TECHNISCHE
UNIVERSITÄT
DARMSTADT

FORMALE ANFORDERUNGSANALYSE UND TESTUNTERSTÜTZUNG IM PRODUKTLINIENKONTEXT

VOM FACHBEREICH 18
ELEKTRO- UND INFORMATIONSTECHNIK
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

DIPL.-ING. FLORIAN MARKERT

GEBOREN AM

26. NOVEMBER 1979 IN FRANKFURT/MAIN

REFERENT: PROF. DR.-ING. H. EVEKING
KORREFERENT: PROF. DR. RER. NAT. A. SCHÜRR

TAG DER EINREICHUNG: 26.01.2012
TAG DER MÜNDLICHEN PRÜFUNG: 06.06.2012

D17
DARMSTADT 2012

ZUSAMMENFASSUNG

Eine aus Anforderungen bestehende Spezifikation stellt die Grundlage für Entwicklungsprojekte dar. Fehler, die in der Spezifikation enthalten sind, wirken sich auf den gesamten Entwicklungsprozess aus. Im schlimmsten Fall pflanzen sie sich durch den Entwicklungs- und Testprozess fort und werden erst beim Abnahmetest entdeckt. Die Vermeidung von Fehlern in Spezifikationen ist daher eine Grundvoraussetzung für eine zügige und kostengünstige Entwicklung. Handelt es sich um sicherheitskritische Systeme, dann können Fehler in der Spezifikation im Ernstfall Menschenleben gefährden oder großen finanziellen Schaden nach sich ziehen.

Eine Überprüfung der Spezifikationen sollte daher nicht nur manuell, sondern auch formal sowie vollständig und so weit wie möglich automatisiert durchgeführt werden. Die immer stärkere Verbreitung von Produktlinien, die sich durch die Strukturierung von Variabilität und Gemeinsamkeiten verwandter Produkte mit dem Ziel der Wiederverwendung auszeichnen, steigert die Komplexität der Überprüfung der Spezifikationen im Entwicklungsprozess. Durch die Vielzahl von Produkten, die aus einer Produktlinie abgeleitet werden können, ist eine manuelle Überprüfung der Spezifikation kaum noch möglich. Handelt es sich um Spezifikationen eingebetteter Systeme, so wird die Überprüfung durch die Aufteilung in Soft- und Hardware weiter erschwert.

Im Rahmen dieser Dissertation wird ein Verfahren vorgestellt, das Spezifikationen auf Widersprüche, Vollständigkeit und Redundanz überprüft. Dieses Verfahren ist sowohl für Spezifikationen einzelner Produkte also auch für solche, die Gemeinsamkeiten und Unterschiede von Produktlinien beschreiben, einsetzbar. Die Analyse der Spezifikationen basiert auf bekannten Algorithmen aus der formalen Verifikation von Hardwareschaltungen. Eine Anwendung dieser Algorithmen für Verhaltensbeschreibungen verschiedener Systeme sowie die Erweiterung durch Variabilität wird erläutert.

Weiterhin wird auf die Generierung von „Orakeln“ eingegangen, die sich für die Voraussage von Testergebnissen eignen und ebenfalls auf Algorithmen aus der formalen Verifikation von Hardwareschaltungen basieren. Außerdem können auf einem „Orakel“ Annahmen über das Systemverhalten bewiesen werden. Zusätzlich wird ein Modell für die strukturierte Darstellung von Testfällen eingeführt. Die Testfälle können auf dem „Orakel“, einem weiteren Systemmodell oder auf dem System selbst ausgeführt werden. Die in dieser Dissertation vorgestellten Konzepte wurden an Beispielen aus dem universitären und dem industriellen Umfeld erprobt.

ABSTRACT

A requirements specification forms the basis of all development projects. Faults incorporated in this initial specification put the entire project at risk. In the worst case, faults propagate through the development and test processes and are not recognized before acceptance testing. Therefore, the avoidance of faults in specifications is the basic prerequisite for efficient and competitive development. In case of safety critical systems faults may even jeopardize human life or result in a huge financial loss.

Specification analysis should not rely exclusively on inspection or review meetings but should also be aided by automated formal methods. The use of product line engineering in the development of systems complicates the specification analysis even more. Product line engineering aims to reuse development and test fragments by structuring common and variable parts of a system. Due to the potentially huge number of products that can be derived from a product line, a manual analysis is barely possible. Further, the requirements analysis is even harder if the specification describes an embedded system due to the Software and Hardware layers.

This thesis describes a method for analysing specifications with respect to contradictions, completeness, and redundancy. The method should be applicable both to specifications dealing with single products and to those dealing with product lines. Well understood algorithms from the field of the formal verification of hardware circuits are applied to the specification analysis. Further, these algorithms are extended using the variabilty information necessary to deal with product lines.

Additionally, the thesis focuses on the generation of “oracles“ that can be utilized as predictors for test case results, building on algorithms taken from the field of the formal verification of hardware circuits. An “oracle“ may be used to prove assumptions that must be true according to the specification. Additionally, a model for structuring the test cases in the field of product line engineering is presented. These test cases may be executed by an “oracle“, another system model or by the system itself. All presented concepts are validated using examples related to academic work and to real world industrial problems.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

BDD Binary Decision Diagram

CSP Communicating Sequential Processes

FMT Feature-Model for Testing

FPGA Field Programmable Gate Array

ITL Interval Language

KNF Konjunktive Normalform

LTL Linear Temporal Logic

PSL Property Specification Language

SAT Satisfiability

SCR Software Cost Reduction

SUT Sytem-Under-Test

SVA System Verilog Assertions

UML Unified Modeling Language

VHDL Very High Speed Integrated Circuit Hardware Description Language

1 EINLEITUNG

Die Entwicklung eingebetteter Systeme, die aus Hard- und Software bestehen, schreitet immer weiter voran. Sie werden heutzutage in den verschiedensten Bereichen eingesetzt und nehmen stellenweise sogar eine dominierende Rolle mit Hinblick auf die Entwicklungs- und Produktionskosten ein. Ihr Einsatzgebiet reicht von der Unterhaltungselektronik über die Fahrzeugelektronik bis hin zur Kommunikationstechnik. Durch immer schneller werdende Hardware und verbesserte Fertigungsprozesse erweitert sich das Einsatzspektrum und die Komplexität eingebetteter Systeme fortlaufend. Dadurch ergeben sich neue Herausforderungen für die Entwicklung und den Test eingebetteter Systeme.

Die Qualitätssicherung stellt einen zentralen Bestandteil eines jeden Entwicklungsprozesses dar. Die Qualität eines Systems hängt stark von den ursprünglichen Anforderungen sowie der Güte des Testprozesses ab. Wird ein in den Anforderungen enthaltener Fehler erst spät im Entwicklungsprozess bemerkt oder werden die Anforderungen durch das Entwicklungsteam fehlinterpretiert, so sind die Kosten für die erforderlichen Verbesserungen immens. Anforderungen eindeutig und fehlerfrei bereits vor Beginn des Projekts zu formulieren stellt daher einen zentralen Punkt auf dem Weg zu qualitativ hochwertigen Systemen dar. Von qualitativ hochwertigen Anforderungen profitieren Entwicklungsprojekte unabhängig davon, ob nach Wasserfall- oder V-Modell entwickelt wird oder ob agile Entwicklungsmethoden wie Scrum zum Einsatz kommen.

Durch die immer stärkere Verbreitung von Produktlinien und der damit einhergehenden Variabilität in der Entwicklung wird die Analyse von Anforderungen weiter erschwert. Es muss nun zusätzlich sichergestellt werden, dass alle Produkte, die aus den Anforderungen einer Produktlinie hervorgehen können, widerspruchsfrei und vollständig beschrieben wurden. Hierfür müssen Automatismen gefunden werden, die die Entwicklung und das Projektmanagement unterstützen, da eine Überprüfung jedes einzelnen Produktes zu aufwändig ist.

Neben der prinzipiellen Machbarkeit von Qualitätssicherungsmaßnahmen müssen immer auch ihre Kosten und der mit ihnen verbundene Zeitaufwand beachtet werden. Man versucht beides durch die Einführung und die Weiterentwicklung von Werkzeugen auf allen Projektebenen, vom Projektmanagement bis hin zur Testfallerzeugung, im Griff zu behalten. Die Verwendung formaler

Methoden ist dabei noch wenig verbreitet, hat aber das Potential, den Funktionsumfang, den Werkzeuge zur Verfügung stellen, sinnvoll zu ergänzen bzw. zu erweitern.

Werkzeuggestützte Qualitätssicherung wird vor allem dann zur Herausforderung, wenn, wie bei eingebetteten Systemen, neben der Software auch noch Hardware mit ins Spiel kommt. Zu Beginn der Entwicklung müssen realisierbare Anforderungen definiert werden. Diese müssen dann auf ihre Konsistenz und Vollständigkeit hin überprüft werden. Weiterhin muss festgelegt werden, welche Anforderung in Hardware und welche in Software abgebildet werden soll. Eine formale Sichtweise auf die Anforderungen kann helfen, eventuelle Probleme schon im Vorfeld zu identifizieren und zu behandeln.

1.1 BEITRAG

Mit Bezug auf die beschriebenen Probleme lassen sich unter anderem folgende Fragestellungen ableiten:

- Wie können Anforderungen, die das Verhalten eines Systems beschreiben, formalisiert werden?
- Welche Einschränkungen bestehen bei der Formalisierbarkeit von Anforderungen und wie wirkt sich die Notation, in der sie vorliegen, auf ihre Formalisierbarkeit aus?
- Wie kann Variabilität zu den formalisierten Anforderungen hinzugefügt werden?
- Wie können formalisierte Anforderungen auf Vollständigkeit, Widerspruchsfreiheit und Redundanz geprüft werden?
- Wie können die formalisierten Anforderungen den Testprozess unterstützen?

Diese Arbeit nimmt Bezug auf die Fragestellungen und bedient sich dabei Methoden aus der formalen Verifikation von Hardware. Ziel dabei ist es, Wege für die Unterstützung der Anforderungsanalyse mit formalen Methoden aufzuzeigen. Ein formaler Ansatz wird in naher Zukunft durch die steigende Komplexität der Anforderungen und durch das Hinzufügen von Variabilität immer wichtiger. Dieser muss überprüfen, ob die zu Grunde liegenden Anforderungen das zu entwickelnde System vollständig beschreiben. Die Verfasser sollten sich über Freiheitsgrade, die der Entwicklung gelassen werden, im Klaren sein. Weiterhin besteht eine Grundvoraussetzung darin, dass ein Anforderungsdokument keine Widersprüche enthält. Diese führen dazu, dass eine Implementierung der Anforderungen unmöglich wird. Zuletzt sollte ein Anforderungsdokument keine redundante Information enthalten. So sind Änderungen im Anforderungsdokument leichter durchführbar. Die formalisierten

und überprüften Anforderungen dienen als Grundlage für die Generierung eines Orakels. Ein Orakel ist ein ausführbares Modell des Systems, das zur Vorhersage korrekten Systemverhaltens dient. Steht ein Orakel während des Systemtests zur Verfügung, so kann dieser auch von Projektbeteiligten durchgeführt werden, die die Anforderungen nur in Teilen kennen. Des Weiteren kann ein Orakel als Grundlage für den formalen Beweis von Sicherheitseigenschaften dienen. Diese beschreiben sichere Zustände eines Systems, in denen es sich während seiner Ausführung befinden muss. Der Beitrag dieser Arbeit lässt sich wie folgt zusammenfassen:

- Bewertung verschiedener Notationsstile, in denen Anforderungen verfasst sein können, mit Hinblick auf ihre Übersetzbarkeit in eine formale Darstellung.
- Bewertung der Ausdrucksstärke und Grenzen der formalen Darstellung.
- Entwicklung einer Methodik für das Hinzufügen von Variabilität in formalen Darstellungen.
- Erweiterung des zur strukturierten Beschreibung von Systemverhalten verwendeten 4-Variablen-Modells durch Variabilität.
- Anwendung bestehender Algorithmen aus der formalen Hardwareverifikation zur Anforderungsanalyse und Orakelgenerierung im Bereich von eingebetteten Systemen und Produktlinien eingebetteter Systeme.
- Entwicklung einer Methodik zum Beweis von Sicherheitseigenschaften über Metriken zur Vollständigkeitsbewertung.
- Vorschlag eines Gesamtkonzepts für die Anforderungsanalyse und Testunterstützung auf Basis von formalen Darstellungen im Bereich der Entwicklung einzelner Produkte und im Kontext von Produktlinien.

1.2 ÜBERBLICK

Die Arbeit gibt zu Beginn (Kap. 2) einen Überblick über dir Grundlagen der Qualitätssicherung und führt formale Sprachen und Logiken (Kap. 3) ein. Es folgt eine Einführung in die Formulierung von Anforderungen (Kap. 4) und in die Produklinienentwicklung (Kap. 5). Anschließend wird auf die Algorithmen aus der formalen Hardwareverifikation, die zur Überprüfung der Anforderungen und zur Generierung von Testorakeln dienen, eingegangen (Kap. 6) und Eigenschaftssätze für Produktlinien eingeführt (Kap. 7). Sodann wird die Verwendung dieser Algorithmen mit Hinblick auf die Produktlinienentwicklung eingebetteter Systeme behandelt (Kap. 8). Bevor die Arbeit mit einer Zusammenfassung und einem Ausblick schließt (Kap. 10) werden experimentelle Ergebnisse vorgestellt (Kap. 9).

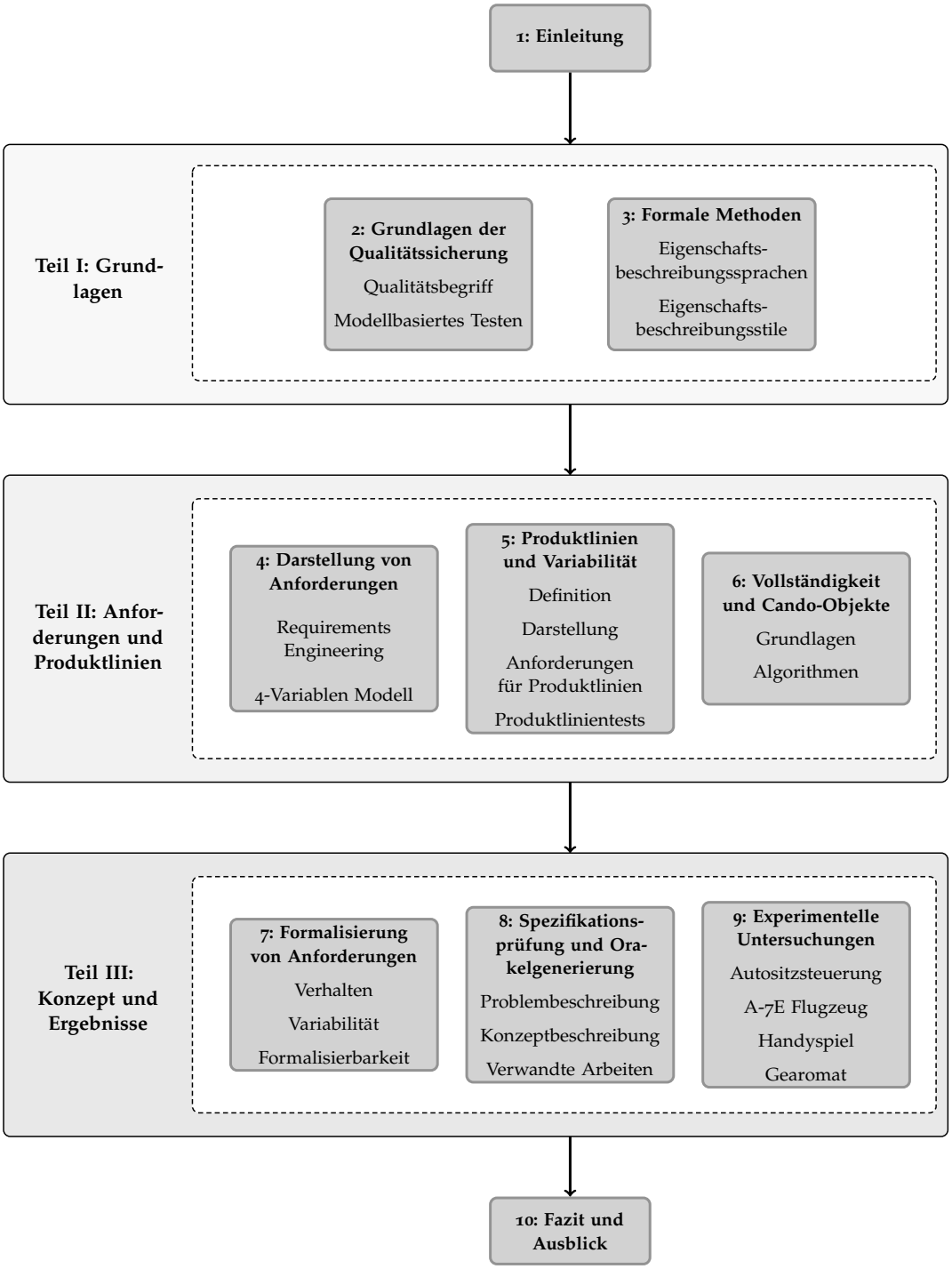


Abbildung 1: Struktur der Arbeit

Teil I

GRUNDLAGEN

In diesem Teil werden die für diese Arbeit relevanten Grundlagen beschrieben. Er beginnt mit der eigentlichen Definition des Qualitätsbegriffs und seiner Anwendung für eingebettete Systeme. Hierfür wird die Qualitätssicherung im Bereich des Softwareengineerings und die im Hardwareentwurf separat betrachtet. Weiterhin werden die Grundlagen des Testens im Software-Lebenszyklus eingeführt und modellbasierte Ansätze vorgestellt. Dem schließt sich eine Einführung in Communicating Sequential Processes und die Erläuterung der Eigenschaftsbeschreibungssprachen ITL und LTL an. Weiterhin werden Eigenschaftsbeschreibungsstile erörtert.

2

GRUNDLAGEN DER QUALITÄTSSICHERUNG

2.1 QUALITÄTSSICHERUNG

Die Sicherung von Qualität nimmt einen zentralen Punkt im Entwicklungsprozess ein. Mangelt es an ihr, so werden Produkte unverkaufbar, die Wartungskosten explodieren oder die Sicherheit von Personen kann gefährdet werden. Die grundlegende Vorgehensweise bei der Qualitätssicherung eingebetteter Systeme sowie der Qualitätsbegriff werden im Folgenden erläutert.

2.1.1 *Qualitätsbegriff*

Der Qualitätsbegriff steht im Zentrum der Qualitätssicherung. Er findet nicht nur im technischen Umfeld Anwendung, sondern wird unter Anderem auch in den Bereichen Ausbildung, Forschung, medizinische Betreuung, Agrarwesen, Finanzprodukte und vielen weiteren verwendet. Im Folgenden wird auf die Qualität eingebetteter Systeme Bezug genommen. Qualität lässt sich nach DIN 55350-11 [DIN95] als „die Gesamtheit aller Eigenschaften und Merkmale, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen“ definieren. Generell gibt es verschiedene Ausprägungen der Qualität, die auf unterschiedliche Bereiche abzielen. Die *Strukturqualität* ist dabei die globalste Interpretation des Qualitätsbegriffs und beschreibt die Qualität einer Firma.

> **Definition 2.1: *System* -** „A collection of components organized to accomplish a specific function or set of functions.“ [IEE90]

Dazu zählen Ausbildung und Fachkenntnisse der Mitarbeiter, Neuheit und Funktionalität der Arbeitsmaterialien, die Qualität der angewandten Methoden und die ausreichende Ausstattung mit Finanzmitteln. Die *Prozessqualität* setzt vor allem die Abläufe und deren Standardisierung in den Fokus. Zu ihr zählen die Punkte Sicherheit, Koordination, Wirksamkeit, Angemessenheit und Zugänglichkeit. Weiterhin wird auch die Ergebnisqualität, die im Mittelpunkt dieser Arbeit steht, bewertet. Sie beschreibt die Qualität des fertigen Produkts. Dabei liegt das Hauptaugenmerk auf der Fehlerfreiheit, der Sicherheit, der Erfüllung der Kundenwünsche und der Dokumentation. Ein Maß für die Gesamtqualität ist nach Kano die Kundenzufriedenheit bezogen auf das

fertige Produkt [ZM06]. Einzelne Anforderungen an ein Produkt werden dabei bewertet und gewichtet, indem der Einfluss auf die Kundenzufriedenheit bei Erfüllung bzw. nicht Erfüllung der selben ermittelt wird. Der Qualitätsbegriff definiert sich also gemäß dem Anwendungsgebiet, unterliegt aber immer Qualitätskriterien, die in einem Qualitätsstandard zusammengefasst sind. Die Qualitätssicherung überprüft die Einhaltung der Qualitätskriterien und definiert ein Maß, die sogenannte Metrik, für die Beurteilung der Qualität.

Aus der Definition der Qualität für das Gesamtprodukt lassen sich Qualitätsanforderungen für die Entwicklung und den Testprozess eines eingebetteten Systems ableiten. Sowohl das gewünschte Verhalten des fertigen Systems als auch die Anforderungen an dessen Qualität müssen in einer Spezifikation festgehalten werden. Anhand dieser Spezifikation wird das System entwickelt und zugehörige Testfälle aufgestellt. Die Qualität der Spezifikation ist folglich ein wesentlicher Gradmesser für mögliche Qualität des entstehenden Gesamtsystems. Da eingebettete Systeme im Normalfall aus Hardware- und Softwarekomponenten bestehen, wird der Qualitätsbegriff für diese beiden Felder im Folgenden noch weiter spezialisiert.

2.1.2 *Qualität des fertigen Produkts*

Die Qualität des fertigen Produkts gliedert sich in drei Unterpunkte, die Standards unterworfen sind, nämlich der Fertigung, dem Fertigungstest und der Interaktion mit der Umwelt. Im Folgenden wird wegen ihrer freien Verfügbarkeit vor allem auf Standards des *Department of Defense* Bezug genommen. Vor der Freigabe muss jeder integrierte Schaltkreis auf seine Tauglichkeit in Bezug auf Umwelteinflüsse getestet werden. Hierfür dient z.B. der MIL-STD-810G [Dep08], in dem genaue Testmethoden festgehalten sind nach denen ein Prototyp zu überprüfen ist.

> **Definition 2.2: *Test* -** „An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.“ [IEE90]

Dabei hängt die Wahl der Tests stark vom eigentlichen Einsatzort ab. Dieser kann sich in Bezug auf Bedingungen, wie Temperatur, Feuchtigkeit, Erschütterungen und Belastung der Atmosphäre mit Partikeln oder chemischen Stoffen, ändern. Weiterhin sind elektromagnetische Effekte nach MIL-STD-461E [Dep07b] zu untersuchen und die Normen einzuhalten. Diese Effekte beziehen sich auf die frequenzabhängigen Abstrahlungseigenschaften des Bauteils oder des Geräts. Außerdem wird die Funktionstüchtigeit des Geräts bei definierter frequenzabhängiger Einstrahlung überprüft. Auch hier hängt die Strenge der

Norm vom Einsatzort ab. Weiterhin wird das Verhalten bei hochfrequenter Einstrahlung und Überspannungspulsen auf Signal- und Spannungsversorgungsleitungen überprüft.

Der Fertigungsprozess integrierter Schaltungen wird in MIL-PRF-38535H [Dep07a] und ihre Testmethoden in MIL-STD-883G [Dep06] festgelegt. Hier werden vor allem Bedingungen definiert, die eine Fertigungsanlage in Bezug auf die Qualität erfüllen muss. Dazu gehören die Dokumentation des Fertigungsprozesses und der zugehörigen Testmethoden. Weiterhin ist definiert, welche Kennzeichnung die fertigen integrierten Komponenten haben müssen und wie der Einpackprozess ablaufen muss.

2.1.3 *Qualitätssicherung im Hardwareentwurf*

Die Qualität zeichnet sich im Hardwareentwurf durch zwei Hauptkomponenten aus. Die Erste bezieht sich auf die Entwurfsphase und die logische Korrektheit des Entwurfs, die Wartbarkeit des Codes und die Systemleistung. Sie stellt also, ähnlich wie im Software Engineering, eher eine abstrakte Sichtweise auf die Qualität dar. Die Zweite nimmt Bezug auf den tatsächlich gefertigten Chip. Dabei können Fertigungsfehler und Umwelteinflüsse die Qualität mindern. Weiterhin beeinflusst jeder produzierte und in Betrieb genommene Hardwarebaustein seine Umwelt durch elektromagnetische Abstrahlung und Wärmeemissionen. Im Folgenden wird auf die in dieser Arbeit relevante Korrektheit während der Entwurfsphase eingegangen:

Entwurfskorrektheit: Die Grundlage eines erfolgreich entwickelten Hardwaresystems bildet der korrekte Entwurf der Schaltungen. Durch sie werden Ausgangs- und Eingangssignale zueinander in Bezug gesetzt. Die Schaltung muss dabei exakt das in der Spezifikation beschriebene Verhalten abbilden. Um das sicher zu stellen gibt es zwei Hauptansätze, die sich in der Praxis gegenseitig ergänzen und auf die im Folgenden kurz eingegangen wird:

Simulation: Die Simulation zählt zu den dynamischen Testverfahren (2.2.2). Bei der Simulation wird ein Modell der Schaltung, das normalerweise in einer Hardwarebeschreibungssprache formuliert ist, durch *Stimuli* angeregt und das Ausgangsverhalten bewertet. Die Stimuli bilden die Eingangswerte der Schaltung. Sie können entweder durch den Tester direkt angegeben oder automatisch erzeugt werden. Weiterhin kann die Schaltung mit zufälligen Eingangsfolgen angeregt werden. Das Ergebnis der Simulation kann entweder automatisch über Assertionsprachen oder manuell durch den Tester ausgewertet werden. Weiterhin benötigt die Simulation Testendekriterien, die sich aus Metriken (2.2.4) ableiten lassen. Das ist nötig, weil ein vollständiger Test bei komplexen Systemen nicht mehr möglich ist.

Formale Verifikation: Bei der formalen Verifikation handelt es sich um ein statisches Testverfahren (2.2.1). Hier wird das entwickelte System nicht ausgeführt. Es werden stattdessen Eigenschaften formuliert, denen das entwickelte System genügen muss. Diese werden dann auf dem System bewiesen. Als Ergebnis erhält man für jede einzelne Eigenschaft entweder die Aussage, dass die Eigenschaft hält, oder es wird ein Gegenbeispiel ausgegeben, das eine Situation zeigt, in der die Eigenschaft nicht hält. Halten alle formulierten Eigenschaften und beschreiben diese das System in vollständiger und beabsichtigter Weise, so ist sichergestellt, dass ein den Anforderungen gemäßes, korrektes System entwickelt wurde. Die formale Verifikation wird im Normalfall immer durch die Simulation ergänzt.

Definition 2.3: ***Modul/Komponente*** **-** Ein abgeschlossener Bestandteil eines Systems.

Gerade zu Beginn der Entwicklung ist die Simulation der formalen Verifikation vorzuziehen, da durch die Simulation auch in unvollständigen Designs Fehler gefunden werden können und so eine erste, funktionierende Version des Moduls bzw. Systems erstellt werden kann. Weiterhin kann es vorkommen, dass die Verifikation an Komplexitätsgrenzen stößt und so die Simulation der einzige Weg ist, die Qualität sicherzustellen.

2.1.4 *Qualitätssicherung im Software Engineering*

Der komplette Lebenszyklus der Software ist im Bereich des Software Engineerings in einem Standard geregelt. Er ist in IEEE 12207 [IEE08] definiert und beschreibt folgende Hauptpunkte:

1. Projektakquise
2. Definition des Projektplans
3. Entwicklungs- und Testprozess
4. Verwendung der fertiggestellten Software
5. Wartung der Software

Dieser Standard bildet die Grundlage für einen Großteil der öffentlichen und privatwirtschaftlichen internationalen Ausschreibungen. Er ersetzte zudem 1998 den MIL-STD-498 und dient als Leitfaden für Softwareprojekte im Verteidigungssektor. Die Hauptpunkte beschreiben das Vorgehen wie folgt:

Projektakquise: Jedes Projekt startet mit der Initiierung, in der vorerst die Notwendigkeit der Entwicklung beschrieben wird. Anforderungen an das System

werden definiert und überprüft und globale Anforderungen an die zu entwickelnde Software werden beschrieben. Im Anschluss daran muss evaluiert werden, ob anhand der definierten Anforderungen bestehende Softwareprodukte zum Einsatz kommen können oder eine Neuentwicklung notwendig ist. Ist eine Neuentwicklung nötig, so wird ein Akquisitionsplan erstellt und die Voraussetzungen für die Zusage an einen Projektpartner definiert.

Definition des Projektplans: Nach der Zusage an einen Projektpartner wird ein Projektplan vom Projektmanagement erstellt. Der Plan beantwortet die für das Projekt grundlegenden Fragen, die alle am Projekt beteiligten Parteien betreffen. Dazu gehören:

- Welches grundlegende Ziel hat die Entwicklung? Welche Probleme sollen mit den zur Verfügung gestellten Finanzmitteln gelöst werden?
- Was soll das fertige Produkt können? Was sind die Teilprodukte und welche Funktionen haben sie?
- Wer ist an dem Projekt beteiligt? Wer trägt die Verantwortung für welche Projektteile?
- Wann soll das Projekt fertiggestellt werden und was sind die Meilensteine auf dem Weg dorthin?

Entwicklungs- und Testprozess: Sind alle Ziele eines Projekts definiert, kann mit der Entwicklung begonnen werden. Diese orientiert sich in den meisten Fällen am *V-Modell* aus Abb. 2 [Wal01]. Vom V-Modell existieren untschiedliche Ausprägungen, die aber immer die beiden Äste Implementierung und Verifikation/Validierung beinhalten und sie miteinander in Beziehung setzen. Die Verifikation bezieht sich auf die Überprüfung, ob das entwickelte System oder dessen Bestandteile den Anforderungsdokumenten genügen.

> **Definition 2.4:** ***Verifikation*** **-** „The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.“ [IEE90]

Die Validierung bewertet im Gegensatz dazu, ob das spezifikationsgetreu entwickelte System überhaupt für den beabsichtigen Gebrauch einsetzbar ist.

> **Definition 2.5:** ***Validierung*** **-** „The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.“ [IEE90]

Dadurch werden die Stufen des Entwicklungsprozesses mit dem Ziel abstrahiert, die Fehlersuche zu erleichtern. Entwicklungen nach dem V-Modell folgen

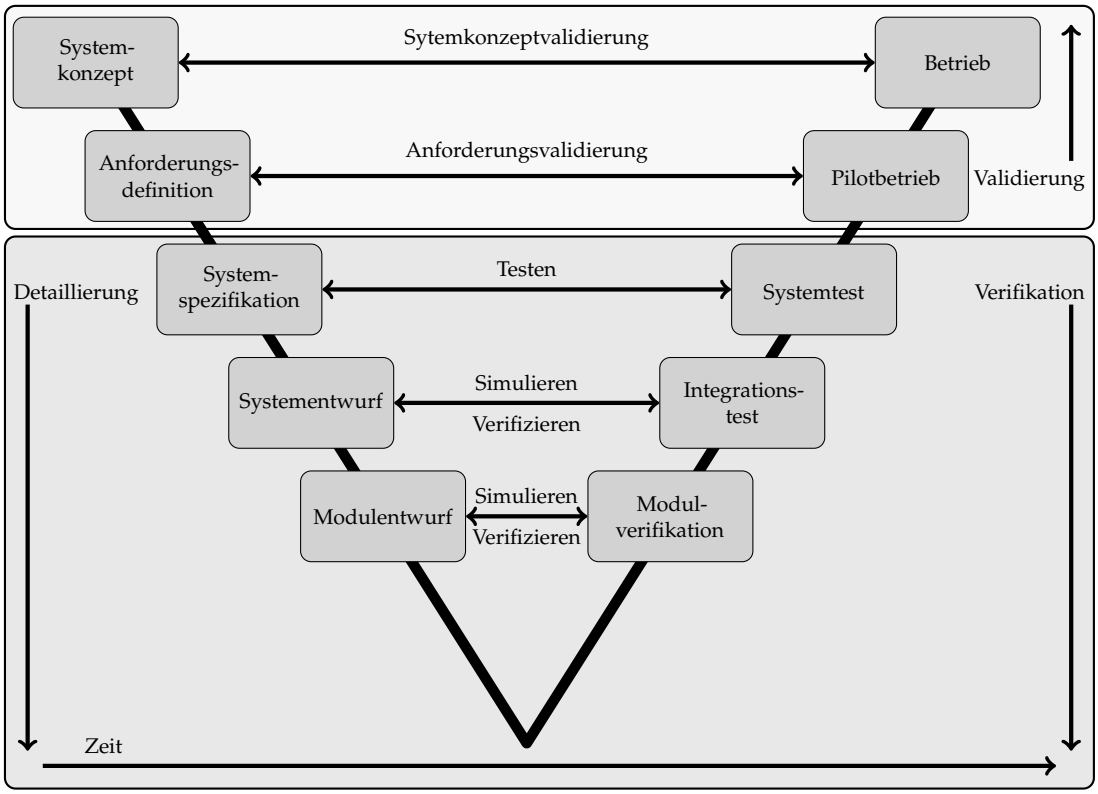


Abbildung 2: V-Modell [Walo1]

einem vordefinierten Ablauf. Nach der Erstellung des Systemkonzepts müssen die Anforderung an die Software, die sich aus den Anforderungen an das fertige Produkt ergeben, erkannt und beschrieben werden. Aus den Anforderungen lässt sich eine Systemspezifikation erzeugen, die grundlegend die gewünschte Struktur des Softwareprojekts beschreibt. Dann wird der Systementwurf erstellt und die Verbindungen, also die Art mit der Module innerhalb des Systems untereinander kommunizieren, modelliert. Nun müssen die Module, aus denen das System besteht, so genau wie möglich beschrieben werden. Aus den Modulen wird die eigentliche Implementierung automatisiert oder manuell erzeugt. Die Module werden im Anschluss einzeln getestet. Ist ein Modul fehlerhaft, so wird der Fehler analysiert und behoben. Tritt während des Modultests kein Fehler mehr auf, so wird mit dem Integrationstest fortgefahren. In ihm wird die Kommunikation zwischen den Modulen getestet und eventuelle Fehler im Systementwurf behoben. Zuletzt wird der Systemtest ausgeführt, der überprüft, ob das entwickelte System die Systemspezifikation erfüllt. Ist das nicht der Fall, so muss die Systemspezifikation oder die entwickelte Software angepasst werden.

Verwendung der fertiggestellten Software: Die Verwendung der Software stellt den Normalzustand des in Betrieb genommenen Systems dar. Er wird

erreicht, wenn alle Tests abgeschlossen sind und das System für seinen Anwendungsbereich freigegeben wurde.

Wartung der Software: Während der Softwarewartung werden bekannte Fehler ausgebessert und Verbesserungen in die Software eingefügt. Dabei kann es sich um die Erweiterung des Funktionsumfangs, generelle Verbesserungen der Bedienbarkeit oder die Veränderung verschiedener Softwarebestandteile handeln.

Der grundlegende, starre Entwicklungsprozess wird in der Praxis häufig durch Veränderungen der Kundenwünsche oder Missverständnisse zwischen den Projektpartnern erschwert. Dadurch ändern sich die Anforderungen während des Entwicklungsprozesses oder neue Anforderungen werden hinzugefügt. Die Konsistenz des Gesamtprojekts muss aber zu jeder Zeit gewährleistet sein. Es darf folglich durch Anforderungsänderungen kein Widerspruch zwischen zwei oder mehr Anforderungen entstehen, da sonst die Implementierung unmöglich ist. Qualität zu messen ist durch die Komplexität der Systeme und Prozeduren faktisch nicht durchführbar. Es besteht einzig die Möglichkeit, die Einhaltung definierter Qualitätsanforderungen zu überwachen.

2.2 TESTARTEN

Man unterscheidet zwei Hauptarten des modellbasierten Tests: den statischen und den dynamischen Test. Beide können sowohl *demonstrativ* als auch *destruktiv* ausgeführt werden. Demonstratives Testen zielt auf eine Testausführung, die zeigen soll, dass das entwickelte System funktioniert. Es wird also so getestet, dass möglichst wenige Tests fehlschlagen. Das steht dem Ansatz des destruktiven Testens gegenüber, der ein System einzig mit dem Ziel ausführt, Fehler zu entdecken. Der destruktive Ansatz ist dem Demonstrativen generell vorzuziehen, da so mehr Fehler gefunden werden können. Die Schwierigkeit bei der Einführung des destruktiven Testens liegt in der Denkweise von Entwicklern begründet, die meist zum demonstrativen Testen neigen und den destruktiven Ansatz leicht als Kritik an ihrer Arbeit missverstehen.

2.2.1 *Statischer Test*

Das statische Testen bezieht sich auf die Analyse des Programmcodes, der zu Grunde liegenden Anforderungen oder des Projektplans. Diese kann in Besprechungen mit vorgegebenem Ablauf, Zielen und Personen stattfinden. Dazu gehören *Audits*, *Reviews*, *Walkthroughs* und *Inspektionen*. Diese Arten des statischen Testens eignen sich besonders gut für nicht automatisierbare oder modellierbare Teile des Testens, sowie für die Plausibilitätsüberprüfung

von Anforderungen und erstelltem Code. Weiterhin kann durch die Beteiligung mehrerer Personen mit unterschiedlichen Qualifikationen ein breiteres Fehlerspektrum gefunden werden.

Code an sich kann durch Software-Werkzeuge statisch analysiert werden. Die Analyse bezieht sich hier auf [Walo1]:

- Syntaktische Informationen (Komplexitätsmaße, Abhängigkeitsgraphen)
- Semantische Informationen (Anomalien im Steuer- und Datenfluss; verwendete, aber nicht deklarierte Variablen)
- Lexikalische Informationen (Prozedurlängen)

Diese Art der statischen Codeanalyse liefert also ohne den Code vorher auszuführen ein Maß für die Codekomplexität und unterstützt den Entwickler gleichzeitig beim Entdecken semantischer Fehler.

Korrektheitsbeweise, wie sie in der formalen Verifikation Einzug gehalten haben, stellen eine weitere Methodik für den statischen Test dar. Hier werden Eigenschaften formuliert, die für ein korrekt arbeitendes System gelten müssen. Es wird dann formal bewiesen, dass das erstellte System diesen Eigenschaften genügt. Problematisch an Korrektheitsbeweisen ist vor allem ihre hohe Komplexität und die damit zusammenhängenden langen bzw. nicht mehr vertretbaren Rechenzeiten. Außerdem muss sichergestellt werden, dass die bewiesenen Eigenschaften die gewünschte Funktionalität vollständig beschreiben.

2.2.2 *Dynamischer Test*

Für den dynamischen Test wird das Testobjekt ausgeführt und die Ausgaben beurteilt. Dabei ist es ab einem genügend großen System nicht mehr praktikabel, alle möglichen Eingangs- und Zustandskombinationen zu testen. Für einen dynamischen Test müssen also Testziele und Testendekriterien formuliert werden. Solche Kriterien werden vor allem durch Metriken bestimmt, auf die in 2.2.4 genauer eingegangen wird. Für einen dynamischen Test muss weiterhin das erwartete, als korrekt angesehene Ergebnis im Vorhinein bekannt sein. Dieses kann dann automatisiert oder manuell, wie in Abb. 3, mit dem Ergebnis des Tests verglichen werden. Weiterhin muss der Testfall reproduzierbar sein, um sicherstellen zu können, dass ein gefundener Fehler durch eine Änderung im System auch behoben wurde (*Retesting*). Außerdem sind reproduzierbare Testfälle für den sogenannten *Regressionstest* unentbehrlich, der nach jeder Änderung im System ausgeführt werden sollte und überprüft, ob durch die Änderung Fehler in bereits getesteten Programmteilen eingeführt oder demaskiert wurden. Ein Test sollte außerdem nicht nur im erlaubten Bereich der Eingabewerte stattfinden, sondern auch Eingabewerte außerhalb der spezifizierten Bereiche abprüfen.

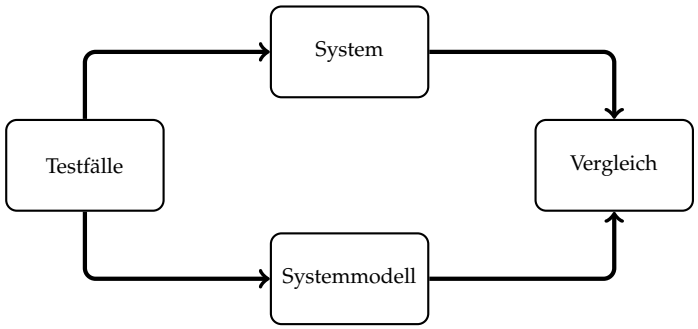


Abbildung 3: Dynamischer Test

Der Testprozess ist streng hierarchisch geregelt und verläuft, nach dem V-Modell (2.1.4), in den Stufen Modul-, Integrations-, System- und Abnahmetest. Die Ermittlung der Testfälle kann in zwei Hauptmethodiken unterteilt werden, den *Black-Box-Test* und den *White-Box-Test*. Beim Black-Box-Test sind keine Informationen über die Implementierung des Systems vorhanden. Testfälle werden allein aus den spezifizierten Bereichen für die Eingangswerte des Systems (2.3.3) bzw. aus dem gewünschten Funktionsumfang erstellt. Der White-Box-Test setzt im Gegensatz dazu ein genaues Verständnis der inneren Struktur (Programmcode, Automaten) des Systems voraus.

2.2.3 *Fehlertypisierung*

In Software können verschiedene Typen von Fehlern [Wal01, IEE90] auftreten, die im Folgenden näher erläutert werden.

> **Definition 2.6: *Fehler* -** Ein Fehler ist der Oberbegriff für eine Fehlhandlung, einen Fehlerzustand und eine Fehlerwirkung und charakterisiert die Nichterfüllung einer definierten Anforderung

Mangel (*defect*): Ein Mangel liegt vor, wenn die Erfüllung der Anforderung nicht angemessen für den erwarteten Gebrauch ist.

Fehlhandlung (*error*): Unter einer Fehlhandlung versteht man die menschliche Handlung, die zum inkorrekten Ergebnis geführt hat. Aus ihr folgt ein Mangel im entstehenden Produkt.

Fehlerzustand (*fault*): Der Fehlerzustand oder innerer Fehler beschreibt die Wirkung einer Fehlhandlung in Software und stellt die Ursache für das Auftreten einer Fehlerwirkung dar.

Fehlerwirkung (*failure*): Die Fehlerwirkung oder der äußerer Fehler ist die sichtbare Manifestation eines Fehlerzustands für den Anwender.

Die Klassifizierung und Dokumentation von Fehlern wird in IEEE 1044 [IEE10] geregelt. Das dient vor allem dazu, einen Überblick, über bereits gefundene Fehler zu erhalten und den Status ihrer Behebung nachvollziehen zu können.

2.2.4 *Metriken*

Beim Software- und Hardwaretest wird Hauptmaß für ihre Qualität meist eine *Metrik* verwendet. Eine Metrik wird passend für den jeweiligen Anwendungsfall definiert und unterliegt klaren Richtlinien, die in der Norm IEEE 1061 [IEE98a] formuliert sind.

> **Definition 2.7: *Qualitätsmetrik* -** „(1) A quantitative measure of the degree to which an item possesses a given quality attribute.
> (2) A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.“ [IEE90]

Vorteile einer wohldefinierten Metrik liegen vor allem in

- der Definition einer messbaren Größe der bereits erreichten Qualität.
- der Möglichkeit aus ihr Akzeptanzkriterien abzuleiten.
- der andauernden Überwachung des Qualitätsniveaus während des Entwicklungsprozesses.
- dem Erkennen möglicher Anomalien oder Probleme im System.

Beim Aufstellen einer Metrik müssen die Qualitätsanforderungen analysiert werden und eine Möglichkeit zur Quantifizierung gefunden werden. Diese wird dann nach einer Kosten-Nutzen-Analyse evaluiert und zuletzt implementiert. Jede Metrik muss mit Hilfe von statistischen Methoden [IEE98a] mit Hinblick auf ihre Aussagekraft validiert werden. Aus dem Softwaretest bekannte Metriken sind die sogenannten Überdeckungsmetriken [Wal01], die beim White-Box-Testen Anwendung finden:

Anweisungsüberdeckung (C_0): Die Anweisungsüberdeckung misst den Prozentsatz der durch den Test ausgeführten Anweisungen im Programm. Bei einer Anweisungsüberdeckung von 100 % wurde jede Codezeile mindestens einmal ausgeführt. Sie sollte immer erreicht werden und kann vor allem dazu genutzt werden, nicht erreichbaren Programmcode aufzuspüren.

Zweigüberdeckung (C_1): Die Zweigüberdeckung beschreibt das Verhältnis von ausgeführten Verzweigungen zu allen im Programm enthaltenen Verzweigungen. Eine Zweigüberdeckung von 100 % durchläuft alle Kanten des Kontrollflussgraphen und führt automatisch zu einer Anweisungsüberdeckung von 100 %. Die Probleme der Zweigüberdeckung bestehen vor allem darin, dass keine Angabe darüber gemacht wird, in welcher Kombination und mit welcher Häufigkeit Verzweigungen auftreten und man Verzweigungen nicht unterschiedlich gewichten kann. Häufig ist es auch schwierig Testfälle so zu gestalten, dass alle Zweige aufgerufen werden.

Bedingungsüberdeckung (C_2): Die Bedingungsüberdeckung betrachtet, anders als die Zweigüberdeckung, nicht nur einzelne Verzweigungen, sondern auch die einzelnen Bedingungen, die zu einer Verzweigung führen. Dabei bestehen verschiedene Möglichkeiten diese in Betracht zu ziehen. Bei der einfachen Bedingungsüberdeckung werden alle Bedingungen, die innerhalb einer Verzweigungsanweisung auftreten, als atomare Bestandteile betrachtet. Logische Verknüpfungen zwischen ihnen werden vernachlässigt. Daher reicht es für die einfache Bedingungsüberdeckung aus, wenn jedes Prädikat einmal wahr bzw. falsch wurde.

Überdeckung aller Bedingungskombinationen (C_3): Hierbei handelt es sich um die vollständigste Art von Kontrollflussüberprüfung. Alle Prädikate innerhalb einer Verzweigung werden in allen möglichen Kombinationen mindestens einmal getestet um eine 100 % Abdeckung zu erhalten. Die Anzahl der auszuführenden Testfälle wächst dabei exponentiell um dem Faktor 2^n, wobei n die Anzahl der Prädikate angibt.

Pfadüberdeckung (C_4): Die Pfadüberdeckung definiert ein Maß, das zur Bewertung der Schleifen innerhalb eines Programms herangezogen werden kann. Es werden alle möglichen Pfade im Kontrollflussgraphen des Programms durchlaufen. Eine Pfadüberdeckung von 100% ist in der Praxis wegen der Vielzahl der möglichen Kombinationen meist nicht zu erreichen.

Bei den beschriebenen Metriken handelt es sich um kontrollflussorientierte Techniken, die keine Vorgaben über die Art des Tests machen. Sie dienen einzig der Bewertung vorhandener Testfälle und definieren nicht, wie diese erzeugt wurden. Am besten geeignet sind diese Techniken zum Testen einzelner Softwaremodule. Das Zusammenspiel verschiedener Module beim Systemtest ist damit allerdings nicht zu bewerten.

2.3 MODELLBASIERTES TESTEN

Beim modellbasierten Testen werden die Testfälle aus einem vorher formulierten Modell abgeleitet [Vig10]. Das Konzept des modellbasierten Testens soll im Folgenden genauer dargestellt werden.

2.3.1 *Modellbegriff*

Das Wort Modell entstammt dem lateinischen Wort *modulus (Maß, Takt, Melodie)* und wird heute für die Beschreibung der abstrakten Darstellung der Realität verwendet.

> **Definition 2.8: *Modell* -** Eine abstrakte Beschreibung der Struktur oder des Verhaltens eines konkreten Systems.

Modelle sind also nur darstellende Hilfsmittel, um Systeme abzubilden. Sie dienen entweder dazu, vorhandene Systeme zu beschreiben und deren Verhalten zu simulieren (z.B. Klimamodelle, Planetenmodelle, Erdkrustenmodelle) oder dazu Systeme zu beschreiben, die später manuell oder automatisch implementiert werden sollen (z.B. Hardwarebeschreibungssprachen, Programmiersprachen, Automaten).

2.3.2 *Orakel*

Ein Orakel im Bereich des modellbasierten Softwaretests manifestiert sich in der Literatur in verschiedenen Ausprägungen. Ein guter Überblick über die zur Zeit eingesetzten Orakel wird von Baresi und Young gegeben [BY01]. Im Folgenden wird eine kurze Zusammenfassung der gängigsten Konzepte im Bereich von Orakeln präsentiert. Der größte Teil der Testorakel basiert auf *Assertions*, die in den Programmcode eingebettet werden. Bei Assertions handelt es sich um Annahmen oder Eigenschaften, die ein Programmierer über das Verhalten der Software vor, während und nach dem Schreiben des Programmcodes formuliert. Eine Assertion besteht aus einem Ereignis, das eintreten muss, damit die Assertion ausgewertet wird und einem Verpflichtungsteil, der dann gelten muss. Zum Formulieren von Assertions wurden spezielle Sprachkonstrukte (z.B. für C oder Java) definiert.

Orakel können außerdem aus Spezifikationssprachen, wie Z [Spi89], abgeleitet werden. Aus ihnen kann sowohl C-Code als auch objektorientierter Code erzeugt werden. Weiterhin ist es möglich, Orakel aus temporaler Logik zu generieren [DR96]. Diese bewerten Sequenzen aus Ereignissen mit Hinblick auf ihre Korrektheit.

Im Rahmen dieser Arbeit wird ein Orakel als Verhaltensmodell für ein System angesehen. Es ist also dazu in der Lage, das Verhalten eines entwickelten Systems anhand der Eingaben und internen Zustände vorherzusagen. Dadurch kann vor oder während der Testausführung eines Prototypen dessen korrektes, also spezifikationsgetreues, Ausgangsverhalten vorhergesagt werden. Ein Tester muss folglich für die Beurteilung der Korrektheit des Tests nicht mehr

die Spezifikation kennen, sondern kann die Testergebnisse mit den durch das Orakel vorhergesagten vergleichen. So wird das Orakel zum Systemmodell aus Abb. 3.

2.3.3 *Klassifikationsbäume*

Die Klassifikationsbaummethode beschreibt einen strukturierten Black-Box-Testansatz, der 1993 von Grochtmann et al. [GGWA93] vorgestellt wurde. In ihrem Zentrum steht der Klassifikationsbaum, der eine strukturierte Darstellung für *Äquivalenzklassen* durch einen azyklischen Graph bietet. Eine Äquivalenzklasse besteht aus einer Menge disjunkter Wertebereiche oder aus einer Menge einzelner, disjunkter Werte, für die ein System, das spezifikationsgemäß arbeitet, ein äquivalentes Verhalten aufweist. Durch diese Darstellung wird der Eingabebereich abstrahiert und strukturiert. Unterstützt wird der Aufbau von Klassifikationsbäumen durch das Tool *Classification Tree Editor for Embedded Systems* (CTE/ES). Abb. 4 zeigt einen Klassifikationsbaum mit zugehörigen Testfällen, der für einen Autositz aufgebaut wurde. Jeder Klassifikationsbaum besteht aus den vier Elementen Wurzelknoten (*Seat*), Kompositionen (*Sensors, Switches*), Klassifikationen (*Position, Speed, Door, All-Triggers*) und den Äquivalenzklassen. Um Testfälle aus einem Klassifikationsbaum abzuleiten, wird dieser durch eine Tabelle erweitert, in der für den jeweiligen Testfall ausgewählte Äquivalenzklassen durch einen Punkt gekennzeichnet werden. Die Testfälle sind je nach Intention des Entwicklers entweder als unabhängig voneinander oder als zeitliche Abfolge zu betrachten. Die Extraktion von Testfällen aus einer

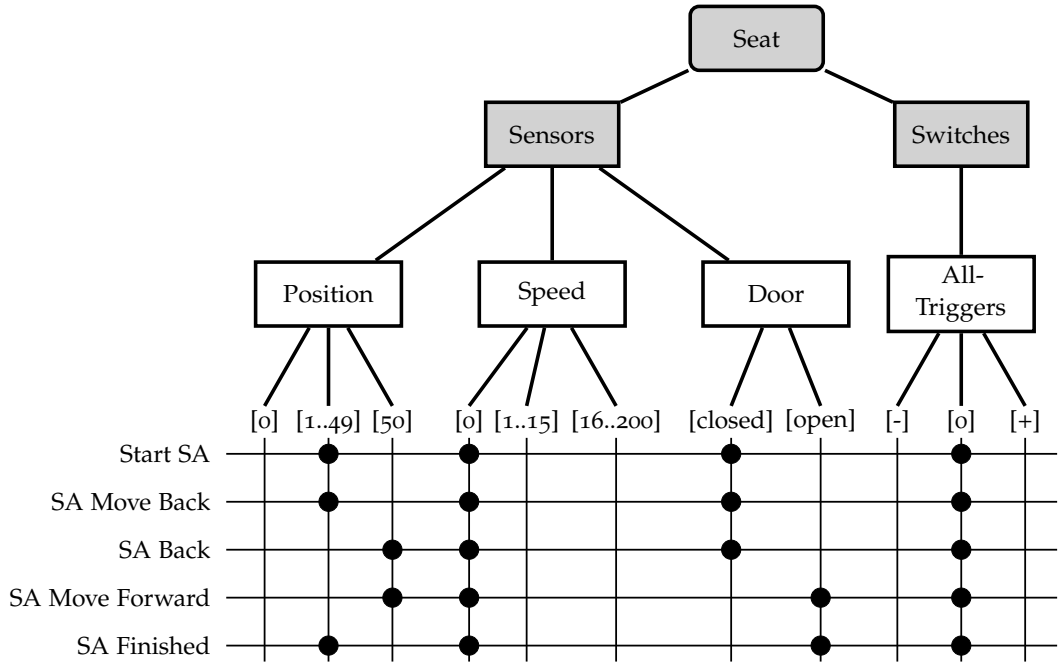


Abbildung 4: Klassifikationsbaum

Spezifikation mit Hilfe eines Klassifikationsbaums verläuft nach folgendem Schema:

1. Die Spezifikation muss evaluiert werden und alle Klassifikationen mit ihren Äquivalenzklassen identifiziert werden.
2. Der Klassifikationsbaum muss aufgebaut werden.
3. Die Tabelle, die die Testfälle beschreibt, muss ausgefüllt werden. Dabei finden verschiedene Heuristiken Anwendung.
4. Alle möglichen Testfälle werden aus dem Klassifikationsbaum erzeugt.

Um den Klassifikationsbaum auch für Stresstests verwenden zu können, ist es sinnvoll Äquivalenzklassen außerhalb der spezifizierten Bereiche hinzuzufügen. Dadurch können Stresstests ausgeführt oder Sensordefekte simuliert werden.

3

FORMALE DARSTELLUNGEN

Eine formale Methode bedient sich einer formalen Darstellung, die automatisiert bewertet, ausgewertet oder transformiert werden kann. Formale Darstellungen müssen eindeutig interpretierbar sein und basieren daher stets auf einer eindeutigen Semantik. Die in dieser Arbeit verwendeten formalen Sprachen und Logiken werden im Folgenden genauer behandelt.

3.1 COMMUNICATING SEQUENTIAL PROCESSES

Der Begriff CSP wurde von Hoare eingeführt [Hoa04]. CSPs beschreiben mittels einer definierten Semantik nebenläufige Prozesse P.

> **Definition 3.1:** ***Prozess (process)*** **-** Ein Prozess ist die mathematische Abstraktion der Interaktion eines Systems und seiner Umgebung

Prozesse werden durch das Auftreten von Ereignissen $e_1 \dots e_n$ gestartet und beendet. Zu jedem Zeitpunkt kann ausschließlich ein Ereignis eintreten.

> **Definition 3.2:** ***Ereignis (event)*** **-** Ein Ereignis stellt eine atomare Aktion ohne Dauer dar. Eine Verkettung von Ereignissen kann zur Beschreibung komplexerer Sachverhalte genutzt werden.

Jedes System wird durch eine Menge von Prozessen charakterisiert und kann innerhalb eines definierten Alphabets $\alpha P = \{e_1 \dots e_n\}$ agieren.

> **Definition 3.3:** ***Alphabet*** **-** Die Menge von Ereignisnamen, die für die Beschreibung eines Systems relevant sind.

Die einfachste Art einen Prozess zu beschreiben ist es, eine Abfolge von Ereignissen, wie z.B. $P = e_1 \rightarrow e_2 \rightarrow P$, zu definieren. Abb. 5 zeigt beispielhaft einen Automaten, der im Folgenden zur Erläuterung der Konzepte von CSPs dient. Der Automat lässt sich durch folgende Prozesse beschreiben:

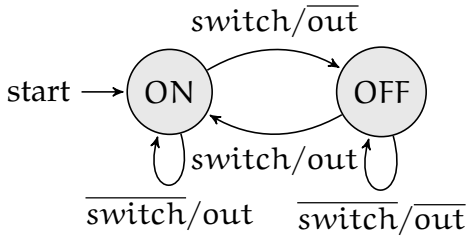


Abbildung 5: Beispielautomat

$P_{INIT} = start \rightarrow P_{ON}$;
$P_{ON} = (\overline{switch} \rightarrow out \rightarrow P_{ON} \mid switch \rightarrow \overline{out} \rightarrow P_{OFF})$;
$P_{OFF} = (\overline{switch} \rightarrow \overline{out} \rightarrow P_{OFF} \mid switch \rightarrow out \rightarrow P_{ON})$

Das Alphabet der Prozesse ist wie folgt definiert:

$\alpha P_{INIT} = \{start\}$; $\alpha P_{OFF} = \alpha P_{ON} = \{switch, \overline{switch}, out, \overline{out}\}$

P_{ON} beschreibt das Verhalten des Automaten im Zustand ON und P_{OFF} im Zustand OFF. Der Initialzustand wird durch P_{INIT} festgelegt. Durch | wird eine Auswahl zwischen den Transitionen getroffen, die vom jeweiligen Zustand ausgehen. Da der Automat ein deterministisches Verhalten zeigt, kann zu jedem Zeitpunkt auch immer nur eine der Bedingungen für die jeweiligen Transitionen wahr werden. Die Umgebung startet den Automaten mit dem Ereignis start. Er befindet sich dann immer in einem der beiden Zustände P_{ON} oder P_{OFF}. Wird der Automat ausgeführt, so ergeben sich dadurch unendlich viele mögliche Ereignisfolgen, wie z.B. $t = \langle e_1, e_2, e_1, e_3 \rangle$.

Definition 3.4: ***Ereignisfolge (trace)*** **-** Eine Ereignisfolge des Verhaltens eines Prozesses ist eine endliche Abfolge von Ereignissen, die bei der Ausführung eines Prozesses bis zu einem bestimmten Zeitpunkt eingetroffen sind.

Diese können wie folgt miteinander konkateniert werden:

$\langle e_1, e_2 \rangle^\frown \langle e_1, e_3 \rangle = \langle e_1, e_2, e_1, e_3 \rangle$

Eine gültige Ereignisfolge des Automaten ist beispielsweise:

$t = \langle start, \overline{switch}, out, switch, \overline{out} \rangle$

Diese Ereignisfolge kann durch den Reduktionsoperator $\upharpoonright$ auf eine Teilmenge der Ereignisse reduziert werden $t \upharpoonright \{\overline{out}, out\} = \langle out, \overline{out} \rangle$.

Die Möglichkeit zu definieren, ob ein Ereignis eine Eingabe in den Automaten oder eine erzeugte Ausgabe beschreibt, besteht in CSPs ebenfalls. Dafür werden Kanäle definiert, über die Werte weitergegeben werden. Die Kommunikation in CSPs besteht immer aus der Angabe des Kanals c und dem übertragenen Wert

v, sowie der Information, ob es sich um eine Eingabe $c?v$ oder eine Ausgabe $c!v$ handelt. Die Prozesse des Automaten können also wie folgt geändert werden:

$P_{INIT} = start?0 \rightarrow P_{ON};$
$P_{ON} = (switch?0 \rightarrow out!1 \rightarrow P_{ON} \mid switch?1 \rightarrow out!0 \rightarrow P_{OFF});$
$P_{OFF} = (switch?0 \rightarrow out!0 \rightarrow P_{OFF} \mid switch?1 \rightarrow out!1 \rightarrow P_{ON})$

Daraus ergibt sich folgendes Alphabet für die Kanäle:

$\alpha start = \alpha switch = \alpha out = \{0, 1\}$

Der Automat kann auf diese Weise mit seiner Umgebung kommunizieren und mit weiteren Systemen verbunden werden, die wiederum durch Prozesse definiert sind.

Ein großes Problem bei der Verwendung von CSPs stellt die Tatsache dar, dass Zeitpunkte nicht genau referenziert werden können. Alle Prozesse bestehen ausschließlich aus einer Aneinanderreihung von Ereignissen, die während des Prozesses irgendwann einmal auftreten. Für Hardwarebeschreibungen sind CSPs daher ungeeignet, da die Anzahl der Takte zwischen zwei Ereignissen nicht oder nur auf Umwegen spezifiziert werden kann. Das äußere Verhalten eines eingebetteten Systems (4.2.1) kann durch CSPs allerdings repräsentiert werden.

3.2 UML-DIAGRAMME

Generell umfasst die Unified Modeling Language (UML) 13 Diagrammarten, die sich in Struktur- und Verhaltensdiagramme gliedern. Im Rahmen dieser Arbeit werden davon ausschließlich *Aktivitätsdiagramme* verwendet.

Ein Aktivitätsdiagramm dient der Veranschaulichung von Abläufen für einen bestimmten Anwendungsfall. Dabei kann es sich um die sequentielle Abarbeitung von Testfällen, einen Geschäftsprozess oder einen speziellen Programmdurchlauf handeln. Abb. 6 zeigt die für diese Arbeit relevanten Elemente von Aktivitätsdiagrammen. Die beiden Hauptelemente eines Aktivitätsdiagramms bilden *Aktion* und *Objekt*. Eine Aktion stellt eine ausführbare Tätigkeit dar, während ein Objekt Daten enthält. Der *Objektfluss* bezeichnet daher auch die Übertragung von Daten zu einem anderen Objekt bzw. die Verwendung von Daten durch eine Aktion. Der *Kontrollfluss* wird mit Hilfe von *Tokens* modelliert. Tokens bewegen sich entlang des Kontrollflusses und werden immer von der jeweiligen aktiven *Aktion* gehalten. Nach Beendigung der Aktion wird das Token wieder freigegeben und kann zur nächsten Aktion fließen. Hierbei ist es möglich, dass ein oder mehrere Tokens gleichzeitig im Aktivitätsdiagramm aktiv sind. Eine *Gabelung* stellt die Aufspaltung eines Tokens in mehrere dar,

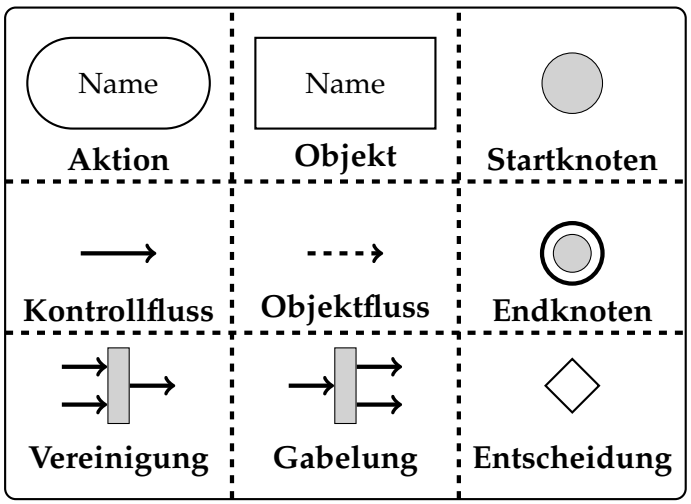


Abbildung 6: Elemente in Aktivitätsdiagrammen

während eine *Vereinigung* mehrere Tokens zu einem vereint sobald alle eingehenden Token vorhanden sind. Die Modellierung von nebenläufigen Abläufen wird so möglich. Eine Entscheidung lenkt den Fluss eines Tokens abhängig von einer Bedingung. Aktivitätsdiagramme beginnen mit einem Startknoten, der ein einziges Token auslöst und enden, sobald ein Token einen Endknoten erreicht.

3.3 EIGENSCHAFTSBESCHREIBUNGSSPRACHEN

Einen integralen Bestandteil dieser Arbeit bildet das Formulieren von Anforderungen mit Hilfe von Eigenschaftsbeschreibungssprachen. Diese bieten eine eindeutige Semantik und eignen sich daher ideal für die rechnergestützte Verarbeitung. Die einzelnen Sprachen unterscheiden sich in ihrer Aussagekraft und in ihrem Anwendungsgebiet. Dabei gibt es Sprachen, wie System Verilog Assertions (SVA) [IEE07], die Property Specification Language (PSL) [IEE05] oder die Interval Language (ITL) [Bor09], die ausschließlich im Hardwareentwurf Anwendung finden und solche, wie Z [Spi89], das ausschließlich im Softwareentwurf verwendet wird. Im Folgenden werden die für diese Arbeit relevanten Eigenschaftsbeschreibungssprachen genauer erklärt. Diese wurden gewählt, da sie mit bestehenden Werkzeugen aus der formalen Hardwareverifikation, die die für diese Arbeite relevanten Algorithmen implementieren, verwendet werden können.

3.3.1 *Linear Temporal Logic*

Bei Linear Temporal Logic (LTL) [Pnu77] handelt es sich um eine Logik, die zeitliche Zusammenhänge innerhalb eines Systems, das als Automat model-

liert werden kann, ausdrückt. LTL-Eigenschaften sind gültig in Bezug auf die Ausführungspfade eines Systems. In LTL können Literale oder Prädikate mit *booleschen Operatoren* $\neg, \vee, \wedge$ sowie durch Implikationspfeile $\rightarrow$ miteinander verbunden werden. LTL stellt außerdem folgende *zeitliche Operatoren* zur Verfügung:

- $X\phi$: ϕ muss im nächsten Zustand halten (*next*).
- $G\phi$: ϕ muss in allen Zuständen halten (*globally*).
- $F\phi$: ϕ muss irgendwann in der Zukunft halten (*finally*).
- $\psi U\phi$: ψ muss mindestens halten bis ϕ folgt (*until*).
- $\psi R\phi$: ψ muss halten bis ϕ folgt. Wenn ϕ nie folgt, muss ψ immer halten. (*release*).

Die Eigenschaft, in der aus einem *Request* ein *Grant* folgen soll, hat in LTL also folgende Form:

$$G((\mathtt{Request} =' 1') \rightarrow X(\mathtt{Grant} =' 1'))$$

Ein Problem von LTL ist vor allem, dass die Formulierung komplexer Sachverhalte in unübersichtlichen LTL-Ausdrücken resultiert.

3.3.2 *Interval Language*

ITL [Boro9] wurde von der Siemens AG (später Infineon Technologies AG und OneSpin Solutions GmbH) als Eigenschaftsbeschreibungssprache entwickelt, um ideal mit dem intern entwickelten Tool zu harmonieren. ITL hat folgende Hauptattribute [Obe10]:

- Eigenschaften werden als Implikationen geschrieben. Aus einer Annahme folgt eine Verpflichtung.
- Systemeigenschaften halten global.
- In Eigenschaften werden Signale über die explizite Angabe von Zeitpunkten in einem endlichen Zeitfenster referenziert.
- Die Syntax orientiert sich an der der Very High Speed Integrated Circuit Hardware Description Language (VHDL) oder an der von Verilog.
- Spezielle Befehlserweiterungen zur Vollständigkeitsanalyse sind vorhanden.

Zur Verdeutlichung ist im Folgenden eine ITL-Eigenschaft gegeben, die beschreibt, dass auf ein *Request* immer ein *Grant* folgen muss:

```
property Beispiel is
assume:
  at t: Request = '1';
prove:
  at t+1: Grant = '1';
end property;
```

Die Eigenschaft besteht aus dem Annahmeteil (*assume*) und dem Verpflichtungsteil (*prove*). Trifft die Annahme zu, so muss auch die Verpflichtung zutreffen. Eine ITL-Eigenschaft beschreibt also immer eine Implikation vom Annahme- auf den Verpflichtungsteil.

Die Zeitbasis von ITL wird durch einen diskreten Zeitpunkt t angegeben. Dieser kann mit $t \pm \mathit{offset}$ bzw. durch die Funktionen `PREV` und `NEXT` relativ zu den übrigen Zeitpunkten in der Eigenschaft verschoben werden. Da ITL für die formale Verifikation von Hardwareschaltungen entwickelt wurde, bezieht sich die Zeit in diesem Bereich auf den Systemtakt. Im Rahmen dieser Arbeit wird ITL zum Formulieren von Eigenschaften für eingebettete Systeme eingesetzt. Durch t wird in diesem Fall nicht der Takt definiert, sondern das Fortschreiten der Zeit durch das Auftreten eines Ereignisses.

Der Vorteil von ITL ist seine bessere Lesbarkeit im Vergleich zu LTL. Die klare Teilung einer Eigenschaft in Annahme und Verpflichtung sowie die strukturierte Referenzierung von Zeitpunkten führen zu einer übersichtlichen Darstellung.

3.3.3 *Eigenschaften*

Für die Formulierung funktionaler Anforderungen eines Systems können Eigenschaften verwendet werden. Im Rahmen dieser Arbeit wird sowohl ITL als auch LTL verwendet. Da die verwendeten Eigenschaften immer nur von einem Zeitpunkt auf den nächsten verweisen, können beide Darstellungsformen äquivalent eingesetzt werden. Dabei werden je nach Anwendungszweck unterschiedliche Eigenschaftsarten unterschieden.

Sicherheitseigenschaften: Jedes System muss sich stets in einem als sicher geltenden Zustandsraum befinden. Dabei spezifizieren Sicherheitseigenschaften, dass nur sichere Zustände erreicht werden sollen. Unsichere werden durch sie ausgeschlossen. Sicherheitseigenschaften müssen daher global und immer gelten und werden in LTL mit $G\phi$ beschrieben, wobei es sich bei ϕ um einen booleschen Ausdruck handelt, der Literale und Prädikate enthält, aber keine temporalen Operatoren. Für diese Arbeit ist die Beschreibung des zeitlichen Bezugs zwischen Ereignissen allerdings relevant. Daher wird hier die Sicherheitseigenschaft als $G(\phi \rightarrow \psi)$ beschrieben, wobei die Verwendung des LTL-X-Operators in ϕ bzw. ψ gestattet ist. ϕ und ψ können also Literale

zu unterschiedlichen Zeitpunkten enthalten. Eine Eigenschaft in der Form $G(\phi \rightarrow \psi)$ führt dazu, dass wenn ϕ auf der linken Seite der Implikation wahr wird, ψ auch gelten muss. Ansonsten wird die Sicherheitseigenschaft verletzt.

Lebendigkeitseigenschaften: Um die Erreichbarkeit von Zuständen innerhalb eines Zustandsraums sicher zu stellen, werden Lebendigkeitseigenschaften verwendet. Wenn nicht anders beschrieben, ist es unwesentlich, in welchem Zeitfenster ein Zustand erreicht werden kann. Lebendigkeitseigenschaften werden in LTL mit $F\phi$ ausgedrückt. Im Rahmen dieser Arbeit können ausschließlich Sicherheitseigenschaften verwendet werden. Lebendigkeitseigenschaften werden mit Sicherheitseigenschaften geeignet umschrieben [Boro9]. Aus folgender Lebendigkeitseigenschaft werden z.B. zwei Sicherheitseigenschaften, bei denen ein weiterer Zustand *In_Bewegung* eingefügt wurde.

Lebendigkeitseigenschaft:
$G(Start \rightarrow F(Ziel))$

Sicherheitseigenschaften:
$G(Start \rightarrow X(In\_Bewegung))$
$G(In\_Bewegung \rightarrow X(In\_Bewegung \vee Ziel))$

Neben diesen beiden Eigenschaftsarten existieren noch weitere (z.B. Fairness-, Fortdauereigenschaften), die hier nicht genauer behandelt werden.

3.3.4 *Eigenschaftsbeschreibungsstile*

Eigenschaften können durch eine Eigenschaftsbeschreibungssprache unterschiedlich ausgedrückt werden und trotzdem das selbe Verhalten beschreiben. Es handelt sich bei Eigenschaften also nicht um eine kanonische Repräsentation des Systemverhaltens. Es ist daher sinnvoll, Richtlinien für das Formulieren von Eigenschaften aufzustellen. Diese sollten sich am gewünschten Einsatzzweck des entstehenden Eigenschaftssatzes orientieren. Im Folgenden werden bekannte Eigenschaftsbeschreibungsstile kurz vorgestellt:

Monitorstil: Der Monitorstil wurde 2002 von Shimizu [Shio2] vorgestellt.

> **Definition 3.5: *Monitor* -** „A monitor is an observer in a group of interacting modules, or agents which communicate via a set of protocol rules“ [Shio2]

Ein Monitor überwacht die Kommunikation zwischen Modulen. Die Ausgänge der Module stellen seine Eingänge dar und sein Ausgang ist ein einzelner boolescher Wert, der angibt, ob die Kommunikation zwischen den überwachten Modulen korrekt verläuft. In Monitoreigenschaften wird also ausschließlich das

korrekte Verhalten des Systems spezifiziert. Der Vorteil liegt in der Möglichkeit, Eigenschaften, die einen großen Teil des Verhaltens abdecken, in mehrere kleinere zu unterteilen, wobei jede einen Monitor darstellt. Die Konjunktion der Monitorausgänge repräsentiert den Monitor für die ursprüngliche Eigenschaft. Monitore sind außerdem ausführbare Verhaltensbeschreibungen, die direkt mit dem System verbunden werden können und so dessen Funktionsfähigkeit auch während des Betriebs überwachen können. Die Beschreibung des korrekten Umgebungsverhaltens kann ebenfalls über Monitore erfolgen.

Transaktionsstil: Eigenschaften, die im Transaktionsstil verfasst sind, argumentieren über ein abgeschlossenes Zeitfenster mit Anfangs- und Endzuständen. Dabei kann es sich um Zustände handeln, die in der Spezifikation explizit erwähnt wurden oder um solche, die beim Schreiben der Eigenschaften als virtuelle Zustände eingeführt wurden. Der Transaktionsstil setzt also voraus, dass sich eine Spezifikation in einzelne Berechnungsschritte separieren lässt. Die Menge aller Berechnungsschritte stellt die gesamte Spezifikation als Zustandsautomaten dar. Ein System, das nur die beiden Transaktionen *lesen* und *schreiben* kennt, kann diese in der Realität in jeder Abfolge ausführen. Die Korrektheit des Systems muss für alle möglichen Abfolgen sichergestellt sein.

3.3.5 *Parallelismus und Nebenläufigkeit*

Zum Verständnis des Zeitbegriffs, der durch ITL und LTL festgelegt wird, ist eine Auseinandersetzung mit den Begriffen Parallelismus und Nebenläufigkeit [Hoa04] nötig. Abb. 7 zeigt die Übersetzung eines *Statecharts* in einen parallelen und einen nebenläufigen Automaten.

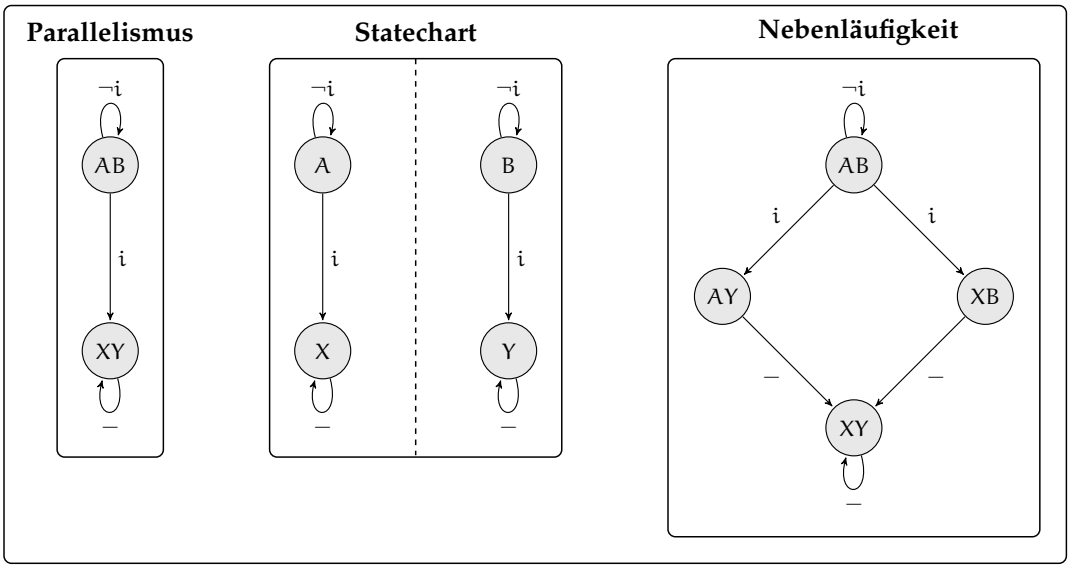


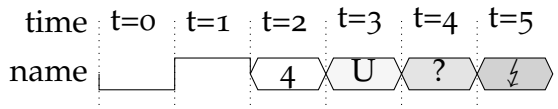
Abbildung 7: Parallelismus und Nebenläufigkeit

Parallelismus wird in dieser Arbeit immer als *synchroner Parallelismus* interpretiert. Die parallele Ausführung bei synchronem Parallelismus unterliegt einem gemeinsamen Takt. Synchron parallel arbeitende Teilsysteme, die als Automat repräsentierbar sind, lassen sich immer in einen gemeinsamen Automaten übersetzen. Tritt ein Ereignis i auf, dann wechseln alle Automaten vom Zustand AB in XY. Dieser Wechsel geschieht bei paralleler Ausführung gleichzeitig in einem Zeitschritt. Beide Transitionen im Statechart werden also gleichzeitig ausgeführt. Bei nebenläufiger Ausführung werden die Transitionen nacheinander in zufälliger Reihenfolge ausgeführt.

Sowohl LTL als auch ITL erlauben die synchron parallele Beschreibung des Verhaltens des Statecharts. Die Formulierung nebenläufigen Verhaltens ist nur über den Umweg über *Random-Inputs* möglich. Random-Inputs stellen Eingänge dar, die Zufallswerte liefern. Dadurch ist die zufällige Auswahl des Pfades durch den nebenläufigen Automaten modellierbar. Im Rahmen dieser Arbeit werden Eigenschaften ausschließlich so geschrieben, dass sie die parallele Ausführung beschreiben. Auf das Einfügen von Random-Inputs wird verzichtet.

3.3.6 *Darstellung von Ereignisfolgen*

Im Rahmen dieser Arbeit werden Ereignisfolgen häufig graphisch dargestellt. Die hierfür verwendet Notation wird anhand der folgenden Beispielereignisfolge erläutert.



Der Wert des Signals $name$ ist zum Zeitpunkt $t = 0$ eine logische $'0'$, zum Zeitpunkt $t = 1$ eine logische $'1'$, zum Zeitpunkt $t = 2$ ein ganzzahliger Wert, zum Zeitpunkt $t = 3$ ein zu Beginn der Ausführung undefinierter Wert, zum Zeitpunkt $t = 4$ ein durch eine Spezifikationslücke undefinierter Wert und zum Zeitpunkt $t = 5$ ein Widerspruch, der dazu führt, dass der Wert nicht eindeutig bestimmbar ist.

Teil II

ANFORDERUNGEN UND PRODUKTLINIEN

In diesem Teil werden Anforderungen und Produktlinien eingeführt und ein Bezug zwischen beiden hergestellt. Dazu werden zuerst funktionale und nicht-funktionale Anforderungen erläutert und die Voraussetzungen für ein gutes Anforderungsdokument definiert. Im Anschluss daran wird der Begriff der Produktlinie und die Darstellung der Variabilität genauer erläutert. Außerdem wird beschrieben, wie man Anforderungen für eine Produktlinie formulieren kann und auf das Testen von Produktlinien eingegangen. Weiterhin werden die Algorithmen, die dieser Arbeit zu Grunde liegen, eingeführt. Dazu zählt die Generierung von Cando-Objekten. Diese basiert auf der automatisierten Übersetzung von Eigenschaftssätzen in ein ausführbares Modell. Weiterhin wird ein Ansatz zur Vollständigkeitsbewertung beschrieben. Dadurch wird es möglich, Aussagen über die Lücken und Widersprüche in Eigenschaftssätzen zu treffen und diese genau zu benennen.

4

DARSTELLUNG VON ANFORDERUNGEN

Im Zentrum dieser Arbeit steht die Überprüfung von Anforderungen und die Generierung von Orakeln. Grundlage dessen bildet die *Spezifikation* aus Abb. 8. Hier werden Anforderungen beschrieben, die das Verhalten und die

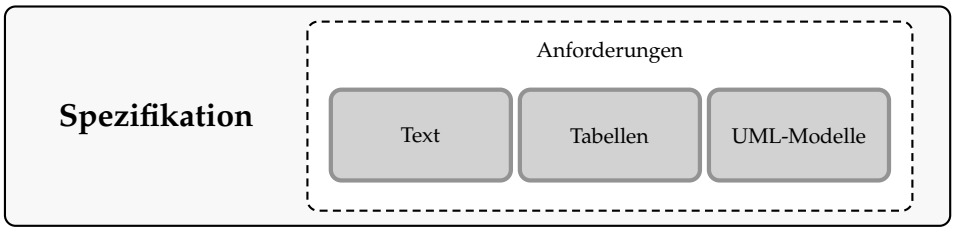


Abbildung 8: Spezifikation

Struktur des Systems repräsentieren. Die Spezifikation besteht aus einem oder mehreren Anforderungsdokumenten, die auf unterschiedliche Art und Weise (4.2) aufgebaut sein können.

4.1 REQUIREMENTS ENGINEERING

Das *Requirements Engineering* zielt auf die Erhebung, Validierung und Verwaltung von Anforderungen an ein Produkt ab. Eine Anforderung (*Requirement*) beschreibt eine einzelne oder eine Menge von Eigenschaften, die das zu entwickelnde System aufweisen soll.

> **Definition 4.1:** ***Anforderung*** **-** „(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.“ [IEE90]

Anforderungen werden durch Marketingüberlegungen, Kostenaufwand und technische Machbarkeit beeinflusst. Man unterscheidet zwei Klassen von Anforderungen, nämlich *nicht-funktionale* und *funktionale* Anforderungen. Nichtfunktionale Anforderung beschreiben vor allem äußere Faktoren wie Temperaturstabilität, Aussehen, Maximalgewicht oder Größe eines Produkts. Eine funktionale Anforderung macht im Gegensatz dazu Vorgaben an das Verhalten eines Systems.

Definition 4.2: ***Funktionale Anforderung*** **-** „A requirement that specifies a function that a system or system component must be able to perform.“ [IEE90]

Stellenweise kommt es vor, dass Anforderungen nicht eindeutig zu einer der beiden Klassen zuzuordnen sind. Das ist beispielsweise bei Anforderungen an die Sicherheit oder Bedienbarkeit eines Produkts der Fall. Bei beiden handelt es sich um Anforderungen, die nicht direkt Aussagen über das gültige Verhalten des Produkts treffen, bei der Entwicklung der Funktionalität des Systems aber trotzdem Einfluss auf diese haben können. IEEE 830 [IEE98b] beschreibt Charakteristiken einer „guten“ Anforderungsspezifikation, wie folgt:

Korrektheit: Eine Anforderungsspezifikation ist nur dann korrekt, wenn nur solche Anforderungen in ihr formuliert sind, die dann auch für das spätere System zutreffen müssen. Dabei handelt es sich um eine auf den ersten Blick eher triviale Definition von Korrektheit. Eine solche Vorgabe ist nötig, da sich Anforderungen während initialen Projektphasen ändern können und erweitert bzw. gekürzt werden. Diese Form von Korrektheit kann ausschließlich durch Projektbeteiligte beurteilt werden. Eine automatisierte Beurteilung der Korrektheit durch formale Methoden ist nicht möglich.

Eindeutigkeit: Die Eindeutigkeit bezieht sich auf Formulierungen und Darstellungsarten von Anforderungen. Eine Anforderung ist nur dann eindeutig, wenn sie von allen Projektbeteiligten ausschließlich auf eine einzige Art interpretiert werden kann. Das setzt voraus, dass Anforderungen auf eine Weise formuliert werden, die für Entwickler, Projektmanager und Kunden nur eine Interpretation zulässt. Es ist fast unmöglich, alle Mehrdeutigkeiten aus Spezifikationen zu eliminieren, die in natürlicher Sprache verfasst sind. Eine Lösung hierfür stellen Spezifikationssprachen oder Werkzeuge dar, die in der Lage sind, Anforderungen graphisch, tabellarisch oder in geordneter Textform darzustellen. Auch hier gilt allerdings die Prämisse, dass Personen mit nicht technischem Hintergrund die jeweilige Repräsentation ohne großen Schulungsaufwand verstehen können müssen.

Vollständigkeit: Eine Spezifikation ist vollständig, wenn sie drei Hauptpunkte erfüllt:

1. Alle Anforderungen, die sich nicht direkt auf das Systemverhalten beziehen, müssen erwähnt und behandelt werden. Dazu zählen Funktionalität, Leistungsfähigkeit, Designvorgaben, Attribute oder externe Verbindungen.
2. Die Systemantwort auf alle möglichen Eingangskombinationen und internen Zustände, unabhängig davon, ob diese als gültig oder ungültig angesehen werden, muss spezifiziert sein.

3. Alle Zeichnungen, Tabellen und Diagramme müssen durch Text ergänzt bzw. beschrieben werden und Fachausdrücke sowie Maßeinheiten definiert werden.

Im Verlauf dieser Arbeit wird Vollständigkeit mit Bezug auf die Systemantwort behandelt.

Konsistenz: Nur solche Anforderungsspezifikationen sind konsistent, die keinen Konflikt von zwei oder mehr Anforderungen enthalten. Es gibt drei Hauptgruppen möglicher Konflikte:

1. Beobachtbare Konflikte: Ein Beispiel ist die Anforderung, dass die Information über einen aufgetretenen Fehler als Hexadezimalzahl ausgegeben werden soll, wenn zu Beginn des Anforderungsdokuments festgelegt wurde, dass nur Dezimalzahlen ausgegeben werden dürfen.
2. Logische oder temporale Konflikte: Ein Beispiel ist ein Ausgang, der an einer Stelle als Addition zweier Eingänge und an einer anderen als Multiplikation beschrieben wird oder eine Anforderung beschreibt z.B., dass B auf A folgt und eine andere, dass beide gleichzeitig auftreten.
3. Zwei oder mehr Anforderungen benutzen für dasselbe beobachtbare Objekt verschiedene Terminologien.

Der Konsistenzbegriff wird in dieser Arbeit mit Bezug auf die logischen und temporalen Konflikte in Anforderungen verwendet.

Priorisierbarkeit: Normalerweise sind nicht alle Anforderungen in einer Spezifikation gleich wichtig. Einige können lebenswichtig sein und andere nur wünschenswert. Eine Priorisierung der Anforderungen hilft vor allem dabei, beurteilen zu können, ob die Entwicklung dem Kundenwunsch entspricht und auf welche Bereiche der Entwicklungsarbeit die meiste Energie konzentriert werden muss.

Verifizierbarkeit: Damit eine Anforderungsspezifikation verifizierbar ist, muss jede in ihr enthaltene Anforderung verifizierbar sein. Das ist nur dann der Fall, wenn es eine endliche, vom Kostenaufwand her vertretbare Möglichkeit gibt, mit der eine Person oder eine technische Einrichtung überprüfen kann, ob ein entwickeltes System der Anforderung gerecht wird [IEE98b]. Eine Anforderung, in der relativierende Ausdrücke (z.B. gut, schnell, normalerweise) enthalten sind, die nicht weiter spezifiziert werden, ist generell nicht verifizierbar.

Modifizierbarkeit: Anforderungen sind nur dann modifizierbar, wenn es ihre Struktur erlaubt, Änderungen einfach, vollständig und konsistent durchzuführen. Ein klare Struktur mit abgegrenzten Einzelanforderungen bildet die Grundlage für eine modifizierbare Spezifikation. Weiterhin muss darauf geachtet werden, dass keine *Redundanz* zwischen den Anforderungen herrscht.

Redundanz an sich ist noch kein Fehler, wenn die Konsistenz gewährleistet ist. Gerade in der Erhaltung der Konsistenz bei kleinen Änderungen liegt allerdings die Schwäche redundanter Spezifikationen. Jede Änderung muss dann nämlich potentiell an mehreren Stellen gemacht werden.

Zuordenbarkeit: Um Zuordenbarkeit zu gewährleisten ist es nötig, dass jede Anforderung eindeutig referenziert werden kann und Anforderungen, auf denen sie basiert, eindeutig referenziert. Erweitert eine Anforderung eine andere oder wird von einer anderen Anforderung erweitert, muss dies also durch den Betrachter des Anforderungsdokuments nachvollziehbar sein.

4.1.1 *Anforderungsanalyse*

Um Anforderungen an ein neu zu entwickelndes Produkt zu erheben und zu analysieren, müssen verschiedene Aufgaben durchgeführt werden. Zuerst werden alle Projektbeteiligten identifiziert.

> **Definition 4.3:** ***Anforderungsanalyse*** **-** „(1) The process of studying user needs to arrive at a definition of system, hardware, or software requirements.
> (2) The process of studying and refining system, hardware, or software requirements.“ [IEE90]

Dazu zählen unmittelbar Auftraggeber, Auftragnehmer, Entwickler und Tester. Weiterhin können potentielle Kunden, Projektgegner, politische Organisationen, Standardisierungs- und Kontrollinstanzen, Regulierungsbehörden sowie kapitalgebende Körperschaften und Banken zu den direkt oder indirekt am Projekt Beteiligten gehören. Die Anforderungen an das Produkt werden gemeinsam mit Hinblick auf Funktionsumfang und Kosten aufgestellt. Als Grundlage dafür dienen Machbarkeitsstudien, Use-Cases, die Definition messbarer Ziele, Prototypen und Spezifikationen.

Die entstandenen Anforderungen müssen während der Anforderungsanalyse mit Hinblick auf die in 4.1 aufgestellten Charakteristiken einer guten Anforderungsspezifikation begutachtet werden. Schwierigkeiten bei der Analyse entstehen vor allem durch den Faktor Mensch:

- Probleme bei der Kommunikation untereinander (unterschiedliches Vokabular, unklare eigene Vorstellungen)
- Unzureichendes technisches Verständnis
- Kosten- und Aufwandsabschätzung kann im Nachhinein zur Veränderung bestehender Anforderungen führen

- Unfähigkeit einzelner Teilnehmer sich in Reviewsitzungen einzubringen
- Entwickler versuchen die Anforderungen an bereits bestehende Projekte anzupassen
- Entwickler ohne Kundenbezug analysieren die Anforderungen

Im Rahmen dieser Arbeit spielt die Begutachtung funktionaler Anforderungen mit Hinblick auf Konsistenz, Vollständigkeit und Redundanz eine wesentliche Rolle.

4.2 FORMULIEREN VON ANFORDERUNGEN

Mit Anforderungen werden Bedingungen aufgestellt, die von einem System eingehalten werden müssen. Dabei ist die Form, in der die Anforderungen für dem Systementwurf formuliert sind, variabel und nicht festgelegt. Im Folgenden werden die für diese Arbeit relevanten Vertreter der Anforderungsdefinition näher erläutert:

Textform: Anforderungen durch reinen Fließtext zu beschreiben, birgt das Risiko der missverständlichen Formulierung und der mehrdeutigen Interpretation. Nicht-funktionale Anforderungen müssen aber meist als Text formuliert werden, da eine Formalisierung oft schwierig ist. Dabei besteht die Herausforderung vor allem in der eindeutigen und konsistenten Formulierung der Anforderungen. Ein ausgiebiger Reviewprozess und definierte Formulierungsvorgaben helfen, die Qualität der Anforderungen sicher zu stellen.

Automaten: Ein Automat stellt eine eindeutige Beschreibung einer Funktionalität dar, deren Verhalten durch das Auftreten diskreter Ereignisse und das Verweilen in definierten Zuständen charakterisiert werden kann. Automaten können das Verhalten entweder vollständig, d.h., für jede mögliche Eingangskombination sind die Ausgänge und die Transition in jedem Zustand definiert, oder unvollständig beschreiben. Eine sinnvolle Beschreibung von Anforderungen setzt meist ein deterministisches, also vorhersagbares Verhalten des Automaten voraus.

Tabellen: Anforderungen in Form einer Tabelle aufzustellen oder durch Tabellen zu ergänzen, kann hilfreich sein, um die Konsistenz der Spezifikation zu wahren. Dazu dürfen die in der Tabelle enthaltenen Informationen nicht zusätzlich in anderer Form (z.B. im Fließtext) vorliegen. Die Konsistenzerhaltung würde sonst erschwert. Weiterhin muss die Syntax der Tabelle im Anforderungsdokument festgehalten werden.

Aktivitätsdiagramme: Durch die Verwendung von Aktivitätsdiagrammen kann der schematische Verlauf modelliert werden. Sie stellen eine Abstraktionsebene dar, in der die eigentliche Funktionalität in den einzelnen Aktionen enthalten ist. Ein Aktivitätsdiagramm ist also immer nur ergänzend zu einer

weiteren Art der Anforderungsformulierung. In diesem Fall muss der Tokenfluss durch Eigenschaften modelliert werden. Je nach aktiver Aktion muss dann der jeweilige Eigenschaftssatz, der hinter der Aktion liegt, wahr sein.

Grundlage dieser Arbeit bilden u.a. Anforderungen, die nach dem 4-Variablen-Modell aufgestellt wurden.

4.2.1 *4-Variablen-Modell*

Das 4-Variablen-Modell [PM95] wurde von Parnas et al. definiert, um die Formulierung von Anforderungen für Echtzeitanwendungen zu unterstützen. Ihm liegt die Teilung der Anforderungen in ein Modell für das sichtbare Verhalten und die möglichen Umgebungsbedingungen auf der einen Seite und die Beschreibung des eigentlichen Softwaresystems auf der anderen Seite zu Grunde.

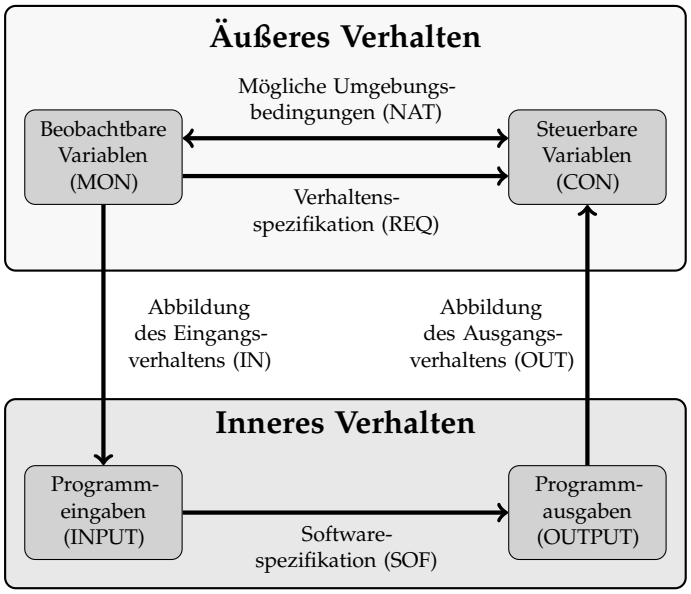


Abbildung 9: 4-Variablen-Modell [PM95]

Abb. 9 zeigt die Struktur des 4-Variablen-Modells, das im Folgenden genauer beschrieben wird:

Beobachtbare Variablen (MON): Ein Wert, der vom System überwacht wird, ist eine beobachtbare Variable. Sie beschreibt ausschließlich die relevante Umgebung (z.B. Luftdruck, Geschwindigkeit, Terminaleingaben) und sagt noch nichts darüber aus, wie der Wert ermittelt wird und in welchem Format er im System verarbeitet wird. Im Folgenden werden die beobachtbaren Variablen mit $m_1, m_2, \ldots m_n$ gekennzeichnet bzw. ihr Wert zum Zeitpunkt t mit $m_1^t, m_2^t, \ldots m_n^t$.

Steuerbare Variablen (CON**):** Ein Wert, der durch das System kontrolliert wird, ist eine steuerbare Variable. Eine steuerbare Variable zeigt sich einem Beobachter durch das mögliche Systemverhalten (Steuerung einer Lampe, Veränderung des Anstellwinkels, Öffnen einer Tür). Im Folgenden werden die steuerbaren Variablen mit $c_1, c_2, \ldots c_n$ gekennzeichnet bzw. ihr Wert zum Zeitpunkt t mit $c_1^t, c_2^t, \ldots c_n^t$.

Mögliche Umgebungsbedingungen (NAT**):** Die möglichen Umgebungsbedingungen beschreiben vor allem physikalische Grenzen. Sie wirken sich gleichermaßen auf beobachtbare wie auf steuerbare Variablen aus. Auf der beobachtbaren Seite kann z.B. der maximal mögliche Luftdruck, das Maximalgewicht oder die minimale Startgeschwindigkeit stehen, während auf der steuerbaren Seite die maximale Helligkeit einer Lampe, die maximale Beschleunigung oder die höchste Zuladung definiert sein könnte. Die Domäne $domain(NAT)$ von NAT beschreibt alle Instanzen von m^t, die auf Grund der Umgebungsbedingungen möglich sind, während der Bildbereich $range(NAT)$ von NAT alle durch die Umgebungsbedingungen möglichen Instanzen von c^t definiert. $(m^t, c^t) \in NAT$ gilt also nur dann, wenn die Umgebungsbedingungen den Wert für c^t erlauben und der beobachtbare Wert m^t definiert ist. Wäre z.B. der physikalisch maximal mögliche Anstellwinkel eines Flugzeuges bei $10°$, dann wäre ein Winkel von $11°$ nicht mehr in $range(NAT)$.

Verhaltensspezifikation (REQ**):** Die Verhaltensspezifikation beschreibt, wie sich die steuerbaren Variablen auf Grund der beobachtbaren Variablen zu verhalten haben. Da sich die Umgebung nicht durch eine Verhaltensspezifikation einschränken lässt, ist die Domäne $domain(REQ)$ von REQ äquivalent oder größer als die Domäne von NAT. Der Bildbereich $range(REQ)$ von REQ beschreibt aber im Unterschied zu NAT nicht mögliches Verhalten, sondern erlaubtes Verhalten. $(m^t, c^t) \in REQ$ gilt also nur dann, wenn die Werte c^t durch ein korrekt arbeitendes System aus den Werten m^t folgen dürfen. Da das Verhalten für alle möglichen Fälle beschrieben werden muss, muss $domain(REQ) \supseteq domain(NAT)$ gelten. Kann sich der Luftdruck z.B. im Bereich zwischen 1 und 10 bar bewegen, so muss in $domain(REQ)$ mindestens das Verhalten für diesen Bereich beschrieben sein. Es darf allerdings auch das Verhalten für 11 bar spezifiziert sein.

Programmeingaben (INPUT**):** Programmeingaben finden über Register statt. Es handelt sich also um den quantisierten Eingangswert und dessen Einheit, der von der Software verarbeitet wird. Im Folgenden werden die Register für die Programmeingaben mit $i_1, i_2, \ldots i_n$ gekennzeichnet bzw. ihr Wert zum Zeitpunkt t mit $i_1^t, i_2^t, \ldots i_n^t$.

Programmausgaben (OUTPUT**):** Programmausgaben finden über Register statt. Es handelt sich also um den quantisierten Ausgangswert und dessen Einheit, der von der Software ausgegeben wird. Im Folgenden werden die

Register für die Programmausgaben mit $o_1, o_2, \ldots o_n$ gekennzeichnet bzw. ihr Wert zum Zeitpunkt t mit $o_1^t, o_2^t, \ldots o_n^t$.

Abbildung des Eingangsverhaltens (IN**):** Beobachtbare Variablen, die vom System verarbeitet werden sollen, müssen eine Entsprechung in den Programmeingaben haben. Die Domäne von IN $domain(IN)$ besteht also aus allen möglichen Instanzen von m^t, während der Bildbereich $range(IN)$ aus allen möglichen Registerwerten i^t besteht. $(m^t, i^t) \in IN$ gilt also nur dann, wenn i^t durch m^t mögliche Programmeingaben beschreibt. Eine mögliche Abbildung des Eingangsverhaltens wäre z.B. die A/D-Wandlung eines Sensorwertes.

Abbildung des Ausgangsverhaltens (OUT**):** Steuerbare Variablen, die vom System gesetzt werden sollen, müssen eine Entsprechung in den Programmausgaben haben. Die Domäne von OUT $domain(OUT)$ besteht also aus allen möglichen Instanzen von o^t, während der Bildbereich $range(OUT)$ aus allen möglichen beobachtbaren Variablen c^t besteht. $(o^t, c^t) \in OUT$ gilt also nur dann, wenn für alle Werte von o^t eine beobachtbare Variable c^t gesetzt werden kann. Eine mögliche Abbildung des Ausgangsverhaltens wäre z.B. die D/A-Wandlung eines Ausgangswertes und die entsprechende Reaktion eines Aktors.

Softwarespezifikation (SOF**):** Die Softwarespezifikation beschreibt das Verhalten der Programmausgaben in Abhängigkeit von den Programmeingaben. Die Domäne von SOF $domain(SOF)$ besteht aus allen möglichen Instanzen von i^t, während der Bildbereich $range(SOF)$ aus allen möglichen Instanzen von o^t besteht. $(i^t, o^t) \in SOF$ gilt also nur dann, wenn die Programmausgaben o^t durch die Programmeingaben i^t durch eine korrekt arbeitende Software erlaubt sind.

Das 4-Variablen-Modell wurde ursprünglich aus der SCR entwickelt. Diese stellt eine Notation für die Beschreibung ereignisbasierter Systeme zur Verfügung, die im Folgenden beschrieben wird.

4.2.2 *Software Cost Reduction*

SCR wurde vom *Naval Research Laboratory* entwickelt, um Echtzeitanforderungen für eingebettete Systeme zu spezifizieren [Maro2, Hen80]. Dabei liegt der Fokus auf der Beschreibung des von außen sichtbaren Verhaltens des Systems. Die Ziele, die eine Spezifikation nach SCR erreichen muss, sind:

1. Es darf nur das Verhalten beschrieben werden. Die eigentliche Implementierung wird dadurch nicht festgelegt, um die Entwicklungsmöglichkeiten nicht unnötig einzuschränken.
2. Das Anforderungsdokument muss einfach zu verändern sein.

3. Das Anforderungsdokument muss einfach durchsuchbar und nachvollziehbar sein.
4. Zukünftig geplante Änderungen und Erweiterungen der Anforderungen müssen ebenso einfach ersichtlich sein, wie die grundlegenden und unveränderlichen Voraussetzungen, die in jeder Version des Anforderungsdokuments gelten müssen.
5. Die Reaktion des Systems auf Hardwareausfälle muss im Dokument beschrieben sein.
6. Vorgaben an die Entwicklung (Interface zu anderen Systemen, Timing-Vorgaben oder Programmiersprache) müssen spezifiziert werden.

Für den Aufbau der Spezifikation gilt daher:

1. Die Themengebiete müssen klar aufgeteilt sein. Es muss eindeutig ersichtlich sein, ob und welche Teilabschnitte abhängig bzw. unabhängig voneinander sind.
2. Die Spezifikation sollte so formal wie möglich gestaltet werden. Fließtext zum Beschreiben von Anforderungen ist zu vermeiden.
3. Jede Anforderung sollte nur einmal vorkommen. Erklärungen zu Anforderungen müssen als solche erkennbar sein.

SCR wird vor allem für die Spezifikation von sicherheitskritischen Anwendungen eingesetzt. Im Fokus liegen Anwendungen, die zur Steuerung von Sicherheitssystemen für industrielle Anlagen, wie Kraftwerke oder Chemiewerke, verwendet werden und die Beschreibung von Systemen aus der Luft- und Raumfahrt [AFB+88, BH00]. Gerade mit Hinblick auf die Gefahren, die von einem Fehlverhalten der Kontroll- und Steuerungssysteme in diesen Bereichen ausgehen, muss die zu Grunde liegende Anforderungsbeschreibung eindeutig und unmissverständlich formuliert sein. Im Folgenden wird genauer auf den durch die SCR vorgeschriebenen Notationsstil eingegangen.

SCR verwendet für die Beschreibung des Systemverhaltens boolesche Werte und Prädikate sowie die logische Verknüpfung beider, die zu einem Zustandswechsel, bzw. zum Setzen eines Ausgangswertes führen. Sie stellt also eine spezielle Form eines Zustandsautomaten dar. Folglich muss ein System, das mit SCR modelliert werden soll, ereignisbasiert sein. Ein auftretendes Ereignis wird mit *@T(Bedingung)* bzw. mit *@F(Bedingung)* gekennzeichnet, wobei die auftretenden Ereignisse im System sequentiell abgearbeitet werden, sodass pro Zeitschritt immer nur ein Ereignis auftreten kann. Ein Ereignis tritt dann auf, wenn die Bedingung wahr (*@T(Bedingung)*) bzw. falsch (*@F(Bedingung)*) wird. Jedes Ereignis kann durch *WHEN(Bedingung)* erweitert werden. *@T(a = 0) WHEN (b > 5)* wird z.B. nur dann wahr, wenn a seinen Wert auf 0 ändert und b gleichzeitig größer 5 ist.

Das Zeichnen des Zustandsautomaten ist bedingt durch die hohe Komplexität des entstehenden Automaten nicht mehr praktikabel. Ein solcher Automat wäre weder nachvollziehbar, noch änderbar. Für die Darstellung des Verhaltens werden daher fünf Arten von Tabellen verwendet. In Abb. 10 ist die Notation

Initial Mode Table

Mode	Defining Conditions
Mode1	a < 0
Mode2	a = 0
Mode3	a > 0

Mode Transition Table

Current Mode	a ↗ 5	b ↗ 0	c ↗ 1	New Mode
Mode1	f	@T	-	Mode2
	-	-	@F	Mode3
Mode2	@T	t	-	Mode1
Mode3	f	t	@F	Mode1
	@F	-	-	

Abbildung 10: SCR-Zustandsübergänge

für die Tabellen zur Beschreibung der Zustandsübergänge (*Mode Transition Table*) und des Anfangszustands (*Initial Mode Table*) exemplarisch dargestellt. Ein Zustand wird in SCR als *Mode* bzw. ein Zustandsautomat als *Mode Class* bezeichnet. Eine Spezifikation kann auch mehrere Zustandsautomaten definieren, die parallel laufen. In der linken Spalte der *Initial Mode Table* wird der Zustand vermerkt, der zu Beginn der Ausführung gilt, wenn die Bedingung auf der rechten Spalte zu Beginn gilt.

Die mittleren Spalten der *Mode Transition Table* enthalten die Bedingungen, die gelten müssen, um einen Zustandsübergang auszulösen. *'@F'* und *'@T'* kennzeichnen das Ereignis, das für den Zustandswechsel verantwortlich ist, *'f'* und *'t'* kennzeichnen die Bedingungen, die beim Auftreten des Ereignisses gelten müssen und '-' beschreibt, dass die jeweilige Bedingung für den Zustandsübergang nicht ausgewertet werden muss. Die linke Spalte enthält den aktuellen Zustand, während in der rechten Spalte der jeweilige Folgezustand angegeben ist. Das Ausgabeverhalten wird durch die in Abb. 11 gezeigten

Condition Table

Mode	Conditions	
Mode1	b != 5	c != 0
Mode2 Mode3	b = 5	c = 0
Light	On	Off

Event Table

Mode	Events	
Mode1	X	@T(b = 5)
Mode2 Mode3	@F(in Mode)	@F(a = 0) WHEN b < 7
Action	Increase z	Decrease z

Selector Table

Mode	Move	Steer
Mode1	Up	X
Mode2 Mode3	Down	Left

Abbildung 11: SCR-Ausgangsverhalten

Tabellen festgelegt. Man unterscheidet hierfür drei Tabellenarten, die *Condition Table*, die *Event Table* und die *Selector Table*. Ein Ausgangswert kann entweder fest durch eine *Condition Table* an einen Eingangswertebereich gebunden werden oder mittels einer *Event Table* an einem bestimmten Ereignis festgemacht werden. Durch die *Selector Table* wird der Ausgangswert alleine durch den aktuellen Zustand definiert.

Durch die SCR-Notation ist ausschließlich die Beschreibung diskreter Systeme möglich. Jede Tabelle dient der Beschreibung des nächsten Systemzustands und dem Ausgangsverhalten abhängig vom jeweiligen *Mode* und den Eingängen. Alle gezeigten Tabellen, bis auf die *Initial Mode Table*, die den Anfangszustand bei Systemstart festlegt, beschreiben einen zeitlichen vorher-nachher Zusammenhang der *Modes* bzw. der Ausgänge. So legt die *Mode Transition Table* z.B. *New Mode* in Abhängigkeit von *Current Mode* und den Eingängen fest. *New Mode* kann ausschließlich dann erreicht werden, wenn ein Eingang seinen Wert ändert und somit ein Ereignis auftritt. Für die Tabellen zur Beschreibung des Ausgangsverhalten gilt analog, dass sich dieses immer nur dann ändern kann, wenn eine Bedingung erreicht wurde (*Condition Table*), ein Ereignis eintritt (*Event Table*) oder ein Zustand erreicht wird (*Selector Table*).

Die CoRE-Methodik ist eine Erweiterung der SCR, die auch auf dem 4-Variablen-Modell basiert. In CoRE wird SCR durch Elemente aus der Objektorientierung erweitert. Für nähere Informationen zur CoRE wird auf [FBWJK92] verwiesen.

5

PRODUKTLINIEN UND VARIABILITÄT

Bei Anforderungen, die Produktlinien beschreiben, muss neben der Formalisierung von Verhalten auch eine Formalisierung der in den Anforderungen beinhalteten Variabilität stattfinden, um eine automatisierte Verarbeitung zu ermöglichen. Diese muss durch ein geeignetes Modell darstellbar sein. Eine *Produktlinie* bezeichnet eine endliche Menge von Produkten, die aus derselben Spezifikation abgeleitet werden können [CN01, PBL05].

Definition 5.1: ***Artefakt*** **-** Ein Artefakt ist ein Bestandteil der Anforderungen, Architektur, der einzelnen Komponenten oder der Testumgebung eines Systems

Definition 5.2: ***Variabilität*** **-** Durch Variabilität wird beschrieben, dass einzelne Artefakte nicht in allen Konfigurationen vorkommen müssen.

Definition 5.3: ***Variationspunkt*** **-** Ein Variationspunkt legt Art und Ort der Variabilität fest.

Definition 5.4: ***Variante*** **-** Durch Varianten werden alle möglichen Belegungen eines Variationspunkts festgelegt.

Definition 5.5: ***Feature*** **-** Ein Feature beschreibt einen charakteristischen Bestandteil eines Systems.

Definition 5.6: ***Konfiguration*** **-** Die Konfiguration eines Produkts beschreibt eine gültige Auswahl von Features mit dem Ziel das Produkt erstellen zu können.

Definition 5.7: ***Produkt*** **-** Ein Produkt ist die Implementierung einer gültigen Konfiguration.

Definition 5.8: *Produktlinie* - Eine Produktlinie ist eine Menge von Systemen, die eine gemeinsame Menge von Features teilen, um bestimmte sich ähnelnde Marktbedürfnisse zu befriedigen.

Die zu Grunde liegende Spezifikation darf also nicht starr ein einziges Produkt definieren, sondern muss dem Designer, Kunden oder Anwender eine Auswahl anbieten. Diese Auswahl bezieht sich auf einzelne Bestandteile, den Funktionsumfang oder Zusatzmerkmale des Grundprodukts und wird durch Variationspunkte gekennzeichnet.

Beim Aufstellen der Spezifikation für eine Produktlinie gewinnt das genaue Differenzieren einzelner Features und deren Bezug zueinander an Bedeutung. Jede Produktlinie beschreibt im Normalfall eine Reihe grundlegender Features, die in jeder Konfiguration vorhanden sein müssen. Zusätzlich können optionale Features in einer Produktlinie enthalten sein. Es ist weiterhin möglich, dass sich zwei oder mehr Features gegenseitig ausschließen oder bedingen.

Für die Abhängigkeit oder die Notwendigkeit von Features innerhalb einer Produktlinie gibt es verschiedene Gründe. Meistens sind diese technischer Natur. So kann die Menge optionaler Features durch die maximale Teilnehmerzahl an einem Bus beschränkt werden. Auch der Energiebedarf und die damit verbundene Bemessung der Stromversorgung kann innerhalb einer Produktlinie variieren. Der Funktionsumfang der Software eines Produktes kann frei gewählt werden, verändert aber unter Umständen Anforderungen an die Prozessorleistung, den Speicherplatz oder den Hauptspeicher. Es existieren weiterhin marketingstrategische Erwägungen, die einzelne Features einer Produktlinie als Pflichtbestandteil definieren oder eine Abhängigkeit zwischen mehreren Features herstellen.

Die Definition einer Produktlinie lohnt sich meist erst ab einer größeren Anzahl ableitbarer Produkte, den Konfigurationen. Das initiale Aufstellen einer Produktlinie verursacht Kosten, die sich erst ab einem bestimmten *Break-Even-Point*, amortisieren. Abb. 12 verdeutlicht diesen Zusammenhang. Weiterhin sinkt die *Time-to-Market*, also die Zeit, die bei der Entwicklung nach dem V-Modell (2.1.4) benötigt wird. Diese ist durch den anfänglichen Aufwand bei der Produktlinienentwicklung bei den ersten Produkten höher als bei der Entwicklung einzelner Produkte. Sie fällt dann auf einen konstanten Wert ab, der unter dem der Entwicklung von Einzelprodukten liegt.

Bei der *Produktlinienentwicklung* handelt es sich um einen fortlaufenden Prozess, in den Kundenwünsche und Entwicklungsvorgaben einfließen. Das Aufspalten einer Produktlinie in einzelne voneinander abhängige Features dient der Strukturierung. Gleichzeitig können einmal entwickelte Features in anderen Produktlinien wiederverwendet werden. Dadurch sinkt die Entwicklungszeit zukünftiger Projekte. Die Definition einzelner Features in den Anforderungen,

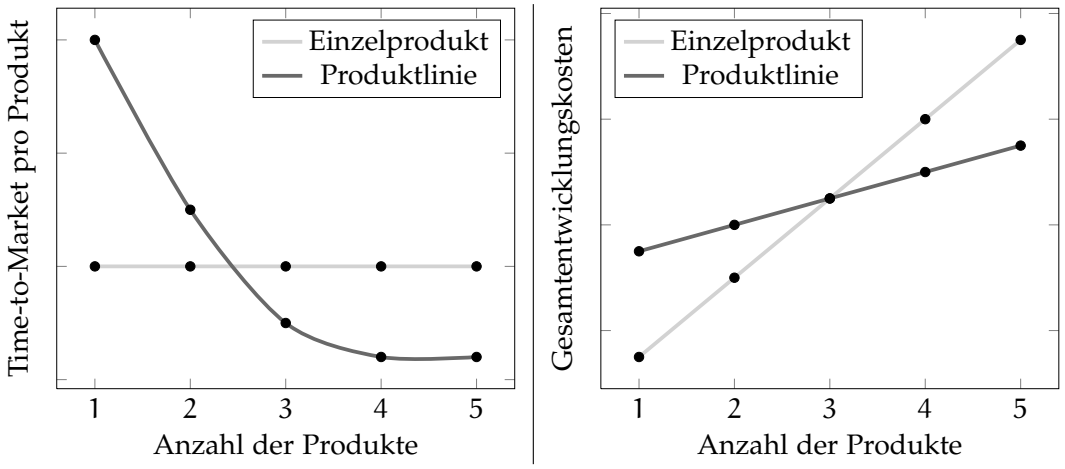


Abbildung 12: Entwicklungskosten und -zeit [PBL05]

die Teil der Produktlinie werden sollen, kann entweder durch den Kunden (*kundengetrieben*) oder durch den Hersteller (*herstellergetrieben*) geschehen [LSR07]. Beim kundengetriebenen Vorgehen stehen die Kundenwünsche im Fokus. Diese sind häufig schwer zu identifizieren und umzusetzen. Demgegenüber steht der herstellergetriebene Prozess. Der Hersteller analysiert den Markt und versucht mit Hinblick auf die Entwicklungs- und Produktionskosten sowie die technische Machbarkeit ein möglichst marktgerechtes Produkt zu entwickeln. In der Praxis wird meist eine Mischung aus kundengetriebenem und herstellergetriebenem Vorgehen gewählt.

5.1 STRUKTUR

Die Produktlinienentwicklung folgt einer vorgegebenen Struktur auf Domänen- (*domain*) und Anwendungsebene (*application*) (Abb. 13). Diese gliedern sich laut [PBL05] jeweils in Anforderungen, Design, Realisierung und Testfälle. Über diesen vier Teilpunkten liegt das *Produktmanagement*, das sich neben der Entwicklung und Produktion vor allem um das Marketing kümmert. Im Rahmen des Produktmanagements werden anhand von Marktanalysen die Funktionalitäten und Features auf Domänen- und Anwendungsebene festgelegt. Weiterhin wird der Zeitplan für die Entwicklung vorgegeben. Das Produktmanagement steht während des gesamten Lebenszyklus der Produktlinie in Verbindung mit den Entwicklungsstufen. Es reagiert auf Vorschläge und Rückmeldung aus dem Entwicklungsprozess und passt die Anforderungen entsprechend an. Das Produktmanagement einer Produktlinie stellt also einen dynamischen Prozess dar.

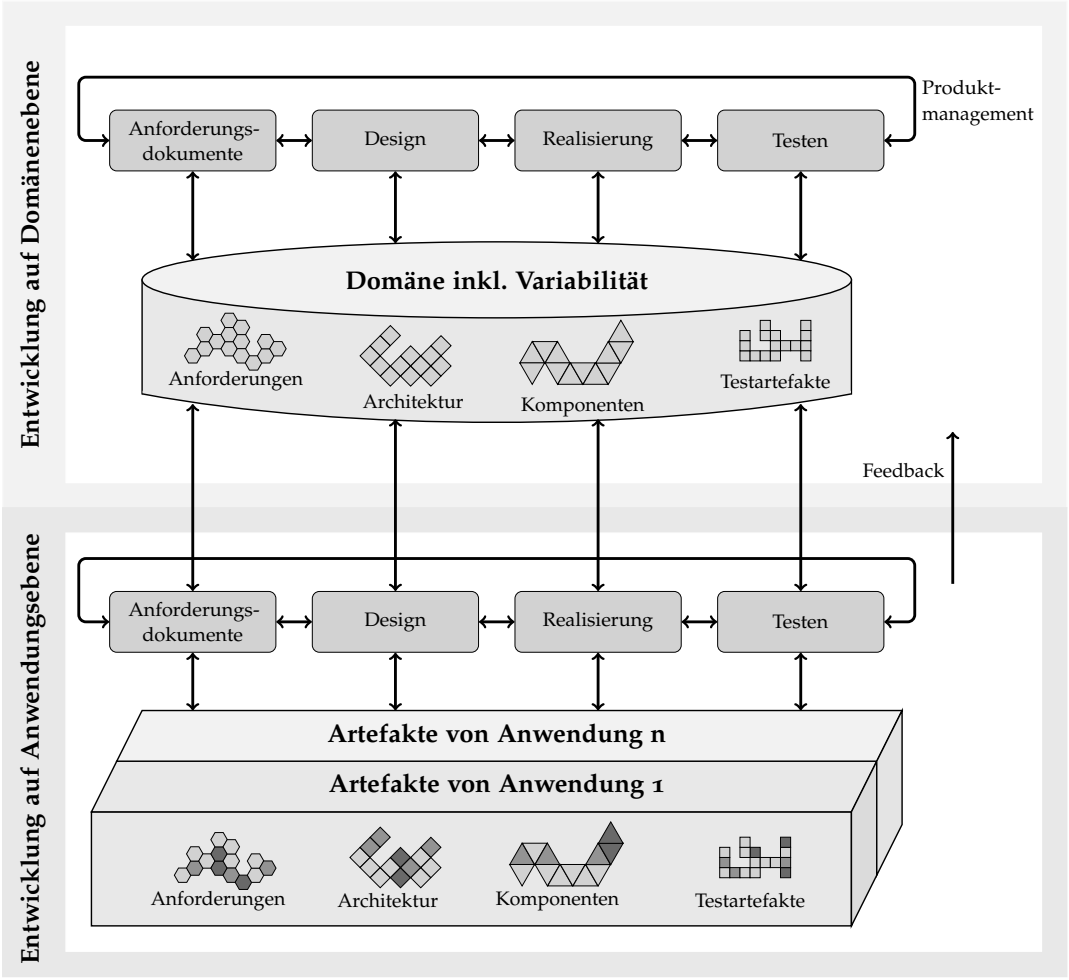


Abbildung 13: Produktmanagement [PBL05]

5.1.1 *Anwendungs- und Domänenebene*

Die *Domänenebene* gibt wiederverwendbare Bestandteile vor und liefert eine globale Beschreibung der gemeinsamen und variablen Bestandteile. Auf ihr wird die Produktlinie modelliert und die einzelnen Features definiert. Weiterhin werden die Features auf Anforderungen, Software- und Hardwarekomponenten sowie Testfälle abgebildet. Auf der Domänenebene können bereits gemeinsame und optionale Bestandteile der Produktlinie entwickelt und getestet werden. Sie dient also gleichsam als Vorlage für die Anwendungsebene.

Auf der *Anwendungsebene* werden die Features aus der Domänenebene wiederverwendet und anwendungsspezifisch erweitert. So entstehen auf der Anwendungsebene die einzelnen Produkte, die aus der Beschreibung innerhalb der Domänenebene abgeleitet wurden. Diese werden aus den für die jeweilige Anwendung benötigten *Artefakten* der Domänenebene erzeugt. Ein Artefakt stellt einen abgeschlossenen Bestandteil des Anforderungsdokuments, des Designs, der Realisierung oder des Tests dar. Es kann äquivalent mit einem Feature

sein oder einen Bestandteil eines Features ausmachen. Artefakte bedürfen unter Umständen Anpassungen, um aus der Domänenebene übernommen zu werden. Die Anwendungsebene stellt zusammengefasst den Lösungsraum für die durch die Domänenebene definierte Produktlinie dar. Variabilität kann dabei beim Kompilieren oder erst während der Programmausführung in das generierte Produkt einfließen. Es existiert nach [PBL05] eine externe, für den Kunden relevante, und eine interne, für den Kunden unsichtbare Variabilität. Nach dem 4-Variablen-Modell (4.2.1) ist Variabilität im äußeren Verhalten äquivalent zur externen Variabilität und Variabilität im inneren Verhalten äquivalent zur internen Variabilität. Beziehungen zwischen interner und externer Variabilität werden nicht explizit, wie im 4-Variablen-Modell, definiert.

Den zentralen Nutzen der Aufteilung von Entwicklungsvorhaben in Domänen- und Anwendungsebene stellt die Möglichkeit der Wiederverwendung von Artefakten aus der Domänenebene in der Anwendungsebene dar. Dies führt zur drastischen Senkung der Entwicklungskosten, da die Artefakte so vom Anforderungsdokument bis hin zum Abnahmetest entweder komplett übernommen oder mit leichten Veränderungen für verschiedene Konfigurationen verwendet werden können. Eine wichtige Voraussetzung für diese Aufteilung stellt die Möglichkeit der Modellierung der gemeinsamen und optionalen Bestandteile einer Produktlinie dar (5.2).

Eine Anpassung des Modells aus Abb. 13 auf das V-Modell wird in [OWES11] vorgeschlagen.

5.1.2 *Entwicklungsschritte*

Im Folgenden soll auf die einzelnen Bestandteile der Domänen- und Anwendungsebene aus Abb. 13 eingegangen werden sowie eine Relation zwischen denselben dargestellt werden:

Anforderungsdokumente: Die Anforderungsdokumente bestehen auf Domänenebene aus gemeinsamen und variablen Features der Produktlinie sowie einem Variabilitätsmodell. Beides bildet die Grundlage für das Design. Das initiale Aufstellen der Anforderungsdokumente wird durch das Produktmanagement durchgeführt. Alle Anforderungen können in beliebiger Form (4.2) vorliegen. Auf Anwendungsebene bilden die Anforderungen ein Dokument, das ein einzelnes Produkt beschreibt und als Vorlage für das Design auf Anwendungsebene dient. Dabei werden Artefakte aus der Domänenebene wiederverwendet bzw. für das jeweilige Produkt angepasst. Weiterhin können durch das Produktmanagement produktspezifische Anforderungen hinzugefügt werden. Während der Lebenszeit der Produktlinie besteht die Möglichkeit, dass die Anforderungen auf Domänenebene durch Vorschläge, die auf Anwendungsebene gemacht werden, erweitert werden.

Design: Während des Designs wird auf Domänenebene eine Referenzarchitektur erstellt, die das System in seine Bestandteile zerlegt und deren Interaktion abstrahiert. Weiterhin werden auf dieser Ebene Regeln definiert, die während der Entwicklung einzuhalten sind. Das Design kann unter verschiedenen Blickwinkeln betrachtet werden. Dazu gehört der Blick auf die logischen Zusammenhänge, auf die Struktur, auf die Prozesse und auf die Abbildung auf Programmcode. Im Designprozess werden Konkretisierungs- und Verbesserungsanfragen bei Unklarheiten in den Anforderungsdokumenten formuliert. Weiterhin werden der Realisierung auf Domänenebene und dem Design auf Anwendungsebene eine Referenzarchitektur und wiederverwendbare Artefakte zur Verfügung gestellt. Das Design auf Anwendungsebene liefert eine Abschätzung über die Machbarkeit der Anforderungen und die Architektur für die Realisierung auf Anwendungsebene. Artefakte, die auf Anwendungsebene dem Design hinzugefügt werden, können auf Domänenebene übernommen werden, wenn sie für weitere, zukünftige Produkte vorgesehen sind.

Realisierung: Auf Domänenebene werden während der Realisierung wiederverwendbare Artefakte implementiert. Dazu gehören Programmcode, Datenbanken und Protokolle. Die Variabilität kann auf dieser Ebene durch Konfigurationsdateien repräsentiert werden. Die Artefakte und Konfigurationsdateien werden für den Test auf Domänenebene sowie für die Realisierung auf Anwendungsebene wiederverwendet. Auf Anwendungsebene wird während der Realisierung die eigentliche zu testende Anwendung fertig gestellt. Während der Fertigstellung werden Fehler, die im Design gemacht wurden, identifiziert und können im Design auf Anwendungsebene korrigiert werden. Um die Anwendung korrekt testen zu können, liefert die Realisierung eine Beschreibung des Umgebungsverhaltens sowie der Ein- und Ausgaben für den Test auf Anwendungsebene. Artefakte der Realisierung auf Anwendungsebene können bei Bedarf in die Domänenebene transferiert werden.

Testen: Das Testen dient auf Domänenebene der Überprüfung der Anforderungen, des Designs und der Realisierung. Die Tests basieren auf den wiederverwendbaren Artefakten der anderen Elemente auf Domänenebene. Weiterhin werden *Testartefakte* für den Test auf Anwendungsebene zur Verfügung gestellt. Testartefakte sind Bestandteile einer Testumgebung, wie Testfälle, ausführbare Modelle oder Klassifikationsbäume. Auf Anwendungsebene dient der Test der Abnahme der Anwendung für den Betrieb. Er orientiert sich am V-Modell (2.1.4) und interagiert mit den Anforderungen, dem Design und der Realisierung auf Anwendungsebene.

Von der Anwendungs- in die Domänenebene gibt es immer ein Feedback. Neue Entwicklungen oder korrigierte Fehler fließen somit in die Produktlinie ein. Dadurch profitieren zukünftige Konfigurationen, aus denen Produkte entstehen sollen. Diese verfügen somit über einen größeren Funktionsumfang und eine höhere Qualität.

5.2 DARSTELLUNG

Darstellungen von Variabilität innerhalb einer Produktlinie müssen intuitiv lesbar sein, um von allen Projektbeteiligten eindeutig verstanden zu werden. So kann Missverständnissen vorgebeugt werden und Änderungen einfach nachvollzogen und hinzugefügt werden. Zu diesem Zweck wurden von Kang et al. [KCH+90] 1990 *Feature-Modelle* vorgestellt. Dabei handelt es sich um einen azyklischen Graphen, dessen Knoten die einzelnen Features eines Produkts repräsentieren [CE00]. Ein Feature kann eine einzelne Funktionalität, eine Gruppe von Funktionalitäten [Bos00] oder eine nicht funktionale Komponente des Produkts [CHE05] repräsentieren. Die Aufspaltung der Spezifikation in einzelne Features ist nicht geregelt, sondern kann variabel angepasst werden.

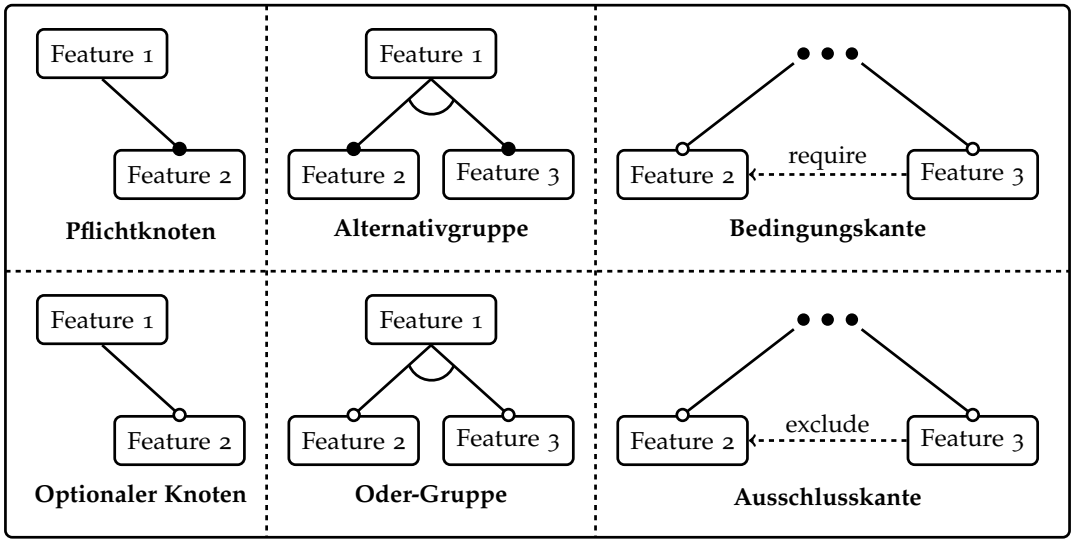


Abbildung 14: Feature-Modell-Notation

So ist es z.B. möglich, ein globaleres Feature-Modell für Marketingzwecke aufzubauen und ein feiner strukturiertes für die Entwicklung.

Die Darstellung eines Feature-Modells folgt den Regeln aus Abb. 14:

- **Pflichtknoten:** Der Knoten muss in jeder Konfiguration der Produktlinie vorhanden sein.
- **Optionaler Knoten:** Der Knoten kann Teil einer Konfiguration der Produktlinie sein.
- **Alternativgruppe:** Es muss immer genau einer der Knoten Teil der Konfiguration sein (1 aus n Bedingung).
- **Oder-Gruppe:** Es darf eine beliebige Anzahl der Knoten Teil des Produkts sein (n aus m Bedingung).

- **Bedingungskante**: Ist ein Feature Teil der Produktlinie, so bedingt dies ein weiteres Feature.
- **Ausschlusskante**: Ist ein Feature Teil der Produktlinie, so schließt dies ein weiteres Feature aus.

Alle aus diesem Modell ableitbaren möglichen Kombinationen ergeben ein gültiges Produkt [HST+08]. Das Feature-Modell stellt also die Struktur der gesamten Produktlinie dar. Diese wird auch als *150%-Modell* bezeichnet. Ein 150%-Modell ist im Normalfall durch die Abhängigkeiten zwischen den einzelnen Features nicht implementierbar und somit auch nicht testbar. Es kann dazu verwendet werden, Produkte durch das Weglassen einzelner Features zu generieren. Diese Methodik bezeichnet man als 150%-Ansatz, dem der kompositionale Ansatz gegenüber steht. Beim kompositionalen Ansatz werden die Features einzeln implementiert und später zu einem Produkt zusammengesetzt. Im Rahmen dieser Arbeit wird der 150%-Ansatz verwendet.

Feature-Modelle können in boolesche Ausdrücke übersetzt werden. In diesem Ausdruck entspricht jedes Feature einem Literal, dessen logische Verknüpfung durch die Knotenart und den Zusammenhang mit den anderen Features bestimmt wird. Der Ausdruck wird nur dann wahr, wenn eine nach der Struktur des Feature-Modells gültige Featurekombination gewählt wurde.

Definition 5.9: ***Gültige Featurekombination***

f sei ein Feature, das als boolesche Variable interpretiert wird, und dessen Auswahl mit $\{1\}$ bzw. dessen Abwesenheit mit $\{0\}$ gekennzeichnet ist.

F sei die Menge aller im Feature-Modell vorkommenden Features $\{f_1, f_2, \ldots, f_n\}$.

F_K sei eine beliebige Featurekombination, also die Belegung aller Features in F mit booleschen Werten.

B_F sei der nach den Regeln in Tab. 1 aus dem Feature-Modell abgeleitete boolesche Ausdruck.

Eine gültige Featurekombination F_K liegt immer dann vor, wenn gilt $B_F(F_K) = 1$

Tab. 1 verdeutlicht die logischen Funktionen. Die Zusammenhänge zwischen Feature-Modellen und boolescher Logik wurden von Czarnecki und Wasowski in [CW07] dargestellt. Sie stellen ein Feature-Modell als booleschen Ausdruck in KNF dar und zeichnen einen *Implikations-Hypergraphen*, der alle aus der

Feature-Modell Komponente	Semantik
Wurzelknoten w	w
f_1 optionaler Folgeknoten von f	$f_1 \rightarrow f$
f_1 verpflichtender Folgeknoten von f	$f_1 \leftrightarrow f$
$f_1, \ldots, f_n$ Alternativgruppe unter f	$(f_1 \vee \ldots \vee f_n \leftrightarrow f) \wedge \bigwedge_{i<j} \neg(f_i \wedge f_j)$
$f_1, \ldots, f_n$ Odergruppe unter f	$f_1 \vee \ldots \vee f_n \leftrightarrow f$
f_1 bedingt f_2	$f_1 \rightarrow f_2$
f_1 schließt f_2 aus	$\neg(f_1 \wedge f_2)$

Tabelle 1: Boolesche Ausdrücke der Feature-Modell-Konstrukte [CW07]

KNF ableitbaren Implikationen enthält. Weiterhin schlagen sie vor, den logischen Ausdruck in Form eines BDDs darzustellen. Als Herausforderung sehen sie die Überführung eines booleschen Ausdrucks in ein Feature-Modell. Sie identifizieren dabei folgende Probleme:

- Viele äquivalente Feature-Modelle sind aus dem selben booleschen Ausdruck ableitbar.
- Alle durch den Implikations-Hypergraphen implizierten Beziehungen zwischen Features darzustellen könnte den Betrachter des entstehenden Feature-Modells überfordern.
- Die logische Struktur ist nicht die einzige strukturierende Voraussetzung für ein sinnvolles Feature-Modell.
- Eine *Brute-Force* Suche nach dem optimalen Feature-Modell für einen booleschen Ausdruck wird nicht skalieren.

Als Lösung für das Problem wird vorgeschlagen, dass ein Algorithmus, der aus booleschen Ausdrücken Feature-Modelle erzeugt, auf jeden Fall den booleschen Ausdruck vollständig und ohne redundante Teildarstellungen im entstehenden Feature-Modell übersetzen muss. Weiterhin muss der Algorithmus deterministisch arbeiten. Er muss für äquivalente logische Ausdrücke auch das selbe Feature-Modell erzeugen. Die Übersetzung von booleschen Ausdrücken in Feature-Modelle ist weiterhin ein ungelöstes Problem. Eine Lösung könnte das Zusammenführen von Feature-Modellen, die Erzeugung von anwenderspezifischen Ansichten auf Feature-Modelle sowie das *Reverse-Engineering* von Feature-Modellen aus bestehendem Code unterstützen.

5.3 ANFORDERUNGEN IM PRODUKTLINIENKONTEXT

Anforderungen, die nicht nur ein einzelnes Produkt, sondern eine Produktlinie beschreiben, stellen Entwickler und Tester vor immer größere Herausforderungen. Dabei stellt die Analyse von Anforderungen für eine Produktlinie sowie die Generierung eines Orakels für den Produktlinientest einen zentralen Punkt dieser Arbeit dar (8.1). Anforderungen für Produktlinien beinhalten stets Variabilitätsinformationen, die durch ein Variabilitätsmodell darstellbar sind. Diese müssen mit den einzelnen Anforderungen verknüpft werden. Zu diesem Zweck wird das Variabilitätsmodell auf die Anforderungen abgebildet. Das genaue Vorgehen ist in Kapitel 7 beschrieben.

5.4 PRODUKTLINIENTESTS

Das Testen einer Produktlinie stellt eine besondere Herausforderung dar. Hierbei soll vermieden werden, dass jedes aus der Produktlinie generierte und produzierte Produkt immer wieder neu, einzeln und vollständig getestet werden muss. Das Testen von Produktlinien anhand von Orakeln bildet einen wesentlichen Bestandteil dieser Arbeit, der in Abb. 15 dargestellt ist. Die vor-

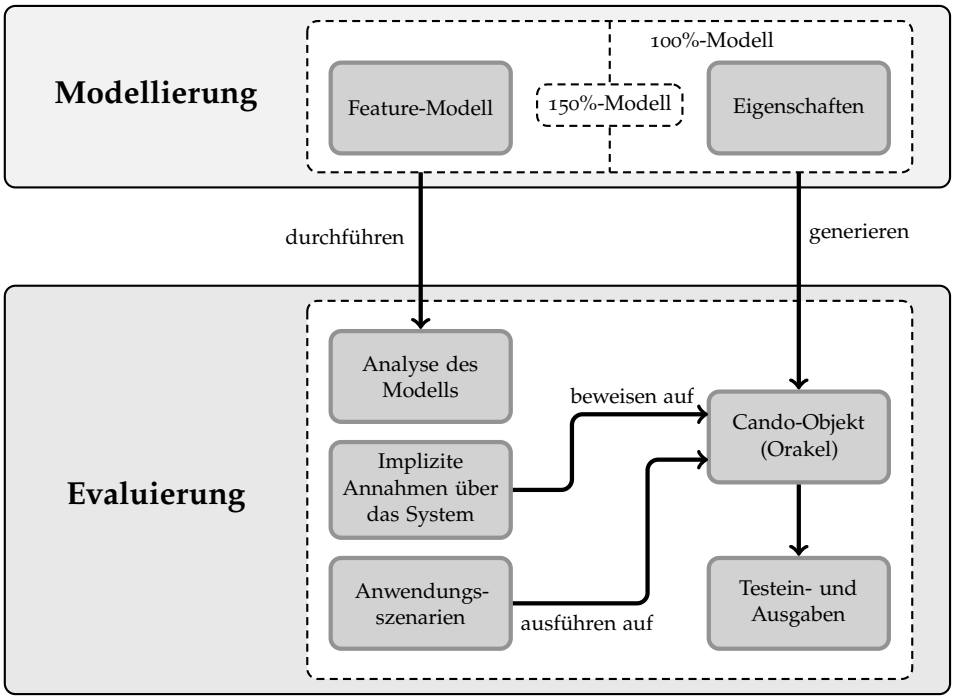


Abbildung 15: Evaluierung

geschlagene Ergänzung des Testprozesses setzt sich aus der Vorhersage von Testergebnissen durch ein Orakel, aus der formalen Überprüfung der Anforderungen und aus dem Beweis von *Systemannahmen* zusammen und wird in Kapitel 8 detailliert behandelt.

Definition 5.10: ***Systemannahme*** **-** Eine Eigenschaft, der ein entwickeltes System genügen muss, um Sicherheit zu garantieren, und die nicht explizit Teil der zu implementierenden Spezifikation ist, ist eine Systemannahme.

Gerade bei Produktlinien sollte das Thema Wiederverwendung einen großen Stellenwert einnehmen. Aus der Forschung lassen sich drei Hauptansätze ableiten:

Keine Wiederverwendung für den Test: Dieser Ansatz widerspricht im Wesentlichen der Idee von Produktlinien. Jedes Produkt wird einzeln getestet. Eine Aufteilung in Anwendungs- und Domänenebene findet nicht statt. Dieser Ansatz wird in [TTK04] als *product-by-product testing* bezeichnet und ist auf Grund der Anzahl ableitbarer Produkte für variantenreiche Systeme heutzutage nicht mehr praktikabel.

Wiederverwendung einzelner Features für den Test: In diesem Ansatz werden Ähnlichkeiten innerhalb der möglichen Konfigurationen ausgenutzt. Es werden Regressionstests angewendet oder Testfällen aus der Domänenebene auf der Anwendungsebene wiederverwendet. Beide Techniken finden vor allem im modellbasierten Testen Anwendung. Einen Überblick mit Hinblick auf Produktlinientests gibt [OWES11]. Diese Techniken reduzieren zwar durch Wiederverwendung den Testaufwand, können aber nicht vermeiden, dass jedes generierte Produkt vollständig, einzeln getestet werden muss.

Wiederverwendung von Feature-Gruppen für den Test: Dieser Ansatz zielt auf die Extrahierung von Untermengen in der Gestalt von Feature-Gruppen oder Produkten und wurde von McGregor [McG01] initiiert. Anstatt jedes Produkt einzeln zu testen, werden mögliche Untermengen generiert und getestet. Cohen et al. [CDS07] entwickelten das *combinatorial interaction testing*. Dieses hat zum Ziel, eine repräsentative Menge von Produkten für die Produktlinie zu finden. Wird diese repräsentative Menge getestet und Fehler korrigiert, so steigt die Qualität der gesamten Produktlinie. Ein Abdeckungsmaß für Produktlinientests auf Basis des *combinatorial interaction testings* wurde außerdem definiert [CDS06].

6

VOLLSTÄNDIGKEIT UND CANDO-OBJEKTE

Sind die in einer Spezifikation enthaltenen Eigenschaften durch die Beschreibung in ITL formalisiert, können sie nach Abb. 15 evaluiert werden. Zum einen wird die Vollständigkeit des Eigenschaftssatzes des 150%-Modells bewertet und zum anderen wird ein Cando-Objekt daraus generiert. Im Folgenden werden beide Schemata genauer erklärt.

6.1 CANDO-OBJEKTE

Der Ansatz der Cando-Objekte entstand im Rahmen der Dissertation von Martin Schickel [Sch09]. Diese zielt darauf ab, Eigenschaftssätze in eine möglichst allgemeine Hardwarebeschreibung zu übersetzen, um die formale Verifikation von Schaltungen zu unterstützen.

6.1.1 *Ursprüngliche Idee von Cando-Objekten*

Ein *Cando-Objekt* ist ein aus einem Eigenschaftssatz erzeugtes, simulierbares Modell. Abb. 16 zeigt seine Struktur. Es verfügt über die selben Ein- und

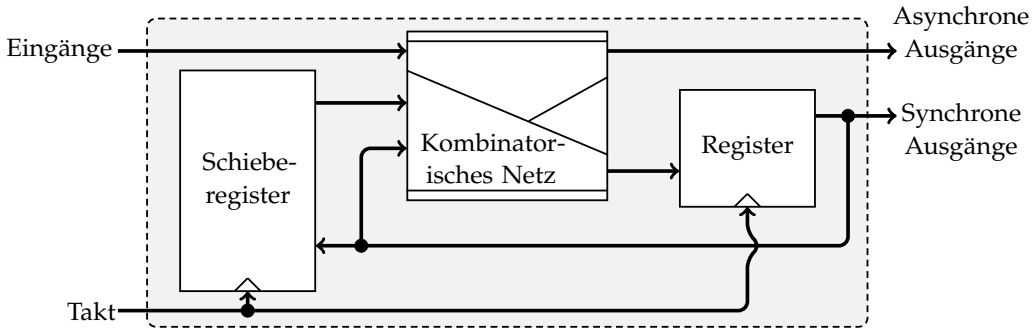


Abbildung 16: Struktur des Cando-Objekts [Sch09]

Ausgänge, die auch eine Black-Box-Repräsentation des System aufwiese. Da Cando-Objekte im Bereich der formalen Hardwareverifikation entwickelt wurden, können die Eingänge asynchron oder synchron zum Systemtakt ihre Werte ändern. Die Funktionalität des Systems wird in einem kombinatorischen Netzwerk abgebildet, das synchrone Ausgänge über einen Registersatz sowie asynchrone Ausgänge ermöglicht. Da sich Eigenschaften über ein Zeitfenster erstrecken, müssen neben den aktuellen Eingangswerten auch vergangene

Eingangswerte und interne Zustände gespeichert werden. Dies geschieht über ein Schieberegister, dessen Werte bei jedem Takt eine Ebene weitergeschoben werden. Der Takt ist das Maß für die Zeit, deren Fortlaufen durch das Auftreten einer steigenden Taktflanke definiert wird. Es handelt sich also um diskrete, durch den Takt bestimmte Zeitpunkte. Ursprünglich wurden Cando-Objekte entwickelt, um Systemeigenschaften zu beweisen.

Definition 6.1: ***Systemeigenschaft*** **-** Eine Eigenschaft, die das Verhalten eines Systems für einen oder mehrere Ausgänge in Abhängigkeit der Eingänge beschreibt.

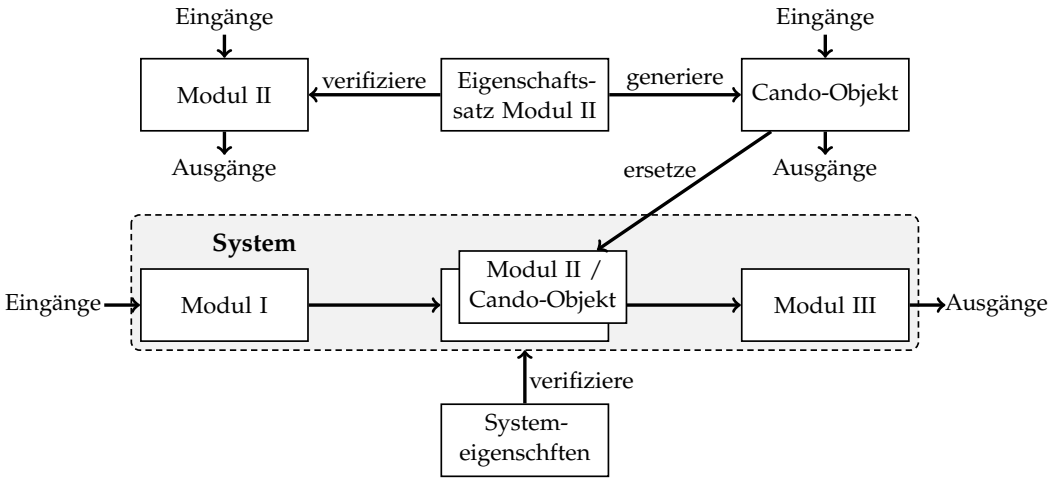


Abbildung 17: Ursprüngliche Verwendung von Cando-Objekten [Sch09]

Abb. 17 zeigt den ursprünglichen Einsatzzweck von Cando-Objekten. In der formalen Hardwareverifikation werden im Normalfall zuerst Eigenschaften für die Module eines Systems geschrieben und auf den Modulen bewiesen. Enthalten die Eigenschaften für ein Modul die komplette relevante Funktionalität und halten sie beim Beweis auf dem Modul, so ist das Modul korrekt und vollständig und kann eingesetzt werden. Werden die Module zu einem System zusammengesetzt, kann der Beweis von Eigenschaften über das System an der großen Komplexität scheitern. Die Idee von Cando-Objekten besteht darin, ein Modul in einem System durch sein Cando-Objekt, das aus den zuvor bewiesenen Eigenschaften erzeugt wird, zu ersetzen und eine Systemeigenschaft auf dem System mit dem eingefügten Cando-Objekt zu beweisen. Die Komplexität dieses Beweises kann niedriger sein und daher kann der Beweis in vertretbarer Zeit ausgeführt werden. Als Folge dessen muss sich ein Cando-Objekt auch in der Simulation genau so verhalten wie es die Eigenschaften beschreiben. Ein weiteres Anwendungsgebiet von Cando-Objekten stellt der *Äquivalenzvergleich* von Eigenschaftssätzen dar. Um die Äquivalenz zweier Eigenschaftssätze zu zeigen, wird aus beiden ein Cando-Objekt erzeugt. Halten beide Eigenschaftssätze auf dem aus dem jeweils anderen Eigenschaftssatz

erzeugten Cando-Objekt, so sind sie äquivalent. Das Werkzeug, das die Generierung der Cando-Objekte ermöglicht (*Candogen*), erzeugt eine den initialen Eigenschaften entsprechende VHDL-Beschreibung. Die initialen Eigenschaften sind in ITL (3.3.2) verfasst. Freiheitsgrade, die durch Spezifikationslücken in Eigenschaftssätzen entstehen (8.3.1), werden in einem Cando-Objekt durch zufälliges Verhalten repräsentiert. Dafür stehen *Random-Inputs* zur Verfügung, die ein zufälliges Verhalten simulieren. Ein Cando-Objekt verhält sich also in den spezifizierten Situationen spezifikationsgetreu und ansonsten zufällig.

6.1.2 *Generierung von Cando-Objekte*

Ein Eigenschaftssatz, der in ITL oder LTL verfasst wurde, eignet sich in der Regel nicht direkt für die Generierung von Cando-Objekten. Grund dafür sind die Freiheitsgrade, die dem Verfasser von Eigenschaften zugestanden werden. So ist es möglich Eigenschaftssätze, die äquivalentes Verhalten beschreiben, auf unterschiedliche Art und Weise zu formulieren. Weiterhin können Widersprüche in dem Eigenschaftssatz enthalten sein, die die Generierung eines ausführbaren Modells unmöglich machen. Ein weiteres Problem bei der Generierung von Cando-Objekten stellt die Einschränkung des Umgebungsverhaltens (*environmental restrictions*) dar. Eine Einschränkung des Umgebungsverhaltens ist immer dann gegeben, wenn ein Zustand des Systems existiert, in dem die Umgebung bestimmte Wertekombinationen nicht annehmen darf, da ansonsten widersprüchliches Verhalten auftritt. Der Eigenschaftssatz muss daher neu strukturiert werden, um ihn für die Übersetzung in ein Cando-Objekt vorzubereiten. Für den detaillierten Ablauf des Algorithmus zur Generierung von Cando-Objekten wird auf [Sch09] verwiesen.

6.2 VOLLSTÄNDIGKEIT VON EIGENSCHAFTSSÄTZEN

Die Überprüfung von Eigenschaftssätzen auf Vollständigkeit ist eine zentrale Voraussetzung für die vollständige formale Verifikation eines Hardwarebausteins. Ein Eigenschaftssatz ist vollständig, wenn der Wert jedes Ausgangs für alle möglichen Eingangskombinationen durch die im Eigenschaftssatz enthaltenen Eigenschaften determiniert ist. Also genau dann, wenn er aus Anforderungen abgeleitet wurde, die nach IEEE 830 (4.1) vollständig sind. Im Rahmen dieser Arbeit wird der Algorithmus zur Vollständigkeitsbewertung aus der Dissertation von Martin Oberkönig [Obe10] verwendet. Dieser Algorithmus ermöglicht es, einen beliebigen Eigenschaftssatz auf Vollständigkeit zu überprüfen, und definiert weiterhin ein Maß für die Vollständigkeit. Es werden außerdem im Eigenschaftssatz enthaltene Widersprüche gefunden.

6.2.1 *Normalform für die Vollständigkeitsbewertung*

Wie schon bei der Erstellung von Cando-Objekten, muss auch für die Vollständigkeitsbewertung der Eigenschaftssatz angepasst werden. Diese Anpassung, auch *Normalisierung* genannt, ist an den Bedürfnissen der formalen Hardwareverifikation ausgerichtet. Für die Normalisierung werden die Ausgänge, die durch den Eigenschaftssatz beschrieben werden, als einzelne Bits betrachtet. Zahlenwerte werden als Bitvektoren repräsentiert und können in ihre einzelnen Bits zerlegt werden. Jedes Bit kann ausschließlich die Werte '0' und '1' annehmen. Die Normalisierung zerlegt die ursprünglichen Eigenschaften derart, dass für jedes Ausgangsbit zwei Implikationen entstehen. Nämlich die Implikation, die alle Eingangskombinationen auf der linken Seite enthält, die den Ausgang v laut Eigenschaftssatz auf '0' setzen, die sogenannte 0-Hülle v_0, und die, die ihn auf '1' setzen, die sogenannte 1-Hülle v_1. Ein Ausgang ist vollständig determiniert, wenn er für jede mögliche Eingangskombination entweder auf '0' oder auf '1' gesetzt wird. Daraus ergibt sich die *Determiniertheitsfunktion*:

$$\mathfrak{D}(v) = v_0 \vee v_1$$

Die Funktion über *Spezifikationslücken* beschreibt alle Eingangskombinationen, in denen ein Ausgang weder auf '0', noch auf '1' gesetzt wird. Sie ergibt sich aus der Negation der Determiniertheitsfunktion:

$$\mathfrak{L}(v) = \neg\mathfrak{D}(v) = \neg(v_0 \vee v_1)$$

Damit ein Eigenschaftssatz konsistent ist, darf er einen Ausgang für jede mögliche Eingangskombination nur entweder auf '0' oder auf '1' setzen. Daraus ergibt sich die *Konsistenzfunktion*:

$$\mathfrak{K}(v) = v_0 \wedge v_1$$

Ein Ausgang ist *vollständig determiniert*, wenn gilt:

$$\mathfrak{D}(v) \equiv 1 \text{ bzw. } \mathfrak{L}(v) \equiv 0$$

Ein Ausgang ist konsistent und damit *widerspruchsfrei*, wenn gilt:

$$\mathfrak{K}(v) \equiv 0$$

Aussagen über den Grad der Vollständigkeit, den sogenannten *Determiniertheitsgrad*, der angibt, für welchen prozentualen Anteil aller Eingangskombinationen der Ausgang definiert ist, erhält man durch:

$\mathfrak{V}(v) = \frac{\#\mathrm{Minterme}}{2^n}$
$\#\mathrm{Minterme}$: Anzahl der durch die Eigenschaften determinierten Situationen
n : Anzahl der Literale

Aussagen über die *Determiniertheit des Gesamtsystems*, das mehrere Ausgangssignale aufweisen kann, erhält man durch:

$$\mathfrak{G} = \frac{\sum^{i} \mathfrak{V}(v_i)}{\#Signale}$$

Freiheitsgrade, die bei der Entwicklung eines Systems dem Entwickler ausdrücklich zugestanden werden, müssen bei der Berechnung der Determiniertheit berücksichtigt werden. Sie werden als Klauselmenge von *Dont-Care*-Werten v_{dc} dem Ausdruck hinzugefügt und es ergibt sich die erweiterte Determiniertheit:

$$\mathfrak{D}_{\mathfrak{F}}(v) = v_0 \vee v_1 \vee v_{dc}$$

Es ergibt sich folgerichtig die erweiterte Spezifikationslücke:

$$\mathfrak{L}_{\mathfrak{F}}(v) = \neg\mathfrak{D}(v) = \neg(v_0 \vee v_1 \vee v_{dc})$$

Der Ausgang einer Spezifikation mit Freiheitsgraden ist also vollständig determiniert, wenn gilt:

$$\mathfrak{D}_{\mathfrak{F}}(v) \equiv 1 \text{ bzw. } \mathfrak{L}_{\mathfrak{F}}(v) \equiv 0$$

Weiterhin ergibt sich, analog zur Berechnung ohne Freiheitsgrade:

$$\mathfrak{K}_{\mathfrak{F}}(v) = v_0 \wedge v_1 \wedge v_{dc} \;\Big|\; \mathfrak{V}_{\mathfrak{F}}(v) = \frac{\#Minterme}{2^n} \;\Big|\; \mathfrak{G}_{\mathfrak{F}} = \frac{\sum^{i} \mathfrak{V}_{\mathfrak{F}}(v_i)}{\#Signale}$$

6.2.2 *Beispiel*

Die Funktionsweise des Algorithmus soll nun beispielhaft mittels folgender LTL-Ausdrücke erläutert werden:

$G(a \wedge b \rightarrow X(x \wedge y))$
$G(\neg a \vee b \rightarrow X(\neg x))$
$G(a \wedge \neg b \rightarrow X(x \wedge \neg y))$

Die Zerlegung in 0- und 1-Hülle ergibt:

1-Hülle von x: $G((a \wedge b) \vee (a \wedge \neg b) \rightarrow X(x))$
0-Hülle von x: $G(\neg a \vee b \rightarrow X(\neg x))$
1-Hülle von y: $G(a \wedge b \rightarrow X(y))$
0-Hülle von y: $G(a \wedge \neg b \rightarrow X(\neg y))$

Durch die Eigenschaften ergibt sich die Funktionstabelle Tab. 2. Für $a \vee b$ muss der Ausgang x laut Eigenschaftssatz sowohl den Wert '0' als auch den Wert '1' annehmen. Somit besteht ein Widerspruch für x im Eigenschaftssatz und es ergibt sich:

$$\mathfrak{K}(x) = ((a \wedge b) \vee (a \wedge \neg b)) \wedge (\neg a \vee b) = a \wedge b \neq 0$$
$$\mathfrak{K}(y) = (a \wedge b) \wedge (\neg a \wedge b) = 0$$

Für die Determiniertheit ergibt sich:

a	b	$X(x \equiv 0)$	$X(x \equiv 1)$	$X(y \equiv 0)$	$X(y \equiv 1)$
0	**0**		✓		
0	**1**		✓		✓
1	**0**	✓			
1	**1**	✓	✓	✓	

Tabelle 2: Funktionstabelle

$\mathfrak{D}(x) = ((a \wedge b) \vee (a \wedge \neg b)) \vee (\neg a \vee b) = 1$
$\mathfrak{D}(y) = (a \wedge b) \vee (\neg a \wedge b) \neq 1$

Für x ist das Verhalten also in jeder Situation determiniert, während für y das nicht der Fall ist. Die Lücken von y sind:

$\mathfrak{L}(y) = \neg((a \wedge b) \vee (a \wedge \neg b) \neq 1) = \neg b$

Da y in zwei von vier möglichen Situationen determiniert ist, ergibt sich $\mathfrak{V}(y) = 50\%$. Die Determiniertheit des Gesamtsystems ist dann:

$\mathfrak{G} = \frac{100\%+50\%}{2} = 75\%$

Handelte es sich bei $\neg b$ um einen Freiheitsgrad für y, so ergäben sich folgende Gleichungen:

$\mathfrak{D}_{\mathfrak{F}}(y) = (a \wedge b) \vee (\neg a \wedge b) \vee \neg b = 1$
$\mathfrak{V}_{\mathfrak{F}}(y) = 100\%$
$\mathfrak{G}_{\mathfrak{F}} = \frac{100\%+100\%}{2} = 100\%$

Das System wäre also vollständig determiniert. Trotzdem muss die widersprüchliche Situation für x aufgelöst werden, um seine Konsistenz zu gewährleisten.

Teil III

KONZEPT UND ERGEBNISSE

In diesem Teil werden die Konzepte präsentiert, die die Anwendung formaler Methoden für Produktlinien eingebetteter Systeme ermöglichen. Dazu gehört die Generierung von Verhaltensmodellen für Produktlinien eingebetteter Systeme, den sogenannten Orakeln, aus formalen Eigenschaften. Weiterhin wird die Vollständigkeit der formalen Eigenschaften bewertet und der Begriff der Vollständigkeit erörtert. Als Grundlage dafür müssen die formalen Eigenschaften um die durch die Produktlinie beschriebene Variabilität erweitert werden. Weiterhin werden die Vorteile der Einführung von Produktlinienmethodiken in die Entwicklung eingebetteter Systeme diskutiert. Schließlich werden die Experimente vorgestellt, die im Rahmen dieser Arbeit durchgeführt wurden. Sie dienen der Plausibilitätsprüfung der vorgestellten Ansätze und sollen Beispiele für Anwendungsgebiete geben.

7

FORMALISIERUNG VON ANFORDERUNGEN

Einen fundamentalen Gesichtspunkt dieser Arbeit bildet die Abbildung des äußeren Verhaltens (4.2.1) in formale Eigenschaften, um daraus ein Orakel (2.3.2) zu generieren und um die Eigenschaften auf Vollständigkeit, Widerspruchsfreiheit und Redundanz zu überprüfen. Abb. 18 zeigt den Zusammenhang zwischen der Spezifikation und der Modellierung. Handelt es sich eine Spezifikation für eine Produktlinie, so müssen Variabilität und Funktionalität aus den Anforderungen extrahiert werden und bilden das 150%-Modell (5.2). Bei Spezifikationen für einzelne Produkte genügt die Extraktion der Funktionalität und es entsteht das *100%-Modell*.

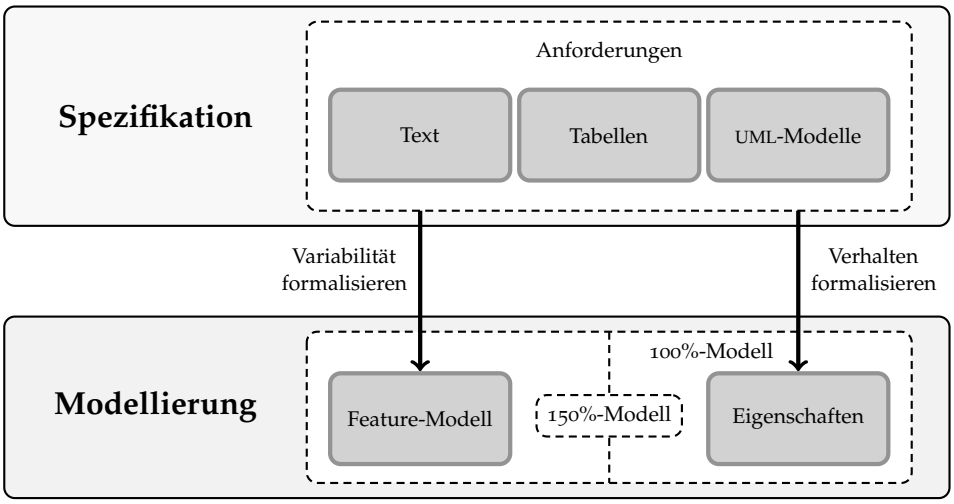


Abbildung 18: Modellierung

7.1 FORMALISIEREN VON VERHALTEN

Ein Eigenschaftssatz beschreibt das Verhalten eines Teils oder der gesamten in den Anforderungen enthaltenen Funktionalität auf eine formale Art und Weise. Im Rahmen dieser Arbeit wird die Beschreibung von Eigenschaften mit ITL (3.3.2) oder LTL (3.3.1) vorgeschlagen. Beide Möglichkeiten bieten eine eindeutige Verhaltensbeschreibung des in den Anforderungen definierten Systems. Das bildet die Voraussetzung für die formale Überprüfbarkeit der Anforderungen und für die Generierung von Orakeln. Es werden im Gegensatz zur Hardwareverifikation keine Eigenschaften verwendet, die das innere Verhalten (4.2.1) eines Systems beschreiben, sondern nur solche, die das äußere Verhalten aufzeigen.

7.1.1 *Anwendung des 4-Variablen-Modells*

Durch das 4-Variable-Modell (4.2.1) wird ein Spezifikationsstil vorgeschlagen, der das System in das äußere, sichtbare Verhalten und das innere, von der Software beschriebene Verhalten aufteilt. Auf Grund dieser Aufteilung ergeben sich einige Vorteile. So kann dadurch vorerst das äußere Verhalten in Zusammenarbeit mit dem Kunden spezifiziert werden, ohne unnötige Vorgaben für die Software- bzw. Hardwareimplementierung zu machen. Die Anforderungen an das Verhalten können bereits auf dieser Ebene analysiert und verfeinert werden. Weiterhin werden die physikalischen und damit die Wertebereichsgrenzen des gesamten Systems durch MON und CON festgelegt. Die Abbildungen des Ein- bzw. Ausgangsverhaltens bleiben dadurch auch während des Entwicklungsprozesses unverändert und machen gleichzeitig klare Vorgaben für den Wertebereich von INPUT und OUTPUT des Systems. Was allerdings nicht vorgegeben wird, ist die Einheit und die Präzision, mit der der Wert in INPUT und OUTPUT verarbeitet wird. Diese Vorgabe wird erst durch IN und OUT gemacht und ist daher anpassbar bzw. veränderbar ohne die übrige Systembeschreibung verändern zu müssen.

Die Softwarespezifikation stellt eine Relation von INPUT und OUTPUT her und muss das Verhalten, das durch REQ vorgegeben ist, abbilden. Erst in ihr werden Programmiersprache, Aufteilung des Systems in Soft- und Hardwarekomponenten und zu verwendende Algorithmen definiert. Das Zusammenspiel von INPUT, OUTPUT und SOF beschreibt das innere Verhalten des Systems und wird in der Literatur oft als das *2-Variablen-Modell* bezeichnet. Dieser Teil der Spezifikation bildet meistens die Grundlage für Softwareentwickler, wobei durch das parallele Verständnis des äußeren Verhaltens Fehler vermieden werden können. Spezifikationen, die nach dem 4-Variable-Modell verfasst sind, ermöglichen es, das äußere Verhalten unabhängig vom inneren Verhalten zu modellieren und formal zu überprüfen. Weiterhin ist es möglich, dieses Modell so anzupassen, dass es sich für die Verwendung mit Produktlinien eignet (7.1.2).

Im Rahmen dieser Arbeit wird auf das äußere Verhalten eingegangen. Dieses bildet die Grundlage einer Spezifikation, da es zeitlich vor dem inneren Verhalten definiert wird. Weiterhin abstrahiert das äußere Verhalten Hardware und Software und ist so auch im Bereich eingebetteter Systeme überprüfbar. Nur dann, wenn das äußere Verhalten vollständig und widerspruchsfrei definiert wurde, kann ein zu entwickelndes System fehlerfrei funktionieren. Fehler, die bei der Definition des äußeren Verhaltens gemacht werden, pflanzen sich immer in die Spezifikation des inneren Verhaltens fort. Dort sind sie, wegen der Modularisierung und der Aufteilung zwischen Soft- und Hardware, schwerer zu erkennen und zu beheben.

Durch das 4-Variablen-Modell kann kein Zeitbezug zwischen den Variablen, die durch die Software verarbeitet bzw. gesetzt werden, und den entsprechenden Variablen aus dem äußeren Verhalten hergestellt werden. Änderungen in Variablen im äußeren Verhalten werden immer gleichzeitig auch in denen des inneren Verhaltens wirksam. Eine Erweiterung des 4-Variablen-Modells wird daher von Miller et al. [MT01] vorgeschlagen. Es wird vorgeschlagen, den SOF-Bestandteil des 4-Variablen-Modells zu erweitern, um den Bezug zwischen Software und Hardware besser darstellen zu können. Die Erweiterung bezieht sich auf die detailliertere Darstellung des inneren Verhaltens. Um Hardwaretreiber beschreiben zu können, werden die Relationen der Software zu IN und OUT konkretisiert. Weiterhin werden die in Software gespeicherten Werte von MON und CON separat dargestellt. Das dient der Modellierung des Zeitbezugs zwischen den in Software gespeicherten Werten und den realen beobachtbaren Werten. Im Normalfall sind beide nicht äquivalent. Der in Software verarbeitete Wert MON′ läuft dem realen Wert MON zeitlich nach, während der in Software festgelegte Wert CON′ sich in der Realität CON erst mit Verzögerung einstellt. Die detaillierte Darstellung der Software ist allerdings nicht Teil dieser Arbeit. Daher wird im Folgenden das ursprüngliche 4-Variablen-Modell verwendet.

7.1.2 *Variabilität im 4-Variablen-Modell*

Das 4-Variablen-Modell beschreibt ein einzelnes Produkt. Es beinhaltet keinerlei Variabilitätsinformation und ist somit im Bereich von Produktlinien nicht anwendbar. Es bedarf also einer Erweiterung des 4-Variablen-Modells durch ein Feature-Modell. Dadurch entsteht das im Rahmen dieser Arbeit vorgeschlagene *variantenreiche 4-Variablen-Modell*, dessen Struktur in Abb. 19 dargestellt ist.

Die Struktur des ursprünglichen 4-Variablen-Modells aus Abb. 9 bleibt unverändert. Es wird einzig eine Abbildung eines Feature-Modells F auf die Bestandteile des 4-Variablen-Modells durchgeführt. Wird eine Konfiguration K gewählt, so wird das variantenreiche 4-Variablen-Modell Var_v in das ursprüngliche 4-Variablen-Modell Var transformiert:

$$Var_v \xrightarrow{F(K)} Var$$

Damit diese Transformation möglich ist, müssen alle Bestandteile von Var_v transformierbar sein.

Abbildung der Ein- und Ausgänge: Die Menge der Ein- und Ausgänge aus dem 4-Variablen-Modell MON, CON, INPUT und OUTPUT, wird auf das Feature-Modell bezogen. Ein Feature kann einen oder mehrere Ein- und Ausgänge bedingen bzw. ausschließen. Da die Auswahl der jeweiligen Features durch die Konfiguration des Feature-Modells festgelegt wird, gilt:

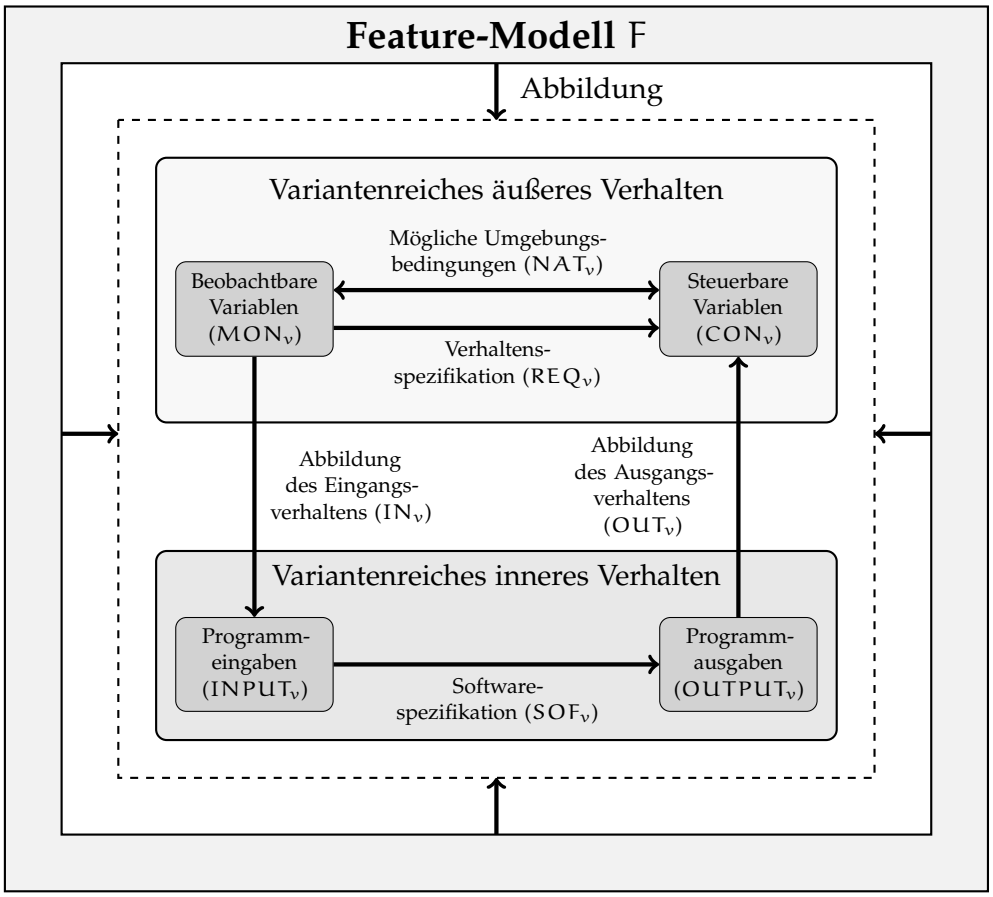


Abbildung 19: Variantenreiches 4-Variablen-Modell Var_v

$MON_v \xrightarrow{F(K)} MON$
$CON_v \xrightarrow{F(K)} CON$
$INPUT_v \xrightarrow{F(K)} INPUT$
$OUTPUT_v \xrightarrow{F(K)} OUTPUT$

Variabilität kann sich in diesem Fall durch das Vorhandensein eines Ein- oder Ausgangs auszeichnen sowie durch die Einheit, in der der Wert gemessen bzw. ausgegeben wird. Das Vorhandensein eines Wertes m_x in MON oder c_x in CON bedingt dabei immer die Entsprechung i_x in INPUT oder o_x in OUTPUT.

Weiterhin wird die Abbildung des Ein- und Ausgangsverhaltens IN und OUT also die Beziehung zwischen MON und INPUT sowie CON und OUTPUT auf das Feature-Modell bezogen:

$IN_v \xrightarrow{F(K)} IN$
$OUT_v \xrightarrow{F(K)} OUT$

$(m_x^t, i_x^t) \in IN$ gilt nur dann, wenn m_x und i_x durch die gewählte Konfiguration bedingt werden. Analog gilt $(o_x^t, c_x^t) \in OUT$ nur dann, wenn o_x und c_x durch die gewählte Konfiguration bedingt werden. Außerdem wird durch IN und OUT der Wertebereich sowie dessen Granularität festgelegt. Diese können ebenfalls variabel sein.

Abbildung der Spezifikationen und Umgebungsbedingungen: Die Umgebungsbedingungen NAT eines Systems beschreiben die maximalen und minimalen Werte, denen es ausgesetzt sein bzw. die es annehmen kann. Sie legen also den Wertebereich von MON und CON fest. Variabilität in NAT hat z.B. folgende Gründe:

- Unterschiedliche klimatischen Bedingungen (Temperatur, Luftdruck)
- Unterschiede in der Konstruktion (steifere Karosserie, stärkerer Motor)
- Unterschiede in der Zielgruppe (privater Einsatz, industrieller Einsatz)

Durch das Feature-Modell ergibt sich die Abbildung:

$$NAT_v \xrightarrow{F(K)} NAT$$

Die Verhaltensspezifikation REQ und die Softwarespezifikation SOF sind ebenfalls variabel. Funktionalität kann je nach gewählter Konfiguration hinzugefügt oder entfernt werden:

$$REQ_v \xrightarrow{F(K)} REQ$$
$$SOF_v \xrightarrow{F(K)} SOF$$

Im Folgenden wird beschrieben, wie Funktionalität aus Anforderungen extrahiert werden und mit Variabilitätsinformationen angereichert werden kann.

7.2 FORMALISIEREN VON FUNKTIONALITÄT UND ZEITVERHALTEN

Die Extraktion von Eigenschaften über die Funktionalität und das Zeitverhalten aus Anforderungen ist auf Grund der unterschiedlichen Möglichkeiten, in denen eine Anforderungsbeschreibung vorliegen kann, nur bedingt automatisierbar. Im Rahmen dieser Arbeit wird die Extraktion manuell vorgenommen. Das Verfahren wird im Folgenden genauer beschrieben:

7.2.1 *Extraktion aus natürlicher Sprache*

Bei der Extraktion aus natürlicher Sprache, müssen die Sprachkonstrukte geeignet interpretiert und in formale Eigenschaften übersetzt werden. Die Sprache muss in einen Annahme- und einen Verpflichtungsteil zerlegt werden. Ein Beispiel hierfür liefert die Anforderung:

...Die Sitzheizung darf nur aktiv sein, wenn der Sitzgurt angelegt ist...

Diese Anforderung beschreibt, dass die Sitzheizung inaktiv sein muss, wenn der Sitzgurt nicht angelegt ist. Das führt zu folgender formalen Darstellung:

$$G(\neg \mathtt{sitzgurt} \rightarrow X(\neg \mathtt{sitzheizung}))$$

Für die Übersetzung muss also der gesamte Text gelesen und interpretiert werden. Unzulänglichkeiten oder Mehrdeutigkeiten in der Sprache können diese Arbeit immens erschweren.

Spezifikationen, die in natürlicher Sprache verfasst sind, definieren meist nicht, ob die Ausführung parallel oder nebenläufig (3.3.5) von statten gehen muss. Im Normalfall wird diese Entscheidung erst durch die eigentliche Implementierung getroffen. Wird z.B. Software entwickelt, die von einem einzigen Prozessor ausgeführt wird, so ist die Ausführung nebenläufig. Das innere Verhalten ist folglich durch nebenläufige Prozesse (7.4.2) definiert. Durch geeignetes Scheduling wird das äußere Verhalten von einem Beobachter parallel wahrgenommen. Da in dieser Arbeit der Fokus auf der Beschreibung äußeren Verhaltens liegt, werden die Eigenschaften so geschrieben, dass durch sie paralleles Verhalten modelliert wird.

7.2.2 *Extraktion aus CSP*

Anforderungen können auch bereits in formaler Form vorliegen. Problematisch ist hierbei, dass die Aussagekraft der jeweiligen formalen Beschreibung die von ITL bzw. LTL übertreffen kann. Eine Übersetzung ist dann nur teilweise oder überhaupt nicht mehr möglich. Das soll nun am Beispiel der Übersetzung von CSP in ITL verdeutlicht werden. Bei CSPs wird der Fortlauf der Zeit durch das Auftreten von Ereignissen definiert. Zu einem Zeitpunkt kann immer nur ein Ereignis auftreten, das die Änderung eines Ein- bzw. Ausgangs bewirkt. Bei ITL wird eine abstrakte, diskrete Größe t eingeführt, die den Fortlauf der Zeit bestimmt. Hier können von einem Zeitpunkt zum Nächsten beliebig viele Ein- und Ausgänge ihren Wert ändern. Die Beschreibung von parallelem Verhalten kann durch beide Sprachen erfolgen. Eine nichtdeterministische Auswahl, wie sie in CSPs existiert, kann nur durch *Random-Inputs* (3.3.5) in ITL nachgebildet werden. Spezifikationen, die vollständig oder teilweise in CSP verfasst sind, können durch den präsentierten Ansatz also nicht korrekt abgebildet werden. Es ist einzig möglich, das durch den Verfasser der Spezifikation gewünschte Systemverhalten durch ITL oder LTL geeignet zu interpretieren.

7.2.3 *Extraktion aus SCR*

Die SCR-Notation (4.2.2) eignet sich ideal dazu, die Extraktion von formalen Eigenschaften aus Spezifikationen in Tabellenform zu demonstrieren. Liegt die

Current Mode	[cond1]	[cond2]	[cond3]	New Mode
[current_mode]	f	@T	-	[new_mode]
...				...

Abbildung 20: Mode Transition Table

Spezifikation, wie bei SCR, in einer genau definierten Tabellenstruktur vor, so kann die Extraktion von Eigenschaften schematisiert werden. Dadurch ist auch eine Automatisierung der Extraktion von Eigenschaften möglich. Eine *Mode Transition Table* aus Abb. 20 wird, wie folgt, behandelt:

```
property [NEW_MODE_NAME] is
assume:
  at t:
        ...
prove:
  at t+1:
  if    {
          PREV(active_mode = [current_mode]) and
          [static_conditions] and
          [changing_condition = true] and
          PREV[changing_condition = false]) or
          ...
        } then
          active_mode = [new_mode];
  elsif {
         ...
        } then
          active_mode = [new_mode];
  else
          active_mode = [old_mode];
  end if;
end property;
```

Für jeden Zustand (*Mode*), in den der Automat wechseln kann, wird eine Eigenschaft geschrieben, die die Bedingungen dafür in einer *if*-Anweisung enthält. Die *if*-Anweisung beinhaltet den derzeitigen Zustand *current_mode*, die Bedingungen *static_conditions*, die gelten müssen und die Bedingung

changing_condition, die für den Zustandsübergang ihren Wert ändern muss. *active_mode* beschreibt den Zustand, in dem sich der Automat befindet und *new_mode* ist der neue Zustand, dessen Erreichen durch die Eigenschaft festgelegt wird. Wird keine Transition zu einem neuen Zustand ausgeführt, so behält der Automat den aktuellen *active_mode* bei und es gilt $\texttt{active\_mode} = [\texttt{old\_mode}]$. Bestehen die Anforderungen aus mehreren *Mode Transition Tables*, so werden alle gleichzeitig ausgewertet und der Wechsel in den neuen jeweiligen *active_mode* geschieht parallel.

Die *Condition Table* beschreibt das Verhalten eines Ausgangs in Abhängigkeit vom Zustand und einer statischen Bedingung. Sie führt zur folgenden Eigenschaftsstruktur:

```
property [OUTPUT_NAME] is
assume:
  -- true
prove:
  at t+1: if (
   PREV(active_mode = [modes] and
   [static_condition] or
   ...
   ) then
     [output] = [value];
   elsif (
    PREV(active_mode = [modes]
       and
    [static_condition] or
    ...
    ) then
     [output] = [value];
    ...
   elsif(
    ...
    ) then
      [output] = [value];
   else
      [output] = PREV[output];
   end if;
end property;
```

Dem Ausgang *output*, dessen Verhalten durch die *Condition Table* beschrieben wird, wird ein Wert *value* zugewiesen. Die Zuweisung richtet sich nach dem jeweiligen Zustand *modes* und einem Eingangswert *static_condition*. Eine *Condition Table* dient zur Festlegung des Ausgangsverhaltens in statischen Wertebereichen. So kann z.B. definiert werden, dass eine Warnleuchte aktiviert wird, wenn der Sensorwert einen bestimmten Bereich überschreitet. Die *Event Table* beschreibt das Verhalten eines Ausgangs in Abhängigkeit vom Zustand und dem Auftreten eines Ereignisses. Sie führt zur folgenden Eigenschaftsstruktur:

```
property [EVENT_NAME] is
assume:
  -- true
prove:
  at t+1: if (
    (active_mode = [Modes] and
    [changing_condition = true]
      and
    PREV[changing_condition =
        false])
    or ...
    ) then
      [output] = [value];
    elsif (
     (active_mode = [Modes] and
     [changing_condition = true]
       and
     PREV[changing_condition =
     false])
     or ...
     ) then
       [output] = [value];
     ...
    elsif(
     ...
     ) then
       [output] = [value];
    else
       [output] = PREV[output];
    end if;
end property;
```

Im Gegensatz zur *Condition Table* muss bei der *Event Table* ein Ereignis auftreten, damit sich der Ausgang ändert. Durch sie wird der Druck auf eine Taste oder das Überschreiten eines Sensorschwellwertes beschrieben. Der Ausgang stellt ein Ereignis, wie z.B. das Inkrementieren eines Wertes, dar. Die *Selector Table* beschreibt das Ausgangsverhalten in Abhängigkeit vom Zustand. Sie führt zur folgenden Eigenschaftsstruktur:

```
property [SELECTOR_NAME] is
assume:
  -- true
prove:
  at t+1: if (
     PREV(active_mode = [current_mode]  or
     ...
     ) then
        [output] = [value];
     end if;
end property;
```

current_mode ist der Zustand, in dem sich der Automat befindet. Dem Ausgang *output* wird abhängig vom Zustand immer der Wert *value* zugewiesen.

Eine in SCR verfasste Spezifikation besteht aus beliebig vielen der oben beschriebenen Tabellen, die in jeder Kombination auftreten können. Wichtig ist daher die Definition des Zeitbezugs zwischen den Tabellen. Wie in 4.2.2 beschrieben, definieren alle Tabellen eine vorher-nachher Beziehung von Modes oder Ausgängen. Werden die Tabellen in Eigenschaften übersetzt, so wird

der Zeitbezug zwischen ihnen durch die diskrete Zeitgröße t geregelt. Sie werden parallel an diskreten Zeitpunkten ausgewertet und können gleichzeitig ihre Werte ändern. Die Änderung des Wertes wird zum nächsten Zeitpunkt wirksam. Somit kann ein Ausgang einer Tabelle, der gleichzeitig Eingang einer zweiten Tabelle ist, eine durch die zweite Tabelle definierte Änderung erst einen Zeitschritt später bewirken. Abb. 21 zeigt eine SCR-Beispielspezifikation,

Current Mode	c	New Mode
M1	@T	M2
M2	@F	M1

Mode	c
M1	c = 1
M2	c = 0

```
property Mode_Table is
assume:
  -- true
prove:
  at t+1: if (
     PREV(active_mode = Mode1)
     and PREV(c /= 1)
     and c = 1
    ) then
      active_mode = Mode2;
    elsif (
     PREV(active_mode = Mode2)
     and PREV(c = 1)
     and c /= 1
    ) then
      active_mode = Mode1;
    else
      active_mode =
       PREV(active_mode);
    end if;
end property;
```

```
property Selector_Table is
assume:
  -- true
prove:
  at t+1: if (
    PREV(active_mode = Mode1)
    ) then
      c = 1;
    elsif (
     PREV(active_mode = Mode2)
    ) then
      c = 0;
    end if;
end property;
```

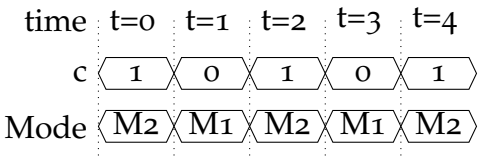


Abbildung 21: SCR-Beispiel

die aus einer Mode Transition Table und einer Selector Table besteht. Weiterhin sind die zur Spezifikation zugehörigen Eigenschaften und ein gültiges Simulationsbeispiel gegeben. Der Wert von c hängt ausschließlich vom Mode ab, während die in der Mode Transition Table definierten Übergänge von der Änderung von c abhängen.

7.2.4 *Extraktion aus Aktivitätsdiagrammen*

Ein Aktivitätsdiagramm (3.2) stellt den strukturierten Ablauf eines Vorgangs an Hand von Aktionen und Objekten dar. Hinter jeder Aktion liegt eine Funktionalität, die das Systemverhalten beschreibt. Um Eigenschaften aus Aktivitätsdiagammen zu extrahieren, muss im ersten Schritt die Struktur des Aktivitätsdiagramms in ITL oder LTL nachgebildet werden. In einem zweiten Schritt muss dann abhängig von der Aktion bzw. den Aktionen, die gerade ausgeführt werden, eine Funktionalität definiert werden. Die Eigenschaften für die Funktionalität müssen folglich nur dann halten, wenn sich das System in der jeweilig zutreffenden Aktion befindet.

Abb. 22 zeigt beispielhaft ein Aktivitätsdiagramm, an dem das Grundlegende Konzept der Übersetzung in ITL erläutert wird.

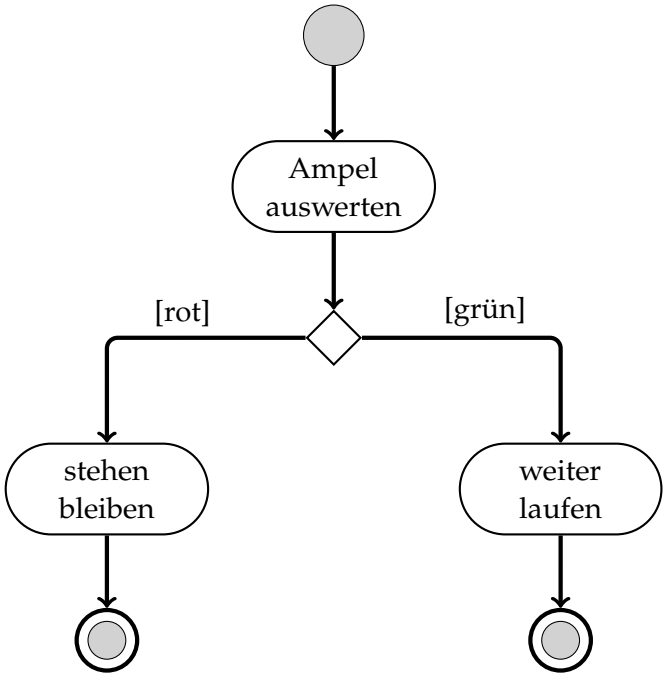


Abbildung 22: Beispielhaftes Aktivitätsdiagrammen

Um das Aktivitätsdiagramm zu starten, wird in ITL, ähnlich wie bei der Darstellung von Zustandsautomaten, ein Resetsignal verwendet.

```
property start is
assume:
  at t: reset = '1';
prove:
  at t+1:
        activity = Ampel_auswerten;
end property;
```

Die weitere Übersetzung beschränkt sich auf die Überführung des Kontrollflusses in ITL. Es wird für jede Aktion eine separate Eigenschaft geschrieben, die angibt, welche Aktionen auf diese folgen können.

```
property start is
assume:
  at t: reset = '0';
  at t: activity = Ampel_auswerten;
prove:
  at t+1: if (Ampel = grün) then
        activity = weiter_laufen;
     elsif (Ampel = rot) then
        activity = stehen_bleiben;
     end if;
end property;
```

Sind im Aktivitätsdiagramm Gabelungen enthalten, so gibt es mehrere gleichzeitig aktive Aktionen. Um diese in ITL zu übersetzen, muss die gleiche Anzahl Platzhalter (*activity*) definiert werden.

Der zeitliche Zusammenhang in Aktivitätsdiagrammen ist durch den Wechsel von einer in die nächste Aktion gegeben. Die Dauer der einzelnen Aktionen ist nicht definiert und kann für jede Aktion variieren. Durch die Eigenschaften wird daher ausschließlich die Struktur des Aktivitätsdiagramms nachgebildet. Ein Wechsel in eine neue Aktion geschieht von t nach $t+1$. Will man die diskrete Zeitspanne, die eine Aktion für ihre Ausführung benötigt modellieren, so muss die Möglichkeit geschaffen werden, das Ende der jeweiligen Aktion zu signalisieren. Dies kann z.B. derart modelliert werden, dass für jede Aktion ein separates Endesignal eingeführt wird, das gesetzt wird, wenn die Aktion ihre Ausführung beendet hat und mit der nächsten Aktion fortgefahren werden kann. Für die Ampel ergäbe sich dann z.B. folgende Eigenschaft:

```
property start is
assume:
  at t: reset = '0';
  at t: activity = Ampel_auswerten;
prove:
  at t+1: if (Ampel = grün and Ampel_auswerten_ende = '1') then
        activity = weiter_laufen;
     elsif (Ampel = rot and Ampel_auswerten_ende = '1') then
        activity = stehen_bleiben;
     else
        activity = Ampel_auswerten;
     end if;
end property;
```

7.3 FORMALISIEREN VON VARIABILITÄT

Spezifikationen für Systeme, die in mehreren Konfigurationen vorliegen können, beinhalten neben der Information über das Systemverhalten auch Informationen über die Variabilität. Die Behandlung von Variabilität stellt eine immense Herausforderung dar. Die Menge, der aus einer Produktlinie generierbaren Produkte übersteigt häufig die Millionenmarke. Bei der Überprüfung der Anforderungen für eine Produktlinie muss sichergestellt werden, dass sie vollständig und widerspruchsfrei sind. Nach Möglichkeiten sollten auch keine Anforderungen redundant sein. Die Grundvoraussetzung für eine automatisierte Überprüfung bildet das Hinzufügen von Variabilität zu Eigenschaftssätzen, das im Folgenden genauer erläutert wird.

7.3.1 *Feature-Modelle*

Voraussetzung für die Behandlung von Variabilität ist die werkzeuggestützte Modellierung eines Feature-Modells (5.2) sowie dessen Übersetzung in einen booleschen Ausdruck. Beides wird durch ein *Plugin* im *Eclipse Framework* erreicht. Es unterstützt die Modellierung beliebiger Feature-Modelle sowie deren Übersetzung in Konjunktive Normalform (KNF). Die KNF wurde gewählt, damit der boolesche Ausdruck nicht nur zu Eigenschaftssätzen hinzugefügt werden kann, sondern auch bei Bedarf direkt durch einen SAT-Solver bearbeitet werden kann.

7.3.2 *Produktlinieneigenschaften*

Um die Anforderungen formal überprüfen zu können, müssen die Eigenschaftssätze durch die im Feature-Modell enthaltene Variabilität erweitert werden. Es entstehen die *Produktlinieneigenschaften*.

> **Definition 7.1: *Produktlinieneigenschaften* -** Wird eine Menge von Eigenschaften durch ein Variabilitätsmodell erweitert, so entstehen Produktlinieneigenschaften. Diese beschreiben für jede durch das Variabilitätsmodell gültig auswählbare Konfiguration spezifisches Verhalten.

Der obere Teil von Abb. 23 zeigt das schematische Vorgehen zum Erzeugen von Produktlinieneigenschaften. Die Voraussetzung für eine Erweiterung stellt die Übersetzung des Feature-Modells in einen booleschen Ausdruck dar (5.2). Dieser Ausdruck wird nur dann wahr, wenn eine gültige Konfiguration gewählt wurde. Er muss gelten, damit die weiteren Eigenschaften über das korrekte

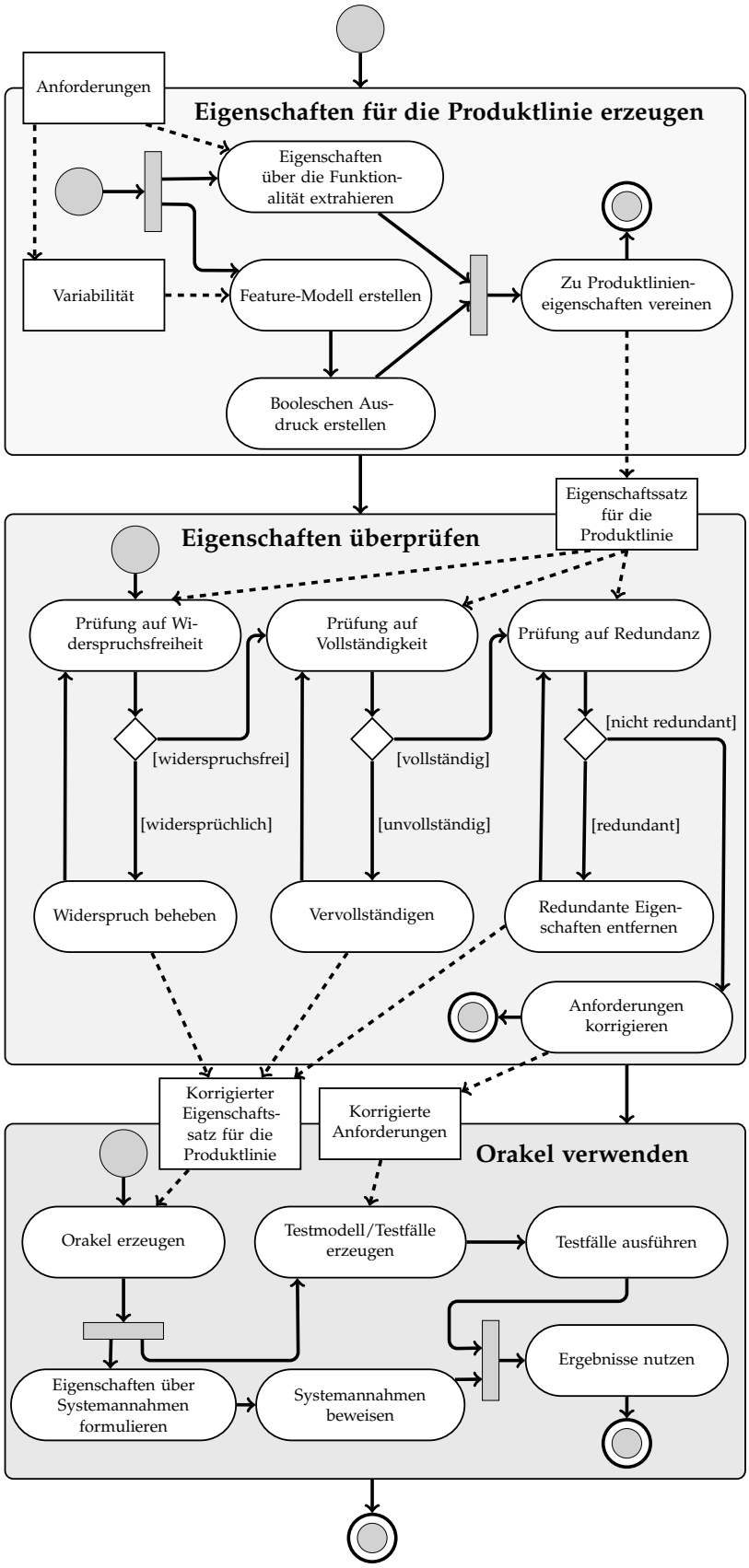


Abbildung 23: Eigenschaften im Produktlinienkontext

Verhalten in Betracht gezogen werden müssen. Eine ungültige Konfiguration beschreibt kein gültiges Produkt und ist somit nicht durch Eigenschaften spezifiziert. Es ergibt sich die LTL-Eigenschaft:

$$G(Feature\_Model \rightarrow (eigenschaft_1, eigenschaft_2, \ldots, eigenschaft_n))$$

Dass das Feature-Modell gültig ist, muss zu jeder Eigenschaft im Eigenschaftssatz, der Produktlinien beschreibt, hinzugefügt werden. Weiterhin müssen die einzelnen Features auf die Eigenschaften abgebildet werden. Eine Eigenschaft beschreibt im klassischen Sinn nur das gültige Verhalten eines Systems. Durch eine Abbildung der Features auf die Eigenschaften lässt sich festlegen, welches Feature oder welche Featurekombination im Produkt enthalten sein muss bzw. nicht enthalten sein darf, damit die Eigenschaft für diese Konfiguration relevant ist. Verallgemeinert lassen sich Eigenschaften erweitern, indem der boolesche Ausdruck über die relevante Feature-Kombination mit der linken Seite der Implikation konjugiert wird:

$$G(((feature\_kombination) \wedge annahme) \rightarrow X(verpflichtung))$$

Die in dieser Arbeit betrachteten Implikationen umfassen immer den vorherigen und den aktuellen Zeitpunkt. Alle in 7.2 präsentierten Formen lassen sich auf diese Form zurückführen. Hängt eine Eigenschaft ausschließlich von Pflichtknoten im Feature-Modell ab, so erübrigt sich die Abbildung der Pflichtknoten auf die Eigenschaft, da der entstehende boolesche Ausdruck für gültige Konfigurationen immer wahr ist. Eine Eigenschaft, die das Verhalten der Sitzheizung beschreibt, ist z.B. nur relevant, wenn die Sitzheizung auch Teil des gewählten Produkts ist.

$$G((sitzheizung_{feature} \wedge \neg sitzgurt) \rightarrow X(\neg sitzheizung))$$

Ein Eigenschaftssatz für die Anforderungen einer Produktlinie besteht also immer aus den Eigenschaften über die Funktionalität, der Eigenschaft, dass das Feature-Modell wahr ist sowie der Abbildung der jeweiligen Features bzw. der Feature-Kombinationen auf die einzelnen Eigenschaften. Eine große Herausforderung beim Aufstellen von Eigenschaften für Produktlinien stellen die Definition der einzelnen Features sowie die Abbildung der Features auf die Elemente in der Domänenebene (5.1) dar. Ein Feature kann eine einzelne Anforderung oder eine Menge von Anforderungen repräsentieren. Weiterhin kann ein Feature verwendet werden, um darunter liegende Features zu strukturieren. Repräsentiert ein Feature eine Menge von Anforderungen, so muss diese Menge abgeschlossen in einem einzelnen Produkt darstellbar sein. Ein Feature darf nie Anforderungen enthalten, die sich in einem Produkt ausschließen würden. Weiterhin sollten verpflichtende und optionale Anforderungen nie auf das selbe Feature abgebildet werden.

Die Definition der einzelnen Features kann entweder schon in den Anforderungen vorgegeben sein oder muss manuell geschehen. Jedes Feature wird

auf eine Menge von Anforderungen oder auf eine einzelne abgebildet. Bei der Wahl der einzelnen Features muss darauf geachtet werden, dass die ihnen zugeordneten Anforderungen einen abgeschlossenen Teilbereich des Produkts repräsentieren.

7.4 FORMALISIERBARKEIT VON ANFORDERUNGEN

Für die Extraktion von Verhalten aus dem Anforderungsdokument wird neben LTL die Eigenschaftsbeschreibungssprache ITL verwendet. Die in dieser Arbeit verwendeten Werkzeuge basieren auf der Interpretation von Eigenschaften in ITL. Da diese aus dem Bereich der formalen Hardwareverifikation stammt, muss erörtert werden, in wie fern sich äußeres Verhalten (4.2.1) auf ITL abbilden lässt. Weiterhin wird das zu Grunde liegende Zeitmodell eingeführt und genaue Abbildungsvorschriften definiert.

7.4.1 *Zeit- und Vollständigkeitsbegriff*

Für die Generierung von Orakeln auf Basis von Cando-Objekten (6.1) und die Prüfung der Eigenschaftssätze auf Vollständigkeit (6.2) werden die Anforderungen als ITL-Eigenschaften formuliert und dann automatisiert bearbeitet. Das Zeitmodell muss so gewählt werden, dass es eine Verbindung zwischen der jeweiligen Art (4.2) der Anforderungsformulierung und den in ITL geschriebenen Eigenschaften herstellt. Weiterhin darf das Zeitmodell nicht über die Beschreibungsmöglichkeiten von ITL hinausgehen.

ITL ist generell ausschließlich dazu in der Lage, synchrones Verhalten zu definierten Zeitpunkten zu beschreiben. Die Zeitpunkte t stellen nach der ursprünglichen Intention den Takt eines Hardwarebausteins dar. Diese Betrachtungsweise eignet sich für den in dieser Arbeit präsentierten Ansatz nicht. Es ist vielmehr nötig, Ereignisse als atomare Aktionen ohne Dauer zu behandeln, die nacheinander oder auch gleichzeitig auftreten können. Die Zeit zwischen dem Auftreten der Ereignisse ist nicht spezifiziert. Insofern ist die Zeitbasis t in der ITL-Beschreibung nicht als fortlaufender Takt zu interpretieren, sondern so, dass das Fortschreiten der Zeit durch das Auftreten mindestens eines neuen Ereignisses ausgelöst wird. Das Zeitmodell wird im Folgenden anhand eines Inverters genauer verdeutlicht:

```
property INVERTER is
assume:
  -- true
prove:
  at t: if PREV(input = '0') then
          output = '1';
        elsif PREV(input = '1') then
          output = '0';
        end if;
end property;
```

Der Zeitpunkt $t-1$, der durch PREV referenziert wird, bezieht sich nicht auf den vorangegangenen Takt, sondern auf ein Ereignis, das zuvor eingetreten ist. An dieses Ereignis schließt sich das Setzen des Ausgangs an. Die Zeit, die zwischen den beiden Ereignissen vergeht, ist nicht definiert. Es muss nur sicher gestellt sein, dass zwischendurch kein weiteres Ereignis eintritt, das output setzt. Abb. 24 zeigt eine Beispielereignisfolge. In ITL wird im Gegensatz zu CSPs

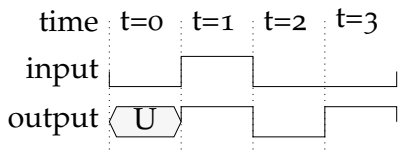


Abbildung 24: Inverter

das parallele Auftreten von input und output erlaubt. output von t hängt hier von input bei $t-1$ ab, während input von t output von $t+1$ bedingt. Hierdurch wird eine kompakte Darstellung möglich, die sich graphisch an der von Signalen einer Hardwareschaltung orientiert (3.3.6).

Welchen Einfluss eine Spezifikationslücke auf das Ausgangsverhalten hat, wird im Folgenden beschrieben. Eine Spezifikationslücke liegt dann vor, wenn ein Ausgang für eine bestimmte Kombination von Eingangswerten nicht definiert ist und die Eigenschaften somit unvollständig sind. Die Eigenschaften werden dafür folgendermaßen verändert:

```
property INC_INVERTER is
assume:
  -- true
prove:
  at t: if PREV(input = '0') then
          output = '1';
        end if;
end property;
```

Für den Fall, dass input 1 wird, ist output undefiniert. Abb. 25 zeigt die ITL Ereignisfolge. ITL erwartet im Gegensatz zu CSPs in jedem Zeitschritt auch

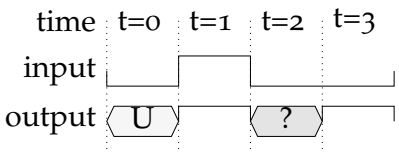


Abbildung 25: Unvollständiger Inverter

einen Wert für den Ausgang. Dieser ist im vorliegenden Fall undefiniert, wenn PREV(input $=' 1'$). output behält seinen vorherigen Wert dann also nicht. ITL besitzt folglich keine Halteeigenschaft.

7.4.2 *Eigenschaften für Orakel*

Im Folgenden werden die möglichen Konzepte und deren Einschränkungen für die Generierung von Orakeln aus ITL aufgezeigt. Abb. 26 zeigt die Struktur

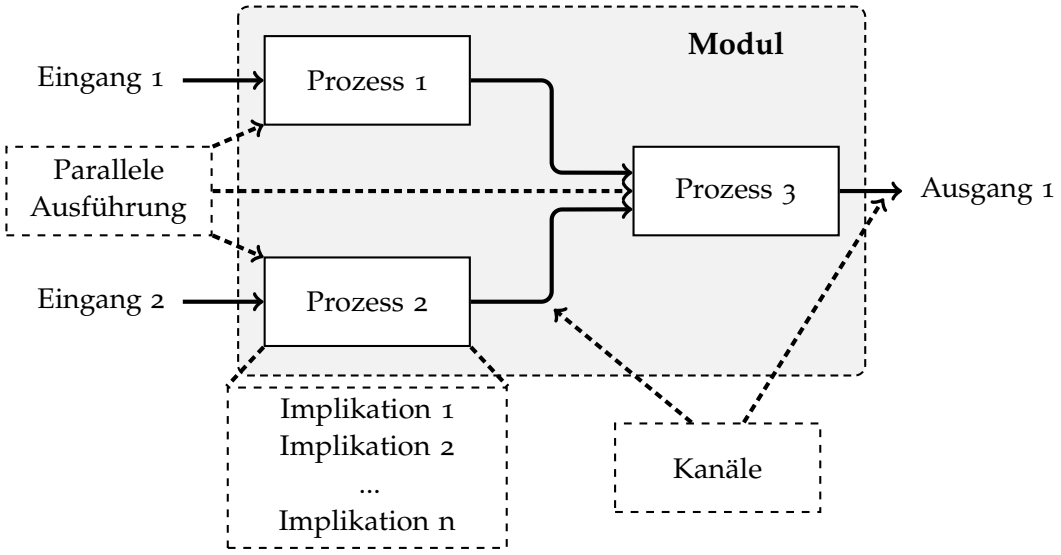


Abbildung 26: Durch ITL beschriebene Struktur

eines durch ITL beschriebenen Moduls. Der Modulbegriff ist äquivalent zu dem, der im Bereich der Cando-Objekte definiert wurde (6.1). Das Modul besteht aus drei parallelen Prozessen, deren Verhalten durch Implikationen, die Berechnungen enthalten können, beschrieben wird und die über Kanäle miteinander und mit der Umgebung des Moduls kommunizieren.

Prozess: Ein Prozess abstrahiert die Interaktionen von Modulen und ihrer Umgebung auf mathematische Art und Weise. Es handelt sich also im Wesentlichen um Eigenschaften, denen ein Modul genügen muss, um korrekt zu

funktionieren. Ein Prozess beschreibt Reaktionen des Moduls auf Ereignisse, die während seiner Ausführung auftreten. Ereignisse werden dabei durch eine Veränderung der Umgebung des Moduls ausgelöst. Ein solches Ereignis kann z.B. durch das Erreichen eines Sensorschwellwerts oder das Drücken eines Tasters auftreten. Weitere Prozesse innerhalb des Moduls können ebenfalls zu Ereignissen führen. Solche Ereignisse können z.B. durch das Ablaufen eines internen Zählers oder durch das Erreichen eines Fehlerzustands auftreten. Eigenschaften, die in ITL verfasst sind, beschreiben die Interaktion von Modulen und ihrer Umgebung auf definierte Art und Weise mit dem Ziel, Eigenschaften für ein korrekt arbeitendes Modul zu formulieren. Jede Eigenschaft stellt also einen Prozess dar, dessen Verhalten durch die in ihr enthaltenen Implikationen definiert wird.

Implikation: Eine Implikation beschreibt eine „wenn..., dann...“ Beziehung zwischen zwei Ereignissen. $a \rightarrow b$ sagt also aus, dass, wenn das Ereignis a gilt, auch b auftreten muss. Ein Prozess besteht aus Implikationen. Dabei existiert eine Vielzahl äquivalenter Darstellungsformen (z.B. in CSP $(a \rightarrow A) \parallel (a \rightarrow B) = a \rightarrow (A \parallel B)$). In ITL sind mehrere, unterschiedliche Darstellungen möglich, die alle die selbe Implikation beschreiben (Abb. 27). Eine ITL-Beschreibung ist also nicht kanonisch.

```
property IMPLICATION is
assume:
  at t: a = 0;
prove:
  at t+1: b = 0;
end property;
```

```
property IMPLICATION is
assume:
  -- true
prove:
  at t+1:
  if (PREV(a = 0)) then
    b = 0;
  end if;
end property;
```

Abbildung 27: Äquivalentes ITL

Berechnung: Die Modellierung eines Moduls erfordert, neben der Modellierung von Zeit, auch die Modellierung von Berechnungen, die durch Prozesse im Modul durchgeführt werden sollen. In Kapitel 9 werden Orakel als simulierbare VHDL-Beschreibung generiert. Daraus ergeben sich einige Einschränkungen für die Berechnungen, die verwendet werden dürfen. Im Folgenden wird ein Modul betrachtet, das den Wert an seinem Eingang verdoppelt und ausgibt [Hoa04].

```
property DOUBLE is
assume:
  -- true
prove:
  at t: right = PREV(left) + PREV(left);
end property;
```

Abb. 28 zeigt eine nach dieser Eigenschaft gültige Ereignisfolge.

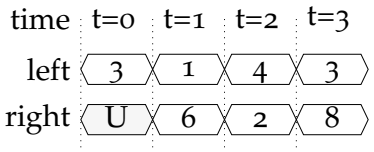


Abbildung 28: Verdoppeln

Die Darstellung unterliegen wieder dem in 7.4.1 beschriebenen Zeitmodell. Die Berechnung hängt von einem variablen Eingang ab, der ganzzahlige Werte verdoppelt.

Kommunikation: Die Kommunikation von Prozessen bzw. Eigenschaften muss modelliert werden können. Kommunikation ist immer ein gerichteter Ablauf mit einem Sender und einer beliebigen Anzahl von Empfängern. In CSP z.B. wird Kommunikation durch Kanäle zwischen Prozessen oder zwischen einem Prozess und seiner Umgebung modelliert. Auf diesen Kanälen können Nachrichten gesendet bzw. empfangen werden. Signale in ITL haben generell keine Richtung. Aus der Eigenschaft für das Verdoppeln von Eingangswerten geht nicht klar hervor, ob es sich bei `left` bzw. `right` um Ein- oder Ausgänge handelt. Es ist auch nicht möglich, diese Information in ITL zu spezifizieren. Da ITL aus der Hardwareverifikation stammt, wird diese Information in der Praxis durch die Komponente, auf der die Eigenschaften bewiesen werden sollen, festgelegt. In ihr wird beschrieben, welche Signale Eingänge und welche Ausgänge sind. Im Bereich von Cando-Objekten wird zu diesem Zweck zusätzlich zu den Eigenschaften in ITL eine *VHDL-Entity* hinterlegt, in der Ein- und Ausgänge sowie deren Typen definiert sind. Man kann mit ITL also augenscheinlich Eigenschaften verfassen, die von einem Ausgangswert auf einen Eingangswert zu einem früheren Zeitpunkt schließen. Solche Eigenschaften sind allerdings nicht beweisbar, wenn der Eingang, auf den geschlossen wird, durch die Umgebung und nicht durch eine verbundene und bekannte Komponente festgelegt wird.

Parallelismus: Parallelismus beschreibt die gleichzeitige Ausführung mehrerer Prozesse. Diese können miteinander interagieren. Das Auftreten eines Ereignisses kann Aktionen mehrerer Prozesse bedingen. Zwei Prozesse sind unabhängig voneinander, wenn gilt: $\alpha P_1 \cup \alpha P_2 = \{\}$. Das jeweilige Alphabet

des einen Prozesses darf also keine Elemente des anderen enthalten. In einem Modul sind Prozesse aber in der Regel teilweise oder vollständig voneinander abhängig. Sind Kanäle definiert, so können Prozesse nicht nur durch ein gemeinsames Alphabet in Bezug auf die Umgebung, sondern auch in Bezug auf gemeinsam genutzte Kanäle voneinander abhängig sein. Parallelismus wird dann zur Herausforderung, wenn mehrere Prozesse auf dieselben Ressourcen zugreifen. Dann kann es z.B. zum Deadlock kommen, der auftritt, wenn sich mehrere Prozesse gegenseitig blockieren. Deadlock-Situationen sollten schon während der Entwicklung eines Moduls erkannt und behoben werden. Eine spezielle Form des Deadlocks bildet der Livelock, in dem das Modul nicht in einem Zustand verharrt, sondern noch Zustandswechsel durchführt, die aber nicht zum gewünschten Ergebnis führen. Das Modul befindet dann in einem abgeschlossenen Zustandsraum, den es nicht mehr verlassen kann. Durch widerspruchsfreie ITL-Eigenschaften besteht alleine die Möglichkeit durch den Beweis der ITL-Eigenschaften Deadlock-Situationen in einem bestehenden Modul zu erkennen. Ein Deadlock wird dadurch nicht erzeugt.

Parallelismus erschwert in ITL vor allem die Modellierung des korrekten Zeitbezugs. Analog zu Hardwarebeschreibungssprachen muss die Formulierung von Zyklen verhindert werden.

Determinismus: Ein Modul verhält sich deterministisch, wenn die Auswahl, welches Ereignis eintrifft, einzig durch die Umgebung kontrolliert wird. Im Umkehrschluss ist ein Modul also nicht-deterministisch, wenn das Ergebnis nicht durch die Umgebung festgesetzt wird und das Zustandekommen des jeweiligen Ergebnisses durch die Umgebung nicht nachvollziehbar ist. Dabei kann das durch die Umgebung kontrollierte Modul intern ein vollkommen deterministisches Verhalten zeigen, das von außen allerdings nur interpretierbar ist. Die nicht-deterministische Beschreibung von Verhalten ist ein probates Mittel, um dem Entwickler Freiheitsgrade in der Gestaltung zu eröffnen. So kann ein Getränkeautomat bei zu viel bezahltem Geld Rückgeld geben. Die Höhe des zurückzuerstattenden Betrags ist festgelegt. Die Stückelung des Rückgelds bleibt allerdings dem Getränkeautomaten überlassen. In ITL kann ohne das Hinzufügen von *Random-Inputs* einzig beschrieben werden, dass die korrekte Menge an Rückgeld gegeben werden muss.

```
property ONEBACK is
assume:
  -- true
prove:
  at t+1: accumulated_output = 1;
end property;
```

Die ursprüngliche Intention von ITL ist es, Eigenschaften zu beschreiben, die auf Modulen oder Systemen bewiesen werden können. Das innere Verhalten

eines Moduls ist dabei immer deterministisch. Will man die Stückelung steuern, so muss ein weiterer Eingang back hinzugefügt werden.

```
property ONEBACK is
assume:
  -- true
prove:
  at t+1:
  if (PREV(back = '0')) then
    output = one_euro;
  elsif (PREV(back = '1')) then
    output = fifty_cent and
    NEXT(output = fifty_cent);
  end if;
end property;
```

In ITL kann das Steuern durch die Umgebung mit Hilfe einer if-Anweisung modelliert werden. Handelt es sich bei back um einen Eingang, den der Benutzer des Automaten steuern kann, so ist das Verhalten des Automaten deterministisch. Stellt back einen *Random-Input* dar, so ist das Verhalten nicht deterministisch, da es von einem Zufallswert abhängt.

8

SPEZIFIKATIONSPRÜFUNG UND ORAKELGENERIERUNG

Ziel dieser Arbeit ist es, Eigenschaftssätze, die aus Anforderungen abgeleitet wurden, zu überprüfen und daraus ein ausführbares Modell zu generieren. Diese Anforderungen müssen nicht dem klassischen Feld der Hardwareentwicklung zuzuordnen sein, sondern können z.B. auch Verhaltensbeschreibungen für eingebettete Systeme sein oder das äußere Verhalten (4.2.1) eines Systems beschreiben. Weiterhin sollen neben Anforderungen für einzelne Produkte auch Anforderungen für Produktlinien (Kapitel 5) verwendbar sein, die sich durch Variabilität auszeichnen. Im Folgenden werden die im Rahmen dieser Arbeit erstellten Konzepte für die Anforderungsanalyse und die Orakelgenerierung präsentiert.

8.1 PROBLEMBESCHREIBUNG

Die Überprüfung von Anforderungen, die das äußere Verhalten eines Systems beschreiben, sollte formal unterstützt werden. Lücken und Widersprüche in den Anforderungen können gerade bei sicherheitsrelevanten Systemen zu schweren Unglücken führen. Eine formale Überprüfung der Anforderungen anhand der in Kapitel 6 präsentierten Algorithmen ist daher von großem Wert.

Das Problem der Anforderungsanalyse wird weiter erschwert, wenn die Anforderungen eine Produktlinie beschreiben. Die Produktlinienentwicklung beginnt mit der Entwicklung auf Domänenebene (5.1). Auf dieser Ebene sind bereits ein Großteil bzw. ein kompletter Satz von der Anforderungen für die Konfigurationen auf Anwendungsebene spezifiziert. Sollte in den Anforderungen auf Domänenebene bereits ein Widerspruch, eine Unvollständigkeit oder Redundanz (8.3.1) vorhanden sein, so werden sich diese Fehler auf die Anwendungsebene fortpflanzen. Durch die Verwendung von Features müssen in den Anforderungen enthaltene Fehler nicht zwangsläufig in jeder Konfiguration vorkommen. Ein Fehler kann z.B. nur bei einer bestimmten Feature-Auswahl oder Feature-Kombination im Produkt enthalten sein. Würden ausschließlich die Anforderungen auf Anwendungsebene überprüft, so könnte ein Fehler auf Domänenebene unerkannt bleiben.

Eine Überprüfung aller Konfigurationen auf Anwendungsebene ist wegen der meist großen Anzahl der möglichen Konfigurationen nicht praktikabel

und eignet sich daher nur für die Überprüfung von Anforderungen außerhalb des Produktlinienkontexts. Es muss ein Weg gefunden werden, mit dem die Anforderungen bereits auf Domänenebene überprüft werden können. Hierbei besteht das Problem unter anderem darin, dass Anforderungen auf Domänenebene überspezifiziert sind. Erst durch die Verbindung mit einem Variabilitätsmodell, wie dem Feature-Modell (5.2), entsteht ein konsistentes Anforderungsdokument. Ein formaler Ansatz, der die Anforderungen bereits auf Domänenebene überprüft, muss daher eine Verbindung zwischen dem Variabilitätsmodell und den Anforderungen herstellen.

Durch die Verwendung eines formalen Ansatzes liegen die Anforderungen in formaler Form vor. Diese bilden gleichsam ein Modell des System- oder Modulverhaltens, das als Grundlage für den Test dienen kann. Wird das Modell in eine ausführbare Form übersetzt, so kann es als Orakel (2.3.2) Verwendung finden. Anforderungen in formaler Form können weiterhin als Grundlage für den Beweis von Systemannahmen dienen. Hierdurch wird durch das Ausschließen unsicherer Situationen gewährleistet, dass sich das System bzw. Modul nur in sicheren Zuständen befindet.

Durch das im Folgenden präsentierte Konzept sollen Lösungshilfen für die folgenden Schritte aus Abb. 13 in der Produktlinienentwicklung angeboten werden:

- Formale Überprüfung der Anforderungsdokumente auf Domänenebene
- Plausibilitätsüberprüfung der Anforderungen auf Domänenebene durch Systemannahmen
- Automatisierte Generierung eines simulierbaren Orakels für die Überprüfung des Designs auf Domänenebene
- Strukturierte Darstellung der Testfälle auf Domänenebene

Außerdem soll die Entwicklung einzelner Produkte in folgenden Punkten unterstützt werden:

- Formale Überprüfung der Anforderungsdokumente auf Anwendungsebene
- Plausibilitätsüberprüfung der Anforderungen auf Anwendungsebene durch Systemannahmen
- Automatisierte Generierung eines simulierbaren Orakels für die Überprüfung des Designs auf Anwendungsebene

Die eingesetzten Techniken für einzelne Produkte bzw. für Produktlinien unterscheiden sich ausschließlich im Vorhandensein eines Variabilitätsmodells. Die zur Überprüfung der Anforderungen und zur Orakelgenerierung eingesetzten Algorithmen sind die selben.

8.2 GESAMTKONZEPT

Abb. 29 zeigt das Konzept, das in dieser Arbeit zur Anforderungsanalyse und Orakelgenerierung für einzelne Produkte sowie im Zusammenhang mit Produktlinien vorgeschlagen wird. Es gliedert sich in drei Bereiche, die mit-

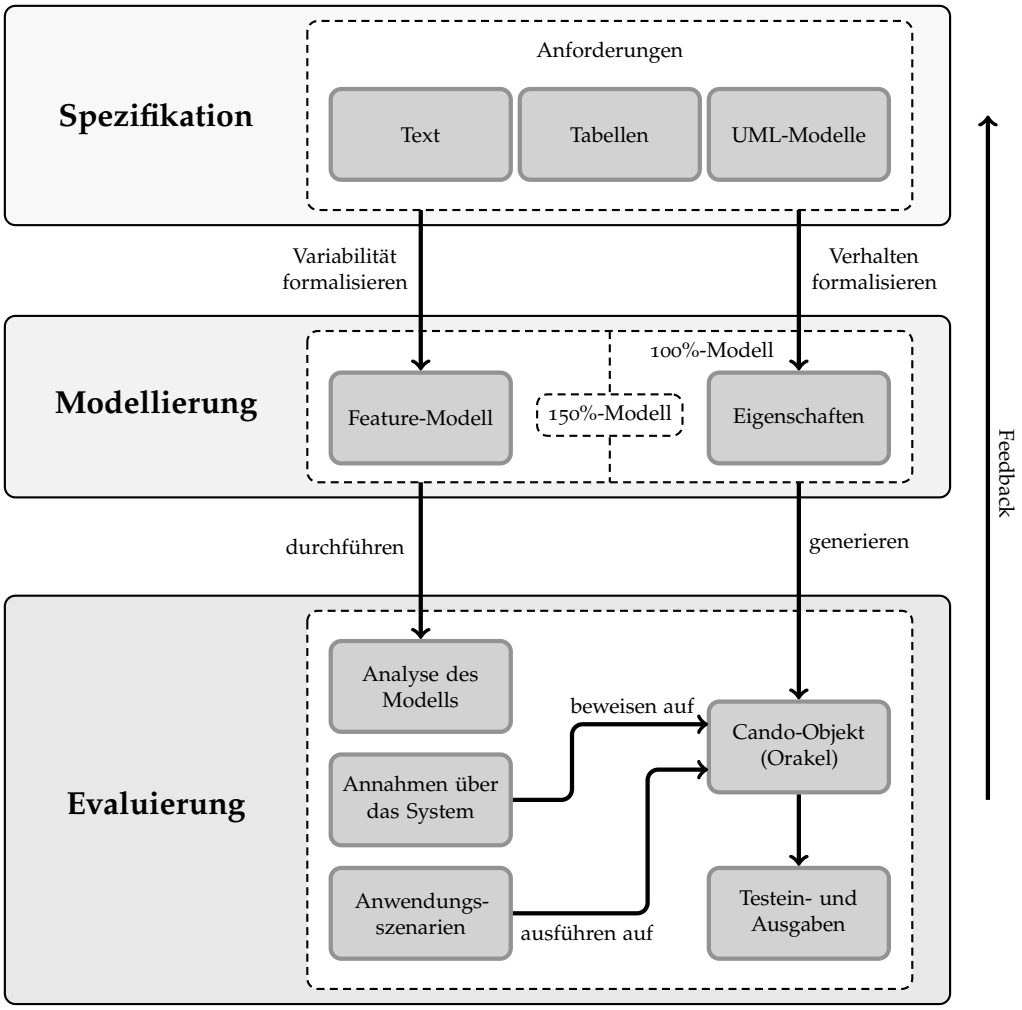


Abbildung 29: Gesamtkonzept

einander verbunden sind. Als Grundlage dient eine Spezifikation, die auf unterschiedliche Art und Weise (4.2) verfasst sein kann. Diese häufig nur teilweise formalen Anforderungen werden formalisiert, indem formale Eigenschaften über das Verhalten extrahiert werden (7.1) und die Variabilität und Gemeinsamkeiten der Produktlinie durch ein Feature-Modell dargestellt werden. Aus den Modelleigenschaften und dem Feature-Modell wird dann das 150%-Modell gebildet, das Informationen über alle Konfigurationen der Produktlinie enthält (7.3). Handelt es sich bei den zu Grunde liegenden Anforderungen nicht um eine Produktlinie, sondern um ein einzelnes Produkt, so entfällt die Formalisierung der Variabilität und es muss einzig das Verhalten formalisiert werden und das 100%-Modell entsteht. Das 100%-Modell bzw.

das 150%-Modell wird dann auf Vollständigkeit überprüft und es werden möglicherweise enthaltene Widersprüche entfernt. Zuletzt wird ein Cando-Objekt aus den Eigenschaften des jeweiligen Modells generiert, das als Orakel dient. Aus der Ausführung von Anwendungsszenarien auf dem Orakel können Testein- und ausgaben für einen abschließenden Systemtest gewonnen werden. Des Weiteren können auf dem Orakel Annahmen über das System bewiesen werden.

Von den unteren Bereichen gibt es ein Feedback in die höheren Bereiche. Werden Widersprüche, Unvollständigkeit oder Fehler gefunden, so müssen diese in er Spezifikation korrigiert werden. In folgenden Fällen gibt es ein Feedback von der Evaluierung in die Spezifikation und Modellierung:

- Eine Systemannahme schlägt fehl, da
  - die Systemannahme Fehler enthält.
  - die Anforderungen über das Systemverhalten fehlerhaft sind.
  - Fehler beim Formalisieren des Verhaltens gemacht wurden.
  - Fehler beim Formalisieren der Variabilität gemacht wurden.
- Die Vollständigkeitsbewertung findet eine Lücke, da
  - die Anforderungen eine ungewollte Spezifikationslücke enthalten.
  - Freiheitsgrade bei der Vollständigkeitsbewertung nicht beachtet wurden.
- Die Vollständigkeitsbewertung findet einen Widerspruch, da die Anforderungen widersprüchlich sind.
- Die Vollständigkeitsbewertung findet Redundanz, da redundante Information in den Anforderungen formuliert wurden.

In folgenden Fällen gibt es ein Feedback von der Modellierung in die Spezifikation:

- Werden beim Formalisieren des Verhaltens Fehler entdeckt, müssen sie in den Anforderungen behoben werden.
- Werden beim Formalisieren der Variabilität Fehler entdeckt, müssen sie in den Anforderungen behoben werden.

Da das Formalisieren von Variabilität und Verhalten manuell geschieht, werden die Anforderungen in diesem Schritt nicht nur manuell auf ihre Konsistenz geprüft, sondern auch auf ihre Sinnhaftigkeit. Diese Überprüfung kann nicht automatisiert werden.

8.3 ANALYSE VON EIGENSCHAFTSSÄTZEN

Die Eigenschaftssätze, die aus der Spezifikation extrahiert wurden, müssen nach Abb. 23 überprüft werden. Im Folgenden werden die möglichen in den Eigenschaftssätzen enthaltenen Fehler erläutert sowie ein Konzept präsentiert, mit dem diese erkannt werden können.

8.3.1 *Fehler in Eigenschaftssätzen*

Die dem generierten Eigenschaftssatz zu Grunde liegende Spezifikation kann Fehler enthalten. Die selben Fehler werden sich bei korrekter Extraktion aus dem Anforderungsdokument auch in dem Eigenschaftssatz für die Produktlinie wiederfinden. Ein Fehler kann eine Entwicklung entweder unmöglich machen, mit ungewollten Freiheitsgraden versehen oder erschweren. Das Anforderungsdokument kann dann nicht mehr als Konform mit IEEE 830 angesehen werden. Die in dieser Arbeit untersuchten Fehler sind in Abb. 30 dargestellt und werden im Folgenden genauer erklärt [MO11].

Widersprüche: Widersprüche in einem Anforderungsdokument führen da-

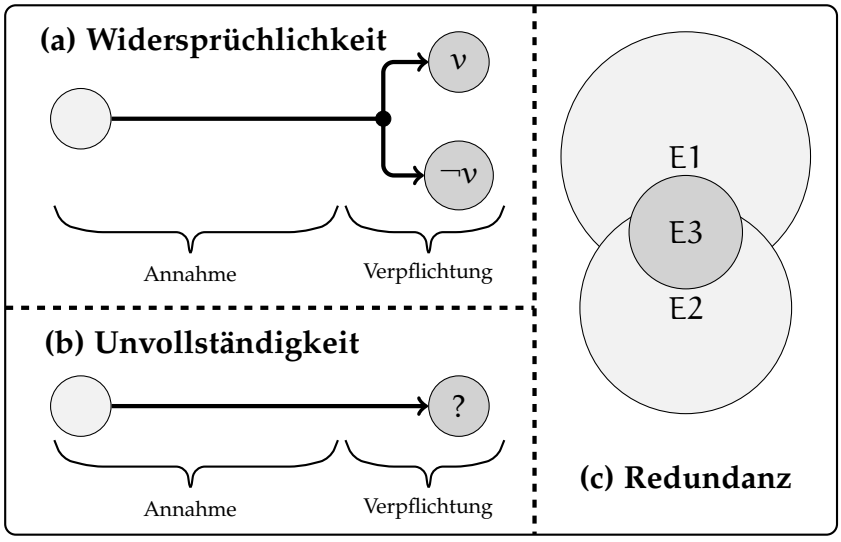


Abbildung 30: Fehler in Eigenschaftssätzen

zu, dass das beschriebene Systemverhalten nicht implementierbar ist. Ein Widerspruch besteht genau dann, wenn eine Implikation zwei mögliche widersprüchliche Nachfolgeereignisse definiert. Da in dem im Rahmen dieser Arbeit verwendeten Algorithmus zur Überprüfung von Anforderungen ausschließlich auf Bitebene gearbeitet wird, besteht ein Widerspruch genau dann, wenn eine Situation existiert, die einen Ausgang sowohl auf *'0'* als auch auf *'1'* setzt. Abb. 30 (a) zeigt einen Pfad durch einen Automaten, der in einem Widerspruch endet.

Unvollständigkeit: Anforderungen sind unvollständig, wenn ein oder mehrere Fälle existieren, in denen das Systemverhalten für bestimmte Pfade nicht definiert ist. Also genau dann, wenn der Ausgang in einer oder in mehreren Situationen nicht determiniert ist. Das führt zu Freiheitsgraden in der Implementierung, die durch den Verfasser der Anforderungen unerwünscht sein können. Abb. 30 (b) zeigt einen unvollständigen Pfad durch einen Automaten.

Redundanz: Einzelne Anforderungen sind Redundant, wenn ihr Inhalt durch eine oder mehrere andere Anforderungen im Anforderungsdokument abgedeckt ist. Sie führen dazu, dass der Eigenschaftssatz Eigenschaften enthält, deren Verhalten durch eine oder mehrere weitere Eigenschaften überdeckt wird. Abb. 30 (c) zeigt die durch drei Eigenschaften abgedeckte Funktionalität eines Systems. Die Funktionalität von *E3* ist vollständig in *E1* und *E2* enthalten. *E3* ist also redundant und kann weggelassen werden.

Die beschriebenen Fehler sind bedingt durch die Komplexität nicht intuitiv im Anforderungsdokument identifizierbar und korrigierbar. Formale Methoden, die bei ihrer Erkennung helfen, bieten sich daher an.

8.3.2 *Konzept*

Abb. 23 zeigt die Vorgehensweise zum Überprüfen von Eigenschaften. Als Grundlage dienen formale Eigenschaften für die Produktlinie. Für deren Erzeugung wird die Variabilität in den ursprünglichen Anforderungen identifiziert und als Feature-Modell bzw. als dessen logischer Ausdruck (5.2) modelliert. Weiterhin werden die Eigenschaften, die die Funktionalität der einzelnen Features beschreiben, extrahiert. Sodann kann der Eigenschaftssatz über die gesamte Produktlinie gebildet werden (7.1). Nachdem der Eigenschaftssatz für die Produktlinie erzeugt wurde, kann dieser formal überprüft werden. Dazu dient der in 6.2 präsentierte Algorithmus. Der Eigenschaftssatz wird nur mit Bezug auf sich selbst geprüft. Es existiert kein weiteres Modell, das korrektes oder vollständiges Verhalten definiert.

Zu Beginn werden die Widersprüche im Eigenschaftssatz erkannt und entfernt. Daraus resultiert ein implementierbares System. Im Beispiel aus 6.2.1 ist der Widerspruch nur von zwei Eingängen abhängig und wird innerhalb eines Zeitschritts sichtbar. Situationen, die zu Widersprüchen führen, können sich in komplexeren Anwendungen über mehrere Zeitschritte erstrecken und mehrere Eingänge involvieren. Des Weiteren werden ausschließlich Widersprüche von Eingängen aufgezeigt, die einzelne Bits repräsentieren. Sind diese Teil eines Bitvektors, der eine Zahl repräsentiert, wird das Auflösen von Widersprüchen weiter erschwert, da nicht alle enthaltenen Bits zu Widersprüchen führen müssen oder nur Bestimmte Werte des Bitvektors einen Widerspruch auslösen. Das Auflösen solcher Widersprüche gestaltet sich als schwierig und führt

häufig zu neuen Widersprüchen. Daher ist die Erkennung und Behebung von Widersprüchen in einer Spezifikation ein iterativer Prozess, der so lange ausgeführt wird, bis eine neue, vollständige Überprüfung keine weiteren Widersprüche findet. Meist können mehrere Widersprüche in einer Iteration behoben werden.

Im zweiten Schritt werden Lücken im Eigenschaftssatz erkannt und analysiert. Handelt es sich beim unvollständig beschriebenen Verhalten um einen ungewollten Freiheitsgrad, so werden die Eigenschaften angepasst. Ansonsten wird der Freiheitsgrad zur Vollständigkeitsbewertung hinzugefügt. Das Finden und Bewerten von Lücken gestaltet sich aus den selben Gründen, die auch bereits im Bereich der Analyse auf Widersprüche galten, als schwierig. Je nach Grad der Vollständigkeit der ursprünglichen Spezifikation, ist die Behebung von Lücken mit großem Aufwand verbunden.

Enthält der Eigenschaftssatz keine ungewollten Freiheitsgrade mehr, so wird er auf Redundanz geprüft. Redundante Eigenschaften werden entfernt. Das vorgeschlagene Schema findet ausschließlich vollständig redundante Eigenschaften in der Spezifikation. Eigenschaften, die nur zum Teil von einer anderen überlagert werden, können nicht entdeckt werden. Hierfür müsste der überlagerte Teil genau identifiziert und dargestellt werden. Die Eigenschaften und damit die Spezifikation müssten so verändert werden, das die entsprechende Funktionalität nicht redundant in ihnen beschrieben wird. Nachdem die redundanten Eigenschaften entfernt wurden, werden die im Eigenschaftssatz gefundenen und behobenen Fehler in den ursprünglichen Anforderungen korrigiert. So entsteht eine vollständige und konsistente Spezifikation, die frei von Redundanz ist.

Die präsentierte Vorgehensweise hat mehrere Vorteile:

- Es ist kein weiteres Modell zur Beschreibung von vollständigem und widerspruchsfreiem Verhalten nötig.
- Die Ereignisfolgen, die Lücken oder Widersprüche erzeugen, werden ausgegeben und können bewertet werden.
- Der formale Eigenschaftssatz kann weiterverwendet werden.

Der Hauptnachteil des Ansatzes liegt im initialen Aufwand für die Modellierung der Anforderungen durch ITL. Weiterhin muss für Anforderungen, die Produktlinien beschreiben, das Feature-Modell aufgestellt werden und die einzelnen Features auf die Anforderungen bzw. die daraus abgeleiteten Eigenschaften abgebildet werden.

8.4 ORAKELGENERIERUNG

Ein Orakel (2.3.2) ist das ausführbare Modell eines Systems, das aus den Anforderungen für ein Produkt abgeleitet ist. Seine Generierung ermöglicht es, Voraussagen über das Systemverhalten treffen zu können. Es wird mit den selben Eingangswerten wie das System stimuliert und liefert Ausgaben, die das korrekte Systemverhalten widerspiegeln. So kann ein Testingenieur intuitiv die Korrektheit des Sytem-Under-Test (SUT) beurteilen. Der aufwendige Abgleich des Systemverhaltens mit der Spezifikation entfällt und Abweichungen vom spezifizierten Verhalten können einfach erkannt werden.

8.4.1 *Konzept*

Die Orakelgenerierung aus formalen Eigenschaften wird durch den in 6.1 präsentierten Ansatz ermöglicht [MO10]. Der Schritt der Orakelgenerierung folgt, wie in Abb. 23 dargestellt, der Behebung von Fehlern in Eigenschaftssätzen, nach der ein widerspruchsfreies und vollständiges Anforderungsdokument entsteht, das frei von redundanter Beschreibung ist. Dieses bzw. die aus ihm abgeleiteten Produktlinieneigenschaften dienen als Vorlage zur Generierung eines Orakels auf Basis eines Cando-Objekts. Das Orakel kann auf mehrere Arten verwendet werden. Es können Testfälle, die auf dem entwickelten System ausgeführt werden sollen, im Vorfeld auf dem Orakel simuliert werden. Ein korrekt implementiertes System muss sich wie das Orakel verhalten. Somit muss das Ausgangsverhalten des Orakels und des implementierten Systems äquivalent sein. Dieser Äquivalenzvergleich kann entweder manuell oder automatisiert durchgeführt werden. Weiterhin können, da es sich beim erzeugten Orakel um eine VHDL-Beschreibung handelt, Metriken (2.2.4) bei der Simulation des Orakels Anwendung finden, um eine Abschätzung über die Testabdeckung zu erhalten. Des Weiteren können Systemannahmen, wie in 8.5.1 beschrieben, auf dem Orakel bewiesen werden. Die Ergebnisse der Beweise und des Äquivalenzvergleichs können zum Systemtest oder zur Verbesserung des Anforderungsdokuments verwendet werden.

Abb. 31 zeigt, wie sich die Orakelgenerierung in den Produktlinienkontext einfügt. Es wird aus den Anforderungen auf Domänenebene erzeugt und dient dann als Orakel für alle Produkte auf Anwendungsebene. Die übersetzten Anforderungen haben in ITL die in Abb. 32 gezeigte Struktur. Der Eigenschaftssatz in ITL besteht aus drei Hauptkomponenten. Die Eigenschaft *FEATURE_MODEL* gibt an, dass der boolesche Ausdruck für das Feature-Modells immer wahr sein muss. Es werden also nur Produkte beschrieben, die als gültige Konfiguration aus den Anforderungen ableitbar sind. Durch die Eigenschaft *INPUTS_RANGE* wird der gültige Eingabebereich festgelegt. Der

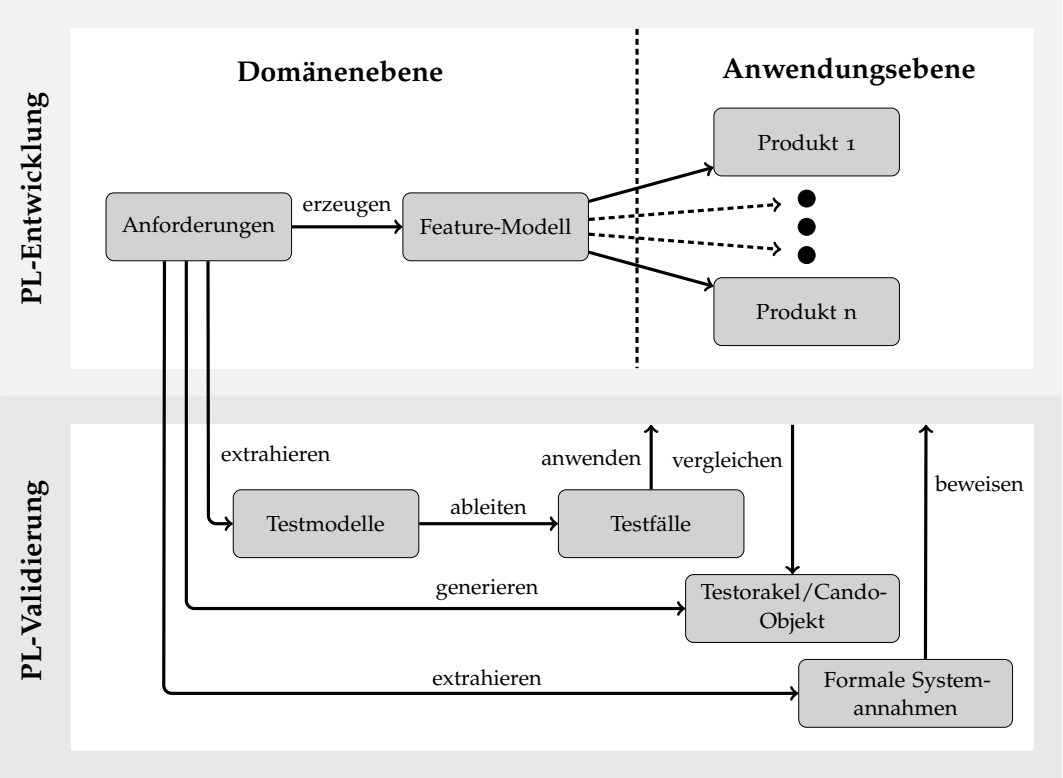


Abbildung 31: Orakel im Produktlinienkontext [MO10]

```
property FEATURE_MODEL is
assume:
  -- true
prove:
  at t+1: if(
   fm = 1) then
     validfm = '1';
  else
     validfm = '0';
  end if;
end property;

property BEHAVIOR_N is
assume:
  at t:
  [relevant_combination] = 1
  [specified assumptions]
prove:
  at t:
  [specified behavior]
end property;
```

```
property INPUTS_RANGE is
assume:
  -- true
prove:
  at t+1: if PREV(
     [input1_out_of_range] or
     [input2_out_of_range] or
     ...
     [inputn_out_of_range]
     ) then
        invalid_inputs = '1';
     else
        invalid_inputs = '0';
     end if;
end property;
```

Abbildung 32: Struktur der Eigenschaften in ITL

Ausgang *invalid_inputs* wird immer dann gesetzt, wenn ein Eingabewert angelegt wird, der sich nicht im spezifizierten Bereich bewegt. Diese Eigenschaft ist nötig, da sich ein Cando-Objekt für nicht spezifizierte Eingabewerte völlig zufällig verhalten darf. Ein nicht definierter Eingabewert muss also erkannt und angezeigt werden. Die Eigenschaft *BEHAVIOR_N* steht beispielhaft für eine Eigenschaft, die das Verhalten des Systems beschreibt. In ihr wird die Annahme durch die Feature-Kombination erweitert, die ausgewählt sein muss, damit die Eigenschaft für das ausgewählte Produkt gilt. Die einzelnen Features werden so auf die jeweiligen Eigenschaften abgebildet. Weiterhin muss das eigentliche Verhalten in der Eigenschaft definiert werden. Das geschieht durch das Hinzufügen geeigneter Annahme- und Verpflichtungsteile. Ein Eigenschaftssatz enthält im Normalfall mehrere verhaltensbeschreibende Eigenschaften. Die Vorteile der Orakelgenerierung lassen sich wie folgt zusammenfassen:

- Es entsteht ein ausführbares Systemmodell.
- Die generierte Hardwarebeschreibung kann mit gängigen Werkzeugen simuliert werden.
- Die Generierung erfolgt vollkommen automatisiert aus dem Eigenschaftssatz.

8.4.2 *Struktur des generierten Orakels*

Die auf einem Orakel ausgeführten Testfälle können mit den Ausgaben der jeweiligen Produkte verglichen werden. Folglich hat ein Cando-Objekt als Orakel die Aufgabe der Vorhersage von Testergebnissen. Da es sich bei einem Cando-Objekt um eine simulierbare VHDL-Beschreibung handelt, können Testfälle auf Orakeln aus Cando-Objekten simuliert werden. Für die Ableitung der Testfälle kann für einzelne Produkte ein Klassifikationsbaum (2.3.3) oder für Produktlinien ein Feature-Model for Testing (FMT) (8.6) verwendet werden. Abb. 33 zeigt die Ein- und Ausgänge des generierten Cando-Objekts. Intern

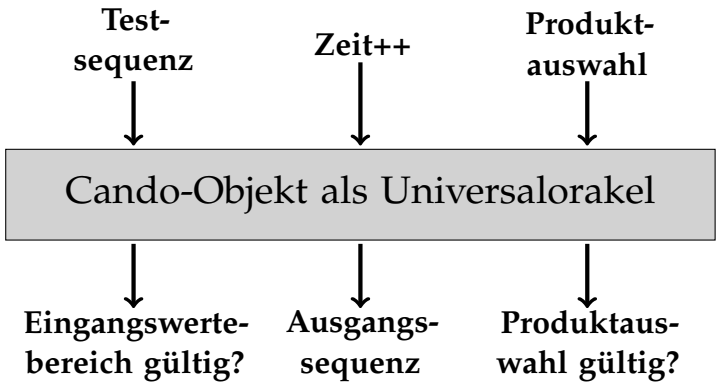


Abbildung 33: Struktur des generierten Orakels [MO10]

besteht es, wie in Abb. 26 gezeigt, aus nebenläufigen Prozessen und Kanälen, die durch ITL-Eigenschaften definiert wurden. Soll ein Testfall ausgeführt werden, so muss zuerst die Produktkonfiguration gewählt werden, die getestet werden soll. Das geschieht über Eingänge, die die auswählbaren Features im Feature-Modell betreffen. Die getroffene Auswahl stellt während der Simulation eine Invariante dar. Sie wird also zu Beginn festgelegt und dann nicht mehr verändert. Die einzige Ausnahme bilden Features, die erst während der Laufzeit gebunden werden (8.6). Diese können sich während der Ausführung ändern.

Systeme werden im Rahmen dieses Ansatzes generell als ereignisbasiert angesehen. Die Ereignisdefinition richtet sich dabei nach der Definition von Hoare in CSPs (3.1). Im Unterschied zu Hoare dürfen allerdings in einem Zeitschritt mehrere Ereignisse auftreten. Eine Ereignisfolge ist folglich nicht durch eine Folge von Ereignissen an den Ein- oder Ausgängen zu diskreten Zeitpunkten bestimmt. Vielmehr existiert für jeden Ein- und Ausgang eine separate Ereignisfolge. In jedem Zeitschritt kann jeder Ein- und Ausgang einen neuen Wert annehmen bzw. seinen alten Wert beibehalten. Am Eingang des Orakels liegen die Werte aus der Testsequenz an, die im nächsten Zeitschritt gelten müssen. Der Fortlauf der Zeit wird durch *Zeit*++ analog zum Systemtakt eines Hardwarebausteins erreicht. Hierdurch wird die Diskretisierung der Zeit erreicht. Am Ausgang des Orakels wird angezeigt, ob das gewählte Produkt nach dem Feature-Modell gültig ist. Weiterhin wird angegeben, ob sich die Eingangswerte in einem gültigen Bereich befinden. Der gültige Bereich wird durch die Anforderungen vorgegeben. Er kann sich z.B. auf die maximale und minimale Geschwindigkeit eines Fahrzeugs beziehen. Weiterhin wird die Ausgangssequenz in Abhängigkeit von der Testsequenz und des gewählten Produkts ausgegeben.

8.5 BEWEIS VON SYSTEMANNAHMEN

Eine *Systemannahme* beschreibt eine Eigenschaft, die implizit durch den zu Grunde liegenden Eigenschaftssatz erfüllt sein muss. Es handelt sich um Sicherheitseigenschaften (3.3.3) oder zeitlich begrenzte Lebendigkeitseigenschaften (3.3.3). Für den Beweis von Systemannahmen wird ein Modell benötigt, auf dem der Beweis durchgeführt werden kann sowie die zu beweisenden Systemannahmen in formaler Form. Als Modell dient im Rahmen dieser Arbeit das erzeugte Orakel oder der Eigenschaftssatz selbst. Auf diesen können die Systemannahmen bewiesen werden.

8.5.1 *Beweis auf Oraklen*

Der Beweis von Systemannahmen auf einem Orakel zielt darauf ab, Spezifikationen auf ihre Sicherheit zu überprüfen. Abb. 34 (a) zeigt schematisch das Vorgehen beim Beweis von Systemannahmen auf einem Orakel. Die Extraktion von Systemannahmen aus Spezifikationen muss manuell geschehen. Sie können entweder als Annahmen über die Sicherheit in der Spezifikation explizit enthalten sein oder im Nachhinein durch Projektbeteiligte formuliert werden. Eine Formalisierung der Systemannahmen in ITL oder LTL bildet die Voraussetzung für ihren Beweis auf dem generierten Orakel. Der eigentliche Beweis geschieht durch kommerzielle Werkzeuge wie *OneSpin 360™ MV* (9.1.5). Eine Systemannahme, die auf dem Orakel hält ist nach der Definition von Redundanz aus 8.3.1 immer zu den ursprünglichen Eigenschaften redundant.

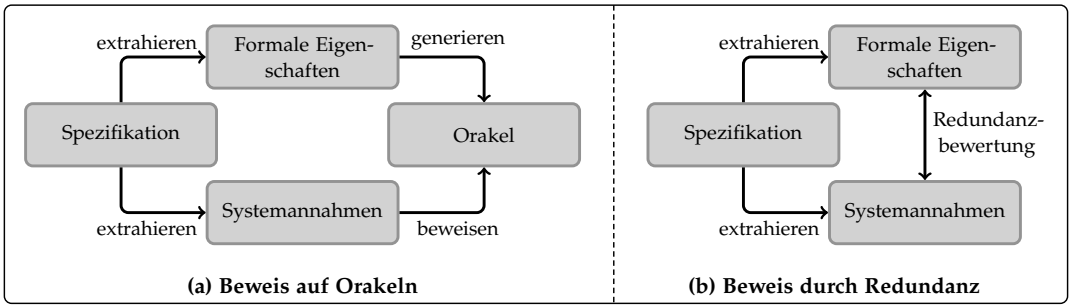


Abbildung 34: Beweis von Systemannahmen

8.5.2 *Beweis durch Redundanz*

Wird aus Zeit- oder Komplexitätsgründen kein Orakel generiert oder steht kein Werkzeug für den Beweis der Systemannahmen zur Verfügung, so können Eigenschaften, die Systemannahmen beschreiben, trotzdem bewiesen werden. Dafür wird der Eigenschaftssatz zuerst auf Vollständigkeit und Widerspruchsfreiheit überprüft. X beschreibt die Menge der gefundenen Lücken.

$\mathfrak{K}(\nu_{\mathfrak{UE}}) \equiv 0$
$\mathfrak{L}(\nu_{\mathfrak{UE}}) \equiv X$

Sodann wird der Eigenschaftssatz durch eine oder mehrere Eigenschaften über Systemannahmen erweitert. Im Anschluss wird der entstehende Eigenschaftssatz auf Vollständigkeit und Widerspruchsfreiheit überprüft. Ändert sich die Anzahl und Art der Lücken im Vergleich zum Eigenschaftssatz ohne die Systemannahmen nicht und werden weiterhin keine Widersprüche gefunden, so sind die Systemannahmen redundante Eigenschaften.

$\mathfrak{K}(\nu_{\mathfrak{UE}\wedge\mathfrak{IG}}) \equiv 0$
$\mathfrak{L}(\nu_{\mathfrak{UE}\wedge\mathfrak{IG}}) \equiv X$

Sie sind folglich bereits durch den ursprünglichen Eigenschaftssatz beschrieben worden und müssen somit auf diesem halten. Das Vorgehen ist schematisch in Abb. 34 (b) dargestellt.

Relevant wird der Beweis von Systemannahmen vor allem dann, wenn es um die Sicherheit von Systemen geht. ISO 26262 [Int11] definiert Sicherheit als die Abwesenheit inakzeptabler Risiken. Es muss gezeigt werden, dass eine als sicher geltende Architektur eines Systems bestimmten Sicherheitsanforderungen genügt. Grundvoraussetzung dafür ist, dass auch die Anforderungen für das System den selben Sicherheitsanforderungen genügen. Ein Beispiel für eine Sicherheitsanforderung ist ein Autositz, der ab einer Geschwindigkeit von 15 km/h nicht mehr elektrisch bewegt werden darf. Die Anforderungen für die Kontrolleinheit der Autositzsteuerung müssen diese Sicherheitsanforderung abbilden.

8.6 INTEGRIERTER TESTANSATZ

Für die Modellierung von Testfällen im Kontext von Produktlinien wurde das FMT [OMS09, SOM10] vorgeschlagen. Ein FMT enthält die erforderlichen Informationen, um einen Klassifikationsbaum für jedes mögliche Produkt einer Produktlinie automatisiert zu erstellen. Dabei wird der Feature-Modell-Ansatz mit dem Klassifikationsbaum (2.3.3) kombiniert. Es wird ausgenutzt, dass Feature-Modelle Variabilität darstellen können und so optionale Parameter eines Systems strukturieren und Klassifikationsbäume eine ideale Repräsentation für Äquivalenzklassen im Black-Box-Test bilden. Neu im FMT ist die optionale Angabe der *Binding Time*.

> **Definition 8.1: *Binding Time* -** Der Zeitpunkt im Produktgenerierungsprozess, an dem ein Feature ausgewählt bzw. in das Produkt integriert wird.

Ein Feature kann während der Planung des Produkts, während des Designs, beim Kompilieren, beim Installieren, beim Startprozess oder erst während der Laufzeit gebunden werden.

Darüber hinaus wird die Kardinalität bei Oder- und Alternativgruppen angegeben. Abhängigkeiten zwischen den Features können ebenso definiert werden wie der Feature-Typ oder zum Feature zugehörige Attribute. Die vom Klassifikationsbaum bekannte Tabelle zum Aufstellen der Testfälle wird auch durch das FMT übernommen.

Wird ein Produkt im FMT ausgewählt, so werden alle Features eliminiert, die nicht Teil des Produkts sind. Dadurch verschwinden auch die jeweiligen Spalten in der Testtabelle des FMTs. Übrig bleiben die Testfälle für das ausgewählte Produkt. Durch die Eliminierung von Spalten kann es vorkommen, dass zwei oder mehr gleiche Testfälle übrig bleiben. Diese müssen eliminiert werden. Abb. 35 verdeutlicht, wie sich das FMT in die Produktlinienentwicklung ein-

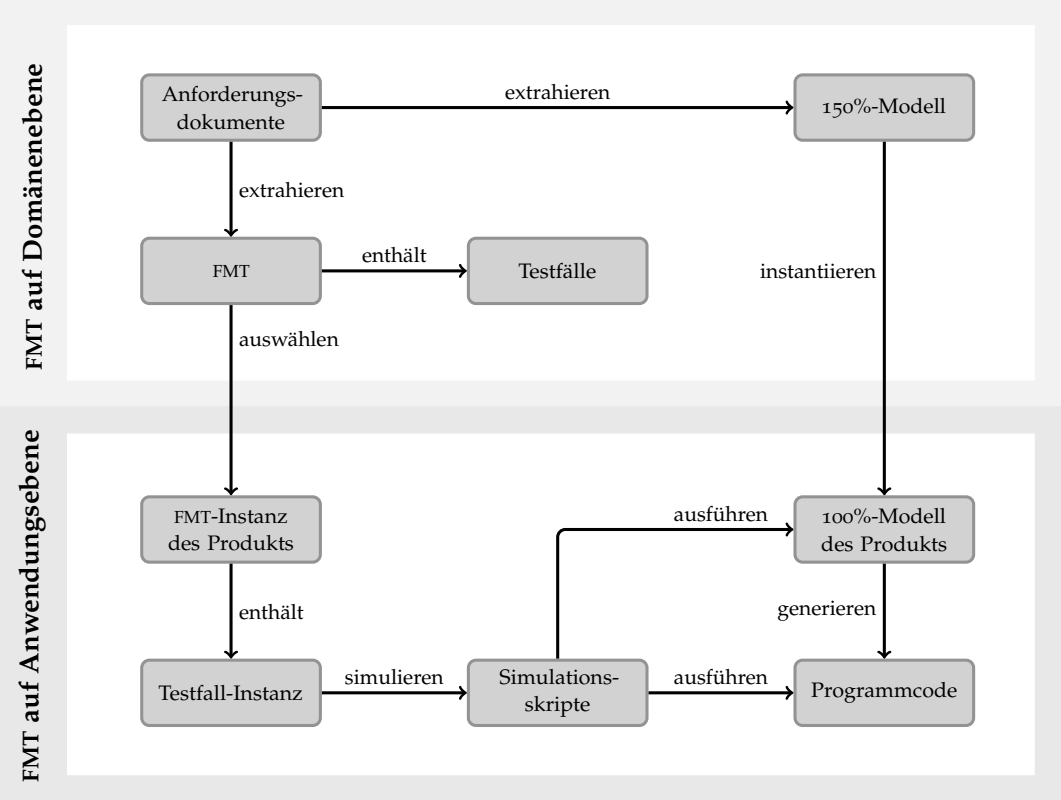


Abbildung 35: FMT-Ansatz [OMS09, SOM10]

gliedert. Das FMT wird auf Domänenebene zusammen mit dem 150%-Modell aus den Anforderungen extrahiert und kann das Feature-Modell entweder ersetzen oder neben ihm existieren. Alle Testfälle sind in der Tabelle des FMTs enthalten. Auf Anwendungsebene werden Instanzen des FMTs und des 150%-Modells gebildet. Das Aufstellen des 150%-Modells sowie die Bildung des 100%-Modells orientieren sich an der in 5.1 präsentierten Methodik. Aus dem FMT entsteht eine Struktur, die vergleichbar zum Klassifikationsbaum ist und in der die Testfälle für das durch das 100%-Modell repräsentierte Produkt enthalten sind. Diese Testfälle dienen als Vorlage zur Generierung von Testskripten, die automatisiert auf dem Programmcode ausgeführt werden können bzw. die zur Simulation des 100%-Modells dienen (9.2.5).

8.7 GRENZEN DES ANSATZES

Der Ansatz ist durch die Komplexität der Zerlegung der Eigenschaften bei der Vollständigkeitsbewertung (6.2) und bei der Orakelgenerierung (6.1) begrenzt. Die Rechenzeit und der Speicherbedarf können bei großen Bitbreiten, auf denen komplexe Operationen durchgeführt werden, exponentiell wachsen. Abhilfe schafft bis zu einem gewissen Grad eine geeignete Zerlegung der Eigenschaftssätze. Eine solche Zerlegung ist zeitaufwendig und führt nicht in allen Fällen zum Ziel. Die Zerlegung beschränkt sich im Normalfall ausschließlich auf die Zerlegung von Bitvektoren. Zeitfenster müssen nicht zerlegt werden, da im Rahmen dieser Arbeit stets Implikationen von einem auf den anderen Zeitschritt durch die Eigenschaften beschrieben werden und größere Zeitfenster nicht auftreten.

Weiterhin ist die Zahlendarstellung auf Integerzahlen beschränkt. Fließkommawerte können nicht repräsentiert werden. Die Multiplikations- und Divisionsoperation sind, da eine Zerlegung in einzelne Bits stattfindet, nur schwer darstellbar.

Die Anforderungen, aus denen die Eigenschaften abgeleitet werden sollen, müssen durch ITL repräsentierbar sein. Das wird vor allem dann zum Problem, wenn die Anforderungen in einer anderen formalen Sprache, wie z.B. CSP, vorliegen, deren Aussagekraft nicht vollständig durch ITL abgedeckt ist.

8.8 VERWANDTE ARBEITEN

Die vorgestellten Konzepte kombiniert Ansätze aus unterschiedlichen Gebieten. Auf den jeweiligen Gebieten existieren Arbeiten, die mit den hier vorgestellten Ansätzen in Bezug stehen bzw. die ergänzend angewendet werden können. Im Folgenden erfolgt eine kritische Auseinandersetzung mit wesentlichen in Bezug stehenden Arbeiten.

In [GR03] wird auf die grundlegende Formulierung von Anforderungen in Textform eingegangen. Diese werden vor allem unter psychologischen Gesichtspunkten bewertet. Weiterhin werden Regeln aufgestellt, die beim Formulieren von Anforderungen als Fließtext zu beachten sind. Dazu gehören u.a.:

- Anforderungen müssen im Aktiv formuliert sein.
- Systemannahmen müssen spezifiziert werden.
- Alle Universalquantoren (immer, jede, nie) müssen überprüft werden.
- Substantive müssen definiert werden.
- Der Text muss klar, kurz ohne unnötige Wiederholungen formuliert sein.

- Generalisierungen im Text sollten vermieden werden, da die Details durch das Wissen des Lesers ausgefüllt werden und dadurch verfälscht werden können.

Solche Anforderungsdokumente helfen bei der Extraktion von formalen Eigenschaften immens. Fehlinterpretationen werden nicht nur beim Formulieren von Eigenschaften, sondern auch bei der Entwicklung und dem Testprozess vermieden. Durch eine Überprüfung von bestehenden Textdokumenten auf Basis der formulierten Regeln können die Anzahl der Widersprüche und der Grad der Unvollständigkeit schon vor der formalen Überprüfung minimiert werden.

In [KAP08] wird auf die Qualität von Spezifikationen eingegangen, die zum Testen eines Systems verwendet werden sollen. Die Verwendung einer funktionalen Sprache wird vorgeschlagen, um eine eindeutige und möglichst kompakte Spezifikation mit klarer Semantik zu erhalten. Hierfür wird die Modellierungssprache *Clean* verwendet. Analog zu Cando-Objekten wird auch in dieser Arbeit in nicht spezifizierten Fällen ein zufälliges Verhalten erlaubt. Nicht-deterministische Anforderungen werden generell in deterministische übersetzt, indem ein möglicher Folgezustand immer erreicht wird. Das System wird als Black-Box modelliert und nur sein äußeres Verhalten (4.2.1) ist sichtbar. Der in der Arbeit beschriebene Ansatz überführt eine Spezifikation in einen Automaten. Determinismus, Vollständigkeit und Erreichbarkeit werden mit simulativen Ansätzen gezeigt. Dadurch steigt die Komplexität im Gegensatz zu formalen Ansätzen.

In [HJL96] wird auf die Überprüfung von Anforderungen in SCR-Notation (4.2.2) mit Hinblick auf Determinismus und Vollständigkeit eingegangen. Hierfür werden die SCR-Anforderungen formal analysiert. Widerspruchsfreiheit wird gezeigt, indem die Bedingungen jeweils paarweise und-verknüpft werden. Der daraus entstehende Ausdruck muss gleich '0' sein. Vollständigkeit ist dann gegeben, wenn die oder-Verknüpfung aller Bedingungen eine Tautologie bildet. Die Definitionen von Konsistenz und Vollständigkeit decken sich also mit der Definition aus 6.2. [HJL96] ist dazu in der Lage, Fehler in *Mode Transition Tables* aufzudecken, die zu einem nicht-deterministischen Verhalten führen. Die Fehler liegen in der Struktur der *Mode Transistion Table* begründet. In ihr werden Bedingungen definiert, die zum Wechseln eines Zustands führen. Diese werden zeilenweise aufgeschrieben. Dabei darf kein Fall möglich sein, in dem zwei Zeilen gleichzeitig wahr werden. Diese Fehler werden sowohl durch Candogen (9.1.2) als auch durch Properlyze (9.1.3) als Widersprüche aufgedeckt, da für die Zustandsvariable bei gleicher Eingangskombination mehrere Zuweisungen möglich sind.

In [Fau01] wird die Anwendung von Produktlinien in Zusammenhang mit SCR und CoRE vorgeschlagen. Die Anforderungen werden zu diesem Zweck

in die *Product-Line Requirements Specification*, die den Anforderungen auf Domänensebene entspricht, und in die *Software Requirements Specification*, die den Anforderungen auf Anwendungsebene entspricht, unterteilt. Das ursprüngliche Anforderungsdokument wird in verpflichtende und optionale Features unterteilt. Weiterhin wird ein Entscheidungsbaum hinterlegt, der alle mögliche Konfigurationen enthält. Wird eine Konfiguration ausgewählt, so wird automatisch eine Spezifikation in CoRE-Notation für das jeweilige Produkt erzeugt. Durch diesen Ansatz wird die Verwaltung von Anforderungen für Produktlinien in SCR oder CoRE-Notation unterstützt. Die Hinterlegung eines graphischen Variabilitätsmodells würde allerdings bei der Strukturierung komplexer Spezifikationen helfen.

In [LP08] wird ein Ansatz beschrieben, der die Erkennung von Widersprüchen in Anforderungen auf der Domänenebene ermöglicht. Ein Widerspruch besteht dann, wenn die Anforderungen abgeleiteten Invarianten widersprechen. Eine Invariante ist eine globale implizite Eigenschaft, die durch die Anforderungen repräsentiert sein muss. Im Kontext dieser Arbeit wird dafür der Begriff Systemannahme verwendet. Um Anforderungen auf Domänenebene zu überprüfen, muss, analog zu 7.3.2, eine Verbindung zwischen den Artefakten der Anforderungen und einem Variabilitätsmodell hergestellt werden. Hierfür werden die Anforderungen in einen Automaten überführt und das Variabilitätsmodell auf diesen abgebildet. Die Invarianten werden sodann anhand zeitlich logischer Ausdrücke auf dem Automaten bewiesen. Hierdurch werden, wie auch beim Beweis von Systemannahmen auf Cando-Objekten 8.3.2, Inkonsistenzen aufgedeckt. Allerdings nur die, für die eine Invariante formuliert wurde. Die Vollständigkeit wird bei diesem Ansatz vernachlässigt. Außerdem muss manuell ein Automat erzeugt werden, was für komplexere Systeme nicht praktikabel ist.

In [LGB08] wird ein *goal* orientierter Ansatz im Zusammenhang von Anforderungen und Produktlinien vorgeschlagen. Ein *goal* beschreibt ein Ziel der Entwicklung und somit die Intention der Projektbeteiligten. Es existieren *hard goals*, die eine funktionale Anforderung beschreiben und *soft goals*, die nichtfunktionale Anforderungen darstellen. Zur Modellierung der aus den Anforderungen extrahierten Struktur und der Funktionalität wird die Verwendung eines plattformunabhängigen Feature-Modells vorgeschlagen dessen einzelne Features auf die jeweiligen *goals* abgebildet werden. Dies deckt sich mit der in dieser Arbeit präsentierten Vorgehensweise. Weiterhin wird vorgeschlagen, ein zweites plattformabhängiges Feature-Modell zur Modellierung der Variabilität in Soft- und Hardware zu verwenden. Dieses wird auf die einzelnen Bestandteile der Hardware und auf die in UML definierten Klassen, Methoden und Variablen der Software abgebildet. Der Aufwand hierfür ist schon bei kleineren Projekten immens. Des Weiteren müssen plattformunabhängiges und plattformabhängiges Feature-Modell bei Änderung der Anforderungen konsistent gehalten werden.

In [vdM07] wird auf die Modellierung von Variabilität in Anforderungen eingegangen. Es wird ein Metamodell für die Variabilitätsmodellierung eingeführt und verschiedene bestehende Konzepte eingeführt. Weiterhin wird das Plattform-Feature-Modell vorgeschlagen, das zur Strukturierung auf Domänenebene verwendet wird und in den Anforderungen enthalten sein soll. Dieses ist syntaktisch und semantisch äquivalent zum Feature-Modell aus dieser Arbeit. Die Strukturierung einer Produktlinie bereits in den Anforderungen durch ein Feature-Modell unterstützt die weitere Entwicklung sowie die Anforderungsanalyse und hilft auch im Kontext dieser Arbeit. Für das Plattform-Feature-Modell werden folgende Qualitätskriterien festgelegt:

- **Adäquatheit:** Das Plattform-Feature-Modell muss die Produktlinie mit Bezug auf Granularität und Relevanz angemessen darstellen.
- **Vollständigkeit:** Vollständig ist das Plattform-Feature-Modell, wenn alle Features bis zur gewünschten Deatailstufe vollständig beschrieben sind.
- **Widerspruchsfreiheit:** Das Plattform-Feature-Modell ist widerspruchsfrei, wenn zwischen zwei Features kein Widerspruch besteht. Ein Widerspruch ist z.B. dann gegeben, wenn zwei Pflichtknoten sich durch eine Ausschlusskante zwischen ihnen gegenseitig ausschließen.
- **Redundanz:** Redundanz ist dann gegeben, wenn Features mit gleichem Namen oder gleichen Attributnamen bestehen oder dann, wenn im Plattform-Feature-Modell Beziehungen bestehen, die zu anderen Beziehungen redundant sein. Das ist z.B. dann der Fall, wenn von einem optionalen Feature eine Bedingungskante auf ein verpflichtendes Feature besteht.

Die weiteren Punkte aus IEEE 830 (4.1) sind ebