FreeRTOS die Allokation und Initialisierung in Form von Konstruktoren beinhaltet. Zum anderen modifiziert sie direkt den Anwendungscode, um beispielsweise Systemaufrufe auszutauschen. Abbildung 6.6 zeigt schematisch die vorgenommenen Änderungen für den *fusion*-Faden aus dem Beispiel.

*Dünn besetzter Speicher*

In frühen Tests hat sich ein weiteres Problem herausgestellt: Ersetzt man den vormals dynamischen Speicher (ein in der Initialisierungsphase allozierter Speicher, der mit Nullen befüllt und damit im BSS-Segment liegt) durch statisch *vorinitialisierten* Speicher, so ist dieser dünn mit Werten ungleich Null besetzt. Beispielsweise besteht im Fadenkontrollblock der Stapelspeicher weiterhin hauptsächlich aus Nullen bis auf die ersten 16 Bytes, die die notwendigen Werte erhalten, damit der Faden an der richtigen Code-Position seine Ausführung beginnt. Der Übersetzer nimmt dies aber zum Anlass, den gesamten Block in das Datensegment zu legen (inklusive der Nullen), was nicht nur das Systemabbild vergrößert, sondern auch dem eigentlichen Sinn konträr die Initialisierungsphase verlängert: Im Gegensatz zum BSS-Segment, bei dem der Initialisierungscode des Systems $n$ Nullen in den RAM schreibt, aber bis auf den Wert $n$ keine Werte aus dem Flash liest, handelt es sich bei einer Initialisierung aus dem Datensegment um eine Kopie: Der Initialisierungscode liest $n$-mal die Null aus dem Flash *und* schreibt sie in den RAM. Das Lesen aus dem Flash ist dabei nicht nur überflüssig, sondern auch noch besonders langsam.

*Lauflängencodierung*

Fiedler ist dieses Problem durch einen weiteren Übersetzerschritt angegangen, der zum einen eine *Lauflängencodierung (Run-Length Enconding, RLE)* auf den Speicher im Datensegment anwendet und zum anderen eine zusätzliche Routine im Initialisierungscode einfügt, der diese Lauflängencodierung wieder decodiert. Durch diesen Zwischenschritt wird wie beim BSS-Segment das aufwendige Lesen aus dem Flash reduziert und dünn besetzte Daten auch im Datensegment sinnvoll gespeichert.

### 6.3.2 Interaktionssynthese

Während bei der Instanzsynthese die Initialisierungsphase des Systems verbessert wird, geht es bei der Interaktionssynthese darum, die reguläre Laufzeit des Systems zu verbessern.

```
1  def enqueue_wrapper(queue, element):
2      do_some_checks(element)
3      enqueue(queue, element)
4
5  def task1():
6      enqueue_wrapper(special_queue, element)
7
8  def task2():
9      enqueue_wrapper(generic_queue, element)
```

**Codeblock 6.7** Doppeldeutigkeit einer Interaktion. Die *special_queue* kann spezialisiert werden, die *generic_queue* nicht. Bei einer Spezialisierung müsste enqueue zwei Implementierungen erhalten: die generische sowie die spezialisierte Variante.

Fiedler zeigt die Spezialisierung am Beispiel von Warteschlangen [Fie23A5]. Er misst dazu die Laufzeiten von verschiedenen Warteschlangenimplementierungen, um Kosten und Nutzen beurteilen zu können. Konkret vergleicht er die in FreeRTOS eingebaute *Queue* (blockorientiert) mit dem ebenfalls eingebauten *Streambuffer* (stromorientiert) und zwei Eigenimplementierungen mit weicher Synchronisation für den Single-Reader/Single-Writer-Fall in einer blockierenden und nicht blockierenden Version. Der *Streambuffer* war in allen Messungen signifikant langsamer als die *Queue* (um den Faktor 1,5 beim Einreihen und 2,2 beim Entnehmen), wohingegen die Eigenimplementierungen in beiden Fällen schneller waren (um den Faktor 0,6 beim Einreihen sowie beim Entnehmen). Dies steht einem höheren Speicherverbrauch gegenüber, da der Code der spezialisierten Warteschlange zusätzlich mit eingebaut werden muss (688 Bytes für die nicht blockierende Variante, sowie 1 632 Bytes für die wartende Variante, vgl. einen Bedarf von 2 464 Bytes für die FreeRTOS-*Queue*, die ggf. entfernt werden kann).

*Doppeldeutigkeit im Aufruf*

Nun muss ARA feststellen, welche Interaktionen überhaupt spezialisierbar sind. Im Fall der Single-Reader/Single-Writer-Warteschlange braucht es dazu die Information einer korrekten Analyse, dass die Warteschlange wirklich nur von einem Leser und einen Schreiber benutzt wird. Fiedler benutzt hierfür die Kanten des Instanzgraphen, um potentiell spezialisierbare Interaktionen zu erstellen, hat aber zusätzlich eine weitergehende Klassifizierung treffen müssen: Stellt die Analyse fest, dass eine Warteschlange nur von einem Faden beschrieben und aus einem Faden gelesen wird, kann die Synthese diese Warteschlange prinzipiell mit einer spezialisierten Variante ersetzen. Der Aufrufkontext kommt dem aber in die Quere, da der Wechsel der Implementierung zwangsläufig über einen Austausch der Systemaufrufimplementierung an dieser Stelle erfolgen muss. Wird diese Stelle aber von mehreren Kontexten aus angesteuert, können verschiedene Instanzen damit angesprochen werden (Quellcode 6.7), die dann aber alle die gleiche Spezialisierung erfahren würden. Es gibt prinzipiell drei Möglichkeiten dem zu begegnen:

1. Ein Trampolin oder eine Art dynamischen Dispatch einführen, das zur Laufzeit abhängig vom Aufrufkontext die richtige Implementierung wählt. Dies resultiert in höheren Laufzeitkosten.

2. Den fraglichen Aufrufcode duplizieren, sodass der Systemaufruf immer genau einem Interaktionsobjekt fest zugeordnet ist. Bei tiefen Aufrufbäumen kann dies sehr viel zusätzlichen Code erzeugen.

3. Nur eindeutige Systemaufrufe spezialisieren.

Für alle diese Varianten braucht es eine Zusatzanalyse, um die fraglichen Stellen zu finden. Konkret ermittelt Fiedler darin pro Aufruf, ob dieser nur Instanzen anspricht, die den gleichen Grad an Spezialisierung vertragen. In der Synthese von ARA wird neben der Zusatzanalyse dann gemäß der dritten Variante spezialisiert.

## 6.4 Evaluation

Die SIA wurde zusammen mit der zugehörigen Synthese mit zwei Echtweltprogrammen für FreeRTOS, Microbenchmarks und kleineren selbst entworfenen Integrationstests evaluiert. Die Analysen alleine wurden darüber hinaus mit einer Vielzahl von weiteren Echtweltanwendungen für andere Betriebssysteme validiert, auf die ich erst im Kapitel 8 detaillierter eingehe. Die Evaluation mit der Synthese zusammen soll dabei nicht nur die Frage beantworten, ob das Verfahren funktioniert, sondern auch, ob und wie viel Verbesserungen die umgesetzten Spezialisierungen bringen. Ich werde hier zuerst die Microbenchmarks vorstellen, die vor allem dazu da sind, das Potential unserer Optimierung auszuloten. Anschließend gehe ich auf die Optimierung zweier Echtweltprogramme ein und diskutiere die Ergebnisse.

### 6.4.1 Messaufbau

Für alle unsere Tests haben wir jeweils eine FreeRTOS-Anwendung mithilfe von ARA gebaut, installiert und vermessen. Die SIA wurde dabei in jedem Fall ausgeführt; die anschließende Synthese aber parametrisiert: Im ersten Fall wird sie nicht durchgeführt, um einen Referenzwert zu erhalten: Da die Analyse nichts verändert, ist dieser Fall äquivalent zu einer Übersetzung ohne ARA. Im zweiten Fall wird bei der Synthese nur teilstatisch spezialisiert, sodass zwar die Speicherallokation statisch erfolgt, aber alle weiteren Schritte dynamisch. Im letzten Fall werden alle Instanzen so stark spezialisiert wie möglich.

Am Ende der Synthese generiert ARA zusammen mit weiteren Kompilierschritten ein Systemabbild, das installiert werden kann. Dafür haben wir die STM32 Nucleo-F103RB-Entwicklungsplatine benutzt, die einen STM32F103-Mikrocontroller beinhaltet (mit einem

ARM© Cortex©-M3 @72 MHz, 128 kB Flash, 20 kB SRAM). Auf der Plattform existiert ein Zyklenzähler, der in den ersten zwei Instruktionen nach Systemstart aktiviert und nach jeder Phase ausgelesen wird (Eigendauer: 10 Zyklen). Die Messungen sind alle mit 100 Wiederholungen ausgeführt. Die Standardabweichung betrug jeweils unter 30 Zyklen und ist deswegen in den Graphen nicht erkennbar. Zusätzlich haben wir den Speicherverbrauch bestimmt. Dies beinhaltet die verschiedenen Segmente im Systemabbild (im Flash) sowie die Segmente im RAM. Für die Interaktionsspezialisierung wird zusätzlich auf allen Programmen weiterhin die INA ausgeführt und das Abbild entsprechend spezialisiert. Für diese Arbeit habe ich die entsprechenden Messungen bis auf den LibrePilot (bei dem die Synthese durch eine neuere Compilerversion in der bisherigen Form fehlschlug) reproduziert. Im Vergleich zu den Messungen aus [›FED+21] fällt dabei die durchgängige Beschleunigung der unveränderten und teilstatischen Variante um etwa 2000 Takte auf. Dies könnte auf den mehrere Versionen neueren Übersetzer und einer dementsprechenden besseren Optimierung zurückzuführen sein (LLVM Version 9 im Vergleich zu Version 14). Die Zyklenzahl der vollstatischen Varianten ist im Gegensatz nahezu gleichgeblieben.

### 6.4.2 Microbenchmarks

```
1  int main() {
2      STORE_TIME_MARKER(main_start);
3      InitBoard();
4      STORE_TIME_MARKER(done_InitBoard);
5      kout.init();
6
7      STORE_TIME_MARKER(begin_queueCreate);
8      queue001 = xQueueCreate(4, sizeof(char));
9      queue002 = xQueueCreate(4, sizeof(char));
10     queue003 = xQueueCreate(4, sizeof(char));
11     [...]
12     queue099 = xQueueCreate(4, sizeof(char));
13     queue100 = xQueueCreate(4, sizeof(char));
14     STORE_TIME_MARKER(done_queueCreate);
15
16     vTaskStartScheduler();
17 }
```

**Codeblock 6.8** Microbenchmark, um den Nutzen der Umwandlung von dynamischen Warteschlangen in statische Warteschlangen zu messen (aus [Fie23A4.4.1]).

Verschiedene Microbenchmarks sollen das allgemeine Optimierungspotential ausloten. Dazu hat Fiedler einen Satz an Anwendungen geschrieben, die für die Synthese in einer Vielzahl an voll spezialisierbaren Systemaufrufen resultieren. Quellcode 6.8 zeigt einen solchen Testfall. Mit der Kombination aus SIA und der Synthese ergeben sich auf diesen Testfällen die Ergebnisse, die in Abbildung 6.5, Abbildung 6.6 und Abbildung 6.7 dargestellt

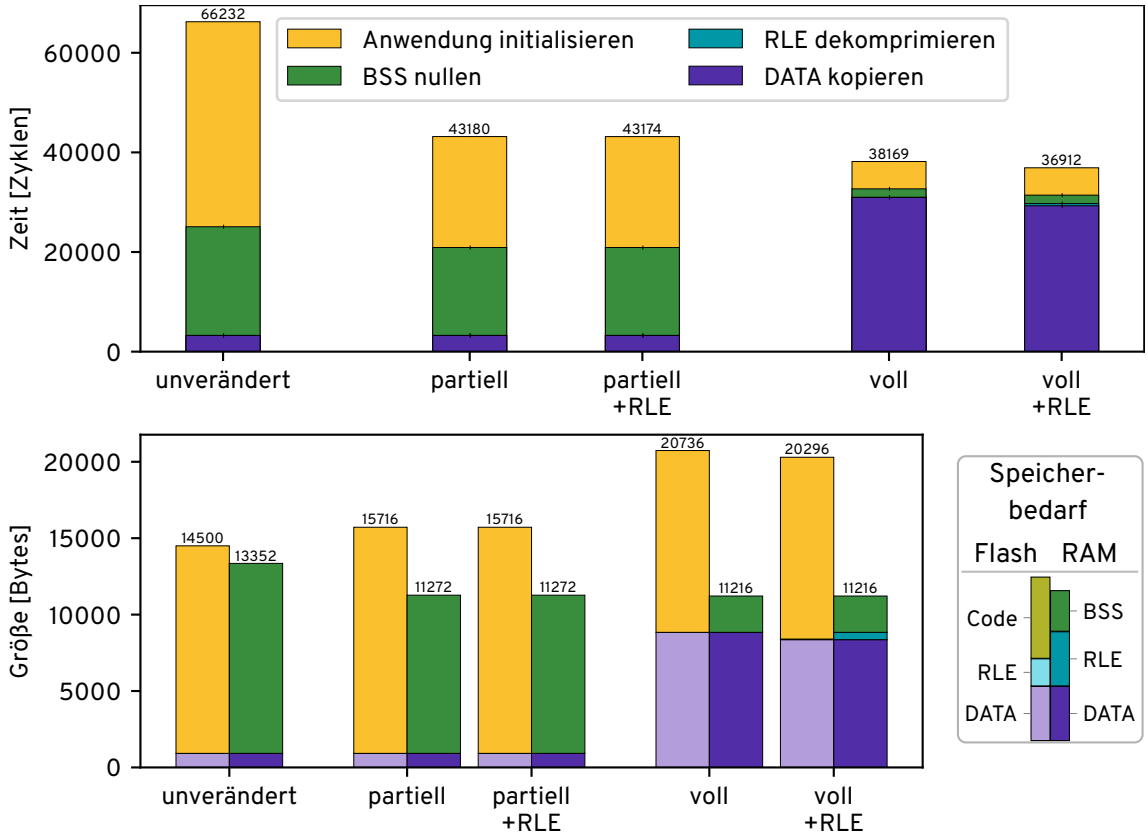



**Abbildung 6.5** Ergebnisse für die statische Warteschlangeninstanziierung, adaptiert aus [›FED+21], erneut gemessen.

sind. Hier ist eine durchschnittliche Verringerung der Laufzeit um 49.8% zu erkennen, sowie eine maximale Reduktion von 54.6%. Es ist durchgängig zu erkennen, dass bereits die partielle Spezialisierung einen Laufzeitgewinn bringt, die volle Spezialisierung aber noch einmal deutlich besser ist. Bei den Fadentests ist der Vorteil der RLE klar ersichtlich. Der Speicherverbrauch steigt in der statischen Variante moderat und wird vor allem durch die RLE wieder ausgeglichen. Durch die Systemaufrufdichte der Anwendungen ist das Benchmark nicht mit einem Echtweltprogramm vergleichbar. Sie verkörpert vielmehr das maximale Optimierungspotential, das durch unsere Optimierung möglich ist.

### 6.4.3 LibrePilot

Der LibrePilot-Quadrocopter ist eine Open-Source-Anwendung[29], die eine allgemeine Steuerungssoftware für Quadrocopter implementiert. Er besteht aus 2 673 Funktionen

[29] `https://www.librepilot.org`, Version 16.0.9

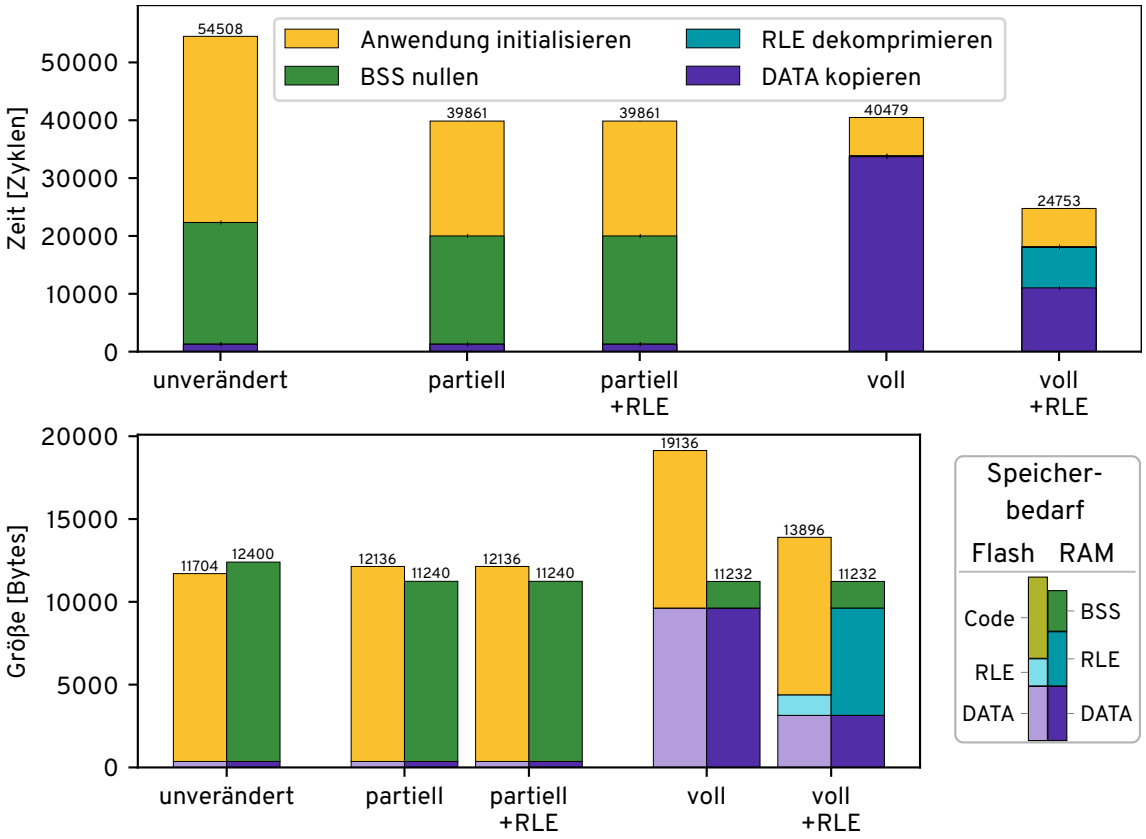



**Abbildung 6.6** Ergebnisse für die statische Fadeninstanziierung, die vor dem Planerstart angelegt werden. Adaptiert aus [›FED+21], erneut gemessen.

mit 17 265 Basisblöcken und 78 787 Codezeilen. Die Anwendung besitzt 2 882 normale Funktionsaufrufe und 223 Systemaufrufe. 33 dieser Aufrufe sind SOCs. Die maximale Länge eines Aufrufpfads zu einem SOC ist 8. Für die Initialisierungsphase iteriert der LibrePilot über eine eigens dafür definierte Sektion mit Funktionszeigern, die mithilfe eines Linker-Skripts gesammelt werden. Dieses zusätzliche Wissen, das nur durch eine Interpretation des Linker-Skripts gewinnbar wäre, erhält ARA durch eine entsprechende manuelle Annotation.

Damit findet ARA 17 (inklusive aller möglichen Kontexte) Fäden erstellende sowie 24 Warteschlangen erstellende SOCs. Von diesen können fünf Fäden vollständig, sowie zwei Fäden teilweise spezialisiert werden. Die Warteschlangen wurden alle als konditional kreiert entdeckt und waren damit nicht spezialisierbar. Abbildung 6.8 zeigt die Ergebnisse der Messung. Es ist zu erkennen, dass der LibrePilot mithilfe der Spezialisierung um 7 592 Zyklen schneller startet, allerdings ist die Verbesserung im Vergleich zur Gesamtstartzeit

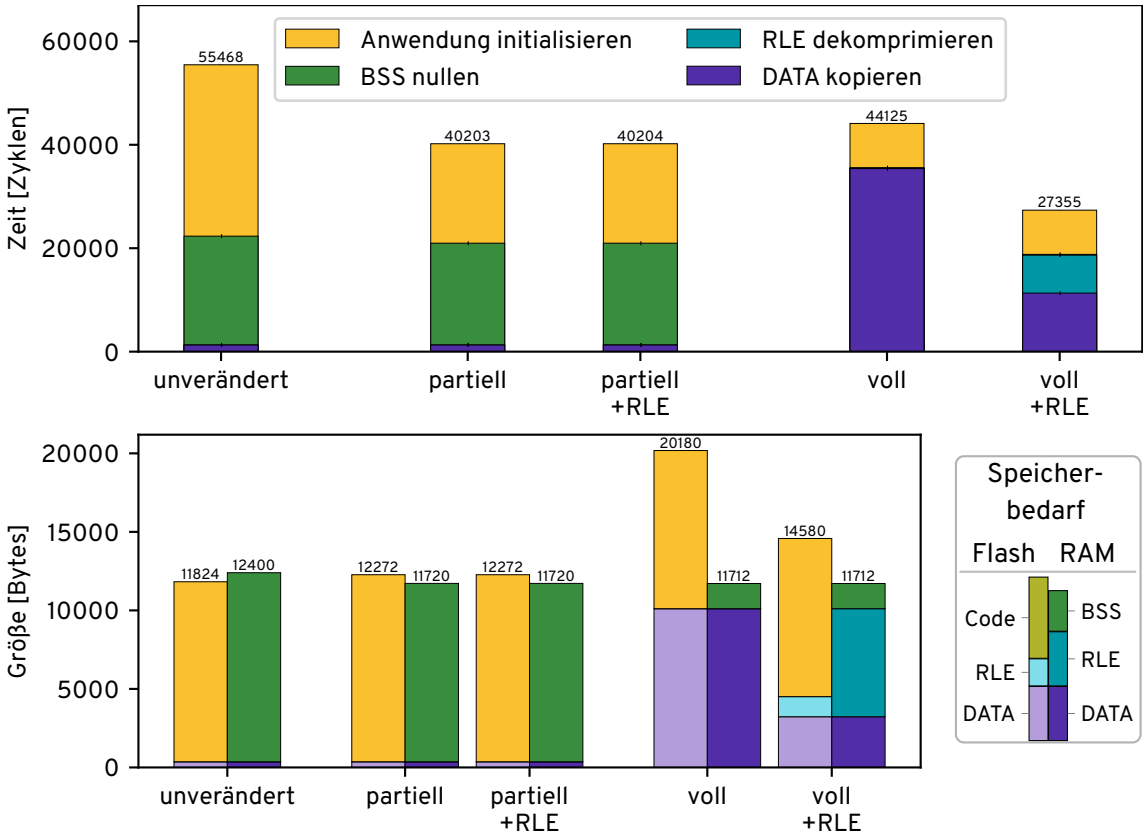



**Abbildung 6.7** Ergebnisse für die statische Fadeninstanziierung, die nach dem Planerstart angelegt werden. Adaptiert aus [›FED+21], erneut gemessen.

sehr gering. Eine manuelle Analyse hat ergeben, dass dies vor allem an weiteren überflüssigen dynamischen Berechnungen liegt. So initialisiert der LibrePilot Daten, indem er diese aus einem anderen statischen Speicher kopiert. Weiterhin alloziert der LibrePilot immer gleichen dynamischen Speicher für die Anwendung, gibt diesen aber nie frei (die gewählte Halden-Implementierung besitzt nicht einmal diese Operation). Beide Operationen wären mit geeigneter Programmierung vollständig statisch ausführbar. Der LibrePilot ist also nicht nur auf Systemebene, sondern auch auf Anwendungsebene unnötig dynamisch. Es lässt sich weiterhin sagen, dass die vorgenommene Optimierung keine Nachteile hat. Da sie zudem nicht destruktiv ist, also das RTOS die gleiche dynamische Funktion bietet wie ohne Optimierung, und die betriebssystemgewahre Analyse und Spezialisierung zusätzlich vollautomatisch stattfindet, kann sie also genau wie die bereits vorhandenen Optimierungen des Übersetzers „mitlaufen“.

Hinsichtlich der Interaktionsspezialisierung bietet der LibrePilot leider keine geeigneten Datenstrukturen, die spezialisiert werden könnten. Dies liegt vor allem an der für ARA unein-

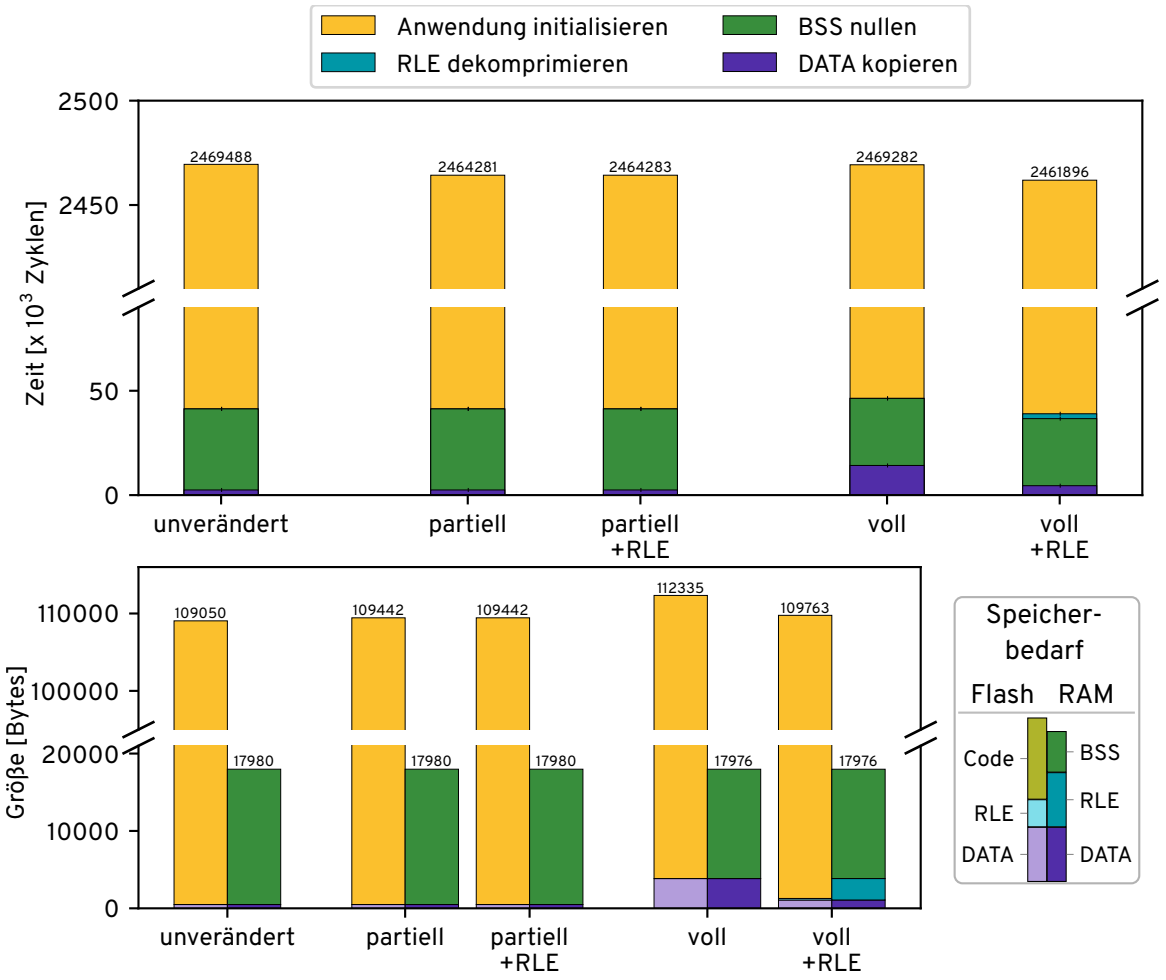



**Abbildung 6.8** Ergebnisse der Optimierung auf den LibrePilot. Adaptiert aus [›FED+21].

deutigen Zuordnung zwischen dem Zeiger, der als Argument verwendet wird und dessen Ziel, der Instanz. Da diese in größeren (teilweise dynamisch allozierten) Kontextobjekten gespeichert werden, die ihrerseits herumgereicht werden, gelingt keine eindeutige Zuordnung. Konkret kann innerhalb der INA die Wertanalyse die Argumente des Kommunikationsaufrufs nicht korrekt bestimmen. Um dieses Problem zu umgehen, wäre eine künstliche Beschränkung für ARA möglich, die für Instanzen globale Variablen voraussetzt, damit aber den LibrePilot ohne Adaption von der Analyse ausschlösse. Diese Beschränkung habe ich bewusst nicht gemacht, sondern die Analyse pessimistisch, aber korrekt angelegt. Somit werden im Zweifel nur einige Stellen nicht optimiert, andere Interaktionen können aber weiterhin gefunden und spezialisiert werden.

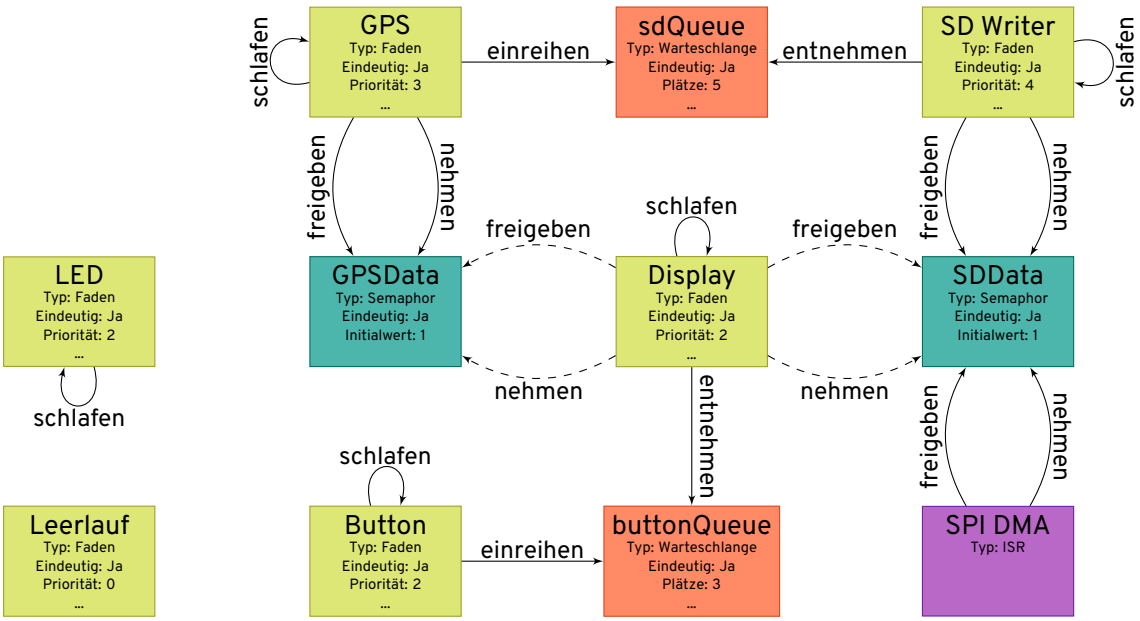



**Abbildung 6.9** Instanzgraph des GPSLogger. Erkennbar sind sechs Fäden, zwei Warteschlangen, zwei Semaphoren und eine ISR. Adaptiert aus [›ESD19].

## 6.4.4 GPSLogger

Der GPSLogger ist eine auf FreeRTOS basierende Anwendung, die GPS-Daten empfängt, darstellt und speichert[30] (ein quelloffener GPS-Tracker). Das System besteht aus einem Microcontroller, der mit einem Display (über $I^2C$), einem GPS-Empfänger (UART), einer SD-Karte (SPI) und zwei Knöpfen (GPIO) verbunden ist. Die Anwendung besteht aus 1 117 Funktionen mit 9 417 Basisblöcken, die durch 78 743 Codezeilen gebildet werden. Sie besitzt 2 028 Funktionsaufrufe und 28 Systemaufrufe, von denen 10 SOCs sind (maximale Aufrufpfadlänge zu einem SOC ist drei). Die Anwendung erstellt alle Systemobjekte vor dem Planerstart, weswegen wir diesen als SSP benutzt haben.

Der detektierte Instanzgraph des GPSLoggers ist in Abbildung 6.9 dargestellt. Er besteht aus sechs Fäden, zwei Warteschlangen, zwei Semaphoren und einer ISR. Alle Instanzen des GPSLoggers sind voll spezialisierbar[31]. Abbildung 6.10 zeigt die Messergebnisse bezüglich der Instanzspezialisierung. Hier ist eine deutliche Beschleunigung der Initialisierungsphase (44.2%) bei quasi gleichbleibendem Speicherverbrauch zu erkennen. Hierbei fällt auf, dass fast das gesamte BSS-Segment wegfällt. Dies resultiert hauptsächlich daraus, dass ARA die Halde verwerfen kann, da sämtliche Instanzen nun statisch initialisiert werden und dynamisch allozierbarer Speicher (der durch die entsprechende Überabschätzung

[30] `https://github.com/grafalex82/GPSLogger`, Git commit: 8808b922
[31] Da die ISR nicht aktiv von FreeRTOS verwaltet wird, ist sie weder dynamisch noch spezialisierbar.

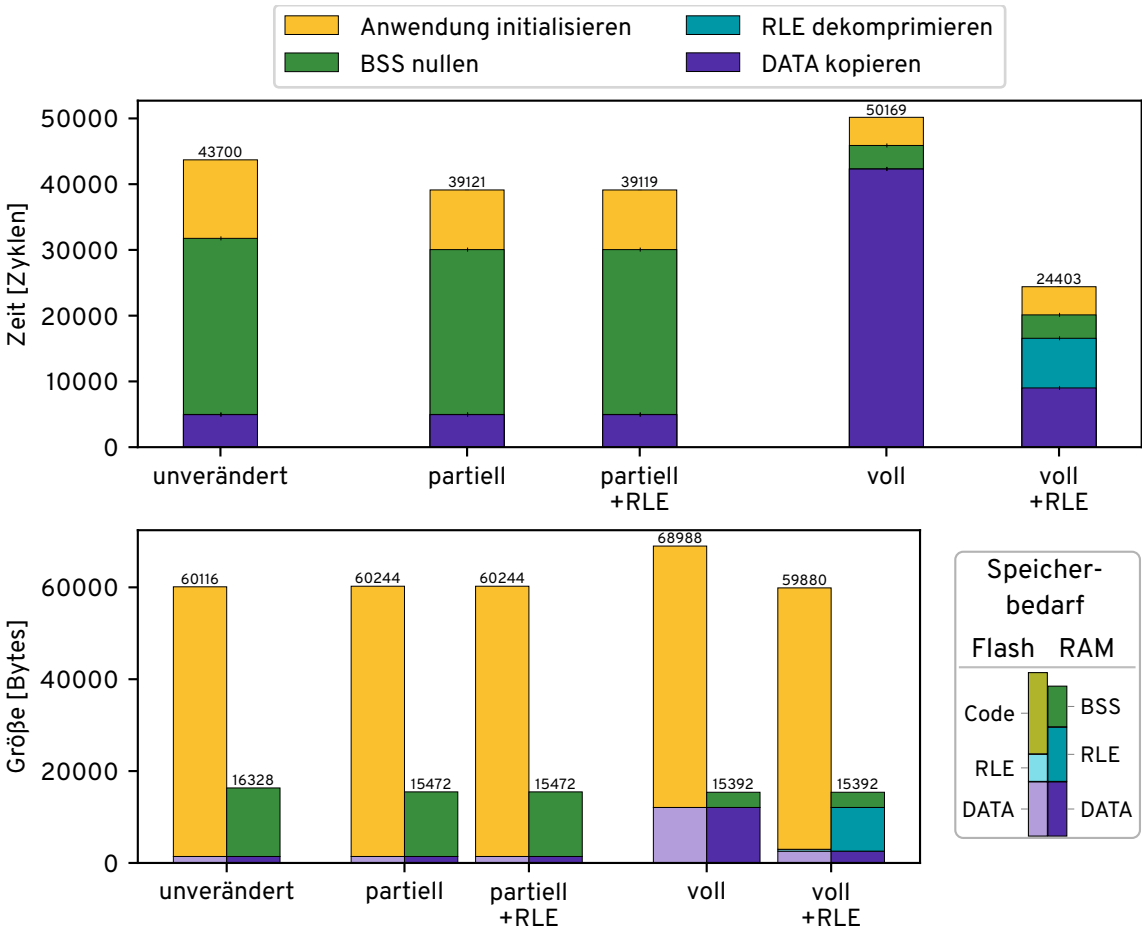



**Abbildung 6.10** Ergebnisse der Optimierung auf den GPSLogger. Adaptiert aus [›FED+21], neu gemessen.

auch noch zu groß für den benötigten Zweck dimensioniert war) damit nicht mehr nötig ist. Weiterhin ist hier der zusätzliche Sinn der Lauflängencodierung zu erkennen, die das Muster eines dünn besetzten Speichers effizient abbilden kann.

Wie aus dem Instanzgraphen erkennbar ist, besitzt der GPSLogger zwei Warteschlangen (die *sdQueue* und die *buttonQueue*), die beide jeweils von einem Faden beschrieben und von einem anderen Faden gelesen werden. Somit sind beide Warteschlangen für die Interaktionsspezialisierung geeignet. Bei den Semaphoren schlägt erneut bedingt durch die Wertanalyse die eindeutige Zuordnung der Kommunikationsaufrufe zwischen dem *Display*-Faden und der Semaphore fehl, sodass diese nicht spezialisierbar sind.

## 6.5 Diskussion

*Optimierung ohne Laufzeitnachteil*

Die SIA und INA sind notwendige Voraussetzungen für die eigentliche Synthese, die in der Lage ist, den Systemstart um bis zu 54.6% zu beschleunigen. Durch den vollautomatischen Ablauf der Optimierung ist diese dabei für die Laufzeit der Anwendung ohne Nachteile und kann genau wie eine Übersetzeroptimierung zugeschaltet werden. Die Kosten sind eine erhöhte Übersetzungszeit, die aber im Vergleich zu denen des Systemstarts viel seltener bezahlt werden müssen. In manchen Fällen kommt auch noch eine geringe Erhöhung der Systemabbildgröße hinzu, die aber bereits zur Übersetzungszeit ermittelt und die Optimierung bei knappem Flash somit auch deaktiviert werden kann.

*Instanzgraph: Stand der Kunst*

Das Zwischenprodukt der SIA und INA ist der Instanzgraph, der nicht nur für Optimierungszwecke dienen kann. Ein weiterer Zweck ist die Programmdokumentation, da er eine schnelle Übersicht über den Aufbau der Anwendung liefert. Er ähnelt damit im Zweck UML-Sequenzdiagrammen, für deren Erstellung es ebenfalls statische Analysen gibt [MKL24, Pag24, TP24] oder Klassendiagrammen, die ebenfalls automatisch aus dem Quellcode extrahiert werden können [Str24,ZEN24,Hea24,Hee24]. All diese Programme haben aber kein Verständnis des RTOS, dessen Kommunikation mit der Anwendung in eingebetteten Systemen wesentlich die Anwendung strukturiert. Speziell auf RTS zugeschnitten und für die Analyse des Echtzeitverhaltens können der Tracealyzer [Per24] und Grasp [HBL10] Diagramme ähnlich zum Instanzgraph liefern. Diese Analysen sind aber im Vergleich zur SIA und INA dynamisch. Durch ihren statischen Charakter sind die Analysen in ARA daher besser geeignet, um beispielsweise automatisiert Dokumentation zu erstellen.

*Instanzgraph: Andere Nutzung*

Außerdem kann der Instanzgraph für weitere Analysen dienen, die das System verifizieren. In ARA sind dazu zwei Analysen implementiert: Eine Analyse gibt eine Fehlermeldung aus, wenn Systemaufrufe zum Schützen von kritischen Bereichen nicht paarweise pro Instanz auftreten. Sie reiht sich damit in die lange Reihe der Programme ein, die kritische Abschnitte verifizieren [DAD14,LSB+19,NPS+09,EA03], benötigt aber, gegeben, dass der Instanzgraph aus Optimierungsgründen bereits vorliegt, so gut wie keine Zusatzkosten. Die Analyse ist korrekt, aber nicht vollständig. Durch ihre Grobgranularität übersieht sie eine Vielzahl von falsch benutzten Sperren, ist aber in der Entwicklung hilfreich, um offensichtliche Falschbenutzung aufzudecken. Überdies prüft ARA bei FreeRTOS die Größe der Halde gegen die Summe der Größen des Stapelspeichers für die einzelnen Tasks und vermeidet bereits zur Übersetzungszeit „Out-of-Memory“-Fehler der Laufzeit.

*SSE-Vergleich*

Im Vergleich zur SSE sind die SIA und INA deutlich selektiver bezüglich der interpretierten Systemaufrufe. Jede Analyse interpretiert für sich genommen weniger Systemaufrufe als die SSE und führt überdies einen deutlich kleineren Zustand mit sich. Die Analysen sind in der flussinsensitiven Variante selektiver als die SSE. Diese ignoriert zwar auch in Form von COMPUTATION-ABBs alle Anweisungen, die nichts mit dem RTOS zu tun haben, aber behält die prinzipielle Kontrollflussstruktur bei. Die flussinsensitive Variante überspringt hingegen

alle ABBs, indem sie auf dem Aufrufgraphen operiert. Alle Varianten implementieren zudem einen Widening-Operator, der ein Mehrfachdurchlauf von Schleifen überflüssig macht: Eine Instanz, die in einer Schleife erstellt wird, wird als solche markiert. Die SIA und INA sind mit dem Algorithmus aus Quellcode 6.3 klassische abstrakte Interpretationen. Der Zustand (eben der Instanzgraph) führt allerdings im Gegensatz zur SSE keine positionsabhängigen Informationen mit sich[32] und wird immer nur erweitert (Instanzen und Interaktionen kommen hinzu, vorhandene werden aber nie geändert). Der Instanzgraph stellt damit einen (infiniten) vollständigen Verband dar, dessen Größenrelation über die Instanzanzahl, Interaktionsanzahl und über die Instanzmengenangabe festgelegt ist:

1. Ein Instanzgraph $I_1$ ist größer als $I_2$, wenn er mehr Knoten oder Kanten besitzt. Bei gleich vielen Knoten oder Kanten gilt die nachfolgende Bedingung.

2. Ein Instanzgraph $I_1$ mit gleich vielen Knoten und Kanten wie $I_2$ ist größer als dieser, wenn es mindestens eine Instanz gibt, die eine größere Mengenangabe besitzt.

*Korrektheit und Vollständigkeit*

Die SIA und INA sind als korrekte Analysen angelegt, damit die nachfolgende Optimierung überhaupt möglich ist: Wenn der Instanzgraph eine Instanz als pseudostatisch klassifiziert, ist sie es auch. Wenn der Instanzgraph keine Interaktion zwischen zwei Instanzen aufzeigt, existiert auch keine[33]. Der Algorithmus stellt dies sicher, da er auf dem vollständigen Kontrollfluss- bzw. Aufrufgraphen operiert (erzeugt von einer vollständigen Analyse) und dort jeden Systemaufruf konservativ interpretiert. Sollte beispielsweise eine Instanz oder Interaktion innerhalb einer Verzweigung oder Schleife erfolgen, erzeugt ARA trotzdem einen entsprechenden Knoten oder eine Kante im Instanzgraphen.

*Forschungsfrage*

Blicken wir erneut auf meine 1. Forschungsfrage: Welche Herausforderungen entstehen auf eingebetteten dynamischen Systemen und können betriebssystemgewahre Analysen diese überwinden? Ich habe in diesem Kapitel pseudostatische Instanzen und zwangsläufig generische Interaktionsmuster identifiziert, die erst durch dynamische Systeme entstehen und Nachteile bezüglich Vorhersagbarkeit und Effizienz der Systeme mitbringen. Weiterhin konnte ich mit der SIA und INA zeigen, dass beide Muster effizient automatisiert erkennbar und mit einer geeigneten Synthese die Nachteile zudem überwindbar sind. Der als Zwischenprodukt entstehende Instanzgraph bringt zudem weitere Vorteile wie Programmdokumentation und Fehlererkennung. Die Forschungsfrage soll damit beantwortet sein.

---

[32] Einzig die Information, ob sich die Instanz vor oder nach dem Schedulerstart befindet, wird erfasst, allerdings nicht innerhalb des Zustandes, sondern über die Art des Algorithmusaufrufs.

[33] Es gibt leider eine Ausnahme dieser Regel: Wenn die Wertanalyse die Beteiligten einer Interaktion nicht finden kann, erzeugt ARA im Instanzgraphen eine Interaktion von/mit Unbekannt, die auch die fragliche Instanz betreffen könnte. In diesem Fall kann eine Interaktion mit der fraglichen Instanz nur über den Interaktionstyp ausgeschlossen werden oder es muss der konservative Fall angenommen werden, dass die fragliche Optimierung nicht durchgeführt werden kann. Der Algorithmus bleibt damit korrekt, verliert aber an Vollständigkeit.

# 7

# Die Analyse von Mehrkernsystemen

## Nur dann synchronisieren, wenn es notwendig ist

Die abstrakte Interpretation versucht, alle Systemzustände zu fassen. Dies stellt auf Mehrkernsystemen eine Herausforderung dar, bei denen die fast beliebige zeitliche Relation zwischen den Kernen zu einer Vielzahl von Zuständen (geringer Aussagekraft) führt. In diesem Kapitel stelle ich eine Analyse vor, die diese Herausforderung angeht: Sie synchronisiert die abstrakten Zustände der Kerne nur dort, wo dies durch die unterliegende Betriebssystemsemantik notwendig ist. Weiterhin behandele ich verschiedene Optimierungen auf Basis der Analyse und evaluiere diese daran.

Wie bereits in Abschnitt 3.6 angesprochen, ist die klassische abstrakte Interpretation nicht direkt für die Analyse von Mehrkernsystemen geeignet. Vor allem ist unklar, in welcher zeitlichen Relation die Fäden auf verschiedenen Kernen untereinander stehen. In Kapitel 4 habe ich mit Astrée und GOBLINT zwei betriebssystemgewahre Analysen vorgestellt, die in der Lage sind, Mehrkernsysteme zu analysieren [Min15, SSS+21, Min12, SVM03]. Sie behandeln dazu diejenigen Daten flussinsensitiv, die über Fadengrenzen hinweg gültig sind, verzichten also speziell bei Mehrkerninteraktionen auf Genauigkeit. Auch Dietrich erkennt für die SSE und SSF die Problematik der Erweiterung auf Mehrkernsysteme [DHL15] und schlägt vor, entweder jeden Kern getrennt zu analysieren oder einen Formalismus zu benutzen, der Parallelität unterstützt [Die19A3.6] (ein Ansatz, der in einem betriebssystemungewahren Kontext bereits von Haur et al. umgesetzt wurde [HBR22]). Aber auch diese Methoden verlieren in Bezug auf Mehrkernsysteme Genauigkeit.

*Zustandsexplosion durch Potenzmenge*

Üblicherweise wird der Genauigkeitsverlust als Notwendigkeit begründet, da der naive Ansatz, eine klassische abstrakte Interpretation auf ein Mehrkernsystem auszuweiten, auf der Bildung von Potenzmengenzuständen basiert [Rin01,Min15]. Die abstrakte Interpretation wird dazu auf jeden Kern getrennt ausgeführt – ein Verfahren, das in einer Menge aus Mengen von Einkernzuständen resultiert – und anschließend die Potenzmenge dieser Zustände bildet. Dazu wird jeder Zustand eines Kerns mit jedem Zustand des anderen Kerns kombiniert, da ohne weiteres eben nicht bekannt ist, wie die Kerne zueinander stehen. Dieses Verfahren skaliert allerdings nicht: Die Potenzmengenbildung führt zu einer Zustandsexplosion.

Ich will mich in diesem Kapitel diesem Problem widmen und mich damit auseinandersetzen, ob eine Mehrkernanalyse im Sinne einer abstrakten Interpretation mit einer ähnlichen Genauigkeit durchführbar ist. Konkret führt mich dies zu meiner 2. Forschungsfrage: Ist eine betriebssystemgewahre abstrakte Interpretation auf Mehrkernsystemen ohne Potenzmengenbildung durchführbar?

*MultiSSE: Idee*

Zur Beantwortung der Frage habe ich die MultiSSE entwickelt, eine Mehrkernanalyse, die die SSE als Grundlage verwendet, aber um ein effizientes Verfahren erweitert, Interaktionen zwischen mehreren Kernen zu erfassen. Sie berechnet dazu gerade nicht die volle Potenzmenge an Zuständen, sondern nutzt aus, dass die Betriebssystemsemantik explizite (selten vorkommende) Stellen vorgibt, die Interaktionen zwischen Kernen hervorrufen und synchronisiert die Kernzustände nur an genau diesen Stellen. Auch diese Analyse soll im Kontext von ARA vor allem helfen, nicht funktionale Eigenschaften zu verbessern und damit durch Spezialisierung das Gesamtsystem zu optimieren. Ich werde darum in diesem Kapitel zuerst die Analyse vorstellen und anschließend auf Spezialisierungen auf dieser Basis eingehen. Das Kapitel beruht auf entsprechender Vorarbeit, die ich bereits zusammen mit Björn Fiedler, Andreas Kässens und Daniel Lohmann in [›EFL23] und [›EKF+24] publizieren konnte.

## 7.1 Voraussetzungen

Die MultiSSE versucht nicht wie die SIA, zusätzlich die Analyse dynamischer Systeme anzugehen, sondern funktioniert nur auf statischen Systemen. Generell stelle ich mit der Analyse einige Anforderungen an die Systeme:

1. Der Ablaufplaner des Systems muss deterministisch arbeiten.
2. Das System muss partitioniert sein. Insbesondere ist jede Aktivität fest einem Kern zugeordnet und wird dort lokal geplant. Jegliche Interaktion zwischen Kernen findet über (eine Teilmenge definierter) Systemaufrufe statt.
3. Das System ist statisch. Alle Instanzen sind zur Kompilierzeit bekannt.
4. Systemaufrufe sind explizit: Sie und ihre Parameter sind statisch im System erkennbar.
5. Die Analyse ist optional in der Lage, Code-Ausführungszeiten zu verwerten. Nur in diesem Fall muss für jeden ABB die schlimmstmögliche und bestmögliche Ausführungszeit bekannt sein (WCET, BCET).

*Voraussetzungen: Beurteilung*

Die ersten drei Bedingungen sind im Echtzeitumfeld nicht ungewöhnlich und werden von entsprechenden RTOSs bereits vorgegeben. So schreibt z. B. ARINC 653 prioritätenbasiertes Scheduling in jeweils eigenen Partitionen innerhalb eines Mehrkernsystems vor und auch der AUTOSAR-Standard erzwingt alle diese Bedingungen [AEE03,AUT13]. Die Implementierung in ARA setzt die Analyse daher praktisch für AUTOSAR-Anwendungen um, kapselt die betriebssystemspezifischen Teile aber in einem Modell (wie in Kapitel 8 erläutert). Die 4. Bedingung ist von der SSE übernommen und für statische Systeme leicht umsetzbar. Wenn vom Anwender gewünscht, kann die MultiSSE Code-Ausführungszeiten für die engere Abschätzung der zu synchronisierenden Einzelsystemzustände verwenden. Nur in diesem Fall setzt sie diese als Ergebnis einer vorherigen Analyse voraus (z. B. [Abs24, Weg17,MDD+24,SWU+21,KP05]).

## 7.2 Demonstration am laufenden Beispiel

*Beispielanwendung*

Um die Technik wie auch den Nutzen der MultiSSE zu verdeutlichen, werde ich, wie bei der SIA, ein laufendes Beispiel verwenden, das ich in diesem Abschnitt vorstelle und anschließend durchgängig verwende. Es handelt sich dabei um ein System mit drei Kernen und vier Tasks, das auf dem AUTOSAR-Standard aufbaut (Abbildung 7.1). Der Task *T11* startet auf Kern 1 automatisch, betritt einen kritischen Abschnitt und aktiviert anschließend alle anderen Tasks, die ihrerseits entweder einen kritischen Abschnitt (*T01* und *T21*) oder eine hochpriore Aufgabe ausführen (*T02*).

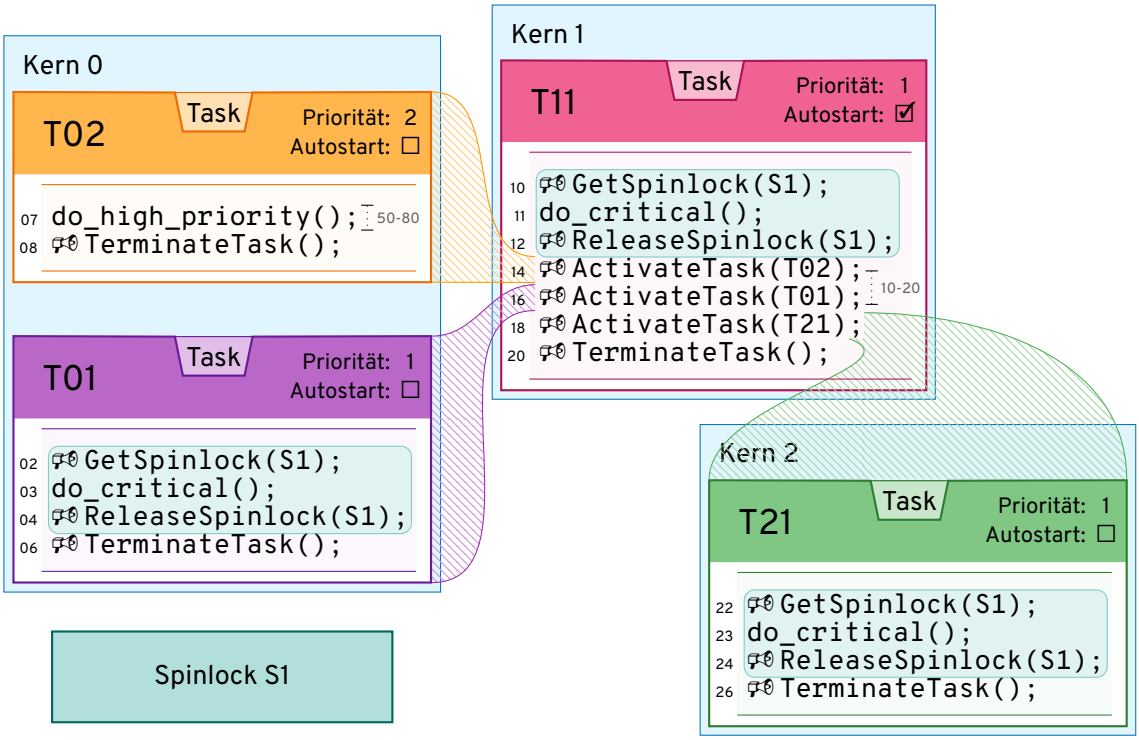



**Abbildung 7.1** Eine Beispielanwendung, bestehend aus vier Tasks, die auf drei Kerne partitioniert sind. Jeder Anweisung ist bereits ihre zugehörige ABB-Nummer zugeordnet (siehe Abbildung 7.2). Für ausgewählte Codeabschnitte sind Zeitintervalle angegeben, derer alle sich in Abbildung 7.2 finden. Task *T11* startet automatisch, und aktiviert die anderen Tasks. Manche Tasks betreten durch Spinlock *S1* geschützte kritische Abschnitte, die entsprechend farblich markiert sind (adaptiert aus [›EFL23]).

*Optimierungspotential*

Zum Anwendungsstart ist ausschließlich *T11* aktiv. Damit läuft dessen kritischer Bereich zwangsläufig unterbrechungsfrei, sodass die Systemaufrufe zum Sperren und Entsperren der ihn schützenden Sperre *S1* (ABB 10 und 12) überflüssig sind und zur Optimierung entfernt werden können. Unter zusätzlicher Berücksichtigung der ABB-Ausführungszeiten (mit Intervallen zwischen BCET und WCET gekennzeichnet) existiert weiteres Optimierungspotential: *T11* aktiviert nach dem kritischen Bereich *T02*, der eine hochpriore, aber langwierige Aufgabe bearbeitet. Mit dem Wissen, dass das Intervall zwischen der Aktivierung von *T02* (ABB 14) und *T01* (ABB 16) mit 10–20 Zeiteinheiten kürzer als die Dauer der hochprioren Aufgabe (ABB 07, 50–80 Zeiteinheiten) ist, ist errechenbar, dass *T01* bei ActivateTask(T01) zwar lauffähig, aber nicht laufend wird. Der übliche Mechanismus, den Planer eines anderen Kerns über einen möglichen Kontextwechsel zu informieren, ist das Senden eines *Interprocessor Interrupts (IPIs)*. Dieser ist hier mangels zu bewirkendem Kontextwechsel unnötig. Zudem ist analysierbar, dass der kritische Bereich von *T21* zwangs-

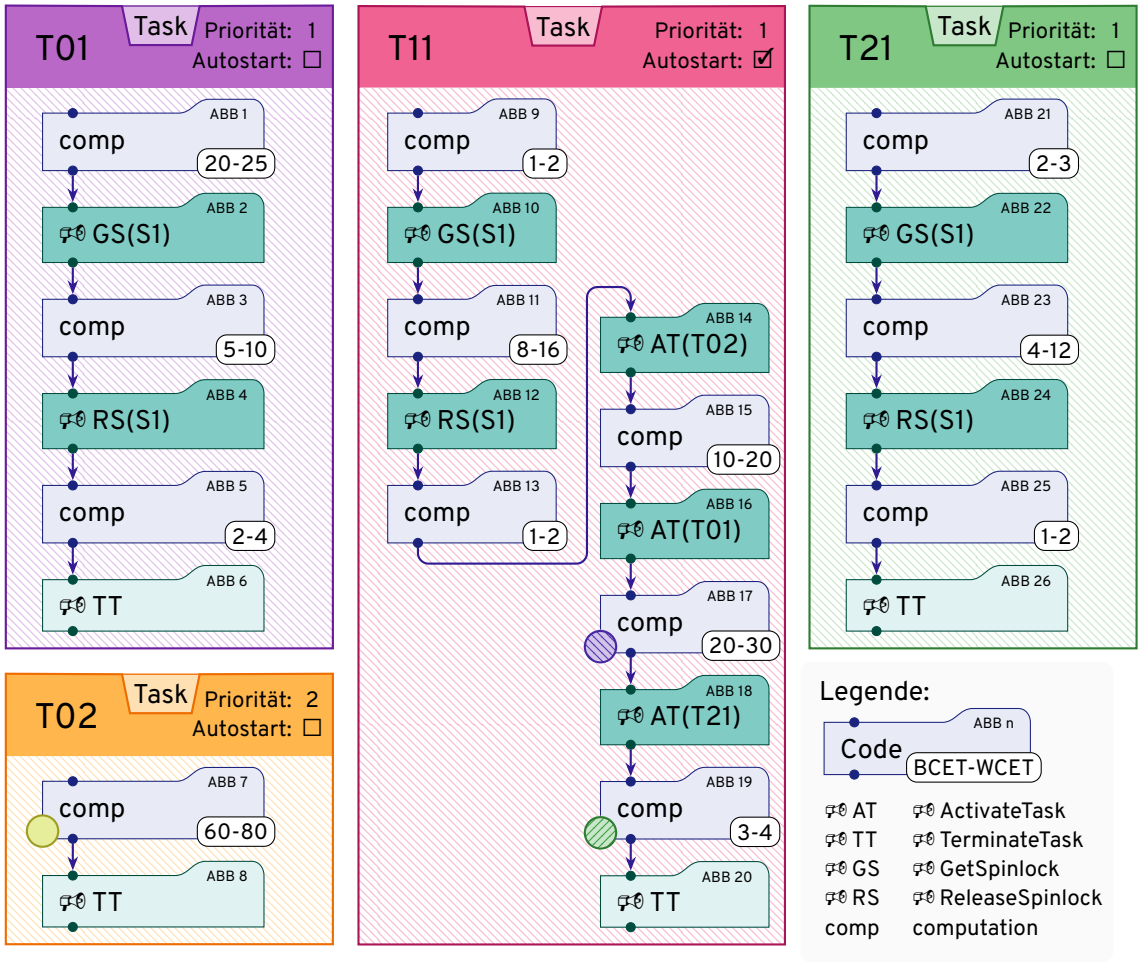



**Abbildung 7.2** ABBs der Beispielanwendung (Abbildung 7.1, adaptiert aus [›EFL23]). Systemaufrufe, die eine Interaktion mit einem anderen Kern bewirken, sind farblich intensiver. Die mit einem farbigen Kreis markierten COMPUTATION-ABBs entsprechen denen in Abbildung 7.5.

läufig *vor* dem des Tasks *T01* läuft, alle kritischen Bereiche also seriell geschehen und *S1* überflüssig machen.

# 7.3 Funktionsweise der MultiSSE (Überblick)

Die MultiSSE hat den ICFG in ABB-Form und die (aus der Konfiguration gewonnene) Instanzliste (der Instanzgraph ohne Kanten) als Eingabe. Abbildung 7.2 visualisiert den ICFG der Beispielanwendung. Aus diesem produziert sie den *Multi-State Transition Graph (MSTG)*, einen Graph, der an den SSTG angelehnt ist, diesen aber stark erweitert.

*Aufrufparameter*

```
Eingabe:
  • instance_list:  Instanzgraph (ohne Kanten)
  • icfg:           Interprozeduraler Kontrollflussgraph

Ausgabe:
  • mstg:           Multi-State Transition Graph

def multisse(instance_list, icfg):
    # Initialisierung
    initial_sp = get_initial_sp(instance_list, icfg)
    queue = Queue(initial_sp)
    mstg.add(initial_sp)

    while not queue.is_empty():
        cur_sp = queue.pop()

        # split_into_single_core erzeugt Gabelungskanten (Definition 28)
        for labss in cur_sp.split_into_single_core():
            # kernlokale Analyse (Abschnitt 7.4.3)
            cross_syscalls = single_core_analysis(labss, mstg, icfg)
            for cross_syscall in cross_syscalls:
                # Synchronisationspartnersuche (Abschnitt 7.4.4)
                pairing_partners, parent_sp = \
                    pairing_partner_search(cross_syscall)
                # Erzeugung neuer Entry-SPs (Abschnitt 7.4.5)
                new_entry_sp = create_sp(cross_syscall,
                                         pairing_partners,
                                         parent_sp,
                                         mstg)
                # SPs verschmelzen und falls notwendig reevaluieren
                queue.push_reevaluations_if_necessary(new_entry_sp)
                if mstg.node_exists(new_entry_sp):
                    mstg.merge_equal(new_entry_sp)
                    continue
                mstg.add(new_entry_sp)

                # Erzeugung neuer Exit-SPs
                for exit_sp in system_semantic(new_entry_sp,
                                               cross_syscall.cpu_id,
                                               icfg):
                    mstg.add(exit_sp)
                    mstg.connect(new_entry_sp, exit_sp, Type("global"))
                    queue.push(exit_sp)
```

**Codeblock 7.1** Zusammengefasste Funktionsweise der MultiSSE (Pseudocode). Hervorgehobene Funktionen werden im Folgenden erklärt (adaptiert aus [›EKF+24]).

*Naiver Ansatz: beliebige Synchronisation*

Wie bereits angesprochen, besteht das Grundproblem einer abstrakten Interpretation darin, herauszufinden, wie verschiedene Kerne in ihrem jeweiligen Kontrollfluss zueinander stehen. Der naive Ansatz nimmt dabei durch die Kreuzproduktbildung implizit an, dass jede Stelle im Kontrollfluss des einen Kerns gleichzeitig zu beliebigen Stellen des Kontrollflusses des anderen Kerns stattfinden kann. Er synchronisiert darum alle Kerne (also determiniert deren gegenseitige Position im Kontrollfluss) zu allen überhaupt möglichen Punkten.

*Besser: seltene Synchronisation*

Die Grundidee der MultiSSE beruht auf zwei Beobachtungen: Interaktionen treten zum einen nur an fest definierten Stellen auf: Zumeist arbeiten die Kerne unabhängig voneinander. Nur eine kleine Menge definierter Systemaufrufe kann eine Aktion auf einer statisch bestimmbaren Menge anderer Kerne hervorrufen und damit auch nur dort eine Synchronisation erfordern (in Abbildung 7.2 sind dies die intensiver hervorgehobenen Systemaufrufe). Zum anderen können Interaktionen zwischen Kernen einen Einfluss auf Folgeinteraktionen haben und damit eben nicht mehr eine beliebige Synchronisation der Kerne zulassen. Beispielsweise ist aus der Kontrollflussstruktur der Beispielanwendung ersichtlich, dass der Systemaufruf GetSpinlock(S1) (ABB 02) von Task *T01* zwangsläufig *nach* dem Systemaufruf GetSpinlock(S1) (ABB 10) von Task *T11* stattfinden muss und keinesfalls parallel passieren kann, da der Task *T11* den Task *T01* überhaupt erst lauffähig macht.

*explizite Synchronisationspunkte*

Die MultiSSE macht beides im MSTG sichtbar: Sowohl werden die Systemaufrufe, die eine Interaktion zwischen Kernen erfordern, explizit in *Synchronisationspunkte (Synchronisation Points, SPs)* abgebildet, als auch die Menge dieser SPs durch die Beachtung von Kausalitätsketten zwischen vorhergegangenen SPs auf ein Minimum beschränkt. Ich will wegen der zentralen Bedeutung des Konzepts hier bereits eine informelle Definition geben, die ich später bei der detaillierten Beschreibung des Algorithmus noch verfeinern werde:

> **Definition 23:** ***Synchronisationspunkt (Synchronisation Point, SP)*.**
> Ein SP ist ein abstrakter Zeitpunkt bei der Systemausführung, der die Kontrollflussposition einer Menge von Kernen zueinander festsetzt.

*Schritte des Algorithmus*

Der Algorithmus besteht aus vier Schritten, bei denen die letzten drei wiederholt ausgeführt werden, solange bis sich der resultierende MSTG nicht mehr ändert („Fixpunkt-Algorithmus“). Ohne zu diesem Zeitpunkt auf die Details einzugehen, findet sich der Algorithmus in Pseudocode in Quellcode 7.1:

1. **Initialisierung**: Der Algorithmus startet mit einem initialen SP, der die anfängliche Position aller Kerne zueinander bestimmt. Alle Kerne befinden sich dabei entweder im Leerlauf oder, sollten sie aufgrund der Systemkonfiguration direkt einen Task starten, in dessen initialem ABB.

2. **Kernlokale Analyse**: Ausgehend vom initialen SP wird der Kontrollfluss nun kernlokal abstrakt interpretiert (ähnlich der SSE). Dazu wird der SP in abstrakte Zustände aufge-

teilt, die nur einen Kern abbilden. Sollte bei der kernlokalen Analyse ein Systemaufruf gefunden werden, der eine Interaktion auf einem anderen Kern zur Folge hat – ein *kernübergreifender Systemaufruf (Cross-Core System Call, CrossSyscall)* –, dann hält die kernlokale Analyse dort an und fährt mit dem nächsten Schritt fort (andere Zweige im ICFG werden weiter traversiert).

3. **Bildung der SPs**: Ein CrossSyscall führt zu einer Interaktion mit einer statisch feststellbaren festen Menge an anderen – *betroffenen* – Kernen und resultiert immer in einem oder mehreren SPs. Für die Bildung eines solchen müssen alle Kontrollflusspositionen der betroffenen Kerne gefunden werden, die potentiell zur gleichen Zeit wie der CrossSyscall stattfinden können. Diese zu finden, ist Aufgabe einer *Synchronisationspartnersuche*. Anschließend werden aus der Menge der Synchronisationspartner die möglichen SPs gebildet.

4. **Interpretation der SPs und aufsplitten in lokale Zustände**: Jeder der neu gebildeten SPs wird anschließend abstrakt interpretiert und damit seine Auswirkung auf alle betroffenen Kerne berechnet. Dies resultiert in einer Menge von neuen SPs, die anschließend wieder in lokale Zustände getrennt werden, sodass die Analyse mit der kernlokalen Analyse (Schritt 2) fortfahren kann.

## 7.4 Die Funktionsweise der MultiSSE im Detail

Bevor ich die einzelnen Schritte im Detail erläutere, ist es notwendig, die von der Analyse verwendeten Konzepte und Datenstrukturen zu definieren.

### 7.4.1 Konzepte und Datenstrukturen

Die MultiSSE arbeitet wie andere abstrakte Interpretationen auf einem abstrakten Zustand. Dazu definiert sie in der grundlegendsten Form den *Multi-Core Abstract System State*:

**Definition 24:** ***Multi-Core Abstract System State (MAbSS)*.**
Der Multi-Core Abstract System State ist ein Tupel aus einer Menge von Kernkontexten und globalen Instanzkontexten

$$\begin{aligned} \text{MAbSS} &= (K, I) \quad K \subseteq \mathcal{K}, I \subseteq \mathcal{I}_g \\ \mathcal{K} &= \{k_0, \ldots, k_{\max}\} \quad \text{Menge aller Kernkontexte} \\ \mathcal{I}_g &= \{i_0, \ldots, i_{\max}\} \quad \text{Menge aller Instanzkontexte} \end{aligned}$$

Instanzkontexte sind im allgemeinen Sinne betriebssystemabhängige Attributmengen, die den abstrakten Zustand einer Instanz erfassen, wie z. B. die Information, ob eine

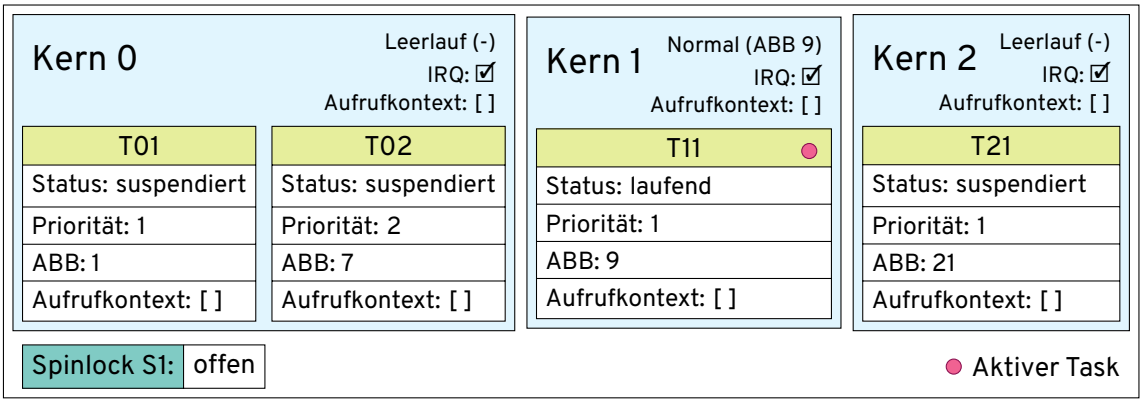



**Abbildung 7.3** Visualisierung des initialen MAbSS, adaptiert aus [›EFL23]. Zu erkennen sind die 4 Tasks (kernlokale Instanzkontexte) auf 3 Kernen (Kernkontexte), wobei ausschließlich *T11* läuft („Status: laufend“ im Instanzkontext, sowie die Ausführung von ABB 9 im Kernkontext). Außerdem ist der globale Instanzkontext des Spinlock *S1* erkennbar, das aktuell nicht sperrt.

Sperre gerade sperrt oder die dynamische Priorität eines AUTOSAR-Tasks. $I$ im speziellen Sinne ist hierbei auf globale Instanzkontexte beschränkt, also für Instanzen, die wie Sperren keinem einzelnen Kern fest zugeordnet sind.

Der Kernkontext ist ein 6-Tupel:

$$
\begin{aligned}
k \in \mathcal{K} = \; & (\text{ABB}, \tau, \text{IRQ}, C, A, I) \\
\text{ABB} = \; & \text{der aktuell ausgeführte ABB} \\
\tau = \; & \text{der aktuell ausgeführte Faden} \\
\text{IRQ} = \; & \text{das „Interrupt Enable Bit“.} \\
& \quad \text{Gibt an, ob Unterbrechungen zugelassen werden.} \\
C = \; & \text{Aufrufkontext (in diesem Fall als Pfad im Aufrufgraph)} \\
A \in \; & \{\text{Leerlauf}, \text{Wartend}, \text{Normal}\} \text{ (Ausführungszustand)} \\
I = \; & \text{Menge an kernlokalen Instanzkontexten}
\end{aligned}
$$

Der Kernkontext besteht damit aus den Informationen, die die Position im Kontrollfluss spezifizieren (ABB, $\tau$ und $C$), Attribute, die den RTOS-internen Zustand angeben ($A$ und IRQ[34]), sowie alle dem Kern fest zugeordneten instanzspezfischen Kontexte.

[34] Dieses Bit kann auch als Minimalrepräsentation des (abstrakten) Hardwarezustands angesehen werden, der aber in diesem Fall vom (abstrakten) Planer des RTOS genutzt wird.

Abbildung 7.3 visualisiert den MAbSS, der den initialen Zustand des Beispielsystems repräsentiert. Ein MAbSS kann als Übermenge eines AbSS (siehe Abschnitt 4.5.1) angesehen werden, der einerseits ein klare Trennung zwischen globalen (zwischen Kernen geteilten Daten) und kernlokalen Daten zieht, und andererseits den AbSS von genau einem Ausführungskontext auf soviele Ausführungskontexten erweitert, wie Kerne im System existieren. Ein MAbSS mit einem Kernkontext entspricht damit prinzipiell einem AbSS. Auf einem MAbSS ist ein Gleichheitsoperator definiert, der zwei MAbSSs als gleich ansieht, wenn sie die gleiche Menge an Kern- und Instanzkontexten besitzen.

Jeder Knoten des MSTG ist ein MAbSS. Dieser kann aber nochmal, je nach Aufbau, in zwei Ausprägungen unterschieden werden. Die erste Ausprägung ist die Spezialisierung des generischen Mehrkernzustandes auf genau einen Kern:

**Definition 25:** ***Local-Core Abstract System State (LAbSS)*****.**
Ein LAbSS ist ein MAbSS, der genau einen Kernkontext und keinen globalen Instanzkontext hat.

$$\begin{aligned} \text{LAbSS} &= \text{MAbSS}_{|K|=1 \wedge I=\emptyset} \\ &= (K, I) \quad \text{mit } |K| = 1 \text{ und } I = \emptyset \\ \cup_{\text{LAbSS}} &= \text{Menge aller LAbSSs} \end{aligned}$$

Die zweite Ausprägung ist die gegenteilige Richtung, die dann auftritt, wenn zwei oder mehr Kerne synchronisiert werden. Ich kann damit meine Definition 23 vervollständigen:

**Definition 26:** ***Synchronisationspunkt (Synchronisation Point, SP)*****.**
Ein Synchronisationspunkt ist ein MAbSS mit mindestens zwei Kernkontexten. Er determiniert damit die entsprechenden Kerne in ihrer Position untereinander.

$$\begin{aligned} \text{SP} &= \text{MAbSS}_{|K|\geq 2} = (K, I) \quad \text{mit } |K| \geq 2 \\ \cup_{\text{SP}} &= \text{Menge aller SPs} \end{aligned}$$

Stellen im Kontrollfluss, die einen CrossSyscall enthalten, resultieren zuerst in einem *CrossSyscall-LAbSS*. Der CrossSyscall wird auf dem Kern des CrossSyscall-LAbSS ausgeführt, benötigt aber eine Synchronisierung mit einer statisch feststellbaren Menge an betroffenen Kernen, deren Position ihrerseits wieder mit einem LAbSS ausgedrückt wird. Dies führt zur Definition eines Synchronisationspartners mit dem die anschließende Definition des MSTG ermöglicht wird:

**Definition 27: Synchronisationspartner.**
Ein Synchronisationspartner in Bezug auf einen CrossSyscall-LAbSS ist ein LAbSS eines betroffenen Kerns, der zur gleichen Zeit wie der CrossSyscall-LAbSS existieren kann.

**Definition 28: Multi-State Transition Graph.**
Der MSTG ist ein Graph, dessen Knoten aus MAbSSs bestehen, die über Kanten verschiedener Typen miteinander verbunden sind, die die Art der Transition beschreiben.

$$
\begin{aligned}
\text{MSTG} &= (V, E) \\
T_E &= \{\text{local}, \text{global}, \text{fork}, \text{join}\}\ (\text{Kantentyp}) \\
V &= \{\text{MAbSS}_0, \ldots, \text{MAbSS}_n\} \\
E &= \{(x, y, f) \,|\, (x, y) \in V^2, f = V \times V \to T_E\} \\
f(x, y) &= \begin{cases} \text{local} & \text{falls } x \in \cup_{\text{LAbSS}}, y \in \cup_{\text{LAbSS}} \\ \text{global} & \text{falls } x \in \cup_{\text{SP}}, y \in \cup_{\text{SP}} \\ \text{fork} & \text{falls } x \in \cup_{\text{SP}}, y \in \cup_{\text{LAbSS}} \\ \text{join} & \text{falls } x \in \cup_{\text{LAbSS}}, y \in \cup_{\text{SP}} \end{cases}
\end{aligned}
$$

Der Kantentyp ist abhängig von der Ausprägung der Knoten, die diese Kante verbindet. Lokale Kanten verbinden kernlokale Zustände, zeigen also eine Transition an, die unabhängig von anderen Kernen stattfindet. Dem konträr sind globale Kanten, die zwei SPs verbinden und damit eine Transition mehrerer Kerne, die in Abhängigkeit stehen, symbolisieren. Kanten zwischen einem CrossSyscall-LAbSS, seinen Synchronisationspartnern und dem resultierenden SP sind vom Typ „join“. Ich werde diese Kanten auch *Mündungskanten* nennen. Kanten, die einen SP und die LAbSSs, in die dieser sich aufgabelt, verbinden, sind *Gabelungskanten* und vom Typ „fork“. Abbildung 7.4 visualisiert einen Ausschnitt des MSTGs des laufenden Beispiels. Zu erkennen ist dort u. a. die systematische Abfolge der verschiedenen Kantentypen, die dem Algorithmus inhärent ist und in den nachfolgenden Abschnitten erklärt wird.

## 7.4.2 Die Initialisierung

Der Algorithmus startet mit einem initialen SP. Dazu bestimmt er für jeden AUTOSAR-Task, ob dieser nach Systemkonfiguration automatisch startet und initialisiert den Kontext entsprechend. Abhängig davon setzt er überdies die Kernkontexte und alle Kontexte der globalen Instanzen. Der initiale SP synchronisiert damit immer alle im System vorhandenen Kerne. Abbildung 7.3 zeigt als MAbSS den initialen SP des Beispielsystems.

## 7.4.3 Kernlokale Analyse

Die MultiSSE fährt anschließend mit der kernlokalen Analyse fort (Quellcode 7.2). Dazu muss sie den SP zuerst in eine Menge von LAbSSs aufsplitten. Sie trennt hierfür aus dem SP jeden Kernkontext – inklusive den Instanzkontexten, die diesem Kern zugeteilt sind – in

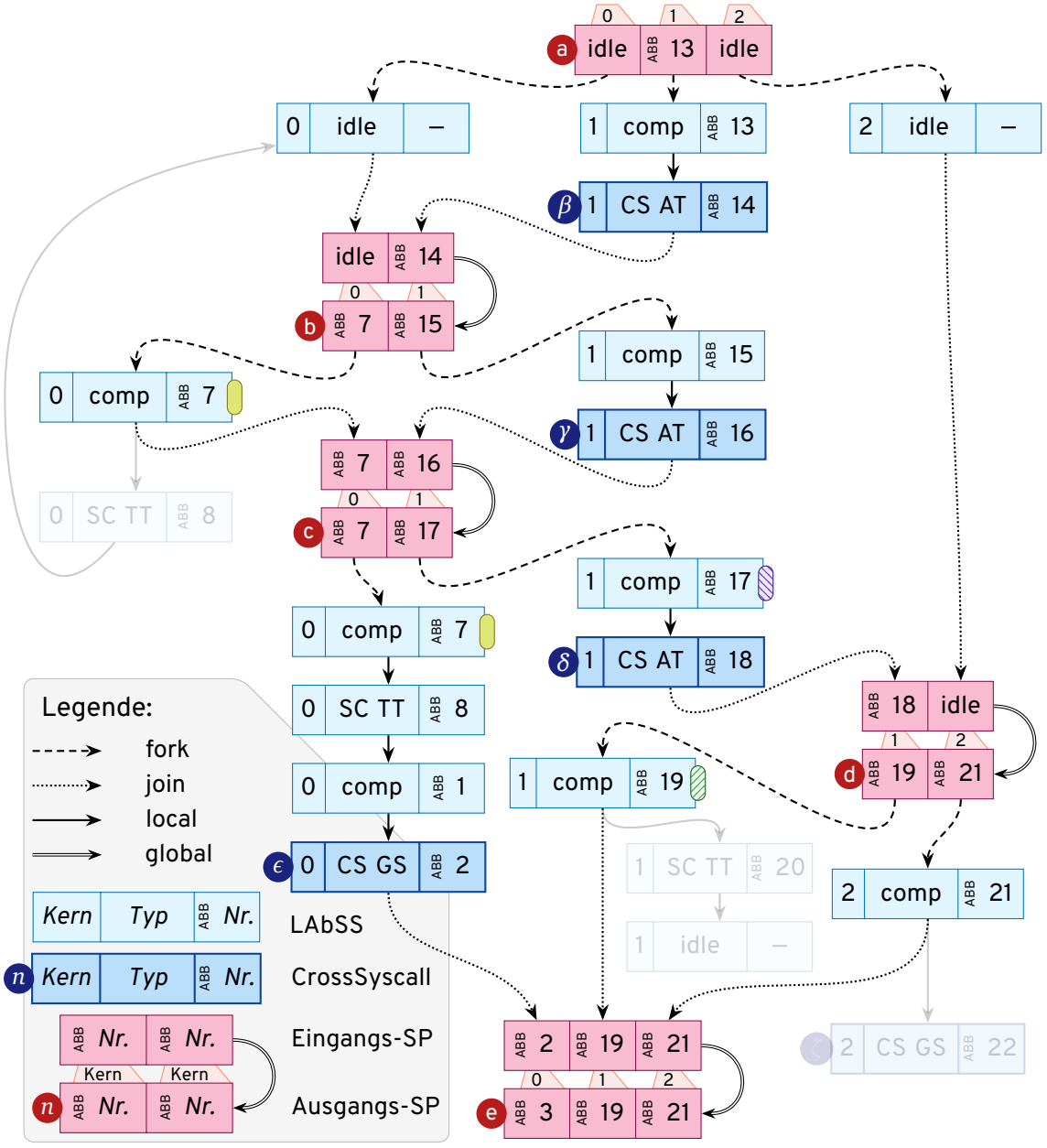



**Abbildung 7.4** Visualisierung eines Ausschnittes des MSTG der Beispielapplikation (Abbildung 7.1). Die abstrakten Zustände sind vereinfacht dargestellt. Der dargestellte Graph visualisiert den Ablauf der Task-Aktivierungen. LAbSSs, die hervorgehoben sind, entsprechen denen aus Abbildung 7.5, adaptiert aus [›EFL23].

```
Eingabe:
  • labss: Der aktuelle LAbSS
  • mstg:  Der unvollständige MSTG
  • icfg:  Der interprozedurale Kontrollflussgraph

Ausgabe:
  • cross_syscalls: Liste aller neuen CrossSyscalls

def single_core_analysis(labss, mstg, icfg):
  queue = Queue(labss)
  cross_syscalls = []  # Rückgabewert
  while not queue.is_empty():
    cur_labss = queue.pop()
    try:
      next_labsss = system_semantic(cur_labss, labss.get_cpu_id(), icfg)
    except IsCrossSyscall:  # Test, ob cur_labss ein CrossSyscall ist
      cross_syscalls.append(cur_labss)
      continue
    for next_labss in next_labsss:
      mstg.add(next_labss)
      mstg.connect(cur_labss, next_labss, Type("local"))
      if mergable(next_labss):  # bereits evaluierte Folgezustände
        merge_with_existent_labss(next_labss)
      else:
        queue.push(next_labss)
  return cross_syscalls


Eingabe:
  • mabss:  Der aktuelle MAbSS
  • cpu_id: Die ID des Kerns, der interpretiert werden soll.
  • icfg:   Der interprozedurale Kontrollflussgraph

Ausgabe:
  • new_mabsss: Liste an neuen MAbSSs

def system_semantic(mabss, cpu_id, icfg):
  active_abb = mabss.cpus[cpu_id].abb
  if type(active_abb) == computation:
    next_mabsss = icfg.follow_abb_chain(mabss)
    return next_mabsss
  if type(active_abb) == call:
    return follow_call_chain(mabss)
  if type(active_abb) == syscall:
    if is_cross_syscall(active_abb) and not
        mabss.contains_all_affected_cores(active_abb):
      raise IsCrossSyscall
    return interpret(active_abb)  # RTOS- und systemaufrufspezifisch
```

**Codeblock 7.2** Algorithmus der kernlokalen Analyse (Pseudocode). Der Algorithmus ähnelt der SSE [Die19A3.4.2] (adaptiert aus [›EKF+24]).

einen eigenen LAbSS. Der globale Kontext wird für diesen Schritt nicht weiter beachtet, da er für die kernlokale Analyse keine Rolle spielt.

Anschließend kann die MultiSSE den LAbSS interpretieren. Sie führt dazu eine Transitionsfunktion für MAbSSs ein (Quellcode 7.2, Zeile 37):

$$(\text{MAbSS}_{g+1,0}, \ldots, \text{MAbSS}_{g+1,n}) = \text{system\_semantic}(\text{MAbSS}_g, c)$$
$$g = \text{Generation}$$
$$c = \text{der zu evaluierende Kern}$$

Diese Transitionsfunktion gilt gleichbedeutend für LAbSSs wie für SPs und ist eine adaptierte Variante der gleichlautenden Funktion der SSE (siehe Abschnitt 4.5.1, insbesondere interpretiert diese Funktion Systemaufrufe durch abstraktes Ausführen und Neuplanen, Zeile 48). Im Unterschied zur SSE erhält sie zusätzlich den Kern, der die Transition ausgelöst hat (bei einem LAbSS ist dies der einzig im Zustand enthaltene Kern, bei einem SP der Kern, auf dem der CrossSyscall ausgeführt wird). Die Funktion arbeitet weitestgehend analog zur Version der SSE mit einigen Unterschieden:

- Soll die Funktion bei der Interpretation eines LAbSS einen CrossSyscall evaluieren, verweigert sie dieses und meldet der umgebenden Analyse eine Liste an betroffenen Kernen zurück, die zur korrekten abstrakten Interpretation noch fehlen (Zeilen 45–47).
- Bei der Interpretation eines SPs interpretiert die Funktion die Auswirkungen des CrossSyscalls auf jeden Kernkontext. Abhängig vom CrossSyscall kann dies in mehreren Folgezuständen resultieren. Bei einem ReleaseSpinlock emittiert die Funktion beispielsweise im Fall einer Sperre, auf die mindestens zwei Kerne warten, mindestens zwei Folgezustände, in denen jeweils ein anderer der wartenden Kerne die Sperre erhält.

Nach jeder Transition speichert die Analyse diese im MSTG (Zeile 20) und verschmilzt dabei gleiche Zustände miteinander (Zeile 23). Die MultiSSE wendet nun analog zur SSE die system_semantic-Funktion auf die LAbSSs an, bis entweder keine neuen Zustände entstehen (beispielsweise hat der Leerlaufzustand eine Transition zu sich selbst) und/oder diese alle in CrossSyscalls enden (Zeile 12).

### 7.4.4 Synchronisationspartnersuche

*originäre, betroffene und synchronisierte Kerne*

Findet die kernlokale Analyse einen LAbSS, der einen CrossSyscall ausführen soll, so stoppt sie für diesen Zweig und wechselt zur Synchronisationspartnersuche (Quellcode 7.3). Der CrossSyscall bewirkt im System eine Interaktion mit mindestens einem anderen betroffenen Kern, wodurch an genau dieser Stelle die Position der anderen Kerne mit der des CrossSyscalls determiniert werden muss. Der Kern, der den CrossSyscall auslöst, heißt *originärer*

```
Eingabe:
  • cross_syscall_labss:  Ein LAbSS, der abstrakt einen CrossSyscall ausführt
  • mstg:                 Der MSTG

Ausgabe:
  • pairing_partners:     Synchronisationspartnerliste (LAbSSs mit Eltern-SPs)

def pairing_partner_search(cross_syscall_labss, mstg):
  current_core = cross_syscall_labss.get_cpu_id()
  affected_cores = get_affected_cores(cross_syscall_labss)
  paths = get_parent_sps(cross_syscall_labss,
                         union(affected_cores, current_core))
  pairing_partners = list()
  for path in paths:
    parent_sp = path[0]
    filtered_mstg = mstg.reachable_and_valid_for(path)
    core_local_barrier_sps = list()
    for affected_core in affected_cores:
      core_local_barrier_sps.append(
          get_core_local_barrier(parent_sp, affected_core, current_core))
    corridors = build_corridors(filtered_mstg, core_local_barrier_sps)
    for corridor in corridors:
      # the corridor border is equivalent to its
      # predecessor and successor SPs
      for corridor_specific_pairing_partners in state_product(corridor):
        pairing_partners.append(
            tuple(corridor_specific_pairing_partners, parent_sp))
  return pairing_partners
```

**Codeblock 7.3** Algorithmus der Synchronisationspartnersuche (Pseudocode, adaptiert aus [›EKF+24]).

Kern („`current_core`“, Zeile 9). Die Menge der betroffenen Kerne („`affected_cores`“, Zeile 10) zusammen mit dem originären Kern ist die Menge der (vom CrossSyscall) *synchronisierten Kerne*. Sie entspricht genau der Menge, die von den zukünftigen SPs synchronisiert wird. Der einfachste (trivial korrekte) Ansatz diese Menge zu finden, ist eine Kombination des CrossSyscall-LAbSS mit allen LAbSSs der betroffenen Kerne, was aber ähnlich der Kreuzproduktbildung in einer Zustandsexplosion resultiert. Die MultiSSE schränkt daher die Menge an möglichen anderen LAbSSs ein, indem sie nur LAbSSs in Betracht zieht, die durch den davor durchlaufenen Kontrollfluss möglich sind.

Konkret passiert dies durch eine Suche ausschließlich ab dem letzten *Eltern-SP*: *Eltern-SP*

**Definition 29: Eltern-SP.**
Ein SP $a$ ist ein Eltern-SP in Bezug auf einen SP $b$, wenn $a$ die gleiche Menge oder eine Übermenge an Kernen synchronisiert wie $b$ und sich auf keinem Pfad von $a$ zu $b$ ein weiterer Eltern-SP befindet.

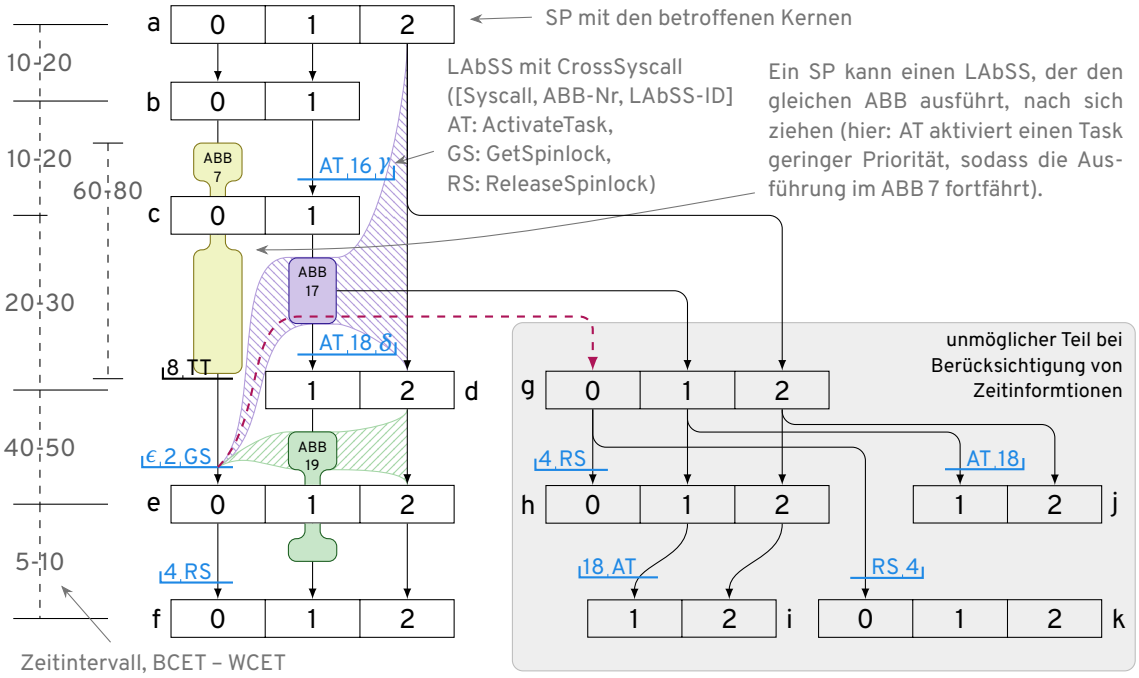



**Abbildung 7.5** MSTG-Ausschnitt des Beispielsystems (Abbildung 7.1) in einer im Gegensatz zu Abbildung 7.4 vereinfachten, aber umfangreicheren Variante (zusammengefasste Ein- und Ausgangs-SPS, zumeist ausgeblendete LAbSSs; nur LAbSS mit expliziter LAbSS-ID sind in der anderen Abbildung). Die Grafik enthält eine gesonderte Darstellung der Konzepte der Synchronisationspartnersuche und Zeitanalyse, adaptiert aus [›EFL23].

Der SP $b$ steht in obiger Definition stellvertretend für die SPs, die aus dem CrossSyscall gebildet werden. Eltern-SPs sind für die Synchronisationspartnersuche eine Barriere im Kontrollfluss. Da sie die gleiche oder eine Übermenge an Kernen synchronisieren, wie die synchronisierten Kerne, muss der Kontrollfluss der betroffenen Kerne bereits *hinter* dem Eltern-SP liegen. Bedingt durch die Verschmelzung der LAbSSs können mehrere Eltern-SPs existieren, die auf disjunkten Pfaden erreicht werden. Die MultiSSE sucht daher zuerst die Menge der Eltern-SPs zusammen mit den Pfaden mit einer adaptierten Breitensuche (Zeilen 11–14). Alle folgenden Schritte werden für jeden Eltern-SP ausgeführt.

*Eltern-SP: Beispiel*

Zur Demonstration der Synchronisationspartnersuche dient Abbildung 7.5. Sie stellt eine Übermenge des MSTG-Ausschnitts aus Abbildung 7.4 dar, konzentriert die Darstellung aber auf die für die Suche wichtigen Teile. Konkret soll hier als Beispiel der Systemaufruf GetSpinlock(S1) (ABB 2, LAbSS $\epsilon$) aus dem Beispielsystem (Abbildung 7.1) dienen (er resultiert schlussendlich in SP e). Da die angefragte Sperre auf allen Kernen des Systems operiert, besteht die Menge der betroffenen Kerne hier aus Kern 1 und 2, die in diesem Fall

der Menge an synchronisierten Kernen äquivalent ist. Die Rückwärtssuche auf dem MSTG nach den Eltern-SPs liefert folglich den SP a mit einem Pfad über SP b und SP c.

*MSTG-Filterung*

Die Analyse fährt mit einer Filterung des MSTG fort (Zeile 16). Durch die Zustandsverschmelzung können einerseits Schleifen im Graph entstehen und andererseits Pfade ausgehend vom Eltern-SP existieren, die einen dem aktuellen Pfad widersprechenden Kontrollfluss repräsentieren. Die MultiSSE entfernt darum zur Schleifenvermeidung sowohl alle eingehenden Kanten zum Eltern-SP als auch solche, die hinter SPs liegen, die den originären Kern synchronisieren, da diese in andere Pfade der Programmausführung verzweigen.

*kernlokale Barriere*

Der Eltern-SP ist garantiert der einzige SP auf dem Weg zum CrossSyscall, der die vollständige Menge der synchronisierten Kerne synchronisiert. Auf dem SP-Pfad vom Eltern-SP zum CrossSyscall können aber andere SPs existieren, die den originären Kern mit einer Untermenge an betroffenen Kernen synchronisieren. Diese SPs agieren als Barrieren in der Zeit: Synchronisationspartner des CrossSyscall-LAbSSs müssen *hinter* diesen SPs liegen. Die Analyse sucht darum nach diesen *kernlokalen Barrieren* (Zeilen 17–20):

> **Definition 30: kernlokale Barriere.**
> Eine kernlokale Barriere in Bezug auf Kern $c$ und einen Eltern-SP ist der letzte SP auf dem Pfad vom Eltern-SP zum CrossSyscall-LAbSS, der den originären Kern mit dem Kern $c$ synchronisiert.

Im Beispiel ist der SP c auf dem Pfad vom Eltern-SP a zum CrossSyscall-LAbSS $\epsilon$ der letzte SP, der Kern 0 mit Kern 1 (einem betroffenen Kern) synchronisiert und ist darum die kernlokale Barriere in Bezug auf den Kern 1. Bei Kern 2 ist der Eltern-SP a gleichzeitig auch die kernlokale Barriere, da kein anderer SP auf dem Pfad Kern 2 synchronisiert.

*Synchronisationskorridor*

Der nächste Schritt im Algorithmus ist das Finden von Synchronisationskorridoren (Zeile 21). Die MultiSSE beginnt damit, die LAbSSs eines bestimmten Kerns als mögliche Synchronisationspartner anzusehen, die auf die kernlokale Barriere folgen. Die Entscheidung für einen bestimmten LAbSS kann aber eine Auswirkung auf benachbarte betroffene Kerne haben, sollten diese in der Zwischenzeit über einen oder mehrere andere SPs synchronisiert worden sein. Im Beispiel ist dies sichtbar: Die Entscheidung für den LAbSS auf Kern 1, der ABB 17 ausführt (lila markiert), beschränkt die Suche nach Synchronisationspartnern auf Kern 2 auf den Bereich zwischen SP a und SP d, da SP d als Barriere für den ABB 17 dient. Der entsprechende Suchbereich ist durch den von Nordwest nach Südost gestreiften Korridor gekennzeichnet. Analoges gilt für den grünen Korridor (SW-NO gestreift): Die Auswahl des LAbSSs auf Kern 1, der ABB 19 ausführt, beschränkt die Suche auf Kern 2 auf die LAbSSs, die SP d folgen.

Sind die entsprechenden Synchronisationskorridore gefunden, bildet die Analyse das Kreuzprodukt der enthaltenen LAbSSs und bildet daraus jeweils einen SP (es zieht dabei nur

LAbSSs in Betracht, die in einem Zeitintervall ausgeführt werden, die also einen COMPUTATION-ABB ausführen oder sich im Leerlauf befinden, Zeile 25). Im Beispiel resultiert die Synchronisationspartnersuche in zwei SPs: SP e (resultierend aus dem grünen Korridor) und SP g (resultierend aus dem lila Korridor).

### 7.4.5 SP-Konstruktion und Interpretation

```
1  Eingabe:
2    • cross_syscall:     Der LAbSS mit dem CrossSyscall
3    • pairing_partners:  Eine Liste an Synchronisationspartnern
4    • parent_sp:         Der Eltern-SP
5    • mstg:              Der MSTG
6
7  Ausgabe:
8    • entry_sp:          Der neue Eingangs-SP
9
10 def create_sp(cross_syscall, pairing_partners, parent_sp, mstg):
11     # construct SP
12     cpu_contexts = list()
13     for labss in chain(cross_syscall, pairing_partners):
14         cpu_contexts.append(labss.get_cpu_context())
15     global_context = parent_sp.get_global_context()
16     entry_sp = MAbSS(cpu_contexts, global_context)
17
18     # connect SP
19     mstg.connect(cross_syscall, entry_sp, Type("join"))
20     mstg.connect(pairing_partners, entry_sp, Type("join"))
21
22     return entry_sp
```

**Codeblock 7.4** Erstellung und Kantenverbindung eines Synchronisationspunktes (Pseudocode, adaptiert aus [›EKF+24])

Sind alle Synchronisationspartner gefunden, also eine Menge an LAbSS, dann wird aus diesen zusammen mit dem CrossSyscall-LAbSS ein neuer SP gebildet (Quellcode 7.4). Die MultiSSE extrahiert dazu aus allen LAbSS die kernlokalen Kontexte (diese sind konstruktionsbedingt disjunkt) und fügt diese mit dem globalen Instanzkontext des Eltern-SPs zu einem neuem SP zusammen (Zeilen 12–16). Die beteiligten LAbSSs und der neue SP werden mit Mündungskanten verbunden (Zeilen 19–20).

Anschließend interpretiert sie diesen SP (einen Eingang) mit der vorhin beschriebenen system_semantic-Funktion, was in einer Menge an Folge-SPs (Ausgänge) resultiert (mit globalen Kanten verbunden) (Quellcode 7.1, Zeile 38). Diese werden anschließend analog zum initialen SP in LAbSSs aufgesplittet (verbunden mit Gabelungskanten) und auf diesen

erneut eine kernlokale Analyse ausgeführt (Quellcode 7.1, Zeile 18). Beispielsweise resultieren die LAbSSs aus dem grünen Korridor in einem Eingangs-SP, der nach der Interpretation in genau einem Ausgangs-SP e resultiert, in dem der Kern 0 die Sperre erhalten hat und beginnt, den kritischen Bereich auszuführen.

### 7.4.6 Terminierung und Reevaluierungen

Grundsätzlich iteriert der Algorithmus solange über den MSTG, bis keine neuen Knoten und Kanten mehr entstehen. Konkret verwaltet er neu entstehende SPs in einer Arbeitsliste und iteriert diese solange, bis sie leer ist (Quellcode 7.1, Zeile 14). Stößt der Algorithmus auf Zustände, die bereits existieren, verschmilzt er diese und nimmt sie nicht neu in die Arbeitsliste auf. Normalerweise gilt ein SP als evaluiert, wenn die LAbSSs, die ihm folgen, bereits die kernlokale Analyse durchlaufen haben und – so sie dann in einem CrossSyscall enden – wiederum mit SPs verbunden sind. Wird ein neuer Zustand mit einem bestehenden verschmolzen, ist dieser bzw. der erzeugende SP für gewöhnlich bereits evaluiert. Es gibt jedoch zwei Ausnahmen dieser Regel (Quellcode 7.1, Zeile 31):

1. Hat ein neuer SP $n$ einen zeitlichen Vorgänger-SP $v$, der eine andere Menge oder Übermenge an Kernen synchronisiert, so kann $n$ für die CrossSyscalls, die aus $v$ resultieren, neue Synchronisationspartner hinzufügen. Diese werden darum erneut evaluiert.

2. Wenn der SP, der erzeugt werden soll, bereits existiert, evaluiert ist und verschmolzen wird, so hat der Algorithmus trotzdem einen neuen Pfad (also eine neue Reihenfolge) entdeckt, der diesen SP erzeugt, sodass der SP (nur) in Bezug auf diese Kante neu evaluiert werden muss.

## 7.5 Berücksichtigung von Zeitinformationen

Die obige Beschreibung zur MSTG-Konstruktion berücksichtigt alle Informationen, die der Kontrollfluss bietet, um Interaktionen zwischen Kernen zu minimieren. Der Algorithmus macht dazu die Ordnung an Systemaufrufen explizit, die implizit im Kontrollfluss steckt.

Die MultiSSE ist aber überdies fähig, das Zeitverhalten einzelner ABBs ebenfalls in die Berechnung aufzunehmen: Viele mit der bisherigen Methode gefundenen Synchronisationspunkte sind in der Realität nicht möglich, da die Kerne auch innerhalb der Kontrollflussstruktur nicht beliebig zueinander stehen, sondern Ausführungszeit für die Abarbeitung des unterliegenden Codes benötigen. Eine Berücksichtigung dieser Ausführungszeiten führt dabei konkret zu einer weiteren Minimierung der Synchronisationspartner und damit allgemein auch zu einer Verkleinerung des gesamten MSTG. Der Algorithmus führt die Zeitanalyse dabei als Teil der normalen Analyse aus, konstruiert SPs erst gar nicht, die

zeitlich nicht möglich sind, und reduziert in der Folge nicht nur die Graphgröße, sondern auch auch die Analysezeit drastisch.

