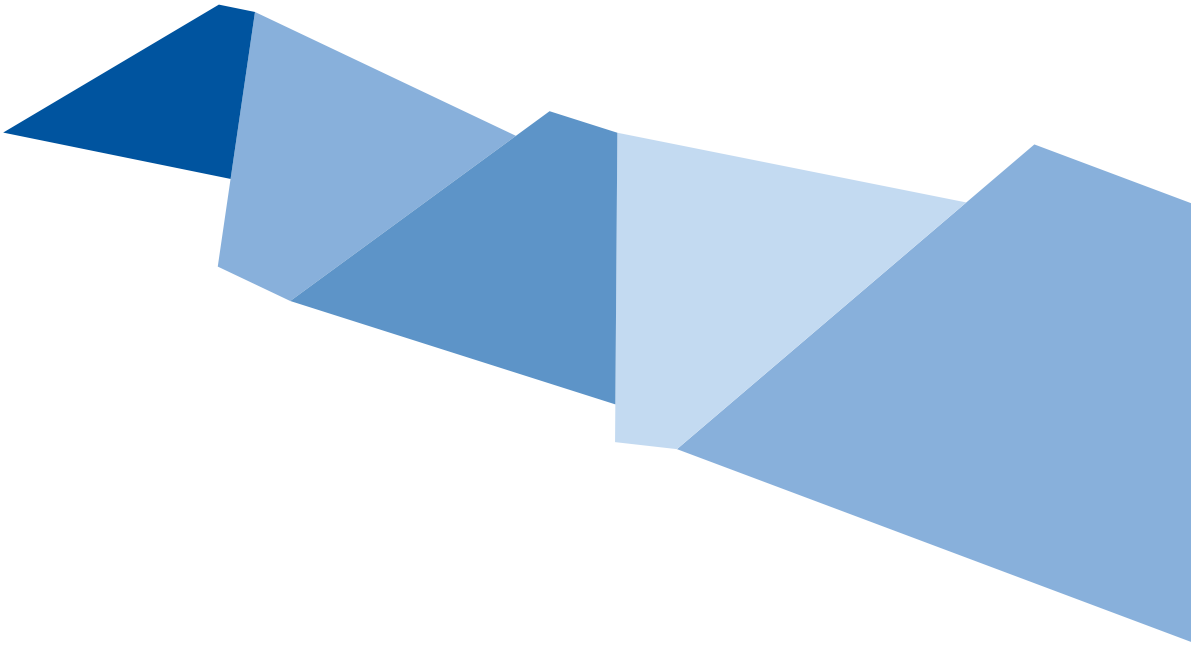


Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

Sebastian Patrick Grobosch

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2019-03
RWTH Aachen University | Department of Computer Science | Oktober 2019

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Ingenieur
Sebastian Patrick Grobosch
aus Bottrop

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr.-Ing. Sebastian Engell

Tag der mündlichen Prüfung: 30. November 2018

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über `http://dnb.d-nb.de` abrufbar.

Zugl.: D 82 (Diss. RWTH Aachen University, 2018)

Sebastian Patrick Grobosch
`sebastian.grobosch@rwth-aachen.de`

Aachener Informatik Bericht AIB-2019-03

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

ISBN 978-3-8440-6984-6

Shaker Verlag GmbH • Am Langen Graben 15a • 52353 Düren
Telefon: 02421 / 99 0 11 - 0 • Telefax: 02421 / 99 0 11 - 9
Internet: `www.shaker.de` • E-Mail: `info@shaker.de`

Zusammenfassung

Die Anzahl der Steuergeräte in Fahrzeugen der Oberklasse ist in den letzten 15 Jahren stetig gestiegen und liegt aktuell bei etwa 100 Stück. Dabei entsteht der Großteil aller Innovationen im Fahrzeug durch Elektronik und Software. Dies macht Software einerseits zu einem der wichtigsten Innovationstreiber für Unternehmen in der Automobilindustrie, andererseits birgt sie ein hohes Risikopotential: Programmfehler.

Durch internationale Normen und strengere Anforderungen an die Software-Qualität wird versucht, diesem Risiko entgegenzuwirken. Um alle Wünsche der Endkunden individuell erfüllen zu können, wächst allerdings die Komplexität der Systeme durch die steigende Variantenvielfalt. Das Testen von solchen Software-Systemen kann dabei nur das Vorhandensein von Fehlern zeigen, jedoch nicht deren Abwesenheit. Eine Garantie, dass ein System die gestellten Anforderungen erfüllt, kann durch formale Methoden gegeben werden.

Die in dieser Arbeit vorgestellten Ansätze tragen dazu bei, die Software-Entwicklung in kleinen und mittleren Unternehmen durch Methoden der formalen Verifikation zu verbessern und zu unterstützen. Dabei werden die Vorteile von kleinen gegenüber großen Unternehmen genutzt und ausgebaut. Dazu zählen die ausgeprägte Nähe zum Kunden sowie ein hohes Maß an Flexibilität und Wirtschaftlichkeit. Die Systemkomplexität der meisten Projekte sowie die Prozessstrukturen können positiv zur Akzeptanz für die Einführung von formalen Methoden in den jeweiligen Entwicklungsprozess beitragen.

Der erste Ansatz befasst sich mit der Analyse von Zeitanforderungen für eingebettete Systeme basierend auf der formalen Methode des Model-Checkings. Dabei wird für ein bestehendes Variantensystem für Steuergeräte ein Task-System mittels Uppaal modelliert und eine Einplanbarkeitsanalyse auf Basis von Zeitautomaten vorgestellt. Zur Verwaltung der Varianten wurde ein Framework basierend auf pure::variants entworfen und eine bestehende Software-Plattform evolutionär in eine Produktlinie umgewandelt. Damit können sich Unternehmen stärker auf die individuellen Kundenwünsche fokussieren und vorhandene Komponenten effizient und mit hoher Qualität wiederverwenden.

Der zweite Ansatz zur Verbesserung der Software-Qualität befasst sich mit der Verifikation von Programmcode eingebetteter Systeme durch den Model-Checker Arcade. Hierbei wurde speziell das Formulieren von formalen Anforderungen und die Anwendbarkeit im industriellen Umfeld untersucht. Mittels dieses Ansatzes konnten sowohl Fehler im Programmcode lokalisiert als auch das Einhalten von Anforderungen gezeigt werden. Der Einsatz von Binär-Code-Verifikation kann den Testaufwand reduzieren, aber nicht ersetzen. Der Vorteil für Unternehmen ist allerdings, dass diese Methode das Ausbleiben von Fehlern beweisen kann, was durch herkömmliches Testen nicht möglich ist.

Insgesamt wurde ein Ansatz zur Integration formaler Methoden in den Entwicklungsprozess eines kleinen und mittleren Unternehmens vorgestellt, erfolgreich mit entsprechender Werkzeugunterstützung umgesetzt und evaluiert. Mit Hilfe der gezeigten Methoden ist es möglich, den Testaufwand zu reduzieren und die Qualität von automobilen Steuerungssystemen schon frühzeitig in den wichtigen Phasen der Entwicklung zu steigern.

Abstract

The number of control units within upper class vehicles has steadily increased over the last 15 years and is currently around 100 units. The majority of all innovations in the vehicle are generated by electronics and software. This makes software, on the one hand, one of the most important drivers of innovation for companies in the automotive industry. On the other hand, it involves a high risk potential: programming errors.

With international standards and stricter requirements for software quality the industry is trying to counteract this risk. However, the complexity of the systems is growing due to the increasing diversity of variants in order to be able to fulfil all the wishes of the end customer individually. The testing of such software systems can only show the presence of errors, but not their absence. A guarantee that a system fulfils the requirements can be provided by formal methods.

The approaches presented in this thesis help to improve and support software development in small and medium-sized enterprises by means of formal verification methods. In doing so, the advantages of small over large companies should be utilised and enhanced. This includes the close proximity to the customer as well as a high degree of flexibility and profitability. The system complexity of most projects as well as the process structures can positively contribute to the adoption of formal methods in the respective development process.

The first approach deals with the analysis of timing requirements for embedded systems based on the formal method of model checking. In this case, a task system is modelled using Uppaal for an existing variant system for control units, and a schedulability analysis based on timed automata is presented. To manage the variants, a framework based on pure::variants was designed and an existing software platform was transformed into a product line. This allows companies to focus more on individual customer requirements and to reuse existing components efficiently and with high quality.

The second approach to improve the quality of software is to verify the program code of embedded systems through the model checker Arcade. Specifically, the formalization of formal requirements and the applicability in the industrial environment were analysed. Errors in the program code could be localized as well as compliance with requirements were shown. The use of binary code verification can reduce the test effort, but will not replace it. The advantage for companies is, however, that this method can prove the absence of errors, which is not possible by conventional testing.

Overall, an approach to the integration of formal methods into the development process of a small and medium-sized enterprise was presented, successfully implemented and evaluated with appropriate tool support. With the methods shown, it is possible to reduce the test effort and to increase the quality of automotive control systems at an early stage in the important phases of development.

Danksagung

Grundlage für die erzielten Ergebnisse ist die kooperative und freundschaftliche Zusammenarbeit des RWTH Lehrstuhls Informatik 11 - Embedded Software mit der VEMAC GmbH & Co. KG, durch die sie erst möglich wurde.

An erster Stelle gilt mein Dank meinem Doktorvater Prof. Dr.-Ing. Stefan Kowalewski für seine wissenschaftliche und methodische Anleitung und konstruktive Rückmeldung über die vielen Jahre hinweg. Prof. Dr.-Ing. Sebastian Engell danke ich für die Übernahme des Zweitgutachtens sowie für die fruchtbaren Hinweise während der Zusammenarbeit im MULTIFORM Projekt. Desweiteren danke ich Prof. Dr. rer. nat. Leif Kobbelt als Vorsitzenden der Prüfungskommision und ebenfalls Prof. Dr. rer. nat. Matthias Müller als Beisitzer.

Prof. Dr.-Ing. Michael Reke hat großen Anteil am Gelingen meiner Dissertation, denn er war unermüdlich an meiner Seite und hatte gewöhnlich zum richtigen Zeitpunkt die notwendigen aufbauenden Worte sowie Ratschläge, mit denen er mich bis zuletzt begleitet und unterstützt hat. Ihm möchte ich daher ganz besonders danken. Bei Dr. Martin Düsterhöft bedanke ich mich ferner sehr für das Initiieren und Ermöglichen meines Promotionsvorhabens.

Frau Margit Offergeld und Herrn Dr. Johannes Offergeld danke ich für ihr entgegengebrachtes Vertrauen, mit diesem und durch die kontinuierliche Begleitung der gesamten VEMAC - im Speziellen seitens Michael Kiausch und Axel Koblenz - diese Arbeit erst realisierbar war. Vor allem die tiefschürfenden Diskussionen und die konstruktive sowie immer amüsante Zusammenarbeit waren bedeutsam für deren Gelingen.

Meinen Studenten Surapa Ratanaprutthakul, Thorsten Will, Kriengkrai Krirkkawin danke ich, denn sie haben tatkräftigen Anteil an den Ergebnissen dieser Arbeit und trugen zu einem anregenden sowie ebenso angenehmen Arbeitsumfeld bei.

Dr.-Ing. Martin Hüfner und Volker Kamin sowie Dr. rer. nat. Sebastian Biallas bin ich dankbar für viele fachliche Gespräche und die multi-formale Zusammenarbeit. Überaus geholfen hat mir Dr.-Ing. André Stollenwerk, indem er mir den direkten Draht zum Lehrstuhl ermöglichte und darüber hinaus mich durch viele Diskussionen inspirierte.

Meinen Eltern Doris und Reinhard sowie meiner Schwester Anja bin ich sehr dankbar für die anhaltende Hilfestellung und Wegbereitung über all die Jahre hinweg, wodurch sie mich erst dahin gebracht haben, wo ich heute stehe.

Ferner danke ich meiner Frau Henrike für die liebevolle und ausdauernde Unterstüzung auch in schwierigen Zeiten sowie für das Rückenfreihalten und zahlreiche Zeitgeschenke. Meine beiden Töchter Antonia und Mathilda haben mir immer wieder neu Mut gemacht und mich motiviert. Ihnen möchte ich vor allem danken und hieran verdeutlichen, dass Durchhaltevermögen und Ausdauer, wovon mir viel abverlangt wurde, lohnenswert und zielführend sind.

Hildesheim, im Oktober 2019

Inhaltsverzeichnis

1 Einleitung

Die Anzahl der Steuergeräte in Fahrzeugen der Oberklasse ist in den letzten 15 Jahren stetig gestiegen und liegt nach Traub [115] aktuell bei etwa 100 Stück. Die Steuergeräte sind miteinander vernetzt und kommunizieren verstärkt mit der Umwelt, sowie mit Diensten im Internet. Diese Verschmelzung mit dem *Internet of Things (IoT)* soll das Leben der Kunden sicherer und komfortabler machen. Nach Frischkorn [45] entstehen dazu 90 % aller Innovationen im Fahrzeug durch Elektronik und Software, wobei etwa 50-70 % der Kosten bei der Entwicklung von Steuergeräten für Software anfallen. Dies macht Software einerseits zu einem der wichtigsten Innovationstreiber für Unternehmen in der Automobilindustrie, andererseits birgt sie auch ein hohes Risikopotential: Programmfehler.

Um dieses Risiko zu senken, steigen die Anforderungen an die Software-Qualität mit der immer stärkeren Verbreitung in automobilen Steuergeräten. Fahrerassistenzfunktionen (*Advanced Driver Assistant Systems (ADAS)*) und dabei im Speziellen autonom fahrende Autos sollen als Megatrend die nächsten Jahre [42] bestimmen und für hohe Umsätze in der Automobilindustrie sorgen. Software wird dabei eine führende Rolle in deren Umsetzung einnehmen. Im Fehlerfall können diese Funktionen allerdings Gefahr für Leib und Leben bedeuten und sind somit als sicherheitskritisch einzustufen. 1998 wurde die Norm IEC 61508 allgemein für sicherheitskritische Elektronik veröffentlicht, woraus spezialisierte Normen unter anderem für Bahnanwendungen (EN 50128) und für Straßenfahrzeuge (ISO 26262) abgeleitet wurden und somit zum aktuellen Stand der Technik zählen. Unternehmen, die Komponenten für Steuergeräte für den Einsatz in Serienfahrzeugen entwickeln, sind verpflichtet, diese Normen bei ihrer Entwicklung zu berücksichtigen, wodurch sowohl Prozesse als auch Werkzeuge beeinflusst werden.

Neben immer höher werdenden Ansprüchen an die Qualität von Software, steigen durch den Einsatz von Varianten die Aufwände für deren Entwicklung und Wartung. Große Automobilhersteller wie z. B. VW [2] nutzen Synergieeffekte von Baukastensystemen für ihre Fahrzeugflotten, um Kosten zu sparen und die unterschiedlichsten Wünsche ihrer Kunden erfüllen zu können. Laut Dziobek et al. [38] dürften aus statistischen Gesichtspunkten z. B. für die C-Klasse von Daimler 2006 keine zwei identischen Fahrzeuge vom Band gelaufen sein. Um Software-Systeme an diese Herausforderungen sowie an den steigenden Kostendruck in der Entwicklung anzupassen, müssen Software-Komponenten nach Mossinger [75] sowohl plattformübergreifend bei einem Hersteller als auch herstellerübergreifend wiederverwendet werden. Die Bemühungen der Hersteller, eine einheitliche Software-Plattform zu schaffen, in der Komponenten von unterschiedlichen Zulieferern zusammenarbeiten und gegeneinander ausgetauscht werden können, mündeten in die *AUTomotive Open System ARchitecture (AUTOSAR)*.

Damit *kleine und mittlere Unternehmen (KMU)* in diesem Umfeld wirtschaftlich sowie technologisch mithalten können, müssen nach Reke [93] die Prozesse zu den Anforderungen der Automobilhersteller kompatibel sein. Durch das Einbringen von entwickelter Software in ein oben beschriebenes Baukastensystem steigt mit den hohen Stückzahlen auch das wirtschaftliche Risiko. Tritt ein Programmfehler im Feld auf, so können unter Umständen durch den Einsatz in unterschiedlichen Modellreihen mehrere hunderttausend Fahrzeuge betroffen sein. Das frühzeitige Finden bzw. Vermeiden von Fehlern bekommt somit einen höheren Stellenwert. Höhere Aufwendungen für das Testen, das Wiederverwenden von projektübergreifendem Wissen sowie der Einsatz von lang erprobten Software-Komponenten können die Kosten und den Aufwand beim Auftreten von Fehlern im Feld reduzieren.

Dabei kann Testen nur das Vorhandensein von Fehlern zeigen, jedoch nicht deren Abwesenheit. Eine Garantie, dass ein System die gestellten Anforderungen erfüllt, kann aufgrund eines formalen Beweises erteilt werden. Dieser Beweis wird durch formale Methoden geführt, die sowohl die Spezifikation als auch die Verifikation umfassen. Laut Heitmeyer [49] haben formale Methoden das Potential, die Kosten der Software-Entwicklung zu reduzieren und deren Qualität zu verbessern. Neben formalen Verifikationsmethoden wie *Abstract Interpretation* und *Theorem Proving* steht auch *Model Checking* im Fokus von Industrie und Forschung. Von Klein et al. [60] wurde die formale Verifikation eines Betriebsystemkerns erfolgreich durchgeführt. Die NASA präsentiert in [110] ihre Erfahrungen mit dem Model-Checking eines Roboter-Systems, und die DARPA möchte mittels Crowdsourcing [3] die hohe Komplexität der Verifikationsaufgaben spielerisch lösen lassen. Dabei muss die Beweisbarkeit des Erfüllens von Anforderungen immer im Kontext der eingesetzten Methoden betrachtet werden, was eine formale Verifikation der verwendeten Werkzeuge mit einbezieht.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist es, die Software-Entwicklung in *KMU* durch Methoden der formalen Verifikation zu unterstützen und zu verbessern. Dabei soll die Qualität der Software gesteigert werden, um damit die Zuverlässigkeit der damit erstellten Produkte zu erhöhen. Durch das Finden bzw. die Vermeidung von Fehlern in frühen Phasen der Entwicklung soll die Anzahl von Fehlern in späteren Projektphasen oder sogar im Betrieb reduziert werden. Um aktuelle Normen und Standards speziell für die Entwicklung sicherheitskritischer Systeme zu erfüllen, können industrieübliche Methodiken zum Testen von Software durch formale Methoden zur Verifikation von Anforderungen ergänzt werden. Dies soll im Kontext von *KMU* untersucht und durch entsprechende Prozesse und Werkzeuge unterstützt werden. Dabei sollen die Vorteile der kleinen Unternehmen in Bezug auf Flexibilität, Prozessstruktur sowie Kundennähe erhalten bleiben und gestärkt werden.

Des Weiteren soll eine Methodik entwickelt werden, um in einer frühen Phase der

Entwicklung von Software für automotive Serien-Steuergeräte eine möglichst präzise Abschätzung des Ressourcenbedarfs in Bezug auf die benötigte Rechenzeit zu erzielen. Dies soll dem Hersteller ermöglichen, seine Abschätzungen bezüglich der zu verwendenden Hardware kosteneffizienter zu gestalten. Auf der Seite des Kunden soll frühzeitig das Vertrauen in das Produkt durch eine geringere Fehlerquote gestärkt sowie spätere Anpassungen der Hardware an höhere Softwareanforderungen vermieden werden. Weiterhin soll diese Methodik die Flexibilität einer projektübergreifenden Software-Plattform basierend auf dem Baukastenprinzip unterstützen.

1.2 Beitrag

Diese Arbeit trägt in folgenden Punkten zum Erreichen der im vorangegangenen Abschnitt festgelegten Ziele bei:

Varianten-basierte Einplanbarkeitsanalyse Um eine Abschätzung des Ressourcenverbrauchs eines zu entwickelnden Systems auf einem Mikrokontroller durchzuführen, wird eine Methode zur Einplanbarkeitsanalyse basierend auf Zeitautomaten entwickelt. Diese kann flexibel auf unterschiedliche Scheduling-Verfahren angepasst werden. Weiterhin wird diese Methode kombiniert mit einem System zur Verwaltung von Varianten, um die Einplanbarkeitsanalyse in Abhängigkeit der gewählten Variante automatisch durchführen zu können. Zur Evaluierung wird ein bestehendes Software-Produkt eines *KMU* für ein automobiles Steuergerät in eine Software-Produktlinie überführt und in das entstandene Framework integriert.

Verifikation von Binär-Code Es wird die Durchführung von formaler Verifikation unter Berücksichtigung der Anforderungen eines *KMU* analysiert und eine Methodik zur Anwendung von Model-Checking entwickelt. Dabei wird speziell die Anwendbarkeit für Software-Entwickler ohne Erfahrung mit dieser Methode berücksichtigt. Hierzu werden Möglichkeiten zur formalen Spezifikation von Anforderungen durch eine Umfrage unter Entwicklern und Forschern evaluiert. Anhand mehrerer Beispiele von Anforderungen werden deren Formalisierung erläutert und die Verifikation dieser formalen Spezifikation an Fallbeispielen durchgeführt. Mit Hilfe der dabei gefundenen Fehler, den bewiesenen Anforderungen sowie der Analyse des Formalisierungsprozesses werden Entwurfsmuster für den produktiven Einsatz der Verifikationsmethode erarbeitet.

KMU-orientierter Entwicklungsprozess Die in dieser Arbeit entwickelten Methoden zur formalen Verifikation werden in einem gemeinsamen Werkzeug zusammengeführt und in einen Entwicklungsprozess im Kontext eines *KMU* eingebettet. Zum einen ist dies die Ressourcenabschätzung in der frühen Phase eines Projektes auf Basis von Uppaal und zum anderen die formale Verifikation des kompilierten Quelltextes mittels Arcade. Dazu wird als erstes der Prozess entsprechend angepasst und als

zweites ein Werkzeug für das Verwalten von Varianten beschrieben, um nachfolgend das im MULTIFORM Projekt entwickelte Design-Framework zur Unterstützung des Entwicklungsprozesses um die entsprechenden Werkzeuge zu erweitern.

Evaluierung Anhand von zwei realen Fallbeispielen aus der Automobilindustrie werden die oben genannten Methoden evaluiert.

1.3 Vorgehensweise und Gliederung

In Kapitel 2 werden die für das Verständnis dieser Arbeit notwendigen Begrifflichkeiten, Definitionen und Fallbeispiele eingeführt. Kapitel 3 beschreibt die Erarbeitung einer Methodik zur Abschätzung des Ressourcenbedarfs im Kontext einer Produktlinie wie auch deren Evaluation an einem Fallbeispiel. Hierbei wird eine Einplanbarkeitsanalyse auf Basis von Zeitautomaten erörtert sowie die Einbettung dieser in ein Werkzeug zur Varianten-Verwaltung. Die Anwendung der formalen Verifikation sowie die Analyse von Möglichkeiten zur formalen Spezifikation werden in Kapitel 4 geschildert und an einem realen System evaluiert. Kapitel 5 beschreibt ein Framework, welches die zuvor entwickelten Methoden zusammenfasst sowie dessen Integration in den Software-Entwicklungsprozess eines *KMU*. Die Arbeit schließt in Kapitel 6 mit einer Zusammenfassung sowie der Bewertung der erzielten Ergebnisse und einem Ausblick auf daraus abgeleitete Arbeiten.

1.4 Bibliografische Hinweise und Beiträge des Autors

In dieser Arbeit werden zwei unterschiedliche Formate für die Angabe von Literaturquellen verwendet. Literaturverweise auf eigene Arbeiten des Autors bestehen aus den Anfangsbuchstaben der Nachnamen der bis zu vier ersten Autoren gefolgt vom Publikationsjahr, z. B. [GKKK11]. Alle weiteren Literaturverweise werden fortlaufend nummeriert und nach dem Nachnamen des Erstautors sortiert angegeben, z. B. [116].

Teile der in dieser Arbeit vorgestellten Ergebnisse wurden bereits in Artikeln veröffentlicht und beruhen in einigen Fällen auf betreuten Abschlussarbeiten. Im Folgenden werden sowohl die Artikel als auch die Abschlussarbeiten in den Gesamtkontext eingeordnet.

Die in Kapitel 3 beschriebene Varianten-basierte Ressourcenabschätzung basiert auf Ideen und Konzepten des Autors. Eine erste Analyse zur Einplanbarkeit von Tasks auf dem im Fallbeispiel verwendeten Mikrokontroller-System wurden von Volker Kamin und dem Autor im Rahmen des MULTIFORM Projektes durchgeführt. Vorbereitende Arbeiten zur Verwaltung von Varianten führte Kriengkrai Krirkkawin in seiner Masterarbeit [62] unter der Leitung des Autors durch. In [GKKK11] wurden die Ergebnisse der weiterführenden Entwicklung in Bezug auf die Einplanbarkeitsanalyse sowie das entstandene Framework veröffentlicht. Unveröffentlicht ist die vom Autor konzeptionierte, durchgeführte sowie evaluierte Umwandlung der beschriebenen Steuergeräte-Plattform in eine Produktlinie.

Der Autor entwarf erste Anforderungen zu den Fallbeispielen und formalisierte diese mittels *Computational Tree Logic (CTL)*. Surapa Ratanaprutthakul evaluierte diese Anforderungen in ihrer Masterarbeit [89] unter der Leitung des Autors mit ARCADE. Die Ergebnisse wurden in [RG11] veröffentlicht. Thorsten Will führte in seiner Diplomarbeit [118] unter der Leitung des Autors die Formalisierung der Anforderungen fort und analysierte eine alternative Methode zur formalen Spezifikation mittels *Safety-Automaten*. Die in Kapitel 4 bisher unveröffentlichte Umfrage unter Entwicklern zur Anwendung von *Computational Tree Logic* bzw. *Safety-Automaten (SA)* zur Formalisierung von Anforderungen wurde vom Autor konzeptioniert, durchgeführt und ausgewertet. Eine Bewertung des erstellten Frameworks sowie erste Ergebnisse der formalen Verifikation wurden in [GRK13] vorgestellt. Unveröffentlicht erweiterte der Autor den Umfang der formalisierten Anforderungen, verbesserte sie und ergänzte diese um die dazu passenden *Safety-Automaten*. Weiterhin evaluierte der Autor die Anforderungen an einem Fallbeispiel, bewertete die Ergebnisse und leitete daraus Hinweise zur Anwendung ab.

Der in Kapitel 5 verwendete Entwicklungsprozess eines *KMU* wurde von Michael Reke in seiner Dissertation [93] beschrieben und in [RGK12] auszugsweise veröffentlicht. Der Autor erweiterte den Prozess um die in dieser Arbeit gezeigten Ansätze und konzeptionierte und entwarf ein Framework zum Varianten-Management. Eine erste Version davon realisierte Kriengkrai Krirkkawin in seiner Masterarbeit [62]. Surapa Ratanaprutthakul setzte in ihrer Masterarbeit [89] die Einbindung des Model-Checking Werkzeugs ARCADE in das Framework um. Der Autor setzte mit Hilfe von Martin Hüfner den zuvor erweiterten Entwicklungsprozess sowie das Framework zum Varianten-Management im *Design Framework* des MULTIFORM Projektes um. Die Ergebnisse wurden in [HSEG13] veröffentlicht.

Der Autor beschreibt als Mitverfasser in zwei Konferenzbeiträgen [RGN11, RG14] sowie in einem Journal-Artikel [KWRG13] die in dieser Arbeit als Fallbeispiele verwendeten Systeme und deren Anwendungsmöglichkeiten.

Ein Großteil dieser Arbeit entstand im Umfeld des EU-Forschungsprojektes MULTIFORM[1]. Dessen Ziel war es, Werkzeuge und Methoden basierend auf unterschiedlichen Modellierungsformalismen zur Entwicklung von komplexen Steuerungssystemen konsistent miteinander zu verbinden. Dabei wurden alle Phasen der Entwicklung, angefangen von der Anforderungsanalyse bis hin zur System-Verifikation, mit einbezogen und die entwickelten Methoden an Fallbeispielen evaluiert.

[1] EU Projekt MULTIFORM, FP7, Vertragsnummer INFSO-ICT-224249

2 Grundlagen und Stand der Technik

Dieses Kapitel führt die für das Verständnis dieser Arbeit notwendigen Begrifflichkeiten und Definitionen in Bezug auf die formale Verifikation und Spezifikation ein. Des Weiteren werden die Anwendung von formalen Methoden in der Industrie beschrieben sowie die verwendeten Fallbeispiele und Entwicklungsprozesse näher erläutert.

2.1 Grundlegendes Prozessmodell (V-Modell)

Speziell in der Automobilindustrie hat sich in der Entwicklung als Prozessmodell das V-Modell durchgesetzt. In [53] existiert eine gute Einführung in die Entstehungsgeschichte und den Aufbau der verschiedenen Versionen des V-Modells. Auf dieser Quelle basiert die folgende Beschreibung.

Boehm [23] hat Ende der 70er Jahre die erste Version (siehe Abbildung 2.1) erdacht. Basierend auf dem Wasserfall-Modell, welches ein eindimensionales Vorgehensmodell darstellt, hat Boehm mit dem V-Modell eine zweite Dimension eingeführt. Dem absteigenden Ast des „V“ steht nun auch ein aufsteigender Ast entgegen. Dieser beinhaltet Testschritte, die in Bezug zu Entwurfsschritten auf dem abfallenden Ast stehen. So steht zum Beispiel der Implementierung des *Codes* das Testen der einzelnen Einheiten mittels *Unit test* gegenüber.

Zu Beginn eines Projektes wird mit dem Kunden zusammen ein grobes Konzept erarbeitet, welches beinhaltet, was für ein Produkt entwickelt werden soll. Daraufhin werden daraus Anforderungen (engl. Requirements) für die Entwicklung abgeleitet, auf die sich die Entwickler und der Kunde einigen. Diese werden als *Requirements Baseline* festgehalten und dienen als Basis der Zusammenarbeit zwischen Auftragnehmer und Auftraggeber. Daraufhin beginnt die eigentliche Entwicklung mit dem Produkt-Design und wird weiter verfeinert, bis letztlich der Quellcode geschrieben ist. Dieser soll das Software-Produkt darstellen, welches die definierten Anforderungen erfüllt.

Im aufsteigenden Ast werden nun die entsprechenden Phasen im abfallenden Ast verifiziert und validiert. Boehm teilt diese Schritte zur Überprüfung mittels der *Requirements Baseline* auf. Er definiert den Unterschied in [23] wie folgt:

> *Verifikation*: „Am I building the product right?“
> *Validierung*: „Am I building the right product?“

Damit sieht er in der Verifikation eine Überprüfung des Produktes hinsichtlich der Erfüllung der aufgestellten Anforderungen bzw. Spezifikation durch das entwickelte

System. Dabei steht am Ende der Überprüfung ein Ergebnis, welches mit „richtig“ oder „falsch“ bewertet werden kann. Bei der Validierung hingegen sieht Boehm keine konkrete Spezifikation gegen die das Ergebnis verifiziert werden kann sondern nur natürlichsprachlich formulierte Dokumente und Vorstellungen des Kunden. Die Ergebnisse aus der Validierung liefern typischerweise „angemessen“ oder „nicht angemessen“ und bewerten damit, ob das entwickelte Produkt dem entspricht, was sich der Kunde vorgestellt hat.

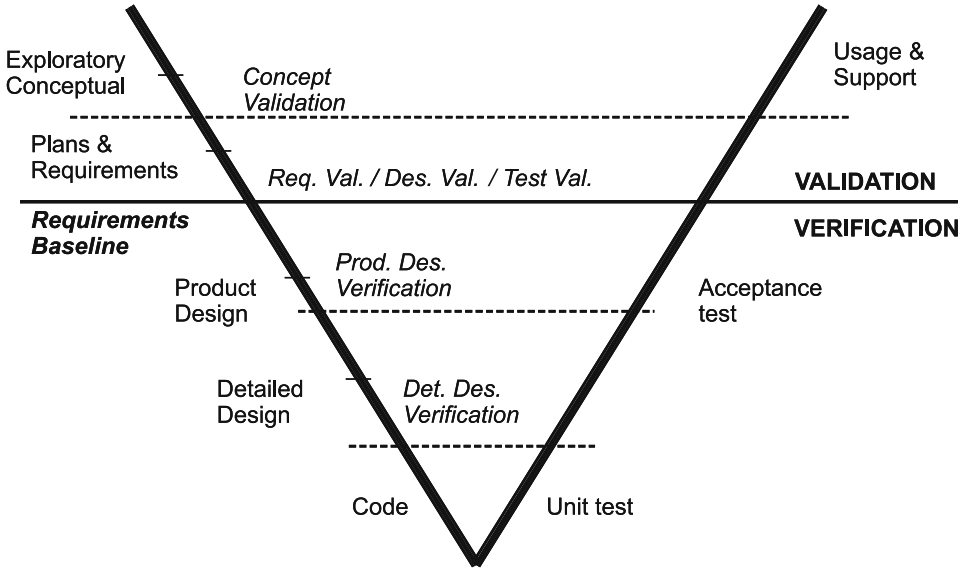


Abbildung 2.1: V-Modell nach [23]

Mitte der 80er Jahre haben deutsche Bundesbehörden damit begonnen, dieses Modell zu standardisieren und in ihre eigenen Software-Entwicklungsprozesse zu integrieren, um die Qualität grundsätzlich zu verbessern und Kosten einzusparen. 1997 wurde eine weiterentwickelte Version vorgestellt, das V-Modell 97. Diese Version berücksichtigte neue Ansätze der Software-Entwicklung, wie z. B. Objektorientierung. Eine wesentliche Erweiterung ist auch die Unterscheidung von verschiedenen Rollen der Prozess-Beteiligten, wie z. B. Entwickler, Qualitätssicherer und Projektmanager. Die im ursprünglichen Modell vorhandene „Requirements Baseline“ entfällt bei der Version von 1997 und lässt die Unterscheidung von Verifikation und Validierung verschwinden. Dafür werden jedoch die Querbezüge zwischen den einzelnen Schritten im abfallenden Ast zu den entsprechenden Verifikations- bzw. Validierungsschritten im aufsteigenden Ast durch einseitig gerichtete Verbindungspfeile deutlicher hervorgehoben. Allgemeine Kritikpunkte, wie z. B. eine unzureichende Skalierbarkeit und Modularisierung, führten zu einer erneuten Weiterentwicklung des V-Modell 97. 2005 wurde das V-Modell XT („eXtreme Tailoring“) veröffentlicht. Diese Version soll durch ihre Anpassbarkeit an unterschiedliche Projekte auf die genannten Kritikpunkte eingehen und stellt den aktuellen Stand der Entwicklung dar.

2.2 Formale Methoden

Wing gibt in [119] eine übersichtliche Einführung in die Grundprinzipien von formalen Methoden. Danach besteht das Grundprinzip einer jeden formalen Methode in der Verwendung von mathematischen Techniken und formalen Logiken zur Spezifikation und Verifikation von *Eigenschaft*en und *System*en. Dabei basieren formale Methoden auf formalen Sprachen mit klar festgelegter Syntax und Semantik. Die Verwendung dieser Methoden zieht sich durch den gesamten Entwicklungsprozess, angefangen bei der *Spezifikation* von Kundenanforderungen, des Architektur-Designs und der Implementierung bis hin zur Verifikation und Evaluation der Ergebnisse in allen Ebenen des Prozesses. Dabei werden formale Methoden verwendet, um Ambiguitäten, Unvollständigkeiten und Inkonsistenzen bei der Entwicklung zu vermeiden bzw. zu verringern und Fehler zu finden.

Grundsätzlich kann man formale Methoden in die Aspekte Spezifikation und Verifikation aufteilen. Die formale Spezifikation nutzt die formale Sprache zur präzisen Beschreibung bzw. Modellierung von Eigenschaften oder Systemen. Die formale Verifikation nutzt die formal-spezifizierten Eigenschaften und Systemmodelle, um die mathematische Korrektheit des Einhaltens der Eigenschaften im System zu zeigen. Dabei ist einer der Vorteile bei der Anwendung von formalen Spezifikationen das Erlangen von strukturiertem und detailliertem Wissen über das präzise Verhalten des zu beschreibenden Systems.

2.3 Definitionen laut ISO26262

Für die Entwicklung von sicherheitskritischen Systemen existiert seit 2011 der Standard ISO 26262 [55]. Dieser wurde abgeleitet aus der Norm IEC 61508, welche allgemein für industriell entwickelte, sicherheitskritische, elektrische sowie elektronische und programmierbare Systeme gilt. ISO 26262 gilt insbesondere für Systeme in Serien-Kraftfahrzeugen bis 3500 kg Gesamtmasse und soll die funktionale Sicherheit von Steuergeräten gewährleisten. Diese funktionale Sicherheit wird in der Norm wie folgt definiert:

> „3.53 Functional safety: Absence of unacceptable risk due to hazards caused by mal-functional behaviour of E/E systems“ [55]

Die Norm definiert sogenannte *Automotive Safety Integrity Level (ASIL)*. Die Einstufung des zu bewertenden sicherheitsrelevanten Systems in ein *ASIL* erfolgt vor dem Beginn der Entwicklung durch eine Gefahren- und Risikoanalyse durch den Gesamtsystemverantwortlichen. Für etwaige Teilsysteme, die ggf. auch durch Zulieferer bereitgestellt werden können, werden dabei ebenfalls *ASIL*-Einstufungen festgelegt. Während der unterschiedlichen Entwicklungsphasen müssen, abhängig von der jeweiligen Einstufung, verschiedene Entwicklungsmethoden angewendet und deren Anwendung für eine Zertifizierung nachgewiesen werden. Dabei wird explizit auf die Entwicklung nach dem V-Modell verwiesen. Die ISO beschreibt einen übergeordneten Systementwicklungsprozess nach dem V-Modell sowie diesem untergeordnete Entwicklungsprozesse für Software und für Hardware, die

ebenfalls dem V-Modell entsprechen. Darin werden explizit die Entwicklungsphasen Anforderungsanalyse, Architektur-Entwurf und Software-Entwurf und -Implementierung unterschieden. Bei den Methoden zur Verifikation der einzelnen Schritte unterscheidet die Norm allgemein zwischen formlosen (engl. informal), semi-formalen und formalen Methoden. Eine Methode wird dabei definiert durch Möglichkeiten zur Spezifikation sowie zur Verifikation. Die Definitionen der Norm lauten [55]:

> „*Informal notation*: Description technique that does not have its syntax completely defined“
> „*Informal verification*: Verification techniques not regarded as semi-formal or formal verification techniques“
> „*Semi-formal notation*: Description technique that has its syntax completely defined, but its semantics definition may be incomplete“
> „*Semi-formal verification*: Verification referring to the simulation that is described in a semi-formal notation and is executable“
> „*Formal notation*: Description technique that has both its syntax and semantics completely defined“
> „*Formal verification*: Mathematical proof of an algorithm or a specification against properties“

Bei formlosen Methoden ist weder eine Syntax noch eine Semantik definiert, und die Verifikation basiert auf keinen mathematischen Beweisen. Beispiele hierfür können Textdokumente für die Spezifikation sowie zur Verifikation manuelle Überprüfungen des Quellcodes bzw. der Architektur oder des Designs sein (engl. Review, Walkthrough).

Eine semi-formale Methode hingegen kann auf eine komplette Syntax zurückgreifen. *Unified Modeling Language (UML)* ist ein Beispiel für die grafische Modellierung von Systemen und Anforderungen. Zur Verifikation können beispielsweise Simulationen eingesetzt werden, um das Einhalten einer Anforderung im System zu zeigen.

Formale Methoden müssen hingegen nach der Norm eine Beschreibungstechnik aufweisen, bei der sowohl eine Syntax als auch eine Semantik vollständig definiert ist. Um die damit beschriebenen Eigenschaften in einem System (hier definiert als Algorithmus) bzw. in einer Spezifikation zu verifizieren, muss eine Möglichkeit zur Verfügung stehen, dazu einen mathematischen Beweis durchzuführen. Zur formalen Spezifikation können z. B. Event-B [1], die Z-Notation [56], temporale Logiken (siehe Unterabschnitt 2.5.3) oder auch Automaten verwendet werden. Bei der formalen Verifikation werden Methoden wie die statische Programmanalyse basierend auf abstrakter Interpretation [32], Symbolische Ausführung [59], Theorem-Beweiser oder die der automatischen Modellprüfung (engl. Model-Checking, siehe Unterabschnitt 2.6.1) verwendet. Die ISO bewertet den Einsatz aller Methoden mit Hilfe von drei Kategorien [55]:

„++: indicates that the method is **highly recommended** for the identified ASIL“
„+: indicates that the method is **recommended** for the identified ASIL“
„o: indicates that the method has **no recommendation** for or against its usage for the identified ASIL.“

Formale Methoden zur Verifikation von Software werden von der ISO für das Design, die Implementierung und das Architektur-Design mit „recommended“ für *ASIL*-C und *ASIL*-D eingestuft. Formale Notationen sind bei allen *ASIL* „recommended“. Die Einstufungen mit „highly recommended“ sind in der finalen Version der ISO für keine formale Methode zu finden.

2.4 Anwendung in der Industrie

In der Industrie werden formale Methoden sowohl in der Verifikation von Hardware als auch von Software eingesetzt. Nach Dill et al. [36] beschleunigten wirtschaftliche Randbedingungen den Einsatz von formalen Methoden speziell in der Entwicklung von Hardware. Kern et al. [58] fasste die verschiedenen Methoden und Anwendungsbeispiele schon 1999 zusammen.

Speziell bei sicherheitskritischen Systemen finden verschiedene Methoden Verwendung, um die Qualität der Produkte zu steigern und das Risiko von Schäden an Leib und Leben zu reduzieren. Die Anwendungsfelder befinden sich dabei maßgeblich in den Bereichen Luft- und Raumfahrt, Automobil, Medizin, Energie und Transport. Tabelle 2.1 listet Forschungsprojekte auf, die sich mit der Entwicklung von formalen Methoden sowie deren Anwendung in der Industrie beschäftigen. Dabei sind sowohl Forschungseinrichtungen als auch Unternehmen an den Projekten beteiligt.

Projekt	Beschreibung
deploy www.deploy-project.eu	Entwicklung von formalen Werkzeugen mit starkem Fokus auf deren Überführung von der Forschung in industrielle Anwendungen.
MBAT www.mbat-artemis.eu	Entwicklung einer Referenz-Plattform zur Validierung und Verifikation von Modell-basierter Software für sichere eingebettete Systeme.
Verisoft XT www.verisoftxt.de	Entwicklung von Methoden und Werkzeugen zur durchgängigen, formalen Verifikation von Computersystemen.
TACLe tacle.knossosnet.gr	Timing-Analysen auf Code-Level Basis für Multi-Core Systeme.
PREDATOR www.predator-project.eu	Untersuchung und Optimierung von Echtzeit-Eigenschaften von harten Echtzeit-Systemen mit der Entwicklung von entsprechenden formalen Werkzeugen und Plattform-Konzepten zur Abschätzung von Ausführungszeiten.
AVACS www.avacs.org	Verifikation von Echtzeit-Anforderungen in verteilten Systemen sowie automatische Verifikation von hybriden Systemen.
FORTE www.forte-projekt.de	Integration von formalen Verifikationsprozessen für C- und VHDL-Programme.

Tabelle 2.1: Beispiele von Forschungsprojekten mit Bezug zu formalen Methoden im industriellen Einsatz

Tabelle 2.2 listet beispielhaft Werkzeuge zur formalen Verifikation von Software während verschiedener Phasen in der Entwicklung auf. Dabei sind dies meist Werkzeuge zur statischen Analyse, die u.a. auf Abstrakter Interpretation [32] oder Model-Checking [29, 40, 88] basieren und C/C++/Java-Code oder Binär-Code analysieren können. Damit können u.a. das Einhalten von Codier-Richtlinien wie z. B. MISRA[2] überprüft werden, aber auch Fehler wie Division durch Null, unerreichbare Code-Segmente, Zugriff auf ungültige Array-Indices oder auch Speicherüberläufe detektiert werden.

[2] http://www.misra.org.uk

Werkzeug	Beschreibung
PC-lint www.gimpel.com	Statische Code-Analyse für C und C++ findet z. B. NULL-Pointer, nicht initialisierte Variablen, nicht verwendete Funktionen/Code und Zuweisungen in Bedingungen.
Polyspace www.mathworks.de/products/polyspace	Statische Code-Analyse mit Hilfe von Abstrakter Interpretation für C, C++ und Ada, erstellt Metriken zur Software-Qualität (zyklomatische Komplexität, Code-Zeilen, etc.), integriert Werkzeugketten zur modellbasierten SW-Entwicklung (z. B. Mathworks Simulink, dSPACE TargetLink, IBM Rational Rhapsody).
Astrée Absint www.absint.com	Analysiert Laufzeiteigenschaften, findet mit Hilfe Abstrakter Interpretation, z. B. Divisionen durch Null, Array-Überläufe, unerreichbaren Code, wird eingesetzt im Luft- und Raumfahrt Bereich.
CodeSonar www.grammatech.com	Statische Code-Analyse von C, C++ und Java sowie x86 Binär-Code.
Goanna www.redlizards.com	Statische Code-Analyse für C und C++, findet z. B. Null-Pointer, Array-Überläufe, Speicherlecks, toten Code, Division durch Null, enthält Schnittstellen zu verschiedenen Entwicklungswerkzeugen (z. B. Eclipse, Visual Studio).
BTC Embedded-Validator www.btc-es.de	Formale Verifikation von dSPACE TargetLink Modellen mittels Model-Checking.
Uppaal www.uppaal.com	Modellierung, Validierung und Verifikation von Echtzeit-Systemen auf Basis von Zeit-Automaten mittels Model-Checking.

Tabelle 2.2: Beispiele von Werkzeugen zur formalen Unterstützung der Software-Entwicklung

2.5 Formale Spezifikation

Formale Spezifikationen sind präzise und eindeutig formulierte Aussagen in einer mathematisch logischen Form. Diese werden meist aus nicht-mathematischen Beschreibungen, wie z. B. natürlichsprachlich verfassten Dokumenten, in eine formale Sprache übersetzt. Nach [65] ist die formale Spezifikation eine Aussage in irgendeiner formalen Sprache und auf irgendeiner Abstraktionsebene über eine Sammlung von Eigenschaften, die irgendein

System erfüllen sollen. In dieser Arbeit beziehen sich diese Aussagen auf Anforderungen an ein System, welche durch formale Spezifikationen ausgedrückt werden. Diese Formalisierung von Aussagen hat folgende Vorteile (aufbauend auf [119]):

- Unterstützung und Verbesserung der **Kommunikation** zwischen Kunden, Entwicklern und anderen Projekt-Mitarbeitern durch eindeutige Beschreibungen.
- Formale Beschreibungen des Systems präzisieren das Verständnis und das Wissen über das Verhalten des Systems.
- Durch den Zwang, Anforderungen formal auszudrücken, gleicht sich das Verständnis dieser auf Seiten des Kunden und des Auftragnehmers an.
- Durch das Formalisieren von Anforderungen können **frühzeitig Fehler** in der Spezifikation gefunden werden, die durch traditionelle Methoden nicht gefunden worden wären.
- Die Spezifikationen können durch die **vorgegebene Semantik** auf Konsistenz geprüft werden.
- Mit Hilfe der formalen Spezifikation kann ein **konsistenter Übergang** zur Implementierung durchgeführt werden.
- Ermöglicht die Verwendung von **formalen Beweisen** zur Verifikation von Korrektheitseigenschaften.

Nachteile und Einschränkungen von formalen Spezifikationen können sein:

- Sehr **komplexer Aufbau** der Syntax und somit für den *normalen* Entwickler unverständlich.
- Die Umsetzung der in-formalen in formale Anforderungen eröffnet neue Fehlerquellen.
- Nur weil die Spezifikation formalisiert vor liegt, garantiert dies **keine Vollständigkeit** der Spezifikation.
- Das formal spezifizierte System ist immer nur eine mathematische Abstraktion des realen Systems, welche sich nie vollständig äquivalent verhalten wird.
- **Unterschiedliche Aspekte** eines Systems können durchaus auch mit unterschiedlichen formalen Spezifikationsmethoden dargestellt werden.

2.5.1 Aussagenlogik

In der Aussagenlogik analysiert und bewertet man den Wahrheitswert von atomaren Aussagen (engl. „**A**tomic **P**roposition“, kurz AP) und deren Junktoren. Junktoren sind logische Verknüpfungen (logische Operatoren) von atomaren Aussagen [90]. Atomare Aussagen sind Aussagen, die nicht mehr in Teil-Aussagen unterteilt werden können, und denen ein Wahrheitswert (wahr: „1“ oder falsch: „0“) zugeordnet werden kann. Komplexere Aussagen entstehen durch das Verknüpfen von atomaren Aussagen mit Junktoren. Beispiel: Eine Variable X hat den Wert 42. Dann ist die atomare Aussage dazu $X = 42$.

Bezeichnung	**Notation**	**Beispiel**	**natürliche Aussprache**
Negation	not, !, ¬	$\neg a$	nicht a
Konjunktion	and, $\wedge$, &	$a \wedge b$	a und b
Disjunktion	or, $\vee$, $\|\|$	$a \vee b$	a oder b
Implikation	$\Longrightarrow$, $\rightarrow$	$a \rightarrow b$	wenn a, dann b
Äquivalenz	$\equiv$, $\leftrightarrow$	$a \leftrightarrow b$	a genau dann, wenn b

Tabelle 2.3: Die gebräuchlichsten Junktoren der Aussagenlogik

Ein Beispiel dazu: „Der Kontakt-Sensor hat ausgelöst und der Airbag des Autos wurde entfaltet.“. Hierbei sind „Der Kontakt-Sensor hat ausgelöst.“ sowie „Der Airbag des Autos wurde entfaltet.“ atomare Aussagen und das „und“ die logische Verknüpfung, ein Junktor. In Tabelle 2.3 sind die gebräuchlichsten Junktoren in der Aussagenlogik aufgelistet. Tabelle 2.4 zeigt die dazu passende Wahrheitstabelle.

a	b	$\neg a$	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \leftrightarrow b$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

Tabelle 2.4: Wahrheitstabelle für Verknüpfungen aus Tabelle 2.3 in der Form nach [81, 120]

Die Aussagenlogik beschränkt sich auf die Analyse von Aussagen, kann kontextabhängige Eigenschaften und Relationen sowie zeitliche Zusammenhänge jedoch nicht ausdrücken

[90]. Dazu muss die Ausdrucksweise der Aussagenlogik erweitert werden. Hieraus ergeben sich die sogenannten temporalen Logiken.

2.5.2 Transitionssystem

Die in automobilen Steuergeräten eingesetzten eingebetteten Systeme zählen typischerweise zu den sogenannten *reaktiven Systemen* [108]. Reaktive Systeme interagieren dauerhaft mittels Ein- und Ausgaben mit ihrer Umgebung und terminieren nicht. Um dieses Verhalten zu modellieren, werden sogenannte *Zustände* des Systems zu einem definierten Zeitpunkt betrachtet. Reagiert das System auf äußere Stimulationen, so kann das System in einen anderen Zustand wechseln. Diese Änderung wird Transition genannt. Nach [11] können reaktive Systeme durch sogenannte Transitionssysteme beschrieben werden. Diese Systeme können nach [11] mit Hilfe von Kripke-Strukturen [61] als zustandsbasierter Transitionsgraph wie folgt definiert werden:

Gegeben sei eine Menge AP von atomaren Aussagen, somit ist eine Kripke-Struktur K über AP definiert als ein Tupel $K = (S, S_0, R, L)$. Dabei gilt:

- AP: Eine atomare Aussage.
- S: Endliche Menge an Zuständen, die das modellierte System einnehmen kann.
- $S_0 \subseteq S$: Menge aller initialer Zustände.
- $R \subseteq S \times S$: Transitionsrelation, die den Übergang von einem Zustand $s_1 \subseteq S$ zu einem Zustand $s_2 \subseteq S$ beschreibt.
- $L : S \to 2^{AP}$: Eine Funktion, die jedem Zustand eine Menge $E_s(S) \subseteq AP$ von aussagenlogischen atomaren Formeln zuordnet.

Typischerweise wird ein Transitionssystem als gerichteter Graph dargestellt, in dem die Knoten Zustände des Systems repräsentieren und die Kanten Transitionen. Betrachtet man ein Ausführungsprogramm, repräsentiert ein Zustand alle Informationen des Programms zum Zeitpunkt der Ausführung einer entsprechenden Code-Zeile, wie z. B. den Inhalt aller System-Register und des gesamten Speichers. Transitionen bilden die Ausführung einer Code-Zeile ab. Ändern sich z. B. Speicherinhalte, wechselt das System in einen anderen Zustand.

Ein **Pfad** π in einem Transitionssystem definiert dabei eine endliche Sequenz von Zuständen, die durch Transitionen verbunden sind: $\pi = s_1 s_2 ...$ mit $s_i \in S$.

Durch das Entfalten eines Transitionssystems über die Zeit entsteht ein unendlicher Berechnungsbaum. Dieser bildet alle möglichen Berechnungspfade in einer Baumstruktur ab. Abbildung 2.2 zeigt dies beispielhaft.

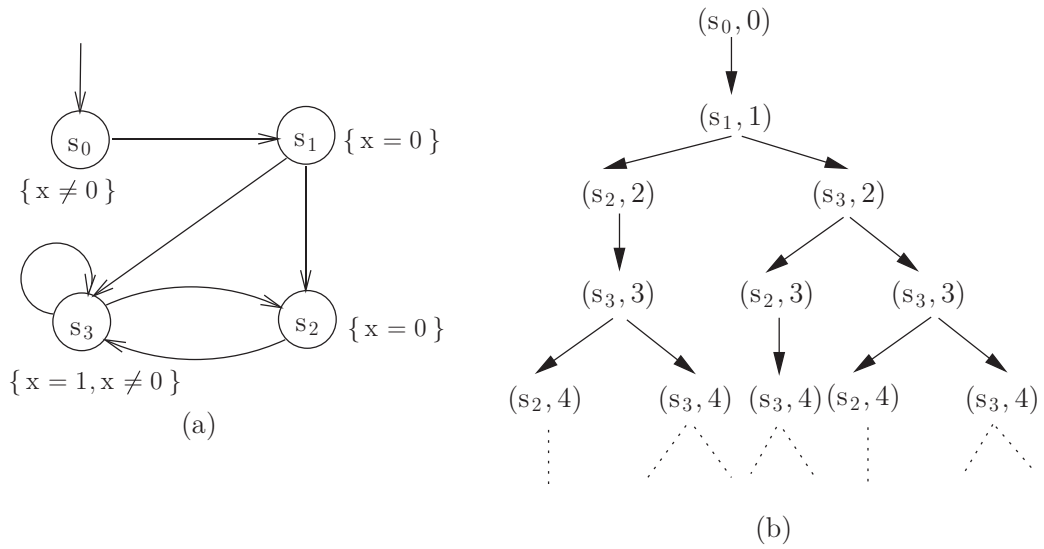


Abbildung 2.2: (a) Transitionssystem und (b) unendlicher Berechnungsbaum mit Startzustand S_0 nach [11]

Für die abgebildete Kripke-Struktur $K = (S, S_0, R, L)$ gilt:

- $S = \{s_0, s_1, s_2, s_3\}$
- $S_0 = \{s_0\}$
- $R = \{(s_0, s_1), (s_1, s_2), (s_1, s_3), (s_3, s_3), (s_3, s_2), (s_2, s_3)\}$
- $L_0 = \{x \neq 0\}; L_1 = \{x = 0\}; L_2 = \{x = 0\}; L_3 = \{x = 1, x \neq 0\}$

2.5.3 Temporale Logik

Die Temporale Logik wurde von Prior [85] 1962 entwickelt und bildet eine spezielle Variante der Aussagenlogik. Diese Logiken beschreiben Eigenschaften von Zuständen in Systemen und betrachten den zeitlichen Ablauf eines reaktiven Systems [11]. Dabei berücksichtigen diese Logiken keine Zeitverläufe mit konkreten Zeitangaben, sondern nur die Reihenfolge der Abarbeitung von Zuständen.

Temporale Logiken bestehen aus speziellen temporalen Operatoren sowie aus sogenannten Pfadquantoren und boolschen Operatoren. Kombiniert man temporale Operatoren, Pfadquantoren, boolsche Operatoren und atomare Eigenschaften, so kann man bestimmte Abfolgen von Zuständen im Berechnungsbaum eines Systems beschreiben (Abbildung 2.3).

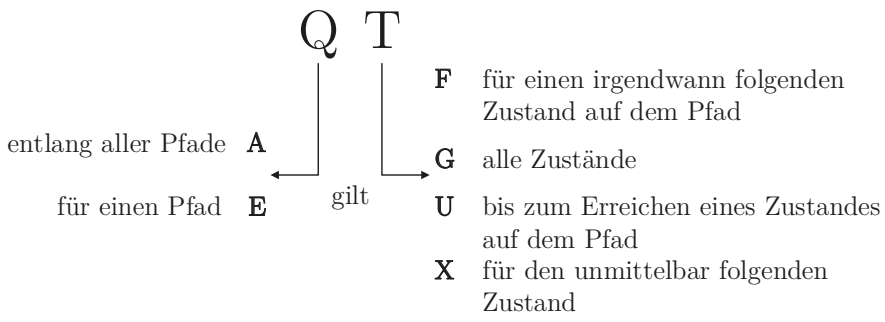


Abbildung 2.3: Kombination von Temporal-Operator und Pfadquantor

Pfadquantoren (**Q**) beschreiben dabei die Verzweigung von Berechnungsbäumen:

- **A** (**A**llquantor): Die Eigenschaften müssen entlang aller Pfade im Berechnungsbaum gelten.
- **E** (**E**xistenzquantor): Die Eigenschaften müssen entlang mindestens eines Pfades im Berechnungsbaum gelten.

Temporal-Operatoren (**T**) beschreiben das Verhalten von ausgewählten Pfaden in Berechnungsbäumen. Dabei gilt $p, q \in AP$

- $\mathbf{F}p$ („in the **F**uture„): Die Eigenschaft p muss in irgendeinem Zustand des Pfades erfüllt sein.
- $\mathbf{G}p$ („**G**lobally“): Die Eigenschaft p muss in allen Zuständen des Pfades erfüllt sein.
- $q\mathbf{U}p$ („**U**ntil“): Falls die Eigenschaft p in einem Zustand des Pfades erfüllt ist, so muss die Eigenschaft q in jedem vorhergehenden Zustand des Pfades erfüllt sein.
- $\mathbf{X}p$ („ne**X**t time“): Die Eigenschaft p muss im nächsten Zustand des Pfades erfüllt sein.

Im folgenden werden beispielhaft temporale Logiken auf Basis von [11, 30] vorgestellt. Diese unterscheiden sich in den Kombinationsmöglichkeiten von Temporal-Operatoren und Pfadquantoren sowie in deren Ausdrucksmächtigkeit.

Computational Tree Logic* (CTL*) [41] ist die ausdrucksmächtigste der drei folgenden Varianten der temporalen Logik. Diese Logik hat keine Einschränkungen, wie Temporal-Operatoren und Pfadquantoren miteinander kombiniert werden.

Es gibt dabei zwei Arten von Formeln:

- Zustandsformeln: beschreiben Eigenschaften eines bestimmten Zustandes innerhalb des Berechnungsbaumes.

- Pfadformeln: beschreiben Eigenschaften entlang eines bestimmten Pfades innerhalb des Berechnungsbaumes.

Für die Syntax von Zustandsformeln in *CTL** gilt:

- Wenn $p \in AP$, dann ist p eine Zustandsformel.
- Wenn f_1 und f_2 Zustandsformeln sind, dann sind $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ Zustandsformeln.
- Wenn f_1 eine Pfadformel ist, dann sind $\mathbf{A}f_1$ und $\mathbf{E}f_1$ Zustandsformeln.

Für die Syntax von Pfadformeln gilt:

- Wenn f_1 eine Zustandsformel ist, dann ist f_1 auch eine Pfadformel.
- Wenn f_1 und f_2 Pfadformeln sind, dann sind $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$, $\mathbf{F}f_1$, $\mathbf{G}f_1$, $f_1\mathbf{U}f_2$ und $\mathbf{X}f_1$ Pfadformeln.

Linear Temporal Logic (LTL) wurde von Pnueli [79] entwickelt und bildet eine temporale Logik mit linearem zeitlichen Verlauf. Damit ist es nur möglich, Formeln zu erstellen, die Eigenschaften von allen Pfaden eines Berechnungsbaumes gleichzeitig beschreiben. Verzweigungen in Pfade können nicht abgebildet werden. *LTL* ist eine Teilmenge von *CTL**, in der alle Formeln die Form $\mathbf{A}f$ haben. f ist dabei eine Pfadformel, in der nur atomare Aussagen als Zustandsformel gelten.
Für die Syntax von Formeln in *LTL* gilt:

- Wenn $p \in AP$, dann ist p eine Pfadformel.
- Wenn f_1 und f_2 Zustandsformeln sind, dann sind $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$, $\mathbf{F}f_1$, $\mathbf{G}f_1$, $f_1\mathbf{U}f_2$ und $\mathbf{X}f_1$ Pfadformeln.

Computational Tree Logic (CTL) wurde 1981 von Emerson und Clarke [29, 40] entwickelt und ist ebenfalls eine Teilmenge von *CTL**, bei der immer einer der Temporal-Operatoren direkt auf einen der Pfadquantoren folgen muss. Mit *CTL* Formeln können im Vergleich zu *LTL* Formeln auch Eigenschaften von einzelnen Pfaden im Berechnungsbaum beschrieben werden.
Für die Syntax von Zustandsformeln in *CTL* gelten dieselben Regeln wie für *CTL**.
Für die Syntax von Pfadformeln gilt im Gegensatz zu *CTL** nur folgende Regel:

- Wenn f_1 und f_2 Zustandsformeln sind, dann sind $\mathbf{F}f_1$, $\mathbf{G}f_1$, $f_1\mathbf{U}f_2$ und $\mathbf{X}f_1$ Pfadformeln.

Timed Computation Tree Logic (TCTL) wurde von Alur [4] bzw. Henzinger [50] Anfang der '90er als Erweiterung von *CTL* um explizite Zeitangaben bzw. Zeitvariablen entwickelt, um Bedingungen an sogenannten *Uhren* ausdrücken zu können, wie sie speziell bei Zeitautomaten (engl. timed automata - TA [5]) verwendet werden. Im Gegensatz zu *CTL* wird der Temporal-Operator Ne**X**t aufgrund der kontinuierlichen Verwendung von Zeit nicht unterstützt.

In dieser Arbeit werden Spezifikationen mittels *CTL* formal dargestellt. Daher werden in Abbildung 2.4 und Abbildung 2.5 die möglichen Kombinationsmöglichkeiten von Temporal-Operatoren mit Pfadquantoren beispielhaft in einem Berechnungsbaum dargestellt. Dabei gilt $p, q \in AP$ und der Startzustand ist s_0. Die Zustände, in denen p gilt, sind schwarz dargestellt und die, in denen q gilt, grau.

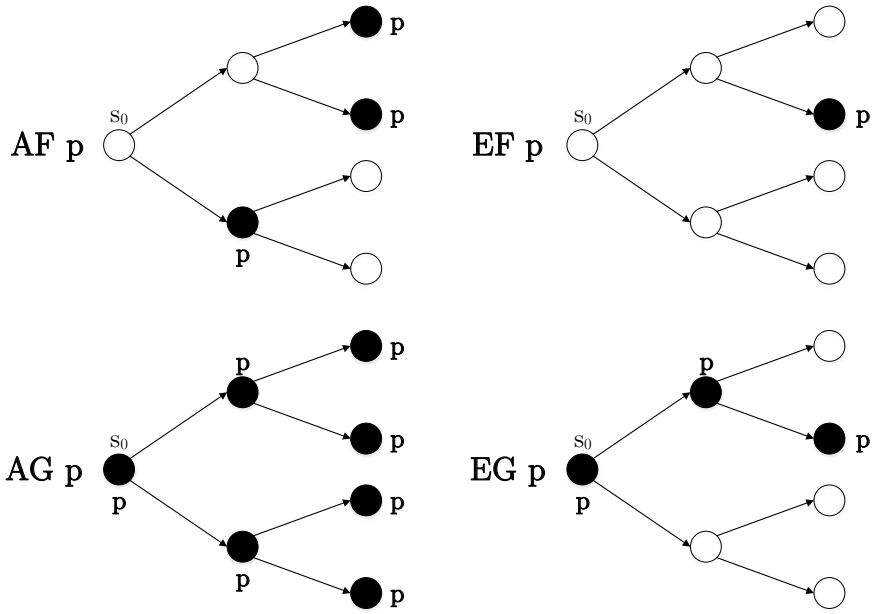


Abbildung 2.4: Visualisierung von beispielhaften *CTL* Formeln

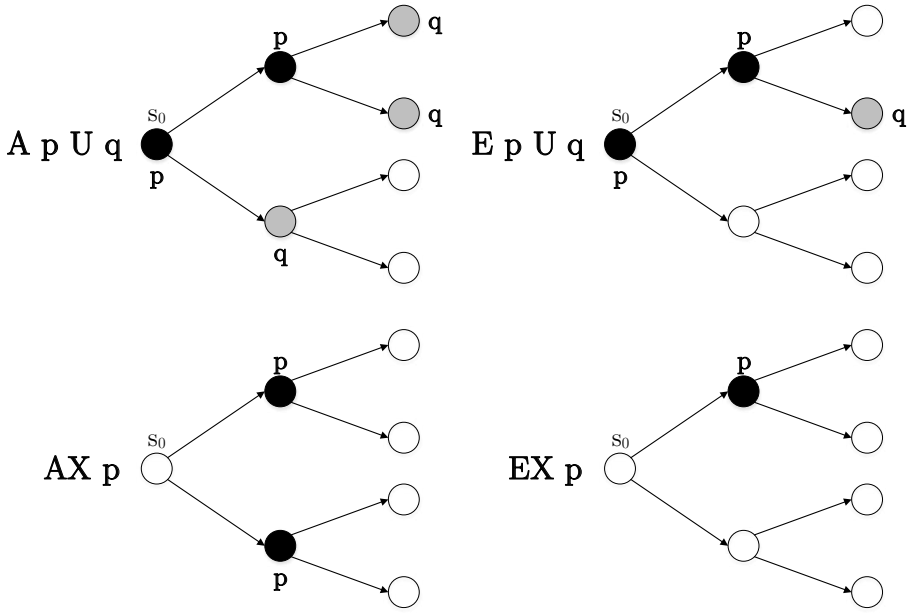


Abbildung 2.5: Visualisierung von beispielhaften *CTL* Formeln

2.6 Formale Verifikation

Verifikation bezeichnet den Nachweis, dass ein System bzw. Programm dessen spezifizierte Anforderungen korrekt erfüllt. Formale Verifikation erweitert diesen Nachweis um mathematische Exaktheit. Die Korrektheit dabei bezieht sich nach [11] immer auf die zugrundeliegende Spezifikation. Ein System bzw. Programm kann nie ohne Bezug auf eine Spezifikation als *korrekt* gelten.

> "The system is considered to be *correct* whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system." [11]

Um eine formale Verifikation durchführen zu können, wird sowohl eine formalisierte Beschreibung des Systems als auch der Spezifikation (vgl. Abschnitt 2.5) benötigt. Dabei erfolgt die Beschreibung des Systems in der Regel als mathematisch eindeutiges Modell und die Spezifikation in einer mathematischen Logik. Die Qualität der Resultate der formalen Verifikation hängt dabei maßgeblich von der Qualität der Modelle und der formalisierten Anforderungen ab.

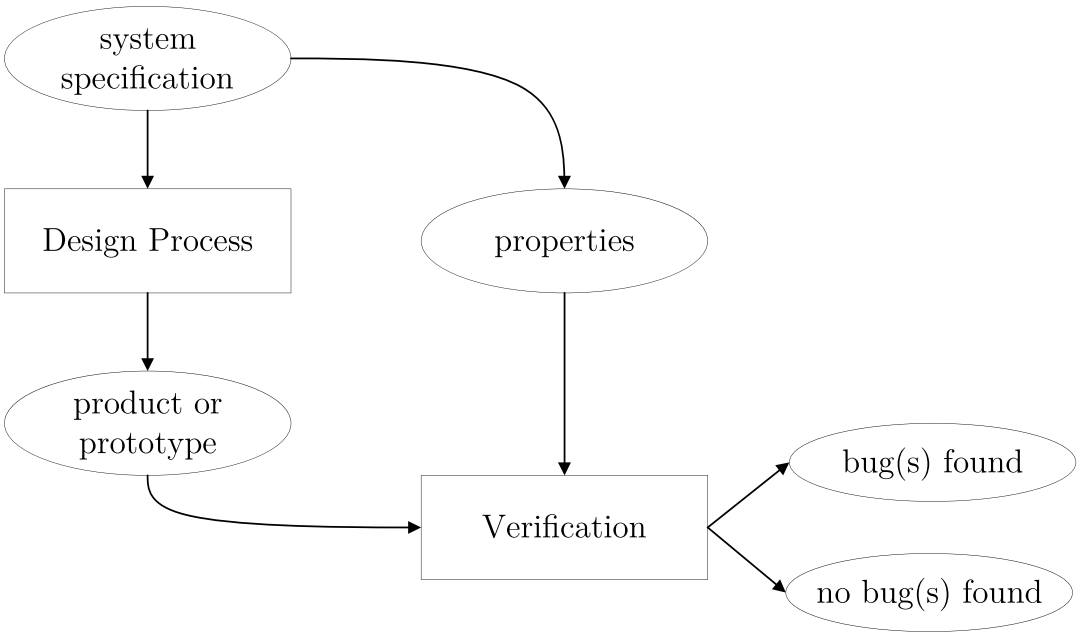


Abbildung 2.6: Schematische Darstellung der Systemverifikation nach [11]

Historisch gesehen zeigte Turing schon 1949 [74, 116] einen Beweis zur Korrektheit eines Programms zur Berechnung der Fakultät einer Zahl mit Hilfe von Additionen. Dabei wurde das Programm in kleinere Abschnitte zerlegt und deren Korrektheit gegenüber den Anforderungen manuell nachgewiesen. Daraus wurde die Korrektheit des gesamten Programms gefolgert.

Hoare verwendet 1969 in [51] Axiome, um die Korrektheit von Programmen zu beweisen, und stützt sich dabei auf die Arbeit von Floyd [44]. Hoare führt 1971 die formale Verifikation am Beispiel seines Sortier-Algorithmus FIND durch [52].

Pnueli schlug dann 1977 die Verwendung von Temporaler Logik zur Spezifikation von Anforderungen vor [79].

Unabhängig voneinander entwickelten Clarke und Emerson [29, 40] und Queille und Sikakis [88] Anfang der ’80er die Modell-Prüfung (engl. Model-Checking). Dabei wird anhand eines Modells geprüft, ob spezifizierte Eigenschaften darin erfüllt sind.

Mögliche Methoden zur Verifikation von Software werden im Folgenden kurz beschrieben:

Review Ein strukturierter Analyse- und Bewertungsprozess, der sowohl manuell als auch automatisch erfolgen kann und auf der Architektur- oder der Code-Ebene durchgeführt wird. Dabei wird das zu untersuchende Artefakt, wie z. B. der Programmcode, systematisch inspiziert und auf Fehler in der Semantik hin untersucht. Ebenfalls wird manuell geprüft, ob alle Anforderungen an das Artefakt erfüllt werden. Diese Methode ist nicht vollständig, und die Qualität der Ergebnisse ist stark abhängig vom Prüfer.

Statische Analysen Diese Methode wird eingesetzt, um z. B. den Programmcode zu analysieren, noch bevor das Programm auf der Zielplattform ausgeführt wird. Dabei finden z. B. Prüfungen auf Einhaltung von Kodier-Richtlinien, das Vorhandensein von doppeltem oder totem Code oder die Abschätzungen vom Speicherverbrauch des Systems statt. Weiterhin kann diese Methode genutzt werden, um Metriken zur Qualitätsbeurteilung zu berechnen.

Simulation Um frühzeitig im Entwicklungsprozess Funktionen losgelöst vom Gesamtsystem zu prüfen, werden Teile des Systems als Modell nachgebildet. Damit können notwendige Umgebungsbedingungen für die Funktion simuliert werden, ohne das Gesamtsystem vollständig implementiert zu haben. Die Qualität der Methode hängt maßgeblich von der Qualität der erstellten Simulationsmodelle ab.

Testen Hierbei wird auf der Basis von erstellten Testfällen das Programm manuell oder automatisiert überprüft. Ein Testfall kann dabei nur einen spezifischen Pfad im Programmablauf überprüfen und dabei Fehler finden. Diese Methode weist einen experimentellen Charakter auf, da nur eine endliche Anzahl an Testfällen betrachtet wird. Die Qualität der Ergebnisse dieser Methode hängt maßgeblich von der Qualität der einzelnen Testfälle ab. Eine hohe Testabdeckung wird dabei nicht zwangsläufig durch eine hohe Anzahl an Testfällen erzielt. Hierbei ist ein Abschätzen, ob die Anzahl der Testfälle ausreicht, schwierig. Beim modellbasierten Testen können Testfälle automatisch direkt aus dem Systemmodell abgeleitet und zur Verifikation verwendet werden. Dies erhöht die Qualität der Testfälle.

Alle aufgezeigten Methoden können Fehler finden, keine kann allerdings deren Abwesenheit beweisen.

> “Program testing can be used to show the presence of bugs, but never to show their absence!“ [33]

Methoden zur formalen Verifikation, die auch die Abwesenheit von Fehlern beweisen können, sind z. B. das Theorembeweisen (engl. Theorem-Proving) oder die Modellprüfung (engl. Model-Checking). Beim Theorem-Proving werden Behauptungen auf Grundlage eines Beweissystems verifiziert. Effizient geschieht dies meist nur unter interaktiver Einbeziehung eines Experten.

> “Theorem provers may also require an expert operator to be used effectively.“ [39]

Das Model-Checking wird im folgenden Kapitel ausführlicher beschrieben.

2.6.1 Model-Checking

Model-Checking bezeichnet die formale Methode zur automatischen Verifikation von Systemmodellen und den dazu aufgestellten Anforderungen durch Exploration aller erreichbaren Zustände des Programms und basiert auf den Arbeiten von Clarke und Emerson [29, 40], sowie Queille und Sikakis [88]. Dazu werden die Anforderungen formal spezifiziert und in temporal logische Formeln überführt (siehe Abbildung 2.7). Mittels eines passenden Modell-Prüfers (engl. Model-Checker) wird das ebenfalls formalisierte Systemmodell auf die gegebenen Korrektheitseigenschaften hin überprüft, wobei dies durch eine systematische Analyse des gesamten Transitionssystems mittels Suchalgorithmen erfolgt. Die Laufzeit dieser Algorithmen hängt von der Anzahl der Zustände des Transitionssystems ab [11]. Dabei kann diese Methode sowohl in der Hardware- als auch in der Software-Entwicklung eingesetzt werden.

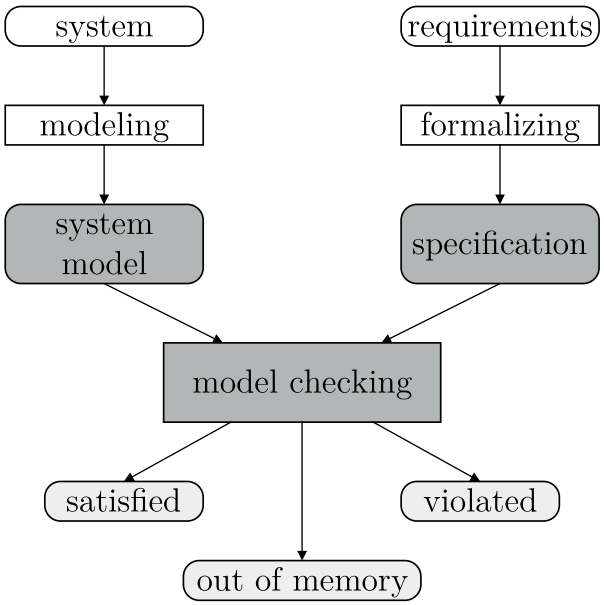


Abbildung 2.7: Prozessbeschreibung des Model-Checkings nach [11]

Üblicherweise bieten Model-Checker für die bessere Interpretation von Ergebnissen und zum besseren Verständnis des Systems sogenannte Gegenbeispiele oder Zeugen als Beweis der Existenz bzw. der Abwesenheit von Fehlern. Beides sind Pfade im Berechnungsbaum, die entweder das Erfüllen oder das Nicht-Erfüllen einer Formel zeigen. Basierend auf den Definitionen für CTL-Model-Checking von Baier und Katoen [11] und Clarke et al. [30] gilt:

- Gegenbeispiel: Ein Gegenbeispiel widerlegt die Gültigkeit einer Pfadformel mit Allquantor (z. B. $\mathbf{AG}f$ oder $\mathbf{AF}f$). Dazu zeigt dieses den entsprechenden Pfad im Berechnungsbaum auf.
- Zeuge: Ein Zeuge bestätigt die Gültigkeit einer Pfadformel mit Existenzquantor (z. B. $\mathbf{EF}f$ oder $\mathbf{EG}f$). Dazu zeigt dieser den entsprechenden Pfad im Berechnungsbaum auf.

Formal wird das Model-Checking wie folgt definiert:
M sei eine Kripke Struktur (z. B. ein Transitionsgraph) und f die Formel einer temporalen Logik. Der Model-Checking Algorithmus findet nun alle Zustände s in M, so dass gilt: $M, s \models f$

Dazu teilt [11] den Prozess des Model-Checkings in drei Phasen ein:

Modeling phase Das Modell des zu betrachtenden Systems sowie die zu prüfenden Anforderungen werden formalisiert.

Running phase Der Model-Checker prüft die Anforderung im System.

Analysis phase Die Ergebnisse werden ausgewertet. Dabei kann:

1. die Anforderung erfüllt sein, oder
2. die Anforderung nicht erfüllt sein, oder
3. der Model-Checker aufgrund der Zustandsraum-Explosion die Exploration des Zustandsraumes in einem sinnvollen Zeitrahmen nicht abschließen.

Explosion des Zustandsraumes

Ein wesentliches Problem beim Model-Checking ist die Zustandsraum-Explosion (engl. State-Space-Explosion). Diese beschreibt die exponentielle Zunahme der zu untersuchenden Zustände beim *naiven* Model-Checking-Prozess. Hierbei wird der gesamte, durch das System-Modell definierte, Zustandsraum ohne Einschränkungen auf die Gültigkeit der spezifizierten Anforderung hin überprüft. Folgendes Beispiel verdeutlicht das Problem: Ein Transitionssystem mit 10 Zuständen und 10 nicht-deterministischen Variablen, wovon 5 boolesche Variablen und 5 Integer (8-Bit, Wertebereich: 0..256) sind, bildet einen Berechnungsbaum mit $10 \cdot 2^5 \cdot 256^5 = 3{,}51 \times 10^{14}$ Zuständen. Erhöht man die Bit-Breite der Integer Variablen von 8-Bit auf 16-Bit (Wertebereich: 0..65535), so ergibt sich ein exponentieller Anstieg der möglichen Zustände zu $10 \cdot 2^5 \cdot 6553^5 = 3{,}87 \times 10^{26}$.

Clarke et al. [28] geben eine allgemeine Übersicht über vorhandene Abstraktionstechniken, um mit der Zustandsraum-Explosion umzugehen. Hierzu zählen „symoblic model checking“, „partial order reduction“, „counterexample-guided abstraction refinement“ und

„bounded model checking“. Weitere Maßnahmen, um den Zustandsraum zu verkleinern, sind je nach verwendetem Werkzeug (Model-Checker) unterschiedlich.

Im Allgemeinen zeichnen sich alle Abstraktionstechniken darin aus, dass diese von einer Über-Approximation des ursprünglichen Modells als Grundlage für das Model-Checking ausgehen. Dies hat zur Folge, dass Ergebnisse aus dem Model-Checking (Zeuge bzw. Gegenbeispiel) unter Umständen nicht im originalen Modell des Systems wiedergefunden werden können.

Vor- und Nachteile

Baier und Katoen listen die Stärken und Schwächen von Model-Checking in [11] auf. Im Folgenden werde diese Punkte zusammengefasst dargestellt:

Die Stärken:

- Anwendbarkeit auf eine große Spanne von Systemen, Hardware-Design als auch Software
- Partielle Verifikation von individuellen Eigenschaften
- Finden von wahrscheinlichen als auch unwahrscheinlichen Fehlern
- Liefern von Zusatzinformationen, um die Ergebnisse besser zu interpretieren (Zeuge bzw. Gegenbeispiel)
- Automatisierbarkeit durch Software-Werkzeuge
- Verstärktes Interesse aus der Industrie
- Integrierbarkeit in bestehende Entwicklungsprozesse
- Mathematische Beweisbarkeit

Die Schwächen:

- Schlecht geeignet für Anwendungen, die datenintensiv sind
- Anwendbarkeit begrenzt durch Unentscheidbarkeitsresultate
- Verwendbarkeit der Resultate hängt von der Qualität des Systemmodells ab
- Prüft nur spezifizierte Anforderungen und garantiert keine vollständige Korrektheit des Systems
- Problem der Zustandsraumexplosion
- Formale Spezifikation von Systemmodell und Anforderung erfordert Expertenwissen

- Das Werkzeug selber kann Fehler enthalten
- Generalisierungen können nicht geprüft werden

Abschließend halten Baier und Katoen [11], unter Berücksichtigung aller Vor- und Nachteile, Model-Checking für eine effektive Technik, um Fehler im Entwurf eines Systems zu finden und das Vertrauen in den untersuchten Systementwurf signifikant zu stärken.

2.7 PCC als Fallbeispiel

Um die in dieser Arbeit entwickelten Methoden zu evaluieren, wird ein Fallbeispiel einer realen Produktentwicklung (*Piezoelectric Controlled Carburetor (PCC)*) gewählt. Dabei handelt es sich um eine Steuergeräte-Plattform für den Einsatz von Verbrennungsmotoren mit einem elektronisch steuerbaren Vergaser als Gemischbildner. Diese Plattform wurde durch die Fa. VEMAC[3] entwickelt und bietet sowohl eine flexible Hardware als auch Software für unterschiedliche Applikationen. Die Haupteinsatzgebiete der Plattform liegen im Bereich von kleinen Verbrennungsmotoren für Zwei- und Dreiräder sowie für handgehaltene und stationäre Arbeitsmaschinen. Allen Anwendungsfällen gemein ist die Anforderung nach Reduktion von schädlichen Abgasen, um Umwelt und Menschen zu schützen. Diese Reduktion wird durch staatliche Regularien gefordert.

Die Basis der *PCC*-Plattform bildet eine flexible und kostengünstige Hardware. Um damit unterschiedliche Applikationen unterstützen zu können, ist die Hardware modular aufgebaut und kann je nach Bedarf unterschiedlich bestückt werten. Die Hauptkomponente dabei ist ein 16-Bit Mikrokontroller aus der R8C Familie der Fa. Renesas[4]. Weiterhin sind Schaltungen zur Signalkonditionierung sowie zur Steuerung von verschiedenen Aktuatoren vorgesehen. Der Mikrokontroller unterstützt dabei das Einlesen und Ausgeben von digitalen (DIO) und analogen (ADC) sowie pulsbreitenmodulierten (PWM) Signalen. Zur Kommunikation mit externen Peripheriegeräten stehen verschiedene serielle- sowie Bus-Schnittstellen zur Verfügung.

Abbildung 2.8 zeigt die Architektur der Steuergerätesoftware. Basierend auf der verwendeten Mikrokontroller-Hardware existieren verschiedene Hardware-Treiber, um direkt auf die entsprechenden Hardware-Komponenten zugreifen zu können. Die Treiber sind gemäß dem HIS[5] Standard implementiert und verfügen damit über eine Schnittstelle (I/O Library) zum direkten Zugriff auf die Hardware. Diese Schnittstelle verwaltet die Konfiguration der verschiedenen Kanäle der darunterliegenden Treiber. Eine einheitliche Schnittstelle zum Zugriff auf die Hardwaretreiber sowie auf einen speziellen Treiber für die direkte Hardwarekommunikation (complex driver) bietet die darüber liegende API-Schicht. Diese ermöglicht es den Funktionsmodulen, mit der Hardware zu interagieren. Die einzelnen Funktionsmodule variieren je nach Anwendungsfall und können aufgrund

[3] `http://www.vemac.de`
[4] `http://www.renesas.eu`
[5] Hersteller Initiative Software, `http://automotive-his.de`

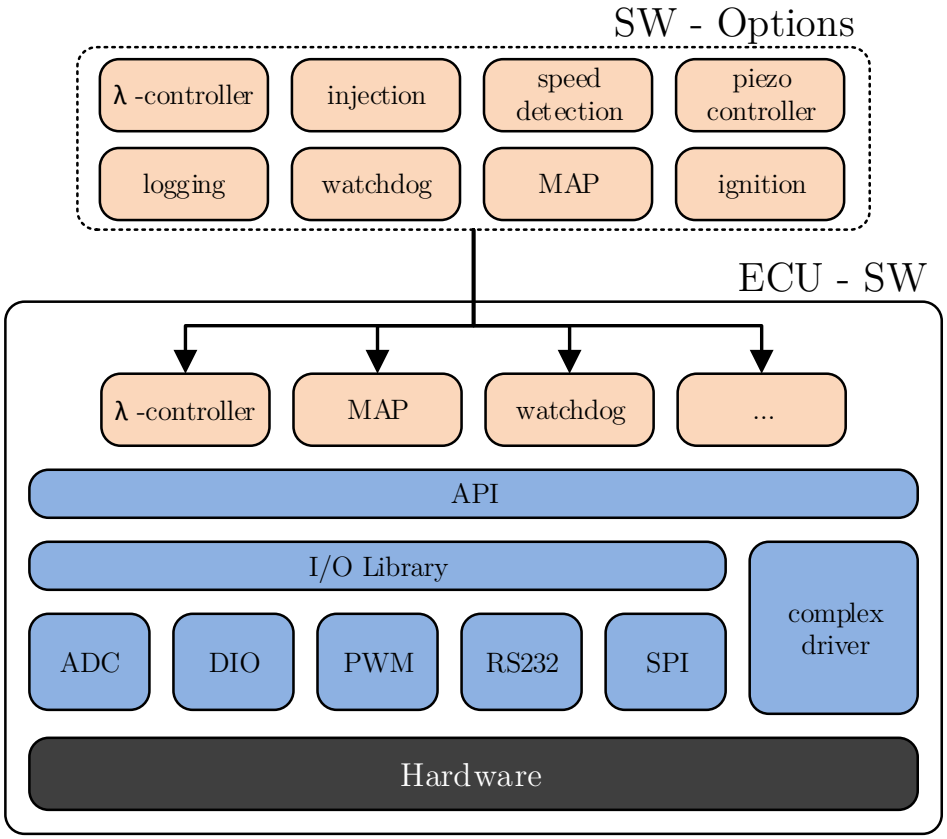


Abbildung 2.8: PCC Software-Architektur nach [RGN11]

der vorgesehenen Modularität ausgetauscht werden. Die möglichen Optionen sind als *SW-Options* in Abbildung 2.8 dargestellt. Um mehr Rechenleistung zur Verfügung stellen zu können, ist als Hardware-Option ein zweiter Prozessor möglich. Somit können die durch Software abgebildeten Funktionen bei Bedarf auf zwei Prozessoren verteilt werden. Diese sind mittels SPI-Bus als Master und Slave untereinander verbunden.

Das Mikrokontroller-System läuft ohne ein spezielles Betriebssystem. Die Abarbeitung der einzelnen Funktionen wird durch einen rudimentären Scheduler sichergestellt. Dieser führt speziell definierte Funktionen der einzelnen Module aus. Die Definition der Funktionen wird während der Entwicklung festgelegt und steht somit vor dem Kompilieren fest. Dabei kann die jeweilige Ausführungsrate aus vordefinierten Zeiten gewählt werden.

Die *PCC* Plattform kann in unterschiedlichen Szenarien eingesetzt werden (siehe Abbildung 2.9). Hierzu gehört die Anwendung als System zur Steuerung und Regelung von Piezo-Vergasern mit λ-Regelung in stationären Geräten [22] sowie in Zwei- bzw. Dreirädern [RGN11]. Weiterhin wird die Plattform auch zur Steuerung von konventionellen Verbrennungsmotoren in Rennkarts, Rasenmähern und unbemannten Flugzeugen eingesetzt.

Durch die vielfältigen Einsatzmöglichkeiten entstehen Varianten mit gleichen aber

auch unterschiedlichen Software-Komponenten, die alle auf einer gemeinsamen Basis-Software aufbauen. Für deren Verifikation bieten sich automatische Methoden an, um den Einsatz von Ressourcen zu reduzieren. Da einige Anwendungen Relevanz hinsichtlich der funktionalen Sicherheit aufweisen bzw. alle motorischen Anwendungen Einfluss auf die Emissionen haben, bietet sich der Einsatz von formalen Methoden an.

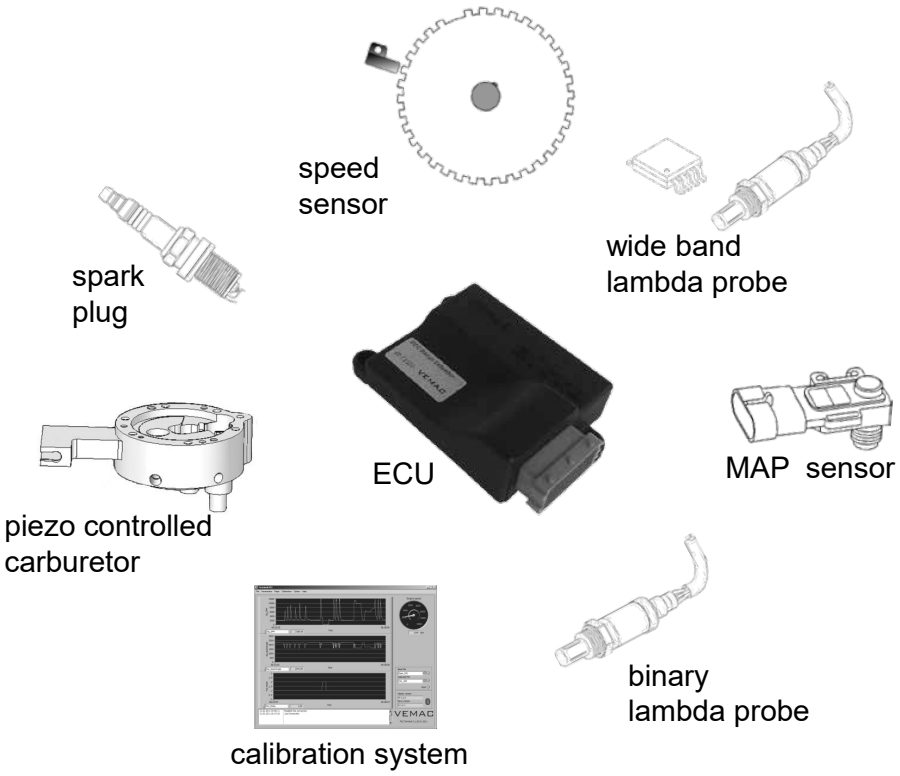


Abbildung 2.9: Unterschiedliche Sensoren und Aktuatoren für den Einsatz in verschiedenen Applikationen [RGN11]

2.8 Watchdog als Fallbeispiel

Ein weiteres Fallbeispiel zur Evaluierung der in dieser Arbeit vorgestellten formalen Methoden bietet die Watchdog-Funktionalität einer Kleinserien-Motorsteuerung. Hierbei wird derselbe 16-Bit Mikrokontroller aus der R8C Familie eingesetzt wie in Abschnitt 2.7.

Die Plattform zur Entwicklung von Steuergeräten für die Kleinserie (VeRa [RGK12, RG14, KWRG13]) ist für sicherheitskritische Funktionen vorbereitet [97]. Neben einem leistungsstarken Mikrokontroller (Haupt-CPU) und einem FPGA zum Verarbeiten von

modellgestützten Software-Funktionen, wie z. B. komplexe Regelalgorithmen für Verbrennungsmotoren, steht dem Entwickler ein 16-Bit Mikrokontroller für sicherheitskritische Funktionen zur Verfügung, die getrennt von der Haupt-CPU verarbeitet werden können. Zum Datenaustausch zwischen FPGA, Haupt-CPU und 16-Bit Mikrokontroller sind ein schneller serieller Bus sowie verschiedene diverse digitale Ein- und Ausgänge vorgesehen.

Eine mögliche Anwendung für die 16-Bit CPU ist die Überwachung der anderen beiden Recheneinheiten der Plattform. Dies geschieht durch Stimulation der entsprechenden Einheit über digitale Signale. Bleibt die korrekte Antwort innerhalb eines definierten Zeitfensters aus, nimmt der Watchdog einen Fehlerfall an. Daraufhin wird ein Reset des gesamten Systems durchgeführt, wobei die einzelnen Recheneinheiten zurückgesetzt werden und mit einem erneuten Hochfahren des Systems beginnen.

2.9 MULTIFORM

Ziel des MULTIFORM Projektes war es, Entwicklung, Integration von Werkzeugen und Methoden um einen vollständig modellbasierten Entwicklungsprozess von komplexen Steuerungssystemen zu unterstützen. Dabei können diese Systeme sowohl eigenständig als auch vernetzt sein. Durch den Einsatz von Modellierung, Verifikation und Testen sollen Fehler frühzeitig entdeckt und vermieden werden.

Der Schwerpunkt dabei lag auf der Interoperabilität zwischen verschiedenen Modellierungswerkzeugen und –methoden, ausgehend von unterschiedlichen Modellierungssprachen. Dabei sollten Werkzeuge von verschiedenen Abstraktionsebenen und unterschiedlichen Steuerungshierarchien miteinander verknüpft werden. Modelle dieser Werkzeuge sollten austauschbar gemacht werden, um deren Zusammenarbeit im übergeordneten Entwicklungsprozess besser unterstützen zu können. Dazu wurden Austauschformate zwischen den verschiedenen Werkzeugen entwickelt bzw. vorhandene Formalismen erweitert. Weiterhin wurde das *Design Framework (DF)* (siehe Unterabschnitt 2.9.1) entwickelt, welches auf den jeweiligen Entwicklungsprozess zugeschnitten werden kann und dabei die verschiedenen Modellierungswerkzeuge über offene Schnittstellen einbindet. Dadurch können die Konsistenz der für die Modelle notwendigen Parameter und Daten über den kompletten Entwicklungsverlauf erreicht werden.

Das Konsortium bestand dabei aus den folgenden Partnern: Technische Universität Dortmund, Technische Universiteit Eindhoven, Université Joseph Fourier Grenoble, RWTH Aachen, Aalborg Universitet, Stichting Embedded Systems Institute, VEMAC GmbH & Co. KG, KVCA A/S

2.9.1 Design Framework

Um Entscheidungen während der Entwicklung von komplexen Systemen nachvollziehbar zu machen und mit modellbasierten Methoden und Werkzeugen zu verknüpfen, wurde

das *DF* als generische Basis entwickelt. Im Folgenden werden angelehnt an [72, 73] die Struktur sowie die Eigenschaften des *DF* näher beschrieben.

Das *DF* ist ein generisches Framework, welches unabhängig von der jeweiligen Domäne des zu entwickelnden Systems auf den Entwicklungs-Prozess angepasst werden kann. Das *DF* besteht aus drei abstrakten Ebenen (siehe Abbildung 2.10):

Entwurfsfluss (engl. Design-Flow): Das Entwerfen wird als Aktivität betrachtet und besteht aus Entwurfsschritten und den dazugehörigen Entscheidungen, die zu den jeweiligen Schritten oder zu einem Ende führen. Dieses Ende tritt z. B. dann auf, wenn eine Entwurfsaktivität nicht zu dem gewünschten Zielsystem führt oder eine Alternative bessere Ergebnisse erzielt. Daraufhin wird dieser Entwurfsast nicht mehr weiter betrachtet, zwecks Dokumentation aber weiterhin aufgehoben.

Entwurfssichten (engl. Design-Views): Eine Sicht kann verschiedene Modelle und Parameter-Sätze enthalten, um in dieser Kombination einen speziellen Aspekt des Systems oder einer Komponente zu untersuchen. Diese enthält Blöcke, um die Systemstrukturen abzubilden.

Modelle (engl. Model): Ein Modell ist eine formale Beschreibung des Systems oder einer Komponente, um tiefgehende Erfahrungen über deren Verhalten zu erlangen. Dabei können unterschiedliche Formalismen verwendet werden, um spezifische Aspekte untersuchen zu können.

Der Entwurf startet mit abstrakten Modellen des gesamten Systems. Diese werden in den darauffolgenden Entwurfsschritten immer weiter verfeinert. Experimente werden dazu genutzt, um unterschiedliche Aspekte wie z. B. Scheduling-Analyse oder die Identifikation von Problemen beim Entwurf frühzeitig zu analysieren. Dazu werden die verschiedenen Informationen in einer oder mehreren Entwurfssichten strukturiert abgelegt.

Das *DF* dient als Unterstützung des Entwicklungsprozesses von komplexen Systemen und bildet eine Plattform für die Zusammenarbeit von interdisziplinären Spezialisten. Es vereinfacht den Umgang mit kontinuierlichen Änderungen an den Spezifikationen und verbindet unterschiedliche Modellformalismen (Syntax und Semantik) von verschiedenen Systemkomponenten. Dabei werden sowohl formale als auch nicht-formale Aktivitäten im Entwicklungsprozess unterstützt. Weiterhin dokumentiert das *DF* den dynamischen Fluss von Informationen und das Treffen von Entwurfsentscheidungen inklusive der dazugehörigen Beweggründe. Es ist eine Anbindung von Modellierungswerkzeugen wie z. B. UPPAAL[6], gPROMS[7], Modelica[8], MATLAB[9] vorgesehen, und es werden die dazu passenden Werkzeuge zur Transformation über *Compositional Interchange Format (CIF)*

[6] `http://www.uppaal.org`
[7] `http://www.psenterprise.com`
[8] `https://www.modelica.org`
[9] `http://www.mathworks.com`

[117] in andere Modell-Formalismen unterstützt. Weiterhin können Werkzeuge zur Datenrepräsentation wie z. B. SVN[10] und IBM Doors[11] verwendet werden.

Um Entwurfsentscheidungen zu treffen, können die Ergebnisse von sogenannten Experimenten analysiert werden. Dazu werden Modelle mit einem ausgewählten Parametersatz ausgeführt und deren Resultate abgespeichert und ausgewertet. Durch die konsistente Dokumentation und Verwaltung von Modellen, Parametern und Ergebnissen sind Experimente reproduzierbar und vergleichbar. Die Möglichkeit zur Ausführung von Skripten erweitert den Funktionsumfang. Weiterhin können automatisch Konflikte zwischen Parametern auf Basis derer Werte und Verknüpfungen in einer frühen Phase des Systementwurfs aufgedeckt werden.

Die Darstellung des Entwicklungsprozesses ist in verschiedenen Sichten möglich, z. B. entlang einer Zeitachse. Die Entwurfsschritte können für parallel arbeitende Teams aufgeteilt und abschließend wieder zusammengefasst werden. Einzelne Komponenten, die im sogenannten *Core Domain Knowledge* hinterlegt wurden, können in anderen Projekten wieder verwendet werden.

[10] `https://subversion.apache.org`
[11] `https://www.ibm.com`

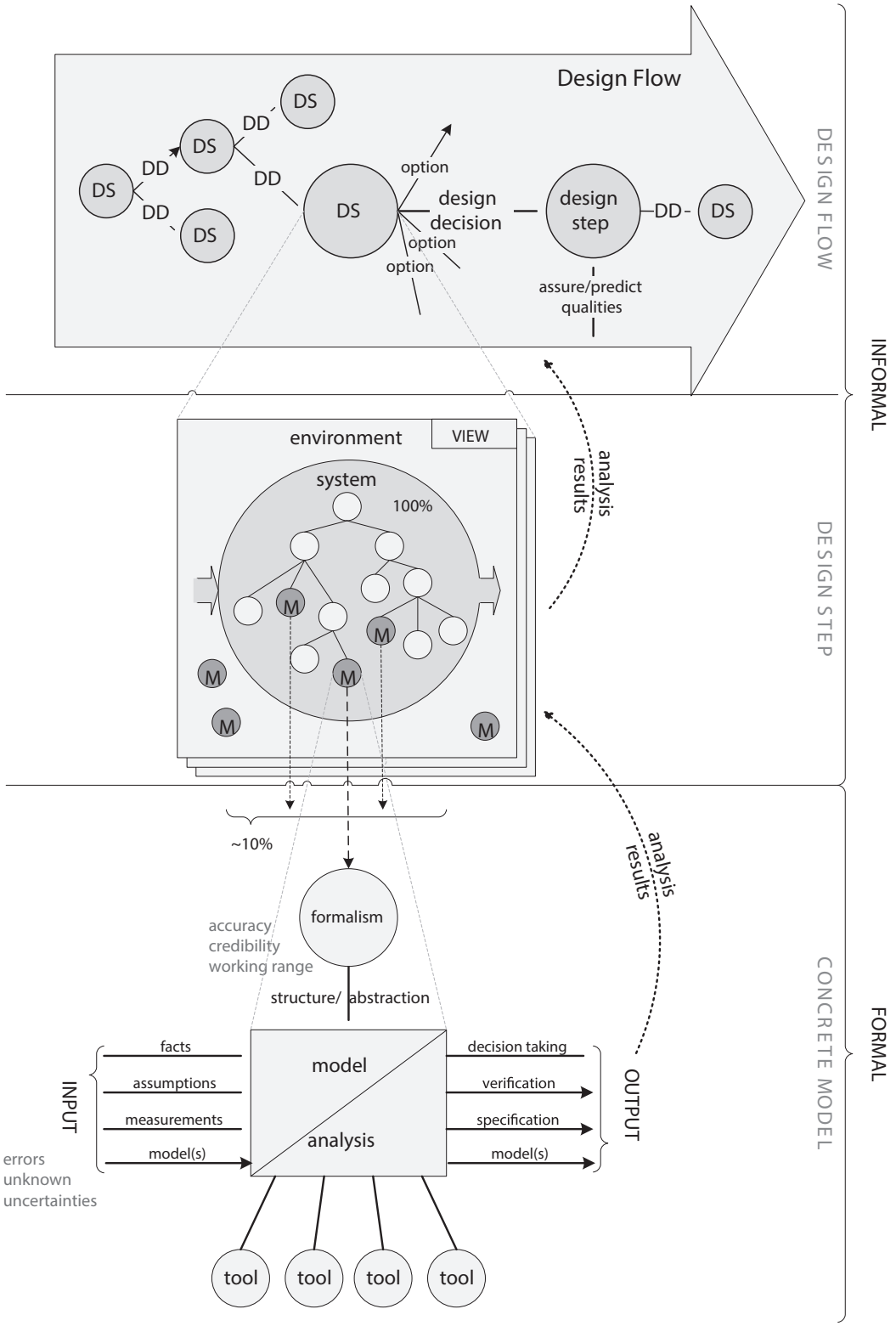


Abbildung 2.10: Modell des Design Frameworks [72]

2.10 KMU-orientierter Entwicklungsprozess

> Laut EU Definition [43] gilt: „Die Größenklasse der Kleinstunternehmen sowie der kleinen und mittleren Unternehmen (*kleine und mittlere Unternehmen (KMU)*) setzt sich aus Unternehmen zusammen, die weniger als 250 Personen beschäftigen und die entweder einen Jahresumsatz von höchstens 50 Mio. EUR erzielen oder deren Jahresbilanzsumme sich auf höchstens 43 Mio. EUR beläuft."

Um die in dieser Arbeit zu entwickelnden formalen Methoden in einem *KMU* einzusetzen, bedarf es eines entsprechenden Entwicklungsprozesses. Dieser muss abgestimmt sein auf die Anforderungen, die sich aus der jeweiligen Domäne ergeben. In dieser Arbeit steht die Entwicklung von Steuergeräten und der dazu passenden Software für die Automobilindustrie im Fokus. Dazu werden im Folgenden zuerst die Anforderungen an einen solchen Entwicklungsprozess festgelegt und dann ein bestehender Prozess eingeführt und beschrieben.

2.10.1 Anforderungen

Anforderungen an einen KMU-orientierten Entwicklungsprozess:

- Die enge Verzahnung von Zulieferer und Automobil-Hersteller fordert Kompatibilität in Bezug auf die Lieferantenbewertung der Hersteller. Hierzu muss es möglich sein, den Prozess nach Automotive SPICE (de-facto Standard von HIS[12] zur Lieferantenbewertung in der Automobilindustrie) bewerten zu lassen.
- Im Vergleich zu großen Unternehmen werden die Projekte in *KMU* meistens von kleinen Teams an einem Standort abgearbeitet. Die damit verbundene Flexibilität soll durch einen *leichtgewichtigen* Prozess unterstützt werden.
- Das Projektmanagement des zu entwickelnden Prozesses darf nur einen kleinen Teil der Arbeit ausmachen, muss jedoch ein Mindestmaß an Budget- und Ressourcenübersicht bieten.
- Der Prozess soll sich am V-Modell (Abschnitt 2.1) orientieren, einem de-facto Standard der Automobil-Industrie.
- In den frühen Phasen der Entwicklung sollen erste Prototypen verfügbar sein, um diese mit dem Kunden diskutieren und auf Änderungen frühzeitig reagieren zu können. Weiterhin kann der Kunde diese Prototypen früh in der eigenen Systementwicklung verwenden, um Zeit und Kosten zu sparen.

[12] Hersteller Initiative Software, `http://automotive-his.de`

- Um Kosten bei der Durchführung von weiteren Projekten einsparen zu können, ist es wichtig, Teilprozesse und deren Ergebnisse einfach wiederverwenden zu können. Dazu gehört auch die Möglichkeit, Entscheidungen für oder gegen einzelne Entwicklungsschritte nachvollziehbar zu machen, auch über Projektgrenzen hinweg. Weiterhin muss zur Fehlervermeidung die Konsistenz aller verwendeten Artefakte sichergestellt werden.
- Die Qualität der Software-Entwicklung soll gesteigert und dazu formale Methoden eingesetzt werden. Dies erfordert die Einbindung von passenden Formalismen zur Beschreibung von Anforderungen und Systemen sowie der entsprechenden Werkzeuge zur Umsetzung der jeweiligen Methoden.

2.10.2 Bestehender Entwicklungsprozess

Ein möglicher Entwicklungsprozess, der Teile der oben genannten Anforderungen abdeckt und auf kleine Unternehmen ausgelegt ist, wird von Reke in [93] beschrieben. Dabei liegt der Fokus auf der Entwicklung von modellbasierter Software für Steuergeräte für die Automobilindustrie. Um die Kompatibilität des Prozesses zu den Entwicklungsprozessen der großen Automobilhersteller zu zeigen, hat Reke den vorgestellten Prozess anhand von Automotive SPICE bewertet und das Erreichen des ersten SPICE Reifegrades festgestellt. Der gesamte Prozess ist in Abbildung 2.11 dargestellt und kann sowohl auf die Hardware- als auch auf die Software-Entwicklung bei automotiven Steuergeräten angewendet werden.

Zur Prozessbeschreibung wurde *Business Process Model and Notation (BPMN)* in der Version 2.0[13] verwendet. Diese Notation wird für die Dokumentation und Beschreibung von Geschäftsprozessen eingesetzt und durch die *Object Management Group* spezifiziert. Der Prozess ist angelehnt an das V-Modell und ist aufgeteilt in Teile, die vom *KMU* und vom Kunden durchgeführt werden. Dabei beginnt jede Entwicklung mit der *Erfassung der Anforderungen* in enger Zusammenarbeit mit dem Kunden. Daraus wird die sogenannte *Anforderungsbasline* erstellt, die als Vertragsgrundlage zwischen dem Kunden und dem *KMU* dienen kann. Weiterhin wird diese als Basis zur *Konzepterstellung* beim *KMU* sowie auch für den *Integrationstest* beim Kunden verwendet. Die aus dem Konzept erarbeitete Softwarespezifikation wird zum einen zum Erstellen der Software und zum anderen zum Erstellen von Testfällen für den Systemtest genutzt. Abschließend wird das erstellte System beim Kunden in einem *Integrationstest* verifiziert. Begleitend existieren sogenannte Unterstützungsprozesse zum *Projektmanagement* und für das Durchführen von *Reviews* sowie für das *Wiederverwenden* einzelner Prozessergebnisse. Wie üblich bei der Vorgehensweise nach dem V-Modell muss beim Auftreten von Fehlern bzw. von Änderungswünschen zu der entsprechenden Phase des absteigenden Astes gewechselt und der gesamte Prozess von dort an erneut durchlaufen werden. Dabei können zwar bestehende Ergebnisse wiederverwendet und angepasst werden, allerdings ist dies immer mit einer Steigerung der Projektkosten verbunden. Anwendung findet

[13] http://www.omg.org/spec/BPMN/2.0/

dieser Entwicklungsprozess in Industrieprojekten speziell für das Kleinserien-Steuergerät VeRa [RGK12, RG14, KWRG13].

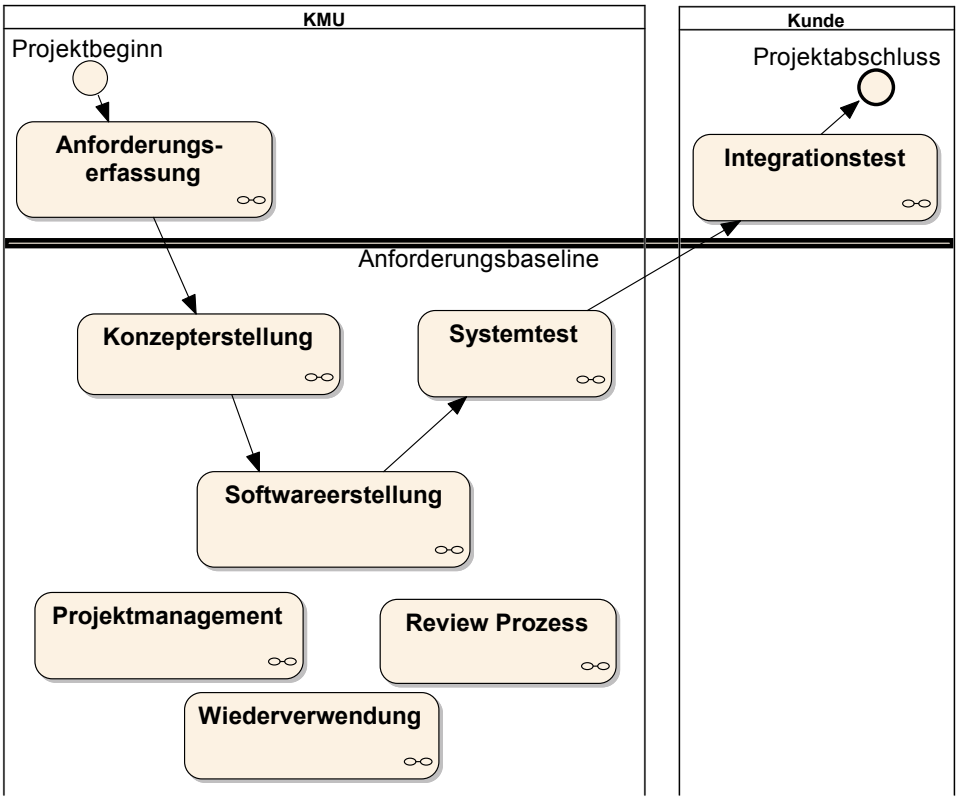


Abbildung 2.11: KMU-orientierter Software-Entwicklungsprozess aus [93]

3 Varianten-basierte Einplanbarkeitsanalyse

Bei der Entwicklung von Steuergeräten für kleine- bis mittelgroße Serien durch kleine Unternehmen treten sowohl weitreichende wirtschaftliche wie auch technische Herausforderungen direkt zu Beginn der Angebotsverhandlungen mit dem Kunden auf. Die Kostenabschätzung muss eng an der Realität ausgerichtet sein, um zum einen den Kunden durch ein zu hohes Angebot nicht zu verlieren und zum anderen die internen Kosten nicht zu überschreiten. Weiterhin müssen Risiken durch kostspielige Rückrufaktionen des Produktes von Anfang an minimiert werden. Durch frühzeitige Bereitstellungen von funktionalen Prototypen soll das Vertrauen des potenziellen Kunden in die technischen Fähigkeiten des Unternehmens gestärkt werden. Diese frühen Prototypen unterstützen zusätzlich die Aufwands- und Kostenabschätzung der Produktentwicklung.

Um so früh wie möglich eine konkrete Vorstellung der technischen Umsetzung zu erhalten, ist die Wiederverwendung von bereits existierenden Komponenten sowohl in der Hardware- als auch in der Software-Entwicklung eine kosteneffiziente Option. Die Entwicklung ist nicht nur schneller und kostenminimierter, sondern durch den Einsatz von gereiften Komponenten auch qualitativ hochwertiger. Das Risiko eines Ausfalls dieser Komponenten ist gering.

Der Volkswagen-Konzern setzt das Wiederverwenden von Komponenten über mehrere Automobil-Modelle seiner unterschiedlichen Marken konsequent in Form seines Modularen Querbaukastens (MQB) [2] um. Dadurch können z. B. Motoren, Antriebssysteme und auch Infotainment-Systeme gemeinsam bei Zulieferern kosteneffizient eingekauft werden. Die Differenzierung unter den jeweiligen Modellen erfolgt speziell durch eigene Designelemente und Funktionalitäten, die durch Software realisiert werden können. Die Stückzahlen der Zulieferer steigen dadurch, und die Anzahl der verschiedenen Produkte wird durch den Einsatz von Varianten abgelöst. Varianten zeichnen sich dabei durch einen hohen Anteil gemeinsam genutzter Komponenten aus und werden in Hardware z. B. mit unterschiedlichen Bestückungen und in Software z. B. durch verschiedene Konfigurationen abgebildet. Nachteile für die Zulieferer sind dabei die starke Abhängigkeit zu einem Kunden sowie das Risiko, dass durch Fehler direkt mehrere Modellreihen betroffen sein können.

Ein weiteres Beispiel eines Software-Baukastens stellt PERSIST [95, 96] dar. Damit werden bei der modellbasierten Entwicklung von Applikationssoftware für die Steuerung von Verbrennungsmotoren Teilmodelle von Komponenten in unterschiedlichen Anwendungen wiederverwendet. Durch ein entsprechendes Framework werden diese Varianten

verwaltet und deren Qualität durch kontinuierliche Validierung und Verifikation sicher gestellt.

Da immer mehr Funktionalitäten durch Software abgebildet werden, steigen die Anforderungen an die verwendeten eingebetteten Systeme in Bezug auf die Rechen- und Speicherkapazität sowie an die integrierten Peripherie Module wie z. B. Bus-Schnittstellen (z. B. CAN, LIN) oder digitalen- und analogen Ein- sowie Ausgängen. Die in den Produktkatalogen der Mikrokontroller-Hersteller vorhandenen zahlreichen Derivate an Recheneinheiten bieten eine große Spanne an möglichen Ausstattungen. Angefangen von kleinen Taktraten im kHz-Bereich und wenig internem Speicher (wenige kB) bis hin zu GHz-Taktraten und MB-Flash-Speichern.

Bei der Auswahl der passenden Mikrokontroller sind die entsprechenden Anforderungen sowie die Fertigungskosten zu berücksichtigen. Die Fertigungskosten hängen von dem ausgewählten Mikrokontroller-Derivat ab. Die Anforderungen an Speicher, Taktrate und peripheren Schnittstellen müssen von diesem erfüllt werden. Bei der Auswahl des Derivats müssen Abschätzungen für den Ressourcen-Verbrauch der benötigten Funktionen gemacht werden. Schätzt man deren Verbrauch zu großzügig ab, muss ein unnötig kostenintensiveres Derivat ausgewählt werden. Fällt die Abschätzung jedoch zu gering aus, können Anforderungen an den Speicherbedarf und an das zeitliche Verhalten der Software ggf. nicht mehr erfüllt werden. Dies kann zu unerwünschten Zuständen des Gesamtsystems führen und muss somit schon im Vorhinein vermieden werden.

Ziel muss es sein, so früh wie möglich im Entwicklungsprozess eine möglichst präzise Abschätzung von Speichernutzung, Rechenzeit und peripheren Schnittstellen zu erhalten sowie Software-Komponenten einfach wiederzuverwenden und erweitern zu können.

In Abschnitt 3.1 wird eine Analyse von Zeitanforderungen für eingebettete Systeme basierend auf der formalen Methode des Model-Checkings vorgestellt. Im darauf folgenden Abschnitt 3.2 wird ein Framework eingeführt, welches zum einen die Verwaltung von Software-Varianten enthält und zum anderen die Abschätzung des Ressourcenverbrauchs integriert. Die Evaluierung der vorgestellten Konzepte erfolgt jeweils anhand einer Plattform zur Entwicklung von Motorsteuergeräten, welche in Abschnitt 2.7 vorgestellt wurde.

3.1 Analyse von Zeitanforderungen

In diesem Kapitel wird das Einhalten von Anforderungen an das zeitliche Verhalten von eingebetteten Systemen analysiert. Dabei wird eine Methode entwickelt, ein Task-System dahingehend zu überprüfen, ob alle im System aktiven Tasks ihre vorab definierten Zeitgrenzen bezüglich der Ausführungsdauer einhalten. Die entwickelte Methode basiert dabei auf formalen Techniken und liefert damit einen Beweis für das Einhalten der Anforderung in jeglicher Situation, die das System durchlaufen kann.

3.1.1 Aufbau eines Task-Systems

Ein eingebettetes System besteht im Normalfall aus einem Betriebssystem, verfügbaren Ressourcen wie z. B. Speicher und Rechenzeit und sogenannten Tasks, über die Funktionen (z. B. Steuerung eines Verbrennungsmotors, Navigation des Fahrzeugs) abgebildet werden. Dabei stellt das Betriebssystem nach [113] den Ressourcenmanger des Systems dar, der neben der Verwaltung von Speicher auch die Zuweisung von Prozessorzeit übernimmt. Dies erfolgt durch den sogenannten Scheduler, der die Aufgaben auf Basis eines Algorithmus abarbeitet. Bei [68] heißt es dazu:

> „A scheduling algorithm is a set of rules that determine the task to be executed at a particular moment."

Man unterscheidet zwischen statischen und dynamischen Scheduling-Verfahren. Bei den statischen Verfahren ist die Ausführungsreihenfolge zum Zeitpunkt des Kompilierens bekannt. Somit ist das Verfahren deterministisch, da zu jedem Zeitpunkt vorab fest steht, welcher Task ausgeführt wird. Bei dynamischen Verfahren wird die Prozessorzeit dynamisch während der Laufzeit des Systems den einzelnen Tasks zugewiesen. Dabei kann z. B. sowohl die Ausführungszeit des Tasks als auch die Anzahl der jeweiligen Tasks variieren.

Speziell in eingebetteten Systemen wird die Abarbeitung von Tasks oft durch externe Ereignisse beeinflusst. Diese Systeme arbeiten eng mit der Hardware zusammen. Durch sogenannte Interrupts [25] kann eine Softwarefunktion mit einem externen Ereignis verbunden werden. Tritt dieses Ereignis ein, so wird die entsprechende Funktion ausgeführt. Das Ausführen dieser Funktion kann je nach Implementierung den aktuell ausgeführten Task unterbrechen. Dieses Verhalten wird präemptiv genannt. Scheduling-Algorithmen können ebenfalls präemptiv arbeiten und laufende Tasks unterbrechen, um anderen Tasks Rechenzeit zu gewähren. Können im System mehrere Interrupts gleichzeitig aktiv werden bzw. existieren mehrere Tasks, die eine Unterbrechung des aktuellen Tasks fordern, so muss es eine eindeutige Möglichkeit geben, zu entscheiden, welcher Interrupt bzw. Task als nächstes ausgeführt werden soll. Zu diesem Zweck existiert sowohl für das Scheduling von Interrupts wie auch von Tasks, die Möglichkeit, Prioritäten zuzuordnen. Diese Prioritäten können sowohl statisch als auch dynamisch während der Laufzeit vergeben und angepasst werden.

Tasks in einem eingebetteten System können unterteilt werden in periodische und a-periodische Tasks. Ein periodischer Task wird in einem fest definierten Zeitabstand regelmäßig ausgeführt. Dieser Zeitabstand wird als Periode (T) bezeichnet. Die Zeit, die der Task benötigt, um ausgeführt zu werden, wird als Ausführungszeit (C) bezeichnet. Abbildung 3.1 zeigt exemplarisch die Abarbeitung eines Systems mit drei verschiedenen Tasks mit jeweils unterschiedlichen Prioritäten. Alle Tasks können unterbrochen werden und auch andere Tasks unterbrechen. *Task 2* und *Task 3* werden in dem Beispiel periodisch und *Task 1* a-periodisch ausgeführt.

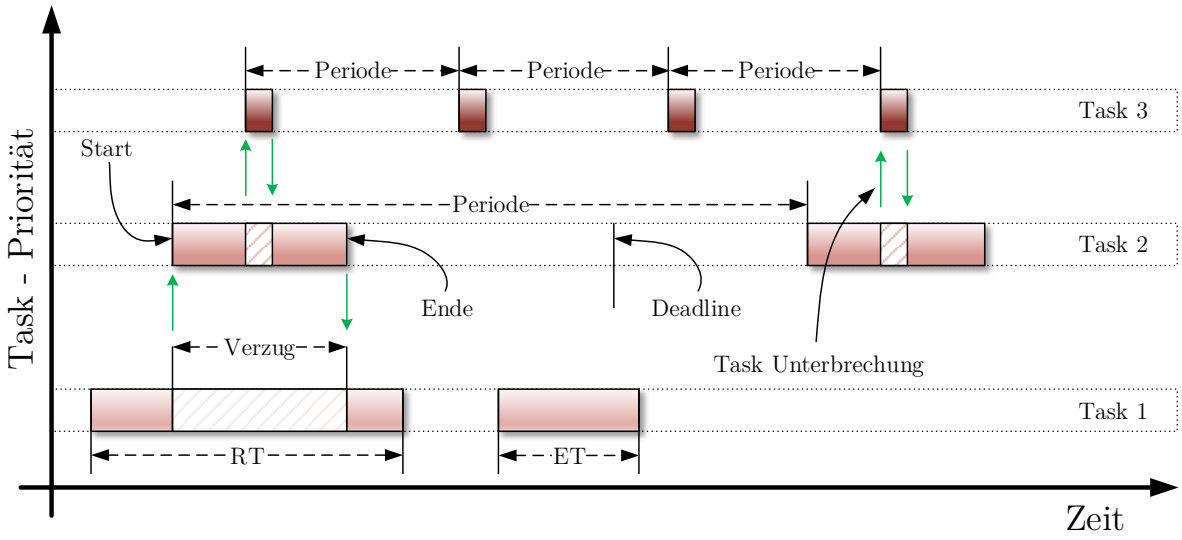


Abbildung 3.1: Prioritätengesteuertes Task System

Neben einer optionalen Ausführungsperiode besitzt jeder Task auch eine sogenannte *Execution Time (ET)* sowie eine *Response Time (RT)*. Als *ET* wird die Zeit bezeichnet, die der Task für die Ausführung im aktuellen Systemzustand benötigt und ist in einem Task-System mit nicht unterbrechbaren Tasks gleich der *RT*. Der Systemzustand ist wiederum abhängig von anderen Tasks, aber auch von den Zuständen der peripheren Hardware. Beinhaltet ein Task z. B. eine bedingte Verzweigung, die abhängig von einem digitalen Eingang der Hardware ist, so kann das Durchlaufen eines der Pfade durch eine Verzweigung sehr kurz und das eines anderen hingegen sehr lange dauern. Die kürzest-mögliche Ausführungszeit eines Tasks wird mit *Best Case Execution Time (BCET)* bezeichnet und die längst-mögliche Zeit mit *Worst Case Execution Time (WCET)*.

Betrachtet man die Möglichkeit, dass ein Task (z. B. *Task 1* in Abbildung 3.1) durch andere Tasks (*Task 2* und *Task 3*) unterbrochen wird, so variiert *ET* von *RT* des Tasks. *RT* beschreibt die Antwortzeit des Tasks, welche mindestens so groß ist wie *ET* und zusätzlich auch die Verzugszeit mit einschließt, die durch unterbrechende Tasks verursacht wird. Wird ein Task nicht unterbrochen, so gilt $ET = RT$. Ähnlich der Ausführungszeit existiert auch für die Antwortzeit eine kürzest-mögliche *Best Case Response Time (BCRT)* sowie eine längst-mögliche *Worst Case Response Time (WCRT)*.

Tasks in eingebetteten Systemen müssen speziell in sicherheitskritischen Applikationen (z. B. das Airbag System) Anforderungen an deren Ausführungszeiten erfüllen [25]. Dies bedeutet, dass Tasks, nachdem diese aktiviert wurden, in einer definierten Zeitspanne ausgeführt werden müssen. Diese Frist wird für jeden Task durch dessen Deadline (D) beschrieben. Überschreitet ein Task während seiner Ausführung diese Frist, so sind die Echtzeitbedingungen des Systems verletzt. Je nach Applikation kann das System z. B. in einen Fehlerzustand wechseln, oder es besitzt andere Mechanismen, um auf dieses Ereignis zu reagieren. Als grundsätzliche Beziehung zwischen Periode, Deadline und

Ausführungszeit gilt nach [68]:

$$C \leq D \leq T \tag{3.1}$$

3.1.2 Anforderungen

Es soll eine Methodik erarbeitet werden, welche es dem Entwickler ermöglicht, auf Basis von vorangegangenen Projekten und mit Wissen über das Verhalten von neu entstehenden Software-Modulen, eine Abschätzung des Systemverhaltens bei einer speziellen Task-Zusammenstellung zu erhalten. Exemplarisch für ein eingebettetes System wird die in Abschnitt 2.7 vorgestellte Steuergeräte-Plattform betrachtet und deren verschiedene Tasks analysiert, um daraus Anforderungen abzuleiten. Zusätzlich fließen auch Anforderungen aus damit abgeschlossenen sowie zukünftigen Projekten des durchführenden kleinen Unternehmens mit in die Betrachtung ein. Im Folgenden sind die Anforderungen aufgelistet:

1. Durch das Analyseverfahren soll das zeitliche Verhalten von Taskzusammenstellungen analysiert und deren Einplanbarkeit garantiert werden können.
2. Die Anwendbarkeit der Methode soll nicht auf ein spezielles Verfahren zum Scheduling begrenzt sein, sondern flexibel auf das zu untersuchende System angepasst werden können.
3. Das zu untersuchende System kann sowohl präemptive als auch nicht-präemptive sowie periodische und a-perodische Tasks beinhalten. Dabei ist die Anzahl der Tasks nicht vorgegeben.
4. Die Methodik sollte automatisiert ablaufen können, um diese in ein umfassendes Framework integrieren zu können.
5. Die Task-Prioritäten werden durch funktionale Anforderungen innerhalb des Gesamtsystems vorgegeben und können unabhängig von der Task-Deadline oder dessen Periode sein.
6. Um die Methodik in zukünftigen Anwendungen einsetzen zu können, muss diese die Möglichkeiten bieten, Multiprozessor-Systeme analysieren sowie System-Ressourcen abbilden zu können.

Um die aufgestellten Anforderungen zu erfüllen, werden im Folgenden verschiedene Verfahren zu Einplanbarkeitsanalyse betrachtet.

3.1.3 Scheduling Analyse, Related Work

Nachdem in Unterabschnitt 3.1.1 definiert wurde, wie ein Task-System aufgebaut und was ein Scheduler ist, soll in diesem Abschnitt analysiert werden, wie die Einhaltung von Echtzeit-Anforderungen untersucht und garantiert werden kann. Echtzeit-Anforderungen an die Software entstehen aus Anforderungen, die an das Gesamtsystem gestellt werden. Normen wie die ISO 26262 in der Automobilindustrie [55] verlangen Garantien, die auf formalen Methoden zur Verifikation basieren. Um die Anforderungen an die Ausführungszeiten eines Task-Systems formal zu untersuchen, haben sich verschiedene Verfahren, sogenannte Scheduling-Analysen, etabliert. Diese Analyseverfahren prüfen, ob eine Zusammensetzung von definierten Tasks mit den entsprechenden Umgebungsparametern (T, C, D) ausführbar ist. Ein Task-System gilt als ausführbar, wenn alle Tasks unter allen Umständen ihre Deadline einhalten. Somit muss für alle n Tasks in einem nicht-präemptiven Task-System folgendes gelten:

$$WCET_i \leq D_i \qquad i \in [0..n] \tag{3.2}$$

Betrachtet man präemptive Tasks, so gilt: $WCET \neq WCRT$. Die Abschätzung der $WCRT$ berücksichtigt nicht nur die $WCET$, sondern auch die Interaktion des Tasks mit dem gesamten Task-System. Wird ein Task unterbrochen, so ergibt sich die $WCRT$ aus der maximalen Dauer der Unterbrechung und der $WCET$.

Es gibt verschiedene Ansätze, die $WCET$ eines Tasks zu bestimmen. Führt man den Task auf der Ziel-Hardware aus und analysiert über eine entsprechend lange Zeit die gemessene Ausführungszeit des Tasks, bekommt man einen ersten Richtwert dafür. Eine belastbare Aussage über die $WCET$ erhält man jedoch erst mit formalen Ansätzen wie z. B. statischer Analyse [69]. Diese Ansätze versuchen, die längsten sinnvollen Pfade durch einen Task zu finden und errechnen daraus die $WCET$. In der Industrie werden Werkzeuge wie die von AbsInt [48] zur Abschätzung eingesetzt. Diese benutzen den Binär-Code des zu analysierenden Systems und führen mehrere Analysen auf dem Code in Kombination mit Modellen der Hardware durch, um präzise Abschätzungen für die $WCET$ zu erhalten. Dies setzt allerdings voraus, dass ein Hardware Modell für den verwendeten Mikrokontroller unterstützt wird.

Andere weitverbreitete Ansätze analysieren die Einplanbarkeit von Tasks in einem Gesamtsystem. Diese klassischen Ansätze basieren auf präemptivem Scheduling anhand von Prioritäten.

Nach [68] kann man anhand der Auslastung (U) bestimmen, ob ein Task-System ausführbar ist. Als Voraussetzung betrachtet man dazu ein präemptives Task-System, in dem alle Tasks feste Prioritäten haben, die einzelnen Task-Perioden Vielfache voneinander sind und jeder Task eine konstante Ausführungszeit besitzt. Die höchste Priorität wird dabei dem Task mit der kürzesten Periode zugeordnet. Für die Auslastung eines Systems

mit n Tasks muss dabei folgendes gelten:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{3.3}$$

Werden im Folgenden sowohl die Deadlines der einzelnen Tasks als auch nicht-harmonische Perioden mit berücksichtigt, wird das Scheduling sowie die Aussage, ob das Task-System ausführbar ist, komplexer. Hierzu wurden Scheduling-Algorithmen und dazu passende Scheduling-Analysen wie das *Rate-Monotonic-Scheduling (RMS)* von [68] oder auch *Deadline-Monotonic-Scheduling (DMS)* von [10, 67] entwickelt. Die zu den Algorithmen passenden Analysen können formal garantieren, dass das untersuchte Task-System ausführbar ist.

Im Fall des statischen, prioritätenbasierten, präemptiven Scheduling-Verfahrens *RMS* gilt nach [68] als hinreichende Bedingung, um zu prüfen, ob ein Task-System ausführbar ist:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{3.4}$$

Die Auslastung U in 3.4 konvergiert für große n gegen $ln(2)$ (69 %). Mit Hilfe des *RMS* lässt sich formal nachweisen, dass dieses System ausführbar ist, jedoch wird es dabei nicht optimal (zu 100%) ausgelastet. Dabei muss die Deadline gleich der Periode sein, und die Prioritäten werden anhand der jeweiligen Periodenlänge vergeben. Der Task mit der längsten Periodenlänge erhält die niedrigste Priorität.

DMS hingegen ist nach [10] ein bis zu 100 % auslastendes Scheduling-Verfahren, mit welchem auch formal die Ausführbarkeit nachgewiesen werden kann. Diese Weiterentwicklung des *RMS* fordert im Gegensatz dazu nicht, dass die Deadline eines Tasks gleich dessen Periode sein muss. Dabei erhält der Task mit der kleinsten Deadline die höchste Priorität.

Der *Earliest-Deadline-First (EDF)* Scheduling-Algorithmus ist ein weiteres Verfahren, welches auf dynamischen Prioritäten beruht. Nach [68] ist dieses Verfahren optimal. Hierbei wird während der Laufzeit die höchste Priorität dem Task zugewiesen, dessen Deadline am nächsten zu der aktuellen Zeit liegt. Die Ausführbarkeit eines nach *EDF* geplanten Task-Systems wird ebenfalls über dessen Auslastung (U) bestimmt.

Bei allen klassischen Verfahren sind die Prioritäten mit Eigenschaften der jeweiligen Tasks verknüpft. Dies verhindert unter Umständen die Ausführung von Tasks, die aufgrund ihrer Funktion eine höhere Priorität haben sollten, als ihnen durch das Scheduling-Verfahren vorgegeben wird. Speziell das Abbilden von Interrupts wird somit nicht adäquat möglich. Ebenso können die klassischen Verfahren Abhängigkeiten zwischen Tasks und zu Ressourcen nicht berücksichtigen und sind nicht für Multiprozessor-Systeme ausgelegt.

Um das breite Spektrum der Anforderungen abzudecken, bieten sich Zeitautomaten, wie in [5] beschrieben, an. Damit kann das zeitliche Verhalten eines Task-Systems modelliert werden und ein dazu passender Modelchecker wie z. B. Uppaal [15, 66] kann dazu genutzt werden, die Ausführbarkeit des Systems formal zu verifizieren. Das auf Uppaal

basierende TIMES [6, 7] bietet eine Möglichkeit, unterschiedliche Scheduling-Verfahren als Zeitautomaten zu modellieren und die Ausführbarkeit mittels entsprechender Anfragen zu prüfen. Dazu wird eine Erreichbarkeitsanalyse mit den erstellten Zeitautomaten durchgeführt. Da das TIMES Projekt nicht mehr weitergeführt wird und dieses in einer frühen Entwicklungsphase stehengeblieben ist, kann ein Einsatz im industriellen Umfeld nicht erfolgen und wird somit nicht weitergehend untersucht. Aufgrund der Tatsache, dass das TIMES Projekt auf Uppaal basiert und in [34, 35, 71] ähnliche Ansätze verfolgt wurden, wird im folgenden Kapitel näher auf die Modellierung eines Task-Systems mittels Uppaal eingegangen.

3.1.4 Zeitautomaten zur Scheduling-Analyse

Um eine flexible Möglichkeit zu erhalten, Task-Systeme auf Ausführbarkeit in verschiedenen Scheduler-Systemen zu prüfen, wird nun Uppaal näher untersucht. Uppaal bietet eine Sammlung von Werkzeugen, um Echtzeitsysteme mittels Zeitautomaten zu modellieren und zu verifizieren [13]. Dazu werden einzelne Zeitautomaten [5] zu Netzwerken zusammengefügt und ein Model-Checker verifiziert, ob das System definierte Anforderungen erfüllt. Weiterhin steht dem Benutzer ein Simulator zur Verfügung, in dem das erstellte Modell durchgespielt werden kann. Speziell Uppaal-Tiga [14] bietet zusätzlich dazu noch die Möglichkeit, die Ergebnisse der Simulation als Gantt-Diagramm [27] zu visualisieren. Diese aus der Scheduling-Analyse bekannte Möglichkeit lässt den Benutzer auch das Zeitverhalten von Tasks in komplexen Scheduling-Verfahren überblicken. Mit Uppaal stehen somit alle notwendigen Werkzeuge bereit, um ein Echtzeitsystem auf Task- und Scheduler-Ebene darzustellen und auch um deren Ausführbarkeit zu prüfen. Dies zeigte auch schon der Ansatz, der in Uppaal TIMES [7] verfolgt wurde.

In der Zusammenfassung von [66] wird Uppaal als parallele Komposition von verschiedenen Zeitautomaten beschrieben. Diese können zum einen getrennt voneinander betrachtet werden, zum anderen gibt es zusätzlich aber auch Mechanismen zur Synchronisation. Dies können spezielle Kanäle zwischen zwei Zeitautomaten sein oder auch sogenannte Broadcast-Kanäle, welchen Zeitautomaten zur Verfügung stehen. Damit können z. B. auch zwei miteinander kommunizierende Prozessoren modelliert werden. Dabei besteht jeder Zeitautomat aus *Zuständen* und *Transitionen*, die diese untereinander verbinden (vgl. Abbildung 3.2).

Alle verwendeten Uhren in einem Zeitautomaten laufen gleich schnell und können zurückgesetzt werden. Jeder *Zustand* kann als initialer Zustand (S1 in Abbildung 3.3) gekennzeichnet werden, wobei nur ein initialer Zustand je Zeitautomat existieren darf. Weiterhin kann jeder *Zustand* mehrere sogenannte *Invarianten* besitzen. *Invarianten* sind Aussagen, die wahr oder falsch sein können. Ist die *Invariante* eines *Zustandes* falsch, so muss dieser über eine seiner *Transitionen* verlassen werden. Wird das Verlassen des *Zustandes* jedoch durch die Einschränkungen (*Guards*) an den jeweiligen Transitionen unterbunden, so befindet sich der Zeitautomat in einem Fehlerzustand, dem *Deadlock*. Zustandsübergänge können nicht nur an *Guards* auf Basis von Variablen geknüpft werden,

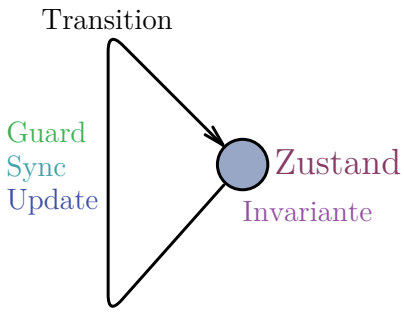


Abbildung 3.2: Syntax eines Uppaal Modells

sondern auch an Uhren. So kann z. B. in Abbildung 3.3 ein Wechsel von Zustand S1 nach S2 erst dann erfolgen, wenn Uhr > 5 ist. Ein Wechsel von S2 nach S1 kann erst stattfinden, wenn Uhr > 10. Die Synchronisation von Zeitautomaten wird über Kanäle gesteuert. Diese können über Transitionen (*Sync*) gesendet oder empfangen werden. Weiterhin können während einer Transition Variablen oder auch Uhren gesetzt werden (*Update*).

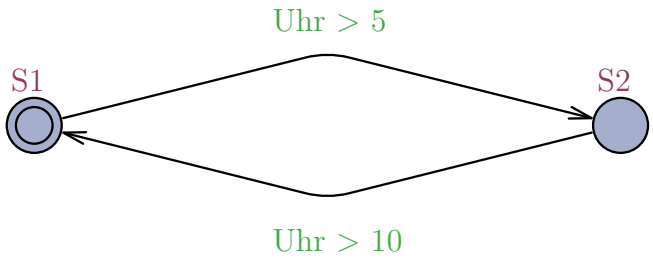


Abbildung 3.3: Zustandsübergänge mit Guards

Zustände können nicht nur als initial gekennzeichnet werden, sondern auch als dringend (*urgent*) und als verpflichtend (*committed*). In [66] heißt es dazu, dass sowohl in dringenden als auch in verpflichtenden Zuständen keine Zeit vergehen darf. Verpflichtende Zustände müssen darüber hinaus nach dem Durchlaufen direkt über eine Transition verlassen werden und können dabei nicht unterbrochen werden.

Der in Uppaal integrierte Model-Checker prüft, ob das erstellte Modell definierten Anforderungen genügt. Dazu werden diese Anforderungen in einer vereinfachten Form von *TCTL* [13] formalisiert. Diese Beschreibungssprache besteht aus Pfad- und Zustandsformeln. Zustandsformeln beschreiben dabei individuelle Zustände im Zustandsraum des Modells, wohingegen Pfadformeln ganze Pfade abdecken [13]. Pfadformeln beschreiben dabei Eigenschaften zur Erreichbarkeit, Sicherheit und Lebendigkeit der Zeitautomaten [13]. Im Gegensatz zu *TCTL* unterstützt Uppaal keine in einander verschachtelten Pfadformeln.

3.1.5 Realisierung der Scheduling-Analyse mittels Zeitautomaten

Aufbauend auf den Ideen vorgestellt in [34, 35, 71] wird im Folgenden ein Scheduler sowie ein generischer Task mit Zeitautomaten modelliert. Dabei werden diese Ansätze speziell auf die System-Architektur des Fallbeispiels sowie auf die Anforderungen aus 3.1.2 angepasst. Zur Vereinfachung wird unter anderem die Analyse nur auf die *WCET* gestützt und das generische Task-Modell so ausgelegt, dass es die Eigenschaften von Interrupts abbilden kann. Teile der nachfolgenden Kapitel wurden bereits in [GKKK11] und [62] veröffentlicht.

Scheduler Modell

Das Modell des Schedulers ist generisch aufgebaut. Es enthält keine Informationen über das zu verwaltende Task-System und benötigt somit auch keine Parameter, um es an unterschiedliche Task-Konstellationen anzupassen. Die Aufgabe des Schedulers besteht darin, die Rechenzeit des Mikrokontrollers nach einem definierten Algorithmus auf die im System vorhandenen Tasks aufzuteilen. Dabei müssen bestimmte Eigenschaften der einzelnen Tasks berücksichtigt werden wie z. B. deren Prioritäten, aber auch ob diese unterbrechbar sind oder nicht. Die meisten Mikrokontroller haben einen Rechenkern, und somit kann die Rechenzeit immer nur an einen Task vergeben werden. Die anderen im System vorhandenen Tasks warten während dessen, bis der Scheduler ihnen Rechenzeit zuweist.

Abbildung 3.4 zeigt das Uppaal Modell des Schedulers. Es beinhaltet mehrere Zustände, von denen drei besonders hervor zu heben sind: *Idle*, *Ready* und *Busy*. Diese drei Zustände bilden die Hauptzustände des Schedulers. Beim Start des Systems beginnt der Scheduler im *Idle* Zustand und wartet, bis das Signal *activate* eines Task ausgelöst wird. Damit signalisiert ein Task, dass dieser bereit ist und nun Rechenzeit zugewiesen bekommen möchte. Bekommt der Scheduler dieses Signal, fügt er den sendenden Task der internen Liste (*isReady*) hinzu. Diese Liste beinhaltet alle rechenbereiten Tasks.

Der aktuelle Zustand wird verlassen und das Signal *doneexecute* an den aktuell rechnenden Task gesendet. Dieser prüft daraufhin, ob er mit seinen Berechnungen fertig ist und terminiert sich daraufhin selbst. Oder aber er rechnet weiter, bis ein neuer höher priorisierter Task aktiv wird. Das Signal *doneexecute* wird benutzt, um das Beenden eines Tasks höher zu priorisieren als das Aktivieren eines neuen Tasks. Durch *checkNotFinish()* wird dann geprüft, ob alle Tasks korrekt beendet wurden. Darauf folgend wählt der Scheduler anhand der Task-Prioritäten aus der Liste der wartenden Tasks den Task aus, der als nächstes Rechenzeit zugewiesen bekommt. Die Modell-Variable *next* enthält die ID des nächsten Tasks. Ist die interne Taskliste leer, wird *next* auf N gesetzt. Dies zeigt nun an, dass kein Task mehr im System wartet und der Scheduler wieder in den Zustand *Idle* wechseln kann, um dort auf das Fertigwerden eines Tasks zu warten.

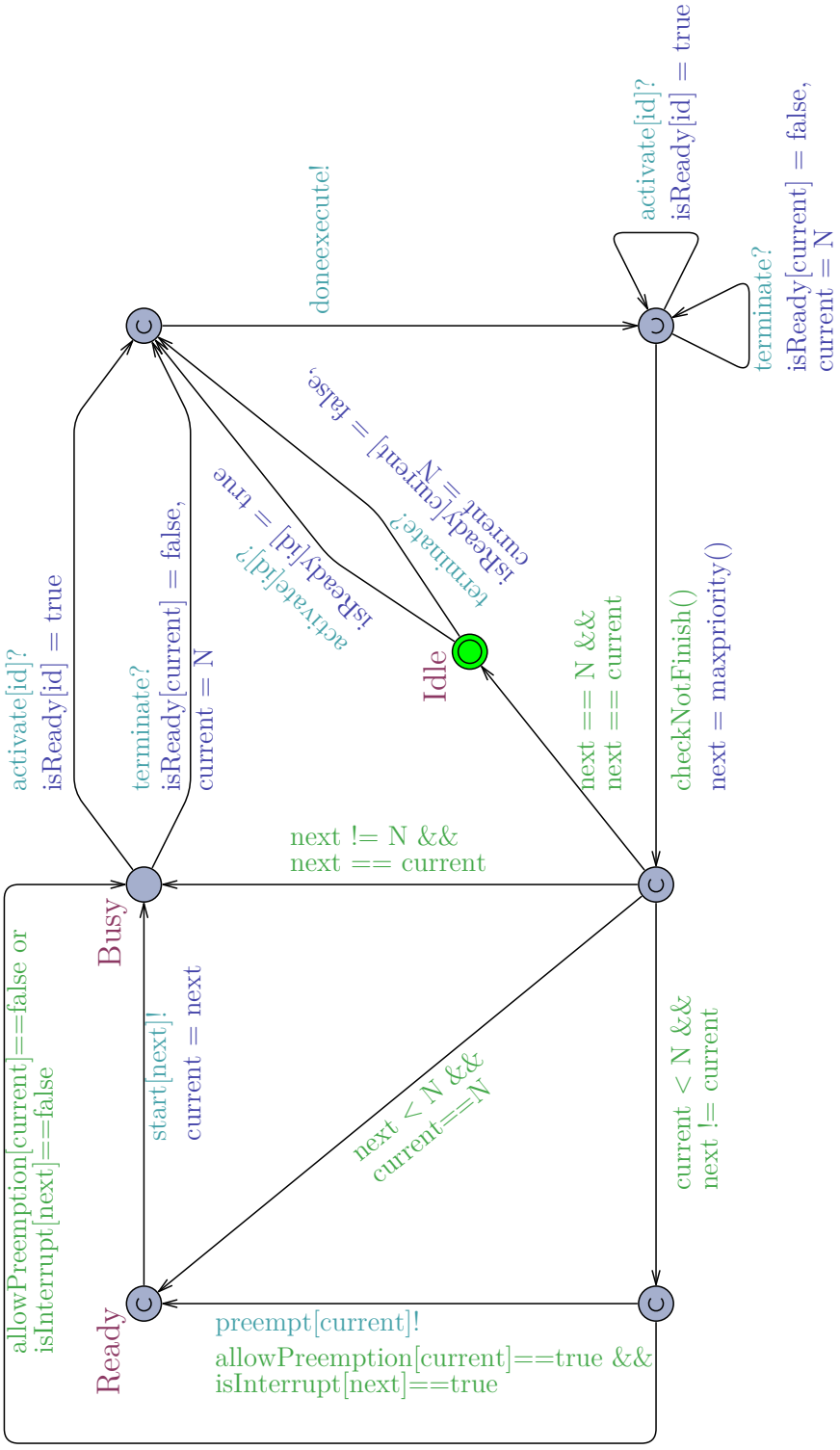


Abbildung 3.4: Zeitautomaten des Scheduler (angelehnt an [62])

Ist die Taskliste nicht leer und warten somit noch andere Tasks auf Rechenzeit, so können die folgenden drei Fälle eintreten:

- **Ein neuer Task beginnt zu rechnen** ($next < N \;\&\&\; current == N$): Es befindet sich nur ein Task in der Taskliste, und dieser hat sich gerade erst selber aktiviert. Dann wechselt der Scheduler in den Zustand *Ready* und startet über das Signal *start[next]* diesen Task. Der Task beginnt zu rechnen, und der Scheduler wechselt in den Zustand *Busy*. Erst wenn ein zusätzlicher Task aktiv wird oder aber der aktuelle Task sich selbst terminiert hat, verlässt der Scheduler diesen Zustand und beginnt erneut mit der Zuweisung der Rechenzeit an den entsprechenden Task.
- **Der aktuelle Task rechnet weiterhin** ($next\ != N \;\&\&\; next == current$): Der Scheduler hat einem Task Rechenzeit zugewiesen. Während dieser Task rechnet, aktiviert sich ein weiterer Task. Dieser wird vom Scheduler in die interne Taskliste aufgenommen. Allerdings hat dieser neue Task keine höhere Priorität als der aktuell rechnende Task. Somit bleibt der aktuell rechnende Task weiterhin aktiv, und der neue Task bleibt wartend. Der Scheduler wechselt also direkt in den Zustand *Busy* und wartet auf das Terminieren des aktuell rechnenden Tasks oder auf das Aktivieren eines neuen Tasks.
- **Der aktuelle Task wird unterbrochen** ($current < N \;\&\&\; next\ != current$): Im Gegensatz zum vorhergehenden Fall hat der sich selbst aktivierende Task eine höhere Priorität als der aktuell rechnende Task. Dies führt dazu, dass der Scheduler auf der einen Seite prüfen muss, ob der aktuell rechnende Task eine Unterbrechung zulässt (*allowPreemption*=true) und auf der anderen Seite prüfen muss, ob der neue Task einen anderen Task überhaupt unterbrechen kann (*isInterrupt*=true). Stellt der Scheduler fest, dass der aktuelle Task generell nicht unterbrochen werden darf oder der neue Task den aktuellen Task nicht unterbrechen kann, so wechselt der Scheduler in den Zustand *Busy*. Dort wartet der Scheduler auf das Terminieren des aktuell rechnenden Tasks oder auf das Aktivieren eines neuen Tasks. Stellt der Scheduler allerdings fest, dass der aktuell rechnende Task unterbrechbar ist und der neue Task diesen unterbrechen kann, so wird das Signal *preempt[current]* an den aktuell rechnenden Task gesendet und der neue Task mit dem Signal *start[next]* gestartet.

Task Modell

Der Zeitautomat des Tasks (siehe Abbildung 3.5) besteht aus fünf Hauptzuständen: *Suspended*, *Ready*, *Running*, *EndPeriod* und *Error*. Zu jeder Instanz gehören Parameter, die Eigenschaften des Taskmodells darstellen. Dabei werden folgende Parameter je Task verwendet:

- **ID**: Die Identitätsnummer eines jeden Tasks ist eindeutig und gilt gleichzeitig als Priorität. Dabei ist 0 die höchste Priorität. Die Prioritäten werden im eingebetteten

System verwendet, um zu entscheiden, in welcher Reihenfolge auftretende Interrupts verarbeitet werden.

- **Periode „T“**: Die Periode des Tasks gibt an, in welchen Zeitabständen der Task automatisch erneut rechenbereit wird.
- **Ausführungszeit „C“**: Die Ausführungszeit gibt die Zeit an, die ein Task zum vollständigen Durchlaufen benötigt. Dabei werden keine Unterbrechungen durch andere Tasks mit einbezogen. Der Wert für diesen Parameter sollte der sogenannten *WCET* gleichen. Diese gibt die maximal mögliche Ausführungszeit eines Tasks ohne jegliche Abhängigkeit zu anderen Tasks an.
- **Deadline „DL“**: Um zu prüfen, ob ein Task seine Anforderungen an die Echtzeit einhält, wird die *Deadline* als obere Zeitschranke verwendet. Verletzt ein Task seine eigene Zeitschranke, so liegt ein Fehler vor.
- **Unterbrechbarkeit „preemptable“**: Wenn ein Task von anderen, höher priorisierten Tasks unterbrochen werden können soll, muss dieser Parameter auf *true* gesetzt werden. In einem eingebetteten System würde ein nicht-unterbrechbarer Task z. B. während seiner Laufzeit global die Möglichkeit von Interrupts deaktivieren. Um dieses Verhalten entsprechend zu modellieren, wird dieser Parameter auf *false* gesetzt.
- **Interrupt „interrupt“**: Über diesen boolschen Parameter kann ein Task als Interrupt gekennzeichnet werden. Wenn ein solcher Interrupt rechenbereit wird, kann dieser den aktuell rechnenden Task unterbrechen, wenn die Priorität des Interrupts höher ist als die des aktuell rechnenden Tasks und der Task unterbrechbar ist (*preemptable*). Nachdem der Interrupt abgearbeitet worden ist, wird die Ausführung des letzten Tasks fortgeführt.

Jede Instanz eines Tasks startet im Zustand *Suspended*. Daraufhin meldet sich jeder Task über den Broadcast Kanal *activate[id]* beim Scheduler an. Dieser erweitert seine interne Taskliste um den entsprechenden Task und berücksichtigt diesen bei der nächsten Verteilung von Rechenzeit. Durch das Nehmen der einzigen Transition werden alle im Task Modell verwendeten Uhren (*ready*, *running*, *wcrt*) zurückgesetzt und damit automatisch gestartet. Eine Uhr wird solange um Zeiteinheiten inkrementiert, bis diese gestoppt oder zurückgesetzt wird. *ready* ist notwendig, um die Deadline des Tasks einzuhalten. Überschreitet der Uhrzähler den Parameterwert *DL*, so wechselt das Taskmodell in den Zustand *Error* und zeigt somit ein Fehlverhalten des Systems an. Die Uhr *running* wird verwendet, um die Zeiteinheiten zu messen, die der Task aktiv rechnend ist. Diese Uhr wird angehalten ($running' == 0$), falls der aktuelle Task von einem anderen, höher priorisierten Task unterbrochen wird. Die Uhr *wcrt* wird benutzt, um die *WCRT* des entsprechenden Tasks zu errechnen. Diese ergibt sich als Maximum über alle möglichen *wcrt* Zählerstände des Tasks.

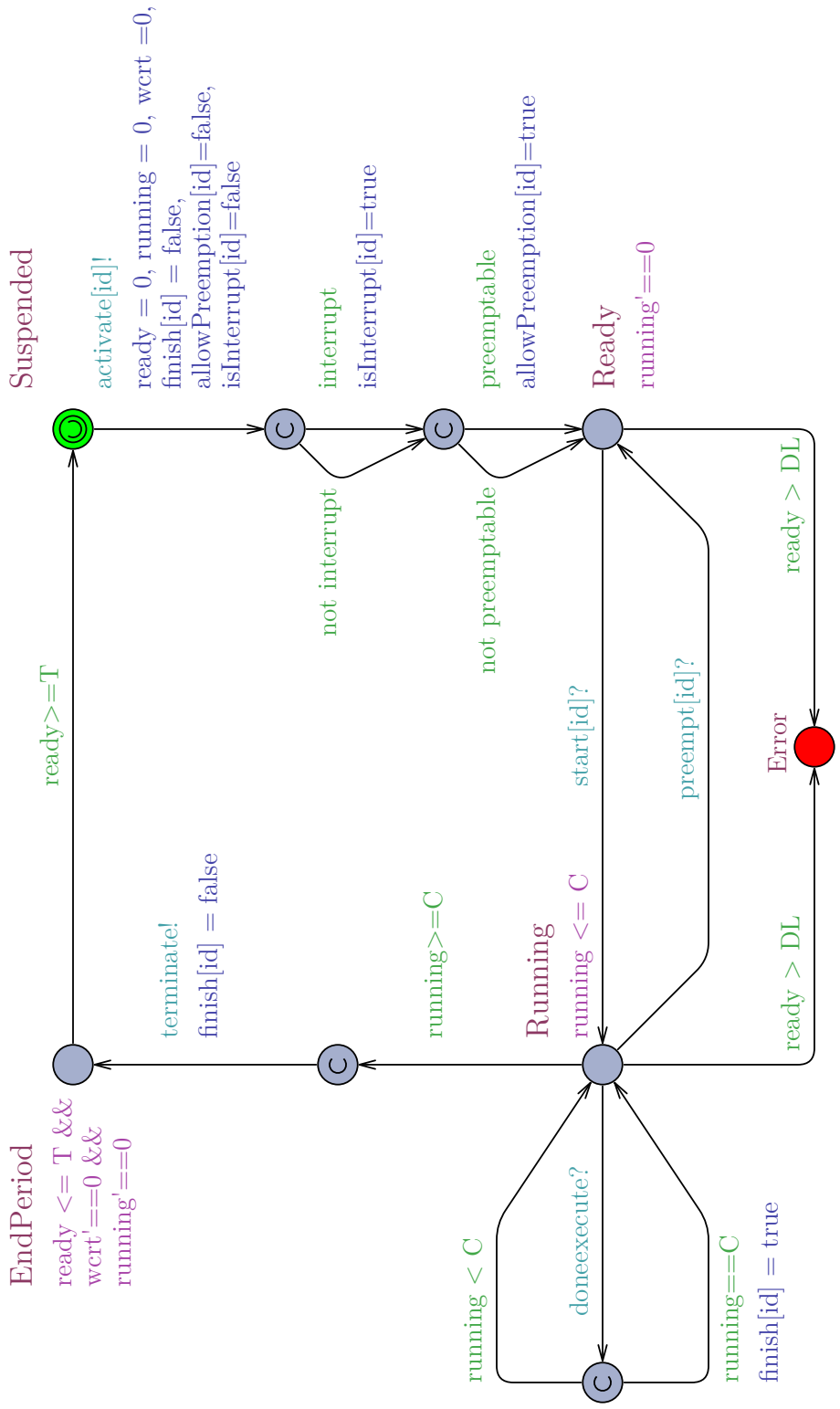


Abbildung 3.5: Generischer Zeitautomat eines System-Tasks (angelehnt an [62])

Die nächsten beiden Zustände, welche als *Committed* gekennzeichnet sind, um keine Zeiteinheiten vergehen zu lassen, initialisieren je nach Parameterwert die dem Task entsprechenden Einträge in den globalen Listen *isInterrupt* und *allowPreemption*. Diese Informationen werden vom Scheduler bei der Vergabe von Rechenzeit berücksichtigt. Mit *isInterrupt* kann das Verhalten eines realen Interrupts und somit ein Unterbrechen anderer Tasks abgebildet werden. Die Möglichkeit zur Deaktivierung von Interrupts während der Ausführung eines Tasks erfolgt über den Parameter *allowPreemption*. Im Zustand *Ready* verweilt jeder Task so lange, bis dieser vom Scheduler Rechenzeit zugewiesen bekommt oder zu lange in diesem Zustand gewartet hat und mittels der Uhr *ready* eine Überschreitung der Deadline festgestellt werden kann. Daraufhin wechselt das Taskmodell in den Fehlerzustand *error* und erzeugt einen Deadlock.

Teilt der Scheduler einem Task Rechenzeit zu, so wechselt dessen Task-Modell-Instanz in den Zustand *running*. Dieser Wechsel wird über das Signal *start[id]* ausgelöst. Wird in diesem Zustand eine Verletzung der Deadline festgestellt, so wechselt das Taskmodell ebenfalls in den Fehlerzustand *Error*.

Erreicht der Zählerstand der Uhr *running* den Wert *C*, so gilt der Task als beendet, und somit meldet sich das Taskmodell über das Signal *terminate* vom Scheduler ab. Dieser nimmt den entsprechenden Task aus der internen Taskliste und setzt die Vergabe der Rechenzeit ohne diesen Task fort. Das Taskmodell wechselt in den Zustand *EndPeriod* und wartet mit der erneuten Aktivierung bis zu seiner nächsten Periode.

Wird, während sich der aktuell rechnende Task im Zustand *Running* befindet, ein weiterer Task rechenbereit, so sendet der Scheduler das Signal *doneexecute*, um ein Terminieren des aktuellen Task prüfen zu lassen, um unter Umständen dem neuen Task Rechenzeit zuzuordnen. Stellt der Task bei dieser Prüfung fest, dass er beendet werden kann, meldet dieser sich beim Scheduler ab und wechselt in den Zustand *EndPeriod*. Um das Beenden eines Tasks endgültig abzuschließen, wird bei erfolgreicher Prüfung die globale Variable *finish[id]* gesetzt. Diese Variable wird entsprechend vom Scheduler ausgewertet, und es wird so lange mit der Aktivierung eines neuen Tasks gewartet, bis das Beenden abgeschlossen ist. Besteht jedoch weiterhin die Anforderung an Rechenzeit aus Sicht des aktuellen Tasks, so muss der Scheduler diesen über das Signal *preempt* unterbrechen und in den Zustand *Ready* vesetzen. Der neue Task erhält daraufhin Rechenzeit, und die entsprechende Task-Instanz wechselt in den Zustand *Running*.

3.1.6 Evaluierung

Die Evaluierung wurde in Teilen schon in [GKKK11, GRK13] und [62] veröffentlicht und basiert auf der Anwendung der vorab eingeführten Methodik auf dem in Abschnitt 2.7 vorgestellten Fallbeispiel. Nachdem in den vorangegangenen Kapiteln Uppaal -Modelle für den Scheduler und den dazu passenden generischen Task aufgebaut worden sind, folgt nun die Formalisierung sowie die Evaluierung der Anforderungen. Für die Einplanbarkeitsanalyse werden die Anforderungen umgangssprachlich wie folgt definiert (vlg. [GRK13]):

1. Es soll festgestellt werden, ob die vorgegebenen Tasks mit ihren Eigenschaften (Ausführungszeit, Periode, unterbrechbar, Interrupt) im Zusammenspiel einplanbar sind.

2. Für eine bessere Fehlerdiagnose sollten die maximalen Antwortzeiten der verschiedenen Tasks aus den einzelnen Ausführungszeiten und unter Berücksichtigung des Ausführungsplans ermittelt werden.

Als nächstes müssen die so formulierten Anforderungen in eine formale Form überführt werden. Mit der ersten Anforderung wird von jedem Task die Einhaltung seiner Deadline gefordert. Daher gilt hierbei Gleichung 3.2, welche beschreibt, dass die *WCRT* aller Tasks kleiner als deren Deadline (D) sein muss. Dafür wurde in das Task-Modell (siehe Abbildung 3.5) ein Error-Zustand integriert, der bei Verletzung der Echtzeitbedingung ($ready > DL$) aktiv wird. Formel 3.5 prüft, ob für einen beliebigen Task dieser Zustand nie erreicht wird. Ist für alle im System vorhandenen Tasks diese Formel erfüllt, so gilt diese Task-Zusammenstellung als einplanbar.

$$A[] \text{ not TaskX.Error} \tag{3.5}$$

Unabhängig vom erstellten Uppaal Modell gilt es zu prüfen, ob der Automat Zustände enthält, in denen Blockierungen (engl. Deadlocks) möglich sind. Ein Zustand gilt in Uppaal als blockierend genau dann, wenn dieser keinen aktuell verfügbaren Nachfolgezustand hat. Aufgrund von z. B. Guards oder Invarianten kann es vorkommen, dass es nicht erlaubt ist, einen Zustand über eine seiner Transitionen zu verlassen, obwohl er es aufgrund einer Invariante müsste. Dies kann auf ein fehlerhaftes Modell hindeuten oder aber ein tatsächliches Problem im Systemverhalten aufdecken. Dies kann explizit mit der folgenden *TCTL* Formel überprüft werden:

$$A[] \text{ not deadlock} \tag{3.6}$$

Um mit dem Uppaal Modell die maximal mögliche Antwortzeit (*WCRT*) ermitteln zu können, wurden Stoppuhren in den erstellten Modellen verwendet. Nach [26] kann der Einsatz von Stoppuhren in Uppaal zu einer Überapproximation des Zustandsraums führen. Mittels Formel 3.7 kann die Anforderung nach der maximalen Antwortzeit (*WCRT*) erfüllt werden. Hierbei werden die entsprechenden Uhren der einzelnen Tasks (Antwortzeit: AT) im Zustandsraum abgefragt. Mit dem Zusatz *sup* zu Beginn der Formel können deren Maxima ermittelt werden.

$$\text{sup: Interrupt1.AT, Task1.AT, Task2.AT, Task3.AT} \tag{3.7}$$

Im Folgenden wird nun die Tauglichkeit der Methodik unter realen Bedingungen evaluiert. Dazu werden exemplarisch Task-Konstellationen untersucht, die aus den SW-Optionen aus dem in Abschnitt 2.7 beschriebenen Fallbeispiel ausgewählt worden sind. Dabei ist die erste Kombination nicht einplanbar und die zweite einplanbar. Verglichen

werden die Ergebnisse der formalen Analyse mittels Uppaal und die realen Messwerten, die mit der Original-Hardware aufgenommen worden sind. Anhand der ersten Kombination soll das Verhalten der Uppaal Modelle bezüglich des Einhaltens der jeweiligen Deadlines geprüft werden. Die zweite Kombination wird zur Evaluierung der in Uppaal ermittelten *WCRT* im Vergleich zur gemessenen maximalen Antwortzeit genutzt.

Im ersten Fall wird die Einplanbarkeit von zwei Tasks untersucht. Dabei ist der erste Task TRA periodisch mit einer Rate von 1000 µs und hat eine Deadline von 1000 µs sowie eine Ausführungszeit von 929 µs. TRA ist unterbrechbar, allerdings kann er keine anderen Tasks unterbrechen. Weiterhin ist in Interrupt INT0 im System vorhanden, der andere Tasks unterbrechen kann. INT0 ist selber unterbrechbar und höher priorisiert als TRA. Seine Ausführungszeit beträgt 258 µs. Da dessen Periode aufgrund des Fallbeispiels an die Motordrehzahl gekoppelt ist, wird diese anhand der maximalen Drehzahl von 12 000 U/min zu 5000 µs bestimmt.

Nun werden die zuvor definierten Formeln auf das Modell angewendet. Das Ergebnis des Model-Checking Prozesses in Uppaal für Formel 3.6 ist ein „Property may be satisfied“. Dies deutet darauf hin, dass evtl. ein Zustand blockiert. Genauere Untersuchungen mit der Formel 3.5 für TRA und INTO zeigen, dass der Fehlerzustand des Tasks TRA erreicht wird. Betrachtet man nun die mit Formel 3.7 ermittelten *WCRT*, so erkennt man, dass die Deadline für TRA nicht eingehalten werden kann. Die formale Analyse zeigt, dass diese beiden Tasks somit unter den definierten Randbedingungen nicht zusammen eingeplant werden können.

Um diese Aussage durch Messungen am realen System verifizieren zu können, wurde ein C-Programm basierend auf der ausgewählten Task-Kombination erstellt und auf den Mikrokontroller geflasht. Um die Ausführungszeiten der beiden Tasks messen zu können, wurde je ein digitaler Ausgang zu Beginn der jeweiligen Funktion auf *high-Pegel* und am Ende auf *low-Pegel* gesetzt. Der so erzeugte Puls wurde mittels Oszilloskop gemessen, und die Ergebnisse sind in Abbildung 3.6 dargestellt. Dabei wurde die Reaktion des Systems auf Drehzahlen von 12 000 U/min und 6000 U/min gemessen. Wie zu erwarten war, verletzt Task TRA je nach Drehzahl seine Deadline, da dieser durch den Interrupt INT0 alle 5 ms respektive 10 ms unterbrochen wird.

Vergleicht man die formale Analyse mit den gemessenen Werten, so zeigt sich eine Übereinstimmung in der Aussage, dass das Task-System in dieser Konstellation nicht einplanbar ist.

Im zweiten Fall wird eine Task-Kombination aus den Tasks TRB und TRC betrachtet. Task TRB hat eine Periode von 100 ms, wohingegen Task TRC alle 500 ms aufgerufen wird. Abbildung 3.7 zeigt die auf der realen Hardware gemessenen Antwortzeiten von TRB über eine Dauer von 5 s. Daraus wurde dessen maximale Antwortzeit von 2250 µs bestimmt. Die von Uppaal errechnete Antwortzeit von 2401 µs wird dabei immer unterschritten. Der Anstieg der mittleren Antwortzeit zum Ende der Messung entsteht dadurch, dass andere Berechnungspfade innerhalb der beiden Tasks gewählt wurden. Eine Messung kann immer nur ein gewisses Zeitfenster abdecken und somit nie alle möglichen Zustände berücksichtigen. Dies erklärt auch die Differenz von gemessener und errechneter maximaler

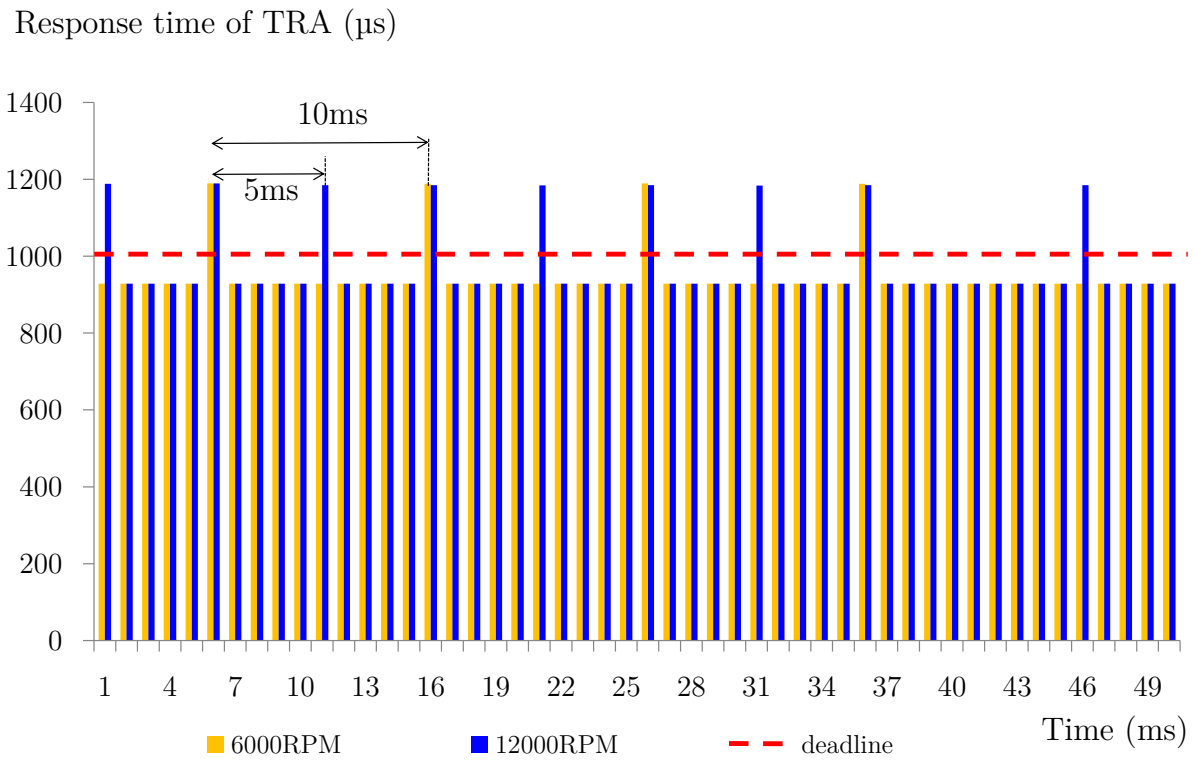


Abbildung 3.6: Antwortzeiten für zwei unterschiedliche Drehzahlen (angelehnt an [GKKK11] und [62])

Antwortzeit. Dabei ist ebenfalls noch zu berücksichtigen, dass die zur Analyse mittels Uppaal notwendigen Antwortzeiten der Tasks auf gemessenen Werten basieren und dadurch ebenfalls nicht alle möglichen Zustände abdecken.

Abschließend kann festgehalten werden, dass die erstellten Modelle des Scheduling-Systems das reale Verhalten adäquat abbilden und zum Einsatz der Einplanbarkeitsanalyse basierend auf den Anforderungen aus 3.1.2 verwendet werden können. Somit können realistische Ressourcenabschätzungen frühzeitig in den Prozess der Angebotserstellung mit einfließen, um damit mögliche Zielkonflikte zwischen preisgünstiger Hardware und den gewünschten Softwarefunktionen aufzuzeigen. *KMU* haben dabei die Vorteile, dass deren ausgeprägte Nähe zum Kunden dadurch zusätzlich gestärkt wird, und dass die Systemkomplexität deren Projekte oft überschaubarer ist als es bei großen Unternehmen meistens der Fall ist. Dieser Aspekt hilft besonders bei der Nachvollziehbarkeit der zu erstellenden Task- und Scheduler-Modelle.

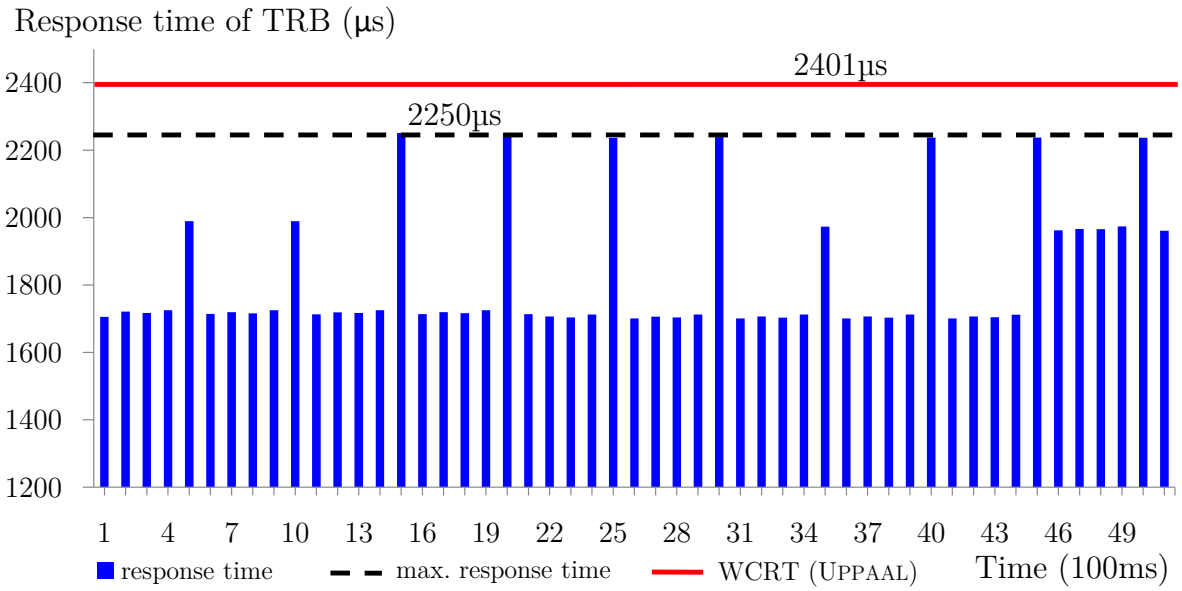


Abbildung 3.7: *WCRT* für einen Task ermittelt mit UPPAAL und gemessen (angelehnt an [GKKK11] und [62])

3.2 Verwaltung von Varianten

Durch immer schneller werdende Produktzyklen und dem dadurch entstehenden Kostendruck steht das Wiederverwenden von Software und Teilen von Software (Modulen) immer mehr im Vordergrund. Ebenso führt die von Kunden geforderte frühzeitige Ressourcenabschätzung zur Notwendigkeit, Informationen von vorhandenen Software-Modulen gezielt zu kombinieren und wiederzuverwenden. Dabei können gemessene Daten wie z. B. Ausführungszeiten oder Speicherbedarf verwendet werden oder auch Daten, die von erfahrenen Entwicklern abgeschätzt werden. Im Zusammenhang mit der in Abschnitt 2.7 beschriebenen System-Plattform ist ein konsistenter Ansatz notwendig, der die variablen Software-Zusammenstellungen unterstützt.

Bei dem betrachteten Fallbeispiel aus Abschnitt 2.7 ist das ursprüngliche Produkt gezielt für einen Anwendungsfall entwickelt worden. Daraus sind nachfolgend weitere Lösungen für unterschiedliche Anwendungsfälle abgeleitet worden, und somit ist auch der Aufwand für die Pflege der teilweise identischen Software-Module gestiegen. In diesen Modulen enthaltene Fehler müssen stets in allen abgeleiteten Produkten mit hohem Aufwand separat behoben und getestet werden.

Ziel ist eine Reduzierung des Aufwandes bei der Pflege der verschiedenen Varianten sowie Steigerung der Qualität bzw. Robustheit der Software durch Wiederverwendung von Teilen des Quellcodes. Dadurch soll eine verkürzte Time-To-Market erreicht werden sowie ein gemeinsames Testen von Software-Modulen. Daraus ergeben sich folgende Anforderungen an ein entsprechendes Framework:

1. Einzelne Software-Module sollen zu verschiedenen neuen Produkten kombiniert werden. (Baukastenprinzip)
2. Abhängigkeiten zwischen den Software-Modulen sollen definierbar sein.
3. Software-Module sollen projektübergreifend wiederverwendet werden können.
4. Der Verbrauch von Ressourcen eines definierten Produktes soll mittels der in Abschnitt 3.1 entwickelten Methode abschätzbar sein.
5. Es soll kein unnötiger Code in den einzelnen Produkten erzeugen werden (z. B. keine `if-else` Varianten bzw. Konfiguration über .ini Dateien).
6. Die Funktionalität des zu erstellenden Frameworks soll flexibel erweiterbar sein und dazu entsprechende Schnittstellen bereitstellen. Mögliche Werkzeuge zur Verifikation von Binär-Code (vgl. Kapitel 4) sollen integriert werden können.

Diese Anforderungen beschreiben die Verwaltung von Software-*Varianten*. Diese Verwaltung wird mit dem Ansatz der sogenannten *Software Produktlinien* durchgeführt. Das Software Engineering Institute der Carnegie Mellon University definiert *Software Produktlinien* wie folgt [77]:

> „A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“

Diese Definition beschreibt die geplante bzw. organisierte Wiederverwendung von Software-Systemen. Diese bestehen aus unterschiedlichen Merkmalen (eng. *Features*) und gemeinsamen Kernkomponenten. Die Zusammenstellung von verschiedenen *Features* wird *Variante* (engl. *Variant*) genannt.

3.2.1 Framework für die Variantenverwaltung

Um *Features* und *Varianten* eines Software-Systems modellieren zu können, gibt es akademische Ansätze wie z. B. das FeaturePlugin [8] sowie FeatureIDE [114]. [78] vergleicht systematisch verfügbare Ansätze und listet Eigenschaften der einzelnen Werkzeuge auf. Da auch kommerzielle Werkzeuge auf dem Markt vorhanden sind, werden diese aufgrund des typischerweise stabileren Programmverhaltens fokussiert. Die beiden Werkzeuge Gears [64] und pure::variants [16, 17, 87] haben sich im industriellen Einsatz bewährt und decken beide die Anforderungen an eine Varianten-Verwaltung auf Basis von C-Code, Textdateien und XML-Dokumenten ab. Da die Wahl des Werkzeugs auch zukünftige Anforderungen abdecken soll und die zu betrachtenden Anwendungsfälle aus der Automobilindustrie stammen, hat hierbei pure::variants den ausschlaggebenden Vorteil. Wie

auch Gears unterstützt es das im automobilen Umfeld häufig eingesetzte IBM Rational DOORS® zur Anforderungserfassung. Darüber hinaus bietet pure::variants auch die Verwaltung von AUTOSAR Modellen sowie MathWorks Simulink® Modellen.

pure::variants ist ein Werkzeug, welches die Variabilität eines Systems mit Meta-Modellen für das Problem (*Feature Model*), die Lösung (*Family Model*) sowie deren Verknüpfung (*Variant Description Model*) ausdrückt [18]. Die daraus entstehende konkrete Variante bildet das Ergebnismodell (*Result Model*). Abbildung 3.8 zeigt die Abhängigkeiten der verschiedenen Modelltypen.

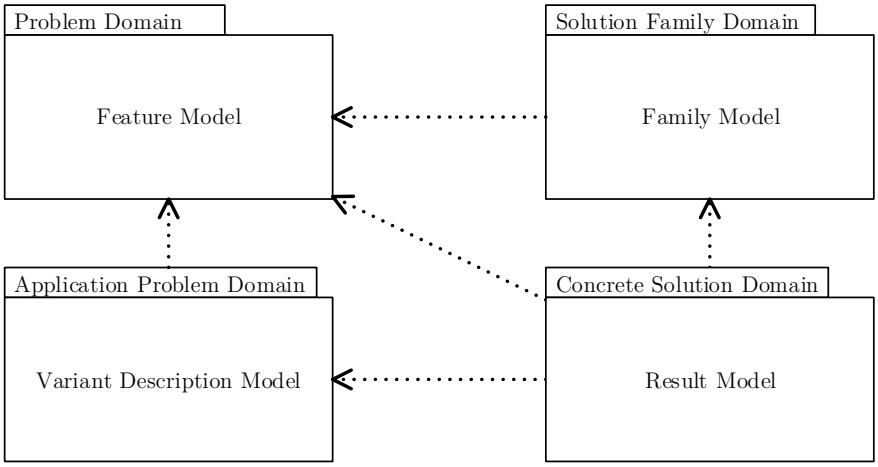


Abbildung 3.8: pure::variants Modelltypen und deren Abhängigkeiten nach [18]

- Das **Feature Model** beschreibt die einzelnen Eigenschaften des Systems. Das Modell kann als hierarchisches Baumdiagramm dargestellt werden, wie das Beispiel der Wetterstation (Abbildung 3.9) zeigt. Für jedes *Feature* können Daten festgelegt werden, auf denen Transformationen aus dem *Family Model* ausgeführt werden. Weiterhin beschreibt das Modell die Abhängigkeiten zwischen den verschiedenen *Features*. Hierzu gibt es folgende Möglichkeiten:
  - **Mandatory** Ein solches *Feature* ist zwingend erforderlich für die Lösung. Somit ist es auch in allen möglichen *Varianten* vorhanden.
  - **Optional** Dieses *Feature* kann, muss aber nicht Teil der Lösung sein. Im Gegensatz zu einem *Mandatory Feature* muss dieses explizit zur Lösungsvariante hinzugefügt werden.
  - **Alternative** Der Benutzer muss sich bei diesen *Features* für genau eines entscheiden. Dieses wird dann zur resultierenden *Variante* hinzugefügt.
  - **Or-Feature** Im Gegensatz zu einem *Alternative Feature* kann unter mehreren so markierten *Features* eines oder auch mehrere ausgewählt werden.

- Das **Family Model** beinhaltet die konkreten Implementierungen des Software-Systems. Dabei werden gemeinsame Kernkomponenten beschrieben sowie mit dem *Feature Model* verknüpfte Komponenten. Diese beschreiben z. B. XML Transformationen im Zusammenhang mit Stellen im Quellcode, an denen einzelne *Features* eigenen Code einfügen können.
- Das **Variant Description Model** beschreibt die Auswahl der konkreten Variante abgeleitet aus dem *Family Model* als Lösung für das gegebene Problem.
- Das **Result Model** stellt eine einzelne konkrete Variante als Ergebnis aus dem *Feature Model*, dem *Family Model* und dem *Variant Description Model* dar.

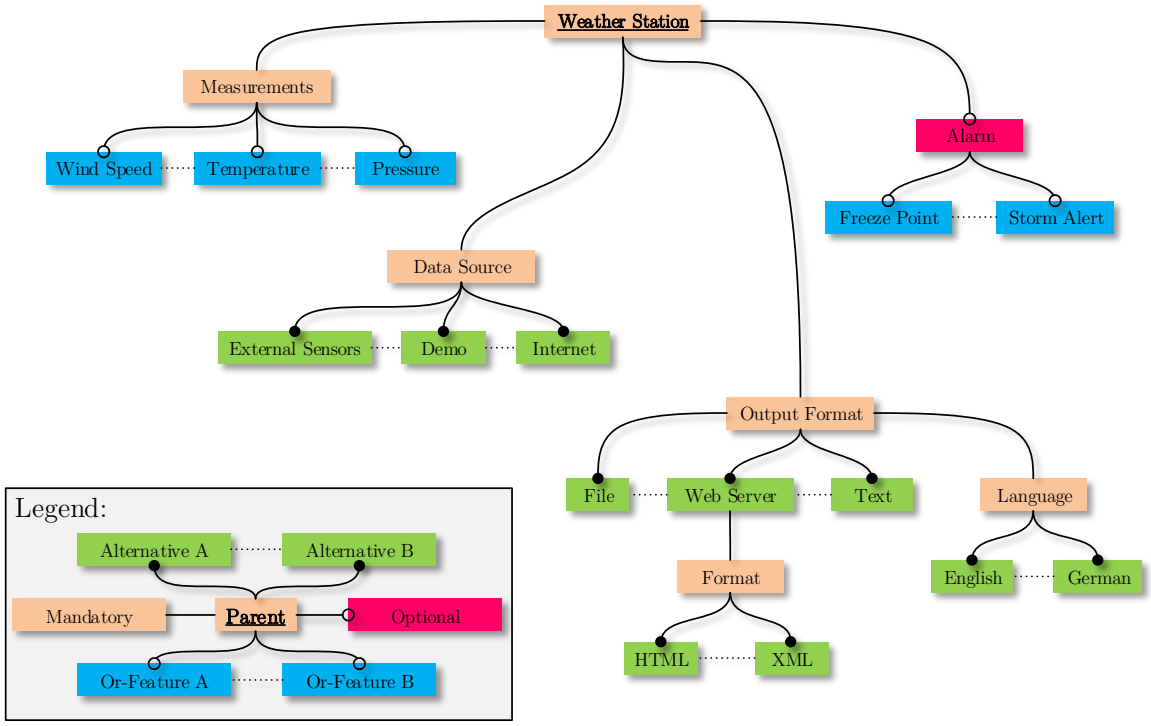


Abbildung 3.9: Feature-Modell am Beispiel einer Wetterstation nach [86]

3.2.2 Umwandlung in eine Produktlinie

Zur Umwandlung eines bestehenden Produktes in eine Produktlinie wird in [9, 63] zwischen drei Vorgehensweisen unterschieden. Der *proactive* Ansatz beschreibt dabei eine von Grund auf neu entwickelte Produktlinie basierend auf Analysen der zu entwickelnden Produkte. Die Ansätze *reactive* und *extractive* hingegen verwenden bestehende Software und wandeln diese je nach Bedarf entweder schrittweise (*extractive*) beziehungsweise komplett (*reactive*) um.

Simon und Eisenbarth beschränken sich in [111] auf die Unterteilung in evolutionäre und revolutionäre Umsetzung. Dabei ist die revolutionäre Umsetzung gleichzusetzen mit dem *proactive* Ansatz aus [63] und die *extractive* bzw. *reactive* Ansätze mit der evolutionären Umsetzung. Im Folgenden werden die insbesondere für *KMU* interessanten Aspekte aus [111] näher betrachtet.

Da das vorhandene Produkt schon von Kunden genutzt wird, muss dieses weiterhin gepflegt werden. Ein evolutionärer Ansatz erlaubt mit geringem Ressourcenaufwand eine gleichzeitige Pflege sowohl des alten Produktes als auch der zu entwickelnden Produktlinie. Weiterhin bietet dieser Ansatz den Entwicklern die Möglichkeit des „learning-by-doing“ und setzt somit keinen expliziten Produktlinien-Experten im Unternehmen voraus. Eine parallele Einführung ist den Erfahrungen aus [112] nach sowohl auf technischer als auch auf organisatorischer Ebene sinnvoll. Ein weiterer Vorteil ist das Erzielen von frühen Ergebnisse und Prototypen, da auf vorhandenen Grundlagen wie existierendem Quellcode aufgebaut werden kann. Die Wiederverwendung von bestehenden Systemteilen reduziert ebenfalls die Entwicklungskosten.

Demgegenüber steht zum einen der Nachteil, dass die bestehende Software-Architektur um eine Produktlinien-Funktionalität erweitert wird und nicht auf diesem Ansatz basiert, was einer konsequenten Umsetzung der Produktlinien-Ideen entgegenstehen kann. Nicht jedes Produkt, welches von Beginn an ohne Anforderungen an eine Produktlinie entwickelt worden ist, kann nachträglich in eine umgewandelt werden. Das Risiko, dass der gesamte Quellcode umstrukturiert werden muss, ist dabei hoch.

Zusammengefasst muss jedes Unternehmen die Vor- und Nachteile mit Blick auf das umzuwandelnde Produkt abwägen. Um die Risiken zu minimieren, sollte in jedem Projektablauf ein frühzeitiger Analyseschritt vorhanden sein, in dem eine Kosten-Nutzen Abschätzung durchgeführt wird. Auf deren Basis muss entschieden werden, ob die Umwandlung in eine Produktlinie weiter verfolgt oder abgebrochen wird.

In [111] werden für die Umstrukturierung in eine Produktlinie fünf notwendige Rollen vorgeschlagen. Dies umfasst einen Programmierer und Software Architekten des ursprünglichen Quellcodes, einen Domänen-Experten, den Endanwender bzw. Kunden, einen Programmierer, der die Umstrukturierung des Codes vornimmt, sowie einen Produktlinien-Experten. Um die Voraussetzungen an *KMU* anzupassen, werden im Folgenden die Prozesse nicht an Personen gekoppelt, da diese nicht zwangsläufig in einem Unternehmen einzeln vorhanden sein müssen bzw. durchaus auch in Personalunion existieren können. Daher werden die notwendigen Personen in notwendige Kompetenzen umgewandelt:

- Wissen über den bestehenden Quellcode sowie die Fähigkeit Programmcode zu entwickeln
- Externe Sicht auf das Produkt, um Anforderungen vorzugeben und ggf. auch zukünftige Anwendungsfälle miteinzubeziehen
- Produktlinien-Erfahrung bzw. Abstraktionsfähigkeit, um in „Features“ denken zu können

[111] beschreibt den Ablauf einer evolutionären Umwandlung eines vorhandenen Software-Produktes in eine Produktlinie. Dabei ist es möglich, mit nur hoch priorisierten Features anzufangen und die Produktlinie in erneuten Durchläufen um weitere Features zu ergänzen. Nach [111] sollen folgende Schritte iterativ durchlaufen werden:

- *Identifikation* von vorhandenen und nutzbaren Funktionen und deren Auflistung in einer Feature-Liste.
- *Priorisierung* der identifizierten Features, um den Aufwand der Umstrukturierung gering zu halten und um schnelle Ergebnisse zu erzielen.
- Feature-*Vorausschätzung*, um mögliche neue Features und Varianten für den Markt vorherzusehen.
- Feature-*Analyse* der für diesen Iterationsschritt ausgewählten Features:
  - *Lokalisierung* der einzelnen Features im Quellcode.
  - Analyse von *Gemeinsamkeiten* und *Variabilitäten*. Was teilen sich die einzelnen Features und wo sind diese eigenständig.
  - Analyse der *Abhängigkeiten* der verschiedenen Features untereinander.
  - Berechnung von *Metriken*, um eine grobe Vorabschätzung durchführen zu können, ob die Umwandlung in eine Produktlinie sinnvoll ist.
- *Aufwandsabschätzung* basierend auf der Feature-Analyse sowie der notwendigen Architektur-Anpassungen. In diesem Schritt erfolgt die Kosten-Nutzen-Abschätzung, auf Basis derer ein Abbruch des Umwandlungsprozesses erfolgen kann. Dabei wird die ursprüngliche Systemarchitektur um Produktlinien-Ansätze erweitert.
- *Umstrukturieren* des Systems aufgrund der entwickelten Features und der entsprechenden Produktlinien-Architektur.

Anwendung auf das Fallbeispiel

In diesem Abschnitt wird nun die Anwendung des vorab beschriebenen Prozesses zur Umwandlung des in Abschnitt 2.7 eingeführten Fallbeispiels in eine Produktlinie durchgeführt. Dazu wird der Prozess und dessen Anforderungen an die Ressourcen für KMU angepasst und auf das Fallbeispiel angewendet. Da das Software-System des Fallbeispiels eine überschaubare Größe hat (siehe dazu Software-Metriken in Tabelle 3.1), wird die Feature-Analyse manuell durchgeführt. In der Arbeit von Krirkkawin [62] wurde das Framework maßgeblich umgesetzt und in der Arbeit von Ratanaprutthakul [89] wurde es nachfolgend um die Anbindung an die formale Verifikation von Binär-Code (Kapitel 4) mittels ARCADE erweitert.

Code-Zeilen (SLOC)	4.663
Kommentar-Zeilen (CLOC)	5.493
SLOC / CLOC	1,18
Anzahl Funktionen	124
Anzahl Dateien	42

Tabelle 3.1: Code-Metriken der Software des Fallbeispiels

Die Umstrukturierung beginnt, wie oben beschrieben, mit der *Identifikation* von vorhandenen Features sowie der Analyse von deren Abhängigkeiten. Abbildung 3.10 zeigt das erstellte Feature Modell auf Basis der vorgegebenen Software des Fallbeispiels. Deutlich ersichtlich ist die Aufteilung in Master- und Slave-Funktionen. Dies basiert auf dem Aufbau der Hardware, welche den möglichen Einsatz von zwei Mikrokontrollern vorsieht. Für den Einsatz als Motorsteuerung ist einer der beiden Mikrokontroller mit Sensoren am Motor verbunden, der die Stellung des Motors (Kurbelwelle) auswertet. Die auf diesen Sensorwert angewiesenen Features sind demzufolge alle von Slave abhängig. Die Software-Architektur bleibt zu Abschnitt 2.7 vorerst unverändert.

Um die neue Architektur der Produktlinie auch für zukünftige Anwendungen auszulegen, folgt als nächster Schritt die *Analyse des Marktes* und der dadurch sinnvollen neuen Features bzw. Varianten. Der ursprüngliche Anwendungsfall des Fallbeispiels lag in der Regelung des λ-Wertes durch die Steuerung der Piezo-Position in einem Ventil. Mögliche neue Anwendungsfälle beziehen sich auf die Steuerung von kleinen Motoren. Dabei werden Features gefordert wie Zündung des Motors, Einspritzung von Kraftstoff sowie die Erkennung der Drehzahl oder die Auswertung von Sensoren zur Lasterkennung (*Manifold Absolute Pressure (MAP)*). Die auf die Identifikation von Features folgende *Priorisierung* war bei der überschaubaren Anzahl an identifizierten Features nicht notwendig. Die Umstrukturierung soll somit alle Funktionen mit einschließen, was eine Priorisierung nicht erforderlich macht.

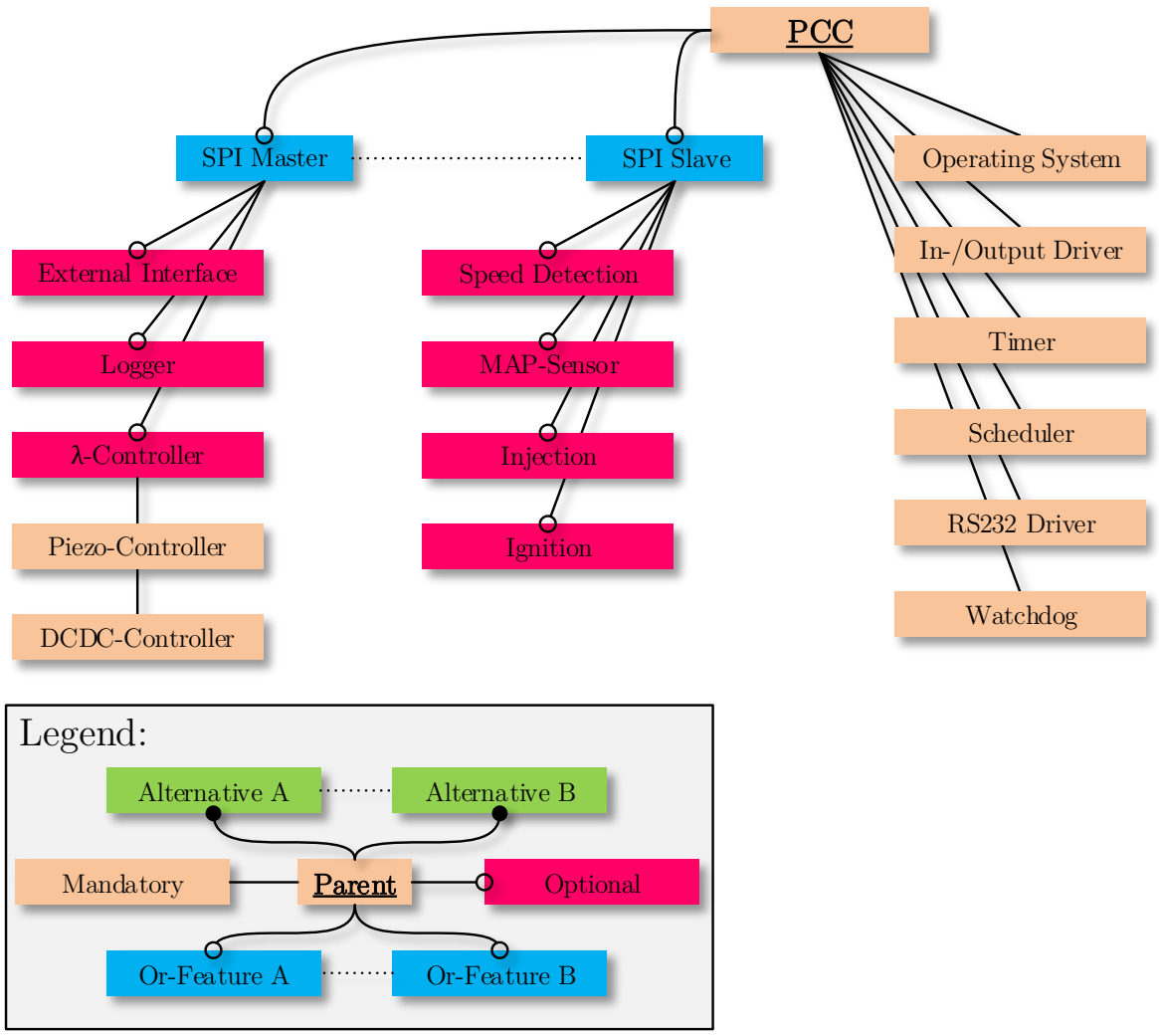


Abbildung 3.10: Feature-Modell des Fallbeispiels

Im Folgenden werden die identifizierten Features entsprechend zu Abbildung 3.10 beschrieben:

- Betriebssystem, welches das System hochfährt, den Speicher vorbereitet (*Operation System*)
- *Scheduler* zur Abarbeitung der vorhanden Tasks
- Treiber für die Ein- und Ausgabe von Signalen (*Input-/Output Driver*)
- *Timer* Funktionen zum Erzeugen und Messen von zeitabhängigen Signalen (z. B. *Pulsweitenmodulation (PWM)*)

- Treiber für die serielle *RS232* Schnittstelle für die Kommunikation, z. B. zur externen Bedatung des Steuergerätes während der Inbetriebnahme
- *Watchdog* zur Überwachung des Systemzustandes
- Erkennung der Drehzahl (*Speed Detection*)
- Funktionen der Motorsteuerung zur Zündung (*Ignition*) und zur Einspritzung von Kraftstoff (*Injection*)
- Auswertung von Sensoren zur Lasterkennung (*MAP*)
- Regelung des λ-Wertes zur Abgasreduzierung (λ-*Controller*), davon abhängig ist die Regelung der Piezo-Ventil-Stellung im Piezo-Vergaser (*Piezo-Controller*), und davon abhängig ist die Regelung der DC/DC-Spannungsversorgung (*DCDC-Controller*) für das Piezo-Ventil
- Datenaufzeichnung (*Logger*)
- Protokoll-Interpreter für die Kommunikation mit externen Anwendungen oder auch Applikationssystemen (*External Interface*)

Die anschließende *Feature Analyse* wurde manuell durchgeführt. Dazu mussten zuerst die identifizierten Features im Quelltext genauer analysiert werden, sowie Gemeinsamkeiten und feature-individuelle Funktionen lokalisiert werden. Abbildung 3.11 zeigt sowohl die Variabilitäten, die in der eingebetteten Software zur Implementierung des Features verwendet werden, als auch diejenigen Teile, die zur Verifikation notwendig sind. Somit hat jedes Feature die Möglichkeit, eigenen Code bzw. eigene Anpassungen an den gemeinsam genutzten Stellen im Quelltext bzw. für die Verifikationswerkzeuge hinzuzufügen.

Folgende Variabilitäten wurden dabei in [62] identifiziert:

- Die Verwaltung von Port-Funktionen (digitale Ein-/Ausgänge, analoge Eingänge, serielle Schnittstellen, etc.) wird global organisiert. Verwendet ein Feature z. B. einen analogen Eingang, so muss dieser in der globalen *Port-Konfiguration* diesem Feature zugeordnet und entsprechend konfiguriert werden.
- In den zentral verwendeten Dateien zur Verwaltung der Laufzeitumgebung können Features Schnittstellenbeschreibungen (*Include Dateien*) einbinden, um deren Funktionen allgemein zugänglich zu machen.
- Features haben die Möglichkeit, eigene Schnittstellen-Funktionen in die globale *Initialisierungsroutine* hinzuzufügen.
- Werden Interrupts von einem Feature verwendet, müssen entsprechende Einträge in der globalen *Interrupt-Vektor-Tabelle* ergänzt werden.

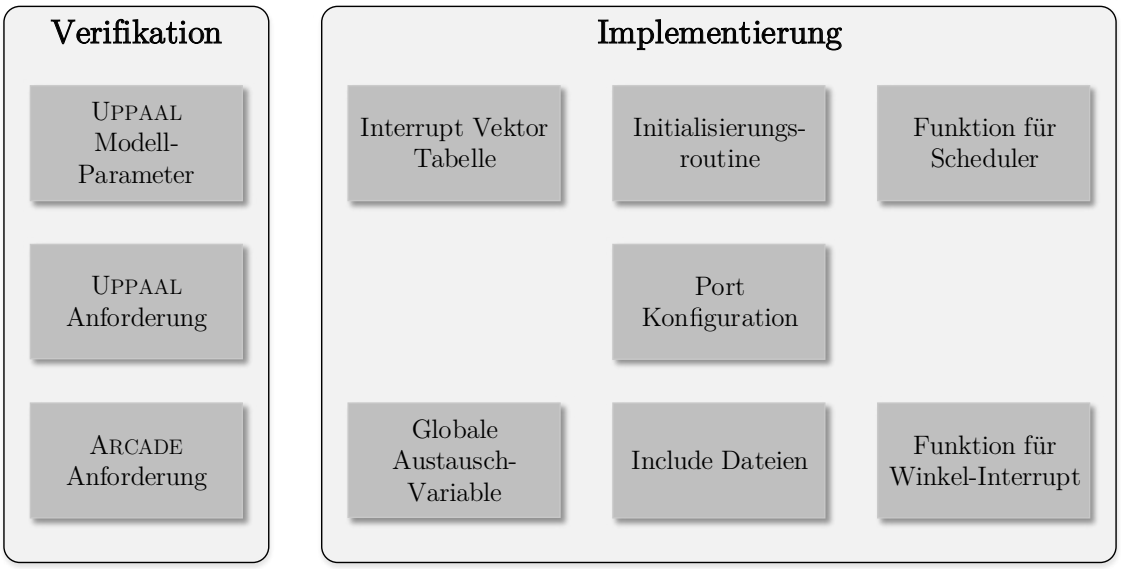


Abbildung 3.11: Feature-Variabilitäten der Produktlinien-Architektur

- Da in einem automobilen Steuergerät viele Funktionen kurbelwellenwinkelsynchron abgearbeitet werden, besteht die Möglichkeit, eine Feature-Funktion während eines globalen *Winkel-Interrupts* ausführen zu lassen.
- Damit periodisch Funktionen ausgeführt werden können, kann jedes Feature Funktionen auswählen und diese System-Taktraten (100ms, 500ms, maximal) zuordnen. (*Funktion für Scheduler*)
- Um anderen Features z. B. Sensor-Signale zur Verfügung zu stellen, können Austausch-Kanäle in Form von *globalen Variablen* definiert werden.
- Werden von einem Feature periodische Funktionen zum Task-Scheduler hinzugefügt, müssen für die Einplanbarkeitsanalyse (siehe Abschnitt 3.1.5) entsprechende UPPAAL *Modell-Parameter* (Periode, Ausführungszeit, Deadline, Unterbrechbarkeit, Interrupt) definiert werden.
- Damit UPPAAL *Anforderungen* an das Task-Modell verifizieren kann, muss für jedes Feature diese entsprechend formalisiert und in einer entsprechend formatierten Datei abgelegt werden.
- Um mittels ARCADE *Anforderungen* im kompilierten C-Code verifizieren zu können, werden beliebige CTL-Formeln für jedes Feature hinterlegt. Diese werden in einer passenden XML-Datei abgespeichert.

Die Kosten-Nutzen-Analyse resultierte aufgrund der Größe des Software-Systems und der Anzahl der identifizierten Features in die Umsetzung des bestehenden Produkts in eine Produktlinie.

3.2.3 Evaluierung

Das Ergebnis ist ein Varianten-Management, welches auf pure::variants basiert. Nach erfolgreicher Durchführung der Umwandlung in eine Produktlinie wurde die dem Ausgangssystem entsprechende Variante mit Hilfe des Werkzeugs erzeugt. Der aus der Variante resultierende C-Quelltext konnte erfolgreich kompiliert und auf die Hardware geflasht werden. Testfälle, die schon für das Ausgangssystem definiert worden waren, konnten erfolgreich ([62]) auch an der generierten Variante validiert werden. Das ebenfalls generierte Modell für Uppaal samt formalisierter Anforderungen wurde in der Benutzeroberfläche von Uppaal mit Erfolg geladen und ausgeführt.

Vergleicht man die in Abschnitt 3.2 definierten Anforderungen mit dem Resultat der Entwicklung des Varianten-Management Frameworks, so zeigt sich, dass alle Anforderungen durch das Framework erfüllt werden. Mittels pure::variants können sowohl Abhängigkeiten zwischen den einzelnen Modulen modelliert sowie der Quelltext der Module in anderen Projekten bzw. Varianten wiederverwendet werden. Es können weiterhin externe Werkzeuge wie z. B. Uppaal mit in den Prozess integriert werden. Über Schnittstellen können entsprechende Aufgaben wie die Einplanbarkeitsanalyse (siehe Abschnitt 3.1) durchgeführt werden. Um die Umwandlung von einem bestehenden Produkt in eine Produktlinie durchzuführen, wurde eine für diesen Fall vorhandene Methodik entsprechend angepasst, durchgeführt und die resultierende Software erfolgreich evaluiert. Speziell *KMU* stärken durch dieses leichtgewichtige Werkzeug ihr hohes Maß an Flexibilität und erhöhen ihre Wirtschaftlichkeit. Dies erreichen sie, indem sie sich stärker auf die individuellen Kundenwünsche fokussieren und dabei vorhandene Komponenten effizient und mit hoher Qualität wiederverwenden können. Die weitere Integration des Frameworks in den bestehenden Entwicklungsprozess eines *KMU* wird in Kapitel 5 beschrieben.

4 Verifikation von Binär-Code

Die Software, die auf üblichen automobilen Steuergeräten Verwendung findet, wurde zuvor entweder manuell codiert oder über einen modellbasierten Ansatz generiert. Bei der manuellen Codierung werden typischerweise Programmiersprachen wie C, C++ oder Assembler eingesetzt. Betrachtet man modellbasierte Kodierverfahren, so werden dort meist Modellierungswerkzeuge wie z. B. Mathworks Simulink oder dSPACE TargetLink eingesetzt. Die damit erstellten Funktionsmodelle werden über integrierte Code-Generatoren in C-Code transformiert. Dieser C-Code wird mittels hardwareabhängigem C-Compiler in Binär-Code übersetzt.

Bei großen Softwareprojekten, bei denen mehrere Entwickler in den verschiedenen Ebenen an der Softwareerstellung beteiligt sind, ist es sinnvoll, in jeder der Ebenen die gegebenen Anforderungen direkt zu verifizieren, um frühzeitig Fehler zu finden. Diese können in frühen Phasen der Entwicklung kostengünstiger behoben werden als in späteren. Dabei haben diese Anforderungen einen unterschiedlichen Grad der Abstraktion, passend zur erstellten Software (Modell, C-Code, Binär-Code). Zur Verifikation werden Test-Methodiken verwendet, bei denen die Software vordefinierten Testfällen genügen muss. Dabei wird meist mit einzelnen Software-Modulen begonnen und nach der Integration in das Gesamtsystem (Fahrzeug) abgeschlossen.

Da die Entwicklung von Software für eingebettete Systeme großteils nicht auf dem Ziel-System selbst, sondern auf den Entwickler-PCs erfolgt, unterscheidet man mehrere Phasen der Verifikation bei der Entwicklung von Steuergeräten. Diese Phasen sind in Abbildung 4.1 dargestellt. Betrachtet man die modellbasierte Software-Entwicklung, so existiert von der Funktionssoftware zuerst ein Modell, welches auf dem Entwickler-PC ausführbar ist und im Model-in-the-Loop (MIL) gegen eine simulierte Umgebung getestet werden kann. In dieser Simulation sind neben dem Sensor- und Aktorverhalten auch andere Steuergeräte abgebildet, die über unterschiedliche Busse mit dem zu entwickelnden Steuergerät kommunizieren. Im Software-in-the-Loop (SIL) Schritt wird aus dem Funktions-Modell Code generiert und dieser auf dem Entwickler-PC nativ ausgeführt. Bei Processor-in-the-Loop (PIL) wird der Code für das Ziel-System kompiliert und in einer dazu passenden virtuellen Umgebung verwendet, um bei Hardware-in-the-Loop (HIL) direkt auf der Ziel-Hardware ausgeführt zu werden. Da die Entwicklungen von Software und Hardware-Komponenten meist parallel erfolgen, können die Software-Komponenten mit diesen Methoden frühzeitig in Umgebungen, die immer realitätsnäher sind, getestet werden.

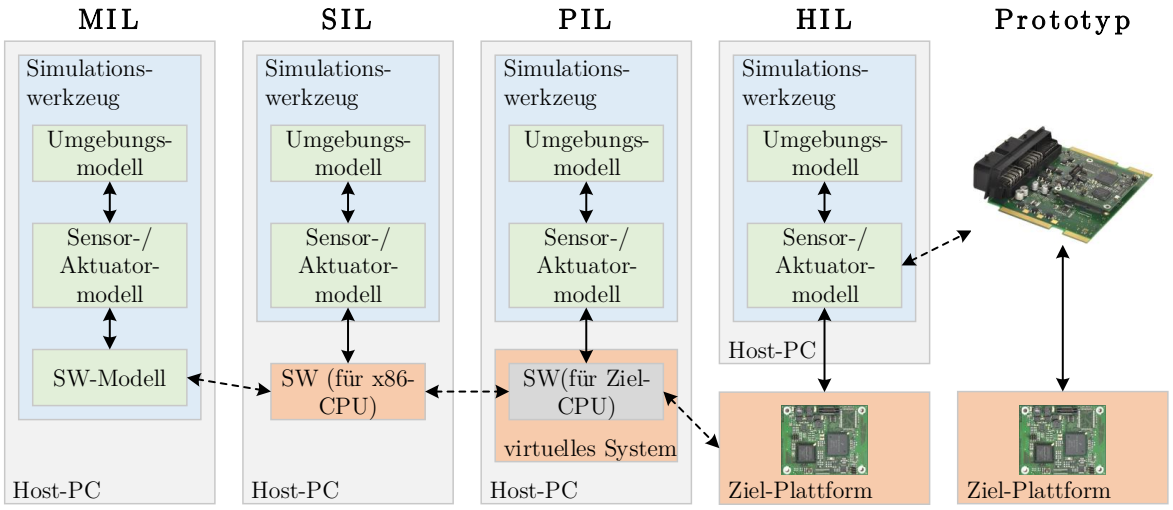


Abbildung 4.1: Unterschiedliche Phasen der Verifikation während der Steuergeräteentwicklung (angelehnt an [80])

Der Vorteil des Testens ist die einfache Anwendbarkeit auf komplexe Systeme und die Unterstützung durch zahlreiche industriell bewährte Werkzeuge. Die Nachteile sind jedoch, dass Testen sehr zeitintensiv ist und es damit nicht möglich ist, die Abwesenheit von Fehlern zu belegen. Testen deckt in keinem Fall alle möglichen Situationen ab, in denen sich eine Software befinden kann, sondern immer nur Ausschnitte davon.

Im Vergleich dazu kann die formale Verifikation einen Beweis für das Einhalten von gestellten Anforderungen liefern. Das Auffinden von Fehlern hängt dabei nicht von deren Auftrittswahrscheinlichkeit ab. Der Einsatz von formalen Methoden ist nicht nur für sicherheitskritische Systeme sinnvoll, sondern auch in der Massenproduktion von eingebetteten Systemen, da dort schon kleine Fehler hohe Kosten verursachen können.

Im Folgenden werden daher Anforderungen zum Einsatz von formaler Verifikation im Software-Entwicklungsprozess aufgestellt. Dabei werden die Randbedingungen bezogen auf die existierende Software-Plattform eines *KMU* in der Automobilindustrie speziell mit einbezogen. Diese umfassen zum einen den im Vergleich zu großen Unternehmen meist deutlich geringen Anteil von spezialisierten Testern unter den zur Verfügung stehenden Entwicklern. Zum anderen müssen kleine Unternehmen sowohl aus Wettbewerbssicht als auch aus regulatorischer Sicht die gleiche Softwarequalität liefern wie große Unternehmen mit entsprechend starren aber bewährten Prozessen. Weiterhin besteht für *KMU* durch Fehler in der Software für Seriensteuergeräte ein erhöhtes finanzielles Risiko, welches bei großen Unternehmen meist einfacher abgefangen werden kann.

Im Anschluss an die Anforderungen wird eine dazu passende Methodik erarbeitet und an einem Fallbeispiel evaluiert.

Anforderungen

- Zur Verifikation der Software sollen im Entwicklungsprozess bereits erzeugte Artefakte verwendet werden.
- Der Einsatz von Ressourcen zum Finden von Fehlern soll verringert werden.
- Fehler in der gesamten Kette der Software-Entwicklung sollen gefunden werden können.
- Die Funktions-Software wird auf eingebetteten Systemen ausgeführt, deren Besonderheiten im Verifikations-Werkzeug berücksichtigt werden sollen.
- Falls Fehler gefunden werden, sollen diese durch den Entwickler nachvollziehbar dargestellt werden.
- Die Formalisierung der Modelle sowie der Eigenschaften soll keine hochgradig spezialisierten Experten erfordern.
- Der Verifikationsprozess soll in vorhandene Entwicklungsprozesse integriert und automatisiert werden können.

4.1 Mögliche Werkzeuge zum Model-Checking

In der Industrie wird Model-Checking vor allem bei der Verifikation von elektronischen Schaltungen eingesetzt, sowie auch zur Überprüfung von Software. Dong et al. [37] vergleicht fünf bekannte Model-Checker zur Verifikation eines Kommunikationsprotokolls, welches in der Industrie eingesetzt wird. Dabei mussten allerdings viele Ressourcen für die Modellierung des Systems verwendet werden, um damit die gewünschten Eigenschaften verifizieren zu können. Neben Model-Checkern, die für die Modellierung des Systems eigene Methoden bereit stellen, existieren auch Werkzeuge, die direkt Programmcode als System-Modell verwenden können. Neben Ada und Java werden auch C und C++ als Programmiersprache unterstützt. In dieser Arbeit liegt der Fokus auf der Entwicklung von automotiven Steuergeräten, welche normalerweise in C, C++ oder Assembler programmiert werden.

Im Avionik sowie im Automotive Umfeld werden industriell-bewährte Model-Checker wie BTC Embedded Validator[14] oder SCADE[15] verwendet. Diese werden häufig in der modellbasierten Software-Entwicklung eingesetzt. Model-Checker auf Basis von C-Code

[14] http://www.btc-es.de
[15] http://www.esterel-technologies.com

hingegen sind z. B. Spin [16], CBMC [17] oder BLAST [18]. Schlich stellt in [99] eine ausführliche Liste verfügbarer C-Code Model-Checker vor.

In [100, 101, 103] wird beschrieben, wieso C-Code Model-Checker für eingebettete Systeme nicht immer ausreichend sind. Hierzu gehört die fehlende Unterstützung aller Hardware-abhängigen Komponenten, wie Interrupts, Register, Zähler und direkte Speicherzugriffe. Weiterhin werden meist nur eine eingeschränkte C-Syntax sowie keine Assembly-Anweisungen unterstützt. Da auf C-Code Ebene keine für die Hardware optimierten Strukturen beachtet werden können, kann dies Auswirkungen auf die Größe des Zustandsraumes haben. Desweiteren können dadurch auftretende Fehler nicht gefunden werden. Weitere Fehler in Bibliotheken z. B. von Dritten sowie im Compiler bleiben auf der C-Code-Ebene unentdeckt.

Java Pathfinder[19] ist ein Model-Checker für Java-Byte-Code. Dieser Code wird in der virtuellen Maschine ausgeführt und somit der Zustandsraum aufgebaut, mit dem das Model-Checking durchgeführt wird. Ein ähnliches Vorgehen, allerdings auf Basis von C++ Code, verwendet StEAM[20]. Für eingebettete Systeme existieren Estes [70] (für Motorola 68hc11) und MCESS [98]. Beide erzeugen ebenfalls den Zustandsraum auf Basis einer virtuellen Maschine. Arcade [21] verwendet Binär-Code für On-The-Fly Model-Checking von Programmen für Mikrokontroller und kann compilierten Code als System-Modell verwenden. Eine Abgrenzung der Model-Checker StEAM, Estes und MCESS zu Arcade erfolgt in [99, 101].

4.2 Binär-Code Model-Checking mit Arcade

Die folgenden Informationen sind zusammengefasst aus [47, 91, 92, 99, 100, 102–106]. Arcade.μC ist ein Werkzeug zur formalen Analyse von Programmen speziell für Mikrokontroller. Es enthält Methoden zur statischen Analyse, abstrakten Interpretation sowie zum Model-Checking. Es wurde am Lehrstuhl Informatik 11 - Embedded Software der RWTH Aachen University in Java entwickelt und basiert auf dem Eclipse Rich Client Framework sowie auf den in Unterabschnitt 2.6.1 beschriebenen Ideen. Die Vorteile der Anwendung von Model-Checking direkt auf den Binär-Code sind die Einbeziehung der gesamten Kette der Code-Entstehung von der manuellen Erstellung über das Verwenden von vor-compilierten Bibliotheken bis hin zum Compiler. Neben Fehlern in dieser Prozesskette können auch Fehler in der Verwendung von hardwarespezifischen Modulen, wie sie häufig bei Mikrokontrollern vorkommen, gefunden werden.

Arcade unterscheidet zwischen den Versionen für speicherprogrammierbare Steue-

[16] `http://spinroot.com`
[17] `http://www.cprover.org/cbmc`
[18] `http://forge.ispras.ru/projects/blast`
[19] `http://babelfish.arc.nasa.gov/trac/jpf`
[20] `http://steam.cs.uni-dortmund.de`
[21] `http://arcade.embedded.rwth-aachen.de`

rungen (ARCADE.PLC) sowie für Mikrokontroller (ARCADE.μC). Bei letzterer werden verschiedene Mikrokontroller unterstützt. Zu den aktuellen zählen ATMEL ATmega16 und ATmega128, Renesas R8C\23.

Die Architektur von ARCADE.μC zeigt Abbildung 4.2. Das Werkzeug besteht hauptsächlich aus den folgenden vier Komponenten:

- Die *Parser*-Module überführen die Eingabedateien in eine interne Repräsentation.
- Die *Statische Analyse* führt eine Vorverarbeitung der Daten durch und stellt diese Ergebnisse dem Model-Checker zur Verfügung.
- Der *Model-Checker* prüft, ob die spezifizierte Eigenschaft im Modell des Systems gültig ist.
- Der *Generator für Gegenbeispiele* erstellt bei Bedarf eine Visualisierung des Zeugen oder des Gegenbeispiels für das Ergebnis des Model-Checkers.

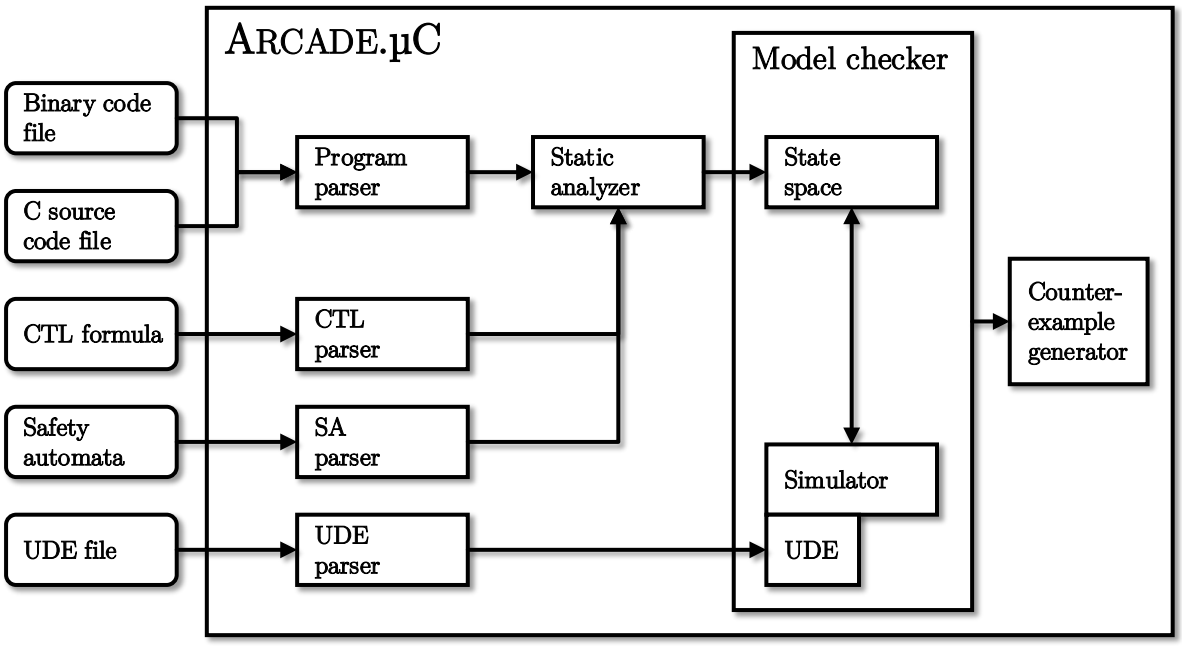


Abbildung 4.2: Architektur von ARCADE.μC. Das Bild basiert ursprünglich auf Schlich und wurde durch Kamin [57] erweitert.

Im Detail läuft der Prozess des Model-Checkings mittels ARCADE wie folgt ab:

Der Benutzer stellt als Eingabe für das System ein *Binary code file* sowie ein *C source code file* zur Verfügung. Ersteres beinhaltet den kompilierten C-Code des Systems als Maschinencode. Das System-Modell wird daraus später automatisch abgeleitet. Hierzu muss der C-Code weder transformiert noch muss das System manuell nachmodelliert werden. Als Format werden Intel Hex, Motorola S Record sowie

Executable and Linking Format (ELF) [31] unterstützt, wobei letzteres neben dem eigentlichen Maschinencode zusätzlich noch Debug-Informationen enthält, um Verbindungen zum C-Code herstellen zu können. Der *Program parser* überführt den Maschinencode in eine entsprechende interne Repräsentation.

Weiterhin muss der Benutzer die formalisierten Eigenschaften, die gegen das System geprüft werden sollen, in Form von *CTL*-Formeln (siehe Kapitel 2.5.3) oder als Safety-Automaten (siehe Kapitel 4.3.2) bereitstellen. Diese können z. B. Aussagen über Register des Mikrokontrollers und Variablen des Programms sowie über den Programmzähler (engl. program counter, PC) enthalten. Die Eigenschaften werden mittels *CTL parser* und *SA parser* entsprechend für den internen Gebrauch analysiert.

Optional kann der Benutzer noch ein Modell der Umgebung dem System hinzufügen, um den Nicht-Determinismus einzuschränken und damit ggf. die Größe des Zustandsraumes zu reduzieren. Dies geschieht über einen Automaten, der in einem *UDE file* (siehe Kapitel 4.2) hinterlegt ist und durch den *UDE parser* den restlichen Modulen zur Verfügung gestellt werden kann.

Vor dem eigentlichen Prozess des Model-Checkings wird das zu prüfende System einer Statischen Analyse (*Static analyzer*) unterzogen. Dieses Modul führt verschiedene Analysen durch und kennzeichnet Zustände für die spätere Verwendung von Abstraktionstechniken. Der Simulator verwendet später diese Kennzeichnung zur Reduktion des zu erstellenden Zustandsraumes. Weiterhin erstellt dieses Modul den Kontrollflussgraphen des Maschinencodes, der zur späteren Visualisierung verwendet wird.

Nach erfolgter statischer Analyse startet der eigentliche Model-Checking Prozess. Hierzu werden verschiedene Algorithmen (lokal, global, Invarianten) verwendet. Das *Model checker* Modul beginnt mit dem initialen Zustand und analysiert, ob die zu prüfende Formel gültig ist. Abhängig vom Ergebnis wird der Folgezustand beim *State space* Modul angefragt. Dieses fordert, je nach Einstellung, den Zustand beim *Simulator* On-The-Fly an oder liest ihn aus dem zuvor erstellen Zustandsraum aus. Daraufhin beginnt das *Model checker* Modul mit der erneuten Prüfung der Formel auf ihre Gültigkeit. Wird eine Formel nicht oder vor Durchlaufen des gesamten Zustandsraumes frühzeitig gültig, so bricht der Prozess ab, und ein Zeuge bzw. Gegenbeispiel wird durch das *Counterexample generator* Modul erzeugt. Diese können vom Benutzer schrittweise in verschiedenen Darstellungen (Kontrollflussgraph, C-Code, Maschinencode) durchgegangen werden.

Das *State space* Modul erzeugt den zum System passenden Zustandsraum und kann, je nach Einstellung, diesen im Arbeitsspeicher oder auf der Festplatte speichern. Zustände können On-The-Fly erzeugt werden oder komplett im Vorfeld. On-The-Fly hat den Vorteil, dass beim Auftreten eines Fehlers der Model-Checking Prozess abgebrochen wird, ohne den gesamten Zustandsraum aufzubauen. Ein Zustand besteht aus allen Registern inklusive Ein- und Ausgabe-Registern, dem kompletten RAM, zusätzlichem Speicher für Kennzeichnungen (z. B. durch das *Static analyzer* Modul) sowie einer Liste aller Folgezustände. Weiterhin beinhalten die Zustände auch die Werte der zu prüfenden Formel.

Das *Simulator* Modul ist teilweise spezifisch für den Mikrokontroller angepasst, da

der Maschinencode hardwareabhängig ist. Jeder Mikrokontroller hat einen eigenen Satz an Instruktionen und eine eigene Semantik sowie zusätzliche spezielle Register, die das Verhalten des Mikrokontrollers unterschiedlich beeinflussen. Der hardwarespezifische Teil kann aus einer standardisierten Architekturbeschreibung (SGDL) synthetisiert werden (siehe [47]). Das Modul wird von anderen Modulen in ARCADE dazu verwendet, um das hardwarespezifische Verhalten abzubilden. Weiterhin wird es genutzt, um Folgezustände für das *State space* Modul zu erstellen. Dabei werden verschiedene Abstraktionstechniken genutzt, um die Größe des Zustandsraumes zu verringern.

User-Defined Environments

ARCADE generiert den Zustandsraum mit Hilfe des Simulators. Dieser kann Nichtdeterminismus verarbeiten, indem er das Verhalten des realen Mikrokontrollers über-abschätzt. Dafür werden z. B. bei jedem Zugriff des Simulators auf Eingänge des Mikrokontrollers diese als nichtdeterministisch angesehen. Der Model-Checker verarbeitet diese, indem er jeden möglichen Zustand der Eingänge berücksichtigt, um die Gültigkeit der Ergebnisse des Model-Checking Vorgangs nicht negativ zu beeinflussen. Dies kann zu einem explosionsartigen Anstieg der Zustände im Zustandsraum führen. Um dieses Verhalten zu begrenzen, werden verschiedene Abstraktionstechniken eingesetzt. Eine weitere Möglichkeit, den Zustandsraum zu begrenzen, besteht darin, die Zustände auszuschließen, die im realen System unmöglich vorkommen können. Dazu kann die Umgebung des Mikrokontrollers modelliert werden mittels *User-Defined Environment (UDE)* [105]. Die Modellierung erfolgt dabei durch endliche Zustandsautomaten. Das erstellte Modell wird dem Simulator beim Generieren des Zustandsraumes zur Verfügung gestellt, um das Verhalten der Umgebung zu berücksichtigen und unpassende Zustände direkt zu verwerfen. Dadurch kann der Umfang des Nichtdeterminismus eingegrenzt und somit eine Explosion des Zustandsraumes vermieden werden. Wird das reale System dabei jedoch fehlerhaft modelliert, so sind die Ergebnisse des Model-Checkings nicht gültig.

Speziell Interrupts generieren, wenn diese im aktuellen Zustand erlaubt sind, eine große Anzahl an Zuständen. Jedem Zustand folgt dabei automatisch ein zusätzlicher Zustand, in dem der Interrupt aktiv wird. Interrupts basieren auf Ereignissen, die von der Umgebung ausgelöst werden. Schränkt man diese Ereignisse nicht über *UDE* ein, so können diese zu jedem möglichen Zeitpunkt (Nichtdeterminismus) ausgelöst werden. Mittels *UDE* kann dieses Verhalten an das reale System angepasst werden.

Das Beispiel in Abbildung 4.3 zeigt zwei Zustände, den initialen sowie einen weiteren. In *Zustand 1* können z. B. alle Interrupts bis auf einen explizit geblockt und dieser eine auf jeden Fall ausgelöst werden. Der Zustandsautomat wechselt ausgehend von dem initialen Zustand in *Zustand 1*, wenn die Transitionsbedingung erfüllt ist. Diese kann dabei das Betreten bzw. Verlassen von Interrupt-Service Routinen sein oder auch das Lesen von Eingängen sowie das Setzen von Ausgängen des Mikrokontrollers .

Neben dem grundsätzlichen bzw. situationsabhängigen Blockieren von Interrupts ist es mit *UDE* auch möglich, Wertebereiche von Eingängen zu beschränken sowie auf das

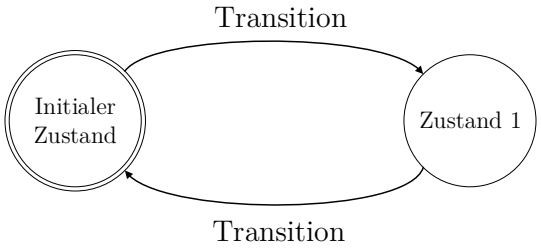


Abbildung 4.3: *UDE* Beispiele

Setzen von Ausgängen zu reagieren. Betrachtet man z. B. den Zugriff auf einen AD-Wandler Eingang mit einer Auflösung von 10-Bit ohne den Einsatz von *UDE*, so erstellt der Simulator aufgrund des Nichtdeterminismus des Eingangs 1024 Folgezustände. Wird mittels *UDE* jedoch der mögliche Wertebereich des Eingangs auf 410.. 600 beschränkt, so werden nur 191 Folgezustände generiert.

Beim Einlesen von Eingängen werden die entsprechenden Wertebereiche festgelegt, mit denen der Simulator daraufhin die passenden Folgezustände generiert. Ist eine Transitionsbedingung an das Setzen eines Ausgangs gekoppelt, so wechselt der Automat in den entsprechenden Zustand, sobald der Simulator den passenden Ausgang beschreibt. Daraufhin können im neuen Zustand z. B. andere Interrupts geblockt werden oder unterschiedliche Wertebereiche für Eingänge gelten.

4.3 Wie können Anforderungen formalisiert werden?

Nach Abbildung 2.7 benötigt der Model-Checking Prozess drei Dinge: Ein Modell des Systems, formalisierte Anforderungen, die verifiziert werden sollen und den Model-Checker. Im speziellen Fall von ARCADE muss das Modell des Systems nicht manuell erstellt werden. Der in Maschinen-Code kompilierte C-Code dient hierbei direkt als Modell des zu verifizierenden Systems. Somit beschränkt sich der Arbeitsaufwand auf das Aufstellen und Formalisieren der Anforderungen, die das System erfüllen sollen.

4.3.1 Related Work

In der praktischen Anwendung von formaler Verifikation scheitern die meisten Entwickler beim Erstellen der formalen Spezifikation an theoretischem Grundlagenwissen über logische Sprachen [20, 49]. Diese Barriere kann durch Mitarbeiter mit Expertenwissen oder durch längere Einarbeitungszeiten bzw. Schulungen reduziert werden. Dies ist allerdings ein Zeit- und Kostenfaktor, der sich speziell bei *KMU*s deutlich bemerkbar macht. Heitmeyer [49] z. B. schlägt drei Wege vor, um die Schwierigkeiten beim Lernen und Anwenden von formalen Methoden zu reduzieren:

- Die Sprache, in der formal spezifiziert wird, soll für den Entwickler *natürlich*, also bekannt und gebräuchlich sein.
- Prozesse zur formalen Analyse sollten so viel wie möglich automatisiert werden.
- Dem Anwender sollen die Resultate der Verifikation verständlich präsentiert werden.

Bitsch beschreibt in [21] folgende Möglichkeiten, welche durch Preusse [83] analysiert und verglichen werden, zur Spezifikation von Anforderungen:

natürliche Sprache
: Diese ist mehrdeutig und unpräzise und somit ungeeignet für den Einsatz zur formalen Spezifikation.

semi-formale Darstellung
: Diese, sowohl textuell als auch grafisch, basiert auf einer vollständig festgelegten Syntax mit jedoch eingeschränkter mathematischer Basis und ist somit nicht direkt für formale Spezifikation verwendbar.

Normsprache
: Dies ist eine konstruierte Terminologie auf Basis einer Sprache eines speziellen Fachbereiches, in der Wörtern stets dieselbe Bedeutung zukommt [21]. Dabei ist ausschließlich die Verwendung von definierten Begriffen aus einem Lexikon bzw. Wörterbuch (eindeutige Syntax) zugelassen, um Vagheiten und Mehrdeutigkeiten zu vermeiden. Lässt sich aus einer Normsprache eindeutig eine formale Darstellung ableiten, so ist diese Möglichkeit für die formale Spezifikation von Anforderungen geeignet.

formale Darstellung
: Diese sowohl textuell als auch grafische Form hat eine eindeutige Syntax und Semantik und ist mathematisch exakt definiert. Dadurch existiert eine eindeutige Interpretation, und sie ist somit geeignet für die formale Spezifikation von Anforderungen.

Die mögliche Normsprache *Safety-Oriented Technical Language (SOTL)* zur Spezifikation von Anforderungen beschreibt Preuße in [84]. Dabei können mittels eines Compilers aus den *SOTL* Ausdrücken *CTL* Formeln abgeleitet werden. *SOTL* berücksichtigt den *natürlichen* Sprachgebrauch von Ingenieuren. Bei komplexeren Anforderungen, z. B. mit ineinander verschachtelten Bedingungen, werden die Ausdrücke länger als ihre Entsprechung in einer formalen Darstellung, z. B. als *CTL* Formeln.

Eine weitere Möglichkeit auf Basis von *Sequence Timing Diagrams (STD)* [107] formale Spezifikationen zu erstellen, beschreibt Preuße in [82]. Die grafische formale Darstellung vereinfacht die Anwendung vor allem bei der Spezifikation von komplexen Sequenzen in Herstellungsprozessen. Dabei werden die Möglichkeiten auf der Ebene von *CTL* Formeln eingeschränkt auf folgendes Aussehen:

$$\mathbf{AF}\ (\ \text{Ausdruck1}\ \wedge \mathbf{EF}\ (\text{Ausdruck2}\ \wedge\ \ldots\ )) \tag{4.1}$$

Bitsch entwickelt in [21] Entwurfsmuster zur Spezifikation für Automatisierungssysteme mittels formaler Darstellung in Temporallogik. Dazu stellt Bitsch speziell für Sicherheitsanforderungen passende Muster bereit. Damit werden dem Entwickler die zu seinen Anforderungen passenden Muster über eine webbasierte Auswahlhilfe zur Verfügung gestellt. Alle Muster sind in einer Datenbank hinterlegt. Dabei werden diese in einer Normsprache, in einer formalen Darstellung textuell, z. B. in *CTL* oder *LTL*, und grafisch in *Grafische Notation zur Spezifikation mit Safety-Pattern (GNSS)* sowie zur Erläuterung in natürlicher Sprache und in Form eines Berechnungsbaums dargestellt. Ein Beispiel verdeutlicht dazu die Verwendung. Über das webbasierte Dialogsystem *Safety-Pattern Instanziierungs-System (SAPIS)* hat der Entwickler Zugang zu den Entwurfsmustern sowie zur automatischen Auswahlhilfe.

In der Industrie wird von der Firma BTC Embedded Systems AG[22] ein Werkzeug (BTC Embedded *Validator*) entwickelt, mit dem dSPACE TargetLink[23]-Modelle durch Model-Checking formal verifiziert werden können. Dazu werden Anforderungen auf textueller Basis mit Hilfe des BTC Embedded*Specifiers* formalisiert [54]. Auch hierbei helfen Entwurfsmuster aus einer vordefinierten Bibliothek [24] dem Entwickler bei seiner Arbeit. Dialog-basiert wird der Entwickler bei der Instanziierung des passenden Entwurfsmusters unterstützt. Zur besseren Verdeutlichung beinhaltet jedes Muster eine Darstellung als Automat, der auf Zustandsautomaten nach Büchi basiert. Weiterhin wird das Muster in natürlicher Sprache beschrieben sowie der zeitliche Verlauf auf einer Achse dargestellt. Eine formale textuelle Darstellung ist nicht vorgesehen. Die Integration in die Werkzeugkette des BTC Embedded *Validators* erhöht den Grad der Automatisierung und reduziert damit mögliche Fehler durch unaufmerksame Bedienung durch den Entwickler.

Der Einsatz von Entwurfsmustern bei [21, 24] wirkt in beiden Fällen unterstützend für den Entwickler und spart dadurch Zeit und somit Kosten. Weiterhin werden Fehler beim formalen Spezifizieren durch Wiederverwenden von erprobten und damit ausgereiften Lösungen reduziert. Dem gegenüber steht die Einschränkung der Möglichkeiten beim Spezifizieren auf die vordefinierten Muster. Speziell bei komplexen Anforderungen steht ggf. kein passendes Muster zur Verfügung oder es konnte nicht gefunden werden.

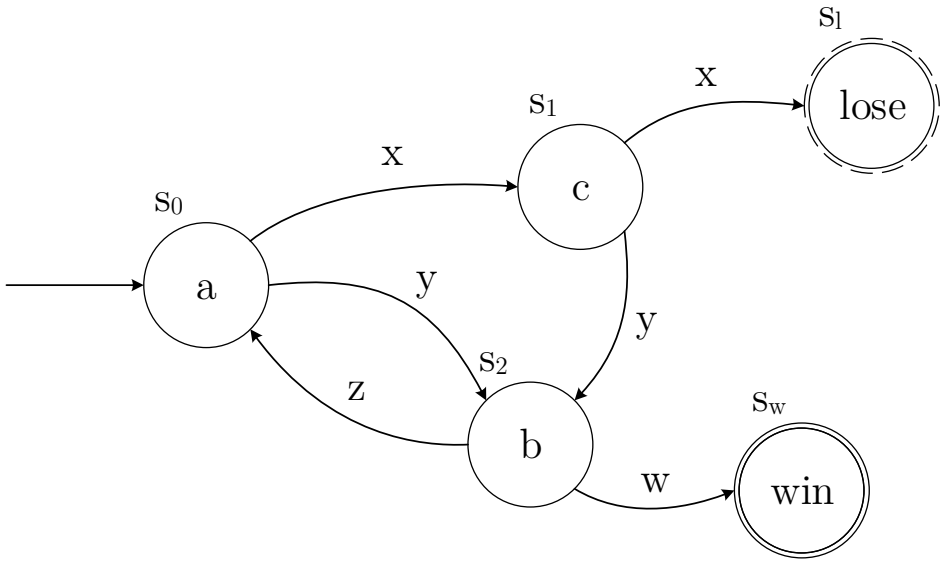
Abschließend wird festgehalten, dass die Darstellung derselben Spezifikation in verschiedenen Formen dem Verständnis dieser zugutekommt sowie dem Austausch und der Akzeptanz unter Entwicklern dienlich ist. Ohne die formale Darstellung kann jedoch keine formale Verifikation erfolgen. Entwurfsmuster unterstützen die Arbeit, stoßen bei komplexen Anforderungen aber an ihre Grenzen. Die Darstellung als Zustands-Automat scheint industriell bewährt zu sein [24, 109]. Eine direkte Möglichkeit zur Eingabe dieser Automaten als Spezifikation ist jedoch durch BTC Embedded*Specifier* nicht möglich. Daher wird im Folgenden eine weitere Möglichkeit beschrieben, mittels Automaten Anforderungen formal zu spezifizieren.

[22] `http://www.btc-es.de`
[23] `http://www.dspace.de`

4.3.2 Safety-Automaten

Neben der Möglichkeit, Anforderungen durch *CTL* zu spezifizieren, können diese in ARCADE auch mittels *Safety-Automaten (SA)* [19] formalisiert werden. Die Grundidee dabei ist, dem Entwickler eine intuitive Möglichkeit zur formalen Spezifikation zu bieten. Dies soll durch die Darstellung der Anforderungen als Automaten erreicht werden. Wenn man von einem Berechnungsbaum als Modell ausgeht, liegt eine ähnliche Darstellung zur Spezifikation nahe. Der Simulator baut, wie auch bei der Verwendung von *CTL*, den Zustandsraum auf. Daraufhin prüft ARCADE, ob sämtliche Pfade im Transitionssystem durch den *SA* abgedeckt sind. Ein *SA* ist definiert als $SA = (S, s_0, R, L, E)$. Dabei gilt:

- S: Endliche Menge an Zuständen.
- $s_0 \in S$: Initialer Zustand.
- $R \subseteq S \times S$: Transitionsrelation, die den Übergang von einem Zustand $s_1 \subseteq S$ zu einem Zustand $s_2 \subseteq S$ abhängig von einer Bedingung beschreibt.
- $L : S \to 2^{AP}$: Eine Funktion, die alle Zustände auf eine Menge von APs abbildet.
- $E \subseteq S$: Endzustand, kann *gewonnen* (win) als auch *verloren* (lose) sein. Ist optional.



s_0: Startzustand
$s_1, s_2 \in S$
$a, b, c, w, x, y, z \in AP$
s_l: verloren Zustand
s_w: gewonnen Zustand

Abbildung 4.4: Beispielhafter *SA*

Abbildung 4.4 zeigt beispielhaft einen *SA*. Dabei kann jeder Zustand eine atomare Eigenschaft enthalten. Gilt diese, ist der Zustand gültig, gilt diese nicht, wird der Zustand ungültig und muss über eine seiner Transitionen verlassen werden. Zustände müssen ebenfalls verlassen werden, wenn die Bedingung einer Transition erfüllt ist. Beim Verlassen eines Zustandes können mehrere Transitionen genommen werden. Daraufhin verweilt der Automat in mehreren gültigen Zuständen.

Die Anforderung, die durch den *SA* modelliert ist, gilt als erfüllt, wenn entweder der Endzustand *gewonnen* erreicht wurde oder alle Pfade im Zustandsraum durchlaufen wurden und sich der Automat danach mindestens in einem gültigen Zustand befindet. Ist kein Zustand im Automaten gültig, so wird das Durchlaufen des Zustandsraumes abgebrochen, und die Anforderung gilt als nicht erfüllt. Ebenso gilt das Erreichen des Endzustandes *verloren* als Nicht-Erfüllen der Anforderung.

4.4 Evaluierung: SA vs. CTL

In diesem Abschnitt der Arbeit geht es darum, zu evaluieren, welche Vorteile die grafische Methode mittels *SA* zur formalen Spezifikation im Gegensatz zur textuellen durch *CTL* bietet. Dazu wird ein nicht-repräsentativer Vergleich der beiden Methodiken unter Software-Entwicklern für eingebettete Systeme durchgeführt.

Im Vordergrund des Vergleichs stehen folgende Fragestellungen:

- Können Entwickler auch ohne entsprechendes Vorwissen Anforderungen formal spezifizieren, oder müssen immer Experten hinzugezogen werden?
- Erfolgt das formale Spezifizieren mit weniger Fehlern durch *CTL* oder durch *SA*?
- Welche Aspekte können den Prozess der formalen Spezifikation unterstützen?
- Hilft der logisch-formale Aufbau von *CTL* bei der Formulierung mehr als der grafische Aufbau der *SA*?
- Ist der akademische Ansatz, mittels *CTL* Anforderungen zu formalisieren, auch im industriellen Umfeld realisierbar?

Um diese Fragen zu beantworten, wurde eine Umfrage erstellt und diese unter Entwicklern mit und ohne Vorwissen bzgl. formaler Spezifikation durchgeführt. Die Teilnehmer stammen sowohl aus dem akademischen als auch aus dem industriellen Umfeld.

4.4.1 Aufbau der Umfrage

Die Teilnehmer erhielten zum Einstieg eine schriftliche Einführung in die Thematik (siehe A). Dazu gab es Beschreibungen zu atomaren Eigenschaften, zur temporalen Logik *CTL*

sowie zu Safety-Automaten *SA*. Allen Beschreibungen folgten kurze Beispiele, um das Verständnis zu dem entsprechenden Thema zu vertiefen.

Danach folgte eine Selbsteinschätzung des generellen Wissensstandes der Teilnehmer zur formalen Spezifikation von Anforderungen in der Software-Entwicklung. Dabei sollte jeder Teilnehmer sein Vorwissen zu acht Themengebieten auf einer Skala von 1 (schlecht) bis 6 (sehr gut) bewerten. Falls dem Teilnehmer keine Aussage dazu möglich war, wurde die Frage mit 0 (keine Aussage) gewertet.

Im Hauptteil der Umfrage ging es um die Entwicklung von sechs formalen Spezifikationen auf Basis von umgangssprachlich formulierten Anforderungen, die typischerweise bei der Entwicklung von Software speziell für Mikrokontroller auftreten. Dabei sollten die Teilnehmer diese Anforderungen sowohl mit Hilfe von *CTL* als auch mit Hilfe von *SA* aufstellen. Die Bewertung erfolgte hierbei mit *richtig*, *falsch* oder *keine Aussage*, wobei hauptsächlich das grundsätzliche Verständnis der beiden Methodiken zur Spezifikation untersucht wurde. Kleinere Unstimmigkeiten und Fehler wurden dabei nicht berücksichtigt.

Die Aufgaben 1 bis 4 wurden so gewählt, dass ihre Bearbeitung sowohl durch *CTL* als auch mittels *SA* dem gleichen Schwierigkeitsgrad entspricht. Aufgabe 5 wurde so gestellt, dass diese einfacher durch den Einsatz von *SA* gelöst werden konnte, da die Lösung eine Automaten-ähnliche Struktur aufweist. Konträr dazu ist Aufgabe 6 nur mittels *CTL* lösbar, da dort die Lösung die Spezifikation einer Liveness-Eigenschaft erfordert, welche nicht durch *SA* ausgedrückt werden kann.

Abschließend sollten die Teilnehmer bewerten, wie hilfreich verschiedene Aspekte der beiden Möglichkeiten zur formalen Spezifikation bei der Lösung der Aufgaben waren. Hierzu zählte das Vorwissen der Teilnehmer, die einleitenden Beschreibungen, der formale Aufbau, die visuelle Darstellungsmöglichkeit sowie die intuitive Nutzbarkeit. Alle Fragen sollten mit derselben Skala (1..6, 0) wie in der anfänglichen Selbsteinschätzung beantwortet werden.

4.4.2 Auswertung der Ergebnisse

Es nahmen insgesamt acht Entwickler an der Umfrage teil. Die Teilnehmer entwickeln im Durchschnitt seit acht Jahren Software. Der Großteil (fünf) ist nicht geübt darin, Anforderungen formal zu spezifizieren, wohingegen drei Teilnehmer sich eher gut damit auskennen (siehe Abbildung 4.5).

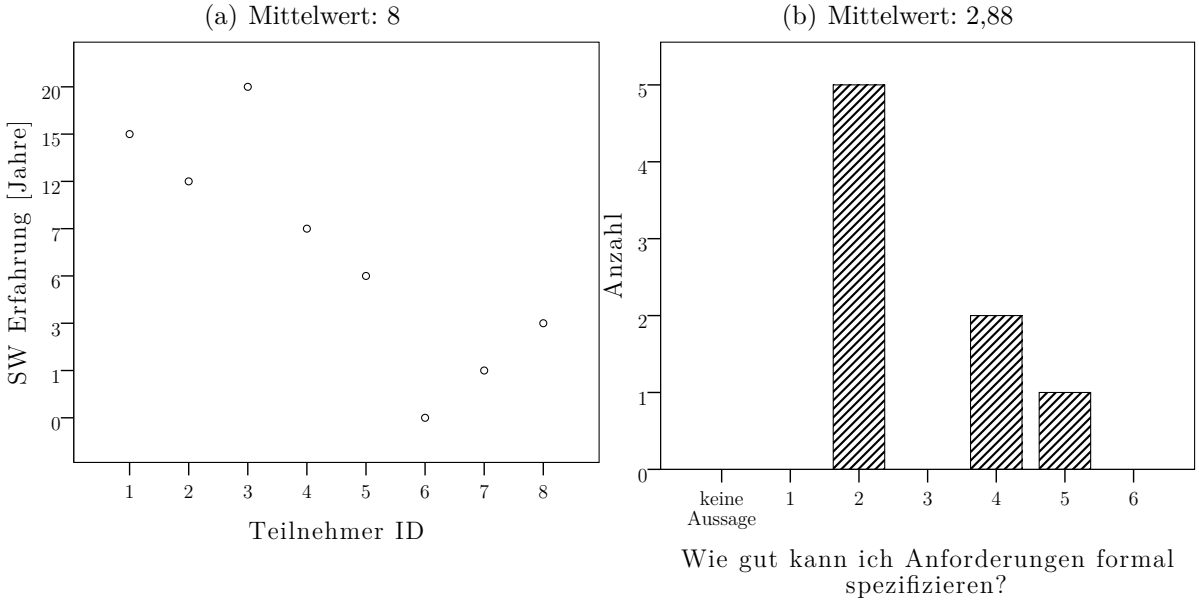


Abbildung 4.5: Wie sah die Gruppe der Teilnehmer aus?

Betrachtet man im Speziellen das Vorwissen in Bezug auf *CTL* bzw. Automaten im Allgemeinen, so wird das Wissen bzgl. Automaten (Mittelwert 3,88) größer eingeschätzt als das über *CTL* (Mittelwert 2,63) (siehe Abbildung 4.6).

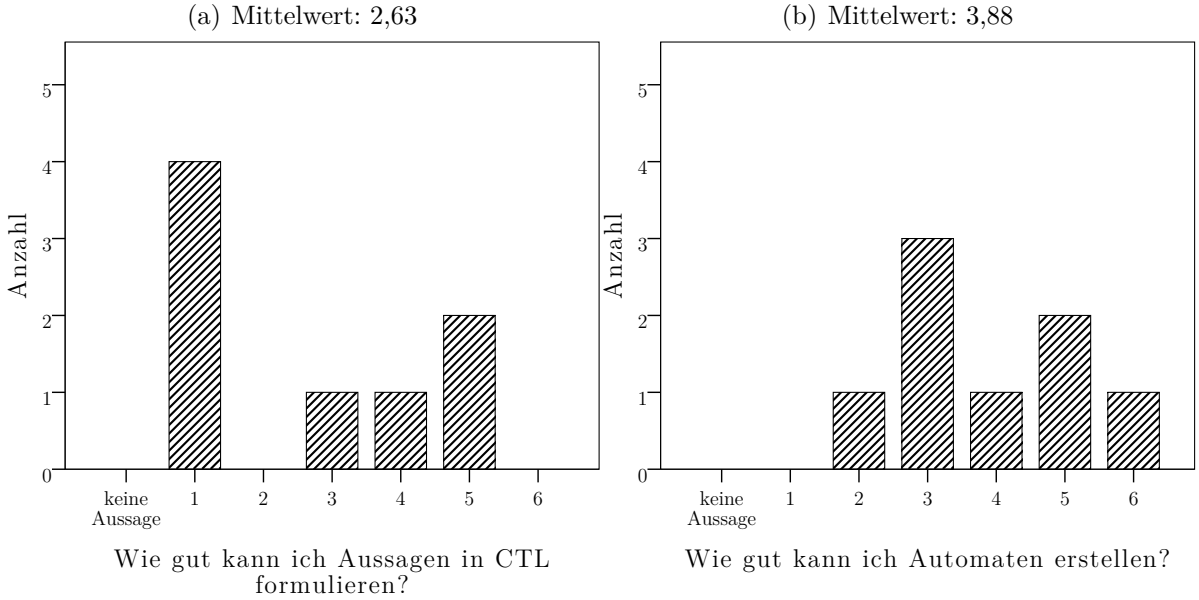


Abbildung 4.6: Wie schätzen die Teilnehmer ihr Vorwissen hinsichtlich SA/CTL ein?

Abbildung 4.7 zeigt die Ergebnisse der Bearbeitung der sechs Aufgaben.

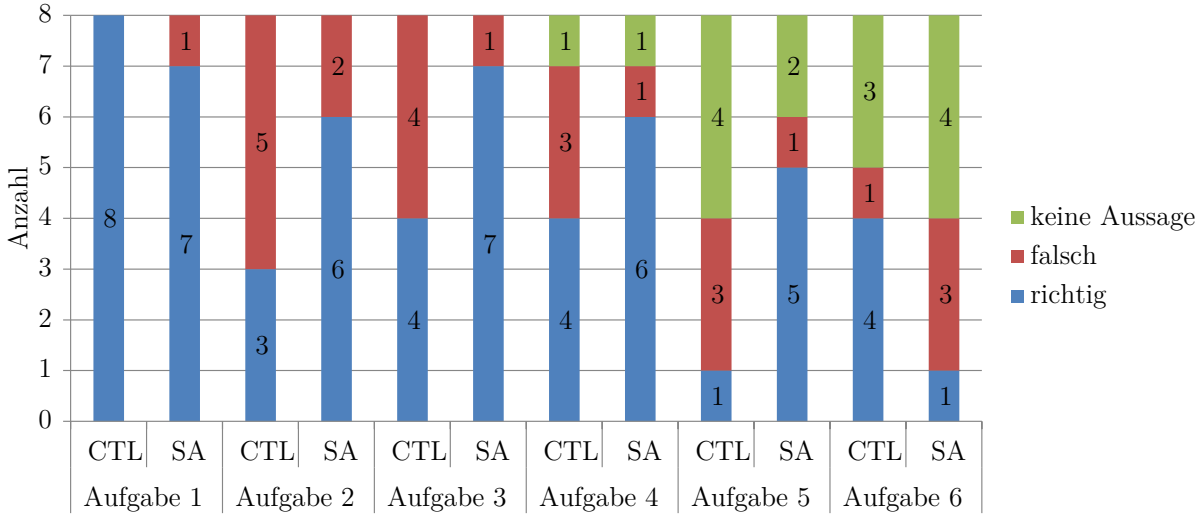


Abbildung 4.7: Auswertung

Aufgabe 1:

Es soll der Wertebereich einer Variablen global überprüft werden. Diese Anforderung muss immer gelten.

CTL: Alle Lösungen waren korrekt.

SA: Fast alle Lösungen waren korrekt.

Aufgabe 2:

Es soll verifiziert werden, dass eine Funktion erreicht und durchlaufen werden kann.

CTL: Oft wurde EF mit AF vertauscht, so dass fälschlicherweise ein kann zu einem muss wird. Es wurde aber nach einem kann gefragt.

SA: Fast alle Lösungen waren korrekt.

Aufgabe 3:

Es soll geprüft werden, ob während der Ausführung einer Funktion ein Hardware-Register gesetzt wird. Hier besteht die Schwierigkeit in der korrekten Verknüpfung von zwei Eigenschaften. Entweder lokalisiert man den einen Programmpfad, in dem das Register fälschlicherweise gesetzt wird, oder man spezifiziert global, dass bei Erreichen der Programm-Position das Register nicht gesetzt sein darf.

CTL: Es wurde häufig der Existenzquantor mit dem Allquantor verwechselt.

SA: Fast alle Lösungen waren korrekt.

Aufgabe 4:

Es soll verifiziert werden, dass während der Initialisierungsphase des Systems die Interrupts deaktiviert bleiben. Die Lösung ist ähnlich zu Aufgabe 3.

CTL: Die Lösungen zeigen die gleichen Probleme wie in Aufgabe 3.

SA: Fast alle Lösungen waren korrekt.

Aufgabe 5:

Es soll geprüft werden, ob sich das System wie die gezeigte Zustandsmaschine verhält.

CTL: Die Lösungen werden schnell unübersichtlich, da viele Zustands-Übergangs-Bedingungen aufgestellt werden müssen. Textuell sind diese nur schwierig zu gruppieren. Fehler treten meist direkt im Ansatz auf. Die Aufgabe konnte nur von einem Teilnehmer korrekt gelöst werden.

SA: Die Anzahl der Bedingungen bleibt gleich im Vergleich zu der Lösung durch *CTL*. Visuell können diese hingegen besser durch die Unterscheidung von Transition und Zustand gruppiert werden. Die Aufgabe konnte von den meisten korrekt gelöst werden.

Aufgabe 6:

Es soll verifiziert werden, dass eine bestimmte Funktion im System immer wieder aufgerufen wird. Da die Anforderung eine Liveness Eigenschaft beschreibt und *SA* diese Eigenschaften nicht beschreiben kann, besteht die erhöhte Schwierigkeit darin, diesen Sachverhalt korrekt zu erkennen.

CTL: Mit Vorwissen bzgl. *CTL* konnte diese Aufgabe durch ein Standard Schema gelöst werden. Dies wurde von der Hälfte der Teilnehmer korrekt gelöst.

SA: Nur ein Teilnehmer erkannte korrekt, dass diese Aufgabe nicht durch *SA* lösbar ist.

Aufgaben 5 und 6 zeigen fast konträre Ergebnisse. Die Aufgaben scheinen mit der entsprechenden Methode intuitiv lösbar zu sein. Mit der jeweils anderen Methode allerdings konnten diese nur von einzelnen Teilnehmern gelöst werden. Diese Teilnehmer konnten ggf. passendes Vorwissen anwenden.

Subjektiv scheint der Großteil der Teilnehmer eher schlecht mit der Formulierung durch *SA* zurechtgekommen zu sein. Die Selbsteinschätzung der Teilnehmer deutet darauf hin, dass die Spezifikation durch *CTL* einfacher gewesen ist (siehe Abbildung 4.8). Den Teilnehmern half jedoch der logisch-formale Aufbau von *CTL* in gleicher Weise wie der grafische Aufbau von *SA* bei der Lösung der Aufgaben (Abbildung 4.9).

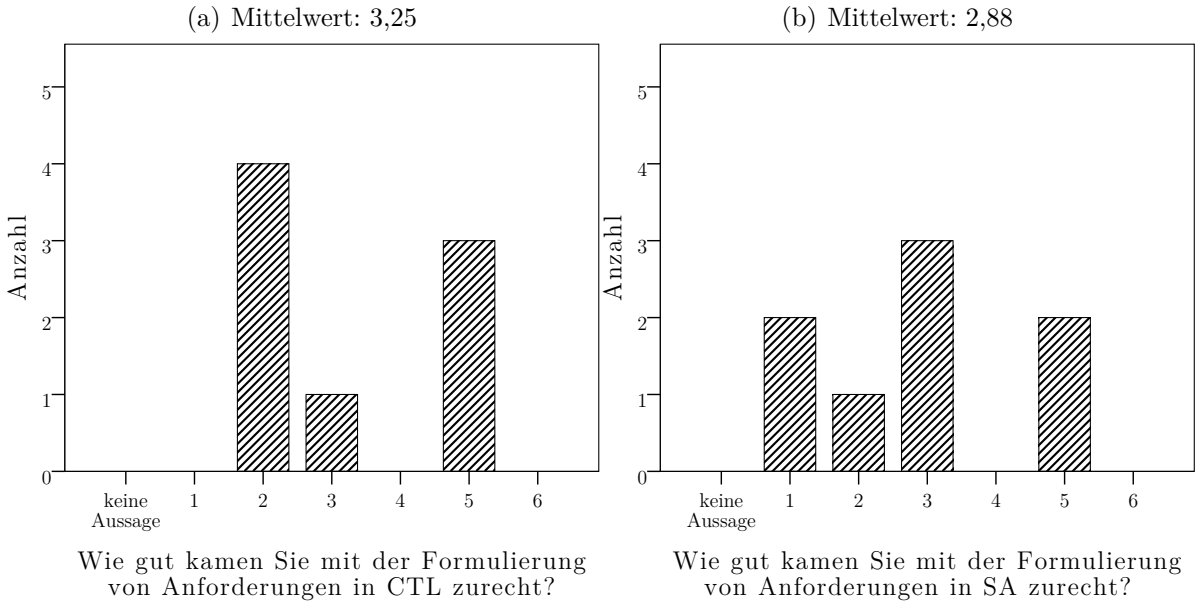


Abbildung 4.8: Wie gut kamen Sie mit der Formulierung von Anforderungen durch CTL/SA zurecht?

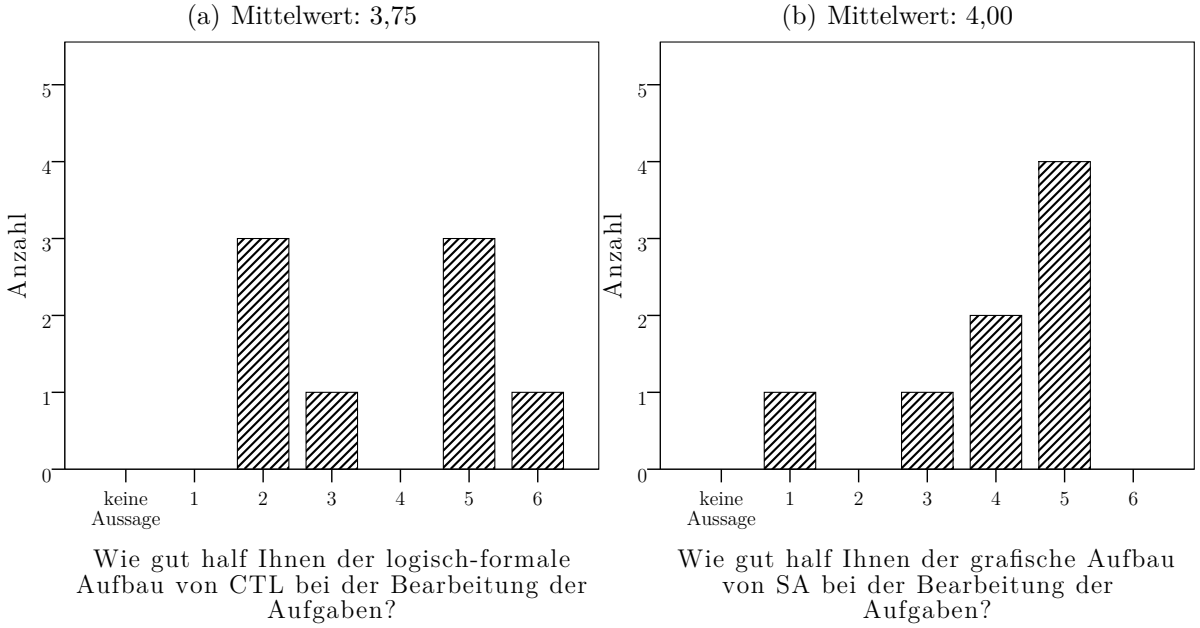


Abbildung 4.9: Bewertung des logisch-formalen Aufbaus von CTL im Vergleich zum grafischen Aufbau von SA.

Bei der Frage nach der intuitiven, selbsterklärenden Anwendbarkeit der beiden Me-

thodiken bewerteten die Teilnehmer die Spezifikation mittels *SA* als eher intuitiv im Vergleich zu *CTL* (siehe Abbildung 4.10).

Abschließend wurden die Teilnehmer gefragt, ob die einleitenden Beschreibungen bzgl. *CTL* und *SA* ausreichend gewesen seien (siehe Abbildung 4.11). Mehr als die Hälfte sah die Beschreibungen als eher schlecht an. In den möglichen Freitextpassagen wurde zusätzlich von den Teilnehmern eine größere Anzahl an Fallbeispielen sowie eine detaillierte Beschreibung der beiden Methodiken gefordert.

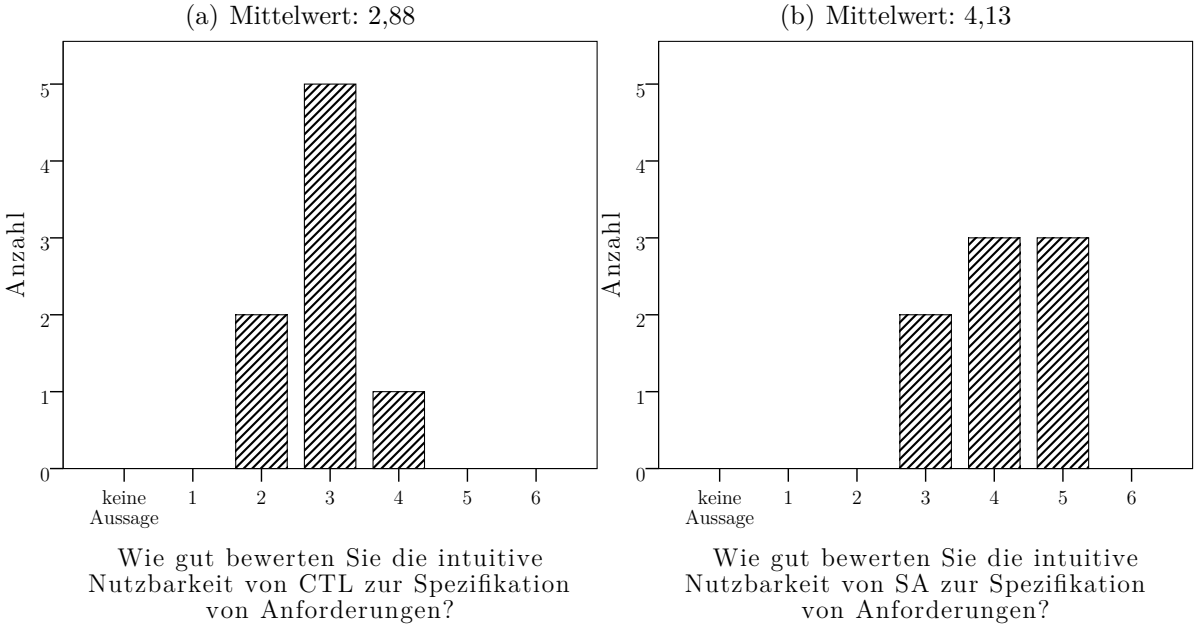


Abbildung 4.10: Bewertung der intuitiven Nutzbarkeit von CTL/SA zur Spezifikation von Anforderungen.

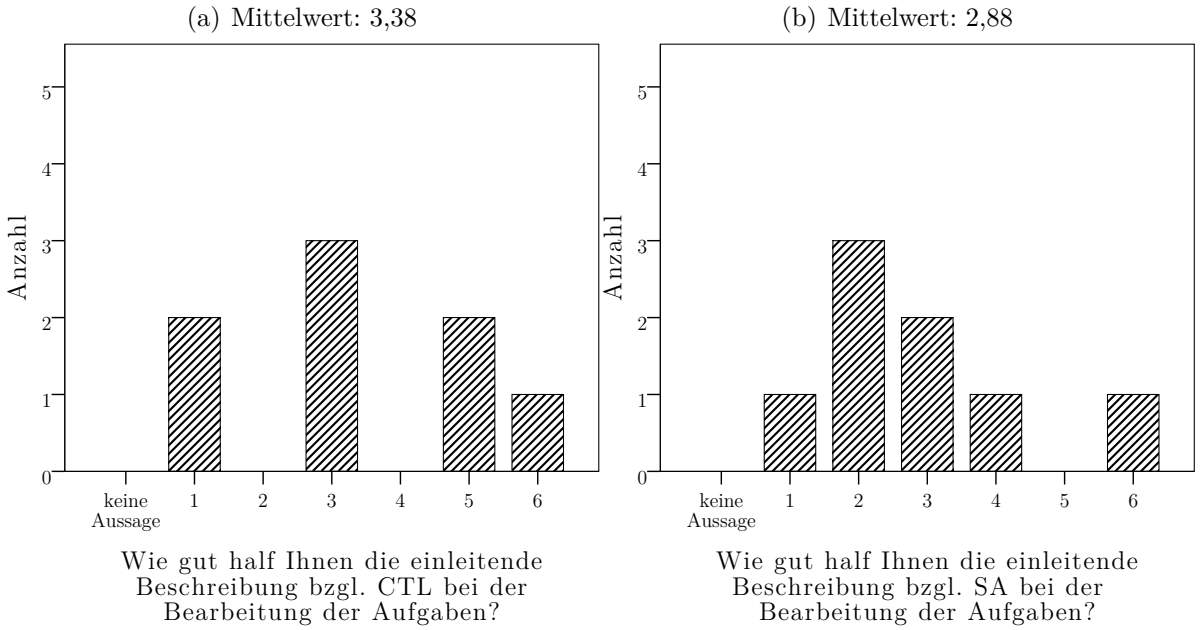


Abbildung 4.11: Ist eine Optimierung der einführenden Beschreibung notwendig?

4.4.3 Fazit

Da dies keine repräsentative Umfrage darstellt, sondern eine subjektive Einschätzung der Teilnehmergruppe ist, stellen die Ergebnisse keine allgemeingültigen Aussagen dar.

Obwohl die Teilnehmer ihr Vorwissen als gering einstuften (Abbildung 4.6) sowie die einleitenden Beschreibungen als nicht ausreichend (Abbildung 4.11), konnte etwas mehr als die Hälfte der Aufgaben korrekt gelöst werden (56 mal richtig von 96 Möglichkeiten, siehe Abbildung 4.7). Die Lösungen zu den Aufgaben (1-4) zeigen, dass mittels *SA* öfter korrekt formal spezifiziert werden konnte.

Mehr als die Hälfte der Teilnehmer konnte trotz schlechter Beschreibung die Aufgaben mittels *SA* intuitiver (Abbildung 4.10 und Abbildung 4.11) lösen.

Wieso die Formulierung mittels *CTL* (Abbildung 4.8) einfacher gewesen zu sein scheint, obwohl die Teilnehmer sowohl die Vorteile von *CTL* als auch von *SA* gleich hilfreich einschätzten (Abbildung 4.9), bleibt ungeklärt.

Im Folgenden werden die zu Beginn gestellten Fragen beantwortet:

- *Können Entwickler auch ohne entsprechendes Vorwissen Anforderungen formal spezifizieren, oder müssen immer Experten hinzugezogen werden?*
  Teilnehmer auch ohne Vorwissen konnten die simplen Aufgaben (1-4) lösen. Die komplexeren Aufgaben (5, 6) benötigten tiefergehendes Wissen und waren mit Vorwissen einfacher lösbar.

- *Erfolgt das formale Spezifizieren mit weniger Fehlern durch CTL oder durch SA?*
  Im Teilnehmerfeld konnten 32 mal Aufgaben mittels *SA* korrekt formal spezifiziert werden, durch *CTL* hingegen nur 24 mal.

- *Welche Aspekte können den Prozess der formalen Spezifikation unterstützen?*
  Durch den grafischen Ansatz bei *SA* können schnell auch ungeübten Entwicklern Anforderungen klar visualisiert und beschrieben werden. Die grafischen Mittel fördern die Wahrnehmung speziell durch die Möglichkeit zur visuellen Trennung bzw. Gruppierung von Systemeigenschaften. Bei *CTL* geschieht dies durch das Setzen von Klammern. Dies kann bei der Darstellung in einem Textfeld-Steuerelement schnell unübersichtlich werden.

- *Hilft der logisch-formale Aufbau von CTL bei der Formulierung mehr als der grafische Aufbau der SA?*
  Der logisch-formale Aufbau von *CTL* scheint den Teilnehmern in gleicher Weise bei der Bearbeitung der Aufgaben geholfen zu haben wie der grafische Aufbau von *SA* (siehe Abbildung 4.9).

- *Ist der akademische Ansatz, mittels CTL Anforderungen zu formalisieren, auch im industriellen Umfeld realisierbar?*
  Bei *simplen* Anforderungen sind CTL-Formeln und SA gleich *einfach* zu erstellen. Werden die Anforderungen komplexer, so muss der Entwickler geschult bzw. geübt sein im Umgang mit logischen Formeln.

Beide Möglichkeiten zur formalen Spezifikation haben ihre Daseinsberechtigung: *CTL* ist ausdrucksstärker, dadurch allerdings komplexer in der Erstellung. *SA* ist intuitiver aber eingeschränkter in dem, was ausgedrückt werden kann. Die durchgeführte Umfrage zeigte, dass auch ungeschulte Teilnehmer durch eine entsprechende Einführung in beide Techniken diese erlernen und korrekt anwenden können. Bei speziellen Anforderungen sollte ein Experte hinzugezogen werden.

Die grafische Modellierung von Anforderungen ist ein sinnvoller Ansatz, um den Einstieg bzw. den Umgang mit formaler Spezifikation zu erleichtern, wie auch in [19] festgestellt wird. *SA* ist nicht so ausdrucksstark im Vergleich zu *CTL*, für simple Probleme aber effektiver und anschaulicher.

Ein Vorteil der grafischen Eingabe von Anforderungen liegt in der visuellen Gruppierung von Bedingungen. Würde man dies auf die textuelle Darstellung von logischen Formeln übertragen, könnten Formeln kognitiv schneller erfassbar werden.

4.5 Beispiele von formalisierten Anforderungen für Arcade

Zur Formalisierung von Anforderungen in *CTL* für ARCADE werden vorweg bestehende Fallbeispiele analysiert, um bei der Erstellung eigener *CTL* Formeln darauf aufzubauen. Dazu werden die in den Arbeiten [92, 101, 104] verwendeten Eigenschaften sowie deren Umsetzung in Spezifikationsformeln betrachtet. Alle Fallbeispiele beziehen sich dabei auf Anwendungen von eingebetteten Systemen. [104] untersucht eine Anwendung aus dem automotiven Umfeld, [92] eine industriell verwendete Strickmaschine und [101] Beispiele aus dem akademischen Umfeld.

Im Vergleich zu PC-basierten Software-Systemen liegt der Fokus bei eingebetteten Systemen zum einen auf dem Einhalten von Echtzeitanforderungen und zum anderen auf dem Zusammenspiel der Hardware mit der integrierten Peripherie wie z. B. digitale Ein- bzw. Ausgänge, Zähler sowie Speicher. Die Verifikation der Echtzeitanforderungen wurde bereits in Kapitel 3 behandelt, so dass im Folgenden speziell die Anforderungen an die Interaktion mit der Hardware im Vordergrund stehen.

Akademisches Beispiel aus [101]

In diesem Fallbeispiel wurden zwei simple Anwendungen untersucht. Die erste ist die Implementierung eines Lichtschalters mit der Möglichkeit, verschiedene Intensitätsstufen des Lichtes einzustellen. Die zweite Anwendung ist die Steuerung einer fiktiven Chemie-Fabrik. Zu jeder Anwendung soll dabei eine Eigenschaft verifiziert werden.

Im Falle des Lichtschalters soll überprüft werden, ob ein digitaler Port auf einen bestimmten Wert gesetzt wird. Dies geschieht mittels der Formel **EF** porta = 0x55, welche überprüft, ob `porta` irgendwann einmal auf den Wert `0x55` gesetzt wird. Darauf aufbauend soll mittels der angepassten Formel **AF** porta = 0x55 überprüft werden, ob dieser Port unter jeden Umständen irgendwann auf diesen Wert gesetzt wird. Nachdem damit ermittelt werden konnte, dass im System ein korrekter Pfad existiert, bei dem der Port niemals auf den definierten Wert gesetzt wird, musste die Formel in Abhängigkeit einer zusätzlichen Variable gesetzt werden. Dies führte zur folgenden Formel:

$$\begin{aligned}&\mathbf{AG}\ ((\text{porta} = \text{0xFF}\ \wedge\ \text{ButtonPressed} = 1) \Rightarrow\\ &\mathbf{A}\ \text{porta} = \text{0xFF}\ \mathbf{U}\ \text{porta} = \text{0x55})\end{aligned}$$

Im Beispiel für die Steuerung der Fabrik soll überprüft werden, ob `porta` je auf einen speziellen Wert gesetzt wird, und wenn dem so ist, ob dieser Zustand dann auch für immer beibehalten wird. Dies ist z. B. für die Realisierung einer Notfallabschaltung notwendig. Dazu wurde folgende Formel erstellt: **EF** porta = 0x20 $\wedge$ **AG** porta = 0x20 $\Rightarrow$ **AG** porta = 0x20. Hierbei wurde, wie im vorangegangenen Beispiel auch, die Implikation ($\Rightarrow$) als „wenn dann“ Operator verwendet. Wenn porta = 0x20 gilt, dann muss überall immer porta = 0x20 gelten. Dabei ist zu beachten, dass die Implikation immer

nur mit den Operatoren **AG** verwendet wird. Bei der Verwendung der Implikation mit **AF** oder **EF** könnte die Formel direkt schon im ersten Zustand fälschlicherweise wahr sein.

Automotives Beispiel aus [104]

Dieses Fallbeispiel untersucht die Anwendbarkeit von ARCADE auf Software, die nicht speziell für die Verifikation mittels Model-Checking angepasst wurde. Dabei greifen die Autoren auf studentische Arbeiten, die während einer Lehrveranstaltung entstanden sind, zurück. Drei ausgewählte Programme werden daraufhin analysiert, ob diese einen Stack-Überlauf haben, ein Ausgabewert immer in einem definierten Intervall liegt und ob eine Variable einen speziellen Wert annimmt, wenn eine andere Variable einen gewissen Wert überschreitet.

Da ARCADE automatisch eine maximale Stack-Größe bestimmt, kann damit manuell auf einen Überlauf des Stack geprüft werden. Mittels **AG** (value $>$ 0 $\wedge$ value $<$ 251) wird die Einhaltung des Wertebereichs der Variablen `value` überprüft. Die Formel **AG** (frequency $>$ 5733 $\Rightarrow$ **AF** velocity = 251) beschreibt die Eigenschaft, dass wenn `frequency` größer als 5733 ist, `velocity` irgendwann auf 251 gesetzt werden soll.

Industrielles Beispiel aus [92]

Bei dieser Anwendung handelt es sich um eine Steuerung für eine industrielle Strickmaschine, die auf einem Intel Mikrokontroller basiert. Die Anwendung nutzt dabei mehrere Peripherie-Komponenten der Hardware zur Interaktion mit dem System und zur Kommunikation mit der Außenwelt.

Die erste Formel wird dazu verwendet, um die korrekte Reihenfolge der Initialisierung einer zwei Byte großen Variablen `Revol` zu prüfen. Zuerst soll das höherwertige Byte, danach das niederwertige Byte im Speicher beschrieben werden. Dieser Schreibvorgang soll überprüft werden, wenn der Startvorgang noch nicht abgeschlossen ist und `startUpCode` noch auf 0 steht. Folgende Formel verwendet dazu die Implikation gekoppelt mit dem Until-Operator (**A...U...**), um diese Reihenfolge zu beschrieben:

AG (startUpCode=0 $\wedge$ Revol=0x0000 $\Rightarrow$ **A** Revol=0x0000 **U** Revol=0xFF00)
$\wedge$ **AG** (startUpCode=0 $\wedge$ Revol=0xFF00 $\Rightarrow$ **A** Revol=0xFF00
U Revol=0xFFFF) $\wedge$ **EF** Revol=0x0000 $\wedge$ **EF** Revol=0xFF00

Die nächste Formel **AF** (Buffer0=0x42 $\wedge$ startUpCodeFinished=1) prüft die korrekte Initialisierung einer Speicherstelle an einem speziellen Punkt im Programm (`startUpCodeFinished` = 1). Mittels **AG** (**EF** mark=MARK_SLEEP) kann überprüft werden, ob eine definierte Markierung im Programmablauf auf jedem Pfad erreicht werden kann. Die Markierung wurde explizit zum Zwecke der Vereinfachung bei der Formulierung der Anforderungen im Quell-Code hinzugefügt. Die Position im Programm könnte alternativ auch durch Programmzeiger definiert werden. Die Formel

AG (**AF** mark=MARK_SENDRPM) prüft die Eigenschaft des Systems, dass wenn eine Nachricht über einen Kommunikationskanal eingeht, jedes Mal korrekt auf diese geantwortet wird. Um das Einhalten der korrekten Reihenfolge beim Durchlaufen einer Zustandsmaschine zu prüfen, wurde folgende Formel aufgestellt:

$$\begin{aligned}
&\mathbf{AG}\ (\text{State=0}\ \Rightarrow \mathbf{A}\ \text{State=0}\ \mathbf{U}\ \text{State=1} \mid \text{State=0}) \wedge \\
&\mathbf{AG}\ (\text{State=1}\ \Rightarrow \mathbf{A}\ \text{State=1}\ \mathbf{U}\ \text{State=0} \mid \text{State=2} \mid \text{State=1}) \wedge \\
&\mathbf{AG}\ (\text{State=2}\ \Rightarrow \mathbf{A}\ \text{State=2}\ \mathbf{U}\ \text{State=0} \mid \text{State=2}) \wedge \\
&\mathbf{AF}\ \text{State=0}
\end{aligned}$$

Hierbei sind alle möglichen Zustandsübergänge mit Hilfe der Implikation einzeln dargestellt und mit dem UND-Operator verbunden. Dabei ist zu beachten, dass die Zustandsmaschine auch in dem aktuellen Zustand verweilen kann.

4.6 Evaluierung an einem Fallbeispiel

Im folgenden Kapitel wird an einem industriellen Fallbeispiel evaluiert, ob formale Verifikation auch in kleinen Software-Projekten im Umfeld eines *KMU* durchgeführt werden kann. Dazu wird das Werkzeug, welches im vorangegangenen Kapitel ausgewählt wurde, zur Modell-Prüfung sowie *CTL* Formeln und *SA* zur Darstellung von Anforderungen eingesetzt. Als Anwendung wird die in Abschnitt 2.8 beschriebene Software eines eingebetteten Systems verwendet. Neben direkten Speicher- und Registerzugriffen bietet dieses Beispiel auch Interrupts sowie die Verwendung von peripheren Hardware-Modulen.

4.6.1 Durchführung

Durchgeführt wurde diese Evaluierung mit der GitHub-Version `0xB53E34B` von ARCADE.μC. Alle Einstellungen bzgl. des Model-Checkings sind auf den Standard-Einstellungen belassen worden. Es wurde ein PC mit i5 QuadCore und 8GB RAM verwendet.

Neben der ursprünglichen Version der Anwendung wurden Versionen der Software untersucht, in denen einzelne Fehler behoben worden sind. Die Fehler werden in Unterabschnitt 4.6.9 näher beschrieben. Tabelle 4.1 stellt den Zusammenhang von Fehler und Software-Version dar. Folgende Versionen der Watchdog-Anwendung wurden verwendet:

- **.Error**: Dies ist die unveränderte Version der Anwendung inklusive aller vorhandenen Fehler.
- **.fixed1**: Alle Fehler bis auf Fehler 4 wurden hierbei behoben.
- **.fixed2**: In dieser Version wurden alle der fünf Fehler behoben.

Version	Fehler 1	Fehler 2	Fehler 3	Fehler 4	Fehler 5
.Error	x	x	x	x	x
.fixed1	o	o	o	x	o
.fixed2	o	o	o	o	o
.fixed3	o	x	o	x	o
.fixed4	o	x	o	o	o

Tabelle 4.1: Versionen der untersuchten Software und die beinhalteten Fehler. (x=vorhanden, o=behoben)

- **.fixed3**: Fehler 1, 3 und 5 wurden in dieser Version behoben.
- **.fixed4**: Alle Fehler bis auf Fehler 2 wurden hierbei behoben.

Als erstes wird jeweils die zu prüfende Anforderung umgangssprachlich beschrieben. Daraufhin wird eine dazu passende, generelle *CTL*-Formel sowie ein entsprechender *SA* aufgestellt und unter Umständen erweitert. Die Formeln und die Automaten werden jeweils detailliert beschrieben. In einem weiteren Abschnitt werden die Ergebnisse der aufgestellten formalen Anforderungen dargestellt und bewertet. Einige der formalisierten Anforderungen wurden durch die studentischen Arbeiten [89] und [118] unterstützt sowie in [RG11] und [GRK13] veröffentlicht. Tabelle 4.2 zeigt eine Übersicht über die Ergebnisse des Model-Checkings mit dem Bezug zu den verwendeten Anforderungen sowie den verifizierten Versionen der Software.

Hinweise zur Syntax

In den generellen Versionen der folgenden Formeln und Automaten werden häufig Platzhalter für Start- und End-Adressen von Funktionen verwendet. Diese werden in den konkreten Beispielen ersetzt durch die passenden Speicheradressen der jeweiligen Software-Version. „#Start_FunktionX“ steht beispielsweise für die Einsprungadresse von FunktionX und #Ende_FunktionX für die Adresse deren Endes. In diesem Kontext wird häufig die Abkürzung *Program Counter (PC)* verwendet. Der Programmzeiger zeigt auf die Speicheradresse, in der der aktuell ausgeführte Befehl steht.

4.6.2 Anforderung 1: Wertebereich von Variablen

Mit dieser Anforderung soll erreicht werden, dass Wertebereiche von Variablen und Hardware-Registern eingehalten werden.

Formel 4.2 ist genau dann wahr, wenn entlang aller Pfade im Zustandsraum für alle Zustände die Einschränkung für die Variable *VarX* gilt. Hierbei kann in der Formel *VarX*

Eigenschaft	CTL Formel	Safety Automat	Software Version .Error	.fixed1	.fixed2	.fixed3	.fixed4
4.6.2	CTL 4.5	SA B.1	valid				
			valid				
4.6.3	CTL 4.11	SA B.2	valid	invalid			
	CTL 4.12		invalid				
	CTL 4.13		invalid	valid			
			invalid	valid			
	CTL 4.14	SA B.4	valid				
			valid				
	CTL 4.15		valid				
4.6.4	CTL 4.18	SA B.5	valid			invalid	
			invalid			valid	
4.6.5	CTL 4.21	SA B.6	invalid		valid		
			invalid		valid		
	CTL 4.22	SA B.7				invalid	valid
						invalid	valid
4.6.6	CTL 4.24	SA B.8	valid				
			valid				

Tabelle 4.2: Auflistung der Ergebnisse

auch durch ein entsprechendes Hardware-Register ausgetauscht werden, um dieses zu überprüfen. Bezogen auf das ausgeführte Programm ist es dabei egal, was dies macht. Die Einschränkungen für den Wertebereich von *VarX* müssen immer gelten. Der Safety Automat in Abbildung 4.12 beschreibt das gleiche Verhalten.

$$\mathbf{AG}\ (\mathrm{VarX} > 20\ \wedge\ \mathrm{VarX} < 100) \tag{4.2}$$

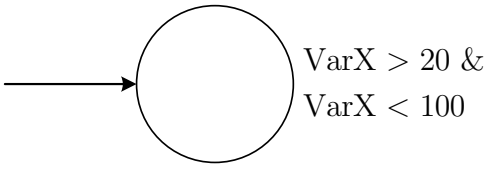


Abbildung 4.12: *SA* zur Überprüfung des Wertebereichs einer Variablen

Am Beispiel einer Geschwindigkeitsberechnung in Quelltext 4.1 soll verifiziert werden, dass die Variable `iFrequency` in der Funktion `SpeedCalc` nur Werte im Intervall [1,15000] annimmt, um Berechnungen in anderen Funktionen, die diesen Wert nutzen, abzusichern. Dazu wird folgende Formel genutzt (vgl. Beispiel aus [104]):

$$\mathbf{AG}\ (\mathrm{iFrequency} >= 1\ \wedge\ \mathrm{iFrequency} <= 15000) \tag{4.3}$$

```
0 UINT32 SpeedCalc (void)
{
2   UINT32 iFrequency = (1500000000) / TimerRegister_A;

4   /* Saturation */
    if (iFrequency > 15000) { iFrequency = 15000; }
6   if (iFrequency < 20)    { iFrequency = 1; }

8   return iFrequency;
}
```

Quelltext 4.1: Einschränkung des Wertebereichs

Die im Programm vorgesehene Eingrenzung des Wertebereichs (Zeile 5 und 6) soll sicherstellen, dass `iFrequency` nur gültige Werte annimmt. Ergibt die Berechnung in Codezeile 2 allerdings einen Wert größer als 15000, so würde am Ende der Funktion `iFrequency` auf den maximalen Wert von 15000 eingeschränkt sein. In Zeile 2 jedoch würde der Model-Checker einen Zustand finden, in dem Formel 4.3 ungültig ist und ein entsprechendes Gegenbeispiel dafür aufzeigen. Um dieses Verhalten zu vermeiden, wird Formel 4.3 um eine Einschränkung (vgl. Lichtschalter aus [101]) bezogen auf die Position im Programmcode erweitert. Dabei muss die Einschränkung erst gelten, wenn das Ende

der Funktion erreicht ist. Daraus resultiert Formel 4.4, welche das gleiche Verhalten wie der Safety Automat in Abbildung 4.13 beschreibt.

$$\begin{aligned}&\textbf{AG}\ (\text{PC} = \#\text{Ende\_SpeedCalc} \Rightarrow \\ &(\text{iFrequency} >= 1\ \wedge\ \text{iFrequency} <= 15000\ ))\end{aligned} \tag{4.4}$$

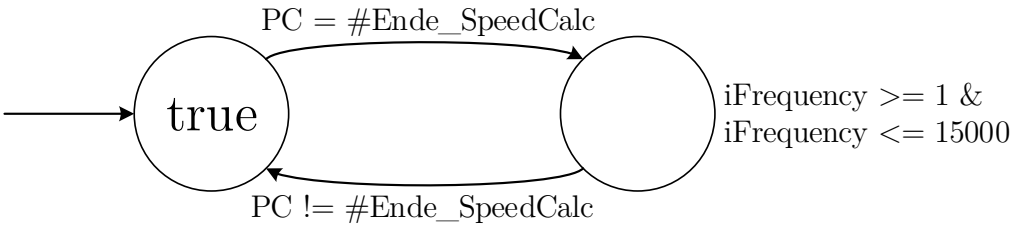


Abbildung 4.13: *SA* zur Wertebereichsprüfung in Abhängigkeit der Position im Programm

Ergebnis

Im Fallbeispiel soll verifiziert werden, dass die globale Variable `ulWatchdogCounter` vor Ausführung der Hauptfunktion `main()` korrekt auf den Wert 0 initialisiert worden ist. Dies prüft die korrekte Funktion der Speicherinitialisierung von initialisierten globalen Variablen. Aus der *CTL*-Formel 4.4 wurde Formel 4.5 sowie von dem Automaten 4.13 die Entsprechung Abbildung B.1 abgeleitet. Beide Anforderungen waren in **.Error** gültig.

$$\textbf{AG}\ (\text{PC} = \text{main} \Rightarrow \text{ulWatchdogCounter} = 0) \tag{4.5}$$

Als bei einem Wechsel auf eine neue Version der Entwicklungsumgebung die Platzierung der Speicherbereiche in den Einstellungen für den Linker fehlerhaft waren, wurde Fehler 6 (Abschnitt 4.6.9) mittels der oben gezeigten Formel entdeckt. Die Überprüfung des Variablenwertes nach dessen Initialisierung war dabei beim Erreichen von `main()` nicht korrekt und somit die Formel ungültig. Das Gegenbeispiel half zur Entdeckung der fehlerhaften Linker-Einstellungen.

4.6.3 Anforderung 2: Aufrufen von Funktionen

Diese Anforderung soll beschreiben, dass eine Funktion *Funktion_Y* im Programmablauf erreicht werden kann. Dazu wird zu Beginn folgende einfache Formel aufgestellt:

$$\textbf{EF}\ (\text{PC} = \#\text{Start\_Funktion\_Y}) \tag{4.6}$$

Formel 4.6 prüft, ob entlang mindestens eines Pfades mindestens ein Zustand existiert, in dem *PC* auf den Anfang von *Funktion_Y* weist. Falls dieser Zustand existiert, so ist es möglich, dass die angegebene Funktion erreichbar ist. ARCADE beweist dies durch einen möglichen Programmablauf als Zeugen. Dabei kann sich das Programm so verhalten, dass der *PC* irgendwann einmal gleich der Startadresse der Funktion ist. Unter Umständen wird diese Funktion im realen Ablauf jedoch nicht angesprungen. Um festzustellen, ob der resultierende Programmablauf auch im realen System vorkommen kann, muss der Entwickler diesen im Simulator anhand des Zeugen nachvollziehen.

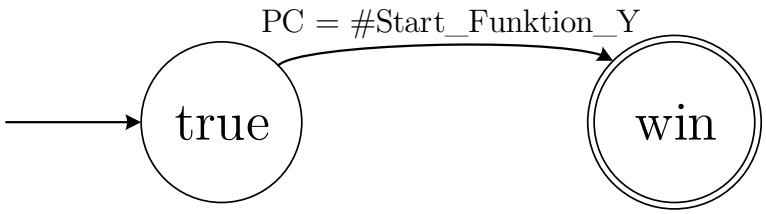


Abbildung 4.14: *SA* zur Überprüfung der Erreichbarkeit einer Funktion

Ein möglicher Safety Automat zur Überprüfung, dass eine Funktion erreicht werden kann, ist in Abbildung 4.14 dargestellt. Da mit Safety Automaten jedoch nicht ausgedrückt werden kann, dass die Funktion angesprungen werden muss, beschreibt dieser Automat lediglich das Verhalten, dass wenn die Funktion angesprungen wird, dies zu einem Beispiel dafür führt. Auch hierbei ist ein manuelles Nachvollziehen des Verhaltens im Simulator notwendig.

Mittels 4.7 kann die Eigenschaft so umformuliert werden, dass die Funktion in jedem Fall erreicht wird. Eine Darstellung der Formel als Safety Automat ist aufgrund der oben beschriebenen Einschränkungen nicht möglich.

$$\mathbf{AF}\ (\text{PC} = \#\text{Start\_Funktion\_Y}) \tag{4.7}$$

Mittels Implikation kann man eine ähnliche Formel dazu aufstellen. Diese beschreibt ebenfalls die Eigenschaft, dass bei Erreichen der Funktion, diese komplett durchlaufen wird. Allerdings existiert hierbei der Nachteil, dass es auch die Möglichkeit gibt, dass die Funktion niemals angesprungen wird, da die Formel aufgrund der Implikation auch gültig ist, wenn PC niemals gleich #Start_Funktion_Y ist.

$$\begin{aligned}&\mathbf{AG}\ (\text{PC} = \#\text{Start\_Funktion\_Y}\ \Rightarrow\\ &\mathbf{A}\ \text{TT}\ \mathbf{U}\ \text{PC} = \#\text{Ende\_Funktion\_Y})\end{aligned} \tag{4.8}$$

Der Safety Automat in Abbildung 4.15 beschreibt dasselbe Verhalten. Wird die Funktion nicht komplett durchlaufen, so wird die entsprechende Transition im Automaten jedoch als „nicht verwendet“ markiert.

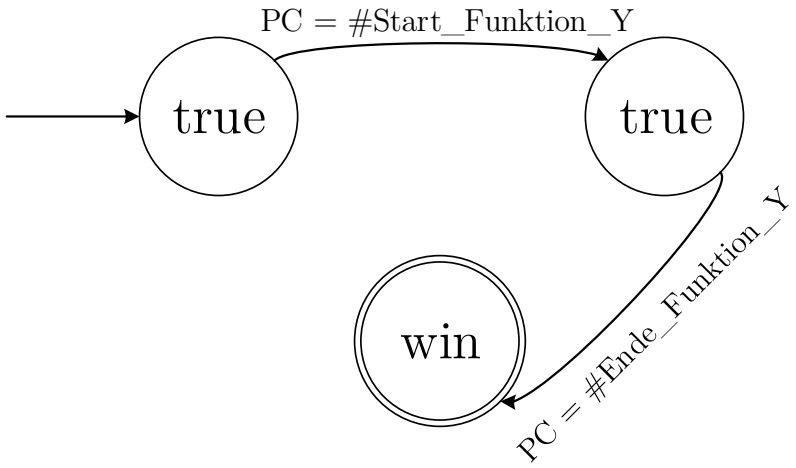


Abbildung 4.15: *SA* zur Überprüfung des Durchlaufens einer Funktion

Ob die Funktion *Funktion_Y* mindestens einmal angesprungen und danach komplett durchlaufen wird, prüft folgende erweiterte Formel:

$$\begin{aligned} &\mathbf{AF}\ (\mathrm{PC} = \#\mathrm{Start\_Funktion\_Y} \wedge \\ &\mathbf{A}\ \mathrm{TT}\ \mathbf{U}\ \mathrm{PC} = \#\mathrm{Ende\_Funktion\_Y}) \end{aligned} \tag{4.9}$$

Die Formel 4.9 beschreibt, dass entlang aller Pfade in allen Zuständen gilt, dass wenn der PC auf den Anfang der Funktion zeigt, von da an alle erreichbaren Zustände (TT) möglich sind, bis das Programm das Ende der Funktion erreicht hat.

Die Formeln 4.7 und 4.9 beschreiben allerdings nur das einmalige Anspringen der Funktion. Möchte man die Anforderungen um die Möglichkeit erweitern, dass die Funktion mehrfach angesprungen können werden soll, so entsteht folgende Formel:

$$\begin{aligned} &\mathbf{AG}\ \mathbf{AF}\ (\mathrm{PC} = \#\mathrm{Start\_Funktion\_Y} \wedge \\ &\mathbf{A}\ \mathrm{TT}\ \mathbf{U}\ \mathrm{PC} = \#\mathrm{Ende\_Funktion\_Y}) \end{aligned} \tag{4.10}$$

Eine Darstellung als Safety Automat ist hierbei nicht möglich. Nach dem ersten erfolgreichen Durchlaufen des Automaten würden die folgenden Durchläufe einer Funktion zwar durch den Automaten abgebildet, jedoch nicht vom ersten Durchlauf unterschieden werden können.

Ergebnis

Am Fallbeispiel soll geprüft werden, ob es möglich ist, dass die Funktion *Funktion_Y* erreicht werden kann. Zuerst wird verifiziert, dass die Hauptfunktion `main` unter allen

Umständen erreicht wird. Die entsprechende Formel basiert dabei auf 4.7 und ist in **.Error** gültig. Ein *SA* dazu existiert nicht.

$$\mathbf{AF}\ (\mathrm{PC} = \mathrm{main}) \tag{4.11}$$

Als weiteres soll das Erreichen der Interrupt-Funktion `tra_irq` geprüft werden. Formel 4.12 ist in **.Error** allerdings ungültig. Daraufhin wurde ein weiterer Fehler (Fehler 2, Abschnitt 4.6.9) gefunden, der durch eine fehlerhafte Einstellung der Interrupt-Priorität diesen versehentlich deaktiviert.

$$\mathbf{AF}\ (\mathrm{PC} = \mathrm{tra\_irq}) \tag{4.12}$$

In Version .**fixed1** ist dieser Fehler behoben, allerdings ist das Ergebnis von Formel 4.12 immer noch ungültig. Bei der genaueren Analyse des Gegenbeispiels wurde festgestellt, dass der Aufruf eines Interrupts im Simulator nicht-deterministisch behandelt wird. Dies bedeutet, es existiert immer eine Möglichkeit, dass der Interrupt nicht ausgeführt wird. Dies ist sinnvoll, wenn Interrupts an externe Ereignisse gekoppelt sind. Bei z. B. Interrupts von internen Zählereinheiten muss dies bei der Auswertung der Ergebnisse des Model-Checkings allerdings berücksichtigt werden.

$$\mathbf{EF}\ (\mathrm{PC} = \mathrm{tra\_irq}) \tag{4.13}$$

Daraufhin kann nur ein einfaches Erreichen der Interrupt-Funktion mittels Formel 4.13 auf Basis von 4.6 verifiziert werden. In **.Error** ist diese Formel aufgrund von Fehler 2 ungültig und in .**fixed1** korrekterweise gültig. Das gleiche Verhalten kann mittels *SA* Abbildung B.2 auf Basis von 4.14 gezeigt werden.

Als nächstes soll das komplette Durchlaufen der Funktion `wdt_watchdog` (Endadresse: `0x4230`) mit Formel 4.14 auf Basis von 4.9 und mit dem Automaten Abbildung B.4 basierend auf 4.15 überprüft werden. Das Model-Checking ergibt für beide Formeln ein gültiges Ergebnis, welches zeigt, dass die Funktion in jedem Fall mindestens einmal erreicht und durchlaufen wird.

$$\mathbf{AG}\ ((\mathrm{PC} = \mathrm{wdt\_watchdog}) \wedge (\mathbf{A}\ \mathrm{TT}\ \mathbf{U}\ \mathrm{PC} = \mathrm{0x4230})) \tag{4.14}$$

Als nächstes soll an derselben Funktion überprüft werden, ob diese immer wieder erreicht und durchlaufen werden kann oder ob es einen fehlerhaften Programmablauf gibt, bei dem dies nicht so ist. Auf Basis von Formel 4.10 wird 4.15 in **.Error** als gültig verifiziert. Ein passender *SA* existiert hierzu nicht.

$$\mathbf{AG\ AF}\ ((\mathrm{PC} = \mathrm{wdt\_watchdog})\ \wedge (\mathbf{A}\ \mathrm{TT}\ \mathbf{U}\ \mathrm{PC} = \mathrm{0x4230})) \tag{4.15}$$

4.6.4 Anforderung 3: Kontrolliertes Zurücksetzen

Mit dieser Anforderung soll das kontrollierte Zurücksetzen (Reset) des Steuergeräts verifiziert werden. Die meisten Mikrokontroller bieten über eine interne Funktion das Zurücksetzen des gesamten Systems an. Diese Funktion kann durch Software ausgelöst werden. Bei dem Zurücksetzen werden alle Ausgänge und Speicherbereiche auf ihren Standardwert zurückgesetzt, und das Programm wird von Beginn an ausgeführt. Dieses Verhalten kann während des Betriebs zu Störungen führen und darf deshalb nur kontrolliert verwendet werden.

```
0 void ResetFkt()
  {
2   prc1=1;    /* Enable writing */
    pm03=1;    /* Reset MCU */
4   prc1=0;    /* Diable writing */
  }
```

Quelltext 4.2: Funktion zum Zurücksetzen des Mikrokontrollers

Das Ausführen des Zurücksetzens erfolgt durch das Setzen des Registers `Reset_Register` von 0 auf 1. Um dieses Verhalten zu kontrollieren, sollte dies an genau einer Stelle im Programm ausgeführt werden, z. B. in der Funktion `ResetFkt` (Quelltext 4.2). Dabei ist `pm03` das Reset-Register.

$$\begin{aligned}&\mathbf{AG}\ ((\ \mathrm{PC} > \#\mathrm{Start\_ResetFkt}\ \wedge\ \mathrm{PC} < \#\mathrm{End\_ResetFkt}) \Rightarrow \\ &(\mathbf{A}\ \mathrm{Reset\_Register} = 0\ \mathbf{U}\ \mathrm{Reset\_Register} = 1))\end{aligned} \tag{4.16}$$

Formel 4.16 beschreibt die Eigenschaft des Systems, dass entlang aller Pfade für alle Zustände gilt: Wenn der PC sich innerhalb der Funktion `ResetFkt` befindet, ist von da an das Reset-Register solange gleich null, bis es auf eins gesetzt wird. Hiermit wird das Setzen eines Registers in Abhängigkeit der Position im Programm beschrieben. Abbildung 4.16 zeigt den dazu passenden Automaten.

Allerdings beschreibt diese Anforderung nur, dass das Zurücksetzen innerhalb der definierten Funktion ausgeführt werden soll. Ob an einer anderen Stelle in der Software ebenfalls dieses Register von 0 auf 1 gesetzt wird, kann damit nicht geprüft werden. Dazu muss die Anforderung wie folgt formalisiert werden:

$$\begin{aligned}&\mathbf{EF}\ (\mathrm{Reset\_Register} = 1\ \wedge \\ &!(\ \mathrm{PC} > \#\mathrm{Start\_ResetFkt}\ \wedge\ \mathrm{PC} < \#\mathrm{End\_ResetFkt}))\end{aligned} \tag{4.17}$$

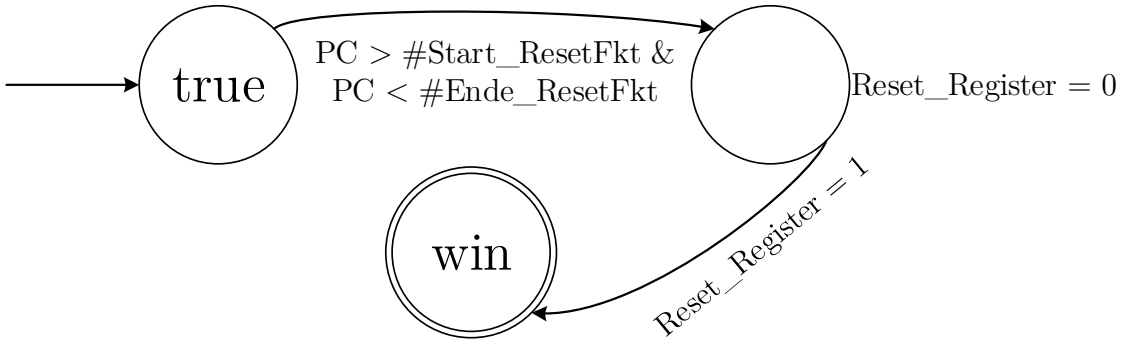


Abbildung 4.16: *SA* zur Überprüfung des korrekten Setzens eines Registers in Abhängigkeit der Programmposition

Formel 4.17 ist genau dann wahr, wenn entlang mindestens eines Pfades mindestens ein Zustand existiert, in dem das Reset-Register auf 1 gesetzt wird und sich der PC gleichzeitig nicht in der Funktion `ResetFkt` befindet. Ist diese Formel erfüllt, so kann über den Simulator von ARCADE die entsprechende Stelle im Programm identifiziert werden. Der Safety Automat in Abbildung 4.17 beschreibt dieselbe Eigenschaft wie Formel 4.17.

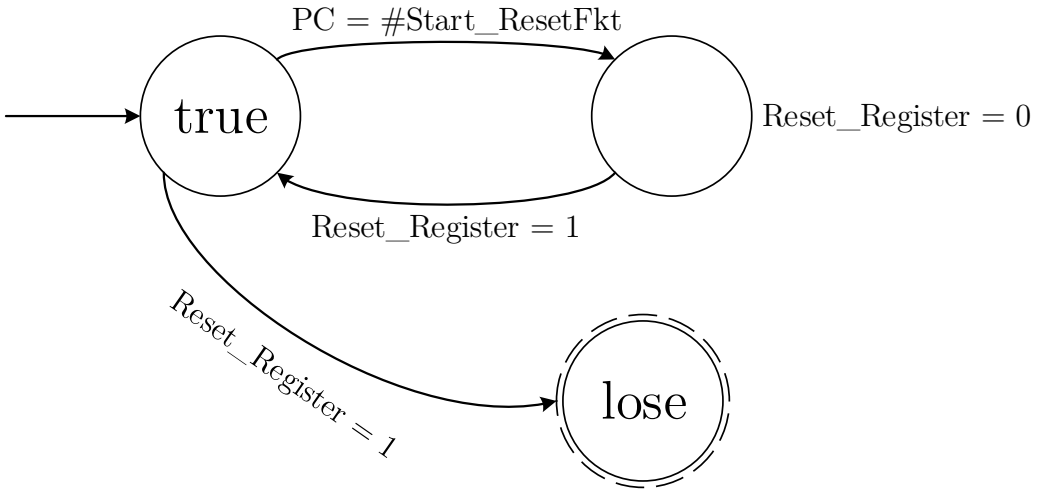


Abbildung 4.17: *SA* zur Überprüfung des Setzens eines Registers exklusiv in einer Funktion

Ergebnis

Am Fallbeispiel wird dies konkret an der Funktion `Reset` (Startadresse: `0x4234`, Endadresse: `0x424C`, Adresse des `pm03`-Registers: `0x00EC Bit 3`) untersucht. Die verwendete Formel 4.18 basiert dabei auf Formel 4.17.

$$\textbf{EF}\ (\$0x00EC{:}3 = 1\ \wedge !(\ PC >= 0x4234\ \wedge\ PC <= 0x424C)) \tag{4.18}$$

In **.Error** ist das Ergebnis des Model-Checkings gültig. Dies bedeutet allerdings, dass ein Pfad im Programmablauf existiert, bei dem außerhalb der Funktion `Reset` das Register gesetzt wird. Mit Hilfe des Gegenbeispieles kann Fehler 1 (Abschnitt 4.6.9) lokalisiert werden. Dieser ist in **.fixed3** behoben. Eine Überprüfung der Formel in **.fixed3** liefert ein ungültiges Ergebnis, welches das korrekte Verhalten ist. Der zu dieser Formel passende *SA* ist in Abbildung B.5 dargestellt und basiert auf Formel 4.17. Die Ergebnisse unter Verwendung des Automaten sind äquivalent zu denen der *CTL*-Formel.

4.6.5 Anforderung 4: Interrupt-freie Initialisierung

Mit dieser Anforderung dürfen während der Initialisierungs-Phase (kurz Init-Phase) keine Interrupts auftreten. Da erst nach erfolgreicher Init-Phase das System bereit ist, korrekt auf Interrupts zu reagieren, sollen diese währenddessen komplett deaktiviert und erst nach der Initialisierung aller Module aktiviert werden.

$$\begin{aligned}&\textbf{AG}\ ((\text{PC} > \#\text{Start\_InitPhase}\ \wedge\ \text{PC} < \#\text{Ende\_InitPhase})\\ &\Rightarrow \text{InterruptRegister} = 0)\end{aligned} \tag{4.19}$$

Dazu prüft Formel 4.19, ob entlang aller Pfade für alle Zustände gilt, dass, wenn der PC sich zwischen *Start_InitPhase* und *Ende_InitPhase* befindet, die Interrupts deaktiviert sind (*InterruptRegister* = 0). Der Safety Automat in Abbildung 4.18 beschreibt dasselbe Verhalten.

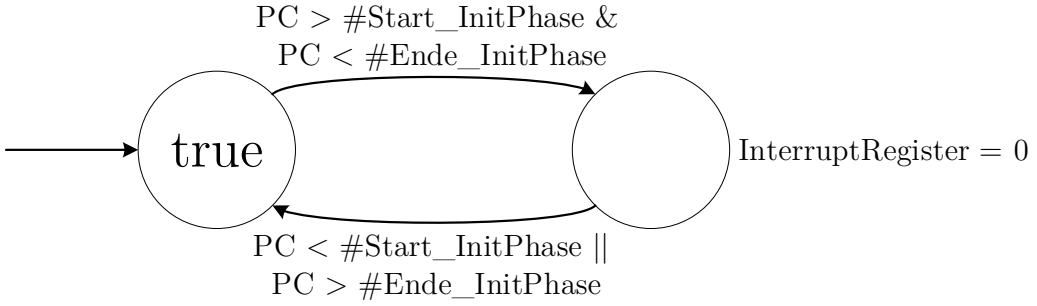


Abbildung 4.18: *SA* zur Überprüfung deaktivierter Interrupts während der Initialisierung

Während der Initialisierungsphase (Quelltext 4.3) werden mehrere Initialisierungsfunktionen von Modulen des Systems ausgeführt. Dazu springt das Programm an den entsprechenden Stellen zu den Subfunktionen. Diese Subfunktionen liegen jedoch nicht im Adressbereich zwischen *Start_InitPhase* und *Ende_InitPhase*. Deshalb kann mit

Formel 4.19 nur überprüft werden, ob nach dem *DISABLE_IRQ* Befehl (Codezeile 2) irgendwo Interrupts versehentlich aktiviert werden und dies auch bleiben. Werden z. B. in einer der Subfunktionen Interrupts aktiviert und kurz danach erneut deaktiviert (siehe Quelltext 4.4), so kann dieses Verhalten nicht erkannt werden.

```
0  void main()
   {
2    DISABLE_IRQ /* #Start_InitPhase */
     /* Initialization Functions */
4    DIO_InitSync(&DioSetting);
     ADC_InitSync(&AdcSetting);
6    PWM_InitSync(&PwmSetting);
     RS232_InitSync(UART_0);
8    TimerA_init();
     TimerB_init();
10   ENABLE_IRQ /* #Ende_InitPhase */

12   while (true)
     {
14     ... /* Task Execution */
     }
16 }
```

Quelltext 4.3: Initialisierungsphase

```
0  void TimerA_init()
   {
2    DISABLE_IRQ
     ...
4    ENABLE_IRQ
   }
```

Quelltext 4.4: Initialisierungsfunktion von TimerA

Eine Erweiterung von Formel 4.19, um auch Subfunktionen entsprechend miteinzubeziehen, führt zu:

$$\begin{aligned} &\mathbf{AG}\ (\text{PC} = \#\text{Start\_InitPhase} \Rightarrow \\ &\mathbf{A}\ \text{InterruptRegister} = 0\ \mathbf{U}\ \text{PC} = \#\text{Ende\_InitPhase}) \end{aligned} \tag{4.20}$$

Formel 4.20 prüft, ob entlang aller Pfade für alle Zustände gilt, dass, wenn der PC am Beginn der Init-Phase (*#Start_InitPhase*) ist, die Interrupts solange deaktiviert sind (*InterruptRegister* = 0), bis der PC das Ende (*#Ende_InitPhase*) der Init-Phase erreicht hat. Für den Programmablauf bedeutet das, egal was das Programm macht, wenn der PC den Anfang der Init-Phase erreicht hat, müssen, von da an bis zum Erreichen des Endes

der Init-Phase, die Interrupts deaktiviert sein. Der Safety Automat in Abbildung 4.19 beschreibt dasselbe Verhalten.

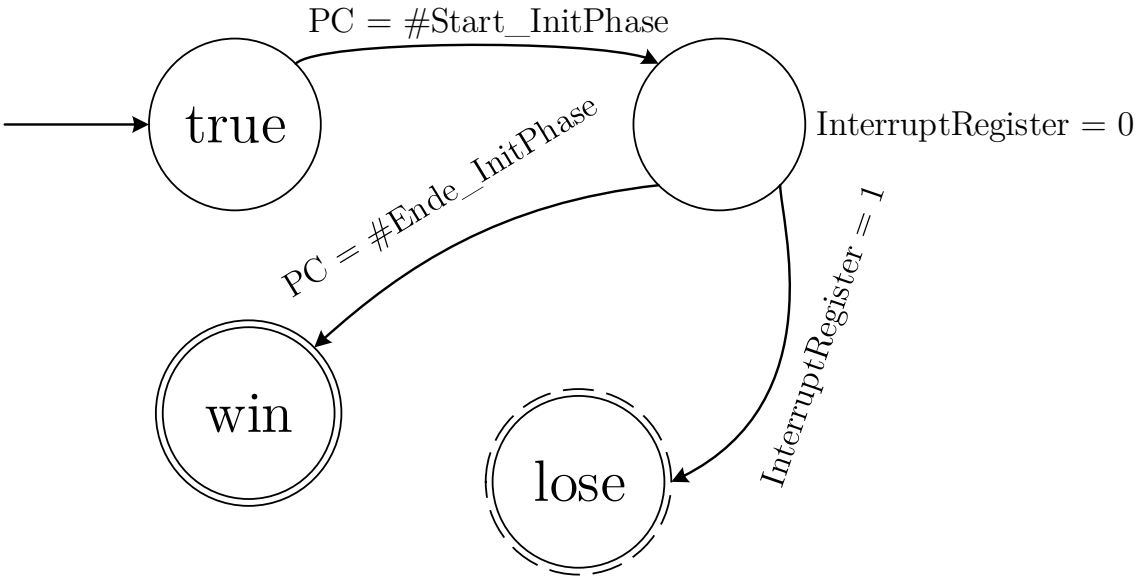


Abbildung 4.19: *SA* zur Überprüfung deaktivierter Interrupts während der Initialisierung einschließlich Subfunktionen

Ergebnis

Im Fallbeispiel wird zuerst geprüft, ob die Init-Phase (Startadresse: `0x4088`, Endadresse: `0x4096`) frei von Interrupts ist. Das zu prüfende Interrupt-Register ist `I`. Auf Basis von Formel 4.19 wird Formel 4.21 abgeleitet. Entsprechend folgt der Automat in Abbildung B.6 dem aus Abbildung 4.18.

$$\mathbf{AG}\ ((\mathrm{PC} > 0\mathrm{x}4088\ \wedge\ \mathrm{PC} < 0\mathrm{x}4096) \Rightarrow \mathrm{I} = 0) \tag{4.21}$$

Sowohl die Formel als auch der Automat decken Fehler 3 (Abschnitt 4.6.9) in **.Error** auf. In der fehlerbereinigten Version **.fixed2** sind sowohl die Formel als auch der Automat korrekterweise gültig. Fehler 3 wird hierbei nur aus dem Grund erkannt, da die Sub-Funktion, welche außerhalb des in der Formel definierten Adressbereiches liegt, die Interrupts dauerhaft aktiviert. Beim Zurückspringen des Programms in den definierten Adressbereich ist somit das entsprechende Register `I` weiterhin auf den Wert 1 gesetzt, was die Formel ungültig werden lässt.

Um in einem nächsten Schritt ein versehentliches kurzfristiges Aktivieren der Interrupts in einer der Subfunktionen ebenfalls auszuschließen, wird Formel 4.22 und der Automat in Abbildung B.7 verwendet. Diese basieren auf Formel 4.20 bzw. auf dem Automaten in

Abbildung 4.19. Damit konnte in **.fixed3** Fehler 4 (Abschnitt 4.6.9) gefunden und dessen korrektes Beheben in **.fixed4** verifiziert werden.

$$\mathbf{AG}\ (\mathrm{PC} = 0\mathrm{x}4088\ \Rightarrow \mathbf{A}\ \mathrm{I} = 0\ \mathbf{U}\ \mathrm{PC} = 0\mathrm{x}4096) \tag{4.22}$$

4.6.6 Anforderung 5: Keine Division durch Null

Dieser Anforderung nach darf während des Programmablaufs keine Division durch Null stattfinden, da dies zu einem unkontrollierten Verhalten führen kann. Quelltext 4.5 zeigt beispielhaft einen Fall, in dem eine Division durch Null auftreten kann. Dabei ist die Variable `div` global. Ein Überlauf während des Inkrementierens von `div` führt dazu, dass beim erneuten Aufruf der Funktion `overflow_test` eine Division durch Null auftritt.

```
0 char div = 254;
  void overflow_test(void)
2 {
    char result = 0;
4   result = 250 / div; // #Division
    div ++;
6 }
```

Quelltext 4.5: Division durch Null

Um zu überprüfen, ob eine nicht erlaubte Division auftreten kann, können jegliche Variablen, die in arithmetischen Operationen als Divisor eingesetzt werden, auf ihre Wertigkeit an den entsprechenden Programmstellen mit folgender Formel überprüft werden:

$$\mathbf{EF}\ (\mathrm{PC} = \#\mathrm{Division}\ \wedge\ \mathrm{div} = 0) \tag{4.23}$$

Formel 4.23 ist genau dann wahr, wenn entlang mindestens eines Pfades mindestens ein Zustand existiert, in dem der PC an der entsprechenden Programmstelle der Division und gleichzeitig die Variable `div` gleich Null ist. Falls diese Anforderung nicht erfüllt ist, kann anhand des Gegenbeispieles eruiert werden, wie ein Programmablauf aussieht, in dem diese Division durch Null möglich ist. Diese Eigenschaft kann ebenfalls mit dem Safety Automat aus Abbildung 4.20 abgebildet werden.

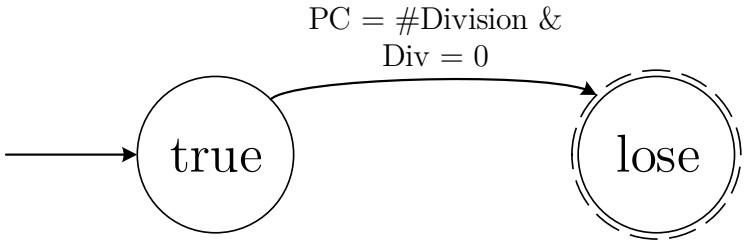


Abbildung 4.20: *SA* zur Überprüfung auf eine mögliche Division durch Null an einer speziellen Stelle im Programm

Um das ganze Programm auf nicht-erlaubte Divisionen hin zu überprüfen, muss mit dem oben beschriebenen Ansatz jede einzelne Division lokalisiert und eine dazu passende Formel manuell erstellt werden. Alternativ dazu kann ein spezielles Register des verwendeten Mikrokontrollers genutzt werden. Das sogenannte *O-Flag* (overflow) wird durch interne Mechanismen des Mikrokontrollers bei Überläufen von arithmetischen Operationen auf den Wert 1 gesetzt. Eine Division durch Null erzeugt einen entsprechenden Überlauf und somit eine Wertänderung des *O-Flags*. Eine diesen Umstand nutzende Formel sieht wie folgt aus:

$$\mathbf{AG}\ \mathrm{O} = 0 \tag{4.24}$$

Diese Formel ist genau dann wahr, wenn entlang aller Pfade in allen Zuständen das *O-Flag* gleich Null ist. Anhand eines daraus resultierenden Gegenbeispieles kann die dazu passende Stelle im Programmablauf lokalisiert werden. Da diese Formel auch jeden anderen Überlauf von arithmetischen Operationen mit einschließt, wurde ARCADE dahingehend erweitert, dass eine automatische Prüfung das Setzen des *O-Flags* im Zusammenhang mit einer Divisions-Operation eine Warnung generiert. Der Safety Automat in Abbildung 4.21 beschreibt dasselbe Verhalten.

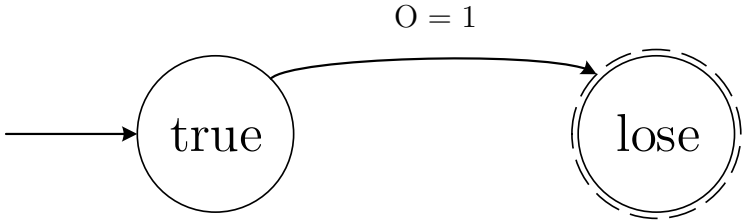


Abbildung 4.21: *SA* zur Überprüfung auf mögliche Divisionen durch Null mittels Overflow-Register

Ergebnis

Das Fallbeispiel wird auf eine mögliche Division mit Null anhand von Formel 4.24 hin überprüft. Dabei wird Fehler 5 (Abschnitt 4.6.9) in **.Error** gefunden. Sowohl der zur

Formel äquivalente Automat aus Abbildung B.8 als auch die Formel selber verifizieren die Anforderung in **.fixed2**. Der Compiler zeigt diesen Fehler ebenfalls an.

4.6.7 Anforderung 6: Kein Überlauf des Stacks

Mit dieser Anforderung soll sichergestellt werden, dass der für den Stack reservierte Speicher ausreicht oder es zu keinem Überlauf des Stacks kommt. Der Stack wird bei Mikrokontrollern dazu verwendet, Rücksprungadressen sowie ganze Funktionskontexte wie lokale Variablen und Registerinhalte bei Aufruf von Sub-Funktionen bzw. beim Auftreten von Interrupts zu sichern. Dabei arbeitet der Stack nach dem LIFO-Prinzip (last in, first out) und dessen Größe wird statisch zur Kompilierzeit festgelegt. Ein Überlauf des Stacks hat, wie auch jeder andere Überlauf eines Speicherbereiches, ein unkontrollierbares Systemverhalten zur Folge. Durch das Speichern von Rücksprungadressen im Stack kann ein Überschreiben dieser kritischere Auswirkungen haben als das bei anderen Speicherbereichen, in denen der Inhalt von Variablen abgelegt ist, der Fall ist.

$$\mathbf{AG}\ \mathrm{TT} \tag{4.25}$$

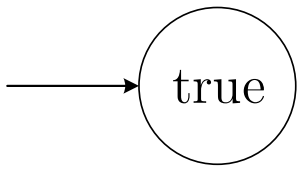


Abbildung 4.22: *SA* als allgemeingültiger Automat

Durch die Formel 4.25 sowie durch den Safety Automat aus Abbildung 4.22 wird ARCADE aufgrund des allgemeingültigen Ausdrucks TT veranlasst, den gesamten möglichen Zustandsraum zu erstellen. Dabei kann die maximal mögliche Größe des Stacks ermittelt und manuell mit dem im Programm eingestellten Wert verglichen werden. Die statische Analyse in ARCADE kann ebenfalls die maximale Stackgröße errechnen, jedoch nur mit einer erhöhten Ungenauigkeit.

Ergebnis

Im Fallbeispiel werden Formel 4.25 und der Automat aus Abbildung B.3 auf **.Error** sowie auf **.fixed2** angewendet. Der Model-Checking Prozess wird bei **.fixed2** nach einer Stunde erfolglos abgebrochen. Bei **.Error** wird ein maximaler Stack errechnet. Jedoch sind bei **.Error** durch den vorhandenen Fehler keine Interrupts aktiv, und somit ist die maximale Größe des Stacks nicht repräsentativ. Hier wäre der Einsatz von anderen Werkzeugen zur statischen Stack-Analyse wie z. B. StackAnalyzer[24] sinnvoll.

[24] `http://www.absint.com/stackanalyzer`

4.6.8 Anforderung 7: Gültige Zugriffe auf Arrays

Die Anforderung soll dazu dienen, dass Zugriffe auf Elemente in einem Array nur in dessen definiertem Rahmen erfolgen. Arrays werden häufig im C-Code verwendet, um Daten zum schnelleren Zugriff in Feldern zu organisieren. Die einzelnen Elemente dieser Felder können durch Indizes adressiert werden. Auf C-Code Ebene ist es möglich, einen Index zu wählen, der nicht mehr im vorab definierten Bereich der Array-Elemente liegt. Erfolgt auf dieses so adressierte Element ein Lese- bzw. Schreibzugriff, so werden die dem Array nachfolgenden Speicherstellen gelesen bzw. beschrieben. Diese sind normalerweise durch andere Variablen belegt. Ein Zugriff darauf führt zu einem unkontrollierten Verhalten des Systems.

```
0 int array[10];
  void ArrayAccess()
2 {
    ...
4   for (indexData = 0; indexData <= 10; indexData++)
    {
6     result += array[indexData]; // #Array_Zugriff
    }
8   ...
  }
```

Quelltext 4.6: Array Zugriff

Quelltext 4.6 zeigt einen Auszug aus einem Programm. Dieses enthält das global definierte Array `array`, welches zehn Elemente enthält. In der Funktion `ArrayAccess` durchläuft nun eine Schleife dieses Array und greift auf dessen Elemente lesend zu. Durch die Schleifen-Bedingung wird diese unbeabsichtigt 11-mal anstelle von 10 mal durchlaufen. Der letzte Zugriff erfolgt demnach nicht mehr auf den vorgesehenen Speicherbereich des Arrays, sondern auf direkt angrenzende Speicherstellen. Hierbei ist zu beachten, dass Array-Indizierung Null-basierend erfolgt, der Zugriff auf `array[10]` somit außerhalb der Array-Grenze liegt.

C-Compiler transformieren C-Code Arrays in Speicherzugriffe des Maschinencodes, wobei Informationen über die Array-Grenzen nicht erhalten bleiben. Die Indizierung im C-Code erfolgt entweder durch einen konstanten Index oder durch eine Variable, die den Wert des Index beinhaltet. Bei konstanten Werten wäre es dem Compiler möglich, ein Überschreiten der Indexgrenze zu erkennen. Bei Einsatz einer Variablen muss erst eine Analyse deren möglicher Wertebereiche erfolgen. Diese Analyse kann entweder direkt im C-Code oder aber auch auf Binär-Code-Ebene geschehen. Eine Voraussetzung hierfür ist die Verwendung einer Variablen als Index wie Quelltext 4.6 zeigt. Mit dem zusätzlichen Wissen über die konkrete Stelle im Programmcode ergibt sich folgende Formel:

$$\begin{aligned} &\mathbf{AG}\ (\mathrm{PC} = \#\mathrm{Array\_Zugriff} \\ &\Rightarrow \mathrm{indexData} >= 0\ \wedge\ \mathrm{indexData} < 10) \end{aligned} \tag{4.26}$$

Formel 4.26 prüft, ob entlang aller Pfade für alle Zustände gilt, dass, wenn sich der PC an der Adresse des Zugriffes auf das Array befindet, die Index-Variable *indexData* zwischen Null und dem maximalen Wert (10) liegen soll. Die Grenzen müssen dabei immer aufwendig manuell nachgehalten werden.

Ergebnis

Die Formel 4.26 ist sehr ähnlich denen aus Unterabschnitt 4.6.2. Um gültige Zugriffe auf Arrays auf Binär-Code-Ebene prüfen zu können, ist es notwendig, lokale Variablen zur Indizierung zu verwenden und dem Compiler die Optimierung dieser zu verbieten. Eine Überprüfung erfolgt allerdings effizienter auf C-Code-Ebene durch statische Analysen mit z. B. PC-lint oder Goanna (siehe Tabelle 2.2).

4.6.9 Gefundene Fehler

Im Folgenden werden die während der Evaluierung gefundenen Fehler näher beschrieben.

Fehler 1:

In der Initialisierungsphase des Hauptprogramms in der Funktion `main()` wird in das Register `pm03` versehentlich ein falscher Wert geschrieben (Quelltext 4.7). Dies führt zu einem ungewollten Reset des Mikrokontrollers.

```
0  void main(void)
   {
2    DISABLE_IRQ
     wdt_watchdog_init();   /* Initialization of basic driver */
4    TimerA_init();         /* Initialization of Schedule Timer TRA */
     tstart_tracr = 1;      /* start timer A */
6    pm03 = 1;              /// Fehler 1
     ENABLE_IRQ
8
     while(1)
10   {
       wdt_watchdog();
12   }
   }
```

Quelltext 4.7: Fehler 1

Fehler 2:

Die Interrupt-Priorität des Zählers TRA wurde während der Initialisierung (siehe Quelltext 4.8) fälschlicherweise auf 0 gesetzt. Dies deaktiviert allerdings den Zähler, und somit wird der Interrupt nie ausgelöst und die entsprechende Interrupt-Service-Routine nie angesprungen. Da der einzige verwendete Interrupt deaktiviert ist, hat dieser Fehler auch zur Folge, dass der Zustandsraum entsprechend kleiner ist, und somit Model-Checking bei einigen Eigenschaften beschleunigt wird.

Fehler 3:

Während der Initialisierungsroutine dürfen keine Interrupts aktiviert sein. In einer Unterfunktion wird allerdings Quelltext 4.8 verwendet. In einer Initialisierungsroutine eines Software-Moduls werden die Interrupts für das Setzen eines speziellen Registers deaktiviert und nach dem Setzen wieder aktiviert. Dieses Vorgehen beim Schreiben auf das Interrupt-Kontroll-Register wird in der Hardware-Dokumentation [76] des Mikrokontrollers entsprechend beschrieben. Damit werden jedoch mit der Ausführung von Programm-Zeile 6 auch alle anderen Interrupts aktiviert. Aufgrund der Anforderung soll dies während der Initialisierungsphase jedoch vermieden werden.

```
0 ...
  /* disable interrupts macro */
2 DISABLE_IRQ;  /// Fehler 3
  /* TRA  interrupt priority level */
4 traic = 0x00; /// Fehler 2
  /* enable interrupts macro */
6 ENABLE_IRQ;   /// Fehler 3
  ...
```

Quelltext 4.8: Fehler 2 und 3

Fehler 4:

```
0       void TimerB_init()
        {
2         ENABLE_IRQ /// Fehler 4
          ...
4         DISABLE_IRQ /// Fehler 4
        }
6
```

Quelltext 4.9: Fehler 4

In einer Unterfunktion einer Initialisierungsroutine (siehe Quelltext 4.9) werden die Interrupts versehentlich aktiviert und danach wieder deaktiviert. In dieser Zeit können andere Interrupts, welche gegebenenfalls noch nicht vollständig initialisiert worden sind, aktiv werden und somit zu unkontrolliertem Verhalten führen.

Fehler 5:

In Quelltext 4.10 wird versehentlich eine offensichtliche Division durch Null durchgeführt. Sowohl die internen Analysen des Compilers als auch die Verifikation mittels Model-Checking finden diesen Fehler.

```
0 ...
  if( uiIOSendValue == 0 )
2 {
    uiCompareVal = 2 / uiIOSendValue; /// Fehler 5
4   uiIOSendValue = 1;
  }
6 ...
```

Quelltext 4.10: Fehler 5

Fehler 6:

Nach einem Software-Update der Entwicklungsumgebung wurden die Konfigurationseinstellungen für den Linker in der Projektdatei nicht korrekt übernommen. Dabei wurden die Adressbereiche für die Initialisierungswerte der globalen Variablen auf andere Bereiche im Speicher verschoben. Da beim Starten des Mikrokontrollers die Initialisierungswerte der Variablen aus dem ROM-Bereich in den dazu passenden RAM-Bereich kopiert werden, wurden die globalen Variablen mit den falschen Werten initialisiert.

4.6.10 Design-For-Verifiability

In diesem Abschnitt sollen die Erfahrungen zum effizienteren Einsatz von Model-Checking festgehalten werden. Ob die Umsetzung nur für neue Projekte oder aber auch noch für vorhandene Projekte lohnenswert ist, muss im Einzelfall abgewogen werden. Im Folgenden werden Hinweise sowohl zum einfacheren Umgang mit der Formalisierung der Anforderungen als auch zur Reduktion des Zustandsraumes gegeben.

Vereinfachung der Formalisierung der Anforderungen:

- Um auch Zwischenergebnisse von komplexen Berechnungen überprüfbar zu machen, kann das Unterteilen dieser Berechnungen in mehrere Zwischenschritte hilfreich sein. Dabei müssen etwaige Optimierungen durch den Compiler beachtet und gegebenenfalls deaktiviert werden.

- Damit bei der Verwendung von Positionsangaben, z. B. über den Programmzeiger und einer expliziten Programmadresse, das manuelle Anpassen dieser Adressen vermieden wird, können spezielle globale Variablen als Markierungen explizit in den Programmcode eingefügt werden. Diese können dabei mit ihrem Namen und einem Wert, der der Markierung zugeordnet ist, in CTL-Formeln oder im Safety Automat verwendet werden. Das Anpassen der Adressen wird z. B. durch das Einfügen von zusätzlichem Code vor der angegebenen Programmposition notwendig.

Reduktion des Zustandsraums (aufbauend auf Ideen aus [91, 104]):

- Verwenden von entsprechenden Abstraktionstechniken, die vom Model-Checking Werkzeug zur Verfügung bereitgestellt werden.
- Es sollten vorzugsweise lokale anstelle von globalen Variablen verwendet werden.
- Die Datentypen der verwendeten Variablen sollten so klein wie möglich gewählt werden (8-Bit, 16-Bit, 32-Bit).
- Ungenutzte Variablen und Code-Teile sollten entfernt werden.
- Die Anzahl der Interrupts sollte gering gehalten werden.
- Um den Aufbau des Kontrollflussgraphen durch die statische Analyse zu ermöglichen bzw. zu vereinfachen, muss indirekter Kontrollfluss vermieden werden. Dazu sollte z. B. auf indirekte Sprunganweisungen verzichtet werden. Das Ersetzen von `switch`-Anweisungen durch `if-else`-Anweisungen (siehe Quelltext 4.11 und Quelltext 4.12) ist hierfür hilfreich.

```
switch(InputValue)
{
  case 0:
    doSomething_1();
    break;
  case 1:
    doSomething_2();
    break;
  case 2:
    doSomething_3();
    break;
  default:
    doSomething_4();
}
```

Quelltext 4.11: Switch-Anweisungen

```
if (InputValue == 0)
{
  doSomething_1();
} else if (InputValue == 1)
{
  doSomething_2();
} else if (InputValue == 2)
{
  doSomething_3();
}
else
{
  doSomething_4();
}
```

Quelltext 4.12: If-Else-Anweisungen

- Modellierung der Umgebung mittels *UDE* (Abschnitt 4.2). Damit können zum einen Fehler, die der Model-Checker findet, welche allerdings im realen System nicht vorkommen können, ausgeschlossen werden. Dazu können z. B. die möglichen Werte, die ein analoger Eingangskanal annehmen kann, auf die real möglichen beschränkt werden (Abbildung 4.24). Weiterhin können Interrupts vollständig unterdrückt (Abbildung 4.23) oder deren Auftreten nur nicht-verschachtelt (Abbildung 4.25) zugelassen werden.

Abbildung 4.23: Blockieren aller Interrupts

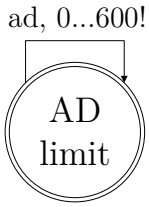


Abbildung 4.24: Limitierung eines analogen Eingangs

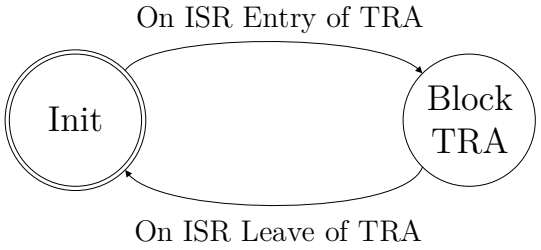


Abbildung 4.25: Blockieren von verschachtelten Interrupts

4.6.11 Fazit

Im Folgenden wird erläutert, inwiefern die in Abschnitt 4 aufgestellten Anforderungen mit dem zuvor beschriebenen Ansatz erfüllt werden:

- *Zur Verifikation der Software sollen im Entwicklungsprozess bereits erzeugte Artefakte verwendet werden.*

  Dadurch, dass das Model-Checking den Binär-Code als Systemmodell nutzt, müssen keine separaten Zwischendarstellungen des Systems erstellt werden. Hierbei bleibt die Konsistenz der Software erhalten, da derselbe Binär-Code nach erfolgreicher Verifikation auf die Ziel-Hardware programmiert wird.

- *Der Einsatz von Ressourcen zum Finden von Fehlern soll verringert werden.*

  Diese Anforderung muss man differenziert betrachten. Zum einen können Testaufwände z. B. für das Überprüfen von Wertebereichen von Variablen reduziert werden.

Mit Unittest würde typischerweise das unerwünschte Verhalten durch entsprechende Testvektoren nachgestellt und dadurch abgesichert. Mittels ARCADE kann dies effizienter erfolgen(siehe 4.6.2). Andere Tests, z. B. auf Systemebene können allerdings nicht ersetzt werden. Zum anderen müssen entsprechende Ressourcen darauf verwendet werden die jeweiligen Anforderungen zu formalisieren und die Ergebnisse des Model-Checkings auszuwerten. Weiterhin muss man jedoch beachten, dass der Einsatz von ARCADE es ermöglicht, Fehler zu finden, die durch herkömmliches Testen nicht gefunden würden (siehe 4.6.5).

- *Fehler in der gesamten Kette der Software-Entwicklung sollen gefunden werden können.*

  Balakrishnan et al. beschreiben in [12] die Wichtigkeit, Analysen von Software direkt auf Basis des Maschinencodes durchzuführen. ARCADE nutzt den Binär-Code und kann somit Fehler, die bei der Umsetzung der Anforderungen in eine Architektur während der Programmierung in C und auch durch den Compilier-Vorgang auftreten, ermitteln. Dieses Vorgehen schließt auch Software-Bibliotheken von Dritten (z. B. Zulieferern) mit ein, die meist in nicht lesbarem Object-Code vorliegen.

- *Die Funktions-Software wird auf eingebetteten Systemen ausgeführt, deren Besonderheiten im Verifikations-Werkzeug berücksichtigt werden sollen.*

  Zu den Besonderheiten bei eingebetteten Systemen zählen neben einer Vielzahl von analogen und digitalen Schnittstellen auch der direkte Zugriff auf die Hardware über Register und das Unterbrechen der Programmausführung durch Interrupts. In ARCADE wird für jeden Mikrokontroller ein entsprechender Simulator entworfen, der die Zugriffe und die Reaktion der Hardware nachstellt. Durch das mögliche Synthetisieren des Simulators anhand einer Hardware-Beschreibung [47] wird das Anpassen des Model-Checkers erleichtert und somit beschleunigt. ARCADE kann aufgrund des Simulators Interrupts, Speicherzugriffe und jegliche vorhandene Peripherie des Mikrokontrollers in das Model-Checking miteinbeziehen.

- *Falls Fehler gefunden werden, sollen diese durch den Entwickler nachvollziehbar dargestellt werden.*

  Je nach Art der Formel kann ARCADE je nach Ergebnis einen Zeugen oder ein Gegenbeispiel generieren. Dieser bzw. dieses zeigt einen Pfad durch das Programm auf, welcher komfortabel durch den Simulator nachvollzogen und somit Fehler lokalisieren kann. Die Darstellung erfolgt dabei sowohl im Kontrollflussgraphen als auch im C- und Assembly-Code.

- *Die Formalisierung der System-Modelle sowie der Eigenschaften soll keine hochgradig spezialisierten Experten erfordern. Spezielle Testabteilungen stehen meist nicht zur Verfügung.*

Mit der Evaluierung in Abschnitt 4.4 konnte gezeigt werden, dass auch Nicht-Experten an das Thema „Formale Spezifikation von Anforderungen“ herangeführt werden können. Die Formalisierung der System-Modelle entfällt bei dem Binär-Code Model-Checking mit ARCADE.

- *Der Verifikationsprozess soll in vorhandene Entwicklungsprozesse integriert und automatisiert werden können.*

  ARCADE bietet diverse Schnittstellen, um in vorhandene Prozessstrukturen integriert werden zu können. Neben der Verwendung des Eclipse RCF kann es auch über Kommandozeilenparameter automatisiert angesteuert werden. Dabei werden Formeln und Modelle getrennt in Dateien abgelegt und können somit durch andere Prozessschritte als Eingabe zur Verfügung gestellt werden. Eine Integration in einen vorhandenen Entwicklungsprozess wird in Kapitel 5 beschrieben.

In der vorangehenden Evaluierung des Model-Checkings von mehreren Anforderungen am Beispiel eines Programms zur Überwachung von verschiedenen Recheneinheiten konnten sowohl Fehler im Programmcode lokalisiert als auch das Einhalten von Anforderungen gezeigt werden. Nach dem Beheben der einzelnen Fehler konnten auch die dazugehörigen Anforderungen erfolgreich verifiziert werden. Alle Ergebnisse sind in Tabelle 4.2 dargestellt.

Dass speziell Interrupts einen großen Einfluss auf den Zustandsraum haben, zeigt das Ergebnis der Stack-Analyse in den beiden Software-Versionen **.Error** und **.fixed2**. In **.Error** sind keine Interrupts aktiv, und ARCADE kann den gesamten Zustandsraum durchlaufen.

Bei der Formulierung der Anforderungen mit *CTL* müssen speziell die Eigenschaften der Implikation beachtet werden. Zwar kann diese umgangssprachlich als „wenn-dann“ Beziehung verwendet werden, dabei darf allerdings nur der AG-Operator eingesetzt werden. Bei AF besteht direkt im allerersten Zustand die Möglichkeit, dass die Formel gültig ist, und damit der Model-Checking Prozess beendet wird.

Bei der Verifikation einiger Anforderungen mittels ARCADE ist aufgefallen, dass dessen große Flexibilität in der Beschreibung der Anforderungen mehr manuelle Arbeit im Vergleich zu darauf spezialisierten Werkzeugen benötigt. Z. B. können die Zugriffe auf Arrays oder die Wertebereiche von Variablen für den Entwickler komfortabler durch statische Analysen auf C-Code Ebene (siehe z. B. PC-Lint oder Absint aus Tabelle 2.2) überprüft werden. Im Vergleich zu ARCADE sind die spezialisierten Werkzeuge zur statischen Analyse meist nicht einfach mit zusätzlichen frei formulierbaren Bedingungen erweiterbar. Dies umfasst z. B. die Abhängigkeit einer Formel von einer bestimmten Position im Programmablauf oder von Werten anderer Variablen.

5 Erweiterung eines KMU-orientierten Entwicklungsprozesses

Um die bisher entwickelten formalen Methoden in einem *KMU* einzusetzen, bedarf es eines entsprechenden Entwicklungsprozesses. Dieser muss abgestimmt sein auf die Anforderungen (siehe Unterabschnitt 2.10.1), die sich aus der automotiven Domäne ergeben. Dazu werden im Folgenden die Erweiterungen des bestehenden Prozessmodelles aus Unterabschnitt 2.10.2 durchgeführt und diskutiert.

5.1 Erweiterung um formale Methoden

Im Folgenden soll nun der in Unterabschnitt 2.10.2 beschriebene Entwicklungsprozess um die in dieser Arbeit entwickelten formalen Methoden erweitert werden. Zum einen ist dies die Ressourcenabschätzung in der frühen Phase eines Projektes auf Basis von Uppaal und zum anderen die formale Verifikation des kompilierten Quelltextes mittels Arcade. Dazu wird als erstes der Prozess entsprechend angepasst und als zweites ein Werkzeug für das Verwalten von Varianten beschrieben, um nachfolgend das Design-Framework (siehe Abbildung 2.10) zur Unterstützung des Entwicklungsprozesses um die entsprechenden Werkzeuge zu erweitern.

5.1.1 Erweiterung um Einplanbarkeitsanalyse

In dem von Reke [93] beschriebenen Entwicklungsprozess werden die *Konzepterstellung* sowie die *Softwareerstellung* dahin gehend erweitert, dass die oben genannten formalen Methoden passend eingesetzt werden können. Abbildung 5.1 zeigt den angepassten Teilprozess der Konzepterstellung inklusive der Anpassung, welche als grüner Kasten dargestellt ist. Nachdem mit dem Kunden durch einen Workshop zur Anforderungserfassung ein gemeinsames Problemverständnis aufgebaut worden ist, beschreibt dieser Teilprozess die Erarbeitung der jeweiligen Konzepte, welche als Grundlage für die folgenden Schritte dienen. Dabei berücksichtigt die Konzepterstellung sowohl das Gesamtsystem als auch die Hard- und Software-Entwicklung. Aus den *Kundenanforderungen* werden zunächst Anforderungen an das System abgeleitet, um daraus die Systemarchitektur zu entwerfen. Nachfolgend werden die Systemanforderungen in Anforderungen für Hard- als auch Software abgeleitet und jeweils ein Design (Hardwarebeschreibung bzw. Softwarearchitektur)

parallel entwickelt. Die Ergebnisse werden in einer *Konzeptdokumentation* zusammengefasst und einem Review-Prozess unterzogen, an dem auch der Kunde teilnimmt. Als Ergebnis eines erfolgreichen Reviews steht die *Konzept Baseline*, die als Grundlage für die weitere Umsetzung des Systems dient und auch zur vertraglichen Absicherung des *KMU* gegenüber dem Kunden verwendet werden kann.

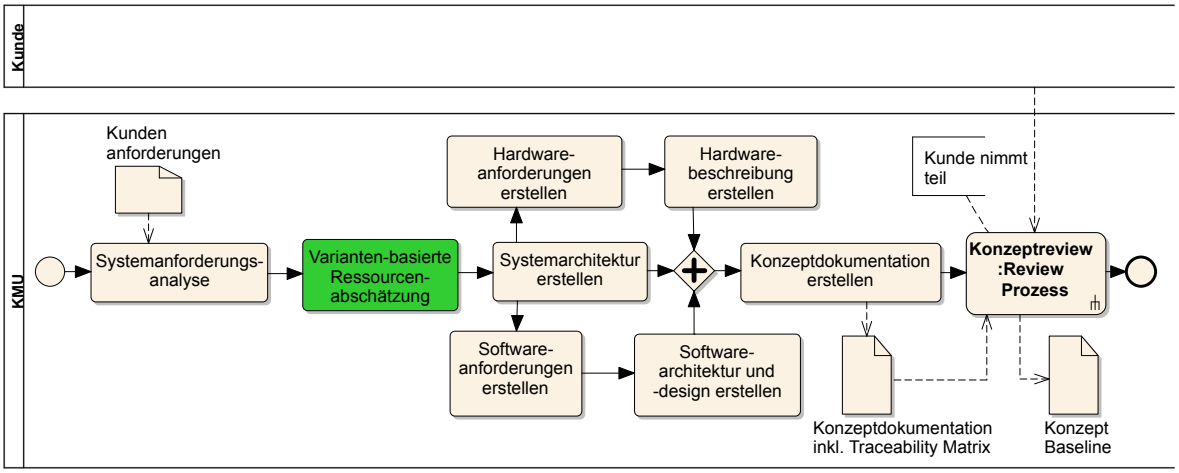


Abbildung 5.1: Prozess zur Konzepterstellung aufbauend auf [93]

Um die formale Methode aus Kapitel 3 in diesen Teilprozess zu integrieren, wird, nachdem die Anforderungen an das System feststehen, eine Abschätzung der Ressourcen auf Grundlage einer zum System passend abgeleiteten Variante durchgeführt. Dazu wird eine Einplanbarkeitsanalyse der verschiedenen Software-Tasks im System auf Basis einer ausgewählten Mikrokontroller-Architektur durchgeführt. Deren Ergebnisse können dazu verwendet werden, um das Hard- und Software-Design zu beeinflussen, indem z. B. eine Vorauswahl des Mikrokontrollers angepasst wird oder Software-Komponenten neu entwickelt werden müssen. Unabhängig vom Entwicklungsprozess erfordert dies den Aufbau von Metadaten (Informationen über die unterschiedliche Software-Module, Tasks, Laufzeiten, Deadlines, etc.) für das jeweilige Produkt, aus dem Varianten abgeleitet werden.

5.1.2 Erweiterung um Binär-Code Verifikation

Der zweite anzupassende Teilprozess ist die *Softwareerstellung*. Dieser ist samt Modifikation in Abbildung 5.2 dargestellt. Im Vergleich zu großen Zulieferern wie z. B. Continental oder BOSCH, führen *KMU* typischerweise kleinere Projekte durch, deren Ergebnis eines, der in der Automobilindustrie üblichen, Prototypenmuster A, B oder C ist. Je nach Muster werden dabei unterschiedliche Phasen des Entwicklungsprozesses anders ausgeprägt.

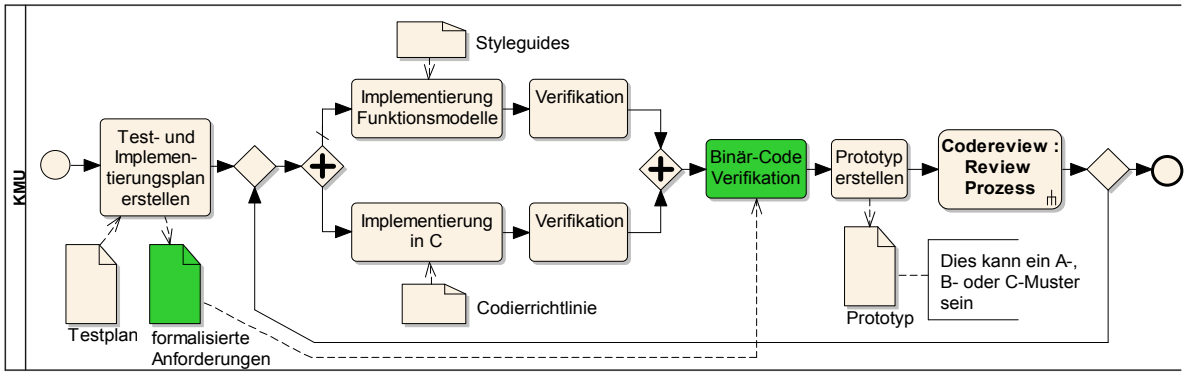


Abbildung 5.2: Prozess zur Softwareerstellung aufbauend auf [93]

Nachdem in der vorausgegangenen *Konzepterstellung* das Design für das System entwickelt wurde, wird als erster Schritt in der *Softwareerstellung* der *Test- und Implementierungsplan* auf Basis des Konzepts erstellt. Dabei ist der ursprüngliche Prozess ausgelegt auf modellbasierte Software-Entwicklung. Daher folgt auf die Erstellung des Testplans eine Aufspaltung der *Implementierung*. Zum einen werden Funktionsmodelle mit entsprechenden Modellierungswerkzeugen (z. B. Mathworks Simulink oder dSPACE TargetLink) erzeugt, aus denen automatisch C-Code abgeleitet werden kann. Zum anderen wird manuell C-Code für andere Module des Systems erzeugt. Beide Implementierungen werden durch entsprechende *Codierrichtlinien* bzw. *Styleguides* unterstützt und durch einen generischen *Verifikationsprozess* auf Korrektheit in Bezug auf die Spezifikation hin überprüft. Anschließend werden beide Implementierungen zusammengeführt, um daraus einen Prototyp (A-, B- oder C-Muster) zu erstellen. Mittels Codereview wird die Qualität der Software geprüft, und bei Bedarf werden entsprechende Korrekturmaßnahmen eingeleitet. Diese erfordern ein Rückspringen im Prozess hin zur Implementierung.

Die Erweiterung um die formale Methode aus Kapitel 4, welche grün dargestellt ist, gilt sowohl für die Implementierung in C als auch für einen möglichen modellbasiert entwickelten Teil. Die Erweiterung umfasst die Verifikation von Binär-Code auf Basis von *formalisierten Anforderungen*, unter der Bedingung, dass ein entsprechend unterstützter Mikrokontroller im Projekt verwendet wird. Dieser führt nach der Implementierung die Verifikation des entwickelten Binär-Codes durch und weist den Entwickler frühzeitig auf gefundene Fehler hin.

5.2 Framework für das Varianten-Management

Um die Umsetzung der neuen Bestandteile des Entwicklungsprozesses komfortabel zu fördern, wurde ein Framework entwickelt, welches die Verwaltung von Varianten sowie die Abschätzung von Ressourcen konsistent unterstützt. Weiterhin ist die Verifikation des Binär-Codes der entstandenen Variante hinsichtlich ihrer Systemeigenschaften in dieses Framework integriert. Abbildung 5.3 zeigt das entstandene Framework.

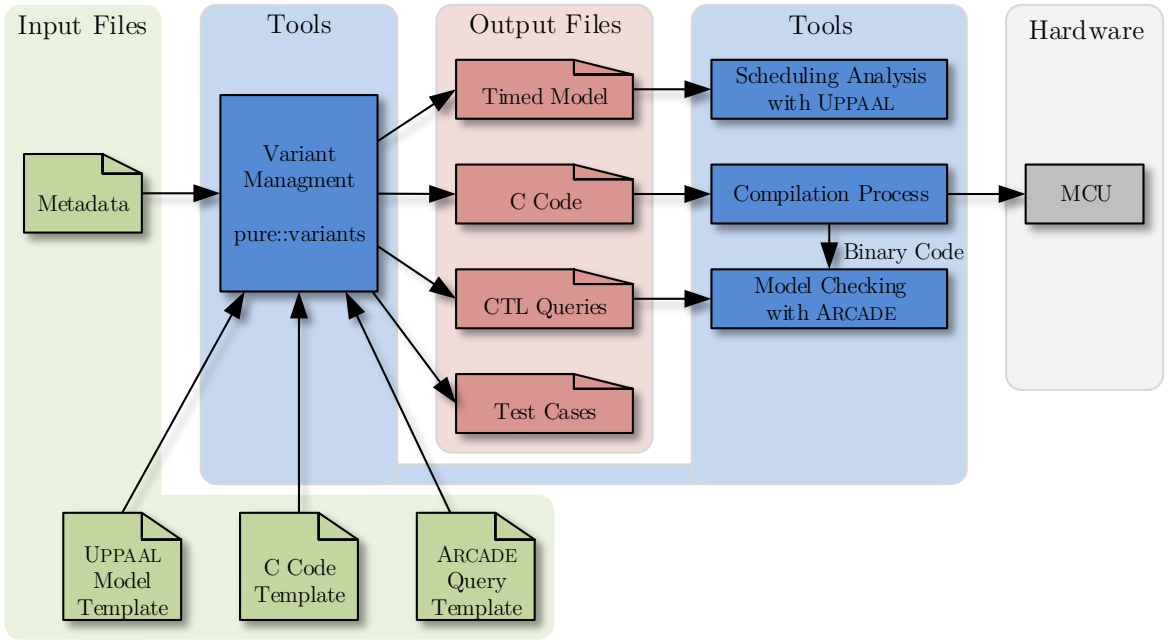


Abbildung 5.3: Varianten-Management-Framework nach [46, 89, 94] und [GKKK11]

Kern des Frameworks bildet das pure::variants Werkzeug, welches für die Verwaltung der Varianten eingesetzt wird. Die Eingabedaten, die für diese notwendig sind, müssen durch den Entwickler bereitgestellt werden. Hierzu zählen folgende Dateien:

- *Metadata* liegt im XML Format vor und beinhaltet alle Informationen bezogen auf alle vorhandenen Features. Neben Namen und Abhängigkeiten zu anderen Features sind auch C-Code Fragmente in dieser Datei abgelegt. Diese Fragmente werden an den entsprechenden gemeinsamen Stellen der Produktlinien-Architektur im Quelltext für den späteren Generierungsprozess eingefügt. Zusätzlich sind für einige Features Modellparameter für die Einplanbarkeitsanalyse hinterlegt sowie Formeln für die Verifikation des Binär-Codes.

- UPPAAL *Model Template* beinhaltet die in Unterabschnitt 3.1.5 erstellten generischen UPPAAL Modelle zur Einplanbarkeitsanalyse.

- *C Code Template* stellt das Grundgerüst der Produktlinien-Architektur inklusive der gemeinsamen Stellen, an denen die einzelnen Features ihren eigenen Quellcode einfügen können, um mit dem Gesamtsystem sowie den anderen ausgewählten Features interagieren zu können.

- ARCADE *Query Template* beinhaltet das Grundgerüst einer ARCADE Formel-Datei im XML Format. Für jedes Feature können mehrere eigene Formeln passend zu den Feature-Eigenschaften in den *Metadata* definiert werden. Diese werden mit

diesem Grundgerüst zusammengefügt, um anschließend zur Verifikation mittels Model-Checking genutzt zu werden.

Das Varianten-Management verarbeitet diese Vorlagendateien sowie die Meta-Daten abhängig von der Variante, welche aus einer Auswahl an vordefinierten Features besteht. Die resultierenden Dateien dienen als Eingabe für die verschiedenen Werkzeuge. Dabei wird ein Modell der Tasks (*Timed Model*) zur Einplanbarkeitsanalyse (Unterabschnitt 3.1.5) mittels UPPAAL verwendet. Weiterhin wird der erstellte Quellcode (*C Code*) kompiliert und kann danach sowohl auf den Mikrokontroller (*MCU*) programmiert als auch für die Binär-Code-Verifikation (Abschnitt 4.2) mit ARCADE genutzt werden. Dazu wird eine Formeldatei (*CTL Queries*) verwendet. Als Letztes werden Testfälle der einzelnen Module der erstellten Variante zentral in einer Datei (*Test Cases*) abgelegt. Diese enthält Spezifikationen zu den Testfällen für die jeweiligen Features, die manuell überprüft werden können.

5.3 Integration in das Design Framework

Im Folgenden wird der in Abschnitt 5.1 angepasste Entwicklungsprozess (Abbildung 5.4) sowie das Varianten-Management durch ein weiteres Framework unterstützt. Das Design Framework (*DF*) ergänzt die Werkzeuglandschaft um Hilfsmittel zur Unterstützung des Prozessmanagements. Hierbei stehen speziell die Aspekte der Nachvollziehbarkeit von Entwurfsentscheidungen, die Wiederverwendbarkeit von Teilentwürfen sowie die Konsistenz von Modellen im Vordergrund.

Die Nachvollziehbarkeit von Entscheidungen während der Entwicklung ist deshalb wichtig, da dieses Wissen bei anderen Projekten mit ähnlichen Fragestellungen Zeit und somit Kosten sparen kann. Weiterhin sollen auch Mitarbeiter, die nicht am Entscheidungsprozess beteiligt gewesen sind, diesen im Nachhinein einfach nachvollziehen und verstehen können, um zum einen die Akzeptanz der Entscheidung zu verbessern und zum anderen die Wiederverwendbarkeit zu erhöhen. Weiterhin sollte es für Mitarbeiter des Projektes möglich sein, bei Fehlern, deren Ursache in einer Entwurfsentscheidung liegen, die Entscheidung entsprechend anpassen zu können. Dies setzt ein einfaches Nachvollziehen der Entscheidung voraus. Da Entscheidungen meist an Modelle und dazu definierte Parameter gekoppelt sind, ist es notwendig, diese konsistent und nachvollziehbar zu hinterlegen.

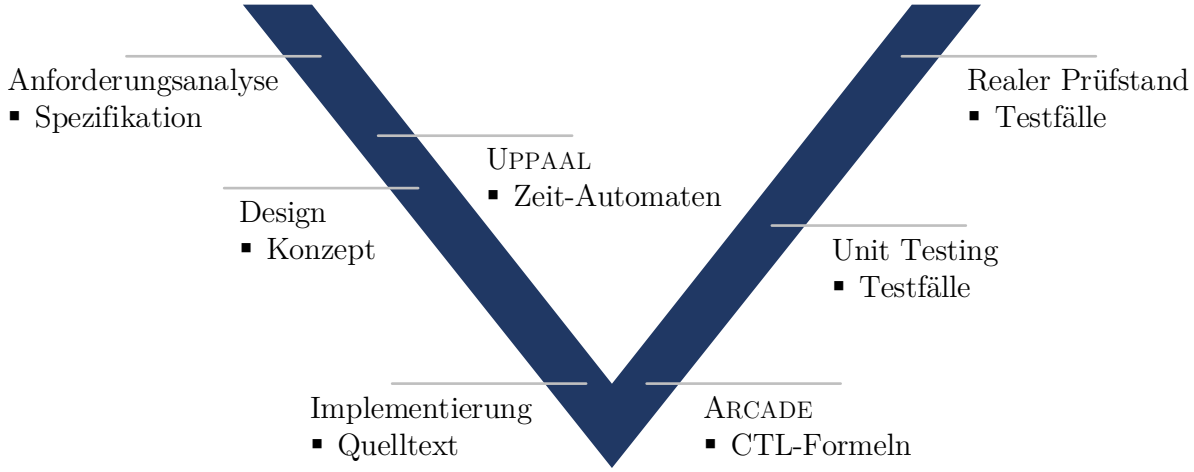


Abbildung 5.4: V-Modell

Um dieses Ziel zu erreichen, wurde der in Abschnitt 5.1 beschriebene Entwicklungsprozess in das *DF* (beschrieben in Unterabschnitt 2.9.1) in Hüfner et al. [HSEG13] integriert. Dazu wurden auch die Werkzeuge des Varianten-Frameworks (Abbildung 5.3), welches auf pure::variants [16, 17, 87] basiert, auf das Eclipse-basierte *DF* übertragen. Die zur formalen Verifikation notwendigen Werkzeuge Uppaal und Arcade wurden durch die Schnittstelle zum Aufruf von Skripten angebunden. Somit konnten die formalen Teilprozesse im Entwicklungsprozess durch ein zentrales Werkzeug automatisiert werden. Die einzelnen Entwurfsschritte wurden mithilfe der Eclipse-Umgebung modelliert. Abbildung 5.5 zeigt das Ergebnis eines exemplarisch durchgeführten Projektes auf Basis der PCC-Plattform (Abschnitt 2.7). Im Folgenden wird der Ablauf des Beispielprojektes, welches in Abbildung 5.5 dargestellt ist, basierend auf [HSEG13] beschrieben:

Requirements Analysis: In diesem Schritt werden die Anforderungen des Kunden z. B. während eines gemeinsamen Workshops erfasst und in einem natürlichsprachlichen Textdokument abgelegt.

System Design: Auf Basis der Anforderungen wird ein Systementwurf erarbeitet, als Dokument abgelegt und mit diesem Entwurfsschritt verknüpft. Dabei können neben Software-Konzepten auch Hardware-Beschreibungen angefertigt werden.

Feature Selection: Passend zum Software-Konzept wird im Varianten-Framework eine Variante für das zu entwickelnde System abgeleitet. Die dazu notwendigen Bibliotheken und Metadaten für pure::variants werden aus dem *Core Domain Knowledge* des *DF* verwendet. Das Modell der Variante kann über den entsprechenden Eclipse-Editor von pure::variants erstellt werden. Das Resultat wird in der Datenablage des *DF* zur weiteren Verwendung gespeichert.

Timing Verification: Mit der im vorangehenden Entwurfsschritt erstellten Variante kann nun die Einplanbarkeitsanalyse mittels UPPAAL durchgeführt werden. Dazu werden die entsprechend generierten Metadaten der Variante als Eingabewerte für das Modell des Zeitautomaten verwendet. UPPAAL kann automatisch über die Skript-Schnittstelle mit dem notwendigen Modell und Parametersatz ausgeführt werden. Das Ergebnis dieses Experiments wird entsprechend im *DF* gespeichert. Führt ein Experiment nicht zu dem gewünschten Ergebnis, so kann dieser Schritt als „Totes Ende“ gesehen werden und ein neuer Entwurfsschritt als Alternative mit angepassten Parametern und Modellen verfolgt werden. Zum Beispiel wurde in *Timing Verification 1* festgestellt, dass die erstellte Variante nicht den Zeitanforderungen genügt. In *Timing Verification 2* wurde daraufhin in einem Experiment eine optimierte Variante untersucht. Dessen Resultat zeigte ein Erfüllen der Anforderungen an das Zeitverhalten und wurde damit zur Basis für die weitere Entwicklung.

Compile: Mit dem C-Code der erstellten Variante wird ein Prototyp der Software erstellt und mittels *Compiler* in Binär-Code kompiliert und in der Datenablage mit Verknüpfung zum aktuellen Entwurfsschritt abgespeichert.

Verification: Auf Basis der im Varianten-Framework erstellten CTL-Formeln wird eine Verifikation des Binär-Codes mit Hilfe von ARCADE durchgeführt. Dazu wird über die Skript-Schnittstelle der Binär-Code und eine Formeldatei an ARCADE übergeben und der Model-Checking Prozess gestartet. Die zurückgelieferten Ergebnisse werden in Abhängigkeit des Entwurfsschritts im *DF* abgelegt.

SW Test: Das erstellte Software-System wird mit den aufgrund der Variante erstellten Testfällen geprüft. Die Ergebnisse werden in einem entsprechenden Testprotokoll festgehalten und im *DF* zur Dokumentation des Entwurfsschritts abgelegt.

Test Bench: Im Vergleich zum vorangegangenen Schritt wird am Prüfstand nicht die Software einzeln geprüft, sondern im Zusammenhang mit der entsprechenden Hardware und anderen notwendigen Komponenten. Die Ergebnisse werden in einem entsprechenden Testprotokoll festgehalten und im *DF* zur Dokumentation des Entwurfsschritts abgelegt.

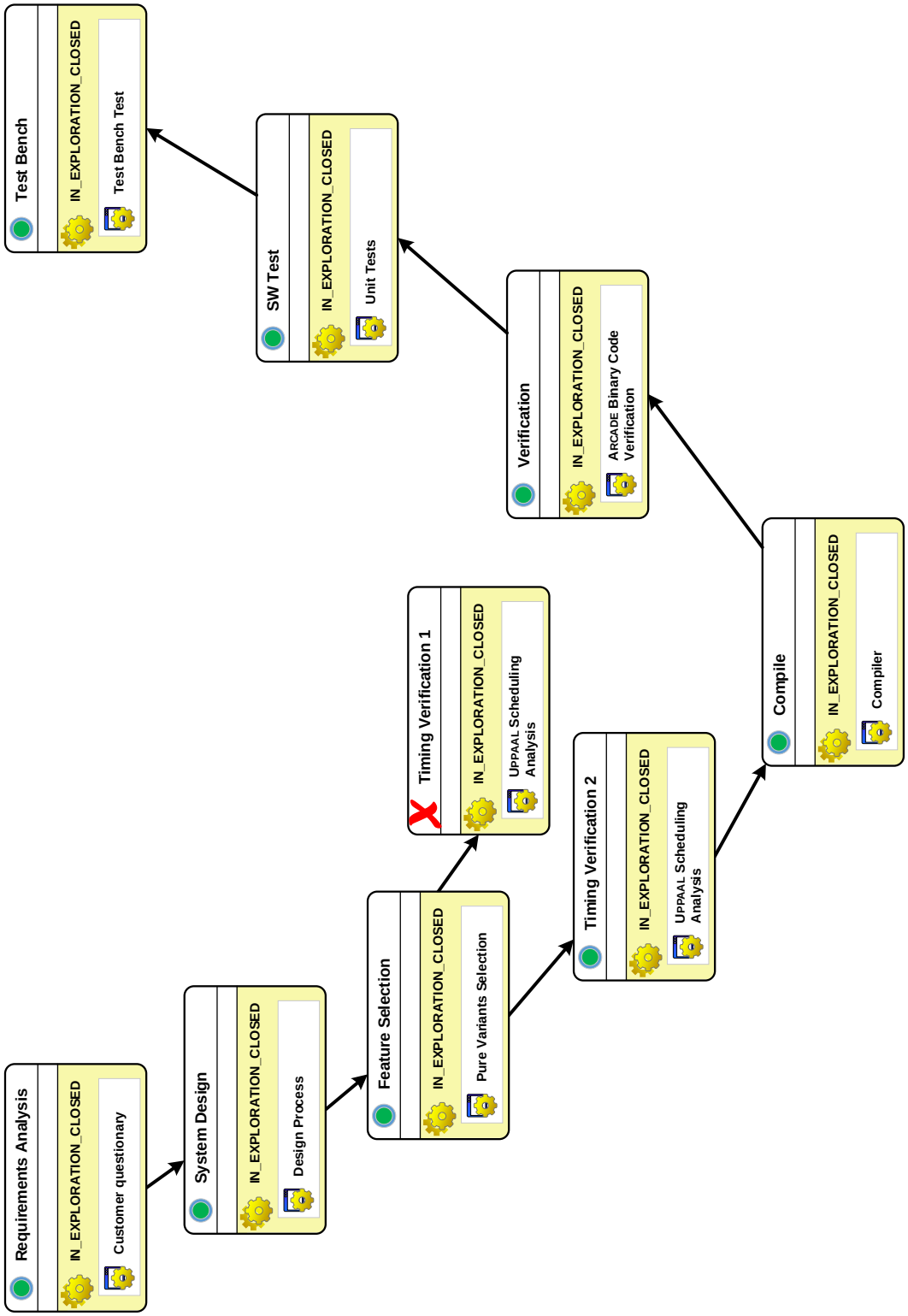


Abbildung 5.5: Umsetzung des Entwicklungsprozesses im Design-Framework basierend auf [HSEG13]

5.4 Fazit

Die beiden Ansätze aus Kapitel 3 und Kapitel 4 wurden in den Entwicklungsprozess eines *KMU* integriert. Dazu wurde zum einen die Phase der Konzepterstellung um die Abschätzung der benötigten Ressourcen auf Basis der ausgewählten Variante erweitert. Zum anderen wurde der Prozess der Softwareerstellung um die Verifikation des erzeugten Binär-Codes ergänzt. Zur komfortablen Unterstützung der Verwaltung der erstellten Produktlinie wurde ein Framework auf Basis von pure::variants entwickelt, welches auch die beiden entwickelten formalen Ansätze zur Verifikation enthält. Damit können Varianten auf Basis der vorgestellten Steuergeräte-Plattform erzeugt und die entsprechenden Verifikationsprozesse durchgeführt werden. Abschließend wurde ein Framework zur Unterstützung des Prozessmanagements an den vorgestellten Entwicklungsprozess angepasst und um das zuvor entwickelte Werkzeug ergänzt. Damit kann der Projektverlauf dokumentiert und die dabei entstehenden Modelle sowie Ergebnisse von Prüfungen dieser Modelle konsistent gespeichert werden. Alle Entwurfsentscheidungen sind transparent nachvollziehbar und können projektübergreifend zur Steigerung von Qualität und Effizienz genutzt werden.

6 Zusammenfassung und Ausblick

Im Folgenden werden die Ergebnisse dieser Arbeit zusammengefasst, in einen Zusammenhang gestellt und abschließend bewertet. In dem darauffolgenden Unterkapitel wird ein Ausblick auf mögliche, weiterführende Arbeiten gegeben.

6.1 Zusammenfassung

Ziel dieser Arbeit war es, die Software-Entwicklung in kleinen und mittleren Unternehmen durch Methoden der formalen Verifikation zu verbessern und zu unterstützen. Zum einen sollte eine Methodik entstehen, die es möglich macht, Entwicklungsprozesse durch formale Verifikationstechniken zu ergänzen und somit Softwaretests zu unterstützen. Zum anderen sollte eine Methodik entwickelt werden, die für automotive Serien-Steuergeräte eine möglichst präzise Abschätzung des Ressourcenbedarfs in Bezug auf die benötigte Rechenzeit erlaubt. Dazu wurden zwei Ansätze zum Einsatz von formalen Methoden vorgestellt.

Der erste Ansatz ist die Analyse von Zeitanforderungen für eingebettete Systeme, basierend auf der formalen Methode des Model-Checkings. Dabei stehen speziell Systeme, die nach dem Baukastenprinzip entworfen sind, im Vordergrund. Hierzu wird mit Hilfe von Uppaal und der Modellierung durch Zeitautomaten ein System zur Einplanbarkeitsanalyse eines Task-Systems vorgestellt. Als Grundlage der Analyse wurde ein generisches Task- sowie ein Scheduler-Modell entwickelt. Diese Modelle sind parametrierbar und können somit an unterschiedliche Task-Zusammenstellungen angepasst werden. Abschließend wurde eine Evaluierung anhand des Fallbeispieles eines automotiven Kleinserien-Steuergerätes durchgeführt. Dabei konnte gezeigt werden, dass die durch das Model-Checking berechneten maximalen Task-Ausführungszeiten mit dem realen Systemverhalten übereinstimmten. Um diese Analyse zur frühzeitigen Ressourcen-Abschätzung in der Praxis zu verwenden, entstand ein Werkzeug zur Verwaltung von Varianten auf der Grundlage von pure::variants. Basierend auf Kundenwünschen kann somit aus einem Komponenten-Baukasten eine Software für ein mögliches Steuergerät zusammengestellt und, aufbauend auf die vorgestellte Einplanbarkeitsanalyse, Entscheidungen für die notwendige Hardware getroffen werden. Damit kann frühzeitig das Risiko für das Auftreten von Timing-Problemen bei der Steuergeräteentwicklung reduziert werden. Dabei haben *kleine und mittlere Unternehmen (KMU)* die Vorteile, dass deren ausgeprägte Nähe zum Kunden dadurch zusätzlich gestärkt wird und dass die Systemkomplexität deren Projekte oft überschaubarer ist als bei großen Unternehmen. Zur Evaluierung des Werkzeugs

zur Verwaltung von Varianten wurde die bestehende Software-Plattform des Fallbeispiels evolutionär in eine Produktlinie umgewandelt und in das Werkzeug überführt. Dazu wurden in der Software entsprechende Features identifiziert, Gemeinsamkeiten und Variabilitäten im Quellcode analysiert und eine Umstrukturierung des Systems auf die erarbeitete Produktlinien-Architektur durchgeführt. Die zur Evaluierung erzeugte Variante glich dem Ursprungssystem, und abschließende Tests zeigten identisches Systemverhalten. Zusammengefasst wurde eine Methodik erarbeitet, welche es Entwicklern und Produktverantwortlichen ermöglicht, eine Abschätzung des zeitlichen Systemverhaltens bei einer speziellen Task-Zusammenstellung zu erhalten. Dies erfolgt auf Basis von vorangegangenen Projekten und mit Wissen über das Verhalten von neu entstehenden Software-Modulen. Das hohe Maß an Flexibilität und Wirtschaftlichkeit der *KMU* wird dadurch weiter gestärkt. Weiterhin können sie sich damit stärker auf die individuellen Kundenwünsche fokussieren und vorhandene Komponenten effizient und mit hoher Qualität wiederverwenden.

Der zweite Ansatz zur Verbesserung der Software-Qualität durch den Einsatz von formalen Techniken befasst sich mit der Verifikation von Programmcode eingebetteter Systeme. Dabei sollten neben Fehlern in der Funktionssoftware auch Fehler bei der Verwendung von hardwarenahen Funktionen eines Mikrokontrollers gefunden werden können sowie Fehler, die durch den Compiler entstehen. Nach einer Analyse von möglichen Werkzeugen fiel die Wahl auf den Binär-Code Model-Checker ARCADE. Für den Model-Checking Prozess durch ARCADE ist neben dem kompilierten Programmcode eine formalisierte Form der zu verifizierenden Anforderung notwendig. Hierzu können in dem Werkzeug Anforderungen mit Hilfe von *Computational Tree Logic (CTL)* bzw. *Safety-Automaten (SA)* formal dargestellt werden. Mittels einer nicht-repräsentativen Umfrage unter Software-Entwicklern wurden die Vor- und Nachteile der beiden Möglichkeiten im industriellen Einsatz herausgearbeitet. Die Ergebnisse zeigen, dass *CTL* ausdrucksstärker ist, allerdings auch komplexer in der Erstellung. *SA* sind intuitiver in der Anwendung aber eingeschränkter in dem, was ausgedrückt werden kann. Die durchgeführte Umfrage zeigte, dass auch ungeschulte Teilnehmer durch eine entsprechende Einführung in beide Techniken diese erlernen und korrekt anwenden können. Bei speziellen Anforderungen sollte jedoch ein Experte hinzugezogen werden. Die grafische Modellierung von Anforderungen durch *SA* ist ein sinnvoller Ansatz, um den Einstieg bzw. den Umgang mit formaler Spezifikation zu erleichtern. Ein Vorteil dieser grafischen Eingabe von Anforderungen liegt in der visuellen Gruppierung von Bedingungen. Würde man dies auf die textuelle Darstellung von logischen Formeln übertragen, könnten Formeln kognitiv schneller erfassbar werden. Mit diesem Wissen wurde der Prozess der Formalisierung von Anforderungen untersucht und eine dabei unterstützende Methodik erarbeitet. Bei der Evaluierung des Model-Checkings an einem industriellen Fallbeispiel konnten sowohl Fehler im Programmcode lokalisiert als auch das Einhalten von Anforderungen gezeigt werden. Nach dem Beheben der einzelnen Fehler konnten die dazugehörigen Anforderungen erfolgreich verifiziert werden. Jedoch konnten speziell Anforderungen, die den Aufbau des gesamten Zustandsraumes erfordern, nur eingeschränkt geprüft werden. Der unter anderem durch Interrupts vergrößerte Zu-

standsraum verhinderte z. B. die Ermittelung der maximalen Stack-Größe auf Standard PC Hardware in einem vorgegebenen Zeitfenster von einer Stunde. Zusammenfassend kann festgehalten werden, dass der Einsatz von Binär-Code-Verifikation in eingebetteten Systemen den Testaufwand reduzieren, allerdings nicht ersetzen kann. Dabei müssen Entwickler die Größe des Zustandsraumes, Aufwände für die Formalisierung der Anforderungen sowie die korrekte Interpretation der Ergebnisse berücksichtigen. Dadurch relativieren sich die aus KMU-Sicht geforderten Einsparungen. Vorteil für Unternehmen ist der Aspekt, dass mit dieser Methode das Ausbleiben von Fehlern bewiesen werden kann, was durch herkömmliches Testen nicht möglich ist.

Abschließend wurden die beiden vorgestellten Ansätze zur Verwendung der jeweiligen formalen Methode zusammen in den Entwicklungsprozess eines *KMU* integriert. Dazu wurde zum einen die Phase der Konzepterstellung um die Abschätzung der benötigten Ressourcen auf Basis der ausgewählten Variante erweitert. Zum anderen wurde der Prozess der Softwareerstellung um die Verifikation des erzeugten Binär-Codes ergänzt. Zur komfortablen Unterstützung der Verwaltung der erstellten Produktlinie wurde ein Framework auf Basis von pure::variants entwickelt, welches auch die beiden entwickelten formalen Ansätze zur Verifikation enthält. Damit können Varianten auf Basis der vorgestellten Steuergeräte-Plattform erzeugt und die entsprechenden Verifikationsprozesse durchgeführt werden. Abschließend wurde ein Framework zur Unterstützung des Prozessmanagements an den vorgestellten Entwicklungsprozess angepasst und um das zuvor entwickelte Werkzeug ergänzt. Damit kann der Projektverlauf dokumentiert und die dabei entstehenden Modelle sowie Ergebnisse von Prüfungen dieser Modelle konsistent nachverfolgt werden. Alle Entwurfsentscheidungen sind transparent nachvollziehbar und können auch projektübergreifend zur Steigerung von Qualität und Effizienz genutzt werden.

Insgesamt wurde ein Ansatz zur Integration von formalen Methoden in Entwicklungsprozesse von kleinen und mittleren Unternehmen vorgestellt, erfolgreich mit entsprechender Werkzeugunterstützung umgesetzt und evaluiert. Mit den gezeigten Methoden ist es möglich, den Testaufwand zu reduzieren und die Qualität von automobilen Steuerungssystemen schon frühzeitig in den wichtigen Phasen der Entwicklung zu steigern.

6.2 Ausblick

Die Modelle zur Abschätzung der benötigten Rechenzeit basieren aktuell auf Einprozessorsystemen. Um diese für zukünftige Generationen von Mikrokontrollern auszulegen, kann das Scheduler-Modell um eine Option zur Verwendung von multiplen Kernen erweitert werden. Weiterhin kann zur Optimierung der Ergebnisse der Einplanbarkeitsanalyse ein Modell ergänzt werden, welches das Auftreten von Interrupts realitätsnäher abbildet. Damit könnte eine genauere Abschätzung über die Auslastung des Systems erzielt werden.

Ein Parameter für die Abschätzung der Rechenzeit beschreibt die maximale Ausführungszeit eines Tasks (*Worst Case Response Time*). In dieser Arbeit wurde ein Wert

für die *Worst Case Response Time (WCRT)* dadurch ermittelt, dass der entsprechende Task auf der Ziel-Hardware ausgeführt und dessen gemessene Ausführungszeit über eine entsprechende Dauer analysiert wurde. Diese Methodik garantiert jedoch nicht, dass die gemessene maximale Ausführungszeit auch gleich der theoretisch möglichen maximalen Ausführungszeit ist. Um dieses Verfahren zu optimieren, können die gemessenen Werte für *WCRT* ersetzt werden durch genauere Ergebnisse aus z. B. statischen Analysen durch entsprechend verfügbare Werkzeuge.

Um den Umgang von Softwareentwicklern mit der Verifikation von Binär-Code zu verbessern, kann es hilfreich sein, eine Bibliothek von Entwurfsmustern oder auch Beispielen für die Umwandlung von umgangssprachlich formulierten Anforderungen in eine formale Form aufzubauen. Hierzu könnte man mit einer Aufteilung in generische und projektspezifische Anforderungen starten und diese sowohl in *CTL* als auch als *SA* ablegen. Durch die projektübergreifende Verwendung können diese kontinuierlich ergänzt und optimiert werden. Weiterhin könnte ein Leitfaden den Prozess der Umwandlung unterstützen, indem er Hinweise und Methoden zur Verfügung stellt, um den Aufwand der Formalisierung zu reduzieren und um Fehler zu vermeiden. Außerdem können darin Hilfen zu Interpretation der Ergebnisse aus dem Model-Checking gegeben werden.

Bei der Binär-Code-Verifikation einiger Anforderungen wurde die Komplexität der Umwandlung in eine entsprechende formale Form sowie der Umfang des Model-Checkings deutlich. Falls der gesamte Zustandsraum für den Verifikationsprozess aufgebaut und durchsucht werden musste, reichten sowohl die verfügbare PC-Hardware als auch das definierte Zeitlimit dafür nicht aus. Hierbei könnte das Einbinden von Methoden zur statischen Analyse von C-Code, die Verifikation z. B. von Zugriffen auf Arrays oder die Ermittelung der notwendigen Größe des Stacks effizienter gestalten.

Der beschriebene Entwicklungsprozess enthält neben manuell programmierter auch modellbasiert entwickelte Software. In dieser Arbeit wurden nur die manuell programmierten Software-Modelle in dem Framework zur Unterstützung des Prozessmanagements (*Design Framework*) realisiert. Hier könnte eine Integration des modellbasierten Entwicklungspfades die Prozesse und Werkzeuge vereinheitlichen. Dazu müssten Mathworks Simulink Modelle mit entsprechenden Parametern verwaltet und durch entsprechende Werkzeuge verifiziert werden können.

A Fragebogen zur Umfrage SA vs. CTL

Evaluierung: Vergleich CTL & SA

Version 1.0

VEMAC

Inhaltsverzeichnis

1 Grundlagen

Anforderungen an ein Software-System sind zumeist in textueller Form gehalten. Für die Überprüfung der Software mit Hilfe von formalen Methoden sind allerdings formale Darstellungen der Anforderungen notwendig. Zwei mögliche Beschreibungsformen sind Automaten (SA: Safety-Automaten) und CTL (Computational Tree Logic) Formeln.

In der Informatik bezeichnet der Begriff der „Formalen Methoden“ eine Vielzahl von natur- und ingenieurswissenschaftlichen Techniken zum Modellieren und zur rigorosen Überprüfung von Computersystemen und Softwareprodukten. Formale Methoden basieren in der Regel auf der Verwendung von mathematischer Logik.

1.1 Atomare Eigenschaft

Egal ob CTL oder SA, bei beiden Methoden zur Formulierung von Anforderungen gibt es so genannte atomare Eigenschaften (gekennzeichnet mit Kleinbuchstaben, z.B. x, y, z). Darauf aufbauend können CTL Formeln oder SAs erstellt werden. Typischerweise sind atomare Eigenschaften durch den Vergleich zweier Terme gegeben. Beispiele dafür sind:

x: PC = FunktionA_Anfang

y: VariableX = VariableY

z: VariableY < 1

w: VariableZ > 23

j: VariableW != 42

Hinweis: Die Abkürzung PC im Beispiel bedeutet Programm Counter und zeigt an, welche Zeile Code der Prozessor gerade abarbeitet. Als Vereinfachung können beliebige Label genutzt werden, um Positionen im Code zu adressieren. „FunktionA_Anfang“ zum Beispiel beschreibt die Startadresse von „FunktionA“.

Diese Eigenschaften können durch logische Operatoren (z.B. **∧** [UND]**, ∨** [ODER]**, ¬** [NICHT]) verknüpft werden.

1.2 CTL

CTL ist eine temporale Logik, die speziell zur Spezifikation und Verifikation von Computersystemen dient. Es können Aussagen über zeitliche Programm-Abläufe (Pfade) getroffen werden. Zum Beispiel kann man die Anforderung:

„Die Variable iTemp muss den Wert x annehmen können“

überprüfen, indem man die Aussage wie folgt formuliert:

„Es gibt im Programmcode einen Weg/Pfad, bei dem die Variable iTemp irgendwann den Wert x erreicht.“

Die CTL Formel zu dem Beispiel lautet:

EF (iTemp = x).

Zur Formulierung dieser Aussagen wird eine entsprechende Logik verwendet. CTL Formeln bestehen aus temporalen Verknüpfungen und atomaren Eigenschaften. Die temporalen Verknüpfungen bestehen dabei aus zwei Komponenten, den Pfadquantoren (E, A) und den darauf folgenden Temporaloperatoren (X, F, G, U). Diese temporalen Verknüpfungen bilden Operatoren um zeitliche Bezüge im Programmverhalten aus zu drücken. Beispiele für damit erstellte CTL-Formeln:d

AG x	Egal was das Programm macht, „x“ muss immer gelten.
AF x	Egal was das Programm macht, „x“ muss irgendwann einmal gelten.
EG x	Das Programm kann sich so verhalten, dass „x“ immer gilt.
EF x	Das Programm kann sich so verhalten, dass „x“ irgendwann einmal gilt.
A x U y	Egal was das Programm macht, irgendwann gilt einmal „y“ und bis dahin muss „x“ gelten.
E x U y	Das Programm kann sich so verhalten, dass irgendwann einmal „y“ gilt und bis dahin „x“ galt.
AF ($\vartheta_1 \wedge$ E ϑ_2 U ϑ_3)	Egal was das Programm macht, es muss irgendwann einmal ϑ_1 gelten. Von da an kann sich das Programm so verhalten, dass irgendwann ϑ_3 gilt und bis dahin ϑ_2 galt.

1.3 Safety-Automaten (SA)

Die zweite Variante zur Prüfung auf Korrektheit sind Automaten als Aussageformat. Mit Automaten lassen sich Anforderungen ebenfalls darstellen. Ein Unterschied zu CTL Formeln ist die grafische Darstellungsform.

Um die Funktion von SAs besser erklären zu können, stellen wir uns ein Spielbrett vor, auf dem der Automat aufgezeichnet ist. Eine Münze gilt als Spielfigur. Der Spielablauf symbolisiert den Programmablauf. Der Automat spielt gegen den Programmablauf.

Jeder Automat hat einen Startpunkt, der Startzustand. Dieser ist mit einem Pfeil gekennzeichnet. Die Münze liegt von Beginn des Spiels an auf diesem Zustand (Abbildung 1). Jeder Zustand kann eine atomare Eigenschaft enthalten. Gilt diese, kann die Münze auf dem Zustand verweilen. Gilt diese Eigenschaft aufgrund des Programmablaufes auf einmal nicht mehr, so muss die Münze vom Spielfeld genommen werden.

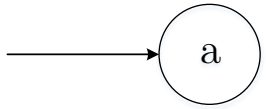


Abbildung 1 Startzustand

Gibt es mehr als einen Zustand, so kann die Münze über Verbindungen (Zustandsübergänge) zwischen den Zuständen in einen anderen Zustand wechseln. Diese können auch mit atomaren Eigenschaften (x, y, z in Abbildung 2) belegt sein. Die Münze muss den aktuellen Zustand verlassen, wenn die Eigenschaft eines Zustandsüberganges erfüllt ist. Ist die Eigenschaft des aktuellen Zustandes nicht erfüllt und gibt es auch keine Zustandsübergänge deren Eigenschaften erfüllt sind, so kann die Münze nirgendwohin wechseln und muss aus dem Spiel genommen werden. Das Spiel läuft so lange weiter, bis alle Münzen aus dem Spiel genommen oder alle möglichen Programmdurchläufe durchgespielt wurden.

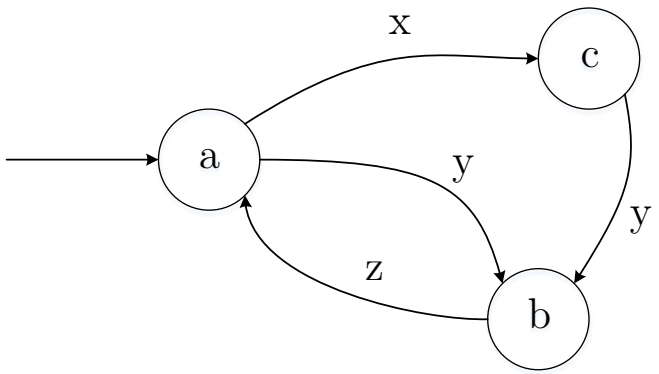


Abbildung 2 Automat mit mehreren Zuständen

Es kann auch vorkommen, dass, wie in Abbildung 3 dargestellt, die Münze den Zustand mit der atomaren Eigenschaft „a“ verlassen muss und dabei zwei Zustandsübergänge möglich sind (die Eigenschaft „x“ gilt gerade). Dann wird die Münze verdoppelt, und beide Folgezustände werden belegt. Fallen beide Münzen dann wieder auf einen Zustand („y“ gilt und beide Münzen wechseln in den letzten Zustand), so wird eine der beiden Münzen vom Spielfeld genommen. Es ist nur eine Münze pro Zustand erlaubt.

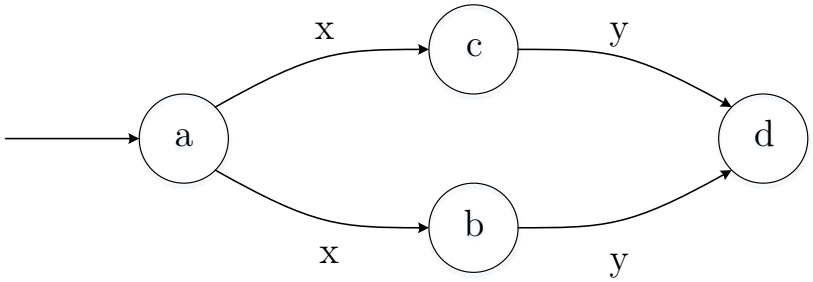


Abbildung 3 Automat mit gleichen atomaren Eigenschaften an den Zustandsübergängen

Es gibt noch zwei weitere Möglichkeiten, wie das Spiel vorzeitig beendet werden kann. Erreicht eine Münze einen sogenannten „Gewonnen-Zustand“ (siehe Abbildung 4) so gilt das Spiel sofort als gewonnen, bzw. bei einem „Verloren-Zustand“ (siehe Abbildung 5) sofort als verloren.

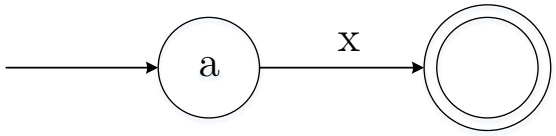


Abbildung 4 Gewonnen-Zustand

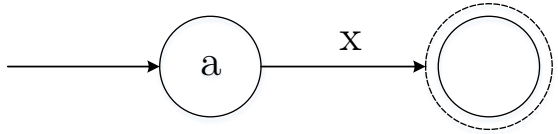


Abbildung 5 Verloren-Zustand

Verliert der Automat das Spiel, gilt die zu zeigende Anforderung als nicht erfüllt. Das Programm erfüllt jedoch garantiert die gestellte Anforderung, wenn der Automat das Spiel gewonnen hat.

2 Evaluierung

2.1 Selbsteinschätzung

Wie viele Jahre Erfahrung haben Sie in der Entwicklung von SW? _____

Wie gut …	**schlecht**	**Gut**				**sehr gut**	**Keine Aussage**
… kenne ich mich mit der Entwicklung von Software grundsätzlich aus?	1	2	3	4	5	6	0
… kenne ich mich mit der Programmierung von Mikrokontrollern aus?	1	2	3	4	5	6	0
… kenne ich mich mit der Spezifikation von Anforderungen aus?	1	2	3	4	5	6	0
… kann ich Anforderungen formal spezifizieren?	1	2	3	4	5	6	0
… kenne ich mich mit Automaten aus?	1	2	3	4	5	6	0
… kann ich Automaten erstellen?	1	2	3	4	5	6	0
… kenne ich mich mit CTL aus?	1	2	3	4	5	6	0
… kann ich Aussagen in CTL formulieren?	1	2	3	4	5	6	0

2.2 Fall 1:

Es soll überprüft werden, ob die Variable „VarX“ immer innerhalb des Bereiches von 100 bis 15000 bleibt. Die Anforderung besagt, dass ein fester Bereich realisiert werden soll und das aus Sicherheitsgründen diese Werte unter keinen Umständen über- bzw. unterschritten werden. Diese Bedingung muss immer gelten.

CTL 1:

SA 1:

2.3 Fall 2:

Es soll geprüft werden, ob das Programm eine Funktion *Fkt_A* aufrufen und komplett durchlaufen kann. Eine Anforderung ist, dass alle relevanten Funktionen erreichbar sein müssen. Hier soll die Erreichbarkeitsanforderung getestet werden, die besagt, dass eine Funktion erreichbar ist. Die entsprechenden Label für die Funktion sind:

Start der Funktion: Fkt_A_START

Ende der Funktion: Fkt_A_ENDE

CTL 2:

SA 2:

2.4 Fall 3:

Es soll geprüft werden, ob das Overflow Register „O“ gesetzt wird („O = 1“), während das Programm sich in der Funktion *SPM_CalculateSpeed* befindet. Die Anforderung besagt, dass keine arithmetische Operation einen Überlauf produzieren darf. Es soll dies an der Funktion *SPM_CalculateSpeed* überprüft werden, während der Programm Counter (PC) innerhalb dieser Funktion ist.

Die entsprechenden Label für die Funktion sind:

Start der Funktion: SPM_CalculateSpeed_START

Ende der Funktion: SPM_CalculateSpeed_ENDE

CTL 3:

SA 3:

2.5 Fall 4:

Es soll geprüft werden, ob das Interrupt Flag I während der Initialisierungsphase auf I = 0 gesetzt ist. Die Anforderung besagt, dass ein gesetztes Interrupt Flag während der Initialisierungsphase zu unerwünschten, evtl. gefährlichen, Zuständen führen kann, da bei I = 1 zu jeder Zeit ein Interrupt kommen kann und dieser evtl. unvollständig konfiguriert agiert. Die Anforderung muss für diese Phase gelten. Danach soll die Interruptkontrolle aktiv sein.

Die entsprechenden Label für die Funktion sind:

Start der Funktion: INIT_START

Ende der Funktion: INIT_ENDE

CTL 4:

SA 4:

2.6 Fall 5:

Gegeben sei folgender Pseudo-Code:

```
state = INIT;
const UINT16 MaxTime = 500;
const UINT16 WD_Init = 2000;


switch( state )
{
    case INIT: /* 1 */
        if ((Ctr >= 0) && (Ctr < WD_Init))
        {
                if (FPGA == 1)
                {
                        state = COMP_OK;
                }
        }
        else
        {
                state = FAILURE;
        }
         break;

    case BIT_SET_WAIT: /* 2 */
         if ((Ctr >= MaxTime)){
             state = GET_BITS;
         }
         break;
```

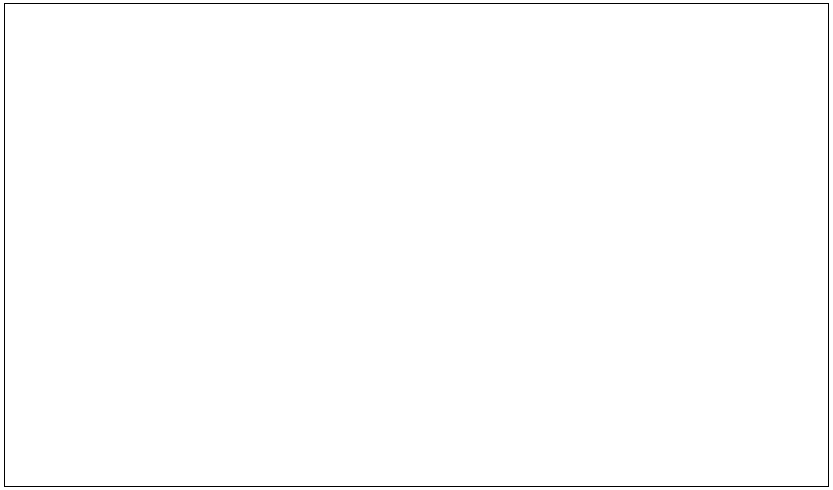
```
    case GET_BITS: /* 3 */
        if( IOport1 == uiCompareVal)
        {
            state = COMP_OK;
            uiFirstRead = 1;
        }
        else
        {
            if (uiFirstRead == 1)
            {
                uiFirstRead = 0;
                Ctr = 0;
                state = BIT_SET_WAIT;
            }
            else
            {
                state = FAILURE;
            }
        }
        break;
    case COMP_OK: /* 4 */
        if(uiIOSendValue == 0)
        {
            uiIOSendValue = 1;
            uiCompareVal = 0;
        }
        else
        {
            uiIOSendValue = 0;
            uiCompareVal = 1;
        }
        port = uiIOSendValue;
        Ctr = 0;
        state = BIT_SET_WAIT;

        break;
    case FAILURE: /* 5 */
        wdt_watchdog_reset();
        state = RESET;
        break;
    case RESET: /* 6 */
        break;
}/* end switch */
```

Versuchen Sie im Folgenden die gültigen Veränderungen der Variablen „state“ in CTL sowie in SA zu modellieren.

CTL 5:

SA 5:

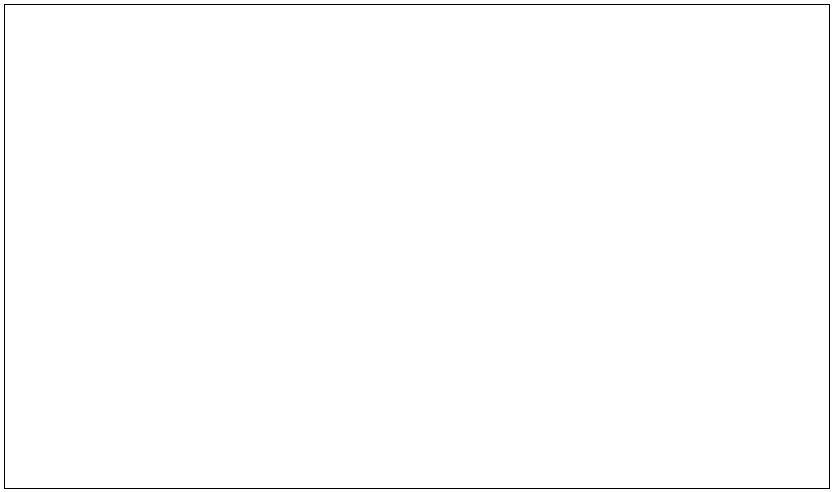


2.7 Fall 6:

Es soll nachgewiesen werden, dass die Funktion Fkt_6 immer wieder aufgerufen wird.
Die entsprechenden Label für die Funktion sind:
Start der Funktion: Fkt_6_START
Ende der Funktion: Fkt_6_ENDE

CTL 6:

SA 6:



2.8 Abschließende Befragung:

Wie gut …	**schlecht**	**Gut**				**sehr gut**	**Keine Aussage**
…, denken Sie, haben Sie die beschriebenen Aufgaben gelöst?	1	2	3	4	5	6	0
… half Ihnen Ihr Wissen bzgl. CTL bei der Lösung der Aufgaben?	1	2	3	4	5	6	0
… half Ihnen Ihr Wissen bzgl. Automaten/SA bei der Lösung der Aufgaben?	1	2	3	4	5	6	0
… kamen Sie mit der Formulierung von Anforderungen in CTL zurecht?	1	2	3	4	5	6	0
… kamen Sie mit der Formulierung von Anforderungen in SA zurecht?	1	2	3	4	5	6	0
… half Ihnen die einleitende Beschreibung bzgl. CTL bei der Bearbeitung der Aufgaben?	1	2	3	4	5	6	0
… half Ihnen die einleitende Beschreibung bzgl. SA bei der Bearbeitung der Aufgaben?	1	2	3	4	5	6	0
… half Ihnen der logisch-formale Aufbau von CTL bei der Bearbeitung der Aufgaben?	1	2	3	4	5	6	0
… half Ihnen der grafische Aufbau von SA bei der Bearbeitung der Aufgaben?	1	2	3	4	5	6	0
… bewerten Sie die visuelle Darstellmöglichkeit von CTL zur Spezifikation von Anforderungen?	1	2	3	4	5	6	0
… bewerten Sie die visuelle Darstellmöglichkeit von SA zur Spezifikation von Anforderungen?	1	2	3	4	5	6	0
… bewerten Sie die intuitive Nutzbarkeit von CTL zur Spezifikation von Anforderungen?	1	2	3	4	5	6	0
… bewerten Sie die intuitive Nutzbarkeit von SA zur Spezifikation von Anforderungen?	1	2	3	4	5	6	0

Grundsätzliche Meinung bzw. Anregung zu CTL:

Grundsätzliche Meinung bzw. Anregung zu SA:

B Verwendete Safety Automaten

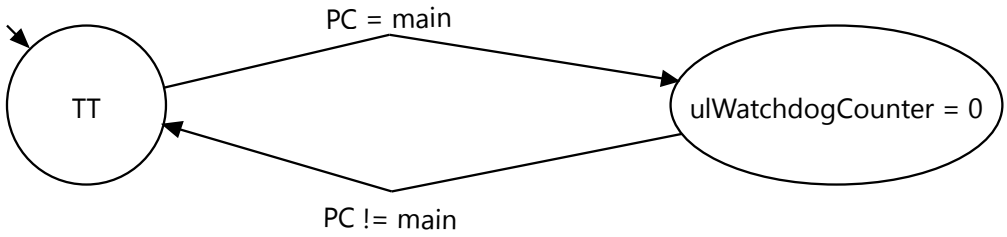


Abbildung B.1: Variable wird korrekt initialisiert

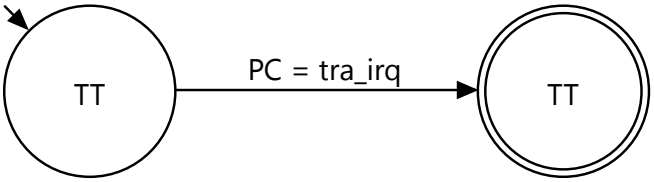


Abbildung B.2: Funktion kann ausgeführt werden

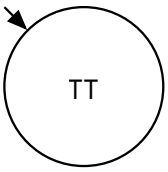


Abbildung B.3: Der gesamte Zustandsraum soll aufgebaut werden

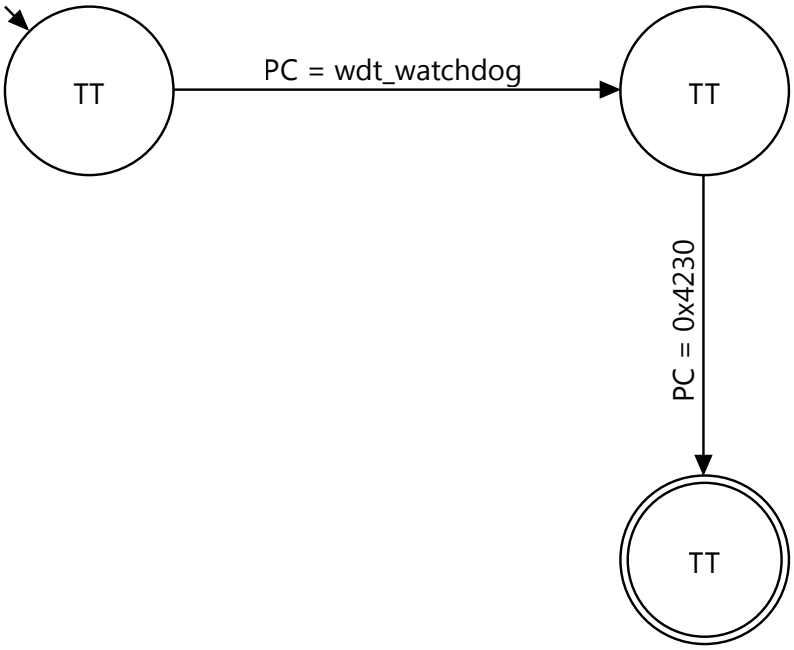


Abbildung B.4: Funktion kann ausgeführt und komplett durchlaufen werden

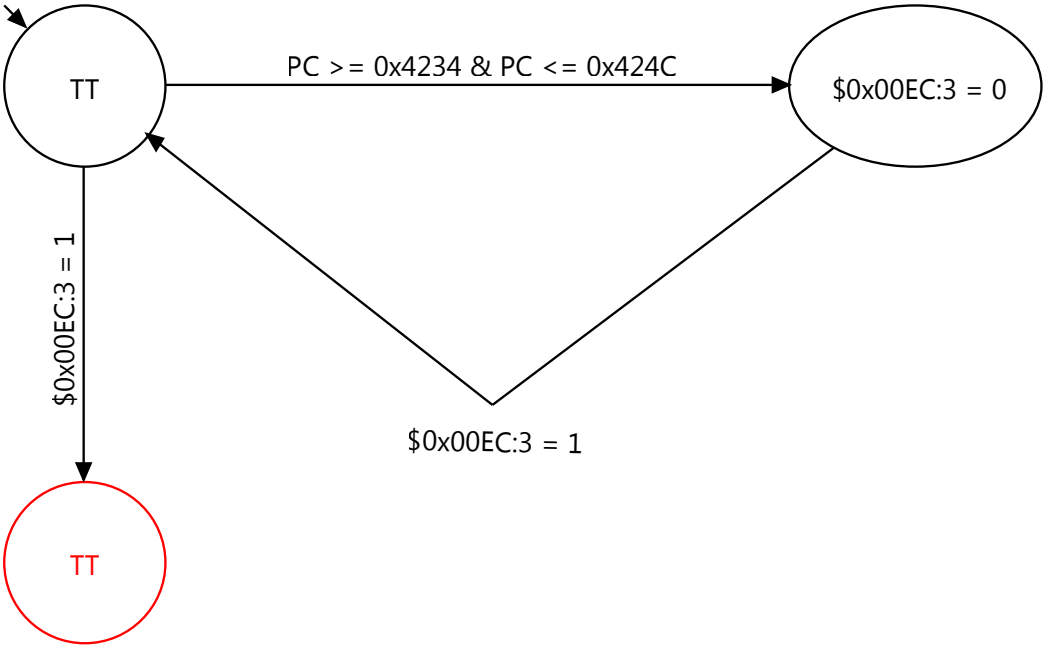


Abbildung B.5: Register wird nur in einer bestimmten Funktion gesetzt

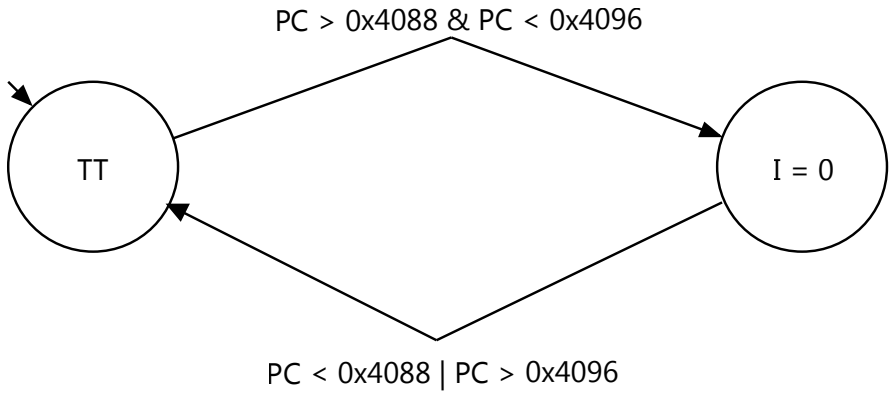


Abbildung B.6: Interrupts sind in der Initialisierungsphase deaktiviert, Version 1

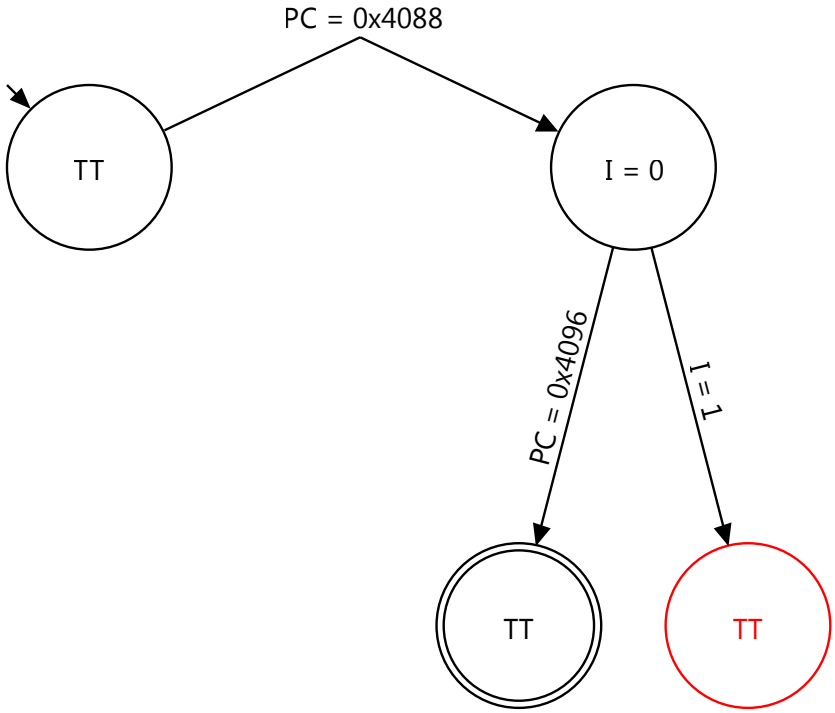


Abbildung B.7: Interrupts sind in der Initialisierungsphase deaktiviert, Version 2

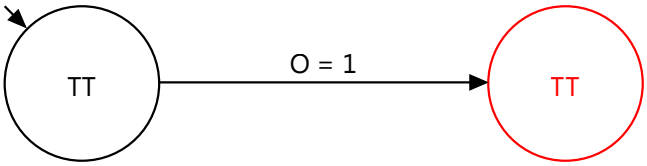


Abbildung B.8: Das Overflow Bit wird nie gesetzt

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

ADAS	Advanced Driver Assistant Systems
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System ARchitecture
BCET	Best Case Execution Time
BCRT	Best Case Response Time
BPMN	Business Process Model and Notation
CIF	Compositional Interchange Format
CTL	Computational Tree Logic
CTL*	Computational Tree Logic*
DF	Design Framework
DMS	Deadline-Monotonic-Scheduling
EDF	Earliest-Deadline-First
ELF	Executable and Linking Format
ET	Execution Time
GNSS	Grafische Notation zur Spezifikation mit Safety-Pattern
IoT	Internet of Things
KMU	kleine und mittlere Unternehmen
LTL	Linear Temporal Logic
MAP	Manifold Absolute Pressure
PC	Program Counter
PCC	Piezoelectric Controlled Carburetor
PWM	Pulsweitenmodulation
RMS	Rate-Monotonic-Scheduling
RT	Response Time
SA	Safety-Automaten
SAPIS	Safety-Pattern Instanziierungs-System
SOTL	Safety-Oriented Technical Language
STD	Sequence Timing Diagrams
TCTL	Timed Computation Tree Logic
UDE	User-Defined Environment
UML	Unified Modeling Language
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time

Symbolverzeichnis

C	Ausführungszeit eines Tasks
D	Deadline eines Tasks
λ	Lambda: Verhältnis von Luft und Brennstoff in einem Verbrennungsprozess
T	Periodenlänge eines Tasks
U	Auslastung eines Task-Systems

Literaturverzeichnis

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[2] Volkswagen AG. Baukastenprinzip: Vielfalt durch einheitliche Standards. Verfügbar unter: `http://www.viavision.org/ftp/735.pdf`, 2012. [Abgerufen am 07.03.2019].

[3] Defense Advanced Research Projects Agency. DARPA-BAA-12-17: Crowd Sourced Formal Verification (CSFV). Initial Announcement, December 2011.

[4] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[5] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335. Springer-Verlag New York, Inc., 1990. doi: 10.1007/BFb0032042.

[6] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - A Tool for Modelling and Implementation of Embedded Systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 460–464. Springer-Verlag, 2002. doi: 10.1007/3-540-46002-0_32.

[7] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *In Proc. of FORMATS?03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003. doi: 10.1007/978-3-540-40903-8_6.

[8] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *The 2004 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '04*, pages 67–72. ACM Press, 2004. doi: 10.1145/1066129.1066143.

[9] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-37521-7.

[10] N. Audsley. Deadline Monotonic Scheduling. *YCS 146, Department of Computer Science, University of York*, October 1990.

[11] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[12] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 202–213. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-69149-5_22.

[13] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004. doi: 10.1007/978-3-540-30080-9_7.

[14] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. Uppaal-Tiga: time for playing games! In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 121–125. Springer Berlin Heidelberg, 2007. doi: 10.1007/978-3-540-73368-3_14.

[15] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin Heidelberg, 1996. doi: 10.1007/BFb0020949.

[16] D. Beuche. *Composition and construction of embedded software families.* Dissertation, Otto von Guericke University Magdeburg, 2004.

[17] D. Beuche. Modeling and Building Software Product Lines with Pure::Variants. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 358–358, 2008. doi: 10.1109/SPLC.2008.53.

[18] D. Beuche. pure::variants. In R. Capilla, J. Bosch, and K. Kang, editors, *Systems and Software Variability Management*, pages 173–182. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-36583-6_12.

[19] S. Biallas, V. Kamin, S. Kowalewski, B. Schlich, S. Sehestedt, and S. Stattelmann. Verifikation von sicherheitsgerichteten SPS-Programmen mit Hilfe von Safety-Automaten. In VDI Wissensforum, editor, *Automation 2013*, VDI Berichte, pages 75–79. VDI-Verlag, 2013.

[20] F. Bitsch. A way for applicable formal specification of safety requirements by tool-support. *FORMS 2003 - Symposium on Formal Methods for Railway Operation and Control Systems 2003*, 2003.

[21] F. Bitsch. *Verfahren zur Spezifikation funktionaler Sicherheitsanforderungen für Automatisierungssysteme in Temporallogik*. PhD thesis, Universität Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2007. URL `http://elib.uni-stuttgart.de/opus/volltexte/2007/3041`. [Abgerufen am 07.03.2019].

[22] R. Blum, M. Düsterhöft, and M. Reke. Precise mixture control for small gasoline engines. *MTZ worldwide*, 68(10):10–13, 2007.

[23] B. W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. In *EURO IFIP 79*, pages 711–719. North Holland, 1979.

[24] *BTC EmbeddedValidator Pattern Library, Release 3.7*. BTC Embedded Systems AG, 2012.

[25] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer US, 2011. doi: 10.1007/978-1-4614-0676-1.

[26] F. Cassez and K. G. Larsen. The Impressive Power of Stopwatches. In *CONCUR 2000 - Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-44618-4_12.

[27] W. Clark, W.N. Polakov, and F.W. Trabold. *The Gantt Chart: A Working Tool of Management*. Ronald manufacturing management and administration series. Ronald Press Company, 1922.

[28] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In R. Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer Berlin Heidelberg, 2001. doi: 10.1007/3-540-44577-3_12.

[29] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1982. doi: 10.1007/BFb0025774.

[30] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.

[31] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. Verfügbar unter: `http://refspecs.linuxbase.org/elf/elf.pdf`, 1995. [Abgerufen am 07.03.2019].

[32] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In

Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973.

[33] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., 1972.

[34] A. David, J. Illum, K.G. Larsen, and A. Skou. *Model-Based Design for Embedded Systems*, chapter Model-Based Framework for Schedulability Analysis Using Uppaal 4.1, pages 93–119. CRC Press, 2010.

[35] A. David, K.G. Larsen, A. Legay, and M. Mikučionis. Schedulability of Herschel-Planck Revisited Using Statistical Model Checking. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *Lecture Notes in Computer Science*, pages 293–307. Springer Berlin / Heidelberg, 2012. doi: 10.1007/978-3-642-34032-1_28.

[36] D. L. Dill and J. Rushby. Acceptance of Formal Methods: Lessons from Hardware Design. *IEEE Computer*, 29(4):23–24, April 1996. doi: 10.1109/MC.1996.488298.

[37] Y. Dong, X. Du, Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, O. Sokolsky, E.W. Stark, and D.S. Warren. Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin Heidelberg, 1999. doi: 10.1007/3-540-49059-0_6.

[38] C. Dziobek, J. Loew, W. Przystas, and J. Weiland. Von Vielfalt und Variabilität. *Elektronik Automotive*, 2:33–37, 2008.

[39] E. A. Emerson. The Beginning of Model Checking: A Personal Perspective. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 27–45. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-69850-0_2.

[40] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1980. doi: 10.1007/3-540-10003-2_69.

[41] E. A. Emerson and J. Y. Halpern. „Sometimes“ and „Not Never“ Revisited: On Branching Versus Linear Time Temporal Logic. *J. ACM*, 33(1), 1986. doi: 10.1145/4904.4999.

[42] ERTRAC Task Force - Connectivity and Automated Driving. Automated Driving Roadmap. Verfügbar unter: `http://www.ertrac.org/uploads/documentsearch/id38/ERTRAC_Automated-Driving-2015.pdf`, July 2015. [Abgerufen am 07.03.2019].

[43] Europäische Kommission. Empfehlung der Kommision vom 6. Mai 2003 betreffend die Definition der Kleinstunternehmen sowie der kleinen und mittleren Unternehmen - 2003/361/EG. *Amtsblatt der Europäischen Union L 124*, Mai 2003.

[44] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31. American Mathematical Society, 1967.

[45] H.G. Frischkorn. Automotive Software - The Silent Revolution, January 2004.

[46] S. Grobosch, V. Kamin, M. Hüfner, R. Hamberg, H. Moneva, T. Punter, A. Skou, R. Izadi-Zamanabadi, and C. Sonntag. 4 reports on first results and experiences obtained for the case studies, Deliverable No: D6.1.3. Public deliverable, The MULTIFORM Project (Grant Agreement: FP7-ICT-2007-224249), 2011.

[47] D. Gückel, B. Schlich, J. Brauer, and S. Kowalewski. Synthesizing Simulators for Model Checking Microcontroller Binary Code. In *Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2010)*, pages 313–316. IEEE, 2010. doi: 10.1109/DDECS.2010.5491761.

[48] R. Heckmann, C. Ferdinand, Absint Angewandte, and Informatik GmbH. Worst-Case Execution Time Prediction by Static Program Analysis. In *IPDPS 2004*, pages 26–30. IEEE Computer Society, 2004.

[49] C. Heitmeyer. Scr: a practical method for requirements specification. In *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, volume 1, pages C44/1–C44/5 vol.1, Oct 1998. doi: 10.1109/DASC.1998.741500.

[50] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 394–406, Jun 1992. doi: 10.1109/LICS.1992.185551.

[51] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10), 1969. doi: 10.1145/363235.363259.

[52] C. A. R. Hoare. Proof of a Program: FIND. *Commun. ACM*, 14(1), 1971. doi: 10.1145/362452.362489.

[53] R. Höhn and S. Höppner, editors. *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Springer, 2008. doi: 10.1007/978-3-540-30250-6.

[54] H.J. Holberg and U. Brockmeyer. Computer-aided Formal Specification to enable a fully automated Requirements-based Testing Process. In *Embedded World Conference 2012, Nürnberg*, February 2012.

[55] ISO. Road vehicles - Functional safety - ISO/DIS 26262. Standard, International Organization for Standardization / Technical Committee 22 (ISO/TC 22), April 2011.

[56] ISO/IEC. Information technology — Z formal specification notation — Syntax, type system and semantics - ISO/IEC 13568. Standard, International Organization for Standardization / International Electrotechnical Commission, July 2002.

[57] Kamin, V. and Embedded Software Laboratory - RWTH Aachen University. Webseite zu ARCADE. Verfügbar unter: `https://arcade.embedded.rwth-aachen.de`, 2013. [Abgerufen am 07.03.2019].

[58] C. Kern and M. R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, April 1999. ISSN 1084-4309. doi: 10.1145/307988.307989.

[59] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7): 385–394, July 1976. doi: 10.1145/360248.360252.

[60] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. doi: 10.1145/1629575.1629596.

[61] S. A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[62] K. Krirkkawin. Analysis and optimization of software architecture for an engine control system. Master thesis, RWTH Aachen University, Aachen, Germany, June 2010.

[63] C. W. Krueger. Easing the Transition to Software Mass Customization. In F. van der Linden, editor, *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43659-1. doi: 10.1007/3-540-47833-7_25.

[64] C.W. Krueger. The Systems and Software Product Line Engineering Lifecycle Framework. Verfügbar unter: `http://www.biglever.com/extras/PLE_LifecycleFramework.pdf`, 2013. [Abgerufen am 07.03.2019].

[65] A. van Lamsweerde. Formal Specification: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00. ACM, 2000. doi: 10.1145/336512.336546.

[66] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. doi: 10.1007/s100090050010.

[67] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982. ISSN 0166-5316. doi: 10.1016/0166-5316(82)90024-4.

[68] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743.

[69] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2-3):183–207, 1999. doi: 10.1023/A:1008138407139.

[70] E. Mercer and M. Jones. Model Checking Machine Code with the GNU Debugger. In P. Godefroid, editor, *Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265. Springer Berlin Heidelberg, 2005. doi: 10.1007/11537328_20.

[71] M. Mikučionis, K.G. Larsen, J.I. Rasmussen, B. Nielsen, A. Skou, S.U. Palm, J.S. Pedersen, and P. Hougaard. Schedulability analysis using UPPAAL: Herschel-Planck case study. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ISoLA'10, pages 175–190. Springer-Verlag, 2010. doi: 10.1007/978-3-642-16561-0_21.

[72] H. Moneva, R. Hamberg, T. Punter, and J. Vissers. Putting Chaos under Control: on how Modeling should Support Design. *INCOSE International Symposium*, 20 (1):172–186, 2010. doi: 10.1002/j.2334-5837.2010.tb01063.x.

[73] H. Moneva, R. Hamberg, and T. Punter. A Design Framework for Model-based Development of Complex Systems. In *32nd IEEE Real-Time Systems Symposium 2nd Analytical Virtual Integration of Cyber-Physical Systems Workshop, Vienna*, 2011.

[74] F.L. Morris and C.B. Jones. An Early Program Proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, April 1984. doi: 10.1109/MAHC.1984.10017.

[75] J. Mossinger. Software in Automotive Systems. *Software, IEEE*, 27(2):92–94, March 2010. doi: 10.1109/MS.2010.55.

[76] N.N. *R8C/22 Group, R8C/23 Group Hardware Manual*. Renesas Technology, rev. 2.00 edition, August 2008.

[77] L.M. Northrop and P.C. Clements. A Framework for Software Product Line Practice, Version 5.0 . Verfügbar unter: `http://www.sei.cmu.edu/productlines/frame_report/index.html`, 2013. [Abgerufen am 07.03.2019].

[78] J.A. Pereira, K. Constantino, and E. Figueiredo. A systematic literature review of software product line management tools. In I. Schaefer and I. Stamelos, editors, *Software Reuse for Dynamic Systems in the Cloud and Beyond*, volume 8919 of *Lecture Notes in Computer Science*, pages 73–89. Springer International Publishing, 2014. ISBN 978-3-319-14129-9. doi: 10.1007/978-3-319-14130-5_6.

[79] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–67. IEEE Computer Society Press, 1977. doi: 10.1109/SFCS.1977.32.

[80] F. Poppen. Gründe für die Virtualisierung eingebetteter Systeme. *iX Developer*, page 7, 2 2014. Holodeck auf der Spur.

[81] E. Post. Introduction to a General Theory of Elementary Propositions. *American Journal of Mathematics*, 43(1):163–185, 1921.

[82] S. Preuße and H.-M. Hanisch. Specification and Verification of Technical Plant Behavior with Symbolic Timing Diagrams. In *Design and Test Workshop, 2008. IDT 2008. 3rd International*, pages 313–318, Dec 2008. doi: 10.1109/IDT.2008.4802520.

[83] S. Preuße and H.-M. Hanisch. *Vergleich von Verfahren zur Spezifikation von Anlagenverhalten*. Technische Berichte 2008/05. Martin-Luther-University, Halle-Wittenberg, Institut für Informatik, Halle, Saale, July 2008.

[84] S. Preusse and H.-M. Hanisch. Specification of Technical Plant Behavior with a Safety-Oriented Technical Language. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 632–637, June 2009. doi: 10.1109/INDIN.2009.5195876.

[85] A.N. Prior. Tense-logic and the continuity of time. *Studia Logica*, 13(1):133–148, 1962. doi: 10.1007/BF02317267.

[86] pure-systems GmbH. *pure::variants User's Guide, Version 3.0 for pure::variants 3.0*, 2012. Dateiname: pv-user-manual.pdf.

[87] pure-systems GmbH. Technical White Paper - Variant Management with pure::variants. Verfügbar unter: `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, 2013. [Abgerufen am 07.03.2019].

[88] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1982. doi: 10.1007/3-540-11494-7_22.

[89] S. Ratanaprutthakul. Integration of Model Checking into the Development Process of Engine Control Systems for SMEs. Master thesis, RWTH Aachen University, Aachen, Germany, January 2011.

[90] W. Rautenberg. *Einführung in die Mathematische Logik*. Vieweg+Teubner, 2008. doi: 10.1007/978-3-8348-9530-1.

[91] T. Reinbacher, M. Kramer, M. Horauer, and B. Schlich. Motivating Model Checking for Embedded Systems Software. In *Mechatronic and Embedded Systems and Applications (MESA08), Beijing, China*, pages 546–551. IEEE Computer Society Press, 2008. doi: 10.1109/MESA.2008.4735653.

[92] T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer. Model Checking Assembly Code of an Industrial Knitting Machine. In *Embedded and Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on*, pages 1–8, 2009. doi: 10.1109/EM-COM.2009.5402986.

[93] M. Reke. *Modellbasierte Entwicklung automobiler Steuerungssysteme in kleinen und mittelständischen Unternehmen*. Dissertation, RWTH Aachen University, Aachen, Germany, July 2013.

[94] M. Reke, S. Grobosch, R. Hamberg, M. Hüfner, V. Kamin, M. Komareji, and C. Sonntag. Final Description of the Case Studies, Deliverable No: D6.1.2. Public deliverable, The MULTIFORM Project (Grant Agreement: FP7-ICT-2007-224249), 2010.

[95] J. M. Richenhagen. *Entwicklung von Steuerungs-Software für den automobilen Antriebsstrang mit agilen Methoden*. Dissertation, RWTH Aachen University, Aachen, Germany, September 2014.

[96] J. M. Richenhagen, S. Pischinger, and A. Schloßer. PERSIST—A Flexible and Automatically Verifiable Software Architecture for the Automotive Powertrain. *Journal of Electrical Engineering*, 2:108–115, 2014.

[97] D. Rixen. Entwicklung und Untersuchung einer modellbasierten Softwarearchitektur für sicherheitskritische Anwendungen. Master thesis, RWTH Aachen University, Aachen, Germany, December 2011.

[98] M. Rohrbach. Model checking of embedded systems software. Master thesis, RWTH Aachen University, Aachen, Germany, 2006.

[99] B. Schlich. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.

[100] B. Schlich and S. Kowalewski. Model Checking C Source Code for Embedded Systems. In *IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2005), Columbia, Maryland, USA*, pages 65–77. NASA, September 2005.

[101] B. Schlich and S. Kowalewski. [mc]square: A Model Checker for Microcontroller Code. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 466–473, Nov 2006. doi: 10.1109/ISoLA.2006.62.

[102] B. Schlich and S. Kowalewski. An Extendable Architecture for Model Checking Hardware-Specific Automotive Microcontroller Code. In E. Schnieder and G. Tarnai, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007), Braunschweig,Germany*, pages 202–212. GZVB, 2007.

[103] B. Schlich and S. Kowalewski. Model Checking C Source Code for Embedded Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(3), 2009. doi: 10.1007/s10009-009-0106-5.

[104] B. Schlich, F. Salewski, and S. Kowalewski. Applying Model Checking to an Automotive Microcontroller Application. In *Industrial Embedded Systems (SIES'07), Lisbon, Portugal*, pages 209–216. IEEE Computer Society Press, 2007. doi: 10.1109/SIES.2007.4297337.

[105] B. Schlich, D. Gückel, and S. Kowalewski. Modeling the Environment of Microcontrollers to Tackle the State-Explosion Problem in Model Checking. In G. Tarnai and E. Schnieder, editors, *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008), Budapest, Hungary*, pages 27–34. L'Harmattan, 2008.

[106] B Schlich, J. Brauer, and S. Kowalewski. Application of Static Analyses for State Space Reduction to Microcontroller Binary Code. *Sci. Comput. Program.*, 76(2): 100–118, 2011.

[107] R. C. Schlör. *Symbolic timing diagrams: a visual formalism for model verification*. Dissertation, Universität Oldenburg, 2002. URL `http://oops.uni-oldenburg.de/301/`. [Abgerufen am 07.03.2019].

[108] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[109] D. Sexton. An Outline Workflow for Practical Formal Verification from Software Requirements to Object Code. In C. Pecheur and M. Dierkes, editors, *Formal Methods for Industrial Critical Systems*, volume 8187 of *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-41010-9_3.

[110] N. Sharygina, J. Browne, F. Xie, R. Kurshan, and V. Levin. Lessons Learned from Model Checking a NASA Robot Controller. *Formal Methods in System Design*, 25 (2-3):241–270, 2004. doi: 10.1023/B:FORM.0000040029.73127.85.

[111] D. Simon and T. Eisenbarth. Evolutionary Introduction of Software Product Lines. In *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 272–283. Springer Berlin Heidelberg, 2002. doi: 10.1007/3-540-45652-X_17.

[112] C. Stoermer and M. Roeddiger. Introducing Product Lines in Small Embedded Systems. In F. van der Linden, editor, *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 101–112. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43659-1. doi: 10.1007/3-540-47833-7_11.

[113] A. S. Tanenbaum. *Modern Operating Systems*. Pearson Education, 3rd edition, 2008.

[114] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 2012. doi: 10.1016/j.scico.2012.06.002.

[115] M. Traub. *Durchgängige Timing-Bewertung von Vernetzungsarchitekturen und Gateway-Systemen im Kraftfahrzeug*. Dissertation, Karlsruher Institut für Technologie (KIT), Institut für Technik der Informationsverarbeitung (ITIV), Aachen, Germany, 2010.

[116] A. M. Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Mathematical Laboratory, Cambridge, 1949. URL `http://www.turingarchive.org/browse.php/B/8`. [Abgerufen am 07.03.2019].

[117] D. A. Van Beek, P. Collins, D. E. Nadales, J. E. Rooda, and R. R. H. Schiffelers. New Concepts in the Abstract Format of the Compositional Interchange Format. In *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 250–255. IFAC, 2009. doi: 10.3182/20090916-3-ES-3003.00044.

[118] T. Will. Evaluierung von formalen Methoden in der Softwareentwicklung von kleinen und mittleren Unternehmen. Master thesis, RWTH Aachen University, Aachen, Germany, Oktober 2012.

[119] J. M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8–23, 1990. doi: 10.1109/2.58215.

[120] L. Wittgenstein. *Tractatus Logico-Philosophicus*. London: Kegan Paul, Trench, Trubner & Co., Ltd.; New York: Harcourt, Brace & Company, Inc, 1922.

Eigene Publikationen

[GKKK11] S. Grobosch, V. Kamin, K. Krirkkawin, and S. Kowalewski. Using Timed Automata in Requirements Analysis for Engine Control Units. In *18th World Congress of the International Federation of Automatic Control (IFAC)*, pages 12503–12508. IFAC, 2011. doi: 10.3182/20110828-6-IT-1002.03679.

[GRK13] S. Grobosch, M. Reke, and S. Kowalewski. Formale Methoden in kleinen und mittleren Unternehmen. In *Embedded Software Engineering Kongress (ESE)*. ELEKTRONIKPRAXIS und MicroConsult, 2013. doi: 10.13140/2.1.3446.1766.

[HSEG13] M. Hüfner, C. Sonntag, S. Engell, and S. Grobosch. A Customized Design Framework for the Model-based Development of Engine Control Systems. In *Industrial Electronics Society, IECON 2013 - 39th Annual Conference of the IEEE*, pages 6916–6921, 2013. doi: 10.1109/IECON.2013.6700279.

[KWRG13] M. Kötter, C. Wirtz, M. Reke, and S. Grobosch. Elektrische Manipulation von Abgasrelevanten Aktuatoren und Sensoren. *ATZelektronik*, 8(3):222–226, 2013. ISSN 1862-1791. doi: 10.1365/s35658-013-0300-9.

[RG11] S. Ratanaprutthakul and S. Grobosch. Model checking and verification method of engine control units. In *Software Engineering (MySEC), 2011 5th Malaysian Conference*, pages 183–188, 2011. doi: 10.1109/MySEC.2011.6140666.

[RG14] M. Reke and S. Grobosch. Configurable Hardware for Electronic Control Units. In *Embedded World Conference 2014, Nürnberg*, February 2014.

[RGK12] M. Reke, S. Grobosch, and S. Kowalewski. Steuergeräte-Entwicklung in kleinen und mittelgroßen Unternehmen. In *Embedded Software Engineering Kongress (ESE)*. ELEKTRONIKPRAXIS und MicroConsult, 2012.

[RGN11] M. Reke, S. Grobosch, and K. Niegetiet. Piezoelectric Controlled Carburettor. In *Proceedings. (SAE-Paper; No. 2011-32-0528)*, SETC: Small Engine Technology Conference, 2011. doi: 10.4271/2011-32-0528.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2016-01 *	Fachgruppe Informatik: Annual Report 2016
2016-02	Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud
2016-07	Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-08	Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features
2016-09	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10	Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++

2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders
2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems
2018-06	Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking
2019-01 *	Fachgruppe Informatik: Annual Report 2019
2019-02	Tim Felix Lange: IC3 Software Model Checking

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.