FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Felix Kurt-Michael Bräunling

Variablenklassifizierung durch statische Analyse in einer AUTOSAR Umgebung

Masterarbeit im Fach Mechatronik

28. September 2018

Please cite as:
Felix Kurt-Michael Bräunling, “Variablenklassifizierung durch statische Analyse in einer AUTOSAR Umgebung” Master’s Thesis, University of Erlangen, Dept. of Computer Science, September 2018.

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

Variablenklassifizierung durch statische Analyse in einer AUTOSAR Umgebung

Masterarbeit im Fach Mechatronik

vorgelegt von

Felix Kurt-Michael Bräunling

geb. am 30. Juni 1994
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: **Dr.-Ing. Peter Ulbrich**
Dr.-Ing. Isabella Stilkerich
Dipl.-Ing. Tobias Klaus
Dipl.-Ing. Nikolay Stefanov

Betreuender Hochschullehrer: **Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**

Beginn der Arbeit: **1. April 2018**
Abgabe der Arbeit: **30. September 2018**

Erklärung

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

Declaration

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

(Felix Kurt-Michael Bräunling)
Erlangen, 28. September 2018

ABSTRACT

Modern embedded system hardware architectures offer different on-system memories to store program data. At the same time software in automotive control systems is getting more complex. It still has to be reusable but has to run efficiently on the targeted hardware. For safety-critical systems aspects such as real-time capability and freedom of interference have to be considered as well. These aspects need to be reflected by the program data binding. This requires a systematic approach, which preferable is automated, to be done efficiently for complex systems. In this thesis, the Magic Memory Handling Tool (MMHT) is developed, which uses static analysis and the system specification to classify program data. This classification is then used to specify memory sections that represent a binding of program data to memories.

To achieve this, a short overview of sources of interference and scheduling theory is given. Different hardware and software based approaches to spatial isolation are presented. A concept for a system design process covering the architecture and implementation phase is presented, in which the developed MMHT will be used. Properties to classify program data are described and a set of classes is derived from these. Methods to influence generic and model-based developed code are shown and an overview of the memory protection mechanism in AUTOSAR OS is given. Based on this a tool is developed that uses the result of sound semantic analysis to classify program data according to the access behavior of a task set in an application and the memory layout of the targeted hardware. For a qualitative evaluation the developed method is applied to an example application.

The results show, that the tool can be used generate a binding of program data to memory that considers run-time efficiency and real-time capability. The sections generated can be used to apply hardware-based memory protection to achieve freedom of interference on a control flow level. By the use of sound semantic analysis freedom of interference is also achieved on a program data level. The developed tool is capable of providing a systematic and automated approach to bind program data to memory. Hereby, it can support the development process of embedded, safety critical systems.

KURZFASSUNG

Moderne eingebettete Systeme ermöglichen es, Anwendungsdaten auf verschiedene Hardwarespeichern zu binden. Gleichzeitig wird die Software für Regelungssysteme in Automobilen immer komplexer. Trotzdem muss sie einen hohen Grad an Wiederverwertbarkeit erfüllen, jedoch effizient auf der verwendeten Hardware ausgeführt werden. Bei sicherheitskritischen Systemen muss auf Echtzeitfähigkeit und Rückwirkungsfreiheit geachtet werden. All diese Aspekte müssen bei der Bindung von Daten auf Speicher berücksichtigt werden. Dies muss systematisch und vorzugsweise automatisiert für komplexe Systeme erfolgen. In dieser Arbeit wird ein Werkzeug vorgestellt, das auf Basis statischer Analyse und der Systemspezifikation eine Klassifizierung der Anwendungsdaten vornimmt. Auf Basis der Klassifikation werden Sektionen vorgesehen, die eine Bindung der Daten auf die Speicher darstellen.

Hierzu werden Interferenzquellen und Ablaufplanung kurz beschrieben. Verschiedene hardware- und softwarebasierte Ansätze für räumliche Rückwirkungsfreiheit werden präsentiert. Es wird ein Konzept vorgestellt, das die Architektur- und Implementierungsebene berücksichtigt und in dem das in dieser Arbeit entwickelte MMHT eingebunden wird. Die Eigenschaften, auf deren Basis Daten klassifiziert werden können, werden vorgestellt und Datenklassen von ihnen abgeleitet. Es werden Methoden zur Beeinflussung der Code-Generierung vorgestellt, der in einem modell- und plattformbasierten Entwicklungsprozess entsteht. Ebenso werden die Hardwarespeicherschutzmechanismen von AUTOSAR OS beschrieben. Aufbauend darauf wird ein Werkzeug entwickelt, das die Ergebnisse einer vollständigen und korrekten semantischen Analyse nutzt, um Anwendungsdaten auf Basis des Kontrollflusszugriffsverhaltens und die Speicherarchitektur der Zielhardware zu klassifizieren. Zur qualitativen Evaluierung des Vorgehens wird es an einer Beispielanwendung durchgeführt.

Aus den Ergebnissen zeigt sich, dass das Werkzeug geeignet ist, eine Bindung zu erzeugen, die sowohl Laufzeiteffizienz als auch Echtzeitfähigkeit berücksichtigt. Die erzeugten Sektionen können verwendet werden, um durch hardwarebasierten Speicherschutz Rückwirkungsfreiheit auf Kontrollflussebene herzustellen. Durch den Einsatz statischer Analyse kann Rückwirkungsfreiheit auf Datumsebene sichergestellt werden. Das entwickelte Werkzeug bietet einen systematischen und automatisierten Ansatz, um Anwendungsdaten auf Speicher zu binden. Hierdurch kann es den Entwicklungsprozess von eingebetteten, sicherheitskritischen Systemen unterstützen.

INHALTSVERZEICHNIS

1 EINLEITUNG

Automobilhersteller (Original Equipment Manufacturer) (OEM) und ihre Kunden fordern immer mehr Funktionalität von Fahrzeugen [Mö10]. In Folge dieser Forderung wird immer mehr Funktionalität in Software, statt in Hardware, umgesetzt [Pre+07], was gleichzeitig in einer Steigerung der Komplexität dieser resultiert [Gri03]. Je nach Einsatzfeld werden allerdings unterschiedliche Anforderungen an die Entwicklung und die funktionale Sicherheit der eingesetzten Software gestellt. Multi-Media Anwendungen und Komfortfunktionen stellen nur weiche Echtzeitanforderungen und sind begrenzt sicherheitskritisch. Auf der anderen Seite finden sich in der Antriebs- und Fahrwerksregelung Anwendungen mit harten Echtzeitanforderungen und starkem Einfluss auf die funktionale Sicherheit des Fahrzeugs [Pre+07]. In dieser Arbeit werden nur Systeme betrachtet, die eine Regelungsfunktion erfüllen und einen starken Einfluss auf die funktionale Sicherheit haben. Unter einem System wird eine Anwendung verstanden, die mindestens einen Sensor, Regler und Aktuator verknüpft [Isoa].

Durch die steigende Softwarekomplexität nimmt auch der Bedarf an Rechenleistung zu, weshalb für die Entwicklung von Steuergeräten vermehrt Mehrkernprozessoren in der Automobilindustrie zum Einsatz kommen. Deren Architekturen stellen neue Herausforderungen an den Softwareentwicklungsprozess sicherheitskritischer, eingebetteter Systeme. Zur Bewältigung dieser Herausforderungen wird im Rahmen des ARAMIS II Projekts an Prozessen und Entwicklungswerkzeugen geforscht, die den Entwicklungsprozess auf Mehrkernplattformen in der Automobil-, Luftfahrt- und Automatisierungsindustrie unterstützen sollen [Kar18].

1.1 Problembeschreibung

In der Entwicklung von sicherheitskritischen, eingebetteten Systemen ergeben sich Probleme aus den Anforderungen der Software für solche Systeme, aus den Gegebenheiten der Zielhardware und den angewendeten Entwicklungsmethoden, die im Entwicklungsprozess adressiert werden müssen. Im Folgenden werden diese Probleme dargestellt.

1.1.1 Softwareanforderungen an Fahrzeugsteuergeräte

Die Anforderungen an Steuergeräte im Automobilbereich sind ähnlich wie bei sicherheitskritischen, eingebetteten Systemen aus anderen Industriebranchen:

- Kosten- und Bauraumeffizienz,
- Echtzeitfähigkeit und

- funktionale Sicherheit [Mö10].

Da solche Systeme möglichst kostengünstig entwickelt und hergestellt werden sollen, muss die Software entsprechend mit limitierten Ressourcen wie Rechenkapazität und Speicher lauffähig gestaltet werden. Aus diesem Grund ist eine ressourcensparende Bindung der einzelnen Softwareeinheiten auf dem Steuergerät eine wichtige Aufgabe des Systementwurfs. Unter Bindung versteht man die Zuordnung von Softwareeinheiten und ihrer Daten auf Hardwareressourcen [TH07]. Softwareeinheiten stellen atomare Softwarekomponenten dar, die alleinstehend getestet werden können [Isoa].

Dies wird auch unter dem Aspekt der Echtzeitfähigkeit deutlich. In Fahrzeugsteuergeräten werden statische Echtzeitsysteme eingesetzt, bei denen Softwareeinheiten vor der Laufzeit auf Hardwareressourcen gebunden werden. Der Vorteil statischer Echtzeitsysteme ist, dass die Einhaltung der zeitlichen Anforderungen im Vergleich zu dynamischen Systemen einfacher bewiesen werden kann [Liu00]. Die hierzu eingesetzten Laufzeitanalysen profitieren stark von vorhersagbaren Ausführungszeiten, welche durch eine gute Bindung von Softwareeinheiten erreicht werden können [Cul+10]. Obwohl Echtzeitfähigkeit nicht mit einer möglichst schnellen Verarbeitung der Softwareaufgaben gleichzusetzen ist, sondern eine rechtzeitige Verarbeitung fordert [Liu00], wird das Einhalten dieser Forderung durch eine gute Bindung begünstigt. Eine solche erhöht die Laufzeiteffizienz, wodurch der verfügbare Schlupf vergrößert und somit die Planbarkeit verbessert wird. Ein Beispiel für eine gute Bindung ist die Platzierung von Softwareeinheiten und ihrer Daten auf möglichst kernnahen Speichern, wodurch der Einfluss von Bus- und Speicherzugriffszeiten auf die Laufzeitanalyse reduziert wird. Gleichzeitig kann die schnellere Zugriffszeit auf kernnahe Speicher die Laufzeiteffizienz erhöhen.

Im Automobilbereich definiert der ISO Standard 26262 die Anforderungen und Richtlinien, um funktionale Sicherheit in einem System zu gewährleisten. Auf Basis einer Gefahren- und Risikobewertung wird ein funktionales Sicherheitskonzept entworfen [Isob], auf Grundlage dessen ein technisches Sicherheitskonzept erstellt wird. Als Folge dieses technischen Sicherheitskonzepts ergeben sich Isolationseigenschaften, die eine Partitionierung von Softwarekomponenten vorgeben. Softwarekomponenten bestehen aus einer oder mehreren Softwareeinheiten, die zusammen eine Funktion erfüllen [Isoa]. Zwischen Partitionen muss Rückwirkungsfreiheit sichergestellt werden. Im Sinne des ISO Standards 26262 sind Partitionen die Sammlung von Softwarekomponenten gleicher Automotive Integrity Safety Level (ASIL)-Einstufungen [Isod], Systeme können aber auch mit anderen Granularitäten partitioniert werden. Hierzu gehört auch die Abgrenzung der Daten[1] der einzelnen Kontrollflüsse untereinander, um nicht erlaubte Zugriffe zwischen den Partitionen und die Korrumpierung von Daten zu verhindern [Isod]. Des Weiteren können auch die einzelnen Daten unter sich, durch das Herstellen von Typ- und Speichersicherheit, voneinander isoliert werden.

1.1.2 Mehrkernprozessoren

Mehrkernprozessoren, wie sie vermehrt im Automobilbereich eingesetzt werden, ermöglichen es, dem steigenden Ressourcenbedarf von Steuergerätesoftware zu entsprechen. Durch ihre hochintegrierte Bauweise helfen sie dabei, Bauraum und damit auch Kosten zu sparen. Gleichzeitig ist ihre effiziente Nutzung nicht trivial. So bieten Mehrkernprozessoren neue Möglichkeiten, die Anforderungen funktionaler Sicherheit umzusetzen [Pre+07]. Durch eine Bindung der sicherheitsrelevanten Softwareeinheiten auf unterschiedlichen Kernen und Speichern kann das Risiko von Rückwirkungen zwischen Softwarekomponenten reduziert werden. Es ergeben sich auf Mehrkernplattformen jedoch auch neue Herausforderungen an die Bindung von Softwarekomponenten. Durch verteilten Speicher, Buszugriffszeiten und unterschiedliche Kernarchitekturen wird eine Analyse des Laufzeitverhaltens

[1] In diesem Fall Variablen und Konstanten der Anwendung

erschwert [Cul+10] sowie die Laufzeiteffizient des Systems beeinträchtigt. Entsprechend muss bei der Bindung auf diese Aspekte geachtet werden.

1.1.3 Modellbasierte Softwareentwicklung

Auf eingebetteten Systemen eingesetzte Software dient der Regelung oder Steuerung von elektromechanischen Fahrzeugkomponenten. Die hierzu verwendeten Regelalgorithmen werden heutzutage meist modellbasiert in domänenspezifischen Sprachen (Domain Specific Language) (DSLs) entwickelt [Mö10]. Dies hat den Vorteil, dass die Regelungsdomäne auf einer höheren Abstraktionsebene einfacher betrachtet werden kann. Auf Grund der unterschiedlichen Abstraktionsebenen, auf denen die Entwicklung von regelungstechnischen Anwendungen und die Systemintegration stattfinden, sind in der modellbasierten Entwicklung die Gegebenheiten der Zielhardware sowie die des Gesamtsystems nicht komplett abgebildet. Für den Einsatz auf der Zielplattform wird die DSL durch Code-Generierung in eine hardwarenahe Zielsprache übersetzt. Dies ist in der Automobilindustrie meist eine Version des C-Standards. Der so entstehende Code wird anschließend in die anwendungsspezifische Systemsoftware, bestehend aus Betriebssystem (BS) und Basissoftware (BSW), integriert. Er ist aber, ohne weiteres Einwirken, weiterhin anwendungs- und hardwareagnostisch.

1.1.4 Plattformgestützte Softwareentwicklung

Eine Möglichkeit, die Wiederverwendbarkeit von Regelungsalgorithmen zu erhöhen und damit Kosten in der Entwicklung zu sparen, ist die Entwicklung einer bibliotheksbasierten Softwareplattform. Dies bietet sich zum Beispiel für die Regelung von elektrischen Antrieben an, da innerhalb eines Motortypen der grundlegende Regelungsalgorithmus ähnlich gestaltet ist und sich nur durch motor- und anwendungsspezifische Parameter unterscheidet. Bei einem solchen Plattformansatz müssen die entwickelten Modelle so generisch wie möglich gestaltet werden, sodass sie für eine Vielzahl von Anwendungen verwendet werden können. Eine Anwendung ist die produkt- oder projektspezifische Kombination von Plattformbibliotheken, Betriebssystemkomponenten und Basissoftwarekomponenten. Die Anwendungsagnostik der Modelle hat allerdings auch zur Folge, dass kein Anwendungswissen eingebracht werden kann und für eine generische Zielhardware entwickelt werden muss. Hierdurch ist es nicht möglich, Bindungsentscheidungen auf Grundlage von anwendungsspezifischen Anforderungen in Modelle einfließen zu lassen. Dies muss manuell, während der Integration des Reglers in das Gesamtsystem, durchgeführt werden, was mit einem hohen Entwicklungsaufwand und damit mit Kosten verbunden ist.

1.2 Lösungsansatz

Aus den in den Kapitel 1.1 beschriebenen Problemen stellt sich die Frage: Wie können die nichtfunktionalen Eigenschaften Laufzeit, Speichernutzung sowie räumliche Isolation in einem das Gesamtsystem berücksichtigenden Entwicklungsprozess durch Maßnahmen auf Implementierungsebene gezielt beeinflusst werden?

Hierzu wird in dieser Arbeit das Magic Memory Handling Tool (MMHT), als Werkzeug zur Unterstützung des Entwicklungsprozesses, entwickelt. Ausgangspunkt hierfür ist eine Anwendung, bei der durch statische Analyse Typ- und Speichersicherheit hergestellt wird. Durch Abbildung der Laufzeitumgebung der Anwendung ist es des Weiteren möglich, das Zugriffsverhalten des Systems zu berücksichtigen. Dies wurde durch eine Erweiterung des statischen Analysewerkzeugs Astrée im Rahmen des ARAMiS II Forschungsprojektes ermöglicht. Das MMHT ermöglicht es, Isolations-

eigenschaften und die Anforderungen der Laufzeiteigenschaften auf Implementierungsebene zu beeinflussen. Hierzu berücksichtigt es die aus der Systemspezifikation stammende Ablaufplanung sowie die Allokationsentscheidungen innerhalb der Anwendung. Ausgehend darauf werden Anwendungsdaten klassifiziert und eine Bindung dieser Daten auf die Hardwarespeicher vorgesehen. Das Ziel der Arbeit ist es, dass die Ergebnisse des MMHTs die folgenden Anforderungen zu adressieren:

Ressourceneffizienz: Die Speicherressourcen der Hardware werden so genutzt, dass die Laufzeiteffizienz des Systems durch die Bindung verbessert wird.

Anpassbarkeit: Die Klassifizierung und die daraus folgende Bindung können ausgehend von einer generischen Anwendung eingesetzt werden und beeinflussen diese, abhängig von der auf Architekturebene entworfenen Systemspezifikation.

Unterstützung des Anwendungsentwicklers: Die Klassifizierung von Daten erfolgt automatisiert. Zusätzlich wird die Code-Generierung aus modellbasiert entwickelten Anwendungen dahingehend angepasst, die Ergebnisse der Datenklassifizierung zu beeinflussen.

Vorhersagbarkeit: Die Speicherressourcen der Hardware werden so genutzt, dass Interferenzquellen im Zugriffsverhalten vermieden werden. Hierdurch wird es Werkzeugen zur Bestimmung des Laufzeitverhaltens ermöglicht, ihre Präzision zu erhöhen.

Rückwirkungsfreiheit: Die Klassifizierung ermöglicht die räumliche Isolation von Daten auf Kontrollflussebene, basierend auf dem Zugriffsverhalten der Anwendung. Hierbei werden die Anforderungen der Sicherheitsarchitektur berücksichtigt.

1.3 Aufbau der Arbeit

Im Folgenden wird beschrieben, wie die vorgestellten Ziele im Rahmen dieser Arbeit erreicht werden. Dazu wird in Kapitel 2 der Stand der Forschung zu Interferenzquellen, der Ablaufplanung in Echtzeitsystemen und zur Schaffung räumlicher Rückwirkungsfreiheit betrachtet. Anschließend wird in Kapitel 3 ein ganzheitlicher Entwicklungsprozess, aufbauend auf dem Stand der Forschung, vorgestellt sowie allgemein beschrieben, auf welche Weise die Ziele dieser Arbeit im Kontext dieses Entwicklungsprozesses erreicht werden sollen. In Kapitel 4 wird eine Implementierung der in Kapitel 3 dargstellten Konzepte vorgestellt. Abschließend wird diese Implementierung anhand einer Beispielanwendung in Kapitel 5 evaluiert und bewertet. In Kapitel 6 werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf die Fortsetzung dieser Arbeit gegeben.

2 GRUNDLAGEN UND VERWANDTE ARBEIT

In diesem Kapitel werden die Hintergründe, auf denen die Arbeit aufbaut, erläutert. Hierzu werden Interferenzquellen, die sich negativ auf die Laufzeiteffizienz auswirken können, beschrieben. Anschließend wird kurz auf die Ablaufplanung in Echtzeitsystemen und auf Probleme bei der Bestimmung von Ausführungszeiten eingegangen. Abschließend werden Maßnahmen zur Schaffung räumlicher Rückwirkungsfreiheit auf Hardware- und Softwareebene beschrieben und ein Fazit gezogen.

2.1 Interferenzquellen

Interferenzen können auf Prozessoren die Laufzeiteffizienz negativ beeinflussen. Im Folgenden werden zum einen Caches als Interferenzquellen beschrieben und mit prozessornahen Speichern eine Alternative zu Caches beschrieben. Zum anderen wird kurz auf Interferenzen durch geteilte Ressourcen eingegangen und wie prozessornahe Speicher die daraus resultierenden Interferenzen reduzieren können.

2.1.1 Caches

In modernen Rechensystemen wird der Hauptspeicher meist in DRAM-Technologie umgesetzt. Die Rate, in der die Leistungsfähigkeit von Prozessoren zunimmt, ist aber um ein fünffaches höher als die Rate, mit der die Datenrate von DRAM ansteigt (siehe Abbildung 2.1). Das selbe Problem stellt sich in eingebetteten Prozessorarchitekturen [MMB03]. Speicher, welche in SRAM-Technologie umgesetzt werden, ermöglichen zwar schnellere Zugriffszeiten, sind aber teurer als vergleichbar dimensionierte DRAM [PH13]. Zur Überbrückung der Leistungslücke werden deshalb verschiedene Speichertechnologien kombiniert eingesetzt. Es werden kleine und direkt mit dem Prozessor verbundene Cache-Speicher in SRAM-Technologie verwendet, auf die teilweise innerhalb eines Taktzykluses zugegriffen werden kann. Der im Vergleich zum Cache-Speicher wesentlich größere Hauptspeicher in DRAM-Technologie ist meistens über einen Datenbus angebunden. Dies führt in Kombination mit der geringeren Datenrate von DRAM zu langsameren Zugriffszeiten auf den Hauptspeicher. In einer letzten Stufe finden sich in der Regel nicht-flüchtige Speicher wie Flash-Speicher und EEPROMs. Daten werden aus den nicht-flüchtigen Speichern zur Laufzeit in den Hauptspeicher geladen. Von dort werden sie bei Bedarf in den Cache-Speicher geladen. Dieses Prinzip wird Speicherhierarchie genannt und ist in Abbildung 2.2 dargestellt [PH13].

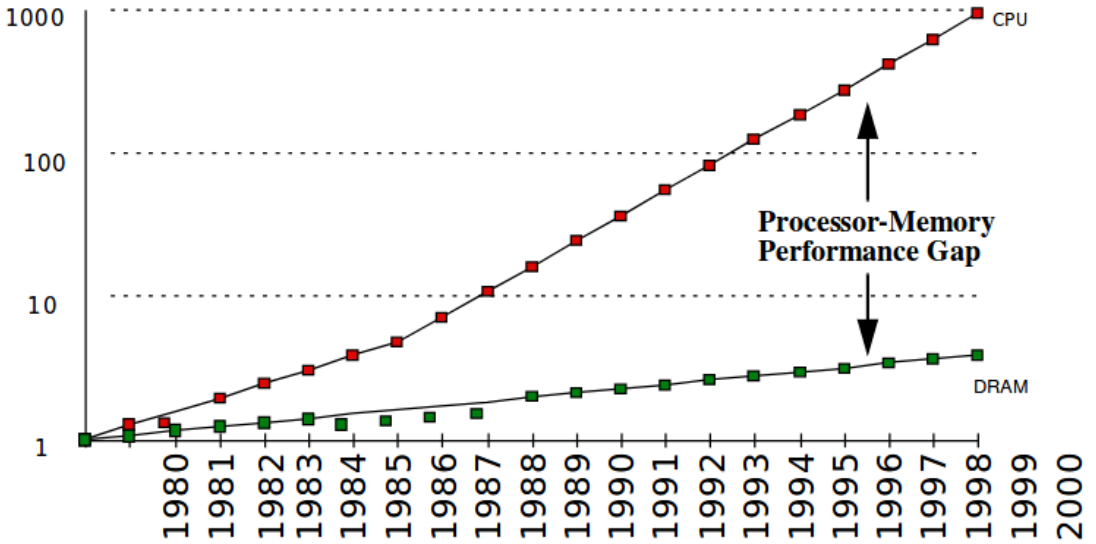


Abbildung 2.1 – Die Prozessor-Speicher-Leistungslücke [PH98].

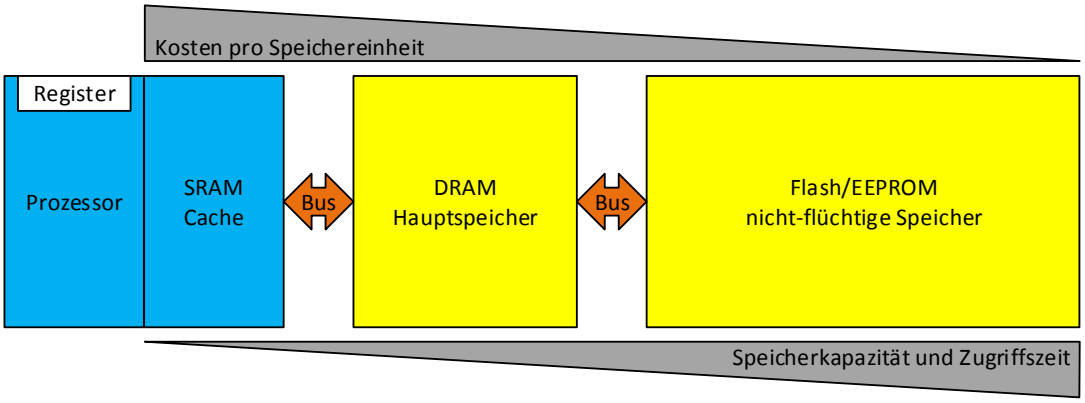


Abbildung 2.2 – Speicherhierarchie mit aufgetragener qualitativer Änderung von Speicherkosten, Speicherkapazitäten und Zugriffszeiten für verschiedene Speichertechnologien [PH98].

Da der Cache-Speicher nicht alle Daten, die vom Prozessor benötigt werden, gleichzeitig enthalten kann, werden zwei Prinzipien zur Befüllung des Caches angewandt, um die Zugriffszeit auf Daten zu reduzieren. Das räumliche Lokalitätsprinzip sagt aus, dass, wenn auf ein Datum zugegriffen wird, die Wahrscheinlichkeit hoch ist, dass darauf folgend auch auf andere, in umliegenden Speicherregionen befindliche, Daten zugegriffen wird [PH13]. Das zeitlichen Lokalitätsprinzip sagt aus, dass, wenn auf ein Datum zugegriffen wird, die Wahrscheinlichkeit hoch ist, dass in naher Zukunft erneut auf das Datum zugegriffen wird [PH13]. Durch die Ausnutzung der räumlichen und zeitlichen Lokalität können Daten in Caches geladen werden, bevor sie benötigt werden und so die durchschnittliche Zugriffszeit auf Daten reduziert werden. Befindet sich ein angefordertes Datum im Cache (Cache-Hit), ist die Zugriffszeit minimal. Im gegensätzlichen Fall (Cache-Miss) erhöht sich die Zugriffszeit, da zuerst der Cache-Miss festgestellt werden muss. Anschließend wird das Datum aus dem Hauptspeicher in den Cache geladen und abschließend das Datum erneut aus dem Cache

angefordert [PH13]. Auf Mehrkernprozessoren verfügt oft jeder Kern über einen eigenen Cache. In einigen Fällen befindet sich zwischen den Prozessoren und dem Hauptspeicher noch ein zusätzlicher, größerer und geteilter Cache.

Ein Problem von Caches kann fehlende Cache-Kohärenz sein. Wird bei Mehrkernsystemen ein Datum in einen kernnahen Cache geladen und dort modifiziert, gibt es zwei grundlegende Strategien, wie verfahren werden kann: Write-Through und Write-Back. Bei einer Write-Through-Strategie wird bei einer Veränderung eines Datums im Cache gleichzeitig das Datum an seinem ursprünglichen Speicherort verändert. Dies kann allerdings nicht so schnell erfolgen, wie Daten im Cache verändert werden. Bei einer Write-Back-Strategie werden Veränderungen eines Datums im Cache erst dann auf den ursprünglichen Speicherort gespiegelt, wenn das Datum aus dem Cache entfernt wird. Dies hat den Vorteil, die Anzahl der Schreibzugriffe außerhalb des Caches zu reduzieren [PH13]. Gleichzeitig führt es allerdings dazu, dass der Wert eines Datums im Hauptspeicher und im Cache-Speicher voneinander abweichen kann. Während dies bei Einzelkernsystemen in der Regel kein Problem ist, da es keine verteilten Cache-Speicher mit unterschiedlichen Datensätzen gibt, kann es in Mehrkernsystemen zu Problemen führen [Fuc10].

Zur Verdeutlichung wird das Problem fehlender Cache-Kohärenz an einem Beispiel beschrieben. In Listing 2.1 wird durch zwei auf verschiedenen Prozessorkernen ausgeführten Fäden ein globales Datum k modifiziert. In Abbildung 2.3 wird die Ausführung auf der CPU und die Speicherzustände des Datums k in den verschiedenen Speichern illustriert. Abbildung 2.4 zeigt die Reihenfolge der Ausführungszeitpunkte und die Intervalle von t_i und t_j, die beschrieben werden. Zum Zeitpunkt t_0 wird k mit 0 initialisiert und in den darauf folgenden Zeitpunkten t_1 bis t_3 auf CPU0 in den Cache geladen, modifiziert und zurück in den Cache geschrieben. Auf Grund der Write-Back-Strategie wird es allerdings erst zu einem Zeitpunkt t_i, zu dem der Eintrag im Cache für k entfernt wird, in den Hauptspeicher zurückgeschrieben. Während der Zeitpunkte t_4 bis t_6 wird das Datum in den Cache der CPU1 geladen, dort modifiziert und in den Cache zurückgeschrieben. Liegt t_i hinter t_4 wird allerdings der unmodifizierte Wert von k=0, statt dem erwarteten Wert von k=-3, aus dem Hauptspeicher geladen und es existieren zwei mögliche Werte für k. Wird k erst nach t_i aus dem Hauptspeicher geladen, wird durch `Thread_2_CPU1` der korrekte Wert aus dem Hauptspeicher geladen. k besitzt dann den erwarteten Wert von k=7. Je nachdem, wann der Zeitpunkt t_j stattfindet, kann es wieder zu dem selben Problem kommen.

```
1  volatile int k = 0;
2
3  Thread_1_CPU0(void){
4      //Do stuff
5      k -= 3;      // t2 k becomes modified by Thread 1 on CPU0
6      // Do stuff
7  }
8
9  Thread_2_CPU1(void){
10     //Do stuff
11     k += 10;     // t4 k becomes modified by Thread 2 on CPU1
12     // Do stuff
13 }
```

Listing 2.1 – Cache-Kohärenz illustrierndes Code-Beispiel.

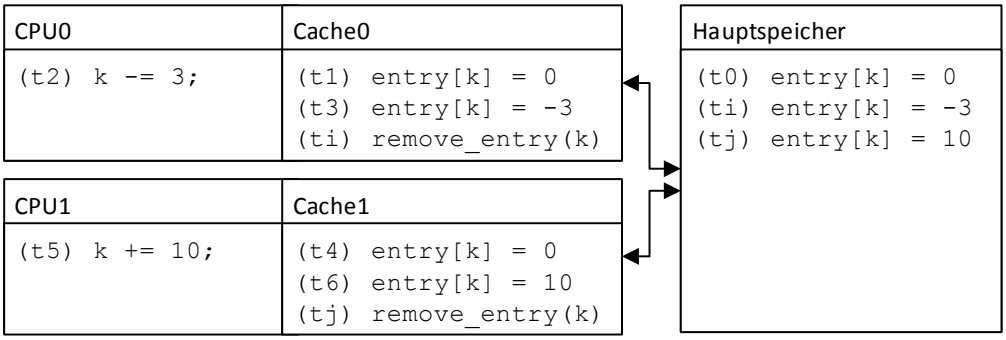


Abbildung 2.3 – Illustration der Programmausführung von Listing 2.1 und die zugehörigen Speicherzustände des Datums k.

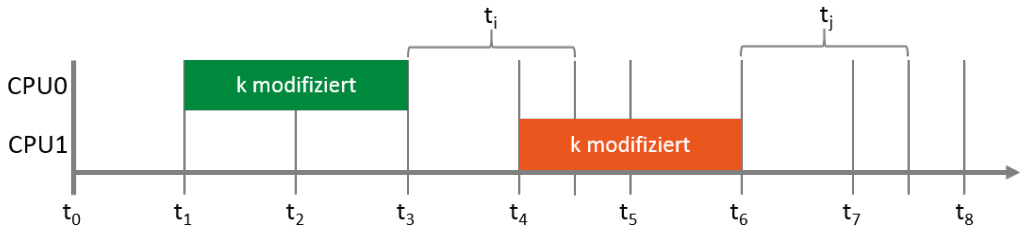


Abbildung 2.4 – Zeitlicher Ablauf der Zeitpunkte.

Eine Architektur mit Cache-Kohärenz verhindert dieses Verhalten. Allerdings verringert die Implementierung von Cache-Kohärenz die Laufzeiteffizienz und führen zu einem erhöhten Datenaustausch zwischen den Caches [GWM92]. Dies bedingt zu Interferenzen auf dem Bus, da der Austausch anderer Daten behindert werden kann. Eine mögliche Lösung ist der Verzicht auf Cache-Kohärenzmechanismen. Dies ist sinnvoll, wenn beispielsweise nur Daten im Cache gespeichert werden, auf die nur lesend zugegriffen wird. Ein anderes Szenario, in dem auf Cache-Kohärenz verzichtet werden kann ist, wenn auf Daten nur durch einen Kern zugegriffen wird und diese deshalb nicht in andere Caches geladen werden müssen. Zur Reduzierung der Zugriffszeiten können alternativ auch Scratchpad-Speicher (Scratchpad Memories) (SPMs) eingesetzt werden [Kan+01]. Dies sind prozessornahe Speicher in SRAM-Technologie, die die gleichen Zugriffszeiten wie Cache-Speicher verwirklichen. Mehrkernprozessoren verfügen in einigen Fällen über einen SPM pro Prozessorkern [Roo12], auf die auch andere Kerne über den Bus zugreifen können. Daten können auf SPM entweder statisch vor der Laufzeit gebunden werden [ABS02] oder dynamisch während der Laufzeit, auf die Anwendung angepasst, geladen werden [Guo+11; Kan+01]. Durch Zugriffssynchronisation kann die Datenkonsistenz in SPM gewahrt werden, ohne, dass zwischen Cache-Speichern Informationen über die Datenkohärenz ausgetauscht werden müssen. Dies verringert Interferenzen auf dem Bus, jedoch kann der Einsatz von Zugriffssynchronisation zu Laufzeitkosten führen.

2.1.2 Geteilte Ressourcen

Durch Prozessoren und komplexe Peripheriegeräte geteilte Ressourcen stellen eine weitere Quelle von Interferenz in Rechensystemen dar. Zu geteilten Ressourcen können:

- Speicher,
- Busse und
- E/A-Peripheriegeräte

gehören [Fuc10]. Greifen zwei oder mehr Konsumenten auf die selbe geteilte Ressource zu, welche zu wenige Anfragen gleichzeitig behandeln kann, müssen Zugriffe abgeblockt werden oder der verdrängte Konsument muss warten, bis die Ressource wieder frei gegeben wird [Fuc10]. Durch den Einsatz von prozessornahen Speichern, wie SPMs, kann die Anzahl der Zugriffe auf geteilte Speicher verringert werden. Dies kann dadurch erreicht werden, indem ein Datum auf dem prozessornahen Speicher des Kerns platziert wird, von dem aus als einziges oder am häufigsten auf das Datum zugegriffen wird. Dies reduziert die Anzahl der Zugriffe über den Bus, auf geteilte Speicher und damit auch Blockadesituationen und -zeiten.

2.2 Echtzeitsysteme

Fahrzeugsteuergeräte, die regelungstechnische Funktionen erfüllen, müssen echtzeitfähig sein. Zur Gewährleistung der Echtzeitfähigkeit muss das Laufzeitverhalten während des Systementwurfs geplant werden und nach dessen Implementierung verifiziert werden. Im Folgenden wird die grundlegende Terminologie der Ablaufplanung vorgestellt und kurz ein Verfahren zur Planung statischer Ablaufpläne beschrieben. Abschließend wird in Bezug auf die in Kapitel 2.1 beschriebenen Interferenzen aufgezeigt, wie diese zu Problemen bei der Laufzeitanalyse führen können.

2.2.1 Ablaufplanung

Ziel der Ablaufplanung ist es, einen zulässigen Ablaufplan zu erstellen. Dieser beschreibt, wann eine logische Einplanungseinheit auf einer Recheneinheit ausgeführt wird. Eine logische Einplanungseinheit wird charakterisiert durch einen Einlastungszeitpunkt, eine Ausführungszeit und einen relativen Termin [Liu00]. Die Parameter der Einplanungseinheiten ergeben sich aus den zeitlichen Anforderungen der Anwendung. Bei den in dieser Arbeit betrachteten Regelungssystemen erfolgt die Einlastung von Einplanungseinheiten zum größten Teil periodisch. Eine solche periodische Einplanungseinheit ist in Abbildung 2.5 mit den oben aufgeführten Größen dargestellt. Periodischen Einplanungseinheiten stehen nicht-periodische und sporadische Einplanungseinheiten gegenüber. Diese werden nicht in zeitlich regelmäßigen Abständen eingelastet, sondern beim Eintreten von Ereignissen [Liu00]. Die Einlastungszeitpunkte von nicht-periodischen und sporadischen Einplanungseinheiten werden durch eine Zwischenankunftszeit charakterisiert, die den zeitlichen Abstand zwischen zwei Einlastungen durch eine Wahrscheinlichkeitsverteilung beschreibt [Liu00].

Ein zulässiger Ablaufplan muss folgende Bedingungen erfüllen:

- Jedem Prozessor wird zu jedem Zeitpunkt genaue eine Einplanungseinheit zugeordnet.
- Jeder Einplanungseinheit wird zu jedem Zeitpunkt genau ein Prozessor zugeordnet.
- Keine Einplanungseinheit wird vor ihrem Einplanungszeitpunkt eingelastet.

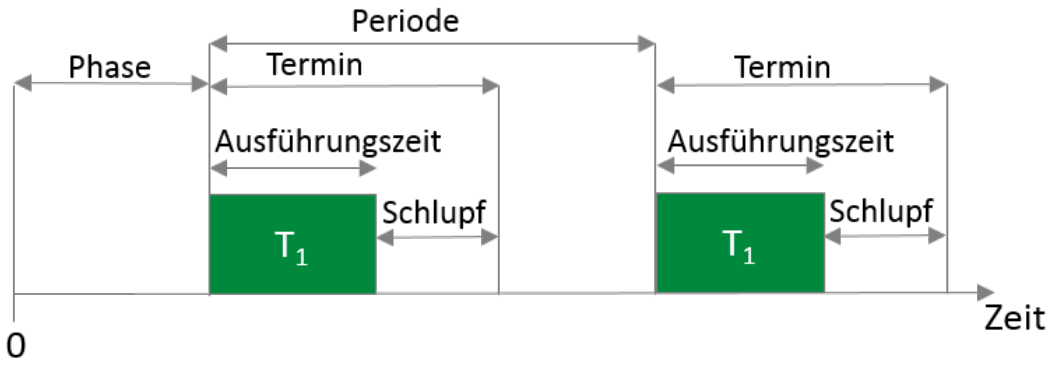


Abbildung 2.5 – Darstellung der charakteristischen Eigenschaften eines periodischen Arbeitsauftrags T_1 nach [Ulb16].

- Die einer Einplanungseinheit zugeteilte Rechenzeit ist gleich ihrer maximalen Ausführungszeit (Worst Case Execution Time) (WCET) oder ihrer tatsächlichen Ausführungszeit.
- Alle Randbedingungen, wie Ausführungsreihenfolgen und die Betriebsmittelnutzung, werden erfüllt.
- Alle Einplanungseinheiten werden vor ihrem Termin vollständig ausgeführt.

Erfüllt ein Ablaufplan diese Bedingungen, ist das System planbar und der Ablaufplan zulässig [Liu00].

Zur Erstellung eines gültigen Ablaufplans existieren verschiedene Verfahren. Diese können entweder vor der Laufzeit zur Erstellung eines statischen Ablaufplans dienen oder während der Laufzeit dynamisch Einplanungsentscheidungen treffen. Ratenmonotone Ablaufplanung (Rate Monotonic Scheduling) (RMS) ist ein Verfahren zur Bestimmung eines statischen Ablaufplans. Bei diesem werden den Einplanungseinheiten Prioritäten zugewiesen, die invers zu ihrer Periode sind. Dies bedeutet, dass eine Einplanungseinheit mit kurzer Periode eine hohe Priorität zugewiesen bekommt. Aus den Perioden und Prioritäten der Einplanungseinheiten ergibt sich der Ablaufplan.

Neben der Unterscheidung in Systeme mit periodischen und Systeme mit nicht-periodischen Einplanungseinheiten, werden Echtzeitsysteme weiter unterteilt. Es wird unterschieden, wie bei der Verletzung eines Termins vorgegangen wird. Unter der Verletzung eines Termines versteht man, dass die Ausführungszeit einer Einplanungseinheit länger dauert als der relative Termin [Liu00]. In weichen Echtzeitsystemen werden Einplanungseinheiten weiter ausgeführt, wenn sie ihren Termin verletzen. Die Ausführung der Einplanungseinheit wird gegebenenfalls beendet. In harten Echtzeitsystemen, die sicherheitskritische Aufgaben erfüllen, muss bei einer Terminverletzung eine Ausnahmebehandlung durchgeführt werden [Liu00]. Ein Beispiel hierfür ist das Verfahren einer Anlage in einen sicheren Zustand. Bei der Ablaufplanung muss auch berücksichtigt werden, dass für diese Ausnahmebehandlung genügend Rechenzeit zur Verfügung steht [Liu00]. Die Terminverletzung einer Einplanungseinheit darf nicht dazu führen, dass dies Auswirkungen auf andere Einplanungseinheiten hat. Dies wird durch zeitliche Isolation der Einplanungseinheiten verhindert, indem zum Beispiel durch eine Ausführungszeitüberwachung vermieden wird, dass nachfolgende Einplanungseinheiten von vorhergehenden verzögert ausgeführt werden.

Orthogonal dazu wird zwischen zeitgesteuerten und ereignisgesteuerten Systemen unterschieden. Bei Ersteren erfolgt die Einlastung zu, durch einen Taktgeber vorgegebenen, Zeitpunkten. Nicht-periodische Einplanungseinheiten werden entweder in die Ausführung eingereiht oder periodisch abgefragt. Bei ereignisgesteuerten Systemen wird die Einlastung von Einplanungseinheiten durch Ereignisse ausgelöst, diese können periodisch oder nicht-periodisch auftreten [Liu00].

2.2.2 Probleme der Laufzeitanalyse

Moderne Rechnersysteme werden zunehmend leistungsfähiger aber auch komplexer. Maßnahmen, welche die durchschnittliche Ausführungszeit von Operationen verbessern sollen, führen gleichzeitig zu einer verringerten Präzision von der Bestimmung von WCETs beziehungsweise von maximal beobachteten Ausführungszeiten (Worst Observerd Execution Time) (WOET).

Ein Beispiel hierfür sind die in Kapitel 2.1.1 beschriebenen Cache-Speicher. Je nachdem, ob ein Cache-Hit oder ein Cache-Miss bei der Anforderung eines Datums durch den Prozessor eintritt, unterscheidet sich die Zugriffszeit auf das Datum. Für eine präzise Bestimmung der maximalen Ausführungszeit muss der Speicherzustand des Caches bekannt sein. Je nach verwendeter Cache-Implementierung gestaltet sich dies jedoch unterschiedlich schwer und erfordert eine genaue Abbildung der Hardware bei statischen Laufzeitanalysen [Cul+10]. Zur Bestimmung von WOETs müssen Laufzeitmessungen so gestaltet werden, dass Cachingeffekte berücksichtigt werden. Die in Kapitel 2.1.1 beschriebenen SPMs bieten eine Alternative für die Verwendung von Cache-Speichern. So können auf Grund der schnellen Zugriffszeiten Cache-Speicher durch SPM ersetzt werden, deren Inhalt jedoch im Vergleich zu Caches bereits vor der Laufzeit [ABS02] oder dynamisch und deterministisch zur Laufzeit [Guo+11] bestimmt werden kann. Dies erhöht die Präzision von WCET-Analysen beziehungsweise WOET-Bestimmungen.

Ein weiteres Problem bei der Bestimmung von Ausführungszeiten stellt der Zugriff auf geteilte Ressourcen dar. Diese führen, wie in Kapitel 2.1.2 beschrieben, zu Blockadesituationen. Eine Möglichkeit, diese zu vermeiden, ist der Einsatz privater Speicher [Cul+10]. Werden die Daten von Funktionseinheiten auf ihnen zugeteilten Speichern abgelegt, wird die Anzahl an konkurrierenden Zugriffen auf geteilte Speicher verringert und es entstehen weniger Blockadesituationen, die sich negativ auf die Präzision der Bestimmung von Ausführungszeiten auswirken. Direkt mit Prozessoren verbundene SPMs sind ein Beispiel für solche privaten Speicher. Eine andere Möglichkeit ist es, unterteilte Speicher, wie zum Beispiel Flash-Speicher, zu nutzen, deren Speichersegmente einzelnen Funktions- oder Recheneinheiten zugeteilt werden. Hierdurch wird die Anzahl der konkurrierenden Zugriffe auf geteilte Speicherressourcen verringert. Je nach eingesetztem Synchronisationsmechanismus kann es beim Zugriff auf geteilte Ressourcen zu Blockadezeiten kommen. Je nach Implementierung sind diese nicht trivial bestimmbar und führen so zu einem Präzisionsverlust bei der Bestimmung von Ausführungszeiten [Cul+10]. Eine Möglichkeit, dies zu verhindern, ist die Anwendung von Synchronisationsmechanismen, die deterministische obere Schranken für Blockadezeiten garantieren [Cul+10].

2.3 Räumliche Rückwirkungsfreiheit

Die Sicherstellung räumlicher Rückwirkungsfreiheit ist eine Maßnahme, um die Fehlerausbreitung zwischen Funktionseinheiten zu unterbinden. Ziel ist es zu verhindern, dass Funktionseinheiten Daten anderer Funktionseinheiten unbeabsichtigt überschreiben können oder auf Speicheradressen zugreifen, die der Funktionseinheit nicht zugewiesen sind [Per+13]. Im Folgenden wird beschrieben,

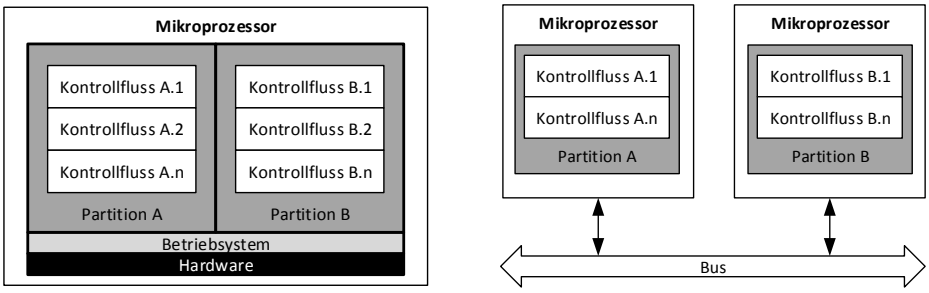
wie räumliche Rückwirkungsfreiheit durch Hardware- und Softwaremaßnahmen sowie statische Analyse sichergestellt werden kann.

2.3.1 Hardwarebasierte Maßnahmen

Um räumliche Rückwirkungsfreiheit zu gewährleisten, kann zum einen die Hardwarearchitektur der Zielplattform genutzt werden. Zum anderen können Hardwareeinheiten Schutz vor unerlaubten Datenzugriffen durchsetzen. Im Folgenden werden Maßnahmen in der Systemarchitektur erläutert, die räumliche Rückwirkungsfreiheit unterstützen. Anschließend wird die Speicherschutzeinheit (Memory Protection Unit) (MPU) als Mittel des hardwarebasierten Speicherschutzes vorgestellt sowie die Verwaltung einer MPU durch das AUTOSAR OS.

2.3.1.1 Systemarchitekturmaßnahmen

Auf Ebene der Systemarchitektur kann durch einen geeigneten Systementwurf räumliche Isolation zwischen Funktionseinheiten hergestellt werden. Zwei grundlegende Ansätze hierzu stellen die *integrierte* und die *förderierte Architektur* dar (siehe Abbildung 2.6). Diese werden auch in der Version des ISO Standards 26262 aus dem Jahre 2009 als Methoden zur Herstellung von Rückwirkungsfreiheit empfohlen [Isoc]. Bei einer integrierten Architektur teilen sich die Funktionseinheiten eine



(a) Integrierte Architektur

(b) Förderierte Architektur

Abbildung 2.6 – Blockdiagrammdarstellung einer integrierten und einer förderierten Architektur nach [Isoc].

Recheneinheit und deren Ressourcen [Sai+15]. Mehrkernsysteme stellen integrierte Architekturen dar [Kop+07]. Dies wird zum Beispiel durch den Zugriff auf geteilte Speicher innerhalb des Prozessors deutlich. Von einer förderierten Architektur spricht man hingegen, wenn die Funktionseinheiten über eigene Rechenressourcen verfügen. Der Austausch von Nachrichten erfolgt in der Regel über ein Bussystem. Während Fahrzeuge als Ganzes auf Grund der im Fahrzeug verteilten Steuergeräte förderierte Architekturen bilden [DNSV10], stellen die einzelnen Steuergeräte integrierte Architekturen dar. Des Weiteren zeichnet sich in der Automobilindustrie ein Trend hin zu integrierten Architekturen für das Gesamtfahrzeug ab [DNSV10].

Zur Herstellung von Rückwirkungsfreiheit in integrierten Architekturen müssen Maßnahmen zur Isolation der einzelnen Funktionseinheiten getroffen werden. Mehrkernsysteme können hierbei durch den Einsatz von privaten Speichern und die Aufteilung von Funktionseinheiten auf die einzelnen Prozessorkerne helfen. Ein solcher Ansatz wird im Rahmen des PikeOS umgesetzt [Sai+15]: Das BS kann im Asymmetric Multi-Processing (AMP) Modus betrieben werden. In diesem Modus operiert

auf jedem Prozessorkern eine eigene Instanz des BS und die Kerne arbeiten unabhängig voneinander. Die Kommunikation erfolgt über dedizierte, geteilte Speicherregionen [Sai+15]. Zur Unterbindung von Fehlerfortpflanzung zwischen den einzelnen Instanzen müssen jedoch weitere Hardware- und Softwaremaßnahmen getroffen werden. Diese müssen Schnittstellen absichern, über die Daten zwischen den Instanzen ausgetauscht werden können und die Interaktion der Instanzen mit geteilten Ressourcen verwalten.

Die Möglichkeiten der Hardware, solche Maßnahmen zu verwirklichen, muss während der Allokation der Funktionseinheiten berücksichtigt werden. Unter *Allokation* versteht man die Zuweisung von Funktionseinheiten auf Ausführressourcen [TH07]. Die Allokation von Funktionseinheiten auf die Kerne stellt jedoch kein triviales Problem dar und ist Gegenstand umfangreicher Forschung (siehe zum Beispiel [HB18] und [Dam+06]).

2.3.1.2 Speicherschutzeinheiten

Eine Möglichkeit, räumliche Isolation durch Hardwaremaßnahmen zu unterstützen, ist der Einsatz von Hardwarespeicherschutzeinheiten. Im Folgenden wird darauf eingegangen, wie hardwareseitig räumliche Rückwirkungsfreiheit zur Laufzeit zwischen Komponenten sichergestellt werden kann und welche Vor- und Nachteile der Einsatz von Speicherschutzhardware hat.

In modernen Prozessoren wird meist eine Speicherverwaltungseinheit (Memory Management Unit) (MMU) eingesetzt, um die Speicherzugriffe der Prozessorkerne auf den Hauptspeicher zu verwalten und Speicherschutz zu ermöglichen. Aus Kostengründen wird Hardwarespeicherschutz auf eingebetteten Systemen durch eine MPU umgesetzt. Diese ist allerdings ebenfalls nicht auf allen Systemen vorhanden. Eine MPU ist ein Hardwarebaustein, der Zugriffe auf Speicheradressen mit Adressbereichen vergleicht und die Zugriffe gegebenenfalls unterbindet. Dadurch können Speicherbereiche vor unerlaubten Zugriffen geschützt werden. Typische geschützte Bereiche sind zum Beispiel Funktions-Stacks oder für das BS reservierte Speicherbereiche.

In Abbildung 2.7 ist eine Beispiel-MPU schematisch dargstellt. Bei dieser Implementierung werden alle Zugriffe auf Adressen in geschützten Speicherregionen gegen in der MPU abgelegte Speicherbereiche geprüft. Neben der Zugriffsadresse wird ebenfalls die Art des Zugriffes (lesend, schreibend, ausführend) kontrolliert. Findet ein Zugriff in einen für den Zugreifenden nicht erlaubten Adressbereich oder auf eine nicht erlaubte Art zu, wird der Zugriff abgeblockt und eine Trap geworfen [Tex]. Dieses Prinzip wird auch auf anderen Architekturen wie ARM Chips [Arm] angewendet. Hierdurch können die Adressbereiche sowohl von Daten- als auch von Instruktionsspeichern gegen Zugriffe aus dem Adressbereich nicht zugeordneter Komponenten geschützt werden. Dies stellt sicher, dass räumliche Rückwirkungsfreiheit zwischen den Softwarekomponenten besteht.

Die Verwaltung der MPU erfolgt meistens durch das BS [Arm]. So wird sowohl in der AUTOSAR OS Spezifikation [AUT17c] als auch in der FreeRTOS Programmierschnittstelle (Application Programming Interface) (API) [Ama] beschrieben, welche Funktionen das BS durch die MPU verwirklichen soll.

Ein Nachteil der Verwendung einer MPU ist, dass entsprechende Prozessorplattformen gekauft werden müssen, die mit MPUs ausgestattet sind. Die Entscheidung für hardwarebasierten Speicherschutz ist also ein Kostenfaktor, der berücksichtigt werden muss. Des Weiteren ist die richtige Konfiguration einer MPU nicht trivial. Je nach Implementierung kann die Genauigkeit der schützbaren Bereiche sehr grob sein [Sti12]. So kann in ARMv8 Prozessoren mit MPU ein beliebig großer Speicherbereich geschützt werden, solange dieser ein Vielfaches von 32 Byte ist; es sind außerdem

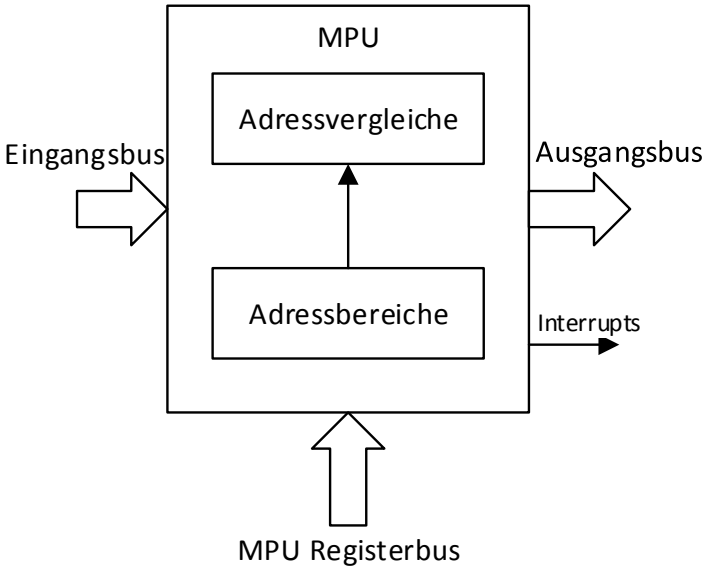


Abbildung 2.7 – Blockdiagramm einer MPU von Texas Instrument in der KeyStone Architektur nach [Tex].

keine Überlappungen der Bereiche erlaubt [Arm]. Die Granularität des Schutzes erlaubt es also nicht, ressourceneffizient ein einzelnes Datum zu schützen, sondern nur Speicherbereiche. Des Weiteren gehen mit dem Einsatz einer MPU auch Laufzeitkosten einher, die berücksichtigt werden müssen [Sti12]. MPUs sind dafür, im Vergleich zu softwarebasiertem Speicherschutz robuster gegenüber umweltbedingten, transienten Fehlern [Sti12].

2.3.1.3 Räumliche Kontrollflussisolation in AUTOSAR

Wie in Kapitel 2.3.1.2 beschrieben erfolgt die Verwaltung von MPUs durch das BS. Im Automobilbereich wird hauptsächlich AUTOSAR OS als BS eingesetzt. AUTOSAR OS ist eine Spezifikation für ein BS, die aus der AUTOSAR Organisation entstanden ist. Diese wurde 2003 von deutschen OEM gegründet, mit dem Ziel die wachsende Komplexität von Software für elektrische Systeme handhabbar zu halten und gleichzeitig die Entwicklung von flexiblen Systemen zu erleichtern [AUT]. Abschließend wird darauf eingegangen, wie durch das AUTOSAR OS die MPU für die Umsetzung von Speicherschutz verwaltet wird. Hierzu wird zuerst auf die Abbildung von Kontrollflüssen in AUTOSAR OS eingegangen.

OS-Applications dienen als logischer Behälter von BS Objekten. Dazu zählen synchrone (in AUTOSAR OS Tasks), asynchrone Kontrollflüsse (in AUTOSAR OS Unterbrechungsroutinen (Interrupt Service Routines) (ISRs)) und Betriebssystemdienste, wie Zähler (Counter) und Alarme (Alarm) [AUT17c]. In Abbildung 2.8 ist der Zusammenhang zwischen Rechenkernen (Cores), Kontrollflüssen und Betriebssystemdiensten dargestellt. Eine OS-Application kann keinen oder mehrere Kontrollflüsse in Form von Tasks oder ISRs verwalten. Jede OS-Application verwaltet keinen oder mehrere Zähler beziehungsweise Alarme. Eine OS-Application wird auf genau einen Rechenkern gebunden. Daraus folgt, dass ein Kontrollfluss auf genau einen Kern gebunden ist. Kontrollflüsse in AUTOSAR OS

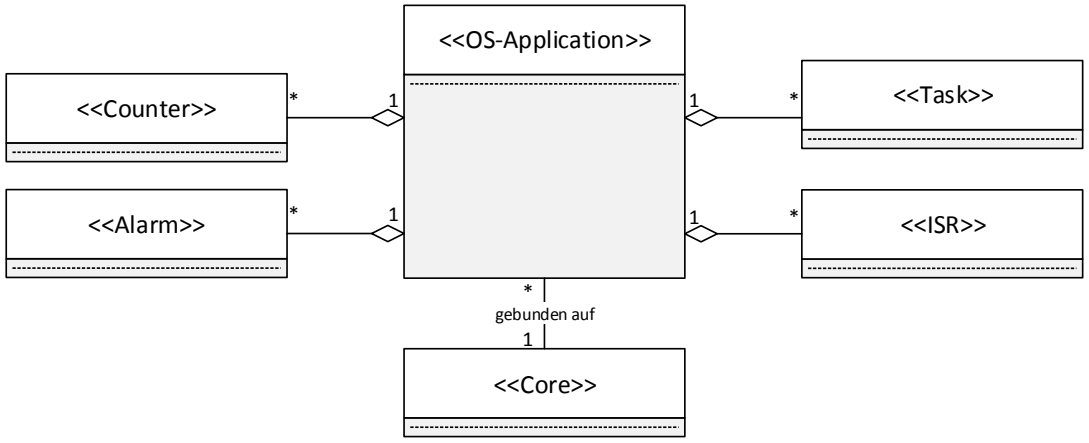


Abbildung 2.8 – Zusammenhang zwischen Rechenkernen, OS-Applications, Kontrollflüssen und Betriebssystemdiensten nach [AUT17c].

können Softwarekomponenten oder Softwareeinheiten abbilden. Obwohl Kontrollflüsse nur auf einem Kern gebunden sind, kann ihre Aktivierung auch kernübergreifend stattfinden [AUT17c].

Alarme stellen Ereignisse dar, die an einem bestimmten Zeitpunkt oder periodisch auftreten können. Diese Ereignisse können zum Beispiel dazu genutzt werden, um einen Task zu aktivieren [AUT17c]. Hierdurch ist es möglich in einem ereignisgesteuerten BS wie AUTOSAR OS periodische Einplanungseinheiten abzubilden.

Speicherschutzregionen werden in AUTOSAR OS auf Ebene der OS-Applications verwaltet. Jeder OS-Application und jedem Kontrollfluss der OS-Application kann ein privater Daten- und Programmspeicherbereich zugewiesen werden. Auf den Speicherbereich der OS-Application können alle Kontrollflüsse, die ihr zugeordnet sind, zugreifen. Zusätzlich kann der Stack-Speicher jedes Kontrollflusses geschützt werden. Diese Form des Speicherschutzes ist allerdings nur auf Hardwareplattformen möglich, auf denen eine Hardwarespeicherschutzeinheit (MPU oder MMU) vorhanden ist [AUT17c]. In Abbildung 2.9 ist die Schaffung von Speicherschutzregionen beispielhaft dargestellt. Die schwarzen, hervorgehobenen Linien in Abbildung 2.9 grenzen Speicherschutzregionen ein, die gestrichelten Linien OS-Applications . Wie aus der Abbildung ersichtlich ist, verfügt der Kernel des BS über seine eigene Speicherschutzregion, die zusammen mit OS-Applications als „trusted“ eingestuft werden. Diese OS-Applications und der Kernel bilden die Trusted Code Base (TCB) [AUT17c]. Die AUTOSAR OS Spezifikation definiert allerdings nicht, was für eine Einstufung als „trusted“ OS-Application notwendig ist. Die Klassifizierung ist also dem Entwickler überlassen.

Die Umsetzung der Speicherschutzregionen erfolgt durch die MPU der Prozessorkerne. Die Anzahl an Speicherschutzregionen, die eine MPU verwalten kann, ist limitiert. So kann ein Aurix TC27x maximal 16 Datenspeicherbereiche verwalten [Tria]. Wenn mehr Speicherschutzregionen benötigt werden, muss das verwaltende BS diese zu Kontextwechseln in der jeweiligen MPU umkonfigurieren. Dies führt zu erhöhten Laufzeiten, weshalb die Anzahl der Speicherschutzregionen, beziehungsweise die Anzahl der notwendigen Umkonfigurationen, niedrig gehalten werden sollte. Jeder Speicherregion wird zudem zugewiesen, welche Zugriffsarten (lesend, schreibend, ausführend) für diese Region erlaubt sind.

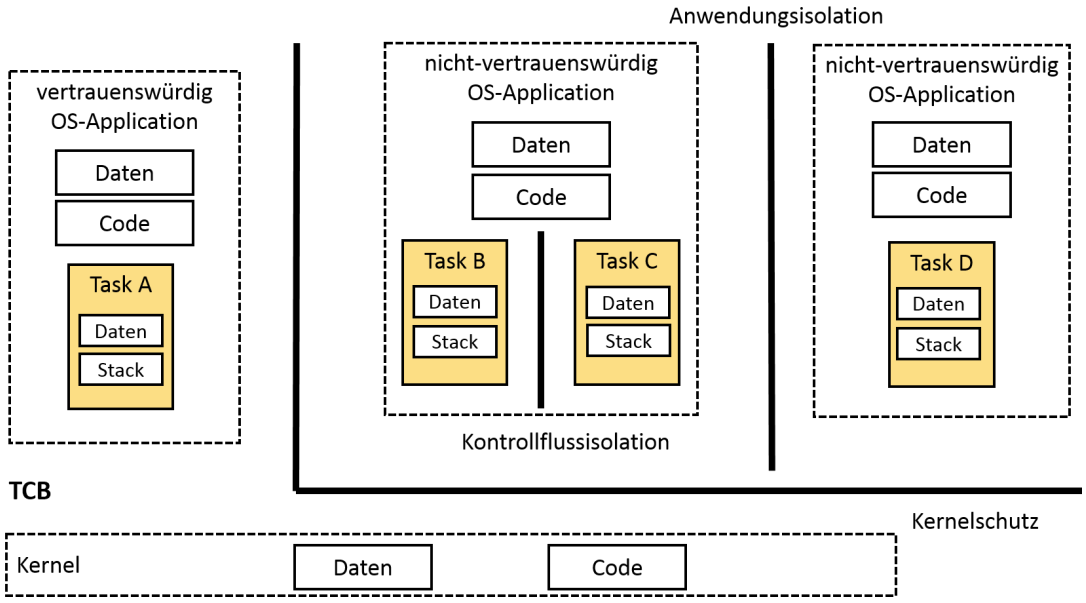


Abbildung 2.9 – Speicherschutzregionen in AUTOSAR OS [Sti17].

Wie in Kapitel 2.3.1.2 beschrieben, ist die Platzierung von Speicherschutzregionen nicht frei wählbar, sondern erfordert je nach Implementierung eine Ausrichtung an bestimmten Speicheradressen. Es kann allerdings möglich sein, dass diese Regionen überlappen. Je nach Implementierung gibt es Unterschiede, wie Zugriffsrechte in überlappenden Speicherschutzregionen gehandhabt werden. Bei Prozessoren der Aurix TC27x-Familie gelten beispielsweise in überlappenden Regionen die Zugriffsrechte, die die größte Zugriffsfreiheit gewähren [Trib]. Erlaubt Region A beispielsweise lesende und schreibende Zugriffe und Region B nur lesende Zugriffe, dann sind in der Schnittmenge der Regionen A und B lesende und schreibende Zugriffe gestattet.

2.3.2 Konstruktive Maßnahmen

Eine Alternative zum Einsatz von hardwarebasiertem Speicherschutz ist softwarebasierter Speicherschutz [Waw09]. Softwarebasierter Speicherschutz kann zum Beispiel durch die Definition von Speicherregionen im Quellcode der Anwendung umgesetzt werden. Die Überprüfung der Einhaltung dieser Regionen bei Datenzugriffen kann statisch vor und dynamisch während der Laufzeit erfolgen [Waw09].

Auf Ebene der Implementierung ist es möglich, die Forderung nach räumlicher Rückwirkungsfreiheit durch eine geeignete Wahl der Programmiersprache zu unterstützen. Rückwirkungsfreiheit wird auf der Implementierungsebene durch konstruktive Typ- und Speichersicherheit erreicht. Unter Typ- und Speichersicherheit versteht man, dass auf Daten nur mit Operationen zugegriffen wird, die auch für den zugehörigen Datentyp definiert sind. Der Zugriff erfolgt nur innerhalb des Datums beziehungsweise nur auf existierende Daten [Aik+06]. Im Folgenden werden zum einen Programmiersprachen wie Java und Ada kurz vorgestellt, bei denen Typ- und Speichersicherheit bereits in der Sprachentwicklung berücksichtigt wurden. Zum anderen wird auf Ansätze wie Cyclone [Gro+05] und CCured [Nec+05] eingegangen. Diese versuchen, durch zusätzliche Laufzeitprüfungen und der Anpassung der C-Programmiersprache, Typ- und Speichersicherheit herzustellen. Abschließend wird

auf MISRA-C [MIS] eingegangen. MISRA ist ein im Automobilbereich verbreiteter Standard, der versucht, typische Programmierfehler in C zu verhindern und die statische Analyse von C-Programmen unterstützen soll.

2.3.2.1 Typsichere Sprachen

Java ist eine der am weitesten verbreiteten Programmiersprachen [Sta18]. Beim Entwurf von Java wurde auf die Ziele Robustheit und Sicherheit geachtet, hierzu werden sowohl Typ- als auch Speichersicherheit durch eine Reihe an Mechanismen erreicht:

- keine impliziten Deklarationen,
- Typüberprüfungen zur Übersetzungszeit,
- Typüberprüfungen zur Link-Zeit von Bibliotheken,
- Referenzen statt Zeiger-Arithmetik,
- Arraygrenzen werden überprüft und
- Typüberprüfung zur Laufzeit [Sun95a; Sun95b].

Durch diese Mechanismen wird Typsicherheit hergestellt, da auf Objekte eines bestimmten Types nur durch Operationen, die diesem Typ zugeordnet sind, zugegriffen werden kann. Hierdurch wird auch gleichzeitig Speichersicherheit erlangt, da Operationen nicht außerhalb eines Objektes erfolgen können. Durch die Einführung von generischen Datentypen[2] in Java 5.0 [Ora04] ist die Typsicherheit nicht mehr für den gesamten Sprachumfang gegeben. Da bei generischen Datentypen Typinformationen nach der Übersetzung verworfen werden, können zur Laufzeit keine Typprüfungen erfolgen. Es existieren jedoch Implementierungen der Java Virtual Machine, die für den Einsatz auf echtzeit- und sicherheitskritischen Systemen entworfen wurden und diese Sprachkonstrukte nicht nutzen [Bol+].

Ada entstand aus einem Projekt des amerikanischen Verteidigungsministeriums. Ziele der Sprache sind Zuverlässigkeit, Wartbarkeit, Effizienz sowie die Berücksichtigung des Menschen als Entwickler [Ada]. Im Sinne der Zuverlässigkeit wird Typ- und damit Speichersicherheit durch:

- explizite Typ- und Variablendeklarationen,
- die Verwendung von nutzerspezifizierten Datentypen mit festen Wertebereichen, statt primitiven Datentypen mit implementierungsabhängigen Wertebereichen,
- den Verzicht auf typenlose Zeiger und
- Typüberprüfungen zu Übersetzungs- und Laufzeit

erreicht [Ada]. Auch durch diese Mechanismen kann in Ada Typsicherheit und damit Speichersicherheit sichergestellt werden.

Beide Sprachen können Typ- und Speichersicherheit herstellen, solange bestimmte Sprachkonstrukte nicht verwendet werden. Allerdings existiert in der Automobilindustrie für eingebettete Systeme eine große Menge an Altcode in C, wodurch die Typ- und Speichersicherheit nur in Projekten, die rein in Java oder Ada umgesetzt wurden, erreicht werden kann.

[2] **Generische Datentypen**: In Java eine Datenklasse, die einen oder mehrere Datentypen deklariert [Gos+18].

2.3.2.2 C-Sprachanpassungen

Anstatt andere Programmiersprachen zu nutzen, wird in Cyclone [Gro+05; Gro+] und in CCured [Nec+05] auf Basis der C-Programmiersprache Typ- und Speichersicherheit hergestellt.

Erklärtes Ziel von Cyclone ist es, den C-Sprachumfang so zu erweitern, dass statische und dynamische Überprüfungen Typsicherheit garantieren können [Gro+02]. Des Weiteren werden moderne Sprachkonstrukte, wie Polymorphie, in Cyclone integriert. Typsicherheit wird durch die Verwendung von annotierten Zeigern erreicht [Gro+05]:

- Fat Pointer (`*@fat x`) sind, im Gegensatz zu einem C-Zeiger, Zeiger, bei denen eine Bereichsprüfung erfolgt, wenn der Zeiger dereferenziert wird oder einen neuen Wert zugewiesen bekommt. Die Bezeichnung "Fat" kommt daher, dass zusätzlich Bereichsinformationen mit dem Zeiger assoziiert werden. Dies ist ähnlich der Implementierung von Referenzen in Java.
- Thin Pointer (`*x`) entsprechen einem klassischen C-Zeiger. Ein solcher Zeiger kann nicht mit Zeigerarithmetik verändert werden, wodurch sichergestellt wird, dass er nur auf ein gültiges Datum zeigen kann.
- Bounded Pointers (`*@numlets(`n`) x`) ermöglichen es, auf den Inhalt eines Arrays mit einem Zeiger zuzugreifen. Hierzu wird durch den Parameter n die Länge des Arrays, beziehungsweise der Zugriffsbereich des Zeigers festgelegt. Dies ermöglicht es, dass zur Übersetzungszeit eine Überprüfung der Zeigerarithmetik durchgeführt werden kann. "Bounded Pointer" sind ein allgemeiner Fall eines "Thin Pointer", der einen "Bounded Pointer" mit $n = 1$ beschreibt [Gro+05].

Orthogonal zu den Zeigertypen können auch Annotationen genutzt werden, die zu zusätzlichen Lauf- und Übersetzungszeitprüfungen führen. Der `@nullable` Vermerk bewirkt zum Beispiel, dass bei der Zeigerdereferenzierung eine NULL-Prüfung der Adresse stattfindet [Gro+]. Zusätzlich können Speicherbereiche annotiert werden, in die ein Zeiger zugreifen darf und die sich ähnlich wie Variablen durch eine Lebenszeit auszeichnen. Dies soll verhindern, dass ein hängender Zeiger dereferenziert wird, indem überprüft wird, ob die entsprechende Region noch aktiv ist. [Gro+02]

CCured verfolgt ebenfalls einen Ansatz, mit strikten statischen Typprüfungen und dynamischen Überprüfungen zur Laufzeit Typ- und Speichersicherheit herzustellen. Um diese zu ermöglichen, verwendet CCured, ähnlich wie Cyclone, spezielle Zeigertypen:

- SAFE: Zeiger, deren Typ nicht verändert wird und deren Ziel nicht durch Zeiger-Arithmetik verändert wird.
- SEQ: Zeiger, deren Typ nicht verändert wird, aber deren Ziel durch Zeiger-Arithmetik verändert wird.
- WILD: Zeiger, deren Typ verändert wird [Nec+05].

Auf Grund ihrer Eigenschaften werden bei SEQ und WILD Zeigern zusätzliche dynamische Prüfungen der Zugriffsgrenzen beziehungsweise des Types durchgeführt. Bei allen Zeigerarten erfolgt bei einer Dereferenzierung eine NULL-Zeiger-Prüfung. Des Weiteren können in CCured Funktionen mit Aufrufparametern dahingehend annotiert werden, dass die korrekte Anzahl und der korrekte Typ der Übergabeparameter durch dynamische Prüfungen sichergestellt wird. CCured verwendet zusätzlich bei der Benutzung von Heap-Speicher automatische Speicherbereinigung und verhindert so, dass unnötiger Speicher durch fehlerhafte dynamische Speicherallokation verbraucht wird. [Nec+05]

Beide Projekte schaffen es, Typsicherheit aufbauend auf C konstruktiv sicherzustellen. Es sind jedoch nicht-automatisierbare Überarbeitungen von Altcode notwendig, um Cyclone oder CCured nutzen zu können. Die zusätzlichen Laufzeitüberprüfungen führen außerdem zu einem erhöhten Rechenbedarf [Gro+05; Nec+05], der die Planbarkeit beeinträchtigen kann. Die Laufzeitüberprüfungen können allerdings Laufzeitfehler verhindern. Sowohl das Cyclone-Projekt als auch CCured wurden jedoch inzwischen beendet und ihre Entwicklung eingestellt.

In MISRA C [MIS] wird ein anderer Ansatz, als in Cyclone und CCured, gewählt: Der C99-Sprachstandard [C99] beziehungsweise der C90-Sprachstandard [C90], die beide hauptsächlich in der Automobilindustrie genutzt werden, werden durch ein Regelwerk auf eine Teilmenge des Sprachstandards beschränkt. Ziel des Standards ist es, zum einen die Fehlervermeidung in C-Programmen, zum anderen die Portabilität von C-Programmen sowie deren Analysierbarkeit durch statische Analysewerkzeuge zu unterstützen. Die Einhaltung bestimmter Regeln des MISRA-Standards kann durch statische Analysewerkzeuge überprüft werden. Durch eine Umsetzung des MISRA C-Regelwerks wird allerdings keine Typ- und Speichersicherheit hergestellt, sondern nur Konstrukte vermieden, die den oben angegebenen Zielen des Standards entgegenstehen. Abweichungen von diesem Regelwerk sind außerdem unter Angabe einer Begründung erlaubt, ohne dass der MISRA-Standard verletzt wird.

2.3.3 Analytische Speichersicherheit

In den Kapiteln 2.3.2.1 und 2.3.2.2 wurde beschrieben, wie konstruktiv durch die Wahl der Programmiersprache, mit den angeführten Nachteilen, Typ- und Speichersicherheit hergestellt werden kann. Diese Eigenschaften sind auch durch den Einsatz statischer Code-Analyse in Anwendungen erreichbar, die in schwach typisierten Sprachen implementiert wurden. Um dies aufzuzeigen, wird zuerst ein Überblick über semantische Code-Analyse und abstrakte Interpretation gegeben. Anschließend wird erläutert, welche Anforderungen ein Werkzeug zur semantischen Analyse erfüllen muss und welche Fehlerkategorien durch sie erkannt werden müssen, um Typsicherheit herstellen zu können. Abschließend wird auf die Möglichkeiten des statischen Analysewerkzeuges Astrée eingegangen, mit denen das Gesamtsystem in die Analyse einbezogen werden kann.

2.3.3.1 Semantische Code-Analyse und abstrakte Interpretation

Semantische Code-Analyse mittels abstrakter Interpretation ist ein Mittel zur Verifikation von Programmen hinsichtlich der Abwesenheit verschiedener Fehlerkategorien. Bei statischen Analyseverfahren wird, im Gegensatz zum klassischen Testen, nicht der Binärcode ausgeführt, sondern der Quellcode oder der Binärcode eines Programmes analysiert. Zur semantischen Code-Analyse wird die Semantik eines Programmes analysiert. Diese beschreibt, welche Operationen bei der Ausführung eines Programmes durchgeführt werden [Aho+06]. Die konkrete Semantik beschreibt dabei alle möglichen Ausführungspfade, die das Programm durchlaufen kann. Sie kann als Verlauf eines Vektors aller Programmzustände über der Programmlaufzeit betrachtet werden (siehe Abbildung 2.10) [Cou05]. Der durch die konkrete Semantik beschriebene Suchraum ist allerdings zu groß, als dass alle möglichen Varianten der Programmausführung auf Fehlerfreiheit überprüfbar sind. Um dieses Problem lösbar zu gestalten, wird durch eine Abstraktionsfunktion die konkrete Semantik auf eine abstrakte Semantik abgebildet. Diese Art der semantischen Analyse wird als abstrakte Interpretation bezeichnet. Die abstrakte Semantik stellt eine Obermenge der konkreten Semantik dar und deckt daher alle Varianten der konkreten Semantik ab. Ist die abstrakte Semantik

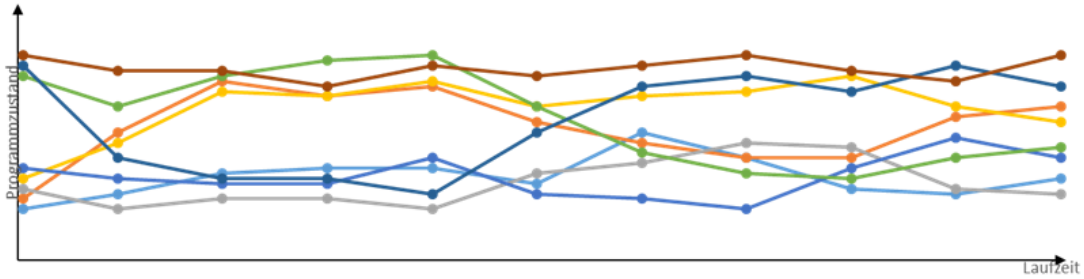


Abbildung 2.10 – Graphische Darstellung der konkreten Programmsemantik nach [Cou05].

sicher, folgt daraus auch, dass die konkrete Semantik sicher ist [Cou05]. Die abstrakte Semantik als Obermenge ist durch einen Schlauch visualisierbar, der über die Vektorverläufe des Programms gelegt wird. Der Schlauch ist in Abbildung 2.11 in grün dargestellt.

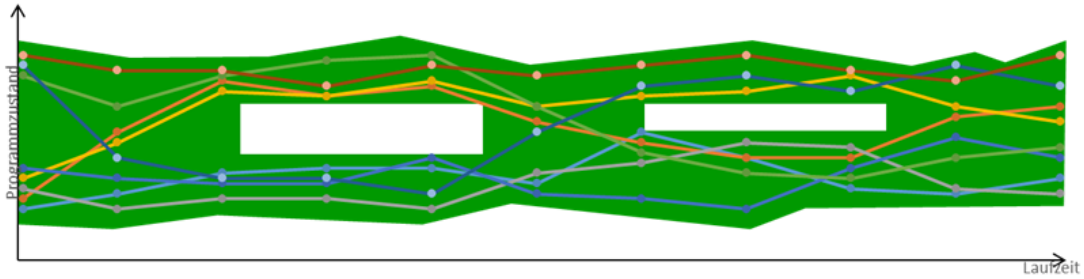


Abbildung 2.11 – Graphische Darstellung der abstrakten Programmsemantik nach [Cou05].

Der so aufgespannte Suchraum kann mathematisch beschrieben und analysiert werden, um Programmfehler zu finden. Programmfehler werden in der semantischen Code-Analyse als verbotene Regionen im Programmvektorraum betrachtet. Verläuft ein Ausführungspfad der konkreten Semantik durch diese Regionen, liegt ein Fehler vor. In der abstrakten Semantik ist dies der Fall, wenn diese (also ihr Schlauch) eine Schnittmenge mit den verbotenen Regionen aufweist (siehe Abbildung 2.12, die verbotenen Regionen sind rot dargestellt) [Cou05].

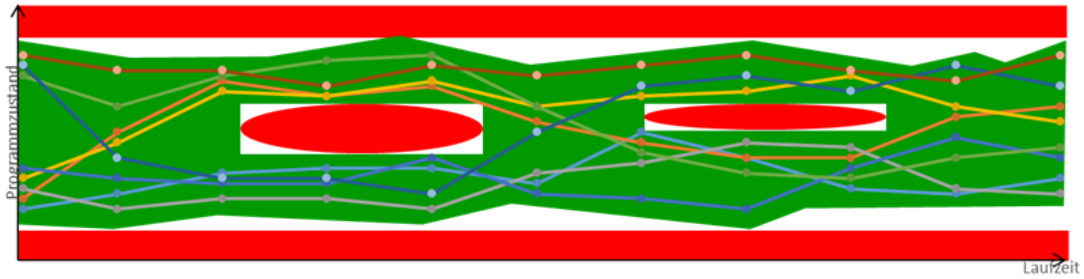


Abbildung 2.12 – Graphische Darstellung der abstrakten Programmsemantik mit verbotenen Regionen im Fehlerfall nach [Cou05].

2.3.3.2 Anforderungen an semantische Code-Analysewerkzeuge

Werkzeuge, mit denen sinnvoll semantische Analysen durchgeführt werden können, müssen diese Anforderungen erfüllen:

- Vollständigkeit und Korrektheit (soundness) ist gegeben, wenn das Werkzeug alle im Programm vorhanden Fehler für die Fehlerkategorien, die es angibt, findet.
- Präzision (precision) wird durch eine möglichst genaue Beschreibung der Vektorverläufe durch die abstrakte Semantik erreicht und vermeidet eine hohe Anzahl an Fehlalarmen.
- Geringe Komplexität erlaubt es, die Laufzeiten der Analyse gering zu halten. [Cul+10]

Ein Werkzeug, bei dem keine Vollständigkeit und Korrektheit gegeben ist, eignet sich nicht, um die Fehlerfreiheit eines Programmes nachzuweisen. Bei einer korrekten Implementierung von abstrakter Interpretation ist diese Eigenschaft gegeben. Durch die Überapproximation im Rahmen der abstrakten Interpretation des Programmzustands kommt es jedoch zu Ungenauigkeiten in der Fehlererkennung und somit zu einem Verlust an Präzision [Our15]. Das bedeutet, dass mehr Fehler gefunden werden, als im Programm vorhanden sind, sogenannte Fehlalarme. Da solche Fehlalarme zu Arbeitsaufwand für den Nutzer führen, ist die Präzision eines korrekten und vollständigen Analysewerkzeugs ein Qualitätsmerkmal [Our15; Sti18]. Gleichzeitig darf die Erhöhung der Präzision nicht dazu führen, dass die Komplexität der Berechnung und damit deren Laufzeit ansteigt. Sonst ist das Werkzeug im Rahmen eines iterativen Entwicklungsprozesses nicht mehr sinnvoll einsetzbar [Cul+10].

2.3.3.3 Fehlerkategorien statischer Analysewerkzeuge

Im Folgenden wird erläutert, welche Fehlerkategorien vollständige und korrekte, semantische Analysewerkzeuge erkennen und wieso es durch diese möglich ist, Typ- und Speichersicherheit analytisch herzustellen.

Die folgenden Fehlerkategorien werden durch vollständig und korrekte, semantische Analysewerkzeuge wie Astréee [Abs] und Polyspace [Thea] detektiert:

1. undefinierte Nutzung von Zeigern und Arrays,
2. ungültige Wertebereiche und Überläufe,
3. undefinierte Shift-Operationen,
4. uninitialisierte Variablen,
5. Divisionen oder Modulo-Operationen mit Null,
6. fehlgeschlagene oder ungültige Anweisungen,
7. falsche oder unerlaubte Funktionsaufrufe,
8. Daten- und Kontrollflussfehler und
9. undefiniertes nebenläufiges Verhalten [Abs; Theb].

Ist ein Analysewerkzeug vollständig und korrekt, kann es die Abwesenheit der oben aufgeführten Fehlerkategorien nachweisen. Um zu illustrieren, wie Typ- und Speichersicherheit nach [Aik+06] durch diese Fehlerkategorien sichergestellt wird, stellt Tabelle 2.1 die Anforderungen der Typ- und

Speichersicherheit den entsprechend detektierten Fehlerkategorien gegenüber. Fehler der Kategorie (6) stammen aus Annotationen innerhalb des Analysewerkzeuges und sind nicht explizit Anforderungen der Typ- und Speichersicherheit zuordenbar. Werden sie allerdings nicht behoben, können Fehler der anderen Kategorien verdeckt werden. Fehler der Kategorie (8) stammen aus fehlerhaften Operationsreihenfolgen oder nichtterminierenden Schleifen, diese lassen sich nicht explizit den Anforderungen der Typ- und Speichersicherheit zuordnen, können jedoch Fehler der anderen Kategorien verdecken. In Klammern ist hinter dem Fehlertyp die zugehörige Fehlerkategorie angegeben, wie sie in Astrée aufgeführt sind [Abs]. Die Fehlertypen und ihre Beschreibungen können [Ast] entnommen werden.

Tabelle 2.1 – Gegenüberstellung der Anforderungen für Typ- und Speichersicherheit zu den von statischen Analysewerkzeugen detektierten Fehlertypen [Abs; Theb].

Anforderung	Fehlertyp (Fehlerkategorie)
Nur für Typ definierte Operationen	Ungültige Zeigervergleiche (1)
	Substraktion von Pointern unterschiedlichen Typs (1)
	Schreibzugriff auf eine Konstante (1)
	Dereferenzierung eines fehlausgerichteten Pointers (1)
	Überläufe von Integer und Gleitzahlen (2)
	Shift-Operationen mit falschem Wert (3)
	Operationen mit uninitialisierten Variablen (4)
	Divisionen oder Modulo-Operationen mit Null (5)
	Modulo-Operationen mit vorzeichenbehafteten Integer-Werten (5)
	Falsche oder unerlaubte Funktionsaufrufe (7)
Zugriff nur auf vorhandene Objekte	Dereferenzierung eines NULL- oder ungültigen Zeigers (1)
	Zeiger auf ungültige oder NULL-Funktion (1)
	Nutzung eines hängenden Zeigers (1)
	Unerlaubte Zeigerarithmetik (1)
	Zeiger-Überlauf bei Derefernzierung (1)
Zugriff nur innerhalb des Objekts	Verlassen des Objektbereichs innerhalb einer Struktur (1)
	Überschreiten der Arraygrenzen (1)
	Dereferenzierung eines fehlausgerichteten Pointers (1)
	Zeiger-Überlauf bei Dereferenzierung (1)

Kann ein statisches Analysewerkzeug die Abwesenheit der in Tabelle 2.1 aufgeführten Fehlertypen nachweisen, ist die Anwendung, von Fehlern während des Übersetzungsprozesses und umweltbedingten transienten Fehlern abgesehen, typ- und speichersicher. Im Gegensatz zu den in Kapitel 2.3.2 vorgestellten Maßnahmen wird hierdurch keine Veränderung der Programmlaufzeit und des Rechenbedarfs durch Laufzeitüberprüfungen verursacht. Die Behebung der durch abstrakte Interpretation gefundenen Fehler kann jedoch zu einer Erhöhung des Rechenbedarfs führen.

2.3.3.4 Berücksichtigung des Laufzeitsystems in der statischen Analyse

Eine statische Analyse kann ebenfalls Fehler durch Nebenläufigkeit (9) erkennen. Die von statischen Analysewerkzeugen erkannten Nebenläufigkeitsfehler sind:

- ungültige Nutzung von synchronisierenden Betriebssystemdiensten,
- Lese-Schreib-Wettlaufsituation,

- Schreib-Schreib-Wettlaufsituation und
- Verklemmungen durch Synchronisationsmechanismen [Ast].

Im Folgenden wird beschrieben, wie die Erkennung dieser Fehler in Astrée, einem Werkzeug zur semantischen Code-Analyse, das sich abstrakte Interpretation zunutze macht, umgesetzt wird. Astrée wurde ausgewählt, da es ein vollständiges und korrektes Analyseergebnis liefert. Gleichzeitig ist die Präzision der abstrakten Interpretation von Astrée hoch, während die Analyselaufzeiten den Einsatz in einem iterativen Entwicklungsprozess erlauben [Our15]. Im Gegensatz zu Frama-C sind in Astrée keine umfassenden Code-Annotationen notwendig, um eine Analyse durchzuführen.

Für die Erkennung der oben genannten Fehlertypen wird ein Prozesssystem[3] aus der Betriebssystemkonfiguration erstellt. Astrée ist in der Lage ARINC-Prozesse, POSIX-Fäden und OSEK/AUTOSAR Arbeitsaufträge zu modellieren und aus den jeweiligen Konfigurationsformaten auszulesen. Ein Astrée-Prozess wird durch seinen Laufstatus (gestoppt, wartend, inaktiv, verdrängt, laufend oder lauffähig) und seine Priorität charakterisiert. Zusätzlich können Synchronisationsmechanismen im Astrée-Prozessmodell berücksichtigt werden [Ast].

Die statische Analyse erfolgt in zwei Phasen: In der sequentiellen Phase wird die Programmsemantik rein sequentiell durchlaufen, von einem festgelegten Einstiegspunkt bis zu dem Endpunkt der Einstiegsfunktion. Hierbei werden durch einen BS-Stub vorhandene Astrée-Prozesse in der Astrée Analyse registriert. Ein Stub ist eine Platzhalterfunktion mit dem gleichen Namen und der gleichen Schnittstelle wie die ersetzte Funktion. In den Stub-Funktionen kann die Funktionalität der ersetzten Funktion vereinfacht umgesetzt werden oder zu Testzwecken nur die Interaktion mit der Funktion betrachtet werden [DW04]. In der zweiten Phase werden die Astrée-Prozesse, in der während der sequentiellen Phase festgestellten Reihenfolge, parallel analysiert, um so die Interaktionen auf den globalen Speicherbereich zwischen den verschiedenen Astrée-Prozessen zu analysieren [Ast]. Diese Analysephase erfolgt iterativ bis die Ergebnisse konvergieren. Zwischen den Iterationsschritten wird die Präzision der Analyse verringert. Der Ablauf wird gemäß den Prioritäten der Astrée-Prozesse durchlaufen und folgt dem Grundsatz, dass der einzig laufende Astrée-Prozess derjenige lauffähige Astrée-Prozess mit der höchsten Priorität ist [Ast]. Astrée ist aber auch durch eine Echtzeitablaufplanung in der Lage, Prozesse mit gleichen Prioritäten einzulasten. Eine genaue Beschreibung der Ablaufplanung während der parallelen Phase kann in [Ast] gefunden werden.

Durch die Analyse des Zugriffsverhaltens auf Variablen durch die Astrée-Prozesse sowie die Berücksichtigung von Synchronisationsmechanismen kann das Vorhandensein von Wettlaufsituationen festgestellt werden. Die Ergebnisse der Nebenläufigkeitsanalyse durch Astrée sind ebenfalls korrekt und vollständig.

2.4 Resümee

In diesem Kapitel wurden Interferenzquellen in Recheneinheiten beschrieben sowie die grundlegende Terminologie der Ablaufplanung erläutert. Des Weiteren wurden hardware- und softwarebasierte Maßnahmen zur Herstellung von räumlicher Rückwirkungsfreiheit vorgestellt.

Je nach Anforderungen der funktionalen Sicherheit müssen in Fahrzeugsteuergeräten Maßnahmen zur Sicherstellung der Rückwirkungsfreiheit getroffen werden. Im voranstehenden Kapitel wurden verschiedene Maßnahmen zur Herstellung räumlicher Rückwirkungsfreiheit vorgestellt. Diese zeigen, dass Rückwirkungsfreiheit nicht nur auf Ebene der Implementierung, sondern auch

[3] Im Kontext vom Astrée ist ein Prozess ein Modell für einen Kontrollfluss auf BS-Ebene, aber auch zur Modellierung von asynchron ausgeführten Kontrollflüssen geeignet [Ast].

bereits beim Entwurf der Systemarchitektur und der Auswahl der Zielplattform berücksichtigt werden müssen. Hierbei dürfen die Forderung nach Kostenreduzierung und Laufzeiteffizienz jedoch nicht außer acht gelassen werden. Maßnahmen zur Sicherstellung von räumlicher Rückwirkungsfreiheit können sich negativ auf die Laufzeiteffizienz auswirken. Sowohl die Entwicklung von softwarebasierten als auch der Einsatz hardwarebasierter Maßnahmen sind mit zusätzlichen Entwicklungskosten verbunden. Des Weiteren ist es im Rahmen der Entwicklung von eingebetteten Systemen in der Automobildomäne nicht möglich, jeden Ansatz zu nutzen. Während mit hardwarebasiertem Speicherschutz die Absicherung von Funktionskomponenten möglich ist, kann die räumliche Rückwirkungsfreiheit auf Ebene einzelner Daten nur schwer und verbunden mit hohen Ressourcenkosten umgesetzt werden. Dies ist hingegen durch die Schaffung von Typ- und Speichersicherheit möglich. Auf Grund hoher Mengen an Altcode im Automobilbereich ist die Verwendung von typ- und speichersicheren Programmiersprachen, wie in Kapitel 2.3.2.1 und 2.3.2.2 beschrieben, jedoch nicht ohne großen Entwicklungsaufwand und damit Kostenaufwand möglich. Die Nutzung statischer Analyse erlaubt es auf Anwendungsbasis, Typ- und Speichersicherheit auch bei typschwachen Sprachen, wie C, die in der Automobilindustrie weit verbreitet ist, sicherzustellen. Dies stellt jedoch keine Sicherheit gegenüber umweltbedingen transienten Fehlern oder Fehlern während des Übersetzungsprozesses dar. Außerdem kann durch Typ- und Speichersicherheit keine logische Trennung von zusammengehörogen Daten und deren Zuweisung zu den Funktionspartitionen erfolgen. Aus diesem Grund ist es sinnvoll einen kombinierten Ansatz aus hardware- und softwarebasiertem Speicherschutz zur Herstellung von Rückwirkungsfreiheit einzusetzen. Dies erlaubt es, durch die Speicherschutzregionen der MPU die Daten der einzelnen Funktionseinheiten untereinander zu isolieren beziehungsweise ermöglicht es, definierte Austauschbereiche zwischen diesen festzulegen.

3 KONZEPT

In diesem Kapitel wird das Konzept eines ganzheitlichen Entwicklungsprozesses für eingebettete Systeme in Fahrzeugsteuergeräten vorgestellt. In Abbildung 3.1 ist dieser Prozess dargestellt. Im Folgenden wird dieses Gesamtkonzept erläutert. Hierzu wird zuerst die Ausgangslage, mit den bereits vorhandenen Werkzeugen und deren Ein- und Ausgangsinformationen beschrieben. Anschließend werden Erweiterungen im Rahmen des ARAMiS II Forschungsprojektes vorgestellt, die eine Verknüpfung dieser Werkzeuge ermöglichen. Im letzten Schritt des Entwurfsprozesses werden die Ergebnisse der vorhergehenden Schritte zusammengeführt. Zu diesem Zweck wird eine erweiterte Nutzung der statischen Analyseergebnisse und die darauf aufbauende Variablenklassifizierung beschrieben. Es werden Möglichkeiten aufgezeigt, wie Einfluss auf die Code-Generierung mit TargetLink (TL) genommen werden kann. Abschließend wird die Zusammenführung der Werkzeuge, wie sie in Abbildung 3.1 dargestellt ist, erklärt und die Vorteile des Prozesskonzeptes vorgestellt.

3.1 Ausgangslage

Im Folgenden werden kurz die bereits vorhandenen Werkzeuge, welche im Entwicklungsprozess eingesetzt werden, mit ihren Ein- und Ausgangsinformationen beschrieben.

3.1.1 ASSIST

Architecture Synthesis for Safety-Critical Systems Tool (ASSIST) [Hil18] ist ein Werkzeug zur Generierung von Allokationen. Basis hierzu ist zum einen ein Hardwaremodell, das die Zielplattform auf der die Anwendung implementiert werden soll, beschreibt. Zum anderen werden die Funktionseinheiten mit den Anforderungen aus der Systemspezifikation und der funktionalen Architektur modelliert. Aus der Kombination des Hardwaremodells und der Funktionseinheiten wird ein Bedingungserfüllungsproblem abgeleitet, welches von ASSIST gelöst wird, um mögliche Allokationen der Funktionseinheiten auf die Zielplattform zu generieren [HB18]. In einem nachgelagerten Schritt nutzt ASSIST diese Allokationen, um auf Basis der temporalen Eigenschaften der Funktionseinheiten einen statischen Ablaufplan des Systems zu generieren [Hil18]. Ergebnis ist eine Konfiguration des Systems.

3.1.2 Astrée

Astrée [Abs] ist ein Werkzeug zur statischen Code-Analyse durch abstrakte Interpretation (siehe Kapitel 2.3.3.1). Astrée wird im Entwicklungsprozess dazu eingesetzt, die Abwesenheit von Laufzeitfehlern nachzuweisen, wie in Kapitel 2.3.3.3 beschrieben. Hierzu verwendet Astrée zum einen den

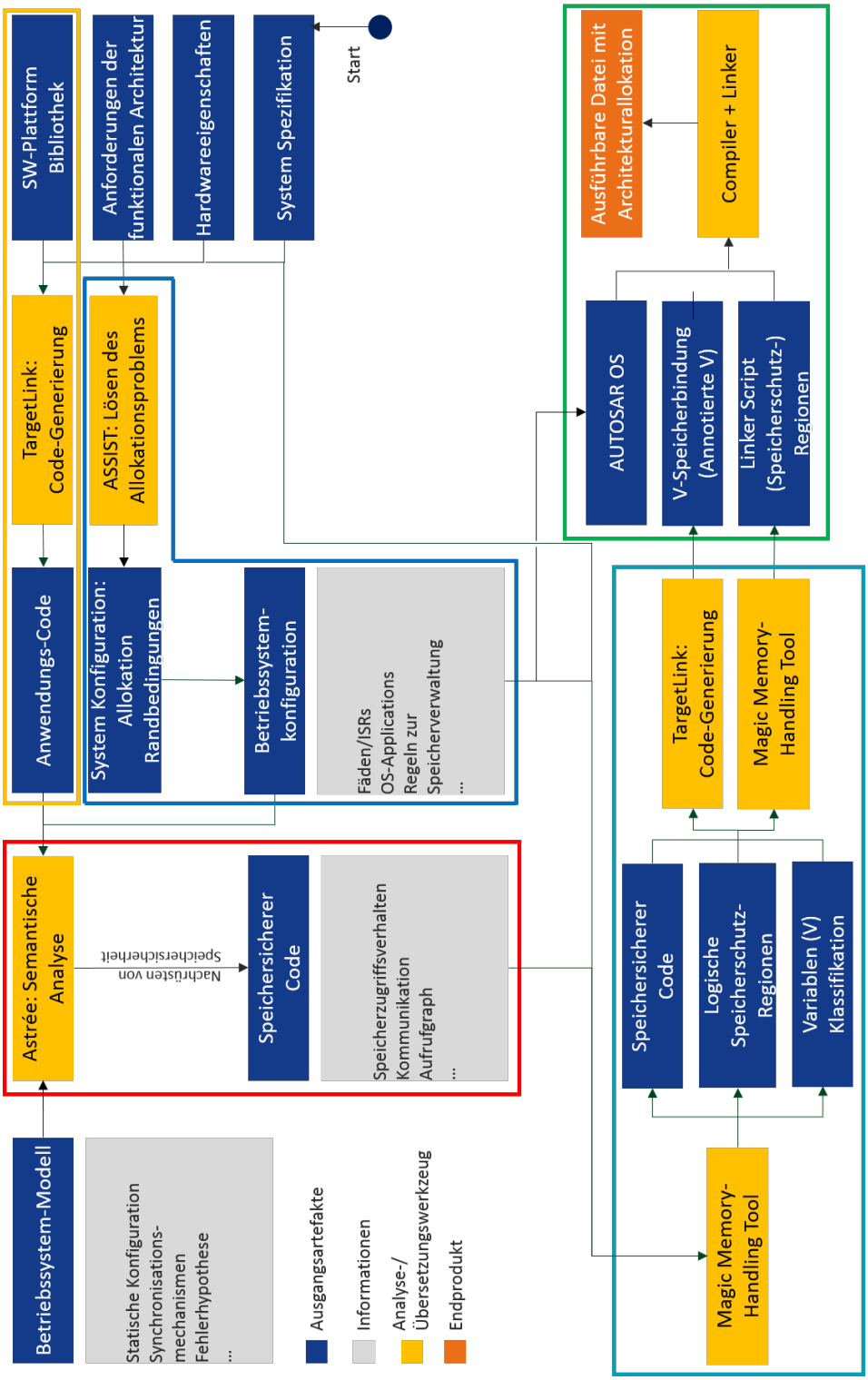


Abbildung 3.1 – Konzept eines das Gesamtsystem berücksichtigenden Entwurfsprozesses für Mehrkernprozessoren [Sti17].

Anwendungscode und zum anderen eine AUTOSAR OS Konfiguration. Letzteres erlaubt es Astrée, wie in Kapitel 2.3.3.4 beschrieben, Nebenläufigkeitsfehler zu detektieren. Die von Astrée gemeldeten Laufzeitfehler werden entweder behoben oder annotiert, wenn es sich um Fehlalarme handelt. Ergebnis der statischen Analyse ist Anwendungscode, der frei von den detektieren Laufzeitfehlern ist. Gleichzeitig stellt eine fehlerfreie Analyse sicher, dass das Programm typ- und speichersicher ist (siehe Kapitel 2.3.3.3).

3.1.3 Code-Generierung

Regelungstechnische Algorithmen werden für den Einsatz in Fahrzeugsteuergeräten modellbasiert in einer DSL, wie zum Beispiel Matlab/Simulink [Thec], entworfen. Diese wird durch einen Code-Generator wie TargetLink (TL) in die Anwendungssprache C übersetzt. Ausgangspunkt hierfür ist ein Modell des Regelalgorithmus, das in C-Bibliotheken übersetzt wird, die in den Anwendungscode, bestehend aus Basissoftware und Betriebssystem, integriert werden. Die Entwicklung dieser Modelle kann auch generisch erfolgen, um die Wiederverwertbarkeit der Entwicklungsergebnisse zu erhöhen. Ist dies der Fall wird der generierte Code im Zuge der Integration nach den Anforderungen der funktionalen Architektur parametriert.

3.2 Konzeption eines ganzheitlichen Entwicklungsprozess

Um einen durchgängigen, ganzheitlichen Entwicklungsprozess zu ermöglichen, ist es notwendig die verschiedenen Werkzeuge miteinander zu verknüpfen. Hierzu wurden im Rahmen des ARAMiS II Forschungsprojektes Erweiterungen an ASSIST und Astrée vorgenommen, die dies erlauben.