*Grundidee* Grundsätzlich arbeitet die MultiSSE für die Zeitanalyse mit dem Vergleich von Zeitintervallen auf den verschiedenen Kernen. Voraussetzung ist dabei eine bereits existierende Zuordnung einer BCET und WCET zu jedem COMPUTATION-ABB[35]. Die Kernidee des Algorithmus ist nun die Beobachtung, dass es ausreichend ist, die Zeitintervalle ausgehend vom letzten SP aus zu berechnen, da zu diesem Punkt bekannt ist, wie die beteiligten Kerne zeitlich zueinander stehen, danach aber unabhängig voneinander ihre Ausführung fortsetzen. Zur Berechnung benutzt der Algorithmus ein lineares Ungleichungssystem, eine Kombination eines linearen Gleichungssystems mit Nebenbedingungen, die in diesem Fall aus den Zeitintervallgrenzen der Codeabschnitte bestehen. Diese Systeme bilden ein Problem der linearen Programmierung und können effizient gelöst werden [Kha79][36].

*Intervalltypen* Der Algorithmus nutzt vier Intervalltypen:

**ABB**: Ein Zeitintervall eines ABBs. Dies ist genau die Zeit, die über die BCET und die WCET des ABBs spezifiziert wird.

**L–L**: Ein kernlokales Ausführungsintervall und somit ein Zeitintervall zwischen zwei LAbSSs. Es wird durch die Zeitintervalle der unterliegenden ABBs berechnet.

**S–S**: Ein Zeitintervall zwischen zwei SPs. Hierbei handelt es sich um das Zeitintervall, in dem ein SP (ein Zeitpunkt) relativ zu seinem Vorgänger ausgeführt werden kann.

**S–L**: Ein Zeitintervall zwischen einem SP und einem LAbSS. Dieses Intervall gibt die Ausführungszeit bis zu einem LAbSS (der z. B. einen CrossSyscall auslöst) in Bezug zu einem SP an. Genau dieser Intervalltyp wird von der Analyse verglichen, muss aber vorher berechnet werden.

Alle Intervalltypen besitzen weiterhin drei relevante Zeitpunkte:

1. $t_0$: Die Startzeit des Intervalls.
2. $t_{\text{min}}$: Die minimale Ausführungszeit des Intervalls.
3. $t_{\text{max}}$: Die maximale Ausführungszeit des Intervalls.

### 7.5.1 Der Aufbau eines Ungleichungssystems

[35] Für die Berechnung des Zeitverhaltens geht die MultiSSE davon aus, dass CALL-ABBs und SYSCALL-ABBs atomar geschehen und nur COMPUTATION-ABBs in Zeitintervallen ausgeführt werden. Diese Annahme führt im Algorithmus zu einer (notwendigen) expliziten Serialisierung aller eine gleiche Schnittmenge an Kernen betreffenden Systemaufrufe. Ich werde die Annahme in Abschnitt 7.5.2 diskutieren.

[36] In ARA nutze ich eine entsprechende Funktion aus der Scipy-Bibliothek (`https://scipy.org/`).

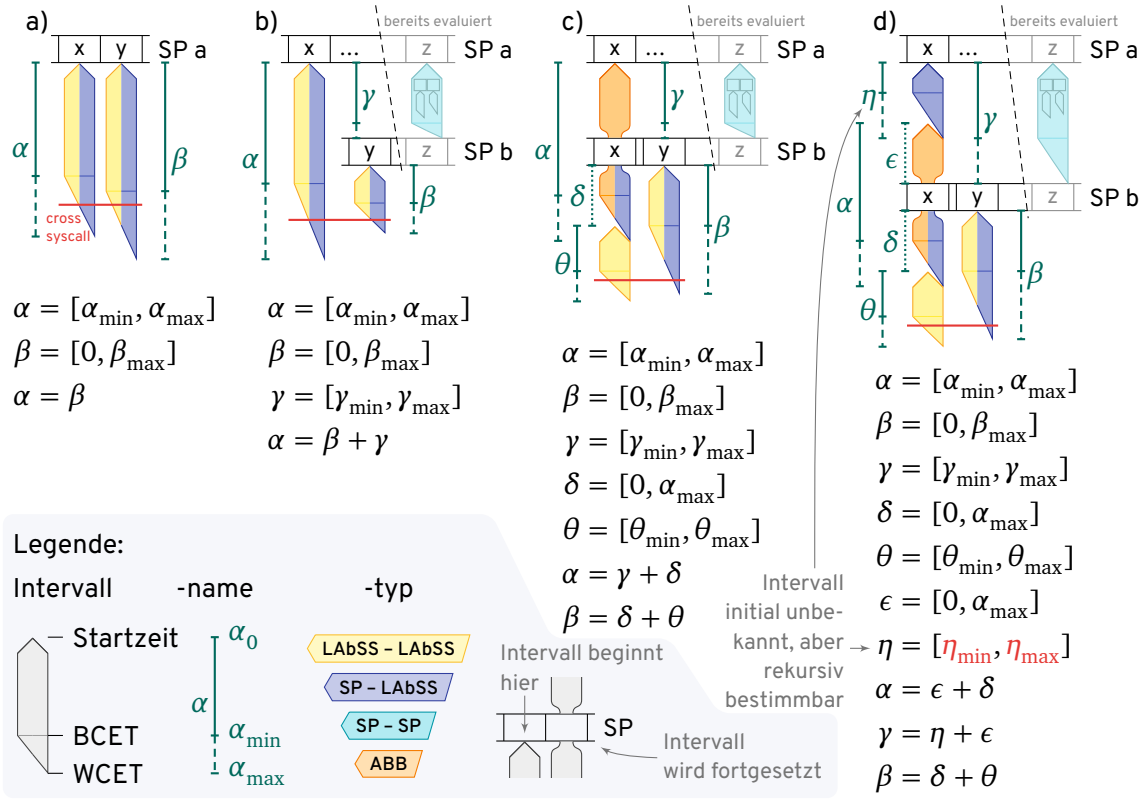



**Abbildung 7.6** Zeitintervallberechnungen, adaptiert aus [›EFL23]. Gezeigt sind vier Bausteine, wie Intervallvergleiche zustande kommen können und das resultierende Ungleichungssystem. Die Intervallfarben deuten auf den Intervalltypen hin. Manche Intervalle haben mehrere Typen.

Die Zeitberechnung der Synchronisationspartnersuche ist mit jeweils einem Ungleichungssystem möglich. Das System kann aus einer Kombination aus vier (komplexer werdenden) Bausteinen gebildet werden, die in Abbildung 7.6 gezeigt sind.

*Grundlegender Vergleich*

Der einfachste Fall a) ist ein Synchronisationspunkt (in der Grafik der SP a), der in zwei kernlokale Graphen aus LAbSSs verzweigt. Falls nun der eine Kern (Kern x) einen CrossSyscall ausführt, der einen anderen Kern (Kern y) betrifft, stellt sich die Frage, welche LAbSSs des anderen Kerns als Synchronisationspartner für den CrossSyscall in Frage kommen. Die normale Synchronisationspartnersuche liefert in diesem Fall eine Liste an Kandidaten, zu der die Zeitanalyse versucht, die folgende Frage zu beantworten:

> *Gegeben ein Kandidaten-LAbSS $k$ und einen CrossSyscall-LAbSS $c$, die den letzten gemeinsamen SP $a$ teilen, kann $k$ zur gleichen Zeit ausgeführt werden wie $c$? Anders formuliert: Endet das* S–L *-Intervall $\alpha$ von SP $a$ zu $c$ zu einer Zeit, die noch im* S–L *-Intervall-$\beta$ von SP $a$ zu $k$ liegt?*

Nur wenn die Frage bejaht werden kann, ist der Kandidat geeignet. Da in diesem Fall der SP a direkt in einen LAbSS verzweigt, handelt es sich bei den Intervallen $\alpha$ und $\beta$ gleichzeitig auch

um Intervalle vom Typ L–L, der Algorithmus muss also das Intervall zwischen zwei LAbSSs berechnen (Start und Ende sind inklusive), die jeweils verschiedene ABBs ausführen. Da die BCET und WCET eines jeden ABB gegeben ist, reduziert sich das Problem auf eine klassische WCET-Analyse, die ich mit einem pfadbasierten Ansatz ausgeführt habe [Hol08, EGL11]. Sind die L–L-Intervalle bekannt, ist das Ungleichungssystem, wie abgebildet, aufstell- und lösbar.

*Vergleich über SP-Kette*

Der Fall b) zeigt einen etwas komplizierten Fall, bei dem die fraglichen S–L-Intervalle nicht beim gleichen SP starten, sondern eine Kette von weiteren SPs dazwischenliegt, die zu einem Zusatzintervall $\gamma$ führen. Das Ungleichungssystem berücksichtigt dieses Zusatzintervall durch simple Addition, braucht dafür aber dessen $t_{\text{min}}$ und $t_{\text{max}}$. Da die Analyse bei der Bildung von SP b bereits eine Synchronisationspartnersuche durchgeführt hat und damit auch eine Zeitanalyse, hat sie die Zeiten bereits im Vorhinein berechnet. Sie speichert diese zwischen, kann sie unmittelbar im Ungleichungssystem einsetzen und dieses somit lösen.

*Unterbrochene ABBs*

Nicht immer führt ein SP auf einem Kern dazu, dass der dort gerade abstrakt ausgeführte ABB gewechselt wird, so kann der Algorithmus entweder nach einem SP zu einem neuen LAbSS wechseln, der den gleichen ABB wie zuvor abstrakt ausführt oder auch nach dem SP direkt zum alten LAbSS zurückwechseln. In diesem Fall wird der ABB zeitlich *unterbrochen*, was in Fall c) dargestellt wird. Dort beginnt die Ausführung des ABB bei SP a, endet aber erst nach SP b. Das Ungleichungssystem wird in diesem Fall komplizierter, da das Intervall $\theta$ (das mit der Ausführung des CrossSyscall endet) nicht direkt mit $\beta$ verglichen werden kann. Der ABB hat vor $\theta$ noch eine unbekannte restliche Ausführungslänge $\delta$, die sich aus seiner Gesamtlaufzeit abzüglich der bereits ausgeführten Zeit zwischen SP a und SP b bildet. Diese Zeit zwischen den SPs ist dem Algorithmus aber bereits wie im Fall b bekannt, weswegen wir das Ungleichungssystem mit einer Zusatzgleichung aufstellen und lösen können.

Im Fall c) hat der unterbrochene ABB seine Ausführung direkt an einem SP begonnen. Dies muss aber nicht sein, wie in Fall d) dargestellt. Dort unterbricht der SP b einen ABB mit dem Zeitintervall $\alpha$, der nicht an einem SP beginnt, sondern ein S–L-Intervall $\eta$ vor sich hat. Im Ungleichungssystem führt das zu einem zusätzlichen Term $\gamma = \eta + \epsilon$, dessen kritischer Teil die Bestimmung des erst einmal unbekannten Intervalls $\eta$ ist. Da es sich um ein S–L-Intervall handelt, kann man es jedoch mit den bereits vorgestellten Methoden rekursiv berechnen. Die Rekursion stoppt dabei garantiert spätestens am initialen SP, da dort alle Intervalle beginnen oder nach einer optionalen Berechnungstiefe, ab der der konservative Fall angenommen wird, dass von einem SP unterbrochene ABBs nach diesem entweder gar nicht oder mit ihrer vollen Laufzeit laufen.

## 7.5.2 Herausforderungen

*atomare Systemaufrufe*

Im Modell der MultiSSE wird angenommen, dass Systemaufrufe atomar passieren, also keine Zeit verbrauchen. Praktisch kann dazu die real anfallende Zeit des Systemaufrufs auf die Intervalle der umliegenden ABBs übertragen werden, ein Ansatz, den schon Dietrich so wählt [Die19A3.4.2]. Diese Annahme ist aber für die betroffenen Kerne eines CrossSyscalls nicht haltbar, da dieser z. B. den dort laufenden COMPUTATION-ABB mit einem IPI unterbrechen könnte und damit dessen Zeit zwangsläufig verlängern würde. Alle fraglichen Stellen sind im MSTG allerdings explizit sichtbar, weswegen in der Implementierung an diesen Stellen im Ungleichungssystem die Zeit um einen konstanten Faktor erhöht werden kann, der der WCET aller IPI-Behandlungsroutinen entspricht (die Zeitanalyse bleibt dadurch korrekt, verliert aber an Vollständigkeit). Zukünftig wäre hier zur Erhöhung der Vollständigkeit noch eine Unterstützung von systemaufrufspezifischen Unterbrechungszeiten denkbar.

*unbegrenzte Schleifen*

Eine weitere Herausforderung der Zeitanalyse ist die Beachtung von Schleifen. Wird ein CrossSyscall in einer Schleife aufgerufen, wird der betroffene Kern mehrfach unterbrochen, u. U. sogar im gleichen ABB, was sich im MSTG ebenso durch eine Schleife ausdrückt. Bei der allgemeinen Synchronisationspartnersuche spielen diese Schleifen keine Rolle und werden ignoriert, da dort nur interessant ist, *ob* eine Synchronisation in diesem Kontext möglich ist und nicht *wann*. Bei der Zeitanalyse muss der Analyse allerdings bekannt sein, wie oft die Schleife durchlaufen wird, um ein korrektes Ergebnis liefern zu können. Um ohne dieses Wissen korrekt zu sein, deaktiviert die MultiSSE bei der Erkennung einer unbegrenzten Schleife für diesen speziellen Teil des Graphen die Zeitanalyse (durch eine Annahme eines Zeitintervalls von $[0, \infty]$. Unbegrenzte Schleifen in einem RTS sind allerdings oft bereits zu Zwecken einer statischen Echtzeitanalyse unerwünscht [KP05]. Begrenzte Schleifen können immer virtuell ausgerollt werden, sodass das Problem gar nicht erst entsteht.

## 7.5.3 Zeitverhalten des Beispielsystems

Um die Zeitanalyse plastisch zu machen, werde ich sie am Beispiel (Abbildung 7.1) anhand des CrossSyscall GetSpinlock(S1) (ABB 2) einmal durchführen. Wir betrachten dabei die Fragestellung, passende Synchronisationspartner im lila Korridor (Abbildung 7.5) zu finden. Die entsprechende grafische Intervalldarstellung findet sich in Abbildung 7.7.

*Aufstellung des Ungleichungssystems*

Der CrossSyscall wird in Folge des SP c ausgelöst. Dieser unterbricht den ABB 7 in seiner Ausführung, beendet diese aber nicht (SP c resultiert aus einem ActivateTask, das einen niederprioren Task auf Kern 0 aktiviert und somit dort keine Umplanung bewirkt; ABB 7 ist in die bisherigen Abbildungen gesondert farblich markiert). Es handelt sich darum um ein Ungleichungssystem des obigen Falls c, da ABB 7 am SP b seine Ausführung startet. Die kernlokale Barriere in Bezug auf Kern 2 ist SP a, weswegen wir überdies die Intervalle bis dort benötigen. Insgesamt ergibt sich das auf der nächsten Seite dargestellte Ungleichungssystem:

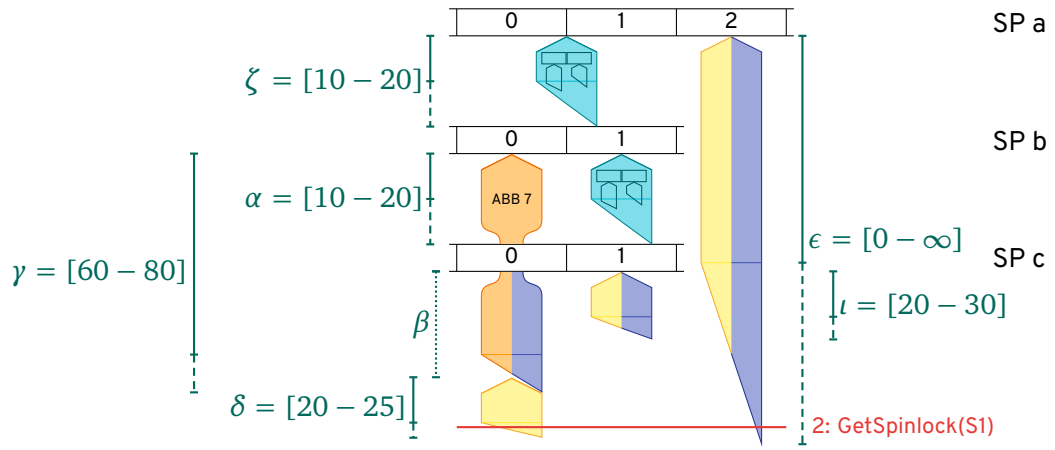



**Abbildung 7.7** Zeitintervallberechnung für passende Synchronisationspartner im lila Korridor für GetSpinlock(S1) des Beispiels (Abbildung 7.1, Abbildung 7.5), adaptiert aus [›EFL23].

| | |
|---|---|
| $\alpha + \beta = \gamma$ | Aufteilung der ABB-Laufzeit |
| $\beta + \delta = \theta$ | Zeit zwischen SP b und ABB 2 (der CrossSyscall) |
| $\theta = \iota$ | Vergleich zwischen Kern 0 mit Kern 1 |
| $\theta + \zeta + \alpha = \epsilon$ | Vergleich zwischen Kern 0 mit Kern 2 |
| $\alpha = [10, 20]$ | Zeit zwischen SP b und SP c |
| $\beta = [0, 80]$ | Verbleibendes Intervall von ABB 7 |
| $\gamma = [60, 80]$ | BCET und WCET von ABB 7 |
| $\delta = [20, 25]$ | Intervall von ABB 7 zu ABB 2 |
| $\iota = [0, 30]$ | Ausführungszeit von ABB 17 |
| $\epsilon = [0, \infty]$ | Leerlaufzustand auf Kern 2 |
| $\zeta = [10, 20]$ | Intervall zwischen SP a und SP b |

*Lösungsbestimmung*

Dieses Ungleichungssystem hat keine Lösung: $\beta$ muss resultierend aus $\alpha$ und $\gamma$ ein Intervall zwischen $[40, 70]$ sein. Dadurch wird $\theta$ auf $[60, 95]$ determiniert, wodurch die Gleichung $\theta = \iota$ mit $\iota = [0, 30]$ unlösbar wird.

Das unlösbare Gleichungssystem bedeutet in diesem Fall, dass – unter Berücksichtigung der Zeitinformationen – im gesamten lila Korridor keine geeigneten Synchronisationspartner existieren, was als direkte Folge den grauen Bereich im MSTG (Abbildung 7.5) als unmögliche Verzweigung gar nicht erst erzeugt. Auf dem Beispiel ist durch die Beachtung von Zeitinformationen allgemein eine Reduktion von 137 Knoten auf 65 Knoten messbar.

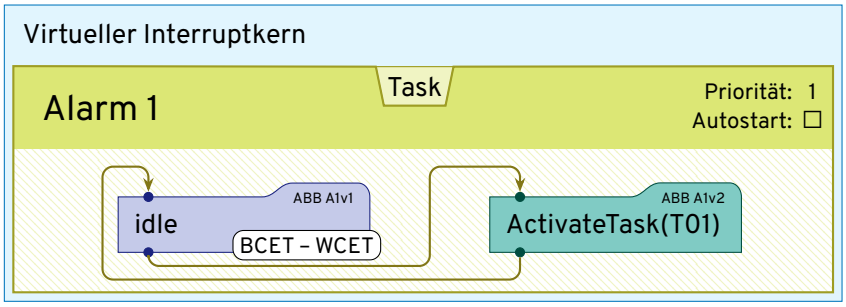



**Abbildung 7.8** Beispiel für einen virtuellen Interrupt-Kern mit einem periodischen Kontrollfluss. Dieser wechselt zwischen Leerlauf und der eigentlichen Interrupt-Aktion (hier einen Task zu aktivieren). Die Zwischenankunftszeit des Interrupts kann trivial als Zeitintervall des „idle“-Blocks spezifiziert werden.

## 7.6 Externe Interrupts

*Abbildung: Interrupts auf virtuelle Kerne*

Bislang bin ich nicht darauf eingegangen, wie externe Interrupts von der MultiSSE gehandhabt werden, die aber gerade in ereignisgetriebenen RTSs üblich sind. Dieser liegt die Beobachtung zugrunde, dass sich aus Sicht eines Kerns ein externer Interrupt nicht von einem IPI unterscheidet – und für IPIs bringt die MultiSSE bereits alles Notwendige mit! Externe Interrupts werden darum in der MultiSSE als Systemaufrufe eines virtuellen Interrupt-Kerns modelliert. Abbildung 7.8 zeigt ein Beispiel eines solchen Kerns. Die Analyse erstellt für jeden Interrupt einen eigenen (virtuellen) Kern, der in einer Schleife (ausgedrückt durch künstliche ABBs) den entsprechenden Systemaufruf auslöst, den auch der Interrupt gemäß der Systemspezifikation ausgelöst hätte. Ein Vorteil dieser Modellierung ist das einfache Einfügen von Zwischenankunftszeiten, die auf den künstlichen Kontrollfluss als BCET und WCET abgebildet werden können. Da sich auch in dieser nun künstlich konstruierten Schleife das Problem der Zeitanalyse mit unbegrenzten Schleifen ergibt, ist es möglich, die Ausführungsanzahl innerhalb der Hyperperiode zu begrenzen und die Schleife damit wieder zu begrenzen. Der künstlich angelegte Kern wird abhängig von der Systemspezifikation entweder zum Systemstart oder nach einem entsprechenden Systemaufruf gestartet, der den zugehörigen Interrupt aktiviert.

*Maßnahmen zur Zustandsminimierung*

Gerade nicht weiter beschränkte Interrupts können sehr viele SPs hervorrufen, da die Analyse davon ausgehen muss, dass der Interrupt andauernd auslöst. Dies war bereits bei der SSE ein Problem, die Interrupts so behandelt, dass sie diese zu jedem COMPUTATION-ABB auslöst [Die19A3.4.3]. Dietrich hat dazu Task-Gruppen eingeführt, die von einem Interrupt ausgelöst werden und innerhalb derer Behandlung der gleiche Interrupt nicht noch einmal ausgelöst werden kann. Der Gedanke dahinter ist der, dass ein Interrupt nicht seine eigene Behandlungsroutine (und die Task-Gruppe gilt hier als solche) unterbrechen

sollte, und das System entsprechend konstruiert ist. Für die MultiSSE habe ich dieses Konzept übernommen, sodass sie die Spezifizierung entsprechender Gruppen unterstützt.

*weitere Anwendungsmöglichkeiten*

Die Abbildung von Interrupts auf einen künstlichen – diesen Interrupt auslösenden – Kontrollfluss schafft eine einfache Modellierungsmöglichkeit für weitere Konzepte. So ist damit (obwohl noch nicht implementiert) die Abbildung von weiteren Abhängigkeiten möglich, die zwischen Anwendung und dem umgebenden System und der Hardware existieren (ich werde in der Diskussion, Abschnitt 7.9, genauer darauf eingehen). Auch die spezifische Umsetzung von Interrupts in der unterliegenden Hardware kann in den Mechanismus integriert werden, wie beispielsweise IRQ-Multicasts oder prioritätenbasiertes Routing von Interrupts, die über eine entsprechende Implementierung der Systemaufrufinterpretation und der Menge an betroffenen Kernen modelliert werden können.

## 7.7 Optimierungen

Die MultiSSE eröffnet die Möglichkeit zu Optimierungen, die auf dem feingranularen Wissen über das Systemverhalten basieren. Die Optimierungen teilen sich in zwei Kategorien auf: Schon bestehende Optimierungen für Einkernsysteme und Optimierungen für Mehrkernsysteme. Ich werde hier nur auf die neuen Optimierungen für Mehrkernsysteme eingehen (die Optimierungen für Einkernsysteme diskutiere ich in Abschnitt 7.9). Dazu habe ich zusammen mit Andreas Kässens zwei Modelle ausgearbeitet, wie sich die Kerne im Optimierungsfall verhalten und er hat auf dieser Basis eine Synthese für AUTOSAR entwickelt, die genau diese Optimierungen auf Basis der MultiSSE umsetzt. Da die Synthesedetails für die Analyse irrelevant sind, die Ergebnisse aber für die Validierung und Verifizierung der Analyse wichtig ist, werde ich das Thema nur zusammenfassend beschreiben[37].

### 7.7.1 Verhinderung von IPIs

IPIs sind sowohl auf dem ausführenden, als auch dem empfangenden Kern teuer [HCK+03]. Sie können außerdem eine Prioritätsumkehr auf dem empfangenden Kern bewirken, falls dort ein hochpriorer Task unterbrochen wird [LMN06, HLS+09]. In AUTOSAR betrifft das die Systemaufrufe `ActivateTask` und `SetEvent`. Auch andere AUTOSAR-Implementierungen haben das Problem erkannt. So sendet das quelloffene Trampoline OS beispielsweise nur einen IPI, wenn dieser notwendig ist, macht dies aber auf Kosten eines kernübergreifenden Schedulers, der über ein globales Lock potentiell das ganze System anhält [BBF+06]. Mit der MultiSSE sind genau diese Fälle nun explizit sichtbar. Stellt der Algorithmus fest, dass ein Faden eines anderen Kerns lauffähig gemacht werden soll,

[37] Eine genauere Betrachtung findet sich in [›EKF+24].

und kann nachweisen, dass dies kein unmittelbares Umplanen beinhaltet, so kann er den normal ausgelösten IPI konstruktiv verhindern. Im Beispiel ist der IPI, der durch den Systemaufruf 🔧 ActivateTask(T01) ausgelöst würde, überflüssig. Wenn wir an das Beispiel aus der Einleitung dieser Arbeit zurückdenken (Abbildung 1.2, Seite 7), enthält dieses ebenfalls einen überflüssigen IPI, der durch den Systemauruf 🔧 ActivateTask(T12) ausgelöst würde. Die MultiSSE kann dies sogar – im Gegensatz zum Beispiel dieses Kapitels – ohne Zeitanalyse erkennen.

*IPI-Modell*

Im Zuge der Synthese haben wir ein detailliertes Modell der Interrupt-Interaktion zwischen zwei Kernen ausgearbeitet, dass hier zusammengefasst in zwei Teile aufgeteilt werden kann: Den optimierbaren Teil einerseits ($\mathrm{IPI}_{\mathrm{opt}}$) und den nicht optimierbaren Teil andererseits ($\overline{\mathrm{IPI}_{\mathrm{opt}}}$). So muss beispielsweise die Implementierung von 🔧 ActivateTask, auch wenn die Aktion nicht unmittelbar zu einem Einlasten des Tasks auf dem anderen Kern führt, den betreffenden Task doch in die Bereitliste setzen, eine damit unoptimierbare Berechnung. Die gesamte Unterbrechung des empfangenden Kerns ist hingegen optimierbar und damit ebenfalls die Prioritätsumkehr vermeidbar.

## 7.7.2 Auslassen überflüssiger Sperren

Die MultiSSE ermöglicht außerdem eine Verklemmungserkennung und das Auslassen von überflüssigen Sperren („lock elision“). Der MSTG zählt alle Möglichkeiten auf, wie die jeweilige Sperre vom konkreten System verwendet wird. Dadurch sind verschiedene Fehlermuster erkennbar: Zum Ersten ist erkennbar, ob das System mögliche Verklemmungen erhält und auf welchem Pfad diese zustande kommen, sodass die Systementwickler diese entfernen können. Die MultiSSE erkennt dies bereits bei der Konstruktion des MSTG (die system_semantic-Funktion muss diesen Fall behandeln) und gibt eine entsprechende Warnung aus. Zum Zweiten erzwingt AUTOSAR, um unnötige Verklemmungen zu vermeiden, für das Verketten von Sperren eine feste Reihenfolge, die in der Systemkonfiguration spezifiziert werden muss. Mit der MultiSSE gelingt es, diese Reihenfolge bereits zur Kompilierzeit zu prüfen. Zum Dritten zeigt sich auch in der Gegenrichtung, ob eine Sperre jemals sperrt. Ist dies nicht der Fall, kann das Paar aus Sperr- und Entsperraufruf (in der Phase zwischen diesen Aufrufen sperrt die Sperre niemals) oder die Sperre insgesamt (die Sperre sperrt während der gesamten Laufzeit nicht) entfernt werden. Die Sperre *S1* aus dem Beispiel kann so von der MultiSSE als überflüssig nachgewiesen werden.

*Sperrenmodell*

Auch für dieses Muster haben wir ein detailliertes Interaktionsmodell erstellt, dass aber im Gegensatz zum Interruptfall nicht in optimierbaren und unoptimierbare Teile trennt. Wie bereits angedeutet, können – im Falle einer Optimierung – immer das Paar von Sperr- und Entsperraufruf entfernt werden. Die Optimierung gilt also immer für zwei Systemaufrufe gleichzeitig. Ich werde sie im Folgenden mit $\mathrm{Lock}_{\mathrm{opt}}$ kennzeichnen.

## 7.8 Evaluation

*Evaluationsszenarien*

Um die MultiSSE zu evaluieren, haben wir – die Evaluation habe ich in Zusammenarbeit mit Andreas Kässens und Björn Fiedler gemacht – auf eine Vielzahl von Szenarien gesetzt. Wir haben zuallererst den Analysealgorithmus selbst evaluiert:

1. Mit Konformitätstests von Trampoline.
2. Mit synthetischen Benchmarks.

Außerdem hat Andreas Kässens im Rahmen seiner von mir betreuten Masterarbeit auf Basis der Analyse eine Synthese entwickelt, die die eingehende Applikation mit einem maßgeschneiderten AUTOSAR OS kompiliert und linkt. Die Implementierung des OS beruht dabei auf dOSEK [Die19A3.7, HLD+15a]. Die Synthese generiert ein System für einen Raspberry Pi 4B mit vier Cortex-A72-Kernen auf Basis der ARMv7-Architektur. Auf der Kombination von Analyse und Synthese konnten wir weitere Evaluationen durchführen:

3. Ein Test der Beispielanwendung (Abbildung 7.1).
4. Eine für Mehrkernsysteme adaptierte Version des I4*Copter*.

Ich werde die einzelnen Szenarien im Folgenden durchgehen, davor aber noch kurz genauer auf die Synthese und deren Optimierungspotential eingehen.

### 7.8.1 Optimierungspotential

*Formel zur Berechnung*

Um das Potential der Optimierungen zu bestimmen, scheint eine Messung von verschiedenen Anwendungen sinnvoll. Allerdings ist der Effekt der Optimierung – beispielsweise die Verbesserung der Laufzeit – zum einen stark anwendungsabhängig (z. B. „Wie viele vermeidbare IPIs existieren in der Anwendung?“) und zum anderen stark hardwareabhängig (z. B. „Wie lange dauert ein einzelner IPI?“). Wir haben darum Formeln erarbeitet, um den Effekt der Optimierung nach geeigneter Messung der Hardwareparameter und einer vorhergehenden statischer Analyse der Anwendung abschätzen zu können. Wir konzentrieren uns dabei auf die Verbesserung der Laufzeit[38]. Die Formeln berechnen dabei die Laufzeitdifferenz zur Anwendung ohne Optimierung aus. Sie bauen direkt auf den bereits vorgestellen Einzeloptimierungspotentialen $\mathrm{IPI}_{\mathrm{opt}}$ und $\mathrm{Lock}_{\mathrm{opt}}$ auf. Vereinfacht lauten sie[39]:

[38] Bei Echtzeitanwendung ist Zeit eine funktionale Größe und damit explizit vorgegeben, allerdings kann eine statisch feststellbare Laufzeitverbesserung durchaus dazu führen, entweder mehr Aufgaben in der Anwendung zu erfüllen oder schwächere (und damit langsamere) Hardware zu ermöglichen.

[39] Eine deutlich detaillierte Betrachtung findet sich in [›EKF+24].

$$\Delta t_{\text{opt}}^{\text{IPI}} = \underbrace{\sum_{i=0}^{\#\text{Tasks}} \underbrace{n_{\uparrow\downarrow,i} \cdot \underbrace{\text{IPI}_{\text{opt}}}_{\text{pro IPI}}}_{\text{pro Task}}}_{\text{pro Kern}} \qquad \Delta t_{\text{opt}}^{\text{lock}} = \underbrace{\sum_{i=0}^{\#\text{Tasks}} \underbrace{n_{\lightning,i} \cdot \underbrace{\text{Lock}_{\text{opt}}}_{\text{pro Sperre}}}_{\text{pro Task}}}_{\text{pro Kern}}$$

$\#\text{Tasks}$ : Anzahl der Tasks, die auf diesem Kern laufen.

$n_{\uparrow\downarrow,i}$ : Anzahl der vermeidbaren Lockaufrufpaare in diesem Task.

$n_{\lightning,i}$ : Anzahl der vermeidbaren IPIs in diesem Task.

Sie summieren also vor allem alle vermeidbaren Einzeloptimierungspotentiale auf, um die Gesamtheit zu erreichen. Die Formel für die IPI-Berechnung ist sowohl von Sender-, als auch auf Empfängerseite benutzbar.

*Konkrete Werte*

Um die Synthese mithilfe der Formeln zu testen, hat Andreas Kässens zusätzlich Microbenchmarks durchgeführt, um die Werte von $\text{IPI}_{\text{opt}}$ und $\text{Lock}_{\text{opt}}$ für die verwendete Hardware zu messen. Für den ARM basierten Raspberry Pi 4B ergaben sich 1140 Zyklen für $\text{IPI}_{\text{opt}}$ auf Senderseite, 1027 Zyklen auf Empfängerseite und 1514 Zyklen für $\text{Lock}_{\text{opt}}$.

## 7.8.2 Konformitätstests von Trampoline

Trampoline ist eine quelloffene Implementierung des OSEK/VDX- und AUTOSAR-Standards [BBF+06]. Die Quellcodedistribution enthält neben dem eigentlich Code auch eine Reihe von Tests, um die Konformität der Implementierung zu testen. Zwölf dieser Tests betreffen den Mehrkernsupport, bei denen allen wir in der Lage waren, mittels der MultiSSE (ohne Zeitanalyse) einen MSTG zu erzeugen, den wir manuell als korrekt verifiziert haben. Für alle Testfälle haben wir keine überflüssigen Sperren oder Verklemmungen gefunden und daher auf eine weitergehende Synthese verzichtet.

## 7.8.3 Synthetische Benchmarks

*Experimentbeschreibung*

Um die Benutzbarkeit und Korrektheit im Allgemeinen zu zeigen, haben wir außerdem die MultiSSE an synthetischen Benchmarks getestet. Björn Fiedler hat dazu den Benchmark-Generator, der in [DL18] eingeführt wurde, auf Mehrkernsysteme erweitert. Die generierten Systeme werden durch sechs Parameter angepasst:

1. Die Kernanzahl. Wir haben Anwendungen mit 2, 4 und 6 Kernen erzeugt.
2. Die Anzahl an Fäden. Hier haben wir mit bis zu 15 Fäden pro Kern getestet.
3. Die Anzahl an Sperren. Wir haben 1 oder 3 Sperren verwendet.

4. Die Anzahl an Sperrenbenutzern. Dies sind alle Fäden, die auf die Sperre zugreifen. Wir haben dort zwischen 2 und 6 Fäden variiert.

5. Die Anzahl an Aktivierungen von Tasks auf anderen Kernen. Wir haben den Parameter auf 10% gesetzt.

6. Die Anzahl an Events. Dort haben die Anwendungen entweder keine oder fünf Events.

Um die Ergebnisse reproduzierbar zu machen, aber dennoch verschiedene Anwendungen zu generieren, haben wir einen Zufallszahlengenerator mit bekanntem Initialwert verwendet.

*Funktionsweise* Der Generator erzeugt eine Applikation nach den folgenden Schritten: Zuerst erzeugt er einen gerichteten azyklischen Abhängigkeitsgraphen mit so vielen Knoten wie Fäden gefordert sind. Jeder Knoten bzw. Faden bekommt anschließend zufällig eine eindeutige Priorität zugewiesen. Anschließend werden die Task-Aktivierungen basierend auf dem Prozentsatz für Aktivierungen auf anderen Kernen verteilt. Tasks, die zu diesem Zeitpunkt noch nicht aktiviert werden, werden anschließend so generiert, dass sie automatisch starten. Für die Zuteilung von Sperren werden für jede Sperre so viel Tasks auf verschiedenen Kernen ausgewählt, wie als Anzahl an Nutzern spezifiziert ist. Dabei können pro Task sowohl verschiedene Sperren genommen werden, als auch auf einem Kern mehrere Tasks verschiedene Sperren benutzen. Anschließend wird die Anzahl an Events als zusätzliche Abhängigkeit zu Tasks generiert, die anschließend ein Event setzen bzw. darauf warten.

*Ergebnisse* Mit dem Generator haben wir 1 384 Anwendungen erzeugt, die bei der Analyse MSTGs mit 43 - 7 844 Knoten (Durchschnitt: 733 Knoten) und 99 - 95 822 Kanten (Durchschnitt: 5303 Kanten) erzeugt haben. Dabei wurden 1 817 IPIs als vermeidbar erkannt, sowie 4 810 der 15 594 Versuche, eine Sperre zu nehmen, in der Art detektiert, dass es dort niemals einen Wettstreit gibt. Die Laufzeit der MultiSSE auf einer Intel®-Xeon®-Gold-6346-CPU auf den Beispielen lag in einem Bereich von 1 s zu 14217 s mit einem Mittelwert von 721 s (12 Minuten). Bei der zusätzlichen Verwendung von Zeitinformationen für jeden ABB hat sich die Graphgröße im Durchschnitt um 67 Knoten (9 %) und 1123 Kanten (21 %) reduziert. Die MultiSSE mit der Zeitanalyse hatte in diesem Fall eine Laufzeit zwischen 1 s und 12548 s mit einem Mittelwert von 455 s (8 Minuten).

### 7.8.4 Die Beispielapplikation

*Analyseergebnisse* Wir haben außerdem die Beispielapplikation (Abbildung 7.1) implementiert, synthetisiert und evaluiert. Dabei haben wir sowohl ein optimiertes System auf Basis der MultiSSE inklusive Zeitanalyse („*zeitiger MSTG*“, „*zeitiges System*“) wie auch einen ohne Zeitanalyse („*zeitloser MSTG*“, „*zeitloses System*“) erzeugt. Der resultierende zeitlose MSTG hatte einen Umfang von 137 Knoten und 816 Kanten. Die MultiSSE konnte die Sperren-Systemaufrufe von Task *T11* als überflüssig erkennen. Der zeitige MSTG bestand nur noch aus 65 Knoten

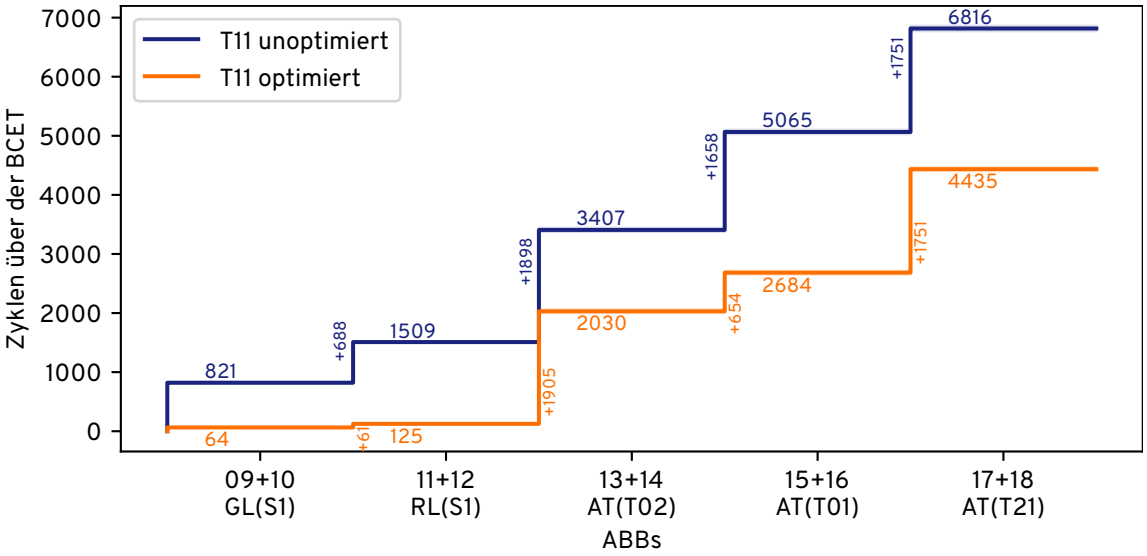



**Abbildung 7.9** Verbesserung der Systemaufrufzeiten für den Task T11 im Beispielprogramm (Abbildung 7.1), adaptiert aus [›EKF+24]

(-53 %) und 196 Kanten (-76 %). In diesem war zusätzlich nachweisbar, dass die Sperre insgesamt überflüssig ist, da alle Sperren-Systemaufrufe global serialisiert passieren. Weiterhin war für den Systemaufruf 🔧 ActivateTask(T01) nachweisbar, dass dieser kein Umplanen auf Kern 0 nach sich zieht und der IPI somit überflüssig ist.

*Syntheseergebnisse*

Bei der Synthese des Systems haben wir die den ABBs zugeordneten Zeiten auf 1000 Zyklen pro Zeiteinheit skaliert. In der Implementierung wartet der jeweilige COMPUTATION-ABB zuerst seine BCET ab. Die Zeit des nachfolgenden „syscall“-ABB wird anschließend als Annäherung an die WCET des COMPUTATION-ABBs gezählt. Als Folge ändern vermiedene oder effizientere Systemaufrufe nicht die BCET (und damit das erforderliche Zeitverhalten), verbessern aber die WCET des Systems. Abbildung 7.9 zeigt die Verbesserung der Ausführungszeit von Task *T11* zwischen dem unoptimierten und dem optimierten System. Zu sehen ist eine Gesamtverbesserung von 2381 Zyklen (35% Verbesserung des Schlupfes, also gegenüber den unoptimierten Zyklen oberhalb der BCET). Trotz vermiedenem Systemaufruf für 🔧 GetSpinlock und 🔧 ReleaseSpinlock benötigt die Anwendung in dieser Zeit 125 Zyklen, was an der Wartefunktion liegt, die ein bisschen länger als die BCET braucht. Die größte Verbesserung ergab dabei der vermiedene IPI mit 1004 Zyklen. Die Formel sagt für diese Verbesserung 1140 voraus, somit liegt die Verbesserung im Rahmen.

Bei der Optimierung der anderen Tasks konnten wir die erwartete Verbesserung nachweisen. *T02* wird nicht mehr von einem IPI unterbrochen und ist damit 900 Zyklen schneller (erwartet: 1027 Zyklen). In *T01* und *T21* konnte die Synthese durch Entfernung der Sperrenaufrufe die Laufzeit um 1415 Zyklen verbessern (erwartet: 1514). Die Unterschiede

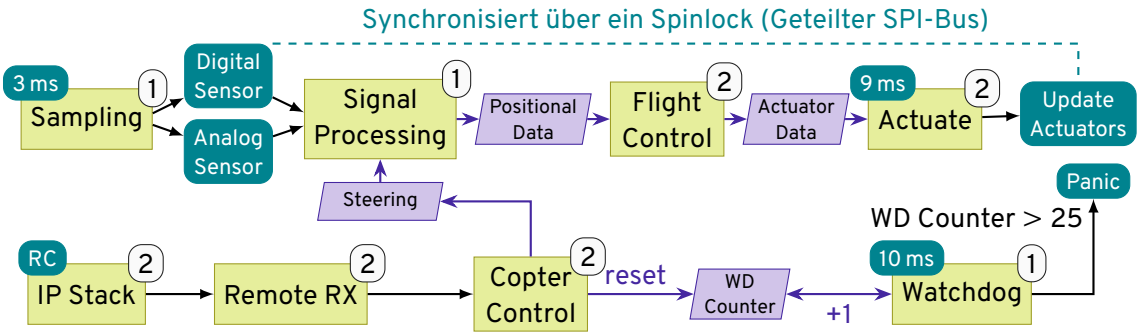



**Abbildung 7.10** Konzeptueller Aufbau des I4*Copter*, adaptiert auf Mehrkernsysteme. Es sind acht Tasks mit ihrem jeweils zugeordneten Kern (oben rechts) dargestellt. Diese kommunizieren über geteilte Daten. Für wichtige periodische Tasks sind die Zeiten angegeben. Das System ist über passenden Treibercode mit dem umgebenden System verbunden. Die Zugriffe auf den SPI-Bus finden synchronisiert über eine Sperre statt. Grafik adaptiert aus [›EFL23].

zwischen Messung und Vorhersage sind auf Caching-Effekte zurückzuführen, bewegen sich aber im akzeptablen Rahmen.

## 7.8.5 Der I4*Copter*

*Anwendungsbeschreibung*

Der I4*Copter* ist eine eingebettete Fluglagensteuerung für einen Quadrokopter, die für ein Einkernsystem auf OSEK/VDX-Basis entworfen wurde. Sie wurde uns freundlicherweise von Ulbrich et al. überlassen [UKH+11]. Für diese Evaluation hat Björn Fiedler die Steuerung auf ein Zweikernsystem portiert, bei dem die zwei Hauptkomponenten auf verschiedenen Kernen laufen: Die Fluglagensteuerung läuft auf dem einen Kern, während der andere Kern die Signalverarbeitung übernimmt. Beide Kerne arbeiten weitestgehend unabhängig voneinander, benutzen aber einen geteilten SPI-Bus. Der Zugriff auf diesen wird als Folge mit einer Sperre exklusiv zugeteilt. Abbildung 7.10 visualisiert den Aufbau.

*Ergebnisse*

Die MultiSSE-Ausführung bei dieser Anwendung auf einem AMD-Ryzen-7-PRO-6850U-System mit 16 Kernen und 32 GiB Hauptspeicher hat 12,57 s Laufzeit benötigt und erzeugt einen MSTG mit 294 Knoten und 2143 Kanten. Bei der zugeschalteten Zeitanalyse reduziert sich der Graph auf 104 Knoten (-65 %) und 104 Kanten (-87 %). Die Analyse hat in diesem Fall gezeigt, dass die Sperre, die den SPI-Bus schützt, nie sperrt. Die Synthese produzierte ein System, dass durch die Einsparung von vier Sperrenaufrufen im Task *Sampling* um 2628 Zyklen (13 %) (erwartet: 3028 Zyklen) schneller war. Im Task *Actuate* konnte ein Sperrenaufrufpaar entfernt werden, was zu einer Reduktion von 1328 Zyklen führte (erwartet: 1514 Zyklen).

## 7.9 Diskussion

Die MultiSSE ist eine Analyse, die den Extremfall einer systematischen Aufzählung aller Systemzustände eines Mehrkernsystems realisiert. Sie kann dabei als Erweiterung der SSE gesehen werden und produziert für den Einkernfall ein gleichwertiges Ergebnis. Damit übernimmt sie sowohl ihre Nachteile aber auch Vorteile, auf die ich im Folgenden eingehe.

*Laufzeitbetrachtung*

Als größter Nachteil kann die weiterhin exponentielle Laufzeit gelten. Dies ist bereits eine Eigenschaft der SSE, wird aber durch die Zwischenkerninteraktionen noch einmal verstärkt, da pro CrossSyscall im Zweifel eine Vielzahl von SPs gefunden werden. Die MultiSSE versucht, die Zustandsexplosion dabei auf ein Minimum zu reduzieren:

- Als erster Schritt arbeitet die Analyse auf ABBs. Die Zusammenfassung aller Anweisungen, die den abstrakten Systemzustand nicht ändern, dient hier der Realisierung einer selektiven Analyse, die unnötige Berechnungen eliminiert.
- Abstrakte Systemzustände, die gleich sind, werden verschmolzen. Dies vermeidet insbesondere die mehrfache (und unnötige) Auswertung von gleichen Pfadstücken, die aber auf verschieden Pfaden erreicht wurden.
- Die Analyse bildet nur SPs, wenn auch wirklich eine Interaktion zwischen den Kernen passiert und auch nur da, wo es möglich ist. Der MSTG agiert dabei als komprimierte Version des Graphen, der das Kreuzprodukt aller Zustände auflistet. Er reduziert dabei die Zustände auf zwei Weisen: Die Trennung zwischen LAbSS und SP sorgt bereits für eine klare Trennung zwischen Zuständen, bei denen die zeitliche Synchronisation eine Rolle spielt und zwischen solchen, wo dies nicht der Fall ist. Es werden also zum Ersten überflüssige Synchronisationszustände vermieden, die Kerne synchronisieren, wo dies nicht notwendig ist. Zum Zweiten werden SPs ausschließlich an Stellen erzeugt, wo die Abhängigkeiten dies zulassen, die der Kontrollfluss bzw. zusätzliche Laufzeitinformationen vorgeben. Das macht die Tatsache explizit, dass Kerne eben nicht beliebig schnell und beliebig unabhängig voneinander laufen.

*Mittel zur SP-Reduktion*

Dadurch, dass die Analyse zur Kompilierzeit läuft, die sehr viel seltener passieren sollte, als die Laufzeit, ist ein lange Analysezeit prinzipiell vertretbar. Die Evaluation zeigt überdies, dass die (nicht auf Geschwindigkeit optimierte) Implementierung zumeist in einem handhabbaren zeitlichen Rahmen abläuft. In zukünftigen Arbeiten wäre eine weitere Reduktion der SPs durch weitere Wissensintegration sinnvoll. Eine Möglichkeit wäre die bereits in Abschnitt 7.6 angerissene Integration von Wissen, das nicht direkt durch den Anwendungscode erkennbar ist. So kann die Anwendung im umgebenden System (oder in der unterliegenden Hardware) eine Aktion auslösen, die eine Rückwirkung auf die Anwendung selbst hat (z. B. ein Interrupt, der nur passiert, wenn das auslösende Gerät vorher aktiviert wurde oder ein Interrupt, der das Ende einer Datenübertragung signalisiert, die aber in der

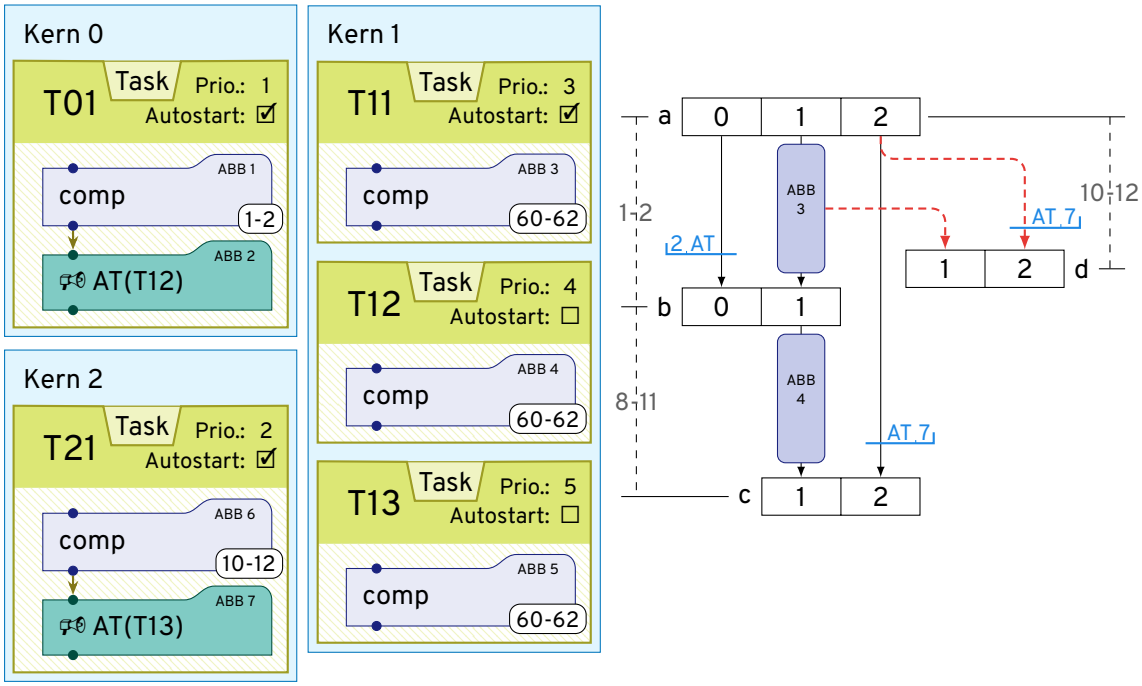



**Abbildung 7.11** Beispiel einer Anwendung, bei der sich SPs im zeitlichen Verlauf gegenseitig ausschließen (hier macht der SP b den SP d unmöglich).

Anwendung erst angestoßen werden muss). Für die Integration in die MultiSSE wäre ein Formalismus wie die *Temporal Logic of Actions (TLA)* [Lam94], *Timed Automata* [AD92, LPY97] oder *Petri-Netze* [RSK03,HBR21] notwendig, um die systemübergreifenden Abhängigkeiten beschreiben zu können. Überdies könnte die Analyse Barrieren berücksichtigen, die vom Anwendungsentwickler vorgegeben werden.

Eine weitere SP-Reduktion könnte durch eine weitere Abstraktionsebene möglich sein: Die SSE benutzt ABBs als Konzept, den Kontrollfluss in betriebssystemrelevante und -irrelevante Teile zu gliedern. Es kann für die MultiSSE sinnvoll sein, analog den lokalen Zustandsgraphen in zwischenkernrelevante und -irrelevante Teile aufzuspalten. Insbesondere habe ich die Beobachtung gemacht, dass CrossSyscalls oft zeitlich (oder logisch) gleichzeitig zu einer ganzen Kette von aufeinanderfolgenden LAbSSs passieren können, was wiederum in einer Menge an SPs resultiert, die zu einer subtil anderen, aber im Gesamten doch sehr ähnlichen Menge an Nachfolgezuständen führen. Hier wäre weitere Forschung denkbar, unter welchen Bedingungen ein Teilgraph von LAbSSs in einen gemeinsamen Metazustand zusammenfassbar wäre, der dann als alleiniger Synchronisationspartner dient und so die Menge der SPs reduziert.

*Verbesserung der Zeitanalyse*

Auch durch eine verbesserte Zeitanalyse lässt sich eine weitere SP-Reduktion erreichen. Momentan führt die Zeitanalyse nur zu einer verringerten Menge an Synchronisationspartnern, indem sie die Zeitverläufe der betroffenen Kerne mit dem originären Kern vergleicht.

Auch andere (nicht betroffene) Kerne können einen Effekt auf einen betroffenen Kern haben, der bei der Synchronisationspartnersuche eine weitere Beschränkung darstellt. Als Beispiel sei hier das System aus Abbildung 7.11 genannt, bei dem der Kern 0 den Task *T12* auf Kern 1 aktiviert (SP b), *zwangsläufig bevor T13* von Kern 2 aktiviert wird (SP c). Nichtsdestotrotz findet die Analyse auch einen SP d zwischen der Aktivierung von *T13* und der Ausführung von *T11*, da sie den ausschließenden Charakter des Systemaufrufs des unbetroffenen Kern 0 (ABB 2) nicht fassen kann. Eine derartige Erweiterung der Analyse ist nicht trivial und bedarf weiterer Forschung: Der Algorithmus muss vor der Synchronisationspartnersuche die Gesamtheit der Vorgänger-SPs kennen, um aus dieser Menge global schließen zu können, dass bestimmte Zustände durch *alle* möglichen SPs so beeinflusst werden, dass sie gar nicht oder mit verringerten Zeitintervall existieren[40]. Notwendig ist dafür eine (aktuell nicht vorhandene) zeitliche Sortierung in der MAbSS-Auswertung.

*Einkern-optimierungen*

Mit der Reduktion auf die SSE im Einkernfall übernimmt die MultiSSE auch sämtliches bereits in der Literatur beschriebenes Optimierungspotential [DHL15,DHL17,Die19], da der MSTG und SSTG in diesem Fall die gleichen Informationen liefert. Konkret ist dies die Vorberechnung von Planerentscheidungen bis hin zu einer in die Hardware integrierten Lookup-Tabelle [DL17]. Weiterhin können auf Basis der Analyse Fehlertoleranzmechanismen integriert werden, die transiente Hardware-Fehler mitigieren. Die Ergebnisse können genutzt werden, um die Abschätzung des schlimmstmöglichen Energieverbrauch signifikant zu verbessern oder den Speicherverbrauch des Systems durch effizientes Teilen des Stapelspeichers zu senken [DL18,WDD+18]. All diese Mechanismen sind auf einzelne Kerne eines Mehrkernsystems anwendbar, nicht nur auf ein mit der MultiSSE analysiertes Einkernsystem. Sie haben aber keinen Vorteil durch die Zusatzinformationen der Inter-Kern-Analyse. Speziell der Einbau von Fehlertoleranzmechanismen sollte aber zusätzlich einfach auf die Interaktion zwischen den Kernen in einem Mehrkernsysteme erweiterbar sein, sodass auch Hardwarefehler, die zu einem Fehlverhalten der Kerne untereinander führen, erkennbar werden. Beispielsweise ist durch die MultiSSE bekannt, dass IPIs zwischen bestimmten Kernen oder zu bestimmten Zeiten nicht existieren, eine Information, die zur Fehlererkennung zur Laufzeit verwendet werden kann.

*Schleifen in der Zeitanalyse*

Ich habe bei der Zeitanalyse Probleme hinsichtlich Schleifen im Programmfluss angesprochen. Hier ist aber noch einmal explizit zu sagen, dass die Zeitanalyse die Analyseresultate zwar weiter verbessert, aber vollständig optional ist. Bedingt durch die Implementierung ist eine Aktivierung sogar pro Synchronisationspartnersuche und dort sogar pro Gleichungsanteil im Ungleichungssystem möglich. Kann also eine Zeit nicht korrekt bestimmt werden, führt dies im Zweifel nur zu einer (geringfügigen) Vergrößerung des MSTG, aber nicht zu einem inkorrekten Graphen.

[40] Zur Komplexitätsverdeutlichung sei im genannten Beispiel angenommen, dass die erste Task-Aktivierung nur bedingt geschähe oder *T01* ggf. abhängig von einem Event auf Kern 0 vor der Aktivierung verdrängt würde.

Durch die erfolgreiche Synthese verschiedener auf der Basis der Analyseergebnisse optimierten Systeme konnten wir den Nutzen des Algorithmus zeigen. Wir haben außerdem ein Verfahren eingeführt, nach geeigneter Vermessung der Hardware-Parameter eine ebenfalls alleinig auf der statischen Analyse basierende anwendungsspezifische Abschätzung der Größenordnung und damit der Sinnhaftigkeit der Optimierung zu gewinnen.

*Forschungsfrage*

Damit kann ich auf meine 2. Forschungsfrage zurückkommen: Ist eine betriebssystemgewahre abstrakte Interpretation auf Mehrkernsystemen ohne Potenzmengenbildung durchführbar? Ich konnte mit der MultiSSE zeigen, dass eine vollständige abstrakte Interpretation durch geschicktes Ausnutzen der tatsächlichen (und selten stattfindenden) Verschränkungen zwischen den Kernen auch ohne Bildung der Potenzmenge möglich ist. Die MultiSSE berechnet dazu bereits im Vorhinein nur Pfade, die aufgrund des vorhergegangenen Kontrollflusses logisch – und zusätzlich optional aufgrund der Ausführungsdauer zeitlich – möglich sind. Ich habe den Algorithmus dazu implementiert und wir haben ihn mit einer darauf aufbauenden optimierenden Synthese mit mehreren Anwendungen als sinnvoll evaluieren können.

# 8

# RTOS-Agnostische Analyse

## Algorithmengeneralisierung und Schnittstellenentwurf

In den letzten beiden Kapiteln wurden zwei konkrete Algorithmen beschrieben, die jeweils spezifische Probleme aus der Domäne gelöst haben. Bereits in den Grundlagen habe ich ARA vorgestellt, das diese und bereits existierende Algorithmen in ein gemeinsames Framework mit dem Ziel einbettet, Systeme und Algorithmen miteinander vergleichbar zu machen. Im Grundlagenteil bin ich allerdings nur auf notwendige Vorverarbeitungsschritte eingegangen. Eine essentielle Aufgabe ist aber auch die Schaffung von RTOS-agnostischen Analysen, die den Algorithmen nicht inhärent ist, sondern aktiv entworfen und evaluiert werden muss. Genau dies ist Thema dieses Kapitels. Ich treffe dazu eine notwendige systematische Klassifizierung der RTOSs wie auch der (alten und neuen) Algorithmen, um zuletzt eine gemeinsame, generische und erweiterbare Schnittstelle zu erstellen und Anwendungen verschiedener RTOS-Schnittstellen miteinander zu vergleichen.

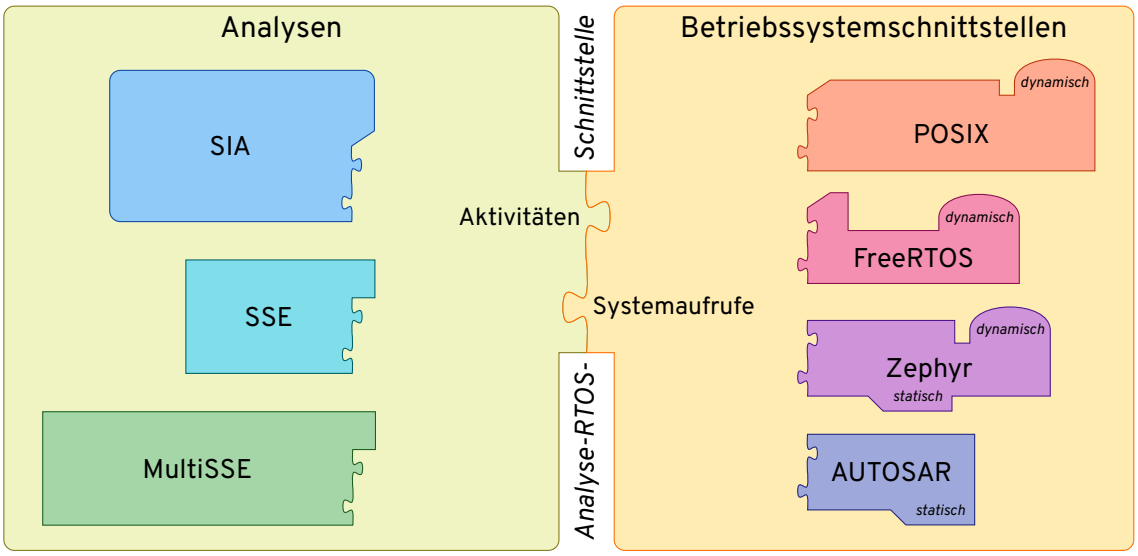



**Abbildung 8.1** Schematischer Aufbau der Analyse-RTOS-Schnittstelle. Die Analysen können über eine gemeinsame Schnittstelle prinzipiell mit beliebigen Betriebssystemschnittstellen kombiniert werden.

*Sinn und Eigenschaften der Schnittstelle*

In den letzten Kapiteln habe ich zwei Analysen vorgestellt und jeweils mit FreeRTOS bzw. AUTOSAR evaluiert. Bislang sind die Analysen aber fest mit den jeweiligen Systemen verknüpft. Es ist beispielsweise mit der momentanen Beschreibung nicht möglich, die Interaktionen des Instanzgraphen aus einer Zephyr-Anwendung zu extrahieren oder eine AUTOSAR-Anwendung mit verschiedenen Analysen zu betreiben.. Eine derartige Generalisierung ist aber ein Grundbaustein, um eine Vergleichbarkeit zwischen verschiedenen Analysen und verschiedenen RTOSs zu schaffen und damit (zukünftige) Optimierungsschritte breit anwenden, testen und beurteilen zu können. Ich habe dazu im Kapitel 5 bereits die Grundlagen von ARA vorgestellt, die es möglich machen, verschiedenartige Analysen in ein einziges gemeinsames Framework einzubetten. Ich habe weiterhin im Kapitel 4 eine theoretische Einordnung der bereits vorhandenen Algorithmen vorgenommen, die feste Grenzen hinsichtlich RTOS-Unterstützung aufzeigen (so kann z. B. die SSE unmodifiziert nur auf Systemen mit statischem RTOS arbeiten). ARA soll aber noch einen Schritt weitergehen und nicht nur verschiedene Analysen anbieten, sondern diese auch (soweit semantisch möglich) mit multiplen Betriebssystemschnittstellen beliebig verschaltbar machen. Konkret soll ARA die folgenden zwei Funktionen bieten:

1. Die gleiche Analyse auf Anwendungen verschiedener Betriebssystemschnittstellen anwenden.

2. Verschiedene Analysen auf der gleichen Anwendung (unter Verwendung genau einer Betriebssystemschnittstelle) ausführen.

Diese Variabilität war bislang nicht möglich, weswegen ich mich in diesem Kapitel dem Problem widme.

*Auftrennung in Analyse und Modell*

Der erste Schritt zur Lösung ist die Auftrennung des gesamten Analysealgorithmus in einen generischen Teil und einen Teil, der spezifisch für die Betriebssystemschnittstelle ist. Im Folgenden werde ich den ersten Teil als *(generischen) Analyseteil* bezeichnen und den zweiten Teil als *Betriebssystemmodell*. Anschließend müssen die beiden Teile über eine gemeinsame *Analyse-RTOS-Schnittstelle* miteinander kommunizieren.

*Schnittstellenentwurf*

Um eine solche Schnittstelle korrekt zu entwerfen, müssen wir uns zuerst die Frage stellen, was Betriebssystemschnittstellen auf der Ebene ihrer Semantik auszeichnet. Wir werden sehen, dass es dort durchaus „kleinste gemeinsame Nenner“ gibt, die ich zur Grundlage der *Analyse-RTOS-Schnittstelle* gemacht habe. Anschließend muss auf dieser Basis der Bedarf der Analysen an die Betriebssystemsemantik evaluiert werden, aus der die Mächtigkeit der *Analyse-RTOS-Schnittstelle* resultiert. Beide Punkte zusammen führen mich zu meiner letzten Forschungsfrage: Welche Gemeinsamkeiten und Unterschiede haben Echtzeitbetriebssystemschnittstellen und wie kann man Analysen davon unabhängig machen? Einen Teil der Lösung – das Auftrennen in Analyse und Systemmodell – habe ich eben bereits beschrieben. Der noch fehlende, wesentliche Teil ist aber der genaue Aufbau der Analyse-RTOS-Schnittstelle. Abbildung 8.1 zeigt die grundlegende Architektur.

Ich werde dazu in diesem Kapitel zuerst auf die Gemeinsamkeiten und Unterschiede von Betriebssystemschnittstellen eingehen, anschließend die Analyseanforderungen herausarbeiten, als Resultat dieser Überlegungen die Implementierung in ARA vorstellen und diese abschließend evaluieren und diskutieren. Dieses Kapitel beruht auf der Vorarbeit, die ich zusammen mit Jan Neugebauer und Daniel Lohmann in [›ENL22] veröffentlicht habe.

Der Gedanke, eine für die Analyse verwendbare Schnittstelle für die Betriebssystemsemantik zu entwerfen, ist nicht neu. Oft wird die Schnittstelle aber nur für ein einziges Betriebssystem geschaffen und heißt dann zusammen mit der Implementierung *Betriebssystemmodell*. Eine große Klasse von Betriebssystemmodellen sind solche, die im Kontext von formalen Methoden verwendet werden, und oft nachweisen sollen, ob das Betriebssystem fehlerhaft arbeitet. Ich bin bereits in Abschnitt 4.1 darauf kurz eingegangen. Betriebssystemmodelle, die eher zum Kontext dieser Arbeit passen, sind die der anderen Analysatoren dOSEK und des RTSCs. Da ARA Algorithmen von dOSEK übernimmt (Abschnitt 4.5), ist die im Folgenden geschilderte Schnittstelle insgesamt ähnlich zu den in der Implementierung verwendeten Mechanismen. Sie erweitert diese aber um einerseits eine (auch in der Implementierung sichtbar) klare Trennung zwischen Analyse und Betriebssystemsemantik, sowie anderseits um Unterstützung für dynamische Systeme (durch Integration des Instanzgraphen) und für Mehrkernsysteme. Der RTSC besitzt ein sehr klar definiertes Modell, auf das ich bereits in Abschnitt 4.4 eingehe. Dieses hat allerdings vor allem das Ziel, die zeitlichen Abhängigkeiten eines RTSs abbilden zu können und bietet keine Abbildung für Betriebssystemkonzepte, die nicht zu den Aktivitäten gehören oder unmittelbar

zur Synchronisierung verwendet werden, wie Warteschlangen, Semaphoren (inkl. des gespeicherten Wertes), Bedingungsvariablen, usw.. Die Kernfunktion, betriebssystemagnostisch zu sein, ist durch die Beschränkung auf den OSEK-OS-Standard für einen Kern nicht gegeben. Der RTSC implementiert zwar im Backend die Möglichkeit, das generierte System auf ein Mehrkernsystem mit unterliegendem AUTOSAR oder POSIX abzubilden, bietet diese Vielfalt aber nicht auf der für ARA gebrauchten Analyseseite an [FKU+16]. ARA implementiert aus diesem Grund eine eigene Schnittstelle, deren systematischen Entwurf ich im Folgenden darstelle.

## 8.1 Echtzeitbetriebssystemschnittstellen

*Eigenschaften*

Wie in den Grundlagen bereits angesprochen (Abschnitt 2.3), habe ich mich bei ARA für die vier Betriebssystemschnittstellen von AUTOSAR, FreeRTOS, Zephyr und POSIX entschieden mit dem Hintergrund der großen Abdeckung von RTOS-Konzepten. Generell unterscheiden sich Echtzeitbetriebssystemschnittstellen in vielen Dimensionen. Sie bieten z. B. verschiedene grundlegende Funktionalitäten (Dateisysteme, Fadenverwaltung, Nebenläufigkeit), unterschiedlich viele Treiber und ggf. ist die Programmiersprache eine andere. Speziell bei RTOSs unterscheiden sich die Systeme zudem in ereignis- und zeitgesteuerte Systeme (Abschnitt 2.2.3). Da das Modell betriebssystemgewahre Analysen unterstützen soll, konzentriere ich mich hier vor allem auf die Dimension der betriebssystemgewahren Analysierbarkeit bei ereignisgesteuerten RTOSs. Die wesentlichen Faktoren, die hier dafür relevant sind, sind die folgenden:

1. Initialisierung des Systems und Initialisierung der Instanzen: Konkret muss eine Analyse sehr verschieden sein, je nachdem, ob das System statische oder dynamische Instanzen besitzt. Sie muss außerdem mit dem jeweiligen Startmechanismus umgehen können: Startet das System im Betriebssystem oder mit einer initialen Funktion? Wann wird der Planer aktiviert, geschieht dies aktiv oder automatisch?

2. Systemkonfiguration: Um passende Informationen für eine möglichst optimale Spezialisierung bieten zu können, muss die Analyse wissen, wie das RTOS vom Systementwickler konfiguriert wurde. Die Analyse kann auf dieser Basis sogar zusätzlich noch die Plausibilität der Konfiguration überprüfen. Diese Mechanismen unterscheiden sich aber von System zu System: Ein System kann mit einer separaten Sprache konfiguriert werden (KConfig bei Zephyr, OIL bei AUTOSAR), mithilfe des Präprozessors (FreeRTOS) oder nicht als Teil der Schnittstelle (POSIX/Linux/BSD).

3. Aktivitäten und deren Planungsstrategie: Alle in dieser Arbeit vorgestellten Analysen gehen davon aus, dass das RTOS Aktivitäten, also Fäden und Unterbrechungen, bereitstellt. Insbesondere die SSE und MultiSSE müssen überdies in der Lage sein, die

Planungsstrategie der jeweiligen RTOS-Schnittstelle auf einen abstrakten Zustand anwenden zu können.

*Unterschiede*

Ich habe bereits im Kapitel 2 die Tabelle 2.2 gezeigt (Seite 24), die die verschiedenen RTOS-Schnittstellen aus dieser Perspektive vergleicht.

*Gemeinsamkeiten*

Wichtig für die Definition der Analyse-RTOS-Schnittstelle sind vor allem konzeptuelle Unterschiede (diese müssen berücksichtigt werden) und Gemeinsamkeiten (diese können als Basis verwendet werden). Bislang habe ich vor allem über die Unterschiede der Systeme gesprochen. Wie aber bereits in den Grundlagen gezeigt (Abschnitt 2.3.5) gibt es zwei ganz zentrale Gemeinsamkeiten, auf denen ich schlussendlich meine Schnittstelle aufbaue:

1. Ein Betriebssystem stellt Aktivitäten (Fäden und Unterbrechungen) zur Verfügung, also Kontrollflüsse, die (größtenteils) unabhängig voneinander ablaufen können und bietet einen Verwaltungsmechanismus (den Scheduler).

2. Die Anwendung kommuniziert mit dem Betriebssystem über Systemaufrufe[41].

Mein Ziel mit der Analyse-RTOS-Schnittstelle ist es, ausschließlich auf diesen beiden Gemeinsamkeiten aufzubauen. Unterschiede werden über das konkrete Modell, aber nicht über die Schnittstelle abgedeckt.

## 8.2 Analyseanforderungen

Nach dem Betrachten des „RTOS-Teil“ der Analyse-RTOS-Schnittstelle fehlt noch der „Analyse-Teil“. Anforderung an die Schnittstelle sind zwangsläufig die Unterstützung der Algorithmen SIA, SSE und MultiSSE. Überdies soll die Schnittstelle aber auch möglichst zukunftssicher die Integration weiterer Algorithmen ermöglichen. Dazu müssen – wie auch schon bei den RTOS-Schnittstellen – Gemeinsamkeiten und Unterschiede der Algorithmen betrachtet werden. Wie in der Einleitung (Abschnitt 1.2) bereits erläutert, haben wir im Forschungsprojekt bereits Grade für Analysen und Synthesen definiert und sie damit hierarchisch eingeteilt. Die Idee für die Schnittstelle ist nun, stets auf dem höchsten (genausten) Grad zu arbeiten. Analysen, die diese Genauigkeit nicht brauchen, verwerfen anschließend die Zusatzinformationen.

Für die Analysen im Detail gelten nun die folgenden Anforderungen an die Schnittstelle.

[41] Hier sei gesagt, dass, selbst wenn der technische Mechanismus der Kommunikation so etwas wie ein Trap ist, es sich konzeptuell um einen Systemaufruf und damit um eine Funktion handelt. Die zentrale Aussage hier ist letztlich, dass die Kommunikation mit dem Betriebssysteme immer über ganz konkrete, statisch bestimmbare Kontrollflusspositionen stattfindet.

## SSE

Die SSE [DHL15] ist eine Analyse auf dem Grad der Interaktionen. Bei detaillierter Betrachtung besteht sie aus zwei Komponenten: Dem Durchlaufen des Kontrollflussgraphen und dem Interpretieren und Auswerten von Systemaufrufen. Dietrich unterteilt den letzteren Teil zudem in die Schritte „Interpretation des Systemaufrufeffekts“ und „abstrakter Lauf des Schedulers“. Diese beide letzten Schritte sind klar abhängig von der Betriebssystemsemantik. Der Kontrollflussgraphdurchlauf (auf Fadenebene) hingegen ist unabhängig vom Betriebssystem[42]. Ein Betriebssystemmodell muss für die SSE also die folgenden Informationen liefern können:

1. Ist ein Funktionsaufruf ein Systemaufruf?
2. Gegeben sei ein abstrakter Zustand: Was ist der Effekt eines bestimmten Systemaufrufs auf diesen Zustand? Um diesen korrekt berechnen zu können, müssen die Systemaufrufargumente bestimmt werden – ein Teil, der wiederum unabhängig von der Betriebssystemsemantik ist, und daher zur Schnittstelle gehört.
3. Gegeben sei ein abstrakter Zustand: Was ist der Effekt eines abstrakten Schedulers?

Die SSE behandelt Interrupts, indem sie in jedem „Computation“-ABB alle davon möglichen aufruft. Auch dieses Wissen muss aus dem Betriebssystemmodell kommen:

4. Welche Interrupts können im System ausgelöst werden?
5. Gegeben sei ein abstrakter Zustand: Was ist der Effekt eines bestimmten Interrupts auf diesen Zustand?

Zu guter Letzt startet die SSE mit einem initialen Systemzustand, den sie anschließend in Folgezustände überführt. Auch der initiale Systemzustand ist stark abhängig vom Betriebssystemmodell:

6. Was ist der initiale Systemzustand?

## MultiSSE

Die MultiSSE ist wie die SSE eine Analyse auf dem Grad der Interaktionen, erweitert diese aber um die Unterstützung von Mehrkernsystemen und Interrupt-Prozessoren. Für die Anforderung an das Betriebssystemmodell ergeben sich daraus ähnliche Fragen:

[42] Das stimmt nicht ganz. Einige Systemaufrufe, die beispielsweise das System ausschalten oder den Scheduler anschalten, bewirken ein Ende des aktuellen Kontrollflusses. Im Betriebssystemmodell sind solche Systemaufrufe gesondert gekennzeichnet und können daher berücksichtigt werden.

7. Ist ein Funktionsaufruf ein Systemaufruf? *Ist ein Funktionsaufruf ein CrossSyscall?*

8. Gegeben sei ein abstrakter Zustand: Was ist der Effekt eines bestimmten Systemaufrufs, *der von einem bestimmten Kern ausgeführt wird,* auf diesen Zustand?

9. Gegeben sei ein abstrakter Zustand: Was ist der Effekt eines abstrakten Schedulers? *Welche Kerne sollen geplant werden?*

Durch die Behandlung von Interrupts mithilfe eines speziellen Interrupt-Prozessors ergeben sich folgende zusätzlichen Anforderungen:

10. Welche Interrupts können im System *nach welchem Muster (z. B. in Form eines synthetischen Kontrollflusses)* ausgelöst werden?

11. *Ist ein Interrupt kernübergreifend?*

12. Siehe Anforderung 5: Gegeben sei ein abstrakter Zustand: Was ist der Effekt eines bestimmten Interrupts auf diesen Zustand?

Auch die MultiSSE braucht das Wissen um den initialen Systemzustand (Anforderung 6).

### SIA

Die SIA liefert Informationen auf dem Grad der Instanzen, braucht also eine Untermenge der Informationen der (Multi)SSE. Konkret sind die Anforderungen, die die SIA an das Betriebssystemmodell stellt:

13. Siehe Anforderung 1: Ist ein Funktionsaufruf ein Systemaufruf?

14. *Gegeben sei der Instanzgraph*: Was ist der Effekt eines bestimmten Systemaufrufs auf diesen Zustand?

Zusammenfassend kann hier gesagt werden: Alle Analysen hängen davon ab, Systemaufrufe von Funktionsaufrufen unterscheiden zu können. Alle Analysen benötigen weiterhin in irgendeiner Form die Auswirkungen des Systemaufrufs auf einen abstrakten Zustand.

## 8.3 Aufbau der *Analyse-RTOS-Schnittstelle*

Ausgehend von den obigen Anforderungen habe ich die Analyse-RTOS-Schnittstelle in Form eines *Betriebssysteminterpreters* gestaltet (die grundsätzliche Idee orientiert sich dabei an der Implementierung in dOSEK [Die19A3.7] und entspricht auch der generellen Vorgehensweise einer abstrakten Interpretation [CC77]). Der Interpreter bekommt einen

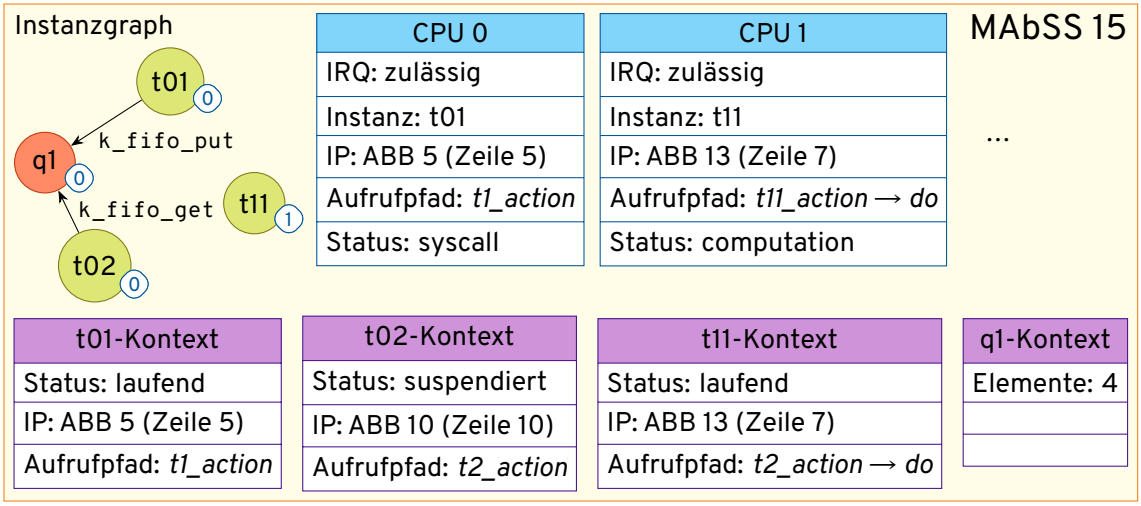



**Abbildung 8.2** Visualisierung eines MAbSS auf Zephyr-Basis, adaptiert aus [›ENL22]. Dargestellt sind die Teile des Zustands: der Instanzgraph, die aktuellen Kernkontexte (CPU 0 bis CPU $n$) und die Instanzkontexte (in diesem Fall drei Zephyr-Tasks und eine Queue. Aktivitäten und CPU-Kontexte haben immer einen *Programmzeiger (Instruction Pointer, IP)* und Aufrufpfad, um die Position im Kontrollfluss zu kennzeichnen.

abstrakten Zustand, um ihn für einen bestimmten Kern zu interpretieren und gibt eine Menge an Nachfolgezuständen aus:

$$Z_{s+1,1}, \ldots, Z_{s+1,n} = \text{interpret}(Z_s, k) \qquad Z: \text{ Zustand}, k: \text{Kern}$$

*Zustandsaufbau*

Als Zustand verwendet die Schnittstelle prinzipiell den MAbSS, den ich bei der MultiSSE präsentiert habe (Definition 24). Es führt aber zusätzlich eine Trennung zwischen Instanz (im Instanzgraph codiert) und Instanzkontext ein. Abbildung 8.2 zeigt ein Beispiel eines solchen MAbSS auf Zephyr-Basis. Der Zustand besteht aus drei Hauptkomponenten:

1. Der Instanzgraph, wie in Abschnitt 6.1 beschrieben. Die Datenstruktur zeichnet aus, dass sie nur wächst, also einmal eingetragene Information unveränderlich ist.
2. Eine Liste an CPU-Objekten. Diese bilden den aktuellen Ausführungszustand ab.
3. Eine Liste an Kontexten. Diese speichern die für das jeweilige Betriebssystemmodell spezifische Instanzkontexte, also alle veränderlichen Daten.

Alle Informationen, die die oben vorgestellten Algorithmen brauchen, sind in diesem Zustand verfügbar: Es ist offensichtlich, dass der Zustand der MultiSSE genügt. Die benötigten Informationen für die SSE ergeben sich aus einem MAbSS mit nur einem Kern. Die SIA benötigt den Instanzgraph, der Teil des Zustandes ist. Die Besonderheit des Zustands ist, dass dort keinerlei Annahme über die Instanztypen getroffen wird. Er stellt nur einen Container für beliebige Instanzen und beliebige Kontexte bereit.

```
class OSBase:
    get_special_steps() -> List[Step]
    get_initial_state(cfg, instances: Graph) -> State

    handle_irq(state, cpu_id: int, irq: int) -> State
    handle_exit(state, cpu_id: int) -> List[State]

    interpret(state, cpu_id: int,
              categories=All) -> List[State]
    schedule(state, cpus=None) -> List[State]

    get_syscalls() -> List[Syscall]
```

**Codeblock 8.1** Methoden der Analyse-RTOS-Schnittstelle

*Aufbau der Schnittstelle*

Auf der Basis des Zustands lässt sich nun die Analyse-RTOS-Schnittstelle definieren, die ein Betriebssystemmodell erfüllen muss, um von einer Analyse genutzt werden zu können. Sie ist in Quellcode 8.1 dargestellt. `interpret()` und `handle_irq()` sind die beiden Funktionen, die den Effekt eines Systemaufrufs bzw. eines Interrupts auf den übergebenen Zustand berechnen. Sie dienen zur Erfüllung der Anforderungen 2, 5, 8, 11, 12 und 14. `schedule()` plant abstrakt einen Zustand und dient zur Erfüllung der Anforderung 3 und 9. `get_initial_state()` liefert den initialen Systemzustand und erfüllt damit die Anforderung 6. Einige Betriebssysteme erfordern noch zusätzliche Verarbeitung. Bei AUTOSAR ist das beispielsweise das Lesen der Systemkonfiguration mithilfe der OIL-Datei. Bei Zephyr werden so statische Instanzen erkannt. Diese Vorverarbeitungsschritte werden per `get_special_steps()` ausgelöst. Die Vorverarbeitungsschritte erkennen überdies vorhandene Interrupts und erfüllen damit die Anforderungen 4 und 10. Zur Erkennung der Systemaufrufe (Anforderung 1, 7 und 13) dient die Funktion `get_syscalls()`.

*konkrete Umsetzung*

Da sowohl für `get_syscalls()` als auch für `interpret()` eine Liste der Systemaufrufe benötigt wird (`interpret()` verzweigt abhängig vom Systemaufruf), habe ich mich etwas syntaktischen Zuckers bedient, um das Leben eines Systemmodellentwicklers zu erleichtern und einen „syscall“-Decorator (ein Konzept der Programmiersprache Python, in der das Modell geschrieben ist) geschaffen, der erlaubt, sowohl die Systemaufrufliste als auch die abstrakte Interpretation des Systemaufrufs in einer Funktion zu kombinieren. Ein Beispiel findet sich in Quellcode 8.2. Es handelt sich dort um die Behandlung von `SetEvent` in AUTOSAR. Gut zu erkennen sind die Metainformationen der Systemaufrufkategorie sowie dessen Signatur. Anschließend spezifiziert der Funktionsname den Systemaufrufnamen und der Funktionskörper den Effekt auf den abstrakten Zustand.

```python
@syscall(categories={SyscallCategory.comm},
         signature=(Arg("task", ty=Task, hint=SigType.instance),
                    Arg("event_mask")))
def SetEvent(cfg, state, cpu_id, args, va):
    task_ctx = state.context[args.task]
    # set the task ready if it already waits
    if task_ctx.status == TaskStatus.blocked and \
        args.event_mask & task_ctx.waited_events != 0:
        task_ctx.status = TaskStatus.ready
        task_ctx.waited_events = 0

    # set the event
    if task_ctx.status != TaskStatus.suspended:
        task_ctx.received_events |= args.event_mask

    # update the instance graph
    cur_task = state.cpus[cpu_id].instance
    for event in get_events(args.event_mask):
        state.instances.add_edge(cur_task, event)

    return state
```

**Codeblock 8.2** Konkrete Implementierung von `SetEvent` im AUTOSAR-Systemmodell

## 8.4 Evaluation der Betriebssystemmodelle

Um die Analyse-RTOS-Schnittstelle zu evaluieren, habe ich diese für AUTOSAR, FreeRTOS, Zeyphr und POSIX implementiert und evaluiert. Ich konnte dabei beim POSIX- und Zephyr-Modell sowie bei der Evaluation auf die Unterstützung von Jan Neugebauer und Kenny Albes zurückgreifen, die diese Aufgaben im Rahmen ihrer von mir konzipierten und betreuten Bachelorarbeiten bzw. in ihrer Funktion als Hilfswissenschaftler erledigt haben. Das Vorgehen war dabei jeweils, Echtweltanwendungen für diese Betriebssystemschnittstellen zu finden, ARA für diese Systeme lauffähig zu machen, und anschließend die möglichen Analysen auf ihnen durchzuführen. Ziel war einerseits die Analyse-RTOS-Schnittstelle und das jeweilige Betriebssystemmodell zu validieren, anderseits aber auch andere Teile von ARA wie die Wertanalyse oder die Vorverarbeitungsschritte zu testen. Die Umsetzung von Optimierungen ist für die Validierung der Analyse dabei zweitrangig, sodass diese nicht im Fokus stand. Es war feststellbar, dass sich einige der Anwendungen sehr „widerspenstig“ ob einer statischen Analyse verhalten, also vor allem sehr viele Entscheidungen dynamisch treffen und Daten dynamisch halten. Wir haben darum überdies einige dieser Anwendungen so modifiziert, dass sie besser analysierbar waren und dennoch lauffähig blieben. Nach der Vorstellung der Evaluation werde ich die Ergebnisse einordnen und diskutieren.

### 8.4.1 Echtweltanwendungen

| | FreeRTOS | | | | Zephyr | | AUTOSAR | POSIX |
|---|---|---|---|---|---|---|---|---|
| Anzahl | GPSLogger | LibrePilot | IronOS | InfiniTime | app_kernel | sys_kernel | I4Copter | libmicrohttpd |
| Codezeilen | 3 235[a]<br>75 508[b] | 78 787[a] | 104 448[a] | 25 434[a]<br>360 896[b] | 814[a] | 768[a] | -[43] | 52 849[a] |
| Basisblöcke | 9 417 | 17 265 | 6 365 | 43 708 | 724 | 726 | 156 | 39 335 |
| Funktionen | 1 117 | 2 673 | 840 | 4 495 | 102 | 97 | 40 | 2 066 |
| Funktionsaufrufe | 100 | 3 105 | 290 | 1 609 | 18 | 38 | 0 | 96 |
| Systemaufrufe | 28 | 223 | 30 | 55 | 51 | 91 | 52 | 88 |
| Maximale Aufrufpfadtiefe | 13 | 8 | 5 | 15 | 2 | 3 | 0 | 7 |

**Tabelle 8.1** Codestatistiken der Echtweltapplikationen (mit ARA extrahiert). Codezeilen sind für die Anwendung[a] selbst und deren Hilfsbibliotheken[b] angegeben.

*Überblick*

Alle Anwendungen, die wir evaluiert haben, finden sich mit ihren Codestatistiken zusammengefasst in Tabelle 8.1. Dort sind ihre grundlegenden Eigenschaften und die verfügbaren Analysen gelistet.

Konkret haben wir folgende Anwendungen untersucht (gruppiert nach RTOS-Schnittstelle):

**FreeRTOS**

Der **GPSLogger** ist eine FreeRTOS-basierte Firmware zum Sammeln, Speichern und Anzeigen von GPS-Daten. Ich habe sie bereits im Abschnitt 6.4.4 vorgestellt. Der GPSLogger zeichnet sich insbesondere durch eine hohe Zahl pseudostatischer Instanzen aus, die alle vor dem Scheduler-Start angelegt werden.

Der **LibrePilot** ist eine Quadrocopter-Firmware. Ich habe sie bereits im Abschnitt 6.4.3 vorgestellt. Auch beim LibrePilot gibt es pseudostatische Instanzen, die aber im Gegensatz zum GPSLogger auch noch von „pseudostatischen“ Daten abhängen (also Daten, die dynamisch sind, es aber nicht sein müssten), was die Analyse erschwert.

[43] Zahl durch die Verwendung einer für die Analysen äquivalenten Portierung unaussagekräftig, da die Originalanwendung mit Clang inkompatibles AspectC++ verwendet.

**IronOS**[44] ist eine Firmware für Lötkolben und steuert beispielsweise den PineCil. Im Gegensatz zu den vorherigen Anwendungen benutzt sie die RISC-V-Architektur.

**InfiniTime**[45] ist eine Firmware für Smart-Watches auf Basis der ARM-Architektur, konkret kontrolliert es die Uhr PineTime. Von den vorgestellten Anwendungen ist sie durch die Verwendung diverser Grafikbibliotheken die größte.

#### Zephyr

Zephyr selbst bringt eine umfangreiche Test-Suite mit. In dieser finden sich auch einige umfangreichere Anwendungen, die als Testfall für die Analysen ARAs gut geeignet sind. Konkret haben wir **app_kernel** und **sys_kernel** als Testfälle untersucht[46]. In der **app_kernel**-Applikation werden alle Instanzen statisch erzeugt, ein Prozess, für dessen Erkennung das OS-Modell in ARA einen gesonderten Vorverarbeitungsschritt benutzt. Während sich die statische Instanzerkennung bei AUTOSAR auf ein korrektes Auslesen der Konfigurationsdatei beschränkt, erfordert dies bei Zephyr eine Analyse der im Code vorhandenen globalen Datenstrukturen, da statische Instanzen über ein spezielles Makro angelegt werden. Die **sys_kernel**-Applikation hingegen legt alle Instanzen dynamisch an.

#### AUTOSAR

Der **I4*Copter*** ist eine Quadrocopter-Firmware, die auf dem AUTOSAR-Standard aufbaut. Ich habe sie bereits auf Seite 140 vorgestellt. Ich habe dabei den I4*Copter* sowohl in der Originalversion [UKH+11] als auch in der für die MultiSSE modifizierten Mehrkernversion untersucht.

#### POSIX

Für POSIX haben wir die Bibliothek **libmicrohttpd** untersucht, die einen einfachen HTTP-Server implementiert[47]. Als konkrete Anwendung haben wir dabei **fileserver_example_dirs** genutzt, die als Beispielanwendung inkludiert ist und die Bibliothek benutzt. Die Implementierung des POSIX-Standards hat die musl libc[48] geliefert.

Die Bibliothek hat sich leider als äußerst „analyse-unfreundlich“ herausgestellt, arbeitet also mit sehr vielen dynamischen Datenstrukturen, weswegen Jan Neugebauer sie, um das Modell dennoch geeignet testen zu können, so modifiziert hat, dass sie – wo möglich – statische Datenstrukturen verwendet. Die Testanwendung ist auch mit der modifizierten Bibliothek weiter lauffähig.

[44] `https://ralim.github.io/IronOS/`, v2.18.2
[45] `https://infinitime.io/`, v0.15.0
[46] Teil von `https://zephyrproject.org/`, v3.7
[47] `https://www.gnu.org/software/libmicrohttpd/` v0.9.73, Commit: 64e91ef6
[48] `http://musl.libc.org/`, basierend auf Commit: aad50fcd, auf ARA angepasst

| RTOS | Anwendung | SIA | INA | SSE | MultiSSE |
|---|---|---|---|---|---|
| FreeRTOS | FreeRTOS | ✓ | ✓* | - | - |
| | LibrePilot | ✓ | ✓* | - | - |
| | IronOS | ✓ | ✓ | - | - |
| | InfiniTime | ✓ | ✗* | - | - |
| Zephyr | app_kernel | ✓ | ✓* | - | - |
| | sys_kernel | ✓ | ✓ | - | - |
| AUTOSAR | I4*Copter* | - | ✓ | ✓ | ✓ |
| POSIX | libmicrohttpd | ✓* | ✓* | - | - |

**Tabelle 8.2** Zusammengefasste Ergebnisse der Anwendung verschiedener Analysealgorithmen auf die vorgestellten Anwendungen. Die Analyse findet entweder alle Instanzen bzw. Interaktionen (✓), funktioniert, findet aber nicht alles (✓*), scheitert an Vorverarbeitungsschritten (✗*) oder ist auf das System nicht anwendbar (-).

Die POSIX-Spezifikation definiert mehr als tausend Funktionen bzw. „Systemaufrufe“. Nach Definition 4 sind aber viele dieser Funktionen keine Systemaufrufe im definierten Sinne. Weiterhin sind einige Systemaufrufe wie `fork` in einem eingebetteten Kontext untypisch. Das POSIX-Modell in ARA implementiert daher nur eine Untermenge an Systemaufrufen, die für eingebettete Systeme sinnvoll ist[49].

## 8.5 Ergebnisse

Die SSE und MultiSSE sind inhärent nur mit einer statischen Betriebssystemschnittstelle benutzbar und damit auf AUTOSAR-Applikationen beschränkt. Eine Ausführung der SIA auf statischen Betriebssystemschnittstellen ist wiederum möglich, aber unsinnig, da Instanzen definitionsgemäß bereits bekannt sein müssen. Der Analysealgorithmus in Kombination mit dem Modell ist nichtsdestotrotz durch die Ausführung der INA evaluierbar, welche Interaktionen zwischen Instanzen erkennt, die auch auf statischen Systemen nicht im Vorhinein bekannt sind.

*Überblick*

Mit diesen Beschränkungen ergeben sich zusammengefasst die Ergebnisse, die in Tabelle 8.2 dargestellt sind. Ich will sie im Folgenden detaillierter beschreiben:

[49] nanosleep, open, pause, pipe, pthread_cancel, pthread_cond_broadcast, pthread_cond_init, Pthread_Cond_Initializer, pthread_cond_signal, pthread_cond_wait, pthread_create, pthread_detach, pthread_join, pthread_mutex_init, Pthread_Mutex_Initializer, pthread_mutex_lock, pthread_mutex_unlock, read, readv, sem_init, sem_post, sem_wait, sigaction, write

### FreeRTOS

*GPSLogger und LibrePilot*

Die Ergebnisse des GPSLogger und LibrePilot habe ich bereits in Abschnitt 6.4 vorgestellt. Hier sei außerdem gesagt, dass in beiden Systemen teilweise Interaktionen zwischen Instanzen nicht zugeordnet werden konnten, da die Wertanalyse die zur Interaktion zugehörige Instanz nicht finden konnte (da auf diese beispielsweise durch eine ummantelnde C++-Klasse zweifach indirekt zugegriffen wird).

*IronOS*

In IronOS hat die Analyse alle 5 Tasks und 1 Queue (verwendet als Semaphore) richtig erkannt. IronOS verwendet FreeRTOS (meistens) nicht direkt, sondern über ein umliegendes „CMSIS RTOS“, insbesondere werden alle Tasks über einen Wrapper erzeugt. ARA kann sie trotz dieser Indirektion korrekt identifizieren. Bei den Interaktionen werden 19 Stück gefunden, die aber durch Systemaufrufe wie `vTaskDelay` erzeugt werden, die auf die Instanz selbst wirken. Andere Interaktionen kann ARA den Instanzen nicht zuordnen, da die Wertanalyse die zugehörige Instanz nicht bestimmen kann.

*InfiniTime*

InfiniTime ist eine per se mittelgroße Anwendung, die aber durch die Verwendung von einer Vielzahl von Bibliotheken (z.B. für Grafik und Bluetooth) sehr groß wird. Dadurch, dass diese Bibliotheken außerdem intensiv mit Callbacks und damit Funktionszeigern arbeiten, hat die SVF den Aufrufgraphen mangels in der LLVM-IR statisch erreichbarer Informationen über die Maßen überapproximiert (also viele Funktionszeiger einer sehr großen Menge an Funktionen zugeordnet). Durch manuelle Annotationen konnte die Größe für die SIA passend reduziert werden, die Interaktionsanalyse benötigt aber impraktikabel lange und wurde daher nicht berücksichtigt. Die SIA liefert insgesamt 6 Tasks, 7 Queues und 3 Mutexe, die die Anwendung auch per manueller Kontrolle erzeugt. Wieder bedingt eine dynamische Zuweisung der Queue- und Mutex-Handhaber die richtige Zuordnung durch die Wertanalyse. ARA weicht in diesem Fall auf eine (eindeutige) Zuordnung zu den lokalen Variablen zusammen mit dem Aufrufpfad aus.

### Zephyr

*app_kernel*

In der app_kernel-Applikation findet die Erkennung der statischen Instanzen 3 Threads, 6 Kernel-Semaphoren, 5 Queues, 3 Pipes und 1 Mutex, die eine manuelle Kontrolle als korrekt identifiziert hat.

Die Objekte sind mit 49 Interaktionen verbunden, von denen die Analyse allerdings bei 2 Interaktionen die Argumente nicht korrekt bestimmen konnte. Diese Interaktionen waren auf die mehrerer Pipes zurückführbar, bei denen die Datenstrukturen dynamisch in einer Liste gespeichert waren und somit der Wertanalyse nicht zugänglich waren.

*sys_kernel*

Im Gegensatz zur app_kernel-Applikation erzeugt die sys_kernel-Applikation ihre Instanzen dynamisch, von denen ARA mithilfe der SIA 22 Threads, 14 Queues, 6 Kernel-Semaphoren und 6 Stacks gefunden hat. Die INA liefert 274 Interaktionen zwischen diesen Instanzen, was eine manuelle Kontrolle bestätigt.

Bei der Analyse hat sich herausgestellt, dass die sys_kernel-Applikation verschiedene Instanzen nacheinander dem gleichem Speicherbereich zuordnet. Die zugrundliegende SIA arbeitet zuerst einmal flussinsensitiv und ist nicht in der Lage, ein solches Muster zu erkennen. Auch der vorgestellte gemischte Modus ist hier keine Hilfe, da es keine das Problem (inhärent) auslösende Systemaufrufe gibt. Das Zephyr-Modell markiert darum diese Objekte als „doppelt“ und unterscheidet die zugehörigen Interaktionen nicht weiter. Um die Instanzen getrennt zu erkennen, müsste die Analyse hier flusssensitiv arbeiten und zwar auch über die Grenzen von SIA und INA hinweg, da Interaktionen nur korrekt erkannt werden können, wenn die „aktuelle“ Instanz bekannt ist – eine Aufgabe für zukünftige Forschung.

**AUTOSAR**

*I4Copter*

Die INA ergibt auf den I4*Copter* angewandt 59 Interaktionen. Die 34 zugrunde liegenden Instanzen erkennt ARA durch das vorherige Auslesen der entsprechenden Konfigurationsdatei.

Die SSE liefert einen SSTG mit 455 263 Knoten und 1 134 793 Kanten. Die SSE-Implementierung von ARA habe ich zusätzlich durch eine Portierung der Unit-Tests aus dem dOSEK-Projekt [DL17] verifiziert.

Auf den MSTG der Mehrkernvariante bin ich bereits in Abschnitt 7.8.5 eingegangen.

**POSIX**

*libmicrohttpd*

Die Ausführung der SIA auf die libmicrohttpd-Anwendung liefert 4 Threads, 1 Pipe, 48 Dateien und 85 Mutexe. ARA findet außerdem 178 Interaktionen, von denen es allerdings bei 39 Interaktionen die beteiligten Instanzen nicht zuordnen kann (erneut bedingt durch die Wertanalyse). Die gefundenen Instanzen sind alle korrekt, allerdings findet ARA einen Mutex nicht, da libmicrohttpd den Mutex-Typen per `typedef` umdefiniert und damit der Analyse unzugänglich macht.

ARA findet bei Ausführung der SIA 47 verschiedene Aufrufkontexte für `fopen` und einen Aufruf von `opendir` und wertet dies als Dateien. Dies entspricht allerdings nicht allen geladenen Dateien, da libmicrohttpd alle Dateien im aktuellen Verzeichnis öffnet, ein Vorgang, der inhärent laufzeitabhängig ist.

## 8.6 Diskussion

Grundsätzlich zeigt die Evaluation, dass die Analyse-RTOS-Schnittstelle funktioniert. Mit ihr ist es erstmals möglich, verschiedene in ARA implementierte betriebssystemgewahre Analysen mit beliebigen Betriebssystemmodellen zusammenzuschalten. Die Evaluation hat aber auch gezeigt, dass es noch einige Herausforderungen gibt.

### Wertanalyse, dynamische Parameter, Objekte und Funktionszeiger

*Dynamik und Analysierbarkeit*

ARA ist ein Framework für betriebssystemgewahre Analysen und versucht deswegen, möglichst keine Instruktionen außer Systemaufrufen zu analysieren. Es benötigt aber dennoch an hauptsächlich zwei Stellen konkrete Werte: Bei Funktionszeigern, um den Aufrufgraphen zu konstruieren und den Systemaufrufparametern. Auch andere statische Analysen wie WCET-Analysen haben an diesen Stellen Probleme. Im industriellen Kontext sind dynamische Datenstrukturen darum vollständig untersagt oder zumindest nicht empfohlen. So rät beispielsweise MISRA C (ein Satz von Regeln für die Entwicklung von eingebetteten Systemen in C) von dynamischem Speicher ab [MIR04]: „Dynamic heap memory allocation shall not be used.“ Holzmann hat für die NASA eine ähnliche Regel formuliert [Hol06]: „Do not use dynamic memory allocation after initialization.“ Praktisch verwenden eingebettete Systeme, wie die analysierten Beispiele zeigen, aber doch Dynamik in Form von z. B. Funktionszeigern und Stapelspeichern. Erschwerend kommt hinzu, dass Programmiersprachen wie C++ Dynamik inhärent eingebaut haben (virtuelle Vererbung verwendet Funktionszeiger) oder die RTOS-Schnittstelle eine solche forciert (in FreeRTOS *müssen* Objekte dynamisch erzeugt werden).

*Verwendung der SVF*

Letztere Herausforderung geht ARA aktiv mit der bereits vorgestellten SIA an. Für die nicht betriebssystemspezifische Dynamik greift es bei der Wertanalyse auf die fortgeschrittene SVF [SX16], die aber dennoch große Probleme bei den analysierten Anwendungen macht. Zum einen werden eine Vielzahl von konkreten Systemaufrufparametern nicht gefunden, sodass ARA auf eine korrekte aber unvollständigere Approximation zurückfallen muss. Zum anderen werden die Ziele von Funktionszeigern nicht erkannt und durch ARA entsprechend überapproximiert, um anschließend in einer komplexer als nötigen SIA bzw. INA zu resultieren: Die Komplexität dieser Analysen hängt direkt vom ggf. durch überapproximierte Funktionszeiger vergrößerten Aufrufgraphen ab.

*Lösungsmöglichkeiten*

Die Lösung kann hier auf zwei Arten geschehen. Einerseits könnte ARA für zukünftige Forschung so erweitert werden, nicht nur eine, sondern verschiedene Werteflussanalysen anzuwenden, in der Hoffnung, präzisere Ergebnisse zu bekommen. Es wird allerdings inhärent immer Werte geben, die eine statische Analyse nicht bestimmen kann. Die Analysen verlieren durch die Behandlung von Dynamik an Präzision und Geschwindigkeit. Sie sind nicht mehr vollständig, aber natürlich immer noch korrekt.

Der andere Weg setzt auf ein Zusammenspiel zwischen statischer Analyse und den Entwicklern der Anwendungen. Diese sind in der Lage, das System „analysefreundlicher“ und „analyseunfreundlicher“ zu gestalten. Verwenden sie also beispielsweise eine betriebssystemgewahre statische Analyse als automatischen ausgeführten Teil der Übersetzung, werden sie das System automatisch auf die Analyse anpassen.

Insgesamt schränkt Dynamik Spezialisierbarkeit ein und ist oft auch noch unnötig. Fiedler liefert dazu eine detaillierte Analyse [Fie23A6].

**Determinismus des Planers**

Gerade für eine interaktionsbasierte Analyse, wie die SSE oder MultiSSE, ist es Voraussetzung, dass der Planer genau spezifiziert ist. AUTOSAR schreibt hier prioritätenbasiertes, partitioniertes Planen vor [AUT13]. RTOSs wie FreeRTOS oder Zephyr sind dort leider unterspezifiziert oder unklarer: In FreeRTOS ist es standardmäßig möglich, Tasks gleicher Priorität zu erstellen, die dann Round-Robin geplant werden. Für eine Analyse bedeutet das eine Zustandsexplosion. In Zephyr ist vollständig unspezifiziert, wie genau das Planen auf mehreren Kernen funktioniert und hängt damit von der konkreten Implementierung der jeweiligen Version ab [Pro23a].

**Widening und Präzision**

Die Analyse-RTOS-Schnittstelle ist zwar in der Lage, Analysen und Betriebssystemmodelle zusammenzuschalten, allerdings setzt eine Analyse wie die SSE statische Systeme voraus und ist damit ohne Weiteres unter FreeRTOS, POSIX und Zephyr nicht lauffähig. Um trotzdem eine interaktionsbasierte Analyse auf diesen Systemen auszuführen, wäre ein geeigneter Widening-Operator notwendig. Dieser könnte insbesondere das Problem umgehen, dass die Instanzanzahl eines bestimmten Systemaufrufs nicht bestimmbar ist. Die SIA leistet hier schon einige Vorarbeit, da sie ein solches Widening für den Instanzgraphen bereits implementiert. Eine mehrfach angelegte Instanz wird dort entsprechend markiert, aber nicht mehrfach im Graphen erzeugt. Bei der Portierung eines solchen Vorgehens ist vor allem die Planung solcher mehrfach oder uneindeutig angelegten Instanzen bzw. Aktivitäten eine Herausforderung: Welche konkrete Aktivität wird lauffähig, falls es potentiell unendlich viele (oder zumindest eine sehr große Menge) dieser Aktivitäten gibt? Blockiert eine uneindeutig angelegte Aktivität höherer Priorität eine eindeutig angelegte Instanz niedriger Priorität? Der naive Ansatz – die Analyse aller Möglichkeiten – führt hier zu einer weiteren Zustandsexplosion. Die genaue Umsetzung ist darum ein Gegenstand weiterer Forschung.

*Vergleichbarkeit durch Schnittstelle*

Die eben geschilderten Herausforderungen betreffen nicht direkt die Analyse-RTOS-Schnittstelle, wären ohne sie aber auch nicht aufgetreten. Der Planer, relevant für die SSE bzw. MultiSSE, ist auf AUTOSAR deterministisch, bereitet aber auf einem Zephyr-System Probleme, das ausschließlich statische Instanzen verwendet. Widening ist auf einem statischen System nicht notwendig. Die Analyse-RTOS-Schnittstelle schafft damit erst die Grundlage für zukünftige Arbeiten, die eine Anpassung der verschiedenen Analysen auf die verschiedenen Betriebssystemschnittstellen erreichen. Sie schafft überdies die Möglichkeit der Vergleichbarkeit verschiedener Systeme, indem sie deren Gemeinsamkeiten herausstellt und

die Analysen diese explizit verwenden lässt. Beispielsweise macht die Schnittstelle eine Messung wie Abbildung 6.3 (Vergleich der SIA-Laufzeiten für verschiedene Anwendungen) überhaupt erst möglich.

*Forschungsfrage*

Kommen wir abschließend zu meiner 3. Forschungsfrage zurück: Welche Gemeinsamkeiten und Unterschiede haben Echtzeitbetriebssystemschnittstellen und wie kann man Analysen davon unabhängig machen? Ich habe in diesem Kapitel die Gemeinsamkeiten (Systemaufrufe, Aktivitäten) und Unterschiede von vier konzeptuell sehr verschiedenen Echtzeitbetriebssystemschnittstellen in Bezug auf eine betriebssystemgewahre Analyse herausgestellt und konnte auf dieser Basis eine Schnittstelle entwickeln, die es ermöglicht, beliebige betriebssystemgewahre Analysen mit einem beliebigen RTOS-Modell zu verschalten. Analysen, Betriebssystemmodelle und die Schnittstelle habe ich weiterhin in einem gemeinsamen Framework, ARA, gekapselt und kombiniert, was den Vergleich verschiedener Analysen und verschiedener Betriebssystemschnittstellen ermöglicht und die zukünftige Erweiterbarkeit um weitere Analysen und Optimierungen sicherstellt und vereinfacht.

# 9

# Zusammenfassung

Ich habe mich in dieser Arbeit mit verschiedenen betriebssystemgewahren Analysen von eingebetteten Systemen beschäftigt. Das Ziel der Analysen ist dabei eine weitergehende Spezialisierung des Systems, indem Anwendung und Betriebssystem möglichst aufeinander maßgeschneidert werden. Dieses Kapitel wird zusammenfassend die aktuellen Probleme beleuchten, die sich daraus ergebenden Forschungsfragen und ihre Lösung vorstellen und anschließend einen Ausblick über anschließende Arbeiten geben.

## 9.1 Ausgangssituation

*abstrakte Interpretation: Astrée und GOBLINT*

Für das Verständnis der aktuellen Probleme mit betriebssystemgewahrer statischer Analyse habe ich zuallererst den Stand der Kunst beleuchtet. Um das Betriebssystem eines eingebetteten Systems – ein Rechensystem, das in ein anderes System eingebettet ist und dort einen vorbestimmten Zweck erfüllt – spezialisieren zu können, ist eine vorhergehende Analyse der Anwendung notwendig. Üblich in diesem Bereich ist die Technik der abstrakten Interpretation [CC77], die den Kontrollfluss der Anwendung Zeile für Zeile traversiert und deren Effekte auf einen abstrakten Zustand berechnet. Zwei große Frameworks, die die abstrakte Interpretation umsetzen, sind Astrée [CCF+05,Min12] und GOBLINT [SSS+21, SVM03]. GOBLINT ist dazu im Gegensatz zu Astrée speziell für mehrfädige Systeme entworfen worden und führt zusätzlich zur abstrakten Interpretation noch ein weitergehendes Konzept, die Beschränkung mit Nebenbedingung, ein. Beide Frameworks sind allerdings darauf ausgelegt, Laufzeitfehler in Anwendungen zu finden und daher nicht geeignet, die nötigen Informationen für eine Spezialisierung des Betriebssystems zu liefern.

*betriebssystemgewahr: RTSC und dOSEK*

Im Gegensatz dazu sind der RTSC [Sch11] und dOSEK [Die19A3.7,HLD+15a] speziell darauf ausgelegt, die Anwendung ob der Verwendung des Betriebssystems zu analysieren, um das System anschließend transformieren zu können. Der RTSC transformiert die Anwendung dabei von einem ereignisgesteuerten in ein zeitgesteuertes System (und verteilt es ggf. auf mehrere Kerne), während dOSEK eine Vielzahl von Spezialisierungen sowohl der Anwendung als auch des Betriebssystems erlaubt. Beide Frameworks sind allerdings auf den veralteten OSEK-RTOS-Standard [OSE05] ausgelegt, ein Betriebssystem mit statischen Instanzen (also der Anforderung, alle für das Betriebssystem relevanten Objekte bereits vor der Laufzeit anzulegen), das außerdem nur auf Einkernsystemen funktioniert.

*RTOS-Schnittstellen*

Ich habe weiterhin noch die Domäne der RTOSs betrachtet, die üblicherweise bei eingebetteten Systemen eingesetzt werden. Diese spannen ein Spektrum an Funktionalität auf, die sich auf für die Analyse wichtige Kerneigenschaften reduzieren. Ich habe das sehr dynamische FreeRTOS und POSIX vorgestellt, bei der alle Instanzen zur Laufzeit angelegt werden müssen, den AUTOSAR-Betriebssystemstandard [AUT13] (den Nachfolger von OSEK) am anderen Ende des Spektrums, der zwar ausschließlich statische Instanzen, dafür aber Mehrkernsysteme unterstützt und Zephyr [Pro22] zwischen den Extremen, das statische und dynamische Instanzen und Mehrkernsysteme unterstützt.

*Probleme*

Zusammengefasst existieren also sowohl RTOSs, die dynamisch sind, als auch solche, die mehrere Kerne unterstützen, aber keine betriebssystemgewahren Analysen, die Anwendungen auf dieser Basis analysieren. Weiterhin sind die bestehenden Analysen und Analyseprogramme auf ein Betriebssystem beschränkt oder verfolgen nicht den Zweck, dieses automatisiert zu verbessern. Eine konkrete Entkopplung von Betriebssystemmodell und Analysealgorithmus fehlt. Genau in diesem Feld habe ich meine Forschungsfragen und die jeweiligen Lösungen angesetzt, auf die ich im nächsten Teil zu sprechen komme.

## 9.2 Forschungsfragen

Die eben vorgestellten Probleme habe ich in drei Forschungsfragen kristallisiert, deren Beantwortung jeweils einen Teilbereich abdeckt. Um alle Forschungsfragen beantworten zu können, habe ich mit ARA ein neues Analyse- und Synthese-Framework entwickelt. ARA ist in der Lage, auf Anwendungen eine Vielzahl beliebig zuschaltbarer Analysen auszuführen, die sich ein gemeinsames Betriebssystemmodell teilen. Die Analyseergebnisse werden anschließend in einer ebenfalls inkludierten Synthese (nicht Teil dieser Arbeit) zur Spezialisierung des Betriebssystems für die jeweilige Anwendung benutzt.

**1. Forschungsfrage: Welche Herausforderungen entstehen auf eingebetteten dynamischen Systemen und können betriebssystemgewahre Analysen diese überwinden?**

*Überblick*

Eine abstrakte Interpretation ohne Widening-Operator wie die SSE, die Planerentscheidungen berücksichtigt, ist an statische Systeme, also Systeme, die ihre Instanzen zur Kompilierzeit deklarieren, gebunden, um verlässlich zu terminieren. Bei dynamischen Systemen sind die genauen Instanzen hingegen erst zur Laufzeit festgelegt und damit den verwandten Arbeiten nicht zugänglich. Sie erfordern eine Analyse, die Instanzen, bei denen die Existenz oder deren Anzahl unklar ist, zusammenfasst, theoretisch durch das Konzept des „Widening“ in der abstrakten Interpretation abgedeckt [RY20A2.3.4].

*Analyse: SIA & INA*

Mit der statischen Instanzanalyse und der Interaktionsanalyse habe ich zwei Analysen entworfen, die dazu in der Lage sind, indem sie den Anwendungscode in den Instanzgraphen zusammenführen, eine Datenstruktur, die die Menge aller Instanzen und deren Interaktionen flussinsensitiv aufzeigt. Ich habe für die Analysen drei verschiedene Algorithmen bzw. Modi entwickelt, die sich in Laufzeit und Mächtigkeit unterscheiden. Der erste Algorithmus traversiert den Kontrollflussgraphen gemäß der abstrakten Interpretation und ist dadurch selbst flusssensitiv (auch wenn er eine flussinsensitive Datenstruktur erzeugt), aber langsam. Der zweite Algorithmus traversiert ausgehend von den Systemaufrufen rückwärts den deutlich kleineren Aufrufgraphen, ist dadurch flussinsensitiv und deutlich schneller (bei den getesteten Anwendungen bis zu Faktor 11). Je nach Betriebssystemschnittstelle gibt es allerdings Systemaufrufe, die nicht atomar sind (mehrere zusammenhängende Systemaufrufe also z. B. die gleiche Instanz definieren). Damit ist deren Reihenfolge wichtig und ein flussinsensitiver Algorithmus für die Analyse ungeeignet. Ich habe darum auch noch einen dritten Algorithmus entworfen, der die beiden obigen Verfahren kombiniert und nur da auf das flusssensitive Verfahren umschaltet, wo dies notwendig ist. Er vereint damit die Vorteile beider Algorithmen, ist schnell und liefert ein vollständigeres Ergebnis.

*Ergebnis*

Zusammen mit einer Synthese von Björn Fiedler, die vormals dynamische Instanzen in statische Instanzen umwandelt, konnte ich einen praktischen Einsatz des Analyse-Algorithmus zeigen, der in zwei untersuchten Echtweltanwendungen für FreeRTOS bis zu 44% Verbesserung der Initialisierungsphase geführt hat. Wir haben in diesem Zuge außerdem

festgestellt, dass viele Anwendungen zwar auf eine dynamische Betriebssystemschnittstelle zurückgreifen, die Dynamik aber nicht verwenden, sondern alle Instanzen in der Initialisierungsphase anlegen und dieses Konzept eine „pseudostatische“ Instanz genannt.

Ich konnte die SIA und INA überdies durch die gemeinsame Analyse-RTOS-Schnittstelle auf 7 weiteren Anwendungen auf Zephyr-, AUTOSAR- und POSIX-Basis anwenden.

**2. Forschungsfrage: Ist eine betriebssystemgewahre abstrakte Interpretation auf Mehrkernsystemen ohne Potenzmengenbildung durchführbar?**

*Überblick* Die abstrakte Interpretation wie initial formuliert [CC77] ist ausschließlich für Einkernsysteme geeignet. Da sie einen fortwährenden Zustand über jede Anweisung der Anwendung entwickelt, ist sie inhärent flusssensitiv: Die Reihenfolge der Zustände ist essentiell für die Analyse. Bei einem System mit mehreren, echt parallelen Kernen ist jedoch diese Eigenschaft nicht mehr gegeben: Es ist im allgemeinen unklar, welche Anweisung der eine Kern parallel zu dem anderen ausführt. Der naive Ansatz ist darum die Ausführung der Analyse pro Kern und der Bildung der Kreuzproduktes aller Zustände, wie auch von Miné erkannt [Min15]. Eine betriebssystemgewahre Analyse wie z. B. die SSE hat das gleiche Problem [DL17]. Seidl [SVM03] und Miné [Min12] lösen das Problem darum mit zwei Ansätzen, die über Beschränkungen mit Nebenbedingung oder der Aufgabe der Flusssensitivität für globale Fakten funktionieren.

*Analyse: MultiSSE* Ich konnte in dieser Arbeit zeigen, dass dies für eine betriebssystemgewahre Analyse nicht notwendig ist. Dazu habe ich die MultiSSE entwickelt, die die SSE zur Ausführung auf mehreren Kernen erweitert. Sie nutzt aus, dass es nur ganz spezielle Stellen – Systemaufrufe – im Anwendungscode gibt, die eine Aktion auf anderen Kernen auslösen, an denen mehrere Kerne miteinander interagieren. Sie begrenzt die Schwierigkeit, die jeweiligen Positionen im Anwendungscode von verschiedenen Kernen zueinander zu bestimmen, auf eben jene Stellen – Synchronisationspunkte in der Nomenklatur der MultiSSE. Davon ausgehend kann die MultiSSE durch zwei Mechanismen die Zustandsanzahl auf einem handhabbaren Niveau halten. Die Analyse bildet dazu erst gar keine Synchronisationspunkte, die durch bereits gefundene Synchronisationspunkte ausgeschlossen werden können. Zum einen geschieht dieser Ausschluss auf Basis des Kontrollflusses: Ist ein Systemaufruf, der einen Synchronisationspunkt bewirkt, bereits durchlaufen, müssen sich alle Kerne in dem jeweiligen Kontrollfluss *dahinter* befinden. Zum anderen kann die MultiSSE optional die Ausführungszeit der Anwendung mit einbeziehen: Sie kann dadurch bestimmen, ob bestimmte Kontrollflusspositionen zueinander auf Basis des letzten gemeinsamen Synchronisationspunkt zeitlich gleichzeitig überhaupt möglich sind.

*Ergebnis* Das Resultat der Analyse ist ein kombinierter Graph aus Einkernzuständen und Synchronisationspunkten, der alle möglichen Abläufe der Anwendung beschreibt. Auf dieser Basis sind weitere Spezialisierungen möglich. Ich habe dazu zwei mehrkernsystemspezifische Spezialisierungen entworfen, die ich zusammen mit Björn Fiedler und einer Synthese von

Andreas Kässens erfolgreich evaluieren konnte. Wir haben weiterhin die Funktionsweise der Analyse mit einer Vielzahl von händischen und synthetisch erzeugten Testfällen validiert und verifiziert.

**3. Forschungsfrage: Welche Gemeinsamkeiten und Unterschiede haben Echtzeitbetriebssystemschnittstellen und wie kann man Analysen davon unabhängig machen?**

*theoretischer Kontext, ARA*

Um verschiedene Analysen auf Anwendungen verschiedener RTOSs laufen lassen zu können, müssten die betriebssystemspezifischen Teile vom restlichen Analysealgorithmus entkoppelt werden. Ich habe diese dazu zuerst in einen theoretischen Kontext gebettet, um anschließend die von den Analysen verwendeten abstrakten Zustände zu vereinen, betriebsspezifische Teile in ein Betriebssystemodell zu kapseln (ein Modell pro RTOS-Schnittstelle) um zuletzt alle Modelle und alle Analysen über eine Analyse-RTOS-Schnittstelle in dem gemeinsamen Framework ARA kombinierbar zu machen.

*Eigenschaften*

Die Schnittstelle baut auf den zentralen Gemeinsamkeiten aller betrachteten RTOSs auf: Der Existenz von Systemaufrufen, von Aktivitäten und eines Planers. Auf dieser Basis funktioniert das konkrete Betriebssystemmodell im Sinne der abstrakten Interpretation als abstrakter Betriebssysteminterpreter, der bei der Auswertung eines Systemaufrufs aktiv wird: Dazu berechnet es die Systemaufrufeffekte auf einem mitgeführten und zwischen den Modellen vereinheitlichten Zustand und emittiert passende Folgezustände. Der Zustand umfasst dabei die Gesamtheit der von den Analysen benötigten Informationen, von denen die für die konkrete Analyse nicht benötigten Teile jeweils ignoriert werden. Auf dieser Basis können Analysen und Betriebssystemmodelle anschließend beliebig verschaltet werden.

*Ergebnis*

Um die Schnittstelle zu evaluieren, habe ich alle möglichen Analysen auf einer Vielzahl von Anwendungen aller unterstützten RTOS ausgeführt. Ich konnte erfolgreich die Interaktionsanalyse, SIA, SSE und MultiSSE auf alle unterstützten Anwendungen anwenden und die für die Spezialisierung notwendigen Informationen extrahieren. Die Auswertung hat ebenfalls ergeben, dass – bedingt durch die Komplexität des Datenflusses einiger Anwendungen – teilweise Interaktionen nicht auffindbar waren, ein Resultat, dass aber auf Probleme mit der Wertanalyse (für die ARA eine Drittanbieterbibliothek verwendet) zurückzuführen ist und kein Problem der Analyse-RTOS-Schnittstelle ist. Insgesamt ermöglicht die Schnittstelle erstmals eine Vergleichbarkeit von Anwendungen verschiedener RTOS miteinander und öffnet insgesamt die Tür für eine Vielzahl weiterer Spezialisierungen, auf die ich im folgenden Ausblick weiter eingehen werde.

## 9.3 Ausblick

*Verwendung als Hinweisgeber, analysierbares RTOS*

Ich habe mit ARA ein RTOS-agnostisches Framework zur betriebssystemgewahren Analyse zum Zwecke der Spezialisierung geschaffen. Diese Arbeit legt dadurch eine Grundlage für

weitere Forschung. So gehört die Schaffung eines *Hardware/Application-Aware Operating System (HAAOS)* zum überliegenden Forschungsprojekt, dass einen Satz von Betriebssystembausteinen schaffen sollen, die zu der von Anwendung und Hardware geforderten Funktionalität zusammengefügt werden können. Das Wissen um die von den Anwendungen verwendeten Konzepte, sowieso die Existenz einer gemeinsamen Schnittstelle liefert hier einen zentralen Teil. Mit dem Wissen um die Notwendigkeiten der Analyse wäre zudem eine auf gute Analysierbarkeit zugeschnittene Betriebssystemschnittstelle denkbar. Sie würde damit „zero cost abstractions“ in Programmiersprachen ähneln, Abstraktionen in Hochsprachen, die zur Entwicklungszeit Konzepte bereitstellen, die sich nicht auf die Laufzeit auswirken, also („weg-“)optimierbar sind (z. B. in Rust [Tea24]). Grundsätzlich habe ich bei den untersuchten Anwendungen die Erfahrung gemacht, dass Entwickler von eingebetteten Systemen oft ein bestimmtes Konzept wollen (z. B. statische Instanzen), dieses aber in der Schnittstelle nicht oder nur viel komplizierter ausdrücken können. Eine Betriebssystemschnittstelle sollte daher diese Konzepte abdecken oder erzwingen. Hier kann die Analyse wieder unterstützend wirken, indem sie Anwendungsentwickler auf entsprechende (wahrscheinlich fehlbenutzte) Konzepte hinweist, wie dies Code-Linting-Werkzeuge machen [Web97].

*Widening in der SSE und MultiSSE*

Die Analyse-RTOS-Schnittstelle schafft zwar die einfache (technische) Möglichkeit, Algorithmen wie die SSE, SSF oder MultiSSE auf dynamischen Systemen anzuwenden, allerdings verbietet der Algorithmus selbst ein solches Vorgehen. Für zukünftige Arbeiten wäre darum eine Erweiterung der SSE um Widening in Hinblick auf unsichere Instanzen wünschenswert. Ich habe zu dem Thema bereits geforscht. Das Grundproblem hier ist die Ausführung des abstrakten Planers, wenn unsichere Instanzen (also in Existenz oder Anzahl unbestimmt) in der zu planenden Menge sind. Am vielversprechensten scheint hier eine partielle Analyse zu sein, die nicht wie die SSE, SSF oder MultiSSE das ganze System auf einmal untersucht, sondern auf Basis einer vorgegebenen oder sogar automatischen Entscheidung nur „wichtige“ Teile mit einem flusssensitiven möglichst detaillierten Algorithmus durchsucht und ansonsten einen einfacheren (ggf. flussinsensitiven) Algorithmus verwendet. Das Vorgehen erinnert damit ein bisschen an den gemischten Modus der SIA und INA, ist aber durch die Berücksichtigung des Schedulers deutlich komplexer.

*Globales Fixpunktverfahren*

Bislang geht ARA schrittbasiert vor: Es wendet notwendige Analyseschritte passend hintereinander an. Manchmal ist aber ein Fixpunktverfahren, dass sich über mehrere Analyseschritte spannt, sinnvoll. Beispielsweise bestimmt die Wertanalyse den Wert eines Funktionszeigers durch die Analyse des Werteflussgraphen, der seinerseits aber wieder vom Aufrufgraphen abhängt, der eine Analyse der Funktionszeiger benötigt. Um den Algorithmus hier vollständig zu halten, macht die Analyse in diesem Fall eine Überabschätzung. Eine wechselseitige Zeigeranalyse und Aufrufgraphkonstruktion kann hier aber zu korrekteren Ergebnissen führen. Dieses Thema können wir noch zwei Schritte weiter denken: Zum einen kann auch die erneute Analyse des bereits optimierten Systems Sinn ergeben. So

senken die Optimierungen auf Basis der MultiSSE beispielsweise den zeitlichen Pessimismus, der im Kontrollfluss zugrunde liegt und schaffen damit Eingangsbedingungen für die MultiSSE, die eventuell noch Folgeoptimierungen ermöglichen. Zum anderen kann diese Rückkopplung die Optimierungskriterien ändern: Die optimale Optimierung kann dann u. U. gar nicht die Optimierung sein, die für diese eine konkrete lokale Stelle optimal ist, sondern eine, die das Systemverhalten in den erlaubten Grenzen so ändert, dass Folgeoptimierungen insgesamt besser werden. Beispielsweise kann eine (durch eine Synthese erzwungene) aktive Verlängerung der Ausführungszeit an einer punktuellen Stelle dazu führen, dass der Kontrollfluss zwischen den Kernen so verschoben wird, dass in einem höherprioren Task insgesamt weniger potentielle Unterbrechungen stattfinden. Es ist außerdem möglich, dass eine aktive Verlängerung der Ausführungszeit dafür sorgen kann, dass das Gesamtsystem überhaupt erst analysierbar wird, da es den aus der MultiSSE resultieren Graphen insgesamt deutlich verkleinert.

## 9.4 Fazit

Ich konnte mit dieser Arbeit zeigen, dass eine betriebssystemgewahre Analyse auch auf dynamischen System und Mehrkernsystemen möglich ist. Weiterhin habe ich den Ansatz verallgemeinert, sodass verschiedene Analysen betriebssystemagnostisch Anwendungen untersuchen können, die gegen verschiedenartige RTOS-Schnittstellen geschrieben wurden.

All das legt die Grundlage zu weitergehenden Optimierungen (erste Ansätze sind in dieser Arbeit bereits beschrieben) und schafft die zusätzliche Möglichkeit der Vergleichbarkeit von Echtzeitsysteme durch ein generisches Framework.

# A

# Anhang

# A.1 Konkrete Systemaufrufe

In diesem Abschnitt gebe ich eine kurze Übersicht über einige konkrete Systemaufrufe der in dieser Arbeit verwendeten RTOSs, wie sie vor allem in den Beispielen benutzt werden. Zumeist sollten die Namen selbsterklärend sein, hier folgt aber trotzdem eine Übersicht kategorisiert in Aktivitäten, Synchronisationsprimitive und Kommunikationsmittel.

## A.1.1 AUTOSAR

**Fäden**

- `ActivateTask` Einen neuen Job aus einem Task erzeugen.
- `TerminateTask` Sich selbst terminieren.
- `ChainTask` `TerminateTask` und `ActivateTask` atomar aufrufen.

**Synchronisation**

- `EnableAllInterrupts` Interrupts anstellen.
- `DisableAllInterrupts` Interrupts ausstellen.
- `GetResource` Eine Ressource anfordern, einen kritischen Bereich betreten.
- `ReleaseResource` Eine Ressource freigeben, einen kritischen Bereich verlassen.
- `GetSpinlock` Einen Lock anfordern, einen kritischen Bereich betreten.
- `ReleaseSpinlock` Einen Lock freigeben, einen kritischen Bereich verlassen.

**Kommunikation**

- `WaitEvent` Den aktuellen Job warten lassen, bis eine andere Aktivität das Event setzt.
- `ClearEvent` Bereits signalisierte Events aus der Eventmaske entfernen.
- `SetEvent` Bei einem Job ein Event setzen und diesen damit ggf. aufwecken.

## A.1.2 FreeRTOS

**Fäden**

- `xTaskCreate` Einen Thread erzeugen.
- `xTaskCreateStatic` Einen Thread mit statischem Stack erzeugen.
- `vTaskDelete` Einen (beliebigen) Thread beenden und löschen.
- `vTaskDelay` Einen Thread für eine bestimmte Zeit schlafen legen.

- vTaskSuspend Einen Thread für unbegrenzte Zeit schlafen legen.
- vTaskResume Einen Thread aufwecken.

**Synchronisation**

- xSemaphoreCreateCounting Eine zählende Semaphore anlegen.
- xSemaphoreCreateBinary Eine binäre Semaphore anlegen.
- xSemaphoreCreateMutex Einen Mutex anlegen.
- vSemaphoreDelete Eine Semaphore löschen.
- xSemaphoreTake Eine Semaphore nehmen (`p()`).
- xSemaphoreGive Eine Semaphore freigeben (`v()`).

**Kommunikation**

- xTaskNotifyGive Einem Thread ein Signal schicken.
- ulTaskNotifyTake Der aktuelle Thread wartet auf ein Signal.

### A.1.3 Zephyr

**Fäden**

- k_thread_create Einen Thread dynamisch erzeugen.
- K_THREAD_DEFINE Einen Thread statisch erzeugen.
- k_sleep Für eine bestimmte Zeit (passiv) warten.
- k_wakeup Einen schlafenden Thread aufwecken.
- k_yield Einen (Re)Schedule auslösen.
- k_thread_suspend Einen Thread für eine bestimmte Zeit schlafen legen.
- k_thread_resume Einen Thread für unbegrenzte Zeit schlafen legen.

**Synchronisation**

- k_sem_init Eine Semphore dynamisch anlegen.
- K_SEM_DEFINE Eine Semphore statisch anlegen.
- k_sem_take Eine Semaphore nehmen (`p()`).
- k_sem_give Eine Semaphore freigeben (`v()`).
- k_mutex_init Einen Mutex dynamisch anlegen.
- K_MUTEX_DEFINE Einen Mutex statisch anlegen.

- `k_mutex_lock` Einen Mutex sperren.
- `k_mutex_unlock` Einen Mutex entsperren.

**Kommunikation**

- `k_condvar_init` Eine Bedingungsvariable dynamisch anlegen.
- `K_CONDVAR_DEFINE` Eine Bedingungsvariable statisch anlegen.
- `k_condvar_wait` Auf eine Bedingungsvariable warten.
- `k_condvar_signal` Eine Bedingungsvariable setzen.

### A.1.4 POSIX

**Fäden**

- `pthread_create` Einen neuen Thread erzeugen.
- `pthread_join` Auf das Ende eines Threads warten.
- `pthread_detach` Einen Thread von seinem Erzeuger entkoppeln.
- `pthread_cancel` Einen Thread beenden.

**Synchronisation**

- `pthread_mutex_init` Einen Mutex anlegen.
- `pthread_mutex_lock` Einen Mutex sperren.
- `pthread_mutex_unlock` Einen Mutex entsperren.
- `sem_init` Eine Semaphore anlegen.

**Kommunikation**

- `pthread_cond_wait` Auf ein Signal eines anderen Threads warten.
- `pthread_cond_signal` Ein Signal an einen anderen Thread senden.

## A.2 Verwendete Software

Diese Arbeit wäre nicht möglich gewesen ohne eine Vielzahl an bereits existierender quelloffener Software, auf der ich aufbauen oder die ich verwenden konnte. Ich möchte diese daher ohne den Anspruch auf Vollständigkeit hier explizit listen und mich bei allen Entwicklern für ihre bezahlte oder unbezahlte Arbeit bedanken. In dieser Arbeit wurden verwendet:

Boost, ConTEXt, Cython, dOSEK, FreeRTOS, GCC, Gentoo Linux, git, GPSLogger, graph-tool, Helix, InfiniTime, IronOS, KDE/Plasma Desktop, Kitty, LaTeX, LaTeX Beamer, LibrePilot, LLVM/Clang, Lua, Meson, musl libc, Neovim, numpy, Okular, OpenSSH, pydot, Python, SVF, TEX, Ti*k*Z/PGF, Versuchung, Zephyr, zsh

## A.3 Artefakte

Zu guten wissenschaftlichen Papieren gehört auch immer eine zusätzliche Bereitstellung der Artefakte. Ich will dieses Vorgehen daher hier übernehmen. Die Ergebnisse dieser Arbeit sind (bis auf die Synthese des LibrePilot und des I4*Copter*) mit dem aktuellen Stand von ARA reproduzierbar. Sie sind in Form von Experimenten unter Verwendung des Versuchung-Werkzeugs erreichbar [DL15]. Um den Aufbau möglichst automatisch zu gestalten, sind alle Experimente, alle Anwendungen und ARA selbst dabei in einem gemeinsamen Metaprojekt organisiert: PARROT[50]. Nach erfolgreicher Konfiguration können dann folgende Experimente ausgeführt werden:

- `meson compile run-sia-runtime`: Erzeugt die Daten für Abbildung 6.3.
- `meson compile analyze_timing-fr-t_queue`$^{HW}$: Erzeugt die Daten für Abschnitt 6.4.2.
- `meson compile analyze_timing-fr-t_task_pre`$^{HW}$: Erzeugt die Daten für Abschnitt 6.4.2.
- `meson compile analyze_timing-fr-t_task_post`$^{HW}$: Erzeugt die Daten für Abschnitt 6.4.2.
- `meson compile analyze_timing-gpslogger`$^{HW}$: Erzeugt die Daten für Abschnitt 6.4.4.
- `meson compile run-multisse-synthetic-tests`: Erzeugt die Daten für Abschnitt 7.8.3.
- `meson compile run-multisse-results`: Erzeugt die Analyse-Daten für Abschnitt 7.8.4 und Abschnitt 7.8.5.
- `meson compile run-multisse-synthesis`$^{HW}$: Erzeugt die Synthese-Daten für Abschnitt 7.8.4 und Abschnitt 7.8.5.
- `meson compile run-app-stats`: Erzeugt die Daten für Abschnitt 8.5.

Experimente, die mit HW gekennzeichnet sind, benötigen die in den jeweiligen Kapiteln genannte gensonderte Hardware. Für die finale Version der Dissertation werde ich eine virtuelle Maschine mit der passenden Software und Anleitung bereitstellen.

[50] `https://scm.sra.uni-hannover.de/research/parrot`. Eine Veröffentlichung auf Github ist geplant.

# Literatur


[Abs24] AbsInt Angewandte Informatik GmbH, aiT Worst-Case Execution Time Analyzers, http://www.absint.de/ait/ (Abgerufen: 2024-08-12).

[AEE03] AEEC, *Avionics Application Software Standard Interface (ARINC Specification 653-1)*, (2003).

[ALS+08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, und Jeffrey D. Ullman, *Compiler: Prinzipien, Techniken und Werkzeuge*, second Auflage (Pearson Studium, München, Deutschland, 2008).

[ANN+20] Benny Akesson *et al.*, An Empirical Survey-based Study into Industry Practice in Real-time Systems, In 2020 IEEE Real-Time Systems Symposium (RTSS) (2020).

[AD92] Rajeev Alur und David Dill, The theory of timed automata, In J. W. de Bakker, C. Huizing, W. P. de Roever, und G. Rozenberg (Herausgeber) Real-Time: Theory in Practice (Springer Berlin Heidelberg, Berlin, Heidelberg, 1992).

[Ama23] Amazon Web Services, Inc, FreeRTOS - A FREE Open Source RTOS. The Free RTOS API functions for creating RTOS tasks and deleting RTOS tasks - xTaskCreate() and vTaskDelete, https://www.freertos.org/a00019.html (Abgerufen: 2023-09-07).

[AGV+22] Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, und Giacomo Travaglini, Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm, In Martina Maggio (Herausgeber) 34th Euromicro Conference on Real-Time Systems (ECRTS 2022) (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022).

[App22] Apple Inc., About iOS 15 Updates, https://support.apple.com/en-ae/HT212788 (Abgerufen: 2022-07-26).

[Atk04] Darren C Atkinson, Accurate call graph extraction of programs with function pointers using type signatures, In 11th Asia-Pacific Software Engineering Conference, IEEE (2004).

[AUT13] AUTOSAR, *Specification of Operating System (Version 5.1.0)*, Technischer Bericht (Automotive Open System Architecture GbR, 2013).

[AUT20] AUTOSAR, *Specification of Operating System (Version R20-11)*, Technischer Bericht (Automotive Open System Architecture GbR, 2020).

[AUT24] AUTOSAR GbR, AUTOSAR History, https://www.autosar.org/about/history/ (Abgerufen: 2024-07-01).

[Deu11] acatech - Deutsche Akademie der Technikwissenschaften, *Cyber-Physical Systems. Driving force for innovation in mobility, health, energy and production*, Technischer Bericht (2011).

[BBL+06] Thomas Ball *et al.*, Thorough static analysis of device drivers, In Yolande Berbers und Willy Zwaenepoel (Herausgeber) Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06) (ACM Press, New York, NY, USA, 2006).

[BR02] Thomas Ball und Sriram K. Rajamani, The SLAM project: debugging system software via static analysis, In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Association for Computing Machinery, New York, NY, USA, 2002).

[BB48] J. Bardeen und W. H. Brattain, The Transistor, A Semi-Conductor Triode, *Phys. Rev.* **74** 230–231 (1948).

[Bar02] 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) (ACM Press, New York, NY, USA, 2002).

[BB19] Leandro Batista Ribeiro und Marcel Baunach, COFIE: a regex-like interaction and control flow description, In 2019 IEEE International Conference on Industrial Cyber Physical Systems (ICPS) (2019).

[BLN+23] Leandro Batista Ribeiro *et al.*, A Modeling Concept for Formal Verification of OS-Based Compositional Software, In Fundamental Approaches to Software Engineering: 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings (Springer-Verlag, Berlin, Heidelberg, 2023).

[BBF+06] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, und Yvon Trinquet, Trampoline: An OpenSource Implementation of the OSEK/VDX RTOS Specification, In IEEE Conference on Emerging Technologies and Factory Automation, 2006. ETFA '06. (IEEE Computer Society Press, Washington, DC, USA, 2006).

[BGC+06] Ramon Bertran *et al.*, Building a Global System View for Optimization Purposes, In Proceedings of the 2nd Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06) (IEEE Computer Society Press, Washington, DC, USA, 2006).

[BHM08] Aske Brekling, Michael R. Hansen, und Jan Madsen, Models and formal verification of multiprocessor system-on-chips, *The Journal of Logic and Algebraic Programming* **77**(1), 1–19 (2008). (The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006))

[BEG+07] Preston Briggs *et al.*, WHOPR - Fast and Scalable Whole Program Optimizations in GCC, Initial Draft, (2007).

[But05] Giorgio C. Buttazzo, Rate Monotonic vs. EDF: Judgment Day, *Real-Time Systems* **29** 5–26 (2005).

[BG06] Giorgio Buttazzo und Paolo Gai, Efficient EDF Implementation for Small Embedded Systems, (2006).

[CS88] David Callahan und Jaspal Sublok, Static analysis of low-level synchronization, *SIGPLAN Not.* **24**(1), 100–111 (1988).

[CDD+05] Dominique Chanet *et al.*, System-Wide Compaction and Specialization of the Linux Kernel, In Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (Association for Computing Machinery, New York, NY, USA, 2005).

[CA11] Jiang Chen und Toshiaki Aoki, Conformance Testing for OSEK/VDX Operating System Using Model Checking, In Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC 2011) (IEEE Computer Society Press, Los Alamitos, CA, USA, 2011).

[CJ21] Nathan Chong und Bart Jacobs, Formally Verifying FreeRTOS' Interprocess Communication Mechanism, In Embedded World Exhibition and Conference (2021).

[CMD62] Fernando J. Corbató, Marjorie Merwin-Daggett, und Robert C. Daley, An Experimental Time-Sharing System, In Proceedings of the May 1-3, 1962, Spring Joint Computer Conference (Association for Computing Machinery, New York, NY, USA, 1962).

[CC77] Patrick Cousot und Radhia Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (ACM, New York, NY, USA, 1977).

[CC14] Patrick Cousot und Radhia Cousot, Abstract interpretation: past, present and future, (2014).

[CCF+05] Patrick Cousot *et al.*, The ASTREÉ Analyzer, In Shmuel Sagiv (Herausgeber) Proceedings 14th European Symposium on Programming (ESOP '05) (Springer-Verlag, Heidelberg, Germany, 2005).

[DAM08] Shuhaizar Daud, R. Badlishah Ahmad, und Nukala S. Murhty, The effects of compiler optimizations on embedded system power consumption, In 2008 International Conference on Electronic Design (2008).

[Dav13] Robert I. Davis, *Burns Standard Notation for Real Time Scheduling*, (2013).

[DGM09] David Déharbe, Stephenson Galvao, und Anamaria Martins Moreira, Formalizing FreeRTOS: First steps, In Brazilian Symposium on Formal Methods, Springer (2009).

[DFK+21] Patrick Denzler *et al.*, Experiences from Adjusting Industrial Software for Worst-Case Execution Time Analysis, In 2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC) (2021).

[Die19] Christian Dietrich, *Interaction-Aware Analysis and Optimization of Real-Time Application and Operating System*, (Dissertation). Leibniz Universität Hannover (2019).

[DHL15] Christian Dietrich, Martin Hoffmann, und Daniel Lohmann, Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems, In Proceedings of the 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15) (ACM Press, New York, NY, USA, 2015).

[DHL17] Christian Dietrich, Martin Hoffmann, und Daniel Lohmann, Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis, *ACM Transactions on Embedded Computing Systems* **16**(2), 35:1–35:25 (2017).

[DL15] Christian Dietrich und Daniel Lohmann, The dataref versuchung, *ACM SIGOPS Operating Systems Review: Special Issue on Repeatability and Sharing of Experimental Artifacts* 51–60 (2015).

[DL17] Christian Dietrich und Daniel Lohmann, OSEK-V: Application-Specific RTOS Instantiation in Hardware, In Proceedings of the 2017 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '17) (ACM Press, New York, NY, USA, 2017).

[DL18] Christian Dietrich und Daniel Lohmann, Semi-Extended Tasks: Efficient Stack Sharing Among Blocking Threads, In Sebastian Altmeyer (Herausgeber) Proceedings of the 39th IEEE Real-Time Systems Symposium 2018 (IEEE Computer Society Press, Nashville, Tennessee, USA, 2018).

[DTS+12] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, und Daniel Lohmann, A Robust Approach for Variability Extraction from the Linux Build System, In Eduardo Santana de Almeida, Christa Schwanninger, und David Benavides (Herausgeber) Proceedings of the 16th Software Product Line Conference (SPLC '12) (ACM Press, New York, NY, USA, 2012).

[DFW+21] Timothee Durand, Katalin Fazekas, Georg Weissenbacher, und Jakob Zwirchmayr, Model Checking AUTOSAR Components with CBMC, In 2021 Formal Methods in Computer Aided Design (FMCAD), IEEE (2021).

[DAD14] D. Hutchins, A. Ballman, und D. Sutherland, C/C++ Thread Safety Analysis, In 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (2014).

[EA19] EETimes und Aspencore, 2019 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments,.

[EA03] Dawson Engler und Ken Ashcraft, RacerX: effective, static detection of race conditions and deadlocks, In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03) (ACM Press, New York, NY, USA, 2003).

[EFL23] Gerion Entrup, Björn Fiedler, und Daniel Lohmann, MultiSSE: Static Syscall Elision and Specialization for Event-Triggered Multi-Core RTOS, In Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'23) (2023).

[EKF+24] Gerion Entrup, Andreas Kässens, Björn Fiedler, und Daniel Lohmann, Applied static analysis and specialization of cross-core syscalls for multi-core AUTOSAR OS, *Real-Time Systems* (2024).

[ENL22] Gerion Entrup, Jan Neugebauer, und Daniel Lohmann, RTOS-Independent Interaction Analysis in ARA, In Proceedings of the 16th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '22) (2022).

[ESD19] Gerion Entrup, Benedikt Steinmeier, und Christian Dietrich, ARA: Automatic Instance-Level Analysis in Real-Time Systems, In Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '19) (2019).

[EGL11] Andreas Ermedahl, Jan Gustafsson, und Björn Lisper, Deriving WCET Bounds by Abstract Execution, In ECRTS 2011 (2011).

[FKD+12] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, und Hitoshi Ohsaki, Formal model-based test for AUTOSAR multicore RTOS, In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE (2012).

[FYD+19] Zhang Feng, Zhao Yongwang, Ma Dianfu, und Niu Wensheng, Fine-Grained Formal Specification and Analysis of Buddy Memory Allocation in Zephyr RTOS, In 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC) (2019).

[Fie23] Björn Fiedler, *Anwendungsgewahre statische Spezialisierung vormals dynamischer Systemaufrufe zur Verbesserung nichtfunktionaler Eigenschaften eingebetteter Echtzeitsysteme*, (Dissertation). Leibniz Universität Hannover (2023).

[FED+18] Björn Fiedler, Gerion Entrup, Christian Dietrich, und Daniel Lohmann, Levels of Specialization in Real-Time Operating Systems, In Proceedings of the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '18) (2018).

[FED+21] Björn Fiedler, Gerion Entrup, Christian Dietrich, und Daniel Lohmann, ARA: Static Initialization of Dynamically-Created System Objects, In Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21) (2021).

[Flo19] Jens Flottau, "Ungeheuerliche" Software im Boeing-Flugzeug, https://www.sueddeutsche.de/wirtschaft/boeing-737-max-absturz-ursache-untersuchung-1.4647909 (Abgerufen: 2019-10-20).

[FKU+16] Florian Franzmann *et al.*, From Intent to Effect: Tool-Based Generation of Time-Triggered Real-Time Systems on Multi-Core Processors, In Proceedings of the 19th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '16) (IEEE Computer Society Press, Washington, DC, USA, 2016).

[Fre23] Free Software Foundation, Inc., 1.2.2 POSIX (The GNU C Library), https://www.gnu.org/software/libc/manual/html_node/POSIX.html (Abgerufen: 2023-02-09).

[FRR+17] F. Abdi, R. Mancuso, R. Tabish, und M. Caccamo, Restart-based fault-tolerance: System design and schedulability analysis, In 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2017).

[FRS+16] F. Abdi *et al.*, Reset-based recovery for real-time cyber-physical systems with temporal safety constraints, In 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA) (2016).

[Gar22] Garmin Ltd., Increased Battery Drain After Updating Software, https://support.garmin.com/en-US/?faq=WyBTzX43H58sDwnGBoK5m7 (Abgerufen: 2022-07-26).

[GMV08] John Giacomoni, Tipp Moseley, und Manish Vachharajani, FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue, In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Association for Computing Machinery, New York, NY, USA, 2008).

[Git22] Github, Inc., Contributors to FreeRTOS/FreeRTOS-Kernel · GitHub, https://github.com/FreeRTOS/FreeRTOS-Kernel/graphs/contributors (Abgerufen: 2022-08-05).

[Gre22] Chris Gregg, *Lecture notes of Computer Systems 107, Lecture 16: Optimization*, https://web.stanford.edu/class/archive/cs/cs107/cs107.1224/lectures/16/16-Optimization.pdf (2022).

[Gro22] IMARC Group, Embedded Computer Market: Global Industry Trends, Share, Size, Growth, Opportunity and Forecast 2023-2028,.

[HCK+03] Per Hammarlund, James B. Crossland, Shivnandan D. Kaushik, und Anil Aggarwal, *Inter-processor interrupts*, *Patent nr. US Patent 8,984,199 B2.* (2003).

[HBR21] Imane Haur, Jean-Luc Béchennec, und Olivier Henri Roux, Formal Schedulability Analysis Based on Multi-Core RTOS Model, In 29th International Conference on Real-Time Networks and Systems (Association for Computing Machinery, New York, NY, USA, 2021).

[HBR22] Imane Haur, Jean-Luc Béchennec, und Olivier H. Roux, Formal Verification of the Inter-core Synchronization of a Multi-core RTOS Kernel, In Adrian Riesco und Min Zhang (Herausgeber) Formal Methods and Software Engineering (Springer International Publishing, Berlin, Heidelberg, 2022).

[Hea24] Headway Software Technologies Ltd., Structure101 – Software Architecture Development Environment (ADE), https://structure101.com/ (Abgerufen: 2024-08-08).

[HP19] John L. Hennessy und David A. Patterson, *Computer Architecture: A Quantitative Approach*, 6 Auflage (Morgan Kaufmann Publishers Inc., 2019).

[HA12] Alexander Herz und Kalmer Apinis, Class-Modular, Class-Escape and Points-to Analysis for Object-Oriented Languages, In Alwyn E. Goodloe und Suzette Person (Herausgeber) NASA Formal Methods (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012).

[Hof14] Wanja Hofer, *Sloth: The Virtue and Vice of Latency Hiding in Hardware-Centric Operating Systems*, (Dissertation). Friedrich-Alexander-Universität Erlangen-Nürnberg (2014).

[HDM+12] Wanja Hofer *et al.*, Sloth on Time: Efficient Hardware-Based Scheduling for Time-Triggered RTOS, In Proceedings of the 33rd IEEE International Symposium on Real-Time Systems (RTSS '12) (IEEE Computer Society Press, 2012).

[HLS+09] Wanja Hofer, Daniel Lohmann, Fabian Scheler, und Wolfgang Schröder-Preikschat, Sloth: Threads as Interrupts, In Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09) (IEEE Computer Society Press, 2009).

[HBD+14] 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) (IEEE Computer Society Press, 2014).

[HLD+15] Martin Hoffmann, Florian Lukas, Christian Dietrich, und Daniel Lohmann, DOSEK: Maßgeschneiderte Zuverlässigkeit, In Wolfgang Halang und Olaf Spinczyk (Herausgeber) Betriebssysteme und Echtzeit (Springer, Berlin-Heidelberg, 2015).

[HLD+15] Martin Hoffmann, Florian Lukas, Christian Dietrich, und Daniel Lohmann, DOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel, In Proceedings of the 21st IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '15) (IEEE Computer Society Press, Washington, DC, USA, 2015).

[HBL10] Mike Holenderski, Reinder J. Bril, und Johan J. Lukkien, Grasp: Tracing, visualizing and measuring the behavior of real-time systems, In In Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (2010).

[Hol08] Niklas Holsti, Computing time as a program variable: a way around infeasible paths, In Raimund Kirner (Herausgeber) 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08) (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2008). (also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3)

[Hol06] Gerard J. Holzmann, The Power of 10: Rules for Developing Safety-Critical Code, *Computer* **39**(6), 95-97 (2006).

[HHF22] Min-Yih Hsu, Felicitas Hetzelt, und Michael Franz, DFI: An Interprocedural Value-Flow Analysis Framework that Scales to Large Codebases, *ArXiv* **abs/2209.02638** (2022).

[HZZ+11] Yanhong Huang *et al.*, Modeling and Verifying the Code-Level OSEK/VDX Operating System with CSP, In Proceedings of the 5th International Symposium on Theoretical Aspects of Software Engineering (TASE'11) (IEEE Computer Society Press, Washington, DC, USA, 2011).

[Inc23] Texas Instruments Incorporated, Arm-based processors product selection | TI.com, https://www.ti.com/microcontrollers-mcus-processors/arm-based-processors/products.html (Abgerufen: 2023-02-09).

[ISO18] ISO, *Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*, Technischer Bericht (ISO, 2018).

[Isa90] James Isaak, Standards-the history of Posix: a study in the standards process, *Computer* **23**(7), 89-92 (1990).

[JPP94] Richard Johnson, David Pearson, und Keshav Pingali, The program structure tree: computing control regions in linear time, In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94) (ACM Press, New York, NY, USA, 1994).

[KMS+17] Daniel Kaestner *et al.*, Finding All Potential Run-Time Errors and Data Races in Automotive Software (SAE International, Detroit, USA, 2017).

[Kha79] L. G. Khachiyan, *A polynomial algorithm in linear programming*, Band 244, S.. 1093–1096 (1979).

[KP05] Raimund Kirner und Peter Puschner, Classification of WCET Analysis Techniques, In Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05) (IEEE Computer Society Press, Washington, DC, USA, 2005).

[KKZ12] Jens Knoop, Laura Kovács, und Jakob Zwirchmayr, Symbolic Loop Bound Computation for WCET Analysis, In Edmund Clarke, Irina Virbitskaite, und Andrei Voronkov (Herausgeber) Perspectives of Systems Informatics (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012).

[Kop11] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Second Edition Auflage (Springer-Verlag, 2011).

[KPS+16] Daniel Kroening, Daniel Poetzl, Peter Schrammel, und Björn Wachter, Sound Static Deadlock Analysis for C/Pthreads, In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (Association for Computing Machinery, New York, NY, USA, 2016).

[Küh17] Dr.-Ing. Claus Kühnel, Echtzeit mit dem Raspberry Pi und Linux PREEMPT_RT, https://www.embedded-software-engineering.de/echtzeit-mit-dem-raspberry-pi-und-linux-preemptrt-a-e8c1c4026307da04cb2224fe64294a22/ (Abgerufen: 2024-06-17).

[LBR11] Karthik Lakshmanan, Gaurav Bhatia, und Ragunathan Rajkumar, AUTOSAR extensions for predictable task synchronization in multi-core ECUs, In Proceedings of the SAE 2011 World Congress (2011).

[LBL+11] Patrick Lam, Eric Bodden, Ondvrej Lhoták, und Laurie Hendren, The Soot framework for Java program analysis: a retrospective (2011).

[Lam94] Leslie Lamport, The Temporal Logic of Actions, *ACM Trans. Program. Lang. Syst.* **16**(3), 872–923 (1994).

[LPY97] Kim G. Larsen, Paul Pettersson, und Wang Yi, Uppaal in a nutshell, *International Journal on Software Tools for Technology Transfer (STTT)* **1**(1–2), 134–152 (1997).

[LA04] Chris Lattner und Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04) (IEEE Computer Society Press, Washington, DC, USA, 2004).

[LGC+13] Nhat Minh Lê, Adrien Guatto, Albert Cohen, und Antoniu Pop, Correct and Efficient Bounded FIFO Queues, In 2013 25th International Symposium on Computer Architecture and High Performance Computing (2013).

[LLH+04] C.T. Lee, J.M. Lin, Z.W. Hong, und W.T. Lee, An Application-Oriented Linux Kernel Customization for Embedded Systems, *Journal of information science and engineering* **20**(6), 1093–1108 (2004).

[LMN06] Luis E. Leyva-del-Foyo, Pedro Mejia-Alvarez, und Dionisio de Niz, Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware, In Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06) (IEEE Computer Society Press, Los Alamitos, CA, USA, 2006).

[Liu00] Jane W. S. Liu, *Real-Time Systems* (Prentice Hall PTR, Englewood Cliffs, NJ, USA, 2000).

[Loc21] Alexander Lochmann, *Aufzeichnungsbasierte Analyse von Sperren in Betriebssystemen*, (Dissertation). Technische Universität Dortmund (2021).

[LSB+19] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, und Olaf Spinczyk, LockDoc: Trace-Based Analysis of Locking in the Linux Kernel, In Proceedings of the 14th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '19) (ACM Press, New York, NY, USA, 2019).

[Loh14] Daniel Lohmann, *Tailorable System Software*, (Habilitationsschrift). Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät (2014).

[MDD+24] Emad Jacob Maroun *et al.*, The Platin Multi-Target Worst-Case Analysis Tool, In Proceedings of the 22nd International Workshop on Worst-Case Execution Time Analysis (WCET '24) (2024). (Accepted)

[Mar07] Peter Marwedel, *Eingebettete Systeme* (Springer-Verlag, Heidelberg, Germany, 2007).

[Mar21] Peter Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things* (Springer Nature, Heidelberg, Germany, 2021).

[MS18] Anders Møller und Michael I. Schwartzbach, *Static Program Analysis*, https://cs.au.dk/~amoeller/spa/ (2018). (Department of Computer Science, Aarhus University)

[MIR04] MIRA Limited, *Guidelines for the Use of the C Language in Critical Systems (MISRA-C:2004)* (2004).

[MS96] Maged M. Michael und Michael L. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, In Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (Association for Computing Machinery, New York, NY, USA, 1996).

[Min12] Antoine Miné, Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs, *Logical Methods in Computer Science* **8**(1), (2012).

[Min15] Antoine Miné, AstréeA: A static analyzer for large embedded multi-task software, In Proc. of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15) (Springer, 2015).

[Min17] Antoine Miné, Static Analysis of Embedded Real-Time Concurrent Software with Dynamic Priorities, *Electronic Notes in Theoretical Computer Science* **331** 3–39 (2017). (Proceedings of the Sixth Workshop on Numerical and Symbolic Abstract Domains (NSAD 2016))

[MB16] Robert Mittermayr und Johann Blieberger, A Generic Graph Model for WCET Analysis of Multi-Core Concurrent Applications, *Journal of Software Engineering and Applications* **9** 182-198 (2016).

[MB21] Robert Mittermayr und Johann Blieberger, Deadlock and WCET analysis of barrier-synchronized concurrent programs, *Computing* **103** 749-770 (2021).

[MKL24] MKLabs Co.,Ltd, StarUML Homepage, http://staruml.io/ (Abgerufen: 2024-08-08).

[NPS+09] Mayur Naik, Chang-Seo Park, Koushik Sen, und David Gay, Effective Static Deadlock Detection, In Proceedings of the 31st International Conference on Software Engineering (IEEE Computer Society, Washington, DC, USA, 2009).

[Nob23] Nobel Prize Outreach, *The Nobel Prize in Physics 1956*, https://www.nobelprize.org/prizes/physics/1956/summary/ (2023).

[OSE05] OSEK/VDX Group, *Operating System Specification 2.2.3*, Technischer Bericht (OSEK/VDX Group, 2005).

[Tra18] U.S. Department of Transportation – National Highway Traffic Safety Administration, *Laboratory Test Procedure for FMVSS111 – Rear Visibility (other than schoolbusses)*, (2018).

[Pag24] Bruno Pagès, BOUML - a free UML tool box, https://bouml.fr/ (Abgerufen: 2024-08-08).

[PH21] David A. Patterson und John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 6 Auflage (Morgan Kaufmann Publishers Inc., 2021).

[Pei14] Tiago P. Peixoto, The graph-tool python library, *figshare* (2014).

[Per24] Percepio AB, Percepio Tracealyzer - Percepio, https://percepio.com/tracealyzer/ (Abgerufen: 2024-08-08).

[PK07] Hendrik Post und Wolfgang Küchlin, Integrated static analysis for Linux device driver verification, In Proceedings of the 6th International Conference on Integrated Formal Methods (Springer-Verlag, Berlin, Heidelberg, 2007).

[Pro22] Zephyr Project, Zephyr – An Operating System for IoT - Zephyr Project, https://www.zephyrproject.org/zephyr-an-operating-system-for-iot/ (Abgerufen: 2022-08-05).

[Pro23] Zephyr Project, Symmetric Multiprocessing — Zephyr Project Documentation, https://docs.zephyrproject.org/latest/kernel/services/smp/smp.html (Abgerufen: 2024-01-23).

[Pro23] Zephyr Project, Zephyr API Documentation: Thread APIs, https://docs.zephyrproject.org/latest/doxygen/html/group__thread__apis.html (Abgerufen: 2023-09-07).

[Pro59] Reese T. Prosser, Applications of Boolean Matrices to the Analysis of Flow Diagrams, In Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference (ACM Press, New York, NY, USA, 1959).

[RUS19] Phillip Raffeck, Peter Ulbrich, und Wolfgang Schröder-Preikschat, Work-in-Progress: Migration Hints in Real-Time Operating Systems, In 2019 IEEE Real-Time Systems Symposium (RTSS) (2019).

[RDH+05] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, und Richard D. Schlichting, Automatic Operating System Specialization via Binary Rewriting (2005).

[Raj90] Ragunathan Rajkumar, Real-time synchronization protocols for shared memory multiprocessors, In Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS '90) (IEEE Computer Society Press, Washington, DC, USA, 1990).

[RHS95] Thomas Reps, Susan Horwitz, und Mooly Sagiv, Precise interprocedural dataflow analysis via graph reachability, In Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Association for Computing Machinery, New York, NY, USA, 1995).

[Ric20] Richard Barry, Real-Time Edge Processing: Get to Market Faster with Future-Proofed Designs, https://www.allaboutcircuits.com/tech-days/summer-2020/amazon-web-services/webinars/richard-barry-real-time-edge-processing-get-to-market-faster-future-proofed-designs/ (Abgerufen: 2022-08-05).

[Rin01] Martin Rinard, Analysis of Multithreaded Programs, In Patrick Cousot (Herausgeber) Static Analysis (Springer Berlin Heidelberg, Berlin, Heidelberg, 2001).

[RY20] Xavier Rival und Kwangkeun Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective* (MIT Press, 2020).

[Rog69] Jr. Hartley Rogers, *Theory of recursive functions and effective computability* (McGraw-Hill, Inc., New York NY, USA, 1969).

[RHL14] Andreas Ruprecht, Bernhard Heinloth, und Daniel Lohmann, Automatic Feature Selection in Large-Scale System-Software Product Lines, In Matthew Flatt (Herausgeber) Proceedings of the 13th International Conference on Generative Programming and Component Engineering (GPCE '14) (ACM Press, New York, NY, USA, 2014).

[RSK03] Carsten Rust, Friedhelm Stappert, und R. Künnemeyer, From Timed Petri Nets to Interrupt-Driven Embedded Control Software., In International Conference on Computer, Communication and Control Technologies (CCCT 2003) (Orlando, Florida, USA, 2003).

[SYY+15] David Sanán *et al.*, Verifying FreeRTOS' Cyclic Doubly Linked List Implementation: From Abstract Specification to Machine Code, In 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE (2015).

[SK80] Randolph G. Scarborough und Harwood G. Kolsky, Improved Optimization of FORTRAN Object Programs, *IBM Journal of Research and Development* **24**(6), 660-676 (1980).

[Sch11] Fabian Scheler, *Atomic Basic Blocks: Eine Abstraktion für die gezielte Manipulation der Echtzeitsystemarchitektur*, (Dissertation). Friedrich-Alexander-Universität Erlangen-Nürnberg, Technische Fakultät (2011).

[SS10] Fabian Scheler und Wolfgang Schröder-Preikschat, The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems, In Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10) (IEEE Computer Society Press, Washington, DC, USA, 2010).

[SBS+10] Horst Schirmeier, Matthias Bahne, Jochen Streicher, und Olaf Spinczyk, Towards eCos Autoconfiguration by Static Application Analysis, In Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA '10) (CEUR-WS.org, Antwerp, Belgium, 2010).

[Sch19] Wolfgang Schröder-Preikschat, *Systemprogrammierung – Grundlage von Betriebssystemen: Sachwortverzeichnis*, https://www4.cs.fau.de/DE/ wosch/glossar.pdf (2019).

[SHB19] Philipp Dominik Schubert, Ben Hermann, und Eric Bodden, PhASAR: An Interprocedural Static Analysis Framework for C/C++, In Tomáš Vojnar und Lijun Zhang (Herausgeber) Tools and Algorithms for the Construction and Analysis of Systems (Springer International Publishing, Cham, 2019).

[SWU+19] Simon Schuster, Peter Wägemann, Peter Ulbrich, und Wolfgang Schröder-Preikschat, Proving Real-Time Capability of Generic Operating Systems by System-Aware Timing Analysis, In 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2019).

[SWU+21] Simon Schuster, Peter Wägemann, Peter Ulbrich, und Wolfgang Schröder-Preikschat, Annotate once – analyze anywhere: context-aware WCET analysis by user-defined abstractions, In Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (Association for Computing Machinery, New York, NY, USA, 2021).

[SSV+11] Martin D. Schwarz *et al.*, Static Analysis of Interrupt-Driven Programs Synchronized via the Priority Ceiling Protocol, In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Association for Computing Machinery, New York, NY, USA, 2011).

[SSS+21] Michael Schwarz *et al.*, Improving Thread-Modular Abstract Interpretation, In Cezara Drăgoi, Suvam Mukherjee, und Kedar Namjoshi (Herausgeber) Static Analysis (Springer International Publishing, Chicago, IL, USA, 2021).

[SVM03] Helmut Seidl, Varmo Vene, und Markus Müller-Olm, Global invariants for analyzing multithreaded applications, *Proc. of the Estonian Academy of Sciences: Phys., Math.* **52**(4), 413–436 (2003).

[SWH10] Helmut Seidl, Reinhard Wilhelm, und Sebastian Hack, *Übersetzerbau. Band 3. Analyse und Transformation* (Springer, 2010).

[Sha80] M. Sharir, Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers, *Comput. Lang.* **5**(3–4), 141–153 (1980).

[SR94] K.G. Shin und P. Ramanathan, Real-time computing: a new discipline of computer science and engineering, *Proceedings of the IEEE* **82**(1), 6-24 (1994).

[SGG05] Abraham Silberschatz, Greg Gagne, und Peter Bear Galvin, *Operating System Concepts*, Seventh Auflage (John Wiley & Sons, Inc., 2005).

[Ste77] Guy Lewis Steele, Debunking the "expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO, In Proceedings of the 1977 Annual Conference (Association for Computing Machinery, New York, NY, USA, 1977).

[SKR90] Bernhard Steffen, Jens Knoop, und Oliver Rüthing, The Value Flow Graph: A Program Representation for Optimal Program Transformations, In Proceedings of the 3rd European Symposium on Programming (Springer-Verlag, Berlin, Heidelberg, 1990).

[Str13] Bjarne Stroustrup, *The C++ programming language*, 4 Auflage (Addison-Wesley Educational, Boston, MA, 2013).

[Str24] Structurizr cloud service, Structurizr, https://structurizr.com (Abgerufen: 2024-08-08).

[SX16] Yulei Sui und Jingling Xue, SVF: Interprocedural Static Value-Flow Analysis in LLVM, In Proceedings of the 25th International Conference on Compiler Construction (Association for Computing Machinery, New York, NY, USA, 2016).

[Tar55] Alfred Tarski, A lattice-theoretical fixpoint theorem and its applications., *Pacific Journal of Mathematics* **5**(2), 285 – 309 (1955).

[TKH+12] Reinhard Tartler *et al.*, Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability, In Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep '12) (USENIX Association, Berkeley, CA, USA, 2012).

[Tea24] Rust Resources Team, *The Embedded Rust Book* (online, 2024). (version, commit 3f9df2b)

[The23] The Linux man-pages project, syscalls(2), https://man7.org/linux/man-pages/man2/syscalls.2.html (Abgerufen: 2023-02-09).

[TOP23] TOP500.org, HPCG List – June 2023 | TOP500 , https://www.top500.org/lists/hpcg/list/2023/06/ (Abgerufen: 2023-09-11).

[TP24] Linus Tolke und Vincenzo Palazzo, ArgoUML | ArgoUML resources and web pages., https://argouml-tigris-org.github.io/tigris/argouml/ (Abgerufen: 2024-08-08).

[Tra19] Gregory Travis, How the Boeing 737 Max Disaster Looks to a Software Developer, https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer (Abgerufen: 2019-04-18).

[UKH+11] Peter Ulbrich *et al.*, I4Copter: An Adaptable and Modular Quadrotor Platform, In Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11) (ACM Press, New York, NY, USA, 2011).

[Val23] Valve Corporation, Steam Hardware & Software Survey, https://store.steampowered.com/hwsurvey/cpus/ (Abgerufen: 2023-09-11).

[VVK08] Jussi Vanhatalo, Hagen Völzer, und Jana Koehler, The Refined Process Structure Tree, In Marlon Dumas, Manfred Reichert, und Ming-Chien Shan (Herausgeber) Business Process Management (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008).

[Var22] Various, *POSIX® 1003.1 Frequently Asked Questions (FAQ Version 1.18)*, https://www.opengroup.org/austin/papers/posix_faq.html (2022).

[VV09] Vesal Vojdani und Varmo Vene, Goblint: Path-sensitive data race analysis, *Annales Univ. Sci. Budapest., Sect. Comp.* **30** 141–155 (2009).

[VCY+16] Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, und Toshiaki Aoki, Verifying OSEK/VDX OS Design Using Its Formal Specification, In Proc. TASE'16 (IEEE Computer Society, 2016).

[Hee24] Dimitri van Heesch, Doxygen homepage, http://www.doxygen.nl/ (Abgerufen: 2024-08-08).

[WDD+18] Peter Wägemann *et al.*, Whole-System Worst-Case Energy-Consumption Analysis for Energy-Constrained Real-Time Systems, In Sebastian Altmeyer (Herausgeber) Proceedings of the 30th Euromicro Conference on Real-Time Systems 2018 (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018).

[Web97] Helmut Weber, *Praktische Systemprogrammierung: Grundlagen und Realisierung unter UNIX und verwandten Systemen* (Springer-Verlag, 1997).

[Weg17] Simon Wegener, Towards Multicore WCET Analysis, In Jan Reineke (Herausgeber) 17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017) (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017).

[WB13] Alexander Wieder und Björn B. Brandenburg, On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks, In Proceedings of the 34th IEEE International Symposium on Real-Time Systems (RTSS '13) (IEEE Computer Society Press, USA, 2013).

[ZEN24] ZEN PROGRAM, NDepend – Homepage, https://www.ndepend.com/ (Abgerufen: 2024-08-08).

[ZAC15] Haitao Zhang, Toshiaki Aoki, und Yuki Chiba, Yes! You Can Use Your Model Checker to Verify OSEK/VDX Applications, In 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015 (2015).

[ZCO14] Min Zhang, Yunja Choi, und Kazuhiro Ogata, A Formal Semantics of the OSEK/VDX Standard in K Framework and Its Applications, In Proc. WRLA'14 (Springer, 2014).

-