Die durch ASSIST gewonnene Allokation und der Ablaufplan sollen in den nachgelagerten Schritten des Entwicklungsprozesses weiterverwendet werden. Um diese in anderen Werkzeugen einzusetzen wird das ARXML-Format von AUTOSAR OS als Austauschformat verwendet. Hierzu wurde ASSIST im Rahmen des ARAMiS II Forschungsprojekts dahingehend erweitert, dass die generierte Allokation von Funktionseinheiten und der daraus resultierende Ablaufplan als AUTOSAR OS Konfiguration exportiert werden können.

Astrée wurde im Rahmen von ARAMiS II dahingehend erweitert, dass es eine solche Konfiguration auslesen kann und diese als Basis für sein internes Betriebssystemmodell (siehe Kapitel 2.3.3.4) nutzen kann. Hierdurch kann auf Basis der von ASSIST bestimmten Systemkonfiguration die statische Analyse des integrierten Anwendungscodes erfolgen.

Ergebnis dieses Integrationsschritts ist die Möglichkeit, eine typ- und speichersichere Anwendung inklusive eines Laufzeitsystems zu entwickeln. Diese berücksichtigt bei der Bindung der Anwendung und ihrer Daten jedoch nicht die Gegebenheiten der Hardware und des Laufzeitverhaltens der Anwendung. Diese Lücke wird durch das MMHT gefüllt. Dieses klassifiziert Anwendungsdaten auf Basis des Laufzeitverhaltens der Anwendung und der Hardwarearchitektur der Zielplattform. Diese Klassifizierung kann anschließend zum einen verwendet werden, Daten effizient auf der Zielplattform zu binden und zum anderen räumliche Rückwirkungsfreiheit durch den Einsatz von hardwarebasiertem Speicherschutz zu unterstützen.

3.3 Instrumentierung der Datenflussanalyse

Zur Berücksichtung des Laufzeitverhaltens der Anwendung durch die Bindung müssen Informationen über das Laufzeitverhalten der Anwendung vorhanden sein. Hierzu kann die Datenflussanalyse von Astrée genutzt werden: Um die in Kapitel 2.3.3.4 beschriebenen Fehlertypen zu finden, führt Astrée eine Datenflussanalyse durch, deren Ergebnisse auch nach der Analyse zur Verfügung stehen (siehe Abbildung 3.2). Bei dieser Analyse werden globale Daten und Daten mit statischer Lebenszeit erfasst. Im Folgenden werden die ablesbaren Ergebnisse dieser Datenflussanalyse kurz beschrieben. Wie in Abbildung 3.2 dargestellt, werden Daten durch ihr Symbol identifiziert. Es wird erfasst, aus

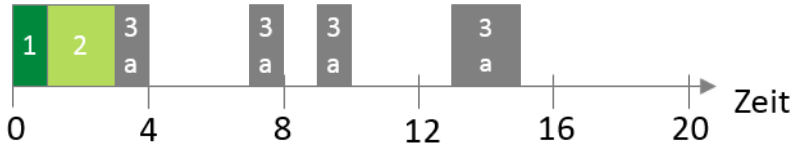
	A	B	C	D	E	F	G
1	1.1. Data flow						
2	Variable	Function	Access	Process	Data races	Shared variable	Location
3	about_x_axis__DiscreteTimeIntegrator	Sa2_controller_about__X_axis	read	Process 3: T1_Controllers	no	no	AttitudeController.c:583.42-78
4	about_x_axis__DiscreteTimeIntegrator	Sa2_controller_about__X_axis	read	Process 3: T1_Controllers	no	no	AttitudeController.c:595.3-39
5	about_x_axis__DiscreteTimeIntegrator	Sa2_controller_about__X_axis	write	Process 3: T1_Controllers	no	no	AttitudeController.c:595.3-39
6	about_y_axis__DiscreteTimeIntegrator	Sa3_controller_about_Y_axis	read	Process 3: T1_Controllers	no	no	AttitudeController.c:701.42-78
7	about_y_axis__DiscreteTimeIntegrator	Sa3_controller_about_Y_axis	read	Process 3: T1_Controllers	no	no	AttitudeController.c:713.3-39
8	about_y_axis__DiscreteTimeIntegrator	Sa3_controller_about_Y_axis	write	Process 3: T1_Controllers	no	no	AttitudeController.c:713.3-39
9	DiscreteIntegrator_wrapper	TASK_T1_Controllers	read	Process 3: T1_Controllers	no	no	copter_var2_4.c:1937.31-57
10	DiscreteIntegrator_wrapper	TASK_T1_Controllers	read	Process 3: T1_Controllers	no	no	copter_var2_4.c:1938.4-30
11	DiscreteIntegrator_wrapper	TASK_T1_Controllers	write	Process 3: T1_Controllers	no	no	copter_var2_4.c:1938.4-30
12	DiscreteTimeIntegrator	STEP_HeightObserver	read	Process 6: T4_HeightObserver	no	no	HeightObserver.c:777.33-55
13	DiscreteTimeIntegrator	STEP_HeightObserver	read	Process 6: T4_HeightObserver	no	no	HeightObserver.c:786.43-65

Abbildung 3.2 – Berichtexport der Datenflussanalyse aus Astrée.

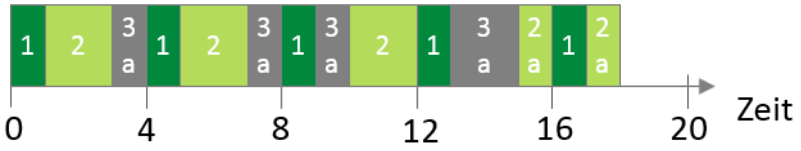
welcher C-Funktion auf ein Datum zugegriffen wird und ob ein lesender oder schreibender Zugriff erfolgt. Jedem Zugriff ist neben der C-Funktion auch noch der Astrée-Prozess zugeordnet, innerhalb dessen die Funktion ausgeführt wird und aus dem der Zugriff erfolgt. Des Weiteren wird erfasst, ob durch einen Zugriff eine Wettlaufsituation entstehen kann und ob es sich bei dem Datum um eine geteilte Variable handelt. Dies bedeutet, dass mehr als ein Kontrollfluss auf das Datum zugreift. Zusätzlich wird noch die Position des Zugriffs im Quellcode angegeben. Die Häufigkeit, mit der innerhalb eines Astrée-Durchlaufs der auf ein Datum zugegriffen wird, lässt sich aus der Anzahl der Einzelzugriffe auf ein Datum bestimmen. Hierbei muss jedoch beachtet werden, dass Astrée Schleifen nur bis zu einem festgelegten Wert ausrollt, beziehungsweise Rekursionen nur bis zu einer festgesetzten Tiefe verfolgt. Dies muss bereits in der Code-Analyse durch geeignete Annotationen berücksichtigt werden.

Ergebnis der Analyse durch Astrée ist also neben dem typ- und speichersicherem Anwendungscode (siehe Kapitel 2.3.3.3) auch eine Analyse des Zugriffsverhaltens auf die Daten der Anwendung. Diese kann im Schritt der Variablenklassifizierung durch das MMHT genutzt werden, indem die Analyse als Bericht aus Astrée ausgeleitet wird.

Die Datenflussanalyse bildet jedoch nicht das korrekte zeitliche Zugriffsverhalten ab, da diese nicht die tatsächliche Ablauffrequenz der Kontrollflüsse berücksichtigt. Um dies zu verdeutlichen, sind in Abbildung 3.3 ein Ablaufplan aus Sicht von Astrée und der geplante, logische Ablaufplan in einer Hyperperiode dargestellt. Die Einplanungseinheiten stellen periodische Arbeitsaufträge dar, die durch Ziffern gekennzeichnet sind. Wird ein Arbeitsauftrag unterbrochen und seine Ausführung dadurch zeitlich verteilt, werden zusammengehörige Teile des Arbeitsauftrags durch Buchstaben gekennzeichnet. Wie aus der Abbildung ersichtlich, wird jede Einplanungseinheit in Astrée nur einmal aufgerufen, während tatsächlich Einplanungseinheit 1 fünfmal und Einplanungseinheit 2 viermal aufgerufen werden. Zur effizienten Nutzung von Speichern wird die Aufruffrequenz später genutzt, um Daten priorisiert auf kernnahen Speicher zu binden, auf die am häufigsten zugegriffen wird. Der berücksichtigte Ablaufplan kann auch, statt aus der logischen Ablaufplanung (siehe Kapitel 2.2.1), aus einer statischen Ablauf-Analyse beziehungsweise einer Messung stammen.



(a) Astrée Ablaufplan



(b) Geplanter, logischer Ablaufplan

Abbildung 3.3 – Vergleich der Betrachtung eines Ablaufplans durch Astrée und dem tatsächlichen Ablaufplan nach [Liu00].

Durch die Zuordnung von AUTOSAR Alarmen und Zählern zu den Kontrollflüssen kann bestimmt werden, mit welcher Periode ein Kontrollfluss ausgeführt wird. Zur Bestimmung der tatsächlichen Aufruffrequenz kann ein Ablaufplan für eine Hyperperiode des Systems erstellt werden. Hierzu wird eine Reihe von Vereinfachungen getroffen, da nur die Frequenz und nicht die tatsächliche Ausführungsreihenfolge betrachtet wird.

1. Das System wird als planbar angenommen und es existiert ein zulässiger Ablaufplan.
2. Die Ausführungszeit der Kontrollflüsse ist Null.
3. Die Periode eines nicht-periodischen Kontrollflusses ist seine minimale Zwischenankunftszeit.
4. Kontrollflüsse werden nicht blockiert.

Ist das System nicht planbar, existiert ein Fehler auf einer höheren Entwurfsebene, der dort behoben werden muss. Da nicht die Einlastungszeitpunkte, sondern nur die Einlastungsfrequenz und damit die Ausführungsfrequenz eines Kontrollflusses interessant ist, können die Ausführungszeit und Blockaden ignoriert werden, da das System als planbar angenommen wird. Jeder Kontrollfluss wird also mit seiner Periode ausgeführt. Um nicht-periodische Kontrollflüsse wie ISRs zu berücksichtigen, werden diese periodisiert. Hierzu wird vom schlimmsten Ausführungsfall ausgegangen: dass nicht-periodische Kontrollflüsse mit ihrer minimalen Zwischenankunftszeit eingelastet werden.

Durch die Zusammenführung der Ergebnisse der Astrée Datenflussanalyse mit dem zeitlichen Verhalten, das im AUTOSAR Modell abgebildet ist, ist es möglich, die tatsächlichen Zugriffsfrequenzen auf Daten zu bestimmen. Dies ermöglicht eine zugriffsfrequenzorientierte Bindung im Rahmen der Variablenklassifizierung.

3.4 Variablenklassifizierung

Zur Zuweisung von Speicherregionen zu den Daten werden diese nach verschiedenen Kriterien klassifiziert. Ziel ist es, durch die Klassifizierung sinnvolle Speicherschutzregionen aufspannen zu können und gleichzeitig die vorhandenen Ressourcen effizient zu nutzen. Im Folgenden wird auf die

verschiedenen Kriterien (Zugriff, Speicherort und Spezifikation) der Klassifizierung eingegangen. Abschließend wird eine Synthese der Klassifizierungen vorgenommen, die eine Bindung und Isolation von Daten ermöglicht.

In den folgenden Kapiteln wird als Beispiel das Listing 3.1 herangezogen, welches auf einem Prozessor, der in Abbildung 3.4 dargestellt ist, implementiert wurde. Im Beispiel sind die Kontrollflüsse `Thread_1_CPU0` und `Thread_2_CPU0` auf den Prozessor CPU0 gebunden und der Kontrollfluss `Thread_3_CPU1` auf den Prozessor CPU1.

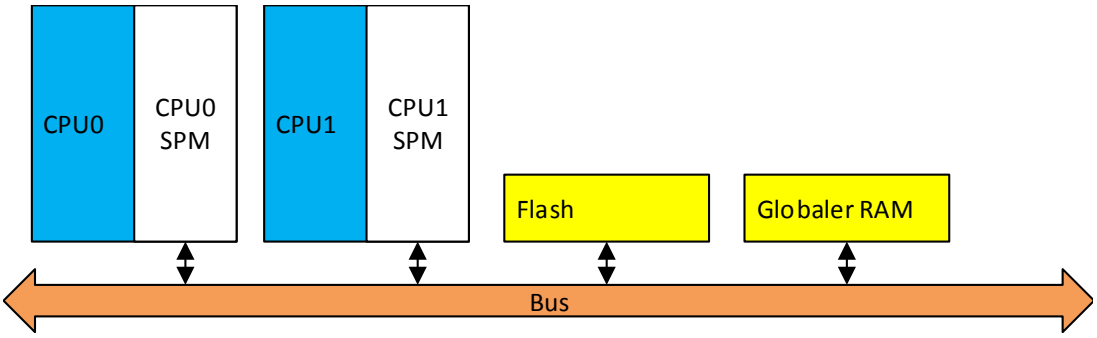


Abbildung 3.4 – Beispiel CPU mit zwei Prozessorkernen, jeweils mit prozessornahen Speichern, einem globalen RAM und einer Flash-Speicherbank.

```
1  int intern_message1 = 1;
2  volatile int message1 = 0;
3  volatile int message2;
4  volatile int message3 = 0;
5
6  Thread_1_CPU0(void){
7      static int t1_var = 0;
8      message1 += intern_message1;
9      // Do stuff
10     t1_var++;
11 }
12
13 Thread_2_CPU0(void){
14     int t2_temp = message1;
15     message2 = 0;
16     // Do Stuff
17     message3 = t2_temp;
18 }
19
20 Thread_3_CPU1(void){
21     const int t3_const = 42;
22     message3 += 1;
23     // Do Stuff
24 }
```

Listing 3.1 – Code-Beispiel zur Variablenklassifizierung.

3.4.1 Klassifizierung nach Zugriff

Auf Basis der Datenflussanalyse kann das Zugriffsverhalten auf ein Datum bestimmt werden. Hierbei fließt ein:

- welche Kontrollflüsse auf das Datum zugreifen,
- wie der Zugriff auf das Datum erfolgt (lesend oder schreibend),
- ob über Prozessorkerngrenzen zugegriffen wird und
- wie häufig die Zugriffe erfolgen.

Aus dem Ort der Deklaration des Datums ergibt sich ein Kontrollfluss, der als Besitzer des Datums verwendet werden kann. Aus dem Zugriffsverhalten folgen die Klassifizierungen:

- Kontrollfluss-lokal, Prozessor-lokal: Dies sind Daten, die nur innerhalb eines Kontrollflusses gelesen oder geschrieben werden.
- Kontrollfluss-global, Prozessor-lokal: Dies sind Daten, auf die durch zwei oder mehrere Kontrollflüsse zugegriffen wird. Alle Kontrollflüsse sind auf den selben Prozessor gebunden.
- Kontrollfluss-global, Prozessor-global: Dies sind Daten, auf die durch zwei oder mehrere Kontrollflüsse zugegriffen wird. Die Kontrollflüsse sind über zwei oder mehr Prozessoren verteilt. Der Zugriff muss also über einen verbindenden Bus erfolgen.

Diese Klassifizierung erlaubt es bereits, die Daten von Kontrollflüssen untereinander abzugrenzen, indem die Speicherschutzregionen nach der Zugriffs-Klassifikation aufgespannt werden. Eine ähnliche Klassifizierung wird auch im Rahmen der in Kapitel 2.3.2.2 vorgestellten C-Sprachanpassungen für Zeiger durchgeführt, die auf bestimmte Speicherregionen begrenzt sind. Während die Beschränkung in Cyclone und CCured auf Datumsebene erfolgt, ermöglicht eine Zugriffsklassifizierung die Eingrenzung auf Kontrollflussebene durch die MPU. Des Weiteren ist bereits bekannt, dass bei Prozessor-globalen Daten die Zugriffszeit vom übertragenden Bus abhängig sind. Variablen der Klasse Kontrollfluss-lokal, Prozessor-lokal können „lokalisiert“ werden, da ihre globale Verfügbarkeit in der Regel nicht sinnvoll ist. Dies bedeutet, dass sie nur auf Funktionsebene sichtbar sind.

Orthogonal ergeben sich weitere Eigenschaften aus dem Zugriffsverhalten: Variablen, auf die nach ihrer Definition nur lesend zugegriffen wird, sind Konstanten. Variablen, auf die nur geschrieben wird oder mit denen außer ihrer Definition nicht interagiert wird, können in der Regel als optimierbar markiert werden. Diese Optimierbarkeit bezieht sich auf die Möglichkeit, das Datum bereits während der Code-Generierung zu entfernen. Eine Ausnahme hierbei bilden Zeiger, über die auf Hardwareregister geschrieben wird. Zur Festellung der Optimierbarkeit muss ein solches Datum genauer betrachtet werden. Tabelle 3.1 zeigt die Zugriff-Klassifizierung der Variablen aus dem Beispiellisting 3.1.

3.4.2 Klassifizierung nach Speicherort

Die Klassifizierung nach dem Speicherort eines Datums wird erst nach der Bindung durch das Datum ersichtlich und erfolgt durch das Treffen von Bindungsentscheidungen automatisch. Die Klassifizierung ergibt sich aus der Speicherarchitektur der Hardware oder kann abstrakt in Speicher-lokal und Speicher-global geteilt werden. Für den Beispielprozessor in Abbildung 3.4 ergeben sich drei mögliche Klassifizierungen:

Tabelle 3.1 – Klassifizierung der Variablen aus Listing 3.1 nach dem Zugriff.

	Konstant	Optimierbar	Lesend-Schreibend
Kontrolfluss-lokal, Prozessor-lokal	`intern_message1` `t3_const`	`message2`	`t1_var` `t2_temp`
Kontrolfluss-global, Prozessor-lokal			`message1`
Kontrolfluss-global, Prozessor-global			`message3`

- Speicher-lokal: Das Datum befindet sich zur Laufzeit im prozessornahen Speicher (CPU0 SPM oder CPU1 SPM).
- RAM-global: Das Datum befindet sich zur Laufzeit im globalen RAM.
- Flash-global: Das Datum wird im Flash-Speicher gehalten.

Verfügt der Prozessor oder die Prozessorkerne zusätzlich über Caches, kann orthogonal noch zwischen Cache-baren und nicht-Cache-baren Daten unterschieden werden.

Die Klassifizierung nach dem Speicherort ermöglicht es festzulegen, an welchem Ort ein Datum gespeichert wird, beziehungsweise aus welchem Speicher es während der Laufzeit geladen wird. Durch eine sinnvolle Klassifizierung der Variablen kann die Laufzeiteffizienz erhöht werden, indem darauf geachtet wird, dass, je nach Hardware, der Speicherort mit der schnelleren Zugriffszeit gewählt wird. Laufzeitanalysen können verbessert werden, indem Daten in Speicher mit gut bestimmbarem Zugriffsverhalten geladen werden oder als nicht-Cache-bar markiert werden.

Im Fall des Beispielprozessors bietet es sich an, so viele Daten wie möglich in prozessornahen Speichern zu platzieren. Bei diesen ist die Zugriffszeit am kürzesten und am vorhersagbarsten. Nur wenn Daten nicht mehr in kernnahen Speicher platzierbar sind, sollten sie in einem der globalen Speicher abgelegt werden. Durch die Priorisierung von kernnahen Speichern wird der Einfluss von Buszugriffszeiten und Verzögerungen durch konkurrierende Zugriffe auf geteilte Ressourcen verringert.

Eine Klasse, die gesondert betrachtet werden muss, sind Variablen, die auf dem Stack abgelegt werden. In einer semantischen Analyse durch Astrée werden diese nicht berücksichtigt. Des Weiteren werden sie nicht explizit gebunden und befinden sich nicht über die gesamte Programmlaufzeit in Datenspeichern. Zur Verfeinerung der räumlichen Isolation könnte mit einer Stack-Analyse die Größe des benötigten Stack-Speichers bestimmt werden. Ausgehend davon können Maßnahmen wie hardwarebasierter Speicherschutz für die Stack-Speicherobjekte umgesetzt werden.

3.4.3 Klassifizierung nach Spezifikation

Aus der Systemspezifikation lassen sich ebenfalls Variablenklassen ableiten. Diese sind meist orthogonal zu anderen Klassifizierungsarten. Ein Beispiel für aus der Spezifikation stammende Klassifizierungen sind Daten, die für Mess- und Kalibriervorgänge verwendet werden. Per Definition sind sie Kontrollfluss-globale Daten, da sie zum einen durch mindestens einen Kontrollfluss der Anwendung verwendet werden und zum anderen durch einen Kalibrier- oder Messkontrollfluss geschrieben beziehungsweise gelesen werden. Da dies allerdings keine intrinsische Eigenschaft des Datums ist, die aus dem Kontext oder der Datenflussanalyse bestimmt werden kann, muss diese Klasse aus der Spezifikation bestimmt werden.

Ähnlich verhält es sich mit Daten, die aus den Anforderungen der funktionalen Sicherheit an bestimmten Speicherorten abgelegt werden müssen. Ein Beispiel hierfür ist ein Datum, das beim

Laden einer CRC- oder ECC-Prüfung unterzogen werden muss. Diese Prüfung kann allerdings nicht von jedem Speicher aus erfolgen, da entsprechende Hardwarebausteine auf dem Speicherbaustein implementiert sein müssen. Auch dies ist nicht aus den Eigenschaften des Datums auf Quellcodeebene ersichtlich, sondern kann nur über die Spezifikation vorgegeben werden. Ein weiteres Beispiel für Variablenklassen, die sich aus Anforderungen der Spezifikation ergeben, sind Daten, die aus Gründen der Partitionierung von Kontrollflüssen nur auf einem bestimmten Speicher platziert werden dürfen, um so Rückwirkungsfreiheit herzustellen. So könnte eine Anforderung sein, dass die Daten von `Thread_1_CPU0` und `Thread_3_CPU1` nicht auf dem selben prozessornahen Speicher abgelegt werden dürfen. Das ermöglicht es, durch die MPU und durch Buszugriffskontrolle zu verhindern, dass beide auf die Daten des jeweils anderen Kontrollflusses schreiben können. Die Klassifizierung wäre dann eine Randbedingung der Form $Speicher_{Thread1} \neq Speicher_{Thread3}$.

3.4.4 Kombination der Klassifizierungen

Um einzelne Daten zu klassifizieren und eine Bindung darauf aufbauend durchzuführen, werden die Klassifikationen zusammengefasst, welche in den Kapiteln 3.4.1 bis 3.4.3 beschrieben werden. Hieraus ergeben sich zwölf Variablenklassen:

1. Prozessor *n*, Kontrollfluss-lokal, Speicher-lokal: Daten[4], auf die nur innerhalb eines Kontrollflusses, der auf einem Prozessorkern *n* gebunden ist, zugegriffen wird und die zur Laufzeit in einem prozessornahen Speicher platziert sind.

2. Prozessor *n*, Kontrollfluss-lokal, Speicher-global: Daten, auf die nur innerhalb eines Kontrollflusses, der auf einem Prozessorkern *n* gebunden ist, zugegriffen wird und die zur Laufzeit auf einem globalen Speicher platziert sind.

3. Prozessor *n*, Kontrollfluss-global, Speicher-lokal: Daten, auf die durch zwei oder mehr Kontrollflüsse zugegriffen wird, die auf dem selben Prozessorkern *n* gebunden sind und in einem prozessornahen Speicher platziert sind.

4. Prozessor *n*, Kontrollfluss-global, Speicher-global: Daten, auf die durch zwei oder mehr Kontrollflüsse zugegriffen wird, die auf dem selben Prozessorkern *n* gebunden sind und in einem globalen Speicher platziert sind.

5. Kontrollfluss-global, Speicher-lokal: Daten, auf die durch zwei oder mehr Kontrollflüsse zugegriffen wird, die auf zwei oder mehr Prozessorkernen gebunden sind und in einem der kernlokalen Speicher platziert sind.

6. Kontrollfluss-global, Speicher-global: Daten, auf die durch zwei oder mehr Kontrollflüsse zugegriffen wird, die auf zwei oder mehr Prozessorkernen gebunden sind und in einem der globalen Speicher platziert sind.

7. Konstanten, Speicher-lokal: Daten, auf die nach ihrer Definition nur noch lesend zugegriffen wird und die aus Softwaresicht konstant sind. Sie werden in einem prozessornahen Speicher platziert.

8. Konstanten, Speicher-global: Daten, auf die nach ihrer Defintion nur noch lesend zugegriffen wird und die aus Softwaresicht konstant sind. Sie werden in einem globalen Speicher platziert.

[4]Wenn nicht anders angegeben, handelt es sich im Rahmen des Kapitels um alle Variablen, die keine echten Konstanten darstellen, das heißt solche, die weder durch Software noch durch Hardware verändert werden.

9. Kalibrierdaten: Daten, die in der Spezifikation für Kalibrierprozesse markiert sind. Sie sind gleichzeitig auch Kontrollfluss-global.
10. Messdaten: Daten, die in der Spezifikation für Messprozesse markiert sind. Sie sind gleichzeitig auch Kontrollfluss-global.
11. Optimierbar: Daten, auf die nach ihrer Deklaration weder durch die Hardware noch durch die Software lesend zugegriffen wird und die deshalb keinen Effekt im Programmablauf haben. Diese Daten können aus dem Programm entfernt werden.
12. Stack-Daten: Daten, die auf Stack-Speichern verwaltet werden.

Platzierungsbedingungen, wie sie in Kapitel 3.4.3 beschrieben werden, müssen über Randbedingungen in die Klassifizierung einfließen und beim tatsächlichen Binden berücksichtigt werden. Dies spiegelt sich in der Zuordnung zu einem Kern und dem verwendeten Speicher wider. Zur Berücksichtigung der Laufzeiteffizienz wird auch die Zugriffsfrequenz, abgeleitet aus dem Abplaufplan, in der Klassifizierung berücksichtigt. Ziel ist es, Daten, auf die häufig zugegriffen wird, möglichst als Speicher-lokale Daten zu klassifizieren. Im Zuge dessen könnte auch eine Aussage darüber getroffen werden, ob Datentypen, wie Konstanten, als Cache-bar markiert werden sollen. Dies wird allerdings in dieser Arbeit noch nicht getan.

Ergebnis der Kombination der vorher aufgeführten Klassen ist eine Menge an Klassen, die sowohl das Zugriffsverhalten der Anwendung als auch die Bindung der Daten auf der Hardware beschreiben. Dies ermöglicht die Generierung von Speicherregionen, indem die gleich klassifizierten Daten eines einzelnen Kontrollflusses in einer Speicherregion zusammen gebunden werden.

3.5 Instrumentierung der Code-Generierung

Wie bereits in Kapitel 1.1.3 beschrieben, werden die Regelungsanwendungen modellbasiert entwickelt und in einer DSL beschrieben. Zur Übersetzung dieser DSL in die Entwicklungssprache C wird ein Code-Generator eingesetzt. Der Code-Generator bietet die Möglichkeit, auf die Umsetzung des Modelles im Code Einfluss zu nehmen und kann dazu genutzt werden, anwendungspezifische Anforderungen zu berücksichtigen. Ziel ist es, die durch Variablenklassifizierung erlangten Speicherschutzregionen als Annotationen in den generierten C-Code einzubringen

Im Rahmen dieser Arbeit wird der TL Code-Generator eingesetzt und im folgenden Abschnitt der prinzipielle Aufbau des Werkzeuges erläutert. Anschließend werden die Möglichkeiten dargestellt, wie auf die Code-Generierung mit TL Einfluss genommen werden kann.

3.5.1 TargetLink

TargetLink ist ein von der Firma dSpace entwickeltes Werkzeug zur Übersetzung von Modellen in Matlab/Simulink in C-Code. Im Folgenden soll darauf eingegangen werden, wie der TL Entwicklungsprozess abläuft, welche für diese Arbeit wichtigen Informationen in der TL Umgebung verwaltet und wo diese abgelegt werden. Was an dieser Stelle nicht betrachtet werden soll, ist der Code-Generierungsprozess an sich.

Modelle eigenen sich gut zur Umsetzung regelungstechnischer Algorithmen, da auch deren Entwurf modellbasiert stattfindet. In Abbildung 3.5 ist der Entwurfs- und Umsetzungsprozess dargestellt. Zu Beginn steht eine Beschreibung eines Streckemodells, das geregelt werden soll, sowie ein zugehöriger

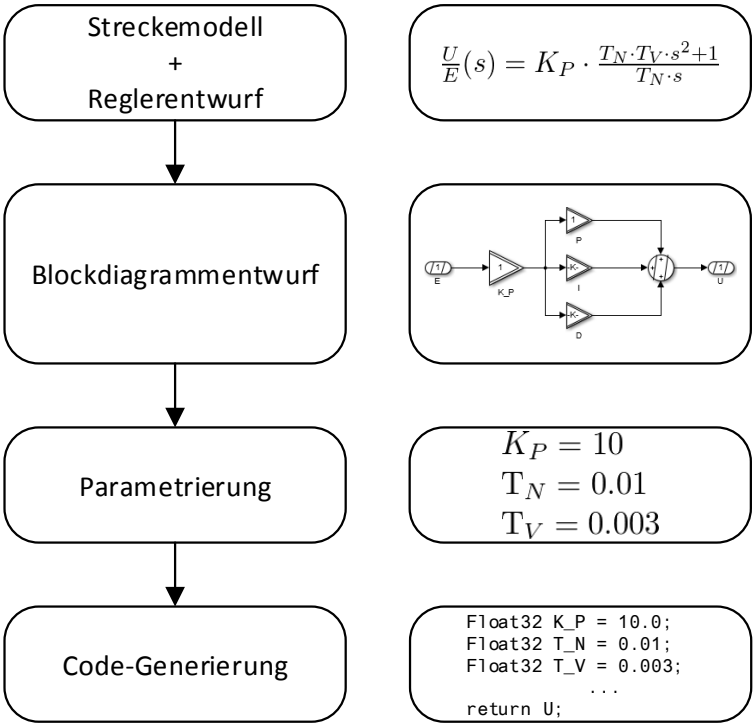


Abbildung 3.5 – Entwurfsprozess in TL mit Beispielen aus den Entwurfsebenen.

Regelungsentwurf. Wie ein Regelungsentwuf zustande kommt, soll an dieser Stelle nicht betrachtet werden; Informationen hierzu können [Föl16] entnommen werden.

Diese Regelentwürfe werden in der graphischen Umgebung von Simulink als Blockdiagramm umgesetzt. Die verwendeten Blöcke stellen mathematische Operationen, logische Operationen oder eine Kombination von mehreren logischen und mathematischen Operationen dar. Das so entstehende Blockschaltbild ist ein Datenflussgraph. Die Verbindungen zwischen den Blöcken bilden die Kanten, die Blöcke die Knoten des Graphs. In der Regelungstechnik sind diese Graphen häufig zyklisch, da eine Rückführung von Werten für die Regelung notwendig ist. Zur Modularisierung von Funktionen, beziehungsweise zur Steigerung der Übersichtlichkeit, werden zusammengehörige Blöcke und ihre Signale zu Subsystemen[5] zusammengefasst. Zur Verwendung von TL werden spezielle Blöcke verwendet, die es erlauben, neben der auszuführenden Funktion auch den aus dem Block generierten Code zu parametrieren. Hierbei bekommen die Subsysteme eine zusätzliche Funktion, indem sie nicht nur als Mittel der Funktionskapselung dienen, sondern gleichzeitig noch Module der Code-Generierung, entweder als Funktion oder als inkludierbare C-Bibliothek, darstellen.

Nach der Abbildung des Regelalgorithmus als TL-Modell und teilweise währenddessen werden die verwendeten Blöcke mit Initialwerten, wie zum Beispiel Faktoren oder Look-Up-Tabellen, parametriert. Dies ist auf drei verschiedene Arten möglich:

- in den individuellen Blockeinstellungen,
- über Matlab Variablen und Skripte oder

[5]Ein Simulink- bzw. TargetLink Subsystem ist ein Behälter für zusammengehörige Blöcke mit festen Ein- und Ausgängen und ist nicht mit dem Subsystem oder der Komponente eines Systems zu verwechseln.

- über das TL Data Dictionary (DD).

Die Parametrierung über die Blockparameter ist, ähnlich wie die Verwendung von magischen Zahlen in Programmen, problematisch, da die Parameter nicht über gleichartige Blöcke geteilt und schnell angepasst werden können. Des Weiteren ist hierdurch kein generischer Entwurf des Modelles möglich, beziehungsweise mit hohem Aufwand während der Einbindung in eine konkrete Anwendung verbunden. Die Verwendung von Matlab Variablen und Skripten, welche wiederum Variablen erzeugen, ist eine gute Alternative zur direkten Parametrierung von Regelungsparametern der Blöcke. Dies liegt darin begründet, dass sich viele Parameter über mathematische Ausdrücke aus den Eigenschaften der Regelstrecke ableiten lassen. Diese Ausdrücke können in Matlab Skripten umgesetzt werden und verringern so den Konfigurationsaufwand für die Einbindung des Modelles in eine konkrete Anwendung.

Zur Anpassung von Parametern der Code-Generierung eignet sich das DD von TL besser. Hierbei handelt es sich um eine Datenbank, die TargetLink Modellinformationen verwaltet. Sie ähnelt in ihrer Funktionsweise einer Header-Datei, da sie auf der einen Seite unabhängig vom referenzierten TL Modell Definitionen für Datentypen und Code Generiungsparameter zur Verfügung stellt. Auf der anderen Seite können individuelle Blöcke durch das DD referenziert werden und ihre Eigenschaften so durch Parameter im DD festgelegt werden. Obwohl Regelungsparameter wie Faktoren von Blöcken im DD festgelegt werden können, ist es nicht möglich, in diesem mathematische Ausdrücke anzugeben. Deshalb ist ein kombinierter Ansatz aus Matlab Skripten und dem DD ein sinnvoller Ansatz, um TL Modelle zu parametrieren.

Nach erfolgter Modellparametrierung kann aus den einzelnen Subsystemen Code generiert werden. Das Ergebnis sind C-Bibliotheken bestehend aus Header-Dateien und C-Dateien. Ein Block wird in eine oder mehrere C-Operationen, in speziellen Fällen auch C-Funktionen, übersetzt. Ein Block kann auch mehrere C-Variablen repräsentieren. Ein Beispiel hierfür ist eine Look-Up-Tabelle, die aus einem Array ihrer Ausgabedaten und einem Array möglicher Eingangsdaten besteht.

3.5.2 Das TargetLink Data Dictionary

Da das DD eine zentrale Rolle in dieser Arbeit erfüllt, soll es im Folgenden näher beschrieben werden. Es stellt eine zentrale Sammlung aller Informationen, abgesehen von der Modellstruktur, die für die Code-Generierung benötigt werden, dar. Hierzu gehören:

- allgemeine Informationen wie der Modellname,
- Variablenrepräsentationen,
- Datentypen,
- Modulinformationen,
- externe Header-Dateien,
- inkludierte DDs,
- Varianteninformationen und
- Spezifikationsdetails.

Diese Informationen werden in einer Baumstuktur abgelegt (siehe Abbildung 3.6) und können über ihren Pfad referenziert werden. Dies vereinfacht die automatisierte Navigation innerhalb des DDs.

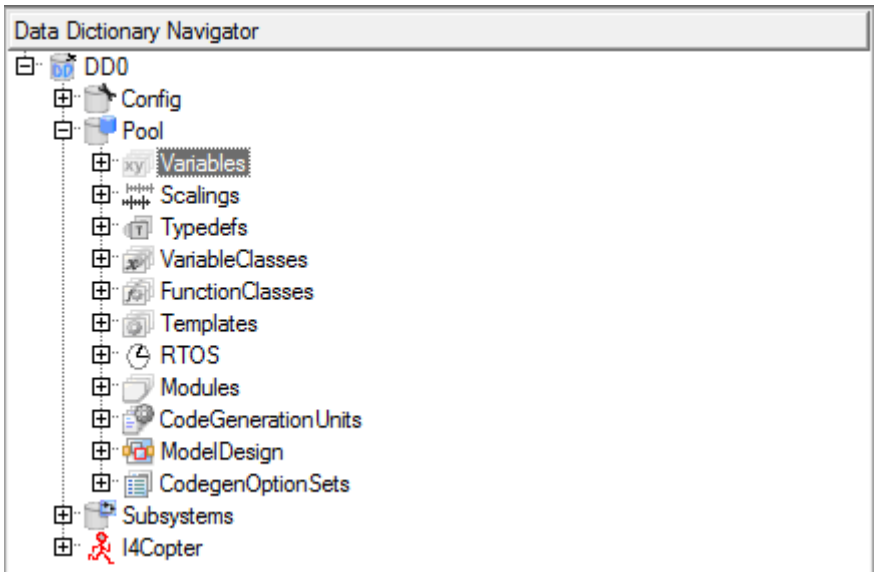


Abbildung 3.6 – Baumstruktur des DD.

Die Möglichkeit weitere DDs zu inkludieren erlaubt es außerdem, häufig genutzte Datentypen oder Variablenarten als Vorlage in mehreren Modellen zu verwenden und diese zentral zu verwalten. Des Weiteren ermöglicht das DD es, Variablen, wie sie im generierten Code auftauchen, individuell anzupassen. Hierzu werden Variablenobjekte angelegt, auf die eine Referenz im Modell erstellt wird. Dies erlaubt es Parameter wie:

- Variablennamen,
- Datentyp,
- Sichtbarkeit und Lebensdauer und eine
- Variablenklasse

festzulegen. Variablenklassen bieten eine Möglichkeit, Variablen zu Gruppen zusammenzufassen. Ein Beispiel hierfür sind spezielle Variablen, die zu Mess- beziehungsweise Kalibrierzwecken anders behandelt werden sollen.

3.5.3 Einflussnahme auf die Code-Generierung

Im Zuge dieser Arbeit soll die Code-Generierung dazu genutzt werden, anwendungspezifisches Wissen in anwendungsagnostische Regelmodelle einzubringen. Auf diese Weise bleibt das verwendete Modell generisch und nur der generierte Code wird an die Anwendung angepasst. Ziel ist es, auf Basis der Variablenklassifizierung, Speicherregionen aufzuspannen. Dazu müssen die Variablen in Klassen zusammengefasst werden und die daraus resultierenden Speicherregion in den C-Code eingebracht werden. Im Folgenden wird beschrieben, wie im Rahmen der Code-Generierung mit TL Informationen in den resultierenden Code integriert werden können. In Abbildung 3.7 sind verschiedene Methoden zur Beeinflussung des generierten Codes dargestellt.

Die Code-Generierung mit TL kann über Matlab Skripte erfolgen, die Parameter im Modell an sich sowie die dort gespeicherten TL Optionen anpassen können. Dies hat den Nachteil, dass Variablen

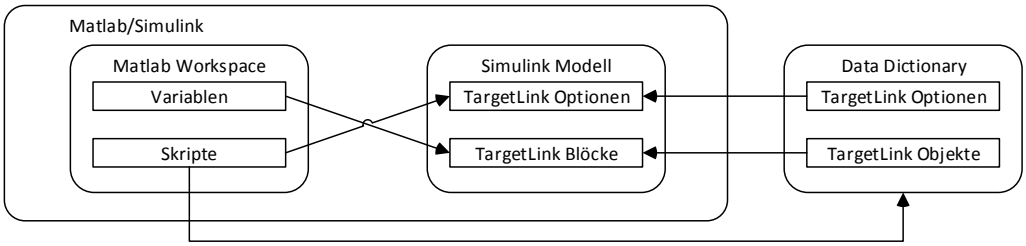


Abbildung 3.7 – Möglichkeiten der Einflussnahme auf die Code-Generierung.

im C-Code nicht ohne großen Aufwand eindeutig den Blöcken zugeordnet werden können, die sie repräsentieren. Wie in Kapitel 3.5.1 beschrieben, kann ein Block auch zu mehreren Variablen übersetzt werden. Des Weiteren muss die Benennung von Variablen im C-Code nicht mit der im Blockdiagramm übereinstimmen. Durch die Aufteilung des Modells in Subsysteme wird die Suche nach der korrekten Repräsentation zusätzlich erschwert. Ein weiteres Problem ist, dass die Sprache Matlab vor allem auf die mathematische Manipulation von Daten ausgelegt ist. Eine effiziente und simple Umsetzung der automatisierten Variablenklassifizierung ist in Matlab nur schwer möglich, weshalb es Sinn macht, diese in einer allgemein nutzbaren Sprache durchzuführen. Ebenso ist es nicht sinnvoll, die Beeinflussung der Code-Generierung und die automatisierte Variablenklassifikation in zwei unterschiedlichen Programmiersprachen umzusetzen, wenn es sich vermeiden lässt.

Die zweite Möglichkeit ist die direkte Beeinflussung des Modells durch TL- und Modellfunktionen. Ähnlich wie die Beeinflussung durch Matlab Skripte stellt sich das Problem der Zuordnung von C-Variablen zu ihren Blockrepräsentationen. Des Weiteren sind die Möglichkeiten zur Interaktion mit dem Modell eingeschränkt und es lassen sich nicht alle Parameter im vollem Umfang beinflussen. Ein weiterer Nachteil ist, dass hierdurch das Modell mit anwendungsspezifischem Wissen verändert wird und somit der generische Entwicklungsansatz verloren geht.

Die letzte Möglichkeit ist die Parametrierung des Modells durch das DD. Dieses ist über die TL Matlab API vollständig anpassbar. Des Weiteren stellen die Variablenobjekte des DD eine eindeutige Repräsentation von Variablen im C-Code dar und bilden somit die einfachste Möglichkeit, die Code-Generierung für Variablen zu beeinflussen. Das Data Dictonary ist außerdem gekapselt vom eigentlichen Modell und nur die Referenzen müssen in das Modell eingebracht werden. Der generische Entwurf des Modells bleibt erhalten. Die Fähigkeit, Variablen durch Variablenklassen zusammenzufassen und so Einfluss auf ihre Eigenschaften zu nehmen, ist des Weiteren eine gute Möglichkeit, die Ergebnisse der Variablenklassifizierung in die Code-Generierung einfließen zu lassen.

3.6 Integration der Werkzeugkette

Ziel des Konzeptes ist es, einen durch alle Entwicklungsebenen durchgängigen Entwurfsprozess für Fahrzeugsteuergeräte darzustellen. Um dies zu ermöglichen sollen die verwendeten Werkzeuge wie in Abbildung 3.1 dargestellt miteinander verknüpft werden. Dies wird durch die, in den Kapiteln 3.2 bis 3.5 beschrieben, Maßnahmen ermöglicht.

Ausgangspunkt des Entwicklungsprozesses ist die Spezifikation des Systems, die funktionale Architektur, die Zielplattform und regelungstechnische Modelle aus einer Bibliotheksplattform. Wie in Kapitel 3.2 beschrieben, wird aus diesen Informationen eine AUTOSAR OS Konfiguration erzeugt, die die Allokation und den Ablaufplan der Anwendung widerspiegelt (blauer Kasten in Abbildung 3.1). Aus den Systemanforderungen und der Bibliotheksplattform wird, unter zu Hilfenahme von TargetLink Anwendungscode generiert (gelber Kasten in Abbildung 3.1). Die Betriebssystemkonfiguration kann durch Astrée eingelesen werden und in Kombination mit dem Anwendungscode aus der Bibliotheksplattform einer Analyse unterzogen werden. Ergebnis der Analyse ist sowohl der typ- und speichersichere Anwendungscode, als auch eine Datenflussanalyse, die das Zugriffsverhalten der Anwendung im AUTOSAR Laufzeitsystem beschreibt (roter Kasten in Abbildung 3.1). Diese Informationen, also die Betriebsystemkonfiguration, die Systemspezifikation, die typen- und speichersichere Anwendung und das Zugriffsverhalten der Anwendung werden vom MMHT in der in Kapitel 3.4.4 beschriebenen Klassifizierung von Daten zusammengeführt. Ausgehend von der Klassifizieriung wird für die Daten eine Bindung auf Speicher erzeugt, aus der sich logische Speicherschutz-Regionen ableiten lassen. Diese werden durch das MMHT in die Code Generierung durch TargetLink eingebracht (türkiser Kasten in Abbildung 3.1).

Endergebnis des gesamten Entwicklungsprozesses ist ein konfiguriertes AUTOSAR OS, eine mit Speicherregionen annotierte, typ- und speichersichere Anwendung und ein Linkerskript, dass diese Speicherregionen widerspiegelt. Der Compiler und der Linker fügen im letzten Schritt diese Bausteine zu einer ausführbaren Anwendung zusammen.

3.7 Resümee

Der in diesem Kapitel vorgestellte Entwicklungsprozess ermöglicht eine durchgängige Nutzung von Werkzeugen. Durch die gemeinsame Nutzung von Austauschformaten wie dem ARXML-Format von AUTOSAR können Informationen aus einem Entwicklungsschritt in den nächsten überführt werden. Dies erlaubt es den Werkzeugen, wie Astrée und dem MMHT, weitere Funktionen umzusetzen, wie die Erkennung von Nebenläufigkeitsfehlern beziehungsweise die korrekte Bestimmung der Zugriffsfrequenz. Die Variablenklassifizierung ermöglicht es, das Laufzeitverhalten, welches aus Implementierungs- und Architekturentscheidungen resultiert, zu nutzen, um automatisiert Bindungsentscheidungen zu treffen und diese in den generierten Code einzubringen. Gleichzeitig erlauben diese Bindungsentscheidungen die Aufspannung von Speicherschutzregionen auf Basis des Datenzugriffsverhalten der Anwendung.

4 UMSETZUNG

Im Folgenden wird die Implementierung des MMHT als Werkzeug zur automatischen Klassifizierung von Variablen und dem Aufspannen von Speicherschutzregionen in einer AUTOSAR-Umgebung beschrieben. Dieses Werkzeug setzt die in den Kapitel 3.4 und 3.5 vorgestellten Konzepte um. Die Gegebenheiten der Hardware müssen bei der Klassifizierung von Variablen berücksichtigt werden. Dementsprechend richtet sich die Umsetzung eines Regelwerks nach der Zielplattform der Anwendung. Im Rahmen dieser Arbeit wird eine Anwendung auf einem Mehrkernprozessor der Infineon Aurix TriCore Familie implementiert, weshalb der Aufbau eines Derivates dieser Familie vorgestellt wird. Es wird außerdem ein Durchlauf der Astrée Analyse beschrieben. Anschließend wird auf die, auf der Analyse aufbauende, Implementierung des MMHT eingegangen, welches die Variablenklassifizierung vornimmt und mit TL interagiert, um die Code-Generierung zu beeinflussen.

Die Skripte, die im Rahmen dieser Arbeit entstanden sind und das MMHT können in [Brä18d] eingesehen werden.

4.1 Aurix TC2xx Prozessoren

Die Klassifizierung von Variablen kann nur erfolgen, wenn die Gegebenheiten der Hardware bekannt sind, auf der die Anwendung implementiert wird. Im Rahmen dieser Arbeit soll die Anwendung beispielhaft auf einem Mehrkernprozessor der Infineon Aurix TC2xx Familie implementiert werden. Im Folgenden wird der für die Arbeit relevante Aufbau eines TC277 erläutert.

In Abbildung 4.1 ist das Blockschaltbild eines TC277 ohne Peripheriegeräte dargestellt. Der Prozessor verfügt über drei Prozessorkerne in einer 32-bit TriCore1.6 Architektur, die in Abbildung 4.1 in blau dargestellt sind. Jedem Kern ist eine Programmspeicherschnittstelle (Program Memory Interface) (PMI) und eine Datenspeicherschnittstelle (Data Memory Interface) (DMI) zugeordnet. Diese dienen zum einen als Schnittstelle des Prozessorkerns zu den zwei Bussen und zum anderen verwalten sie den prozessornahen Programm- (PSPR) beziehungsweise Datenspeicher (DSPR) in SRAM-Technologie und die zugehörigen Caches. Es sind zwei Prozessorkernvarianten implementiert: zwei Performance-Kerne (TC1.6P) und ein Efficiency-Kern (TC1.6E). Die Unterschiede zwischen den beiden Varianten sind in Tabelle 4.1 dargestellt. Jeweils ein Performance- und ein Efficiency-Kern sind mit einem zusätzlichen Gleichtakt-Prozessor (Lockstep in Abbildung 4.1) ausgestattet. Neben den prozessornahen Speichern verfügt der Aurix TC277 über einen globalen RAM mit 32 kB in der lokalen Speichereinheit (Local Memory Unit) (LMU) in SRAM-Technologie sowie über zwei Flash-Speicherbänke in der Programmspeichereinheit (Program Memory Unit) (PMU). Der D-Flash mit 384 kB dient der Ablage von Programmdaten, während der P-Flash mit 4 MB die Programminstruktionen enthält [Mic18]. Alle Zugriffe der Prozessorkerne auf die Speicher anderer

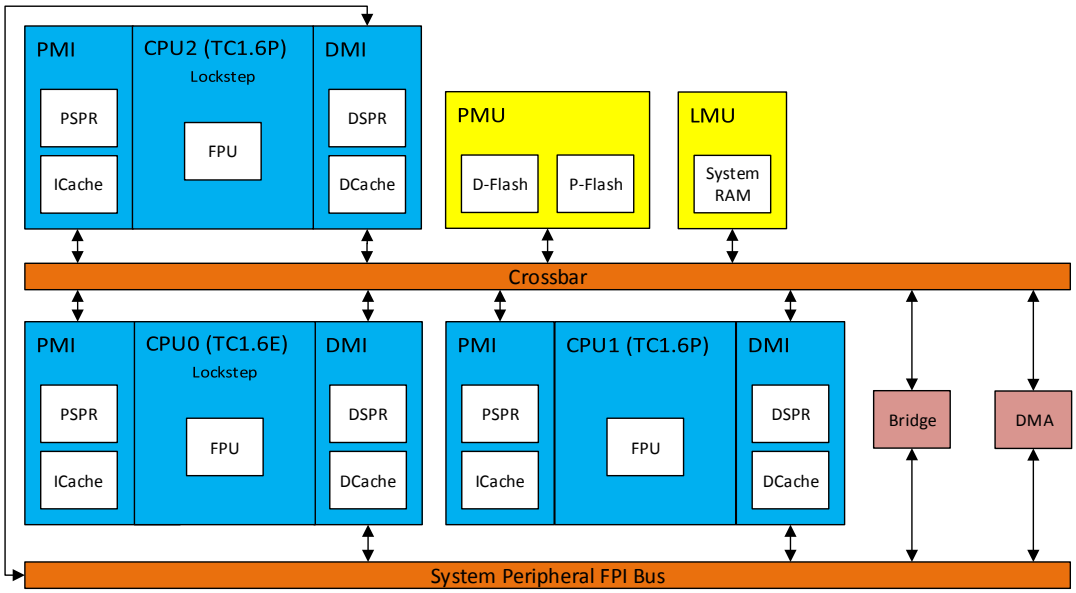


Abbildung 4.1 – Blockschaltbild eines Infineon Aurix TC277 ohne Peripheriegeräte nach [Tria].

Tabelle 4.1 – Gegenüberstellung eines Performance- und eines Efficiency-Kerns nach [Mic18].

Eigenschaft	Performance	Efficiency
max. Taktfrequenz	200 MHz	200 MHz
PSPR	32 kB	24 kB
I-Cache	16 kB	8 kB
DSPR	120 kB	112 kB
D-Cache	8 kB	128 B
Pipeline	4-stufig	6-stufig

Kerne oder die globalen Speicher erfolgt über die Crossbar. Hierbei handelt es sich um ein Bus-System, das auch gleichzeitige Übertragungen zwischen verschiedenen Bus-Teilnehmern ermöglicht, um so die Durchsatzrate des Busses zu erhöhen [Tria]. In Tabelle 4.2 sind die Zugriffszeiten auf die verschiedenen Speicher als inaktive Taktzyklen der Prozessoren angegeben. Bei inaktiven Taktzyklen wartet die CPU auf ein Datum, um mit der Ausführung fortzufahren. In der Praxis sind diese Zeiten im Durchschnitt auf Grund der Instruktionspipeline geringer [Tria]. Aus der Tabelle ist ersichtlich, dass aus Zugriffen, die über die Crossbar erfolgen, mindestens fünf inaktive Taktzyklen folgen. Ein Vergleich eines lesenden Zugriffs auf den eigenen DSPR (60 ns) und eines lesenden Zugriffs auf den globalen Daten-Flash (200 ns) zeigt, dass ein Lesezugriff auf den globalen Flash 3,3-mal so lange dauert wie ein Lesezugriff auf den DSPR [Keß18]. Schreibzugriffe auf den Flash (65 µs [Keß18]) dauern auf Grund des Aufbaus des Flash-Speichers und seiner Ansteuerung [Tria] wesentlich länger. Ein Schreibzugriff auf den DSPR hingegen benötigt nur 40 ns [Keß18]. Daraus lässt sich schließen, dass für eine gute Laufzeiteffizienz Zugriffe über den Bus und auf geteilte Speicher vermieden werden sollten. Stattdessen sollten möglichst die prozessornahen DSRP-Speicher genutzt werden. Dies wird in der Klassifizierung der Daten berücksichtigt, indem zuerst eine Bindung der Daten auf den DSPR-Speichern versucht wird.

Tabelle 4.2 – Zugriffszeiten auf die Prozessorspeicher in inaktiven Prozessortaktzyklen [Tria].

Zugriffsmodus	Prozessortaktzyklen
Eigener DSPR, lesend/schreibend	0
Fremder DSPR, lesend	5
Fremder DSPR, schreibend	0
Eigener oder fremder PSPR, lesend	5
Eigener oder fremder PSPR, schreibend	0
Eigener PSPR, Instruktion lesend	0
Fremder PSPR, 1. Instruktion lesend	5
Fremder PSPR, weitere Instruktionen lesend	0
P-Flash, 1. Zugriff	5 + Wartezyklen
P-Flash, weitere Zugriffe	0

4.2 Astrée Datenflussanalyse

Zur Durchführung einer Variablenklassifikation wird, neben den Informationen aus der BS-Konfiguration, eine Datenflussanalyse des Anwendungsprogrammes benötigt, da aus dieser das Zugriffsverhalten bestimmt wird. Im Rahmen dieser Arbeit wird das statische Analysewerkzeug Astrée verwendet, das während der semantischen Analyse eine Datenflussanalyse erstellt.

Als Eingangsdaten benötigt Astrée den Anwendungscode, der sich aus dem generierten C-Code und dem Integrationscode zusammensetzt. Zusätzlich wird die Konfiguration des AUTOSAR OS übergeben, um den Aufbau eines Astrée-Prozesssystems zu ermöglichen.

Bevor die Ergebnisse der Datenflussanalyse genutzt werden können, müssen alle Fehler der Kategorien 1 bis 8 (siehe Kapitel 2.3.3.3) beseitigt oder annotiert werden. Ist dies nicht gegeben, sind die Ergebnisse der Datenflussanalyse eventuell nicht vollständig, da Ausführungspfade nicht betrachtet wurden oder noch Fehler in Zeigerzugriffen bestehen. Aus Astrée wird die Datenflussanalyse als Bericht ausgeleitet. Dieser enthält, wie in Kapitel 3.3 beschrieben, Informationen zu jedem Zugriff auf ein Datum, inklusive des zugreifenden Kontrollflusses und der Zugriffsart.

4.3 Magic Memory Handling Tool

Aufbauend auf der Betriebssystemkonfiguration, der Systemspezifikation und der von Astrée erstellten Datenflussanalyse nimmt das MMHT eine Variablenklassifizierung vor und führt diese in die Code-Generierung zurück. Ziel ist hierbei, dass in einer generisch entwickelten Anwendung die Anforderungen des Gesamtsystems wie Echtzeitfähigkeit, Laufzeiteffizienz und räumliche Isolation erfüllt werden können. Im Folgenden wird auf die Funktionsweise des MMHT eingegangen. Dazu werden die drei Phasen:

1. Parser und Informationsaggregation,
2. Klassifizierung und
3. Code-Generierung

betrachtet. Diese Struktur findet sich auch so im MMHT-Programmcode wieder und ermöglicht durch definierte Schnittstellen den Austausch der Implementierung von Phasen oder Teilen dieser

Phasen. In Abbildung 4.2 sind der Aufbau der Phasen sowie ihre Ein- und Ausgangsinformationen dargestellt.

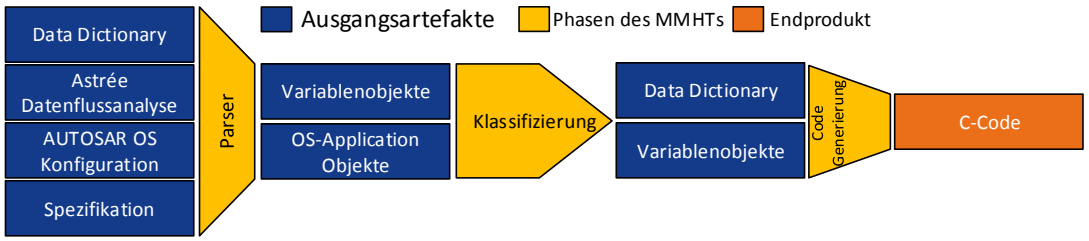


Abbildung 4.2 – Phasen des MMHTs mit Ein- und Ausgangsinformationen.

Dass MMHT wurde in Python 3 implementiert, da diese Sprache gute Schnittstellen zu Matlab sowie den verschiedenen genutzten Austauschformaten bietet.

4.3.1 Parser und Informationsaggregation

Zur Klassifizierung von Variablen werden Informationen aus verschiedenen Quellen benötigt. Diese teilen sich in drei Kategorien:

- Systemspezifikation,
- Datenflussanalyse und
- Betriebssystemkonfiguration.

Informationen aus der Systemspezifikation sind zum einen, welche Variablen Kalibrier- oder Messvariablen darstellen. Zum anderen Platzierungsbedingungen, wie zum Beispiel Vorgaben, dass OS-Applications nicht auf dem selben Kern gebunden werden dürfen. Aus der Datenflussanalyse und der Betriebssystemkonfiguration kann das Zugriffsverhalten auf die Daten abgeleitet werden. Die Betriebssystemkonfiguration enthält darüber hinaus Informationen über das Kontrollflusssystem der Anwendung. Ziel der Parserstufe ist es, alle notwendigen Informationen in drei Objekten zu sammeln, die anschließend von der Klassifizierungsstufe verwendet werden können. Die Objekte stellen dabei die definierte Schnittstelle zwischen beiden Phasen dar. Im Folgenden wird zuerst erläutert, welche Informationen benötigt werden. Anschließend wird vorgestellt, wie jeweils die benötigten Informationen aus den Quellen extrahiert werden. Abschließend wird beschrieben, wie die Informationen durch die Austauschobjekte der ersten und zweiten Stufe repräsentiert werden.

Zur Klassifizierung werden folgende Informationen benötigt:

1. Variablenname: der Name der Variable im C-Code und im DD.
2. Variablenmodul: der Name des C-Moduls, in dem die Variable deklariert wird.
3. Besitzender Kontrollfluss: der Name des Kontrollflusses, der als Besitzer der Variable dient. Auf Grund der Implementierung des von TL generierten Codes kann dieser aus dem Kontrollfluss bestimmt werden, in dem die Variable deklariert wird.
4. Lesend oder schreibend zugreifende Kontrollflüsse.
5. Zugriffsverhalten: Welche Kontrollflüsse auf welche Art auf das Datum zugreifen.

6. Zugriffsfrequenz: Mit welcher Frequenz greift ein Kontrollfluss auf das Datum zu.
7. Zugehörige OS-Application: Welcher OS-Application ist das Datum zugeordnet.
8. OS-Application Name: der Name der OS-Application in der AUTOSAR Konfiguration.
9. Kern der OS-Application: der Prozessorkern, dem die OS-Application zugeordnet ist.
10. Kontrollflüsse der OS-Applications: welche Kontrollflüsse der OS-Application zugeordnet sind.
11. Platzierungsbedingungen: Unter welchen Bedingungen dürfen Daten von OS-Applications auf die Kerne gebunden werden.
12. Kalibrier- und Messvariablen: Welche Variablen sind als Kalibrier- und Messvariablen in der Spezifikation angegeben. Wie in Kapitel 3.4.3 beschrieben, müssen diese gesondert behandelt werden.

4.3.1.1 Spezifikationsparser

Um Informationen über Kalibrier- und Messvariablen zu bekommen, werden A2L-Dateien ausgelesen. Dies ist ein Dateiformat, das symbolische Namen mit Hardwareadressen verknüpft, um so Messprotokollen wie XCP Zugriff auf diese Daten zu ermöglichen [Vek]. Diese Dateien können zum Beispiel aus einem TL DD ausgeleitet werden. Der zugehörige Parser `variables_from_a2l` durchläuft die A2L-Datei und speichert die Variablen nach den Kategorien „Messvariablen“ und „Kalibriervariablen“ in einem Python-Dictionary. Ein Python-Dictionary ist eine Hash-tabellenartige Datenstruktur, in der Schlüsseln Daten zugeordnet werden.

Um Informationen über Platzierungsbedingungen zu erlangen, werden diese durch eine simple Syntax im `extract_constraints_from_spec`-Parser beschrieben. Diese erlaubt es zu beschreiben, ob Daten von OS-Applications auf dem Kern einer anderen OS-Application platziert werden dürfen (siehe Kapitel 3.4.3). Außerdem ist es möglich zu beschreiben, ob Daten einer OS-Application auf dem Kern einer anderen OS-Application platziert werden müssen, wenn diese auch darauf Zugriff hat. Durch diese beiden Bedingungen kann eine räumliche Trennung durch die Platzierung auf unterschiedlichen kernnahen Speichern ermöglicht werden. Für sichere Rückwirkungsfreiheit zur Laufzeit muss die räumliche Isolation durch MPUs oder Buszugriffskontrollen sichergestellt werden (siehe Kapitel 2.3.1.2). In Listing 4.1 ist ein Beispiel der Syntax dargestellt. Der Operator `!=` bedeutet, dass die Daten des linken und des rechten Operanden nicht auf den selben Kern gebunden werden dürfen (exklusiv). Der Operator `==` sagt aus, dass Daten des linken und des rechten Operanden auf den selben Kern gebunden werden sollen (inklusiv). Die Operanden sind jeweils die Namen von OS-Applications aus der Spezifikation. Hinter `//` folgt ein Kommentar. Der Parser `extract_constrains_from_spec` liest in seiner aktuellen Form die ersten drei, durch Leerzeichen getrennten, Zeichenketten aus. Die erste und dritte Zeichenkette werden als Operanden, die zweite als Operator betrachtet. Daraus werden die Platzierungsbedingungen abgeleitet.

```
1 OS_App1_Core0 != OS_App1_Core1 // Exclusive constraint
2 OS_App1_Core0 == OS_App1_Core2 // Inclusive constraint
```

Listing 4.1 – Beispiel für die Syntax von Platzierungsbedingungen.

Ergebnis der Parser ist zum einen ein Python-Dictionary aller Mess- und Kalibriervariablen. Zum anderen ein Python-Dictionary, das inklusive und exklusive Platzierungsbedingungen enthält.

4.3.1.2 Astrée-Berichtparser

Die Astrée-Datenflussanalyse wird als komma-separierte Datei ausgeleitet. Diese besteht aus den Spalten:

- Variable: Symbol der Variable.
- Function: C-Funktion, aus der der Zugriff erfolgt.
- Access: Art des Zugriffs (lesend/schreibend).
- Process: Astrée-Prozess aus dem der Zugriff erfolgt
- Data races: beschreibt, ob Astrée eine Wettlaufsituation beim Zugriff auf dieses Datum festgestellt hat.
- Shared variable: beschreibt, ob auf dieses Datum von mehreren Prozessen zugegriffen wird.
- Location: Stelle im Quellcode, an der die zugreifende Operation deklariert wird.

Entsprechend der Spaltenbezeichnung wird jede Zeile in einem Python-Dictionary abgelegt und in einer Liste gespeichert. Diese Liste wird anschließend benutzt, um für jedes erfasste Datum ein Python-Variablenobjekt anzulegen. Diese werden über die TargetLink Matlab API mit einem DD-Variablenobjekt verknüpft. Hierzu wird der eindeutige Handle der DD-Variablenobjekte verwendet. Basis der Verknüpfung ist zum einen die „Location“ der Variable und zum anderen ihr Symbol, das in der Spalte Variable enthalten ist. Verfügt eine Variable nicht über eine Repräsentation im DD, weil sie zum Beispiel nicht aus dem generierten Code, sondern aus dem Betriebssystem stammt, wird sie mit einem Standardhandle versehen. Dieser Standardwert kann niemals ein Handle eines DD-Variablenobjekts sein. Anschließend wird aus der „Process“-Spalte bestimmt, welche Kontrollflüsse auf das Datum zugreifen. Die „Access“-Spalte beschreibt die Art des Zugriffes, also lesend oder schreibend. Aus der Häufigkeit, mit der eine Variable in dem Bericht auftaucht, kann bestimmt werden, wie häufig auf das Datum innerhalb eines Astrée-Durchlaufs zugegriffen wird.

Indem zuerst versucht wird, jedem Variablensymbol ein eindeutiges Handle zuzuordnen, kann vermieden werden, dass Daten mit dem gleichen Symbol in einem Python-Variablenobjekt zusammengefasst werden. Dies bedeutet aber auch im Umkehrschluss, dass Variablen, die nicht aus dem generierten Code stammen, nicht immer eindeutig identifizierbar sind und deshalb falsch oder gar nicht klassifiziert werden. Um dieses Risiko zu vermeiden, erfolgt bei Variablen mit dem Standardhandle als Handle ein zusätzlicher Vergleich des Symbols und des Moduls, in dem sie deklariert werden.

Ergebnis ist eine Liste von Python-Variablenobjekten, die Informationen je Datum über das Zugriffsverhalten aus der Astrée-Datenfluss beinhalten.

4.3.1.3 AUTOSAR Konfigurationsparser

Die AUTOSAR Konfiguration wird in Form einer ARXML-Datei gespeichert. Dies ist ein durch den AUTOSAR Standard spezifiziertes Austauschformat für AUTOSAR Modelle, das auf dem XML-Standard aufbaut [AUT17a].

Der Parser `os_apps_from_arxml` dient dazu, OS-Applications zu charakterisieren. Aus der ARXML-Datei wird ausgelesen, welche OS-Applications existieren, auf welche Kerne diese gebunden sind und welche Kontrollflüsse ihnen zugeordnet werden (siehe Abbildung 2.8).

Im anschließenden Schritt (`map_var_to_os_app`) werden die so gefundenen OS-Applications und ihre Kontrollflüsse mit Python-Variablenobjekten verknüpft. Dies erfolgt auf Basis der zugreifenden Kontrollflüsse. Greift mehr als ein Kontrollfluss auf ein Datum zu, wird in der aktuellen Implementierung davon ausgegangen, dass der zuerst schreibende Kontrollfluss das Datum besitzt. Dies ist eine Annahme, die auf Grund der Code-Generierung mit TL als Ursprung der Anwendung getroffen werden kann. Die entsprechende Verknüpfung wird im Python-Variablenobjekt als Wert eingetragen.

In einem zweiten Schritt wird aus der ARXML-Datei das Ablaufsystem ausgelesen. Dies ist notwendig, da die aus der Astrée Datenflussanalyse bestimmte Zugriffsfrequenz nur für einen Systemdurchlauf gültig ist. Die tatsächliche Aufruffrequenz der Kontrollflüsse wird nicht beachtet, wie in Kapitel 3.3 beschrieben. Auf Grund der Vereinfachungen (siehe Kapitel 3.3) kann auf die Erstellung eines vollständigen Ablaufplans verzichtet werden. Stattdessen kann die Periode (P) eines Kontrollflusses T_i durch die Hyperperiode (H) dividiert werden, um die Ausführungsfrequenz zu bestimmen:

$$f_{\text{Ausführung},i} = \frac{P_i}{H} \tag{4.1}$$

Anschließend wird die Zugriffsfrequenz der Variablen aus Astrée mit der Aufruffrequenz der Kontrollflüsse aus der Ablaufplanung multipliziert, um so die tatsächliche Zugriffsfrequenz zu bestimmen.

Ergebnis ist zum einen eine Liste von OS-Applications-Objekten, die das Kontrollflussmodell beschreiben. Zum anderen entsteht ein Python-Dictionary, das den einzelnen Kontrollflüssen ihre Aufruffrequenz innerhalb einer Hyperperiode zuweist.

4.3.1.4 Austauschobjekte

Zur Verarbeitung der gesammelten Informationen werden diese Informationen in Form von drei Austauschobjekten zusammengefasst:

- *Variable*,
- *OSApplication* und
- eine Liste der Kalibrier- und Messvariablen.

In Abbildung 4.3 sind die Klassendiagramme der *Variable*- und der *OSApplication*-Objekte dargestellt. Grau gefärbte Attribute werden erst in späteren Phasen des MMHT benötigt und sind am Ende der Parserstufe noch nicht mit Werten gefüllt.

Variablen-Objekte werden entweder eindeutig durch ein DD-Handle (`handle`) identifiziert oder durch ihren Namen (`name`) und ihr Modul (`modul`). Dies ist in der Vergleichsfunktion (`__eq__`) des Objektes implementiert und verhindert während des Parsens der Astrée Datenflussanalyse, dass Daten zusammengefasst werden. Dies kann passieren, wenn kein Handle für ein Datum existiert und Name sowie Modul gleich sind. Ein Beispiel hierfür sind Daten, die auf Sichtbarkeitsebene von C-Funktionen deklariert werden, aber nicht aus dem DD stammen. Der besitzende Kontrollfluss wird über seinen Namen dem *Variable*-Objekt im Attribut `task` zugeordnet werden. Die auf das Datum zugreifenden Kontrollflüsse werden zum einen nur durch ihren Namen, der Variable als Liste `accessing_tasks` zugeordnet. Zum anderen werden in zwei Python-Dictionarys (`read_task`,

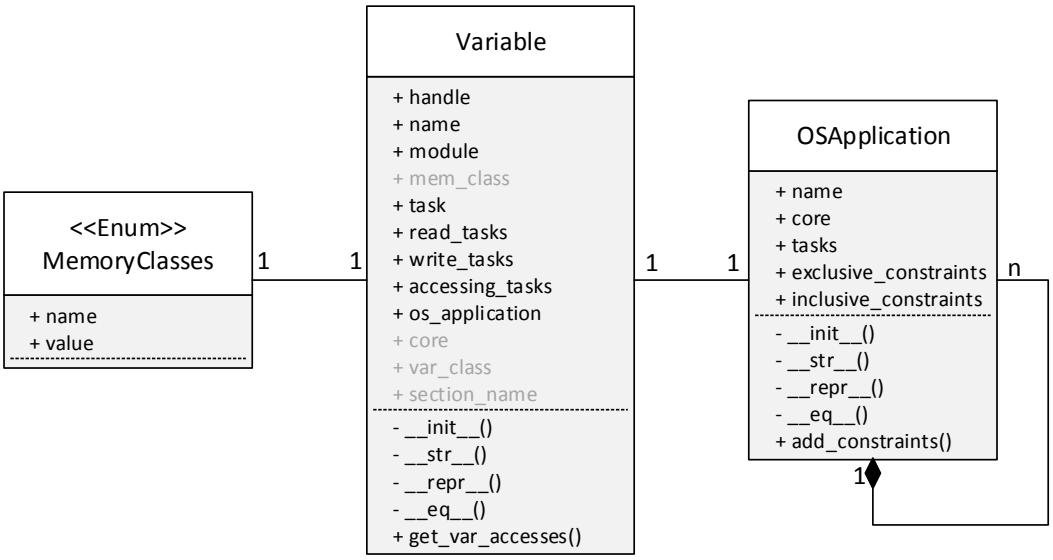


Abbildung 4.3 – Klassendiagramm der Austauschobjekte *Variable* und *OSApplication*

`write_task`), sortiert nach lesenden und schreibenden, die zugreifenden Kontrollflüsse mit deren Zugriffsfrequenzen im Python-Variablenobjekt verwaltet. Die OS-Application wird dem Datum auf Basis des besitzenden Kontrollflusses zugeordnet. Diese Relation wird über die Zuordnung eines *OSApplication*-Objekts durch das Attribut `os_application` umgesetzt.

In *OSApplication*-Objekten werden die Platzierungsbedingungen für die einzelnen Instanzen von OS-Applications als Referenz auf ein anderes *OSApplication*-Objekte verwaltet, getrennt in inklusive und exklusive Platzierungsbedingungen. Ausgehend von dem in Kapitel 4.3.1.1 beschriebenen Python-Dictionary der Platzierungsbedingungen weist die Funktion `add_constraints` jedem *OSApplication*-Objekt die relevanten Platzierungsbedingungen zu.

Im letzten Schritt der Parserstufe werden Austauschobjekte serialisiert, sodass in wiederholten Durchläufen des MMHTs die Parserstufe übersprungen werden kann. Hierzu wird `pickle`, eine Serialisierungs-Bibliothek von Python, verwendet.

4.3.2 Klassifizierung

Auf Basis der in der Parserstufe gesammelten Informationen erfolgt die Klassifizierung. Um die Laufzeiteffizienz zu erhöhen, sollen Daten, auf die häufig durch Kontrollflüsse zugegriffen wird, in kernnahen Speichern platziert werden. Um dies sicherzustellen, wird die Liste der *Variable*-Objekte nach der absoluten Anzahl an Zugriffen auf ein Datum sortiert. Hierzu wird die Anzahl der lesenden und schreibenden Zugriffe durch alle Tasks auf das Datum aufsummiert, um so die absolute Anzahl an Zugriffen zu bestimmen.

Anschließend erfolgt die eigentliche Klassifizierung nach den in Kapitel 3.4.4 vorgestellten Klassen. Hierzu wird nach einem Regelwerk vorgegangen, das auch die Platzierung in den kernnahen Speichern berücksichtigen soll. In Abbildung 4.4 ist das Regelwerk in einem Flussdiagramm visualisiert. In den Teildiagrammen $\Psi 1$ und $\Psi 2$ ist jeweils die Unterscheidung nach konstanten und nicht-konstanten Daten dargestellt. Die durch einen „/ “ abgetrennte Ziffer, zum Beispiel $\Psi 1/1$, bedeutet, dass innerhalb des Teildiagramms der Zustand 1 erreicht werden kann.

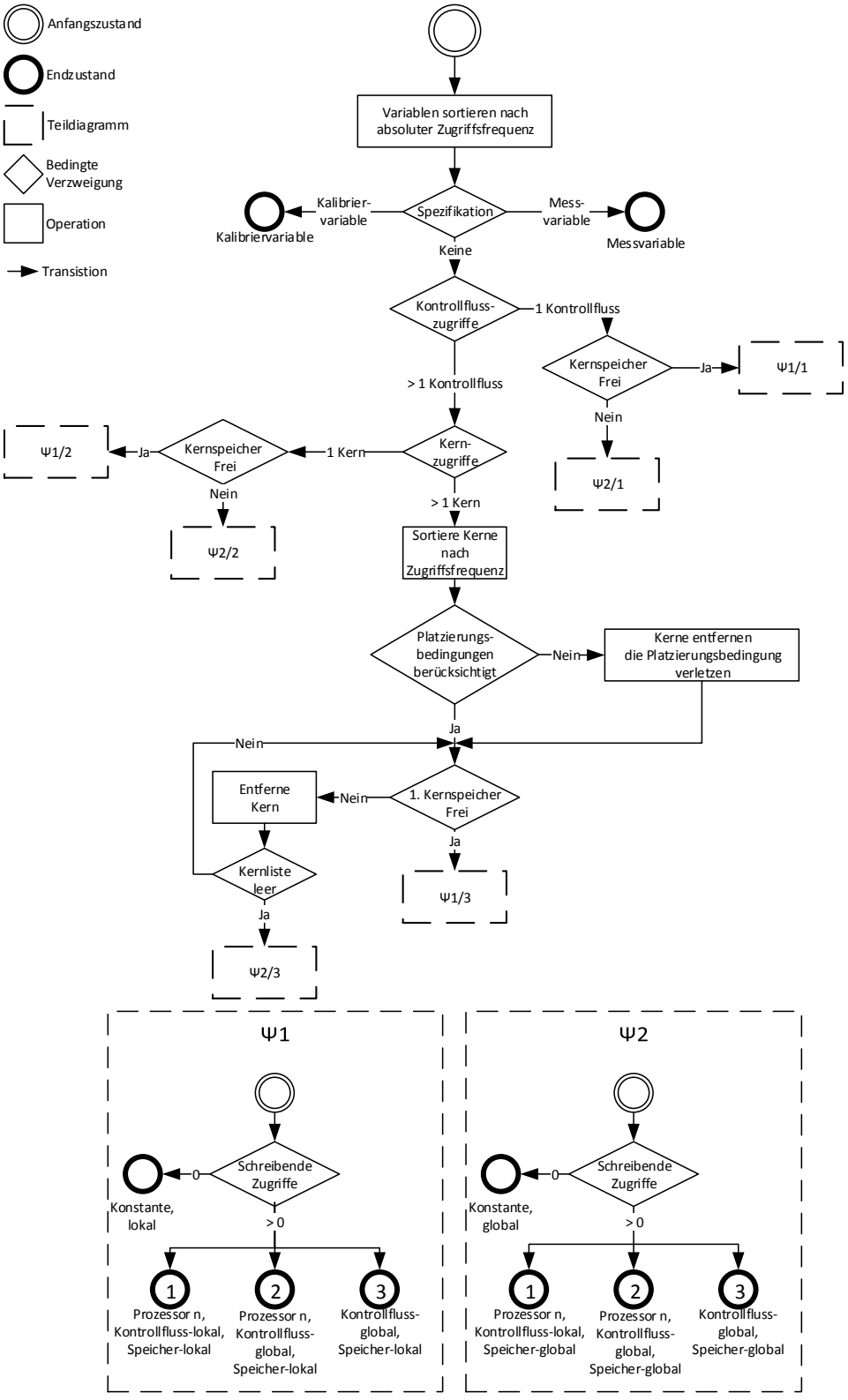


Abbildung 4.4 – Flussdiagramm der Variablenklassifikation.

Die Klasse *Stack-Variablen* wird nicht vergeben, weil solche Variablen nicht durch die Astrée-Datenflussanalyse erfasst werden. Die Klasse *Optimierbar* wird ebenfalls nicht verwendet, da hierzu eine genauere Betrachtung des Datums vorgenommen werden muss (siehe Kapitel 3.4.4).

Wie aus dem Diagramm ersichtlich, wird zuerst geprüft, ob ein Datum eine Kalibrierungsvariable oder eine Messvariable darstellt. Anschließend werden Variablen mit nur einem zugreifenden Kontrollfluss klassifiziert, also solche die Kontrollfluss-lokal sind (in Abbildung 4.4 Zustände, die innerhalb der Teildiagramme Ψ mit 1 markiert sind).

Für Kontrollfluss-globale Daten wird weiter in CPU-lokal und CPU-global unterteilt. Bei ersteren wird erneut überprüft, ob die kernnahen Speicher bereits belegt sind (in Abbildung 4.4 Zustände, die innerhalb der Teildiagramme Ψ mit 2 markiert sind). Bei CPU-globalen Daten wird bestimmt, auf welche Kerne die zugreifenden Kontrollflüsse gebunden sind (in Abbildung 4.4 Zustände, die innerhalb der Teildiagramme Ψ mit 3 markiert sind). Kerne, die auf Grund von Platzierungsbedingungen nicht in Frage kommen, werden vorher aussortiert. Anschließend wird über die in Frage kommenden Kerne in absteigender Reihenfolge ihrer Zugriffsfrequenz auf das Datum iteriert. Ist auf keinem der Kerne Speicher frei, wird das Datum im globalen Speicher platziert.

Nach der Klassifizierung der Kern- und Kontrollflusseigenschaften wird überprüft, ob der kernnahe Speicher schon belegt ist. In der aktuellen Implementierung ist hierfür nur eine Platzhalterfunktion implementiert, die nach einer Standardverteilung zurückgibt, dass der Speicher voll oder leer ist. Hierfür existierte zum Zeitpunkt der Arbeit keine Informationsquelle, die Aussagen über den Speicherstand der kernnahen Speicher erlaubt. Neben den durch das MMHT analysierten Variablen können Betriebssystemdienste, Funktionsstacks und Registerzustände in kernnahen Speicher gesichert werden. Diese müssen bei der Bindung der Daten auf die Speicher auch berücksichtigt werden. Des Weiteren spielen Effekte wie Adressausrichtung von Daten eine weitere Rolle. Dementsprechend können nicht einfach die Speichergrößen der analysierten Objekte aufsummiert werden. Je nach Zustand der kernnahen Speicher erfolgt die Klassifizierung dann als Speicher-lokal oder Speicher-global. Die Berücksichtigung der Speicherzustände ist ein offener Punkt dieser Arbeit, der zu einem späteren Zeitpunkt gelöst werden wird.

Nach jeder Klassifizierung nach Kontrollfluss und Speicherort wird jeweils in einer Klassifizierung ($\Psi 1$ und $\Psi 2$) überprüft, ob es sich bei der Variable um eine Konstante handelt. Hierzu wird die Anzahl der schreibenden Kontrollflüsse verwendet. Existieren keine schreibenden Kontrollflüsse, handelt es sich aus Softwaresicht um eine Konstante, da Astrée Schreibvorgänge während der Deklaration nicht als Zugriff erfasst. Ausgenommen hiervon sind Kalibrier- und Messvariablen.

Endergebnis ist ein *Variable*-Objekt pro Datum, dem eine Speicherklasse in Form einer Enumeration der Klasse *MemoryClasses* (`mem_class`) sowie ein Kern `core` zugewiesen wurde (siehe Abbildung 4.3).

4.3.3 Code-Generierung

Ziel der Code-Generierungsphase ist es, auf Basis der Klassifizierung eine Bindung der Daten auf die ihnen zugeordneten Speicher zu erreichen. Gleichzeitig soll die Bindung so durchgeführt werden, dass darauf aufbauend Speicherregionen zu Speicherschutzregionen in AUTOSAR OS zugeordnet werden können. Abschließend wird durch TL Code generiert, der diese Bindung widerspiegelt.

Beide Anforderungen werden durch die vom Linker verwendeten Speichersektionen erfüllt. Ein ähnliches Vorgehen wird auch in der AUTOSAR Spezifikation beschrieben [AUT17b].

Ein Linker oder Binder bildet die im Code verwendeten Symbole auf konkrete Speicheradressen ab [Lev99]. Hierfür wird in einem Linker-Skript das Speicherlayout der Zielplattform durch ihre physischen Adressbereiche beschrieben [Lev99]. In Abbildung 4.5 ist beispielhaft der Adressraum

des Aurix TC27x dargestellt. Wie aus der Abbildung ersichtlich, werden über den Adressraum alle

15	Interne Peripheriegeräte	0xF000 0000
14	reserviert	0xE000 0000
13	Lokaler DSPR	0xD000 0000
12	Lokaler PSPR	0xC000 0000
11	LMU SRAM	0xB000 0000
10	P-Flash, D-Flash	0xA000 0000
9	LMU SRAM	0x9000 0000
8	P-Flash	0x8000 0000
7	CPU0 PSPR/DSPR	0x7000 0000
6	CPU1 PSPR/DSPR	0x6000 0000
5	CPU2 PSPR/DSPR	0x5000 0000
4	CPU3 PSPR/DSPR	0x4000 0000
3	CPU4 PSPR/DSPR	0x3000 0000
2	CPU5 PSPR/DSPR	0x2000 0000
1	CPU6 PSPR/DSPR	0x1000 0000
0	CPU7 PSPR/DSPR	0x0000 0000

Abbildung 4.5 – Speicherlayout des Aurix TC27x nach [Tria].

kernnahen Speicher (0 - 7) sowie die globalen Speicher (8 - 11) angesprochen. Zur Bindung von Funktionen und Daten werden Sektionen verwendet, die im Adressraum der Zielplattform durch den Linker platziert werden. Diese Sektionen können durch den Linker automatisch aneinander gereiht oder durch den Entwickler spezifiziert werden. Eine Mischform ist ebenfalls möglich, in der Sektionen durch den Entwickler spezifiziert werden und Untersektionen in diesen durch den Linker aneinandergereiht werden. Das Erstellen dieser Linkerskripte erfolgt auf Grund der zeitlichen Beschränkung nicht in den Rahmen dieser Arbeit. In der Code-Generierung werden die später im Linkerskript nutzbaren Sektionen den Daten nach der Klassifizierung zugewiesen.

Um Sektionen im C-Code zu deklarieren existieren zwei Ansätze: die Verwendung von `#pragma`-Anweisungen oder durch `attribute`-Qualifizierer. Ziel-Compiler dieser Arbeit sind der Tasking TriCore-Compiler [Alt] und der HighTec TriCore GCC [Hig], welche beide Ansätze unterstützen. In Listing 4.2 ist ein Beispiel für beide Varianten gegeben.

```
1 ...
2 #pragma section type "test" // Everything that follows is placed in section ↘
      test
3 int i = 0;
4 int j = 0;
5 #pragma clear // Clears previous #pragma statements
6
7 int k __attribute__((section("test"))) = 0; // Places k in the section test
8 ...
```

Listing 4.2 – Beispiel für die Zuweisung von Sektionen im C-Code.

Zur Zuweisung von Sektionen zu Daten wird die Code-Generierung von TL verwendet. Dies ist allerdings nur für Daten möglich, die auch eine Repräsentation im DD haben. Andere Daten werden zwar ebenfalls mit Sektionsnamen versehen, deren Deklaration im C-Code wird allerdings in dieser Phase nicht verändert. Zur Beeinflussung der Code-Generierung werden die in Kapitel 3.5.3 erwähnten DD-Variablenklassen verwendet. Diese fassen DD-Variablenobjekte mit gleichen Eigenschaften im C-Code zusammen. Zu diesen Eigenschaften zählen:

- eine Beschreibung der Variablenklasse,
- der Zugriffsraum (Scope) der zugehörigen Variablen,
- Qualifizierer wie `volatile` und `const`,
- die Ausleitbarkeit in eine A2L-Datei als Kalibrier- oder Messvariable bestimmen,
- Möglichkeiten der Optimierung während der Code-Generierung, wie das Entfernen des Datums oder das freie Platzieren des Datums im Code und
- Parameter zum Hinzufügen von Prefixen oder C-Anweisungen vor und nach Code-Segmenten.

Die Eigenschaften „Typeprefix“ und „DeclarationStatement“ können verwendet werden, um `attribute`-Qualifizierer, beziehungsweise `#pragma`-Anweisungen in den Code einzubringen.

Durch die Funktion `assign_variable_classes` werden zum einen DD-Variablenklassen im DD angelegt, wenn diese nicht existieren. Zum anderen werden sie sowohl den DD- als auch dem Python-Variablenobjekt zugewiesen. Der Name der DD-Variablenklasse entspricht dem Namen der zugewiesenen Sektion und folgt dem Namensschema:
`<OS-Application>_<Kern-ID>_<Kontrollfluss>_<Variablenklasse>`. Diese Sektionen ermöglichen es, auf Ebene der Kontrollflüsse Speicherschutzregionen aufzuspannen. Durch die feingranularere Aufteilung in die Zugriffsklassen können die zugehörigen Variablen auf die verschiedenen Speicher verteilt werden und die Zugriffsrechte, entsprechend der anwendungsabhängigen Anforderungen, für die Speicherschutzregionen festgelegt werden.

In einem letzten Schritt wird über die Matlab API der TL-Code-Generierungsprozess gestartet werden. Zusätzlich werden die Eigenschaften der Python-Variablenobjekt in einer komma-separierten Datei ausgeleitet. Dies dient zum einen der Dokumentation und manuellen Überprüfung der Ergebnisse. Zum anderen können auf Basis der Ergebnisse, Variablen, die nicht im DD repräsentiert sind, angepasst werden.

4.4 Aufspannen von Speicherschutzregionen

Die in Kapitel 4.3.3 erzeugten Sektionen können anschließend in einem AUTOSAR OS Konfigurator wie DaVinci [Vec] OS-Applications zugewiesen werden. Hierzu werden Datenschutzregionen angelegt, die als Grenzen die Sektionsinformationen zugewiesen bekommen. Den einzelnen Datenschutzregionen können pro Kontrollfluss oder OS-Application Zugriffsrechte (lesend oder schreibend) zugewiesen [AUT17c]. Diese Datenschutzregionen werden zur Laufzeit durch das BS verwaltet, indem die Hardwarespeichereinheiten dem aktuellen Programmkontext entsprechend konfiguriert werden.

4.5 Resümee

In diesem Kapitel wurden die Hardwaregegebenheiten der Prozessorfamilie beschrieben, auf die diese Implementierung des MMHT abzielt. Anhand der verschiedenen Phasen: Parser, Klassifizierung und Code-Generierung wurde der Aufbau des MMHTs vorgestellt und auf die Eingangs- und Ausgangsergebnisse jeden Schrittes sowie die notwendigen Transformationen eingegangen. Abschließend wurde kurz aufgezeigt, wie die erzeugen Speicherregionen dazu genutzt werden können, Speicherschutzregionen im AUTOSAR OS umzusetzen.

5 EVALUATION

Zur qualitativen Evaluation der Ergebnisse der Variablenklassifizierung durch das MMHT soll eine Beispielanwendung analysiert und klassifiziert werden, die auf dem in Kapitel 4.1 vorgestellten Aurix TC277 implementiert wird. Hierzu wird auf die Beispielanwendung des *I4Copters* [Ulb+11] eingegangen, die im Rahmen dieser Arbeit verwendet wird. Die Anpassung an dem Simulink-Modell des *I4Copters*, die für seine Verwendung nötig waren, werden beschrieben. Aus der Anwendung werden verschiedene Systemvarianten abgeleitet, die unterschiedliche Szenarien zur Variablenklassifizierung darstellen. Diese Varianten werden einer Astrée-Analyse unterzogen, deren Ergebnisse kurz mit den daraus resultierenden Anpassungen vorgestellt werden. Abschließend werden die Ergebnisse der Variablenklassifizierung sowie die daraus entstehenden Speicherzuweisungen diskutiert.

Das Modell sowie der generierte Code des *I4Copters* sind unter [Brä18b] zu finden. Die Systemvarianten und die dazugehörigen Dokumente können unter [Brä18c] eingesehen werden.

5.1 I4Copter Modell

Als Anwendung wird in dieser Arbeit das Reglermodell des *I4Copters* genutzt. Der *I4Copter* ist ein Forschungsprojekt, bei dem Fragestellungen der Domänen Echtzeitsysteme und Regelungstechnik an einem Quadrocopter betrachtet werden [Ulb+12]. Im Folgenden wird der Modellaufbau des *I4Copter*-Reglers betrachtet, um ein Verständnis für die Anforderungen der Anwendung zu schaffen.

In Abbildung 5.1 ist der Datenflussgraph (DFG) der obersten Ebene des *I4Copters* dargestellt, welcher sich aus dem Simulink Modell ergibt. Es handelt sich um eine kaskadierte Regelung mit

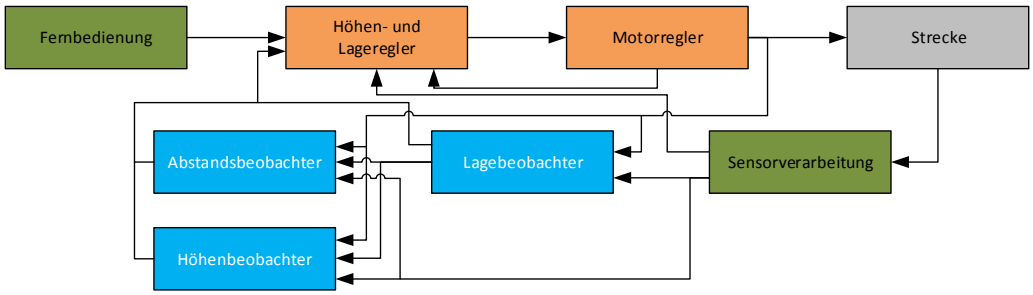


Abbildung 5.1 – DFG des *I4Copter* Reglers.

Zustandsbeobachtern. In der ersten Stufe der Kaskade werden Lage- und Höhe des Quadrocopters, in der zweiten Stufe die Motoren der Propeller geregelt. Diese Knoten sind orange hinterlegt. Die blau hinterlegten Beobachter rekonstruieren nicht messbare Zustände aus der zurückgeführten Ausgangsgröße des Reglers und den Sensorsignalen aus der Sensorverarbeitung. Eine Fernbedienung und die Sensorverarbeitung liefern Eingangssignale, die zum einen als Führungsgröße in den Regler eingehen und zum anderen zur Rekonstruktion der Zustände genutzt werden. Die grau hinterlegte Strecke simuliert im Modell den geregelten Quadrocopter und dient zur Simulation der Regelung. Sie wird deshalb im Folgenden nicht mehr betrachtet, da die Streckensimulation in der Anwendungssoftware nicht benötigt wird.

Die einzelnen Knoten des DFG in Abbildung 5.1 stellen Funktionskomponenten dar. Diese bestehen in den meisten Fällen aus nur einer Funktionseinheit. Nur der Höhe- und Lageregler besteht aus zwei Einheiten, die jeweils für die Höhen- beziehungsweise Lageregelung verantwortlich sind.

Die einzelnen Komponenten werden mit unterschiedlichen Frequenzen ausgeführt. In Tabelle 5.1 sind die daraus folgenden Perioden für die Komponenten aufgeführt. Die Perioden wurden dem Modell entnommen, da die letzten Angaben aus Forschungsarbeiten aus 2012 stammen. Die Ausführungszeiten wurden [Reb12] entnommen. Diese stellen vermutlich nicht die tatsächlichen WCETs dar, des Weiteren ist unbekannt auf welcher Plattform sie bestimmt wurden, weshalb sie nur der Vollständigkeit halber angegeben werden. Diese Perioden werden später in Kapitel 5.3 verwendet,

Tabelle 5.1 – Perioden und Ausführungszeiten der Funktionskomponenten.

Komponente	Periode	Ausführungszeiten
Fernbedienung	-	-
Höhen- und Lageregelung	9 ms	kombiniert 0,15 ms
Motorregler	9 ms	
Sensorverarbeitung	9 ms	0,3 ms
Lagebeobachter	3 ms	0,6 ms
Abstandsbeobachter	3 ms	0,6 ms
Höhenbeobachter	3 ms	0,6 ms

um ein AUTOSAR OS-System für das Modell zu erstellen.

5.2 Modellvorbereitungen

Das Modell des *I4Copters* wurde in Simulink implementiert. Auf Grund der vielfältigeren Möglichkeiten von TL auf die Code-Generierung Einfluss zu nehmen, wird im Rahmen dieser Arbeit ein TL-Modell erstellt. Im Folgenden werden die Änderungen beschrieben, die vorgenommen wurden, um zum einen das Simulink-Modell in ein TL-Modell zu überführen. Des Weiteren wird beschrieben welche Schritte unternommen wurden, um die gezielte Beinflussung von Variablenrepräsentationen im generierten Code zu ermöglichen.

5.2.1 Überführung in ein TargetLink-Modell

TL verwendet ein spezielles Blockset, um die Parametrierung von Blöcken hinsichtlich der Code-Generierung zu ermöglichen. Entsprechend müssen alle Simulink-Blöcke zu TL-Blöcken erweitert

werden. Alle Subsysteme[6], aus denen Code generiert werden soll, müssen zu TargetLink Subsystemblöcken transformiert werden.

Im Zuge der Arbeit wird aus den Subsystemen Höhe- und Lageregelung, Motorregler sowie allen Beobachtern Code generiert. Diese Subsysteme sind äquivalent zu den in Kapitel 5.1 beschriebenen Funktionskomponenten und stellen deren Implementierung im Modell dar. Wie in Kapitel 5.1 beschrieben, wird das Subsystem der Strecke nicht übersetzt, da es nur Simulationszwecken dient. Das Subsystem der Sensorverarbeitung wird ebenfalls nicht übersetzt, sondern später als Stub-Funktion implementiert. Dies hat den Grund, dass das Sensorverarbeitungs-Subsystem auch Sensorblöcke enthält, aus denen TargetLink keinen Code generieren kann und die in einer praktischen Anwendung durch Peripherietreiber ausgelesen werden. Die Komponente Fernbedienung wird im Modell im Rahmen einer C-Funktion eingebunden, die allerdings neben der Funktion Fernbedienung noch weitere Funktionen erfüllt. Diese werden im Rahmen der Arbeit nicht benötigt, weshalb die Komponente Fernbedienung ebenfalls als Stub-Funktion implementiert wird.

TL ist in der Lage, Subsysteme automatisch in TL-Subsysteme zu überführen. Im selben Schritt werden auch alle Blöcke im Subsystem zu TL-Blöcken erweitert. In einigen Fällen kann es hierbei allerdings zu Fehlern kommen, da bestimmte Konstrukte durch TL nicht erweiterbar sind. Im Rahmen dieser Anpassung wurden folgende Blockschaltungen angepasst:

- Integratorblöcke mit einem Vorfaktor müssen in eine Kombination aus einem Integratorblock und einem Multiplikationsblock überführt werden. TL akzeptiert nur Integratorblöcke mit einem Vorfaktor gleich eins.
- Integratoren akzeptieren keine Vektoren als Operanden. Entsprechende Signale müssen in ihre Komponenten zerlegt werden und durch einzelne Integratoren verarbeitet werden. Anschließend können die Signalkomponenten wieder zusammengeführt werden.
- Multiplikationsblöcke, bei denen durch Matrizenmultiplikation die Dimensionen des Eingangssignals verändert werden, müssen mit den Faktordimensionen annotiert werden.

Sind alle Subsysteme überführt worden, kann TL C-Code generieren. Je Subsystem wird eine C- und eine Header-Datei angelegt sowie gegebenenfalls Header-Dateien mit Typdefinitionen und C-Makros. Zur Verbesserung der Lesbarkeit und der Modularität des generierten Codes werden Untersysteme der TL-Subsysteme als C-Funktionen übersetzt, die in der C- beziehungsweise Header-Datei des Moduls platziert werden. In Abbildung 5.2 sind die generierten Module, beziehungsweise Subsysteme mit ihren Funktionen aufgelistet. Für bestimmte Blöcke wird durch TL automatisch eine

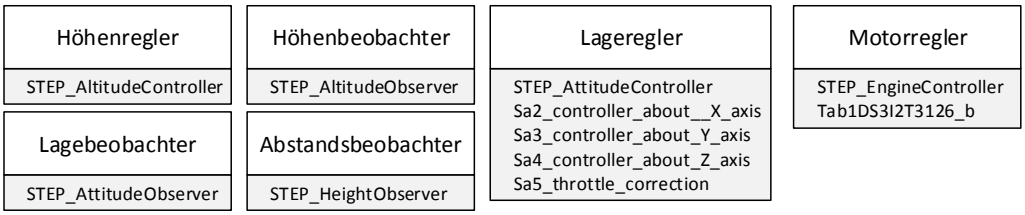


Abbildung 5.2 – Module mit den zugehörigen Funktionen.

Funktion generiert. Ein Beispiel hierfür ist die Funktion `Tab1DS3I2T3126_b` des Moduls Motorregler.

[6]Wird im Folgenden von Subsystemen gesprochen, bezieht sich dies immer auf ein Subsystem im Kontext von Simulink oder TargetLink.

Diese wird aus einer Look-Up-Tabelle generiert, die auf Basis von Eingangssignalen Ausgangsignale, die vor der Ausführung berechnet werden, ausgibt. In diesem speziellen Fall wird zwischen den Ausgangssignalen noch linear interpoliert.

Der aus TL generierte Code kann in eine Anwendung integriert werden. Allerdings ist er, so lange das Modell nicht dementsprechend angepasst wurde, weiterhin anwendungsagnostisch.

5.2.2 Generierung von Variablenobjekten

Wie in Kapitel 3.5.2 beschrieben, kann die Repräsentation von Variablen im generierten C-Code über Variablenobjekte beeinflusst werden. Dies wird im Rahmen dieser Arbeit genutzt, um die Variablenklassifizierung durch DD-Variablenklassen den Variablenobjekten zuzuweisen. Um dies zu ermöglichen, müssen die Blöcke über Referenzen mit den zugehörigen Variablenobjekten verknüpft werden. Zur Vereinfachung und Replizierbarkeit des Prozesses wurde ein Matlab-Skript geschrieben das:

1. die Erstellung von Variablenobjektgruppen[7],
2. die Erstellung von Variablenobjekten,
3. die Zuordnung dieser zu Variablenobjektgruppen und
4. die Verknüpfung von TL-Blöcken mit den zugehörigen Variablenobjekten

automatisiert. Im Folgenden wird dieses Vorgehen kurz beschrieben.

Ausgangspunkt ist eine Liste der Subsysteme, für die Variablenobjekte erstellt werden sollen. Für jedes dieser Subsysteme wird eine Variablenobjektgruppe im DD angelegt. Anschließend werden über die TL API alle TL Blöcke des Subsystems abgefragt. Jeder TL Block wird eindeutig durch einen Fließkomma-Handle identifiziert. Diese Handle werden in einer Liste gespeichert, über die iteriert wird und je nach Blocktyp ein oder mehrere Variablenobjekte im DD angelegt werden. Während der Erweiterung von Blöcken durch TL im Zuge der Modellvorbereitung wird jedem Block ein Datentyp und eine Standard-Variablenklasse zugeordnet. Diese werden als Zustand gespeichert und im anschließenden Schritt den neu generierten Variablenobjekten zugewiesen. Je nach Block werden weitere Informationen, wie die Dimension der Eingangs- und Ausgangssignale oder bei Bussignalen eine Liste aller enthaltenen Signale, erfasst und gegenbenfalls weitere Variablenobjekte erzeugt. Nach der erfolgreichen Erstellung eines Variablenobjekts wird der Block-Eintrag für die Variable mit dem Objekt referenziert. Jede weitere Parametrierung des Blocks erfolgt nun über das DD. Endergebnis im DD ist ein Baum mit Subsystemen-Variablenobjektgruppen als Elternknoten und DD-Variablenobjekten als Blätter. Im Simulink-Modell sind alle für die Code-Generierung relevanten Blöcke mit einem oder mehreren Variablenobjekten im DD verknüpft.

Diese können nach der Variablenklassifizieriung durch das MMHT genutzt werden, um die Code-Generierung zu beeinflussen.

[7] Variablenobjektgruppen sind Möglichkeiten Variablenobjekte im DD zu sortieren, in diesem Fall wurde als Sortierkriterium die Modulzugehörigkeit verwendet.

5.3 I4Copter Softwaresystem-Varianten

Im Folgenden werden vier Varianten beschrieben, wie die Funktionskomponenten des *I4Copters* (siehe Kapitel 5.1) auf Kontrollflüsse im Rahmen des AUTOSAR OS abgebildet und in eine Anwendung integriert werden können. Dazu gehören:

1. Ein Zwei-Kernsystem, mit je zwei Kontrollflüssen pro OS-Application.
2. Ein Drei-Kernsystem mit Unterbrechungsroutine, die einen asynchronen Kontrollfluss darstellt.
3. Ein Drei-Kernsystem mit aus der Spezifikation stammenden Platzierungsbedingungen.

Ziel der Varianten ist es, verschiedene Partitionierungsszenarien zu modellieren.

5.3.1 Variantenunabhängige Modellierung

Alle Varianten bauen auf den selben Kontrollflüssen auf und unterscheiden sich nur in der Abbildung dieser Kontrollflüsse auf OS-Applications und Kerne. In Tabelle 5.2 ist die Zuordnung der Funktionskomponenten auf die Kontrollflüsse dargestellt.

Tabelle 5.2 – Zuordnung der Funktionskomponenten des *I4Copters* auf Kontrollflüsse.

Kontrollfluss	Periode	Priorität	Funktionskomponente
`T1_Controllers`	9 ms	6	Höhenregler Lageregler
`T2_EngineController`	9 ms	7	Motorregler
`T3_AttitudeObserver`	3 ms	10	Lagebeobachter
`T4_HeightObserver`	3 ms	9	Abstandsbeobachter
`T5_AltitudeObserver`	3 ms	8	Höhenbeobachter
`T6_DSP`	9 ms	11	Sensorverarbeitung
`AT1_Remote`	9 ms	2	Fernbedienung

Die Prioritäten wurden nach der Ausführungsreihenfolge des Regelsystems vergeben, je höher die Nummer der Priorität, desto höher ist die Priorität [AUT17c]. `AT1_Remote` wird nur in Variante 2 genutzt, um durch eine ISR einen asynchronen Kontrollfluss abzubilden. Die in Tabelle für `AT1_Remote` angegebene Periode stellt die minimale Zwischenankunftszeit des asynchronen Kontrollflusses dar. ISRs können die gleiche Priorität wie ein Task in AUTOSAR besitzen. In allen anderen Varianten ist die Funktionalität der Fernbedienung in `T6_DSP` abgebildet.

Allen Varianten ist gemein, dass zwei Kalibrierungsvariablen (`VI_Gain_Value`, `x_axis__VI_Gain`) und eine Messvariable (`x_axis__VI_Integrator`) über eine A2L-Datei spezifiziert werden (siehe Kapitel 3.4.3).

5.3.2 Variante 1: Zwei-Kernsystem

In Abbildung 5.3 ist die Verteilung der in Tabelle 5.2 beschriebenen Kontrollflüsse auf OS-Applications und Kerne für die erste Variante dargestellt. Ziel dieser Variante ist es, ein möglichst einfaches Kontrollflusssystem zu modellieren. Es existieren nur periodische Einplanungseinheiten, also sind alle Kontrollflüsse in AUTOSAR OS als Tasks abgebildet. Es werden nur zwei der drei Kerne des Aurix TC277 verwendet und die Anzahl der OS-Applications wird gering gehalten. Aus diesem Grund sind vier der Kontrollflüsse jeweils zu zweit in einer OS-Application zusammengefasst.

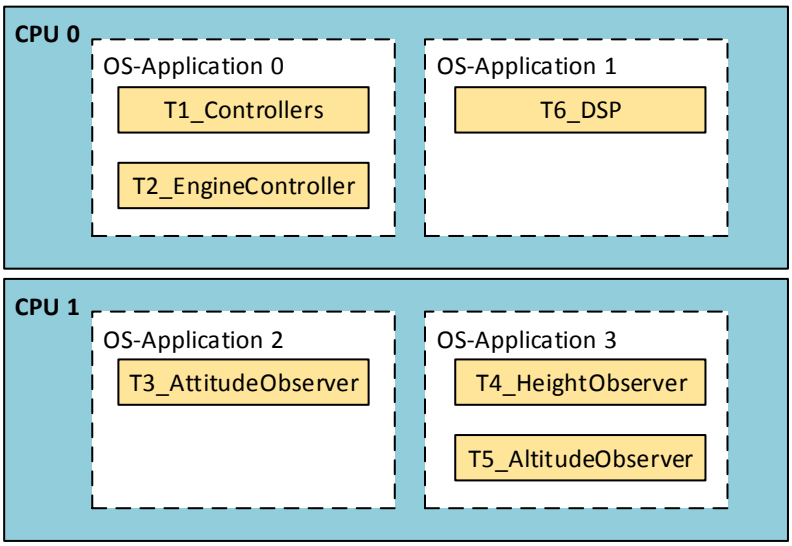


Abbildung 5.3 – Verteilung der Kontrollflüsse auf OS-Applications und Kerne in Variante 1.

5.3.3 Variante 2: Unterbrechungsroutinen

In Abbildung 5.4 ist die Verteilung der in Tabelle 5.2 beschriebenen Kontrollflüsse auf OS-Applications und Kerne für die zweite Variante dargestellt. Ziel dieser Variante ist die Berücksichtigung von

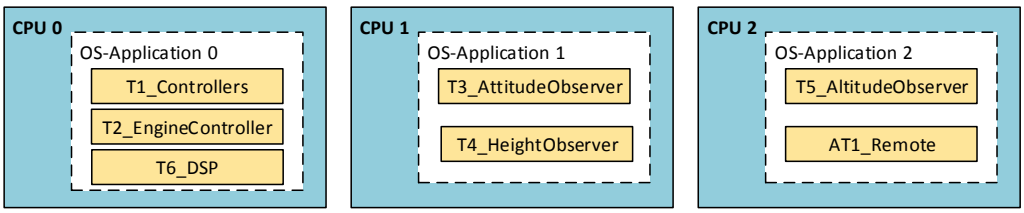


Abbildung 5.4 – Verteilung der Kontrollflüsse auf OS-Applications und Kerne in Variante 2.

asynchronen Kontrollflüssen durch eine zusätzliche ISR (`AT1_Remote`). Außerdem werden alle drei Kerne des Aurix TC277 genutzt. Hierdurch erhöht sich die Anzahl an kernübergreifenden Zugriffen. Gleichzeitig finden in dieser Umsetzung innerhalb eines Kernes keine Zugriffe über OS-Application-Grenzen hinweg statt, da auf jedem Kern nur eine OS-Application gebunden ist.

5.3.4 Variante 3: Platzierungsbedingungen

In Abbildung 5.5 ist die Verteilung der in Tabelle 5.2 beschriebenen Kontrollflüsse auf OS-Applications und Kerne für die dritte Variante dargestellt. Ziel dieser Variante ist es, Platzierungsbedingungen von Daten zwischen Kontrollflüssen zu berücksichtigen. Hierzu wurde zwischen dem Kontrollfluss

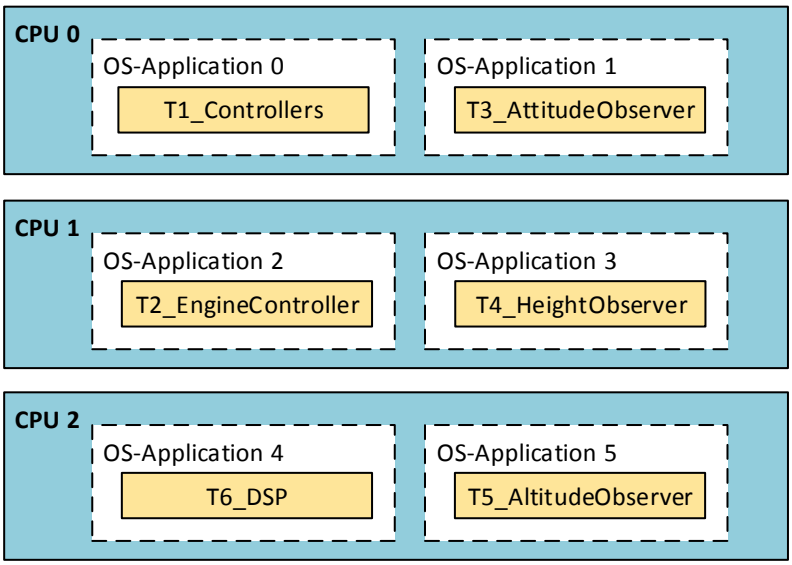


Abbildung 5.5 – Verteilung der Kontrollflüsse auf OS-Applications und Kerne in Variante 3.

`T6_DSP` und jeweils `T3_AttitudeObserver` beziehungsweise `T4_HeightObserver` exklusive Platzierungsbedingungen festgelegt. Gleichzeitig wurden die Kontrollflüsse auf verschiedene Kerne verteilt, sodass die Platzierungsbedingungen auch auf Kontrollflussebene eingehalten werden. In Listing 5.1 ist die Beschreibung der oben genannten Platzierungsbedingungen, in der in Kapitel 4.3.1.1 beschriebenen Syntax, dargestellt. Der Austausch von Daten zwischen diesen Kontrollflüssen kann nur noch über definierte, geteilte Speicherbereiche erfolgen, die aus der Klassifikation bestimmt werden.

Zum Vergleich der Auswirkung von Platzierungsbedingungen wird Variante 4 betrachtet. Diese enspricht Variante 3, berücksichtigt jedoch keine Platzierungsbedingungen.

5.3.5 Aufbau des Integrationscodes

Zur Umsetzung der in Kapitel 5.3.1 bis 5.3.4 beschriebenen Systemvarianten wurde der generierte Code in einer Anwendung integriert. Hierzu werden die Kontrollflüsse für das BS deklariert. Diese Deklarationen gleichen C-Funktionen: der Kontrollfluss wird mit einem Namen deklariert und anschließend in einem Funktionskörper die Operationen des Kontrollflusses definiert. Für die Funktionskomponenten, für die Code generiert wurde (AltitudeController, AltitudeObserver, Attitude-

```
1 OS_App0_Core2 != OS_App1_Core0 // T6_DSP not on same core as ↘
      T3_AttitudeObserver
2 OS_App0_Core2 != OS_App1_Core1 // T6_DSP not on same core as T4_HeightObserver
```

Listing 5.1 – Platzierungsbedingungen im Rahmen von Variante 3.

Controller, AttitudeObserver, EngineController und HeightObserver), erfolgt in ihren Kontrollflüssen das Setzen ihrer Eingänge und der Aufruf der zugehörigen C-Funktion (siehe Abbildung 5.2). Jeder Kontrollfluss folgt damit dem Eingabe-Verarbeitung-Ausgabe (EVA)-Prinzip, da zuerst Eingänge aus Vorgängerprozessen gesetzt werden. Anschließend erfolgt die Verarbeitung der Eingaben durch die C-Funktionen, die abschließend die Ausgänge setzen.

Die Signalproduzenten (Sensorverarbeitung und Fernbedienung aus Abbildung 5.1) werden ebenfalls als Kontrollflüsse deklariert. Ihre Funktionskörper enthalten aber nur ein Setzen ihrer Ausgänge mit zufälligen Werten, um so die Sensor- und Eingabewerte zu simulieren. Für die Evaluation wurde nicht auf die Plausibilität der Werte geachtet, sondern nur auf die Korrektheit der Datentypen, da nur die allgemeine Funktionsweise und nicht die konkrete Anwendung des *I4Copters* modelliert werden soll. In den Varianten 1, 3 und 4 werden alle Ausgangswerte im Kontrollfluss `T6_DSP` generiert, das beinhaltet die aus der Komponente Fernbedienung stammenden Werte. In Variante 2 wird zusätzlich eine ISR deklariert (`AT1_Remote`), die wie in Kapitel 5.3.3 beschrieben, als asynchroner Kontrollfluss dient. In diesem werden die Ausgangssignale der Fernbedienung ebenfalls durch zufällige Werte gesetzt, anstatt wie vorher im `T6_DSP` Kontrollfluss.

Neben den eigentlichen Kontrollflüssen werden ebenfalls die Leerlaufkontrollflüsse der Kerne sowie Systemzähler-Kontrollflüsse deklariert.

5.4 Durchführung der Astrée Datenflussanalyse

Um die Ergebnisse der Astrée Datenflussanalyse zu nutzen, müssen erst alle gefundenen Fehler der Kategorien (1) bis (8) (siehe Kapitel 2.3.3.3) beseitigt werden. Die Analyse des *I4Copters*-Systems ergab eine Vielzahl von Variablenüberläufen (2) und eine Division durch Null (5) (siehe Abbildung 5.6). Diese können auf die Funktion `Tab1DS3I2T3126` zurückgeführt werden. Die Funktion

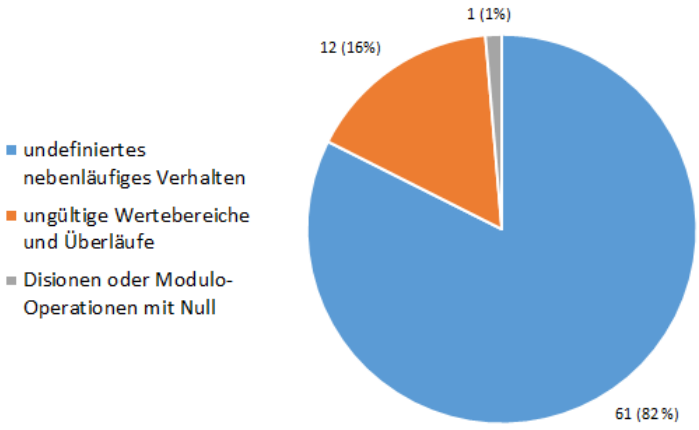


Abbildung 5.6 – Verteilung der Fehler in der Astrée Analyse.

wird automatisch von TL für Look-Up-Tabellen mit Interpolation generiert. Das hierzu verwendete

Konstrukt konnte durch Astrée nicht mit genügend hoher Präzision analysiert werden, sodass es zu einem Fehlalarm kam. Durch Umstrukturierung der Funktion konnte die Präzision erhöht werden und bei einer anschließenden Analyse wurden nur noch Fehler der Kategorie (9) gefunden. Dies hat den Grund, dass keine Synchronisationsmechanismen im System verwendet werden und so Wettlaufsituationen entstehen können. Diese haben allerdings keinen Einfluss auf das von Astrée ermittelte Zugriffsverhalten.

Wird eine AUTOSAR Konfiguration mit Alarmen verwendet, verliert die Analyse auf Grund einer unzureichenden Implementierung der Analyse an Präzision. Die Analyse ist zwar vollständig und korrekt, die verwendete abstrakte Semantik ist jedoch nicht präzise genug. Hierdurch entstehen, im Vergleich zu der vorhergehenden und fehlerfreien Analyse, 155 neue Alarme, die Überläufe (2) melden. Diese Alarme stellen als Resultat des Präzisionsverlustes Fehlalarme dar.

In der Datenflussanalyse wurden von Astrée 206 Daten in den Varianten 1, 3 und 4 erfasst, auf die insgesamt 613-mal zugegriffen wird. In der Datenflussanalyse der Variante 2 wurden 207 Variablen erfasst, da eine zusätzliche Variable für eine ISR hinzukommt. Auf die Daten von Variante 2 wird ebenfalls 613-mal zugegriffen. Diese Ergebnisse werden, wie in Kapitel 4.3.1.2 beschrieben, in einem kommaseparierten Bericht ausgeleitet und dem MMHT zugeführt. Anschließend werden die Daten in einem Durchlauf des MMHTs klassifiziert.

5.5 Auswertung der Ergebnisse

Ohne Linker-Skripte können die Ergebnisse der Datenklassifizierung nicht in einem lauffähigen System bewertet werden. Im Folgenden wird aus diesem Grund eine qualitative, statische Betrachtung der Ergebnisse der Datenklassifizieriung durchgeführt. Zuerst wird die bestimmte Ausführfrequenz der Kontrollflüsse überprüft. Anschließend werden die Ergebnisse der Klassifizierung allgemein für die verschiedenen Varianten beschrieben. Es wird auf Besonderheiten der Varianten und Unterschiede zwischen diesen eingegangen. Die Einhaltung der Platzierungsbedingungen in Variante 3 wird betrachtet, indem ein Vergleich mit Variante 4 gezogen wird. Abschließend wird für eine Beispielmenge an Daten die Klassifizierung nachvollzogen und diskutiert.

Die hier präsentierten Ergebnisse wurde in einem Jupyter Notebook aufbereitet, das unter [Brä18a] eingesehen werden kann.

5.5.1 Überprüfung der bestimmten Aufruffrequenz

In Tabelle 5.3 sind die vom MMHT bestimmten Aufruffrequenzen der einzelnen Kontrollflüsse dargestellt. Ein Vergleich mit den tatsächlichen Perioden (siehe Tabelle 5.1) zeigt, dass die Ausführfrequenzen wie in (4.1) beschrieben:

$$f_{\text{Ausführung},i} = \frac{P_i}{H}$$

richtig bestimmt wurden.

5.5.2 Vergleich der Varianten

Das MMHT hat alle 206 (Variante 1, 2 und 3)/207 (Variante 2)von Astrée erfassten Daten klassifiziert (siehe Kapitel 5.4). Von diesen konnte für 155 eine Repräsentation im DD in Form eines DD-Variablenobjekts gefunden werden. Die 51/52 Daten, die keinem DD-Variablenobjekt zugeordnet werden konnten, fallen in die folgenden Kategorien:

Tabelle 5.3 – Im Zuge der MMHT-Überprüfung betrachtete Daten.

Kontrollfluss	Ausführfrequenz in der Hyperperiode
`T1_Controllers`	1
`T2_EngineController`	1
`T3_AttitudeObserver`	3
`T4_HeightObserver`	3
`T5_AltitudeObserver`	3
`T6_Controllers`	1
`AT1_Remote`	1

- Daten aus den Modulen Fernbedienung und Sensorverarbeitung, für die kein Code generiert wurde: 29.
- Daten für die auf Grund von Limitierungen von TargetLink keine DD-Variablenobjekte angelegt werden konnten: 18.
- Daten aus dem Integrationscode beziehungsweise dem Astrée BS-Stub: 4/5.

Diese Daten werden durch das MMHT klassifiziert und es werden ihnen Sektionsnamen zugewiesen. Sie werden jedoch bei der Code-Generiung mit TL nicht berücksichtigt.

5.5.2.1 Verteilung der Klassen

In Abbildung 5.7 ist die Verteilung der Klassen innerhalb der verschiedenen Varianten dargestellt. Aus der Abbildung ist ersichtlich, dass alle Daten in allen Varianten klassifiziert werden konnten, da die Klasse "Keine Klasse"nicht vergeben wurde. Speicher-globale Klassen wurden nicht vergeben, da, wie in Kapitel 4.3.2 beschrieben, für die Verteilung auf globale und lokale Speicher weitere Informationen über den Speicherzustand bekannt sein müssen. Die in Abbildung 5.7 dargestellte Verteilung spiegelt das Szenario wider, dass alle Daten auf kernnahen Speichern platziert werden können. Aus Abbildung 5.7 ist ebenfalls ersichtlich, dass es, bei gleicher Spezifikation von Sonderdaten wie Kalibrier- und Messvariablen, keine Unterschiede in der Klassifizierung dieser Sonderdaten gibt. So sind in allen Varianten je zwei Kalibrier- und eine Messvariable klassifiziert worden.

Auch ist ersichtlich, dass mit 129 beziehungsweise 130 Daten ein Großteil der Daten der Klasse *Prozessor n, Kontrollfluss-lokal, Speicher-lokal* zugewiesen werden. Nur die global definierten Austauschvariablen werden als *Prozessor n, Kontrollfluss-global, Speicher-lokal* beziehungsweise *Kontrollfluss-global, Speicher-lokal* klassifiziert, da sie Daten zwischen verschiedenen Kontrollflüssen austauschen. Die Summe der Daten beider Klassen ist in allen Varianten, außer Variante 2, gleich. Die Abweichung in Variante 2 stammt aus der Klassifizierung des Datums `i`, das in der Sensor-Stub-Funktion als Zählvariable verwendet wird, welche in Variante 2 durch zwei Kontrollflüsse aufgerufen wird. Dementsprechend ist sie nicht mehr *Prozessor n, Kontrollfluss-lokal, Speicher-lokal* sondern *Kontrollfluss-global, Speicher-lokal*.

5.5.2.2 Verteilung der Daten auf die Kontrollflüsse

Betrachtet man die Verteilung der Daten auf die einzelnen Kontrollflüsse innerhalb der Varianten, zeigen sich minimale Unterschiede (siehe Abbildung 5.8). Die Unterschiede zwischen Variante 1 beziehungsweise 2 und 3 beziehungsweise 4 in Kontrollfluss `T2_EngineController` stammen aus der Zuordnung des Besitzers eines Datums. Astrée erzeugt zur Verwaltung von Kontrollflusssystemen

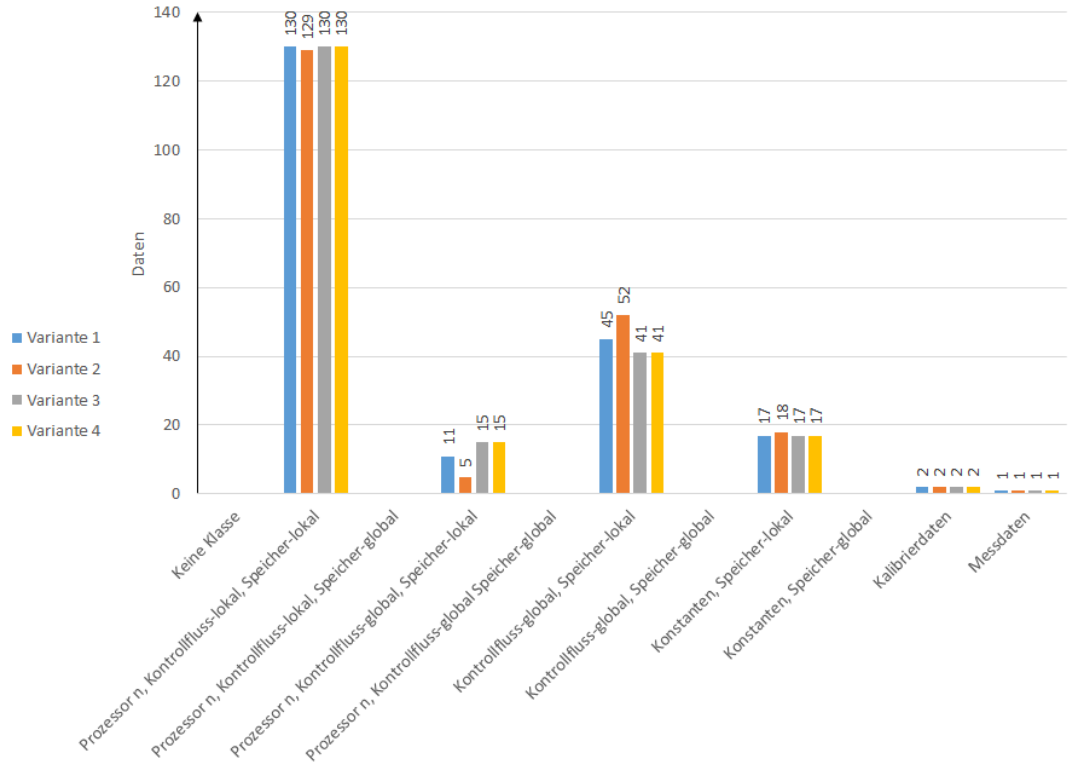


Abbildung 5.7 – Verteilung der Datenklassen innerhalb der Varianten.

eigene Variablen, je nach Allokation der Kontrollflüsse in der Betriebssystemkonfiguration ändert sich der Zugriff auf diese Variablen. In diesem Fall werden die Variablen `__ASTREE_Task` und `OSError_params` statt `T2_EngineController` in Variante 1 dem Kontrollfluss `T6_DSP` in Variante 3 und 4 zugeordnet. In Variante 2 erfolgt die Zuordnung zu Kontrollfluss `T4_HeightObserver`.

Die geringere Anzahl an Daten, die in Variante 2 dem Kontrollfluss `T6_DSP` zugeordnet werden, ist der Zuordnung dieser Daten zur ISR `AT1_Remote` zuzuschreiben. Zusätzlich wird, wie in Kapitel 5.4 beschrieben, eine zusätzliche Variable von Astrée zur Verwaltung der asynchronen Kontrollflüsse angelegt. Ansonsten ist die Zuteilung der Daten auf die Kontrollflüsse konstant, wie es auf Grund des gleichen Integrationscodes auch zu erwarten ist.

5.5.2.3 Verteilung der Daten auf die Kerne

Betrachtet man die Verteilung der Daten auf den einzelnen Kernen werden die Unterschiede in der Bindung von OS-Applications deutlich. Die Verteilung von Daten auf die einzelnen Kerne, für die verschiedenen Varianten ist in Abbildung 5.9 dargestellt. Diese Zuteilung folgt zum größten Teil aus der Bindung der OS-Applications auf die Kerne. Wie aus Abbildung 5.7 ersichtlich sind in Variante 1 141, in Variante 2 134 und in Variante 3 oder 4 145 der Daten als *Prozessor n* klassifiziert worden. Diese werden entweder auf dem der OS-Application zugehörigen Kernspeicher oder im globalen Speicher platziert. Nur bei Prozessor-globalen und Kontrolfluss-globalen Daten werden Daten auf anderen Kernen platziert. Dies ist der Fall, wenn durch diese öfter auf das jeweilige Datum

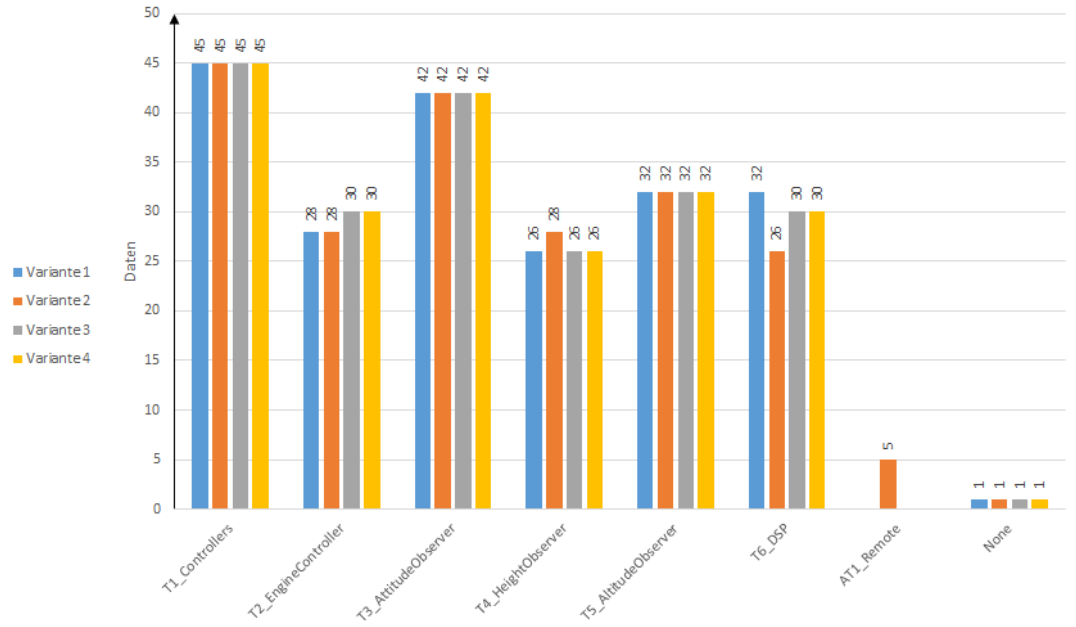


Abbildung 5.8 – Verteilung der Daten auf die Kontrollflüsse innerhalb der Varianten.

zugegriffen wird, als durch den Kern, auf dem der besitzende Kontrollfluss gebunden ist. Dies ist, je nach Variante, nur für 41 bis 52 der Daten der Fall.

In Abbildung 5.10 ist die Anzahl der Daten je Variante dargestellt, die auf einen anderen Kern als ihre OS-Application gebunden wurden. Da in Variante 1 nur zwei Prozessorkerne verwendet werden, existieren entsprechend weniger Möglichkeiten, wie die OS-Applications und damit deren Daten auf die kernnahen Speicher verteilt werden. In Variante 2 herrscht hierbei die größte Freiheit, da OS-Applications auf alle Prozessorkerne gebunden sind; gleichzeitig bestehen keine Platzierungsbedingungen wie bei Variante 3, die die Verteilung der Daten einschränken. Die höhere Anzahl an umverteilten Daten in den Variante 1 und 2 ist jedoch in der Platzierung des Kontrollflusses `T6_DSP` begründet. 19 der 30 erfassten Daten von `T6_DSP` werden mit den Kontrollflüssen `T3_AttitudeObserver`, `T4_HeightObserver` und `T5_AltitudeObserver` geteilt; elf dieser 19 werden alleine mit `T3_AttitudeObserver` geteilt. Ebenso werden diese drei Kontrollflüsse mit einer Periode von 3 ms ausgeführt, woraus sich eine höhere Zugriffsfrequenz für diese Daten ergibt, als für `T6_DSP`, der nur alle 9 ms ausgeführt wird. Diese drei Kontrollflüsse sind in Variante 1 und Variante 2 jedoch auf andere Kerne gebunden als der Kontrollfluss zur Sensorverarbeitung, weshalb in beiden Varianten alle geteilten Daten des `T6_DSP`-Kontrollflusses auf andere Kerne gebunden werden. In den Varianten 3 und 4 wird `T6_DSP` auf den gleichen Kern gebunden wie `T5_AltitudeObserver`, weshalb die Anzahl der umplatzierten Daten geringer ist.

Aus Abbildung 5.10 ist der Einfluss der Platzierungsbedingungen erkennbar. Diese verhindern, dass Daten von `T6_DSP` auf den Kernen 0 und 1 platziert werden, da dort die Kontrollflüsse `T3_AttitudeObserver` beziehungsweise `T4_HeightObserver` gebunden sind. Dementsprechend werden die Daten von `T6_DSP` auf Kern 2 platziert, was dazu führt, dass in Variante 3 nur sechs Daten, statt wie in Variante 4 22 Daten, auf andere Kerne verschoben werden. In Kapitel 5.5.3 wird näher auf die Platzierungsbedingungen von Variante 3 eingegangen.

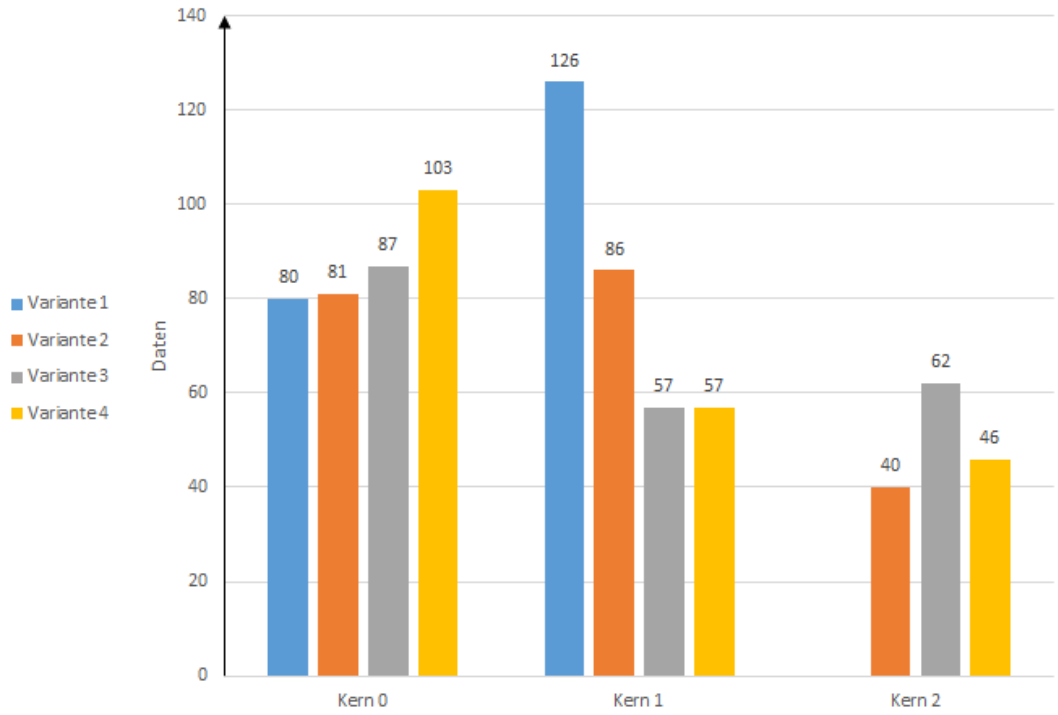


Abbildung 5.9 – Verteilung der Daten auf die Kerne innerhalb der Varianten.

5.5.2.4 Resultierende Sektionen

Auf Basis der Klassifizierung werden, wie in Kapitel 4.3.3 beschrieben, Sektionen zugewiesen, die zur Bindung und dem Aufspannen von Speicherschutzregionen eingesetzt werden können. In Abbildung 5.11 ist die Verteilung dieser Sektionen auf die Kerne für jede Variante dargestellt. Aus der Abbildung ist ersichtlich, dass für die Beispielanwendung auf keinem Prozessorkern mehr als 16 Sektionen verwendet werden. Dies bedeutet, dass während der Laufzeit keine Umkonfiguration der MPU stattfinden muss, wenn für jede Sektion eine Speicherschutzregion aufgespannt wird (siehe Kapitel 2.3.1.3). Die Anzahl der Sektionen bestimmt sich daraus, wie viele Klassen pro Kontrollfluss verwendet werden. Die maximale Anzahl an Sektionen bestimmt sich also aus dem Produkt der Anzahl der Klassen und der Anzahl der Kontrollflüsse:

$$\text{maximale Anzahl Sektionen} = \text{Anzahl Kontrollflüsse} \cdot \text{Anzahl Variablenklassen} \tag{5.1}$$

Im Beispiel des *I4Copters* werden in den Varianten 1, 3 und 4 jeweils sechs Kontrollflüsse verwendet, in der Variante 2 sieben. Wie in Kapitel 3.4.4 werden zwölf Klassen verwendet. Zwei dieser Klassen, *Stack-Daten* und *Optimierbar*, werden jedoch nicht vergeben. Deshalb ergeben sich für die Varianten 1, 3 und 4 maximal 60, für die Variante 2 maximal 70 mögliche Sektionen. Des Weiteren muss berücksichtigt werden, dass die Beispielanwendung mit maximal sieben Kontrollflüssen relativ klein ist. Beim *elektromechanischen Wankstabilisator* der Firma Schaeffler [FHP17] werden die Softwarekomponenten auf 17 Kontrollflüsse abgebildet. Dies entspricht fast einer Verdreifachung im Vergleich zum *I4Copter*. Mehr Kontrollflüsse bedeuten, dass wesentlich mehr Sektionen erzeugt und darauf aufbauend genauso viele Speicherschutzregionen aufgespannt werden können. Hierdurch wird das Umkonfigurieren der MPU bei Kontextwechseln durch das BS erforderlich. Es muss also

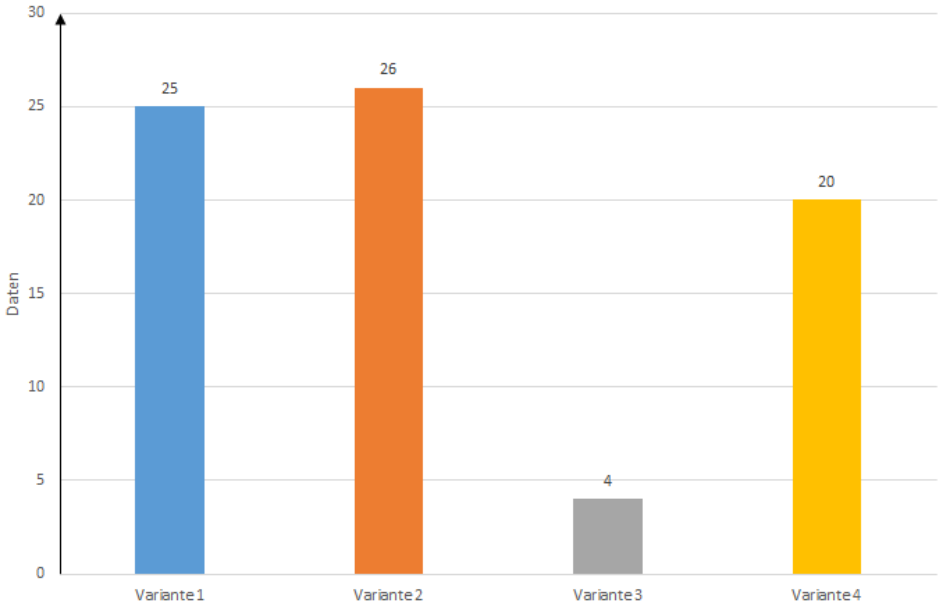


Abbildung 5.10 – Anzahl der Daten, die auf einen anderen Kern als ihre OS-Application gebunden wurden.

beim Aufspannen von Speicherschutzregionen abgewogen werden, welche Sektionen tatsächlich durch die MPU geschützt und die dadurch entstehenden Laufzeitkosten in Kauf genommen werden müssen.

5.5.3 Einhaltung der Platzierungsbedingungen in Variante 3

In Variante 3 (siehe Kapitel 5.3.4) werden Platzierungsbedingungen verwendet, um zusätzlich zur räumlichen Isolation durch die Speicherschutzregionen die Daten auf getrennten kernnahen Speichern abzulegen. Die festgelegten Datenplatzierungsbedingungen werden auf Ebene OS-Applications beschrieben, da diese auch die Platzierung der Kontrollflüsse auf den Kernen beschreiben. Den Platzierungsbedingungen unterliegen 98 Daten. Für alle Daten insgesamt ergeben sich für die zwei exklusiven Platzierungsbedingungen aus Listing 5.1 128 Bedingungen auf Datumsebene, die eingehalten werden müssen. 60 dieser Datumsbedingungen stammen aus OS-Application 4, die den Kontrollfluss `T6_DSP` beinhaltet. Dieser hat 30 Daten, die jeweils nicht auf den Kern der OS-Application 1 beziehungsweise der OS-Application 3 platziert werden dürfen. 42 beziehungsweise 26 der Datumsbedingungen stammen aus den OS-Applications 1 beziehungsweise 3. Die Anzahl der Bedingungen entspricht in diesem Fall der Anzahl der Daten, da jedes Datum nur einer Platzierungsbedingung unterliegt. Alle 128 Datumsbedingungen wurden eingehalten, das bedeutet, dass kein Datum auf einem Kern platziert wurde, für den es eine exklusive Bedingung auf OS-Application-Ebene gibt. Die Auswirkung dessen zeigt sich in Abbildung 5.10: In Variante 4 werden 22 Daten auf einen anderen Kern verschoben, in Variante 3 nur 6. Die 16 Daten, die in Variante 3 nicht verschoben werden, sind alle Daten aus dem Kontrollfluss `T6_DSP`, die vorher auf Kern 0 gebunden wurden. Dies ist jedoch auf Grund der Bedingung $\text{Kern}_{\text{OS-Application 4}} \neq \text{Kern}_{\text{OS-Application 3}}$ nicht möglich.

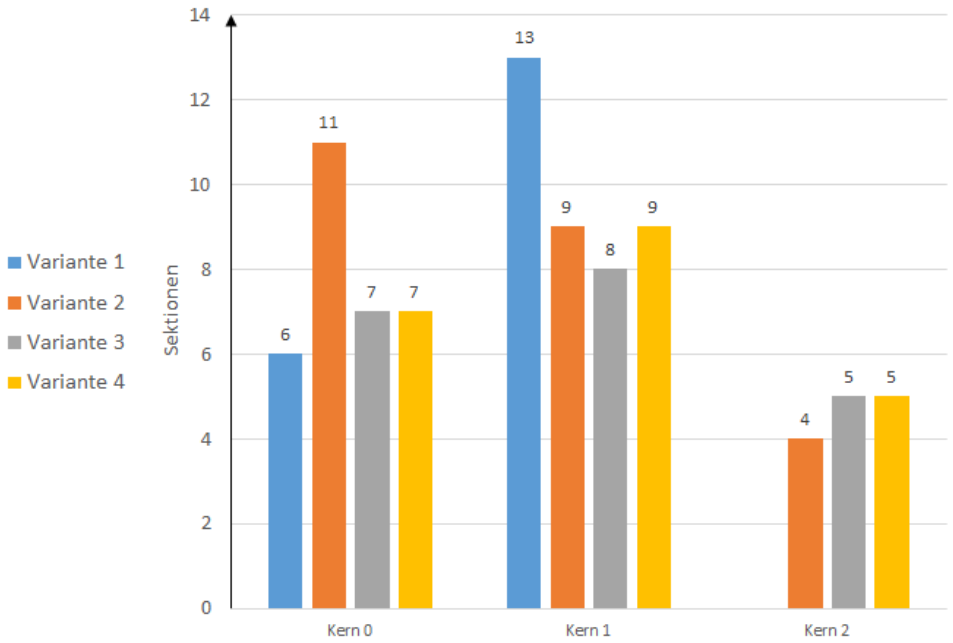


Abbildung 5.11 – Anzahl der Sektionen je Kern für alle Varianten.

5.5.4 Empirische Überprüfung der Klassifizierung

Im Folgenden wird für einzelne Daten beispielhaft die Klassifizierung und die Zuteilung auf die Kerne nachvollzogen. Hierdurch soll das Vorgehen des MMHT anhand von Beispieldaten überprüft werden. In Tabelle 5.4 sind die betrachteten Daten aufgelistet.

Tabelle 5.4 – Im Zuge der MMHT-Überprüfung betrachtete Daten mit dem aus der Anwendung bestimmten besitzenden Kontrollfluss und der zugehörigen OS-Application.

Name	Besitzender Kontrollfluss
`x_axis__VI_Gain`	`T3_AttitudeObserver`
`Out_ThrustZ_g_altCon`	`T1_Controllers`
`Out_thrustZ_N_attCtr`	`T1_Controllers`
`Out_accX_g_dsp_value`	`T6_DSP`
`Sb2_Lookup_Table1_axis`	`T2_EngineController`

Die betrachteten Daten erlauben es, die Pfade des Flussdiagramms in Abbildung 4.4 zu durchlaufen. Dies ermöglicht es, zu überprüfen ob die Klassifizierung der Variablen durch das MMHT wie erwartet erfolgt und ob die Eingangsinformatonen richtig ausgelesen und weiterverarbeitet werden. Im Rahmen der empirischen Überprüfung der Klassifizierung wird davon ausgegangen, dass die Ergebnisse der Astrée Datenflussanalyse vollständig und korrekt sind. Ebenso wird angenommen, dass keine Fehler auf Architekturebene vorhanden sind, die sich auf die Spezifikation des Systems auswirken, welche für die Variablenklassifizierung verwendet wird.

5.5.4.1 Überprüfung der einzelnen Variablenklassifizierungen

`x_axis__VI_Gain` ist der Faktor eines *Gain*-Blocks im Lagebeobacher-Subsystem. Dieser wird im Rahmen dieser Arbeit beispielhaft über eine A2L-Datei als Kalibriervariable markiert. Diese Variablen werden im ersten Schritt der Variablenklassifizierung betrachtet. `x_axis__VI_Gain` wird in allen Varianten korrekt als Kalibriervariable erkannt und bekommt die Klasse *Kalibrierdaten* zugewießen. In der Zugriffsanalyse, die Kalibrierkontrollflüsse nicht berücksichtigt, wird auf sie nur durch einen Kontrollfluss zugegriffen, weshalb sie auf dessen kernnahen Speicher platziert wird, so lange dieser nicht belegt ist. Die Zuordnung des Datums erfolgt korrekt zum besitzenden Kontrollfluss `T3_AttitudeObserver` und der jeweiligen OS-Application innerhalb der Variante. Die Zuteilung der Sektion erfolgt auch korrekt: zum Beispiel in Variante 3 wird es der Sektion `OS_App1_Core1_1_T4_HeightObserver_16` zugewiesen. 16 ist die Nummer, die der Klasse *Kalibrierdaten* zugewiesen ist.

`Out_ThrustZ_g_altCon` ist ein Ausgangswert des Höhenreglers. Dieser wird von der C-Funktion des Höhenregler gesetzt und von der C-Funktion des Lagereglers ausgelesen. Daraus lässt sich ableiten, dass es sich nicht um eine Konstante handelt. Beide C-Funktionen sind dem Kontrollfluss `T1_Controllers` zugewiesen. Da keine Zugriffe durch andere Kontrollflüsse auf das Datum erfolgt, ist es *Prozessor n, Kontrollfluss-lokal*. Es wird dem kernnahen Speicher zugeordnet, so lange dieser nicht belegt ist. In jeder Variante wird er der OS-Application 0 zugeordnet, die auf Kern 0 gebunden ist. Das MMHT klassifiziert auch dieses Datum in allen Varianten korrekt. Die zugewiesene Sektion ist in allen Varianten `OS_App0_Core0_0_T1_Controllers_0`. 0 ist die Ziffer, die der Klasse *Prozessor n, Kontrollfluss-lokal, Speicher-lokal* entspricht.

`Out_ThrustZ_N_attCtr` ist ein Ausgangswert des Lagereglers. Dieser wird von der C-Funktion des Lagereglers gesetzt und vom Motorregler ausgelesen. Der Lageregler ist dem Kontrollfluss `T1_Controllers` und der Motorregler dem Kontrollfluss `T2_EngineController` zugewiesen. Es handelt sich also um ein Datum der Klassifizierung *Kontrollfluss-global*. In den Varianten 1 und 2 sind beide Kontrollflüsse auf den Kern 0 gebunden. Bei diesen Daten handelt es sich also um eine *Prozessor n* Variable, die je nach Speicherzustand auf den lokalen Speicher des Kerns 0 oder auf einen globalen Speicher gebunden wird. Die zugewiesene Sektion heißt `OS_App0_Core0_0_T1_Controllers_2`. Die Ziffer 2 steht für die Klasse *Prozessor n, Kontrollfluss-global, Speicher-lokal*.

In den Varianten 3 und 4 sind beide Kontrollflüsse auf unterschiedliche Kerne gebunden, respektive auf Kern 0 und Kern 1. Damit ergibt sich die Klassifizierung *Prozessor-gobal*. Beide Kontrollflüsse werden mit der selben Periode ausgeführt und greifen nur jeweils einmal auf das Datum zu. In diesem Fall wird das Datum auf den kernnahen Speicher des besitzenden Kontrollflusses platziert, so lange dieser noch nicht belegt ist. Die zugewiesene Sektion ist in den Varianten 3 und 4 `OS_App0_Core0_0_T1_Controllers_6`. Die Ziffer 6 steht für *Kontrollfluss-global, Speicher-lokal*. Die Klassifizierung des Datums durch das MMHT ist also korrekt.

Das Datum `Out_accX_g_dsp_value` ist ein Ausgangswert der Sensorverarbeitung. Dieser wird durch den Kontrollfluss `T6_DSP` geschrieben und durch den Kontrollfluss `T3_AttitudeObserver` gelesen. In allen Varianten sind diese Kontrollflüsse auf unterschiedliche Kerne gebunden, dass Datum ist in jedem Fall *Prozessor-gobal*. Da der Beobachterkontrollfluss mit der Periode 3 ms und der Sensorkontrollfluss mit der Periode 9 ms ausgeführt wird, greift der hochfrequente Kontrollfluss öfter auf das Datum zu, als der besitzende Kontrollfluss T6_DSP. Für eine bessere Laufzeiteffizienz und zur Reduzierung konkurrierender Zugriffe bietet es sich an, das Datum auf den jeweiligen kernnahen Speicher von `T3_AttitudeObserver` zu legen. Dies ist in den Varianten 1,2 und 4 pro-

blemlos möglich, so lange der kernnahe Speicher nicht belegt ist. In Variante 3 ist die Platzierung jedoch durch die Platzierungsbedingung $\text{Kern}_{\text{OS-Application 4}} \neq \text{Kern}_{\text{OS-Application 1}}$ ausgeschlossen. Das Datum muss also entweder im Speicher des Kerns platziert werden, auf dem `T6_DSP` gebunden ist oder auf einen globalen Speicher. In Variante 4 zum Beispiel wird dem Datum die Sektion `OS_App0_Core2_0_T6_DSP_6` zugewiesen. Das Datum wird also auf dem Kern 0, der Platzierung der OS-Application 1, gebunden, anstatt auf Kern 2, der Platzierung von OS-Application 4. In Variante 3 hingegen erfolgt die Zuweisung zu Sektion `OS_App0_Core2_2_T6_DSP_6` und damit zu Kern 2. Die Platzierungsbedingung, die bei Variante 3 vorgegeben ist, wird also eingehalten. Auch die Klassifizierung des Datums durch das MMHT erfolgt korrekt.

`Sb2_Lookup_Table1_axis` ist ein Array, dass die Eingangswerte einer Look-Up-Tabelle beschreibt. Der Zugriff erfolgt nur durch den Kontrollfluss `T2_EngineController`. Es ist also ein Datum der Klasse *Prozessor n, Kontrollfluss-lokal*. Außer seiner Definition wird auf das Datum nur lesend zugegriffen, es handelt sich also um eine Konstante. Dies ist bereits in TL so markiert. Das MMHT sieht diese Zuweisung jedoch nicht, da nur die Ergebnisse der Datenflussanalyse aus Astrée, die AUTOSAR OS Konfiguration, Platzierungsbedingungen und die Spezifikation der Kalibrier- und Messvariablen verwendet wird. Durch das aus der Datenflussanalyse stammende Zugriffsverhalten kann das MMHT jedoch erkennen, dass auf dieses Datum nur lesend zugegriffen wird. Dem Datum wird, je nach Speicherzustand die Klasse *Konstante-lokal* oder *Konstante-global* zugewiesen. Die resultierende Sektion ist zum Beispiel in Variante 2 `OS_App0_Core0_0_T2_EngineController_8`. Die Ziffer 8 steht für die Klasse *Konstante-lokal*.

Wie in Kapitel 4.3.2 beschrieben, wird die Klassifizierung *Speicher-global* in der jetzigen Implementierung nicht vergeben, da hierfür weitere Informationen über den Speicherzustand notwendig sind. Durch die Anpassung der Platzhalterfunktion ist es jedoch möglich, dass alle Daten als *Speicher-global* klassifiziert werden.

5.6 Resümee

Die Betrachtung der fünf in Tabelle 5.4 aufgeführten Daten durchläuft unter Einbezug aller Varianten jeden Pfad des Flussdiagramms in Abbildung 4.4. Aus der Klassifizierung der oben genannten Daten lässt sich deshalb ableiten, dass die Parserstufe des MMHT die genutzen Informationen richtig aus den Informationsquellen ausliest und zusammenführt. Die darauf aufbauende Klassifikation erfolgt nach den in Kapitel 4.3.2 beschriebenen Regeln und liefert die erwarteten Klassifizierungen der Daten. Anhand des, nach der Klassifizierung generierten Codes, ist ersichtlich, dass auch die Zuordnung der Sektionen zu den Daten wie erwartet funktioniert.

6 FAZIT

Abschließend werden im Folgenden die Ergebnisse dieser Arbeit zusammengefasst und ein Überblick der noch offenen Punkte gegeben.

6.1 Ergebnis der Arbeit

Im Zuge der steigenden Komplexität von Software in eingebetteten Systemen und der damit einhergehenden Komplexitätssteigerung der Hardware müssen Wege im Entwicklungsprozess gefunden werden, die diese Komplexität handhabbar machen. Um die Anforderungen sicherheitskritischer, eingebetteter Systeme wie Kosteneffizienz, Echtzeitfähigkeit und Rückwirkungsfreiheit zu erreichen, müssen diese sowohl auf der Architektur- als auch auf der Implementierungsebene beachtet werden. Im Rahmen dieser Arbeit wurde mit dem MMHT ein Werkzeug entwickelt, das diese Anforderungen auf Ebene der Implementierung berücksichtigt und dabei die in Kapitel 1.2 geforderten Ziele erreicht:

Ressourceneffizienz: Durch die Berücksichtigung des Zugriffsverhaltens auf Daten können diese durch das MMHT auf Kernspeicher gebunden werden, durch die im häufigsten Zugriffsfall die geringsten Zugriffszeiten entstehen.

Anpassbarkeit: Durch Austausch der auf Architekturebene erstellten Systemspezifikation in Form einer Betriebssystemkonfiguration können für den selben Anwendungscode unterschiedliche Allokationsszenarien berücksichtigt werden. Diese spiegeln sich in der erzeugten Bindung von Daten wider.

Unterstützung des Anwendungsentwicklers: Durch die Automatisierung der Bindungserstellung sowie der Beeinflussung der Code-Generierung werden diese Aufgaben für den Anwendungsentwickler vereinfacht. Durch die Nutzung eines plattformbasierten Entwicklungsprozesses kann auf Hardwareebene ein prozessorfamiliengültiges Regelwerk zur Klassifizierung und Bindung von Daten verwendet werden. Bei einer plattformbasierten Softwareentwicklung kann anwendungsspezifisches Wissen erst im Code-Generierungsprozess eingebracht werden und das Modell anwendungsagnostisch gehalten werden. Wird einer oder beide dieser Ansätze verfolgt, steigt die Wiederverwertbarkeit der Entwicklungsergebnisse und erleichtert so den Entwicklungsprozess.

Vorhersagbarkeit: Durch die Bevorzugung von Speichern mit vorhersagbaren Zugriffsverhalten und der Reduzierung von Zugriffen über die Systembusse durch die Bindung, kann die Präzision von Laufzeitanalysen erhöht werden.

Rückwirkungsfreiheit: Die erzeugte Bindung und die hierzu verwendeten Speichersektionen können in der AUTOSAR OS Konfiguration dazu genutzt werden, Speicherschutzregionen mit definierten Zugriffsberechtigungen zu erzeugen und so räumliche Rückwirkungsfreiheit zwischen den Funktionseinheiten herstellen. Durch den Einsatz statischer Analyse kann zudem Typ- und Speichersicherheit auf Datumsebene erzeugt werden.

Das MMHT erfüllt also die an es gestellten Anforderungen und stellt damit eine Möglichkeit dar, die Komplexität des Entwicklungsprozesses für sicherheitskritische, eingebettete Systeme im Automobilbereich handhabbarer zu gestalten.

6.2 Offene Punkte

Das MMHT ist in der Lage, auf Basis der Variablenklassifikation Sektionen für die Bindung von Daten im Speicher eines Mehrkernprozessors zu erzeugen, dies kann jedoch noch ausgebaut werden. In einem späteren Schritt sollen diese Sektionen automatisch in einem Linker-Skript auf die Speicher gebunden werden, um so ein lauffähiges Programm aus der klassifizierten Anwendung erstellen zu können. Dies ermöglicht außerdem eine, über die Evaluation in Kapitel 5 durchgeführte hinausgehende Bewertung des Klassifizierungsergebnisses und seines Einflusses auf das lauffähige Programm. Die Möglichkeiten der Klassifizierung sollen außerdem erweitert werden, sodass die Klasse *Optimierbar* zuverlässig Daten zugewiesen werden kann. Die Zustände der Speicher müssen für eine korrekte Zuteilung auf lokale und globale Speicher berücksichtigt werden. Dies ist in der aktuellen Implementierung nur durch eine Stub-Funktion umgesetzt. Der Aufbau der Speicher an sich kann ebenfalls berücksichtigt werden, um die Anzahl konkurrierender Zugriffe zu reduzieren. Die Möglichkeit, bestimmte Daten wie Konstanten zu cachen wird noch nicht genutzt, um die Laufzeiteffizienz zu verbessern. Des Weiteren kann die Zuordnung von Daten auf besitzende Kontrollflüsse dahingehend erweitert werden, sodass diese nicht von den Gegebenheiten der Implementierung abhängt. Als letzten Punkt soll die Zuteilung der Daten auf die Speicher durch ein Allokationswerkzeug wie ASSIST validiert werden. Hierzu kann der Speicherverbrauch, der aus der erzeugten Datenbindung entsteht, als Randbedingung in einem zweiten Durchlauf von ASSIST eingebunden werden. Dies ermöglicht es zu überprüfen, ob die vorher getroffenen Allokationsentscheidungen weiterhin gültig sind.

ABKÜRZUNGSVERZEICHNIS

AMP	Asymmetric Multi-Processing
ASSIST	Architecture Synthesis for Safety-Critical Systems Tool
ASIL	Automotive Integrity Safety Level
API	Programmierschnittstelle (Application Programming Interface)
BS	Betriebssystem
BSW	Basissoftware
DD	Data Dictionary
DFG	Datenflussgraph
DSL	domänenspezifische Sprache (Domain Specific Language)
DSPR	Data Scratchpad RAM
DMI	Datenspeicherschnittstelle (Data Memory Interface)
EVA	Eingabe-Verarbeitung-Ausgabe
ISR	Unterbrechungsroutine (Interrupt Service Routine)
LMU	lokalen Speichereinheit (Local Memory Unit)
MMHT	Magic Memory Handling Tool
MMU	Speicherverwaltungseinheit (Memory Management Unit)
MPU	Speicherschutzeinheit (Memory Protection Unit)
OEM	Automobilhersteller (Original Equipment Manufacturer)
PMI	Programmspeicherschnittstelle (Program Memory Interface)
PMU	Programmspeichereinheit (Program Memory Unit)
PSPR	Program Scratchpad RAM
RMS	Ratenmonotone Ablaufplanung (Rate Monotonic Scheduling)
SPM	Scratchpad-Speicher (Scratchpad Memory)

TCB	Trusted Code Base
TL	TargetLink
WCET	maximale Ausführungszeit (Worst Case Execution Time)
WOET	maximal beobachtete Ausführungszeit (Worst Observed Execution Time)

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

QUELLCODEVERZEICHNIS

LITERATUR

[Abs] *Safety Manual for aiT, Astrée, RuleChecker, StackAnalyzer*. AbsInt Angewandte Informatik GmbH. Mai 2018.

[ABS02] Oren Avissar, Rajeev Barua und Dave Stewart. "An optimal memory allocation scheme for scratch-pad-based embedded systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 1.1 (2002), S. 6–26.

[Ada] *ISO/IEC 8652:2012(E) with COR.1:2016 — Ada Reference Manual*. International Organization for Standardization, 2016.

[Aho+06] Alfred V. Aho u. a. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Aik+06] Mark Aiken u. a. "Deconstructing process isolation". In: *Proceedings of the 2006 workshop on Memory system performance and correctness*. ACM. 2006, S. 1–10.

[Arm] *Memory Protection Unit (MPU)*. 1.0. ARM. Juli 2016. URL: https://static.docs.arm.com/100699/0100/armv8m_architecture_memory_protection_unit_100699_0100_00_en.pdf.

[Ast] *a³ for C - User Documentation*. 18.04. AbsInt Angewandte Informatik GmbH. Mai 2018.

[AUT17a] AUTOSAR. *ARXML Serialization Rules*. Specification. 2017.

[AUT17b] AUTOSAR. *AUTOSAR - Specification of Memory Mapping*. Specification. 2017.

[AUT17c] AUTOSAR. *AUTOSAR - Specification of Operating System*. Specification. 2017.

[Bol+] Greg Bollella u. a. *The Real-Time Specification for Java*. The Real-Time for Java Expert Group, The Reference Implementation Team.

[Brä18a] Felix Bräunling. *evaluation Repository*. 2018. URL: https://gitlab.cs.fau.de/master_thesis/evaluation.

[Brä18b] Felix Bräunling. *I4Copter ControlModel Repository - TargetLink Branch*. 2018. URL: https://gitlab.cs.fau.de/i4copter/ControlModel/tree/targetlink.

[Brä18c] Felix Bräunling. *I4Copter System Repository*. 2018. URL: https://gitlab.cs.fau.de/master_thesis/scripts.

[Brä18d] Felix Bräunling. *scripts Repository*. 2018. URL: https://gitlab.cs.fau.de/master_thesis/I4Copter_System.

[C90] *ISO/IEC 9899:1990*. International Organization for Standardization, 1996.

[C99] *ISO/IEC 9899:1999*. International Organization for Standardization, 2007.

[Cou05] P. Cousot. "Abstract interpretation". MIT course 16.399, http://web.mit.edu/16.399/www/. 2005.

[Cul+10] Christoph Cullmann u. a. “Predictability considerations in the design of multi-core embedded systems”. In: *Proceedings of Embedded Real Time Software and Systems* (2010), S. 36–42.

[Dam+06] W. Damm u. a. “Mapping Task-Graphs on Distributed ECU Networks: Efficient Algorithms for Feasibility and Optimality”. In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. 2006, S. 87–90. DOI: 10.1109/RTCSA.2006.42.

[DNSV10] Marco Di Natale und Alberto Luigi Sangiovanni-Vincentelli. “Moving from federated to integrated architectures in automotive: The role of standards, methods and tools”. In: *Proceedings of the IEEE* 98.4 (2010), S. 603–620.

[DW04] N.B. Dale und C. Weems. *Programming in C++*. Jones und Bartlett Publishers, 2004. ISBN: 9780763732349. URL: https://books.google.de/books?id=mxZBPSjSEYUC.

[FHP17] Martin Fritz, Tobias Hillenbrand und Thomas Pfund. “48-V-Technologien im Fahrzeug”. In: *ATZextra* 22.1 (2017), S. 28–33.

[Fuc10] R. Fuchsen. “How to address certification for multi-core based IMA platforms: Current status and potential solutions”. In: *29th Digital Avionics Systems Conference*. 2010, 5.E.3–1–5.E.3–11. DOI: 10.1109/DASC.2010.5655461.

[Föl16] O. Föllinger. *Regelungstechnik: Einführung in die Methoden und ihre Anwendung*. Lehrbuch Studium. Vde Verlag GmbH, 2016. ISBN: 9783800742011. URL: https://books.google.de/books?id=ZDcHkAEACAAJ.

[Gos+18] James Gosling u. a. *The Java Language Specification - Java SE 10 Edition*. Oracle America, 2018.

[Gri03] K. Grimm. “Software technology in an automotive company - major challenges”. In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, S. 498–503. DOI: 10.1109/ICSE.2003.1201228.

[Gro+] Dan Grossman u. a. *Cyclone*. URL: https://cyclone.thelanguage.org/wiki/.

[Gro+02] Dan Grossman u. a. “Region-based memory management in Cyclone”. In: *ACM Sigplan Notices* 37.5 (2002), S. 282–293.

[Gro+05] Dan Grossman u. a. “Cyclone: A Type-Safe Dialect of C”. In: *C++ User's Journal* (2005).

[Guo+11] Y. Guo u. a. “Optimal Data Allocation for Scratch-Pad Memory on Embedded Multi-core Systems”. In: *2011 International Conference on Parallel Processing*. 2011, S. 464–471. DOI: 10.1109/ICPP.2011.79.

[GWM92] Anoop Gupta, Wolf-Dietrich Weber und Todd Mowry. “Reducing memory and traffic requirements for scalable directory-based cache coherence schemes”. In: *Scalable shared memory multiprocessors*. Springer, 1992, S. 167–192.

[HB18] Robert Hilbrich und Michael Behrisch. “Improving the Efficiency of Dislocality Constraints for an Automated Software Mapping in Safety-Critical Systems”. In: *SE'18: Software Engineering-Tagung der Gesellschaft für Informatik (GI)*. 2018. URL: https://elib.dlr.de/117758/.

[Hil18] Robert Hilbrich. *Architecture Synthesis for Safety-Critical Systems (ASSIST)*. Juli 2018. URL: https://github.com/RobertHilbrich/assist-public.

[Isoa] *ISO 26262-1:2011 Road vehicles – Functional safety – Part 1: Vocabulary*. International Organization for Standardization, 2011.

[Isob] *ISO 26262-3:2011 Road vehicles – Functional safety – Part 3: Concept Phase*. International Organization for Standardization, 2011.

[Isoc] *ISO 26262-6:2009 Road vehicles – Functional safety – Part 6: Product development at the software level*. International Organization for Standardization, 2009.

[Isod] *ISO 26262-6:2011 Road vehicles – Functional safety – Part 6: Product development at the software level*. International Organization for Standardization, 2011.

[Kan+01] M. Kandemir u. a. "Dynamic Management of Scratch-pad Memory Space". In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, S. 690–695. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379049. URL: http://doi.acm.org/10.1145/378239.379049.

[Keß18] Sebastian Keßler. "Jittermessungen am TC265". Dez. 2018.

[Kop+07] H. Kopetz u. a. "Automotive Software Development for a Multi-Core System-on-a-Chip". In: *Fourth International Workshop on Software Engineering for Automotive Systems (SEAS '07)*. 2007, S. 2–2. DOI: 10.1109/SEAS.2007.2.

[Lev99] John R Levine. *Linkers & Loaders. 1999*. San Francisco: Morgan Kaufmann Publishers, 1999.

[Liu00] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN: 9780130996510.

[MMB03] Aleksandar Milenkovic, Milena Milenkovic und Nelson Barnes. "A performance evaluation of memory hierarchy in embedded systems". In: *System Theory, 2003. Proceedings of the 35th Southeastern Symposium on*. IEEE. 2003, S. 427–431.

[Mö10] J. Mössinger. "Software in Automotive Systems". In: *IEEE Software* 27.2 (2010), S. 92–94. ISSN: 0740-7459. DOI: 10.1109/MS.2010.55.

[Nec+05] George C. Necula u. a. "CCured: Type-safe Retrofitting of Legacy Software". In: *ACM Trans. Program. Lang. Syst.* 27.3 (Mai 2005), S. 477–526. ISSN: 0164-0925. DOI: 10.1145/1065887.1065892. URL: http://doi.acm.org/10.1145/1065887.1065892.

[Our15] Alain Ourghanlian. "Evaluation of static analysis tools used to assess software important to nuclear power plant safety". In: *Nuclear Engineering and Technology* 47.2 (2015). Special Issue on ISOFIC/ISSNP2014, S. 212 –218. ISSN: 1738-5733. DOI: https://doi.org/10.1016/j.net.2014.12.009. URL: http://www.sciencedirect.com/science/article/pii/S1738573315000091.

[Per+13] Jon Perez u. a. "A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning". In: *Proc. 1st WMC, RTSS* (2013), S. 25–30.

[PH13] David A. Patterson und John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269, 9780124077263.

[PH98] David A. Patterson und John L. Hennessy. *Computer Organization and Design (2Nd Ed.): The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 1-55860-428-6.

[Pre+07] A. Pretschner u. a. "Software Engineering for Automotive Systems: A Roadmap". In: *Future of Software Engineering, 2007. FOSE '07*. 2007, S. 55–71. DOI: 10.1109/FOSE.2007.22.

[Reb12] Anja Rebhan. "Modellbildung und Beobachterentwurf für die Lagedynamik eines Quadrocopters". Bachelorarbeit. Friedrich-Alexander Universität Erlangen-Nürnberg, 2012.

[Roo12] Julius Roob. “An Introduction to the Research on Scratchpad Memory: Definition, Hardware, Known Implementations and WCET Optimisation”. In: (2012).

[Sai+15] Selma Saidi u. a. “The shift to multicores in real-time and safety-critical systems”. In: *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press. 2015, S. 220–229.

[Sti12] Michael Stilkerich. “Memory Protection at Option - Application-Tailored Memory Safety in Safety-Critical Embedded Systems”. doctoralthesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2012.

[Sti17] I. Stilkerich. “Assisted Memory Protection and Memory Management Through Astree”. 2017.

[Sti18] Michael Stilkerich. “Semantische Code-Analyse Tools: Astrée vs. Polyspace”. Interne Schaeffler Präsentation. 2018.

[Tex] *KeyStone Architecture Memory Protection Unit (MPU)*. Texas Instruments. Juni 2013. URL: http://www.ti.com/lit/ug/sprugw5a/sprugw5a.pdf.

[TH07] Jürgen Teich und Christian Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung, 2. Auflage*. Springer, 2007. ISBN: 978-3-540-46822-6.

[Tria] *Aurix TC27x D-Step User’s Manual*. 2014-12. Infineon Technologies AG. 81726 Munich, Germany, 2014.

[Trib] *TriCore V1.6 Core Architecture*. V1.0. Infineon Technologies AG. 81726 Munich, Germany, Mai 2012.

[Ulb+11] Peter Ulbrich u. a. “I4Copter: An adaptable and modular quadrotor platform”. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM. 2011, S. 380–386.

[Ulb+12] P. Ulbrich u. a. “Taking Control: Modular and Adaptive Robotics Process Control Systems”. In: *2012 IEEE International Symposium on Robotic and Sensors Environments Proceedings*. 2012, S. 55–60. DOI: 10.1109/ROSE.2012.6402632.

[Ulb16] Peter Ulbrich. “Echtzeitsysteme Vorlesung”. Okt. 2016. URL: http://www4.cs.fau.de/Lehre/WS16/V_EZS/.

[Waw09] Christian Walter Alois Wawersich. “KESO: Konstruktiver Speicherschutz für Eingebettete Systeme”. Diss. Friedrich-Alexander-Universität Erlangen-Nürnberg, Okt. 2009.

[Kar18] Karlsruher Institute für Technologie. *ARAMIS II*. 2018. URL: https://www.aramis2.org/.

[Ora04] Oracle. *New Features and enhancements J2SE 5.0*. 2004. URL: https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html.

[Sta18] Stack Overflow. *Developer Survey Results 2018*. Jan. 2018. URL: https://insights.stackoverflow.com/survey/2018.

[AUT] AUTOSAR. *AUTOSAR*. URL: https://www.autosar.org/.

[Abs] AbsInt Angewandte Informatik GmbH. *Astrée: Laufzeitfehleranalyse*. URL: https://www.absint.com/astree/index_de.htm.

[Alt] Altium Limited. *Tricore VX Software Development Tools*. URL: https://www.tasking.com/products/tricore-vx-software-development-tools.

[Ama] Amazon Web Services. *Memory Protection Unit (MPU) Support*. URL: https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[Hig] HighTec EDV-Systeme GmbH. *Development Platform - C/C++ compiler suites for automotive and industrial*. URL: https://hightec-rt.com/en/products/development-platform.html.

[MIS] MISRA Consortium. *Activities - MISRA C*. URL: https://www.misra.org.uk/Activities/MISRAC/tabid/160/Default.aspx.

[Mic18] MicroConsult GmbH. "AURIX Family Training". 2018.

[Sun95a] Sun Microsystems Computer Company. "The Java Language: A White Paper". In: (1995).

[Sun95b] Sun Microsystems Computer Company. "The Java Language Environment: A White Paper". In: (Okt. 1995).

[Thea] The MathWorks Incorporation. *Polyspace - Making Critical Code Safe and Secure*. URL: https://de.mathworks.com/products/polyspace.html.

[Theb] The MathWorks Incorporation. *Polyspace Results in Polyspace Bug Funder - By Category*. URL: http://de.mathworks.com/help/bugfinder/defects-runtime-checks-misra-coding-rules.html#bty2ut6-2.

[Thec] The MathWorks Incorporation. *Simulink - Simulation und Model-Based Design*. URL: https://de.mathworks.com/products/simulink.html.

[Vec] Vector Informatik GmbH. *Entwurf von AUTOSAR-Softwarekomponenten mit DaVinci Developer*. URL: https://vector.com/vi_davinci_developer_de.html.

[Vek] Vektor Informatik GmbH. *Beschreibungsdateien für steuergräteinterne Größen*. URL: https://vector.com/vi_datadescription_ecu1_de.html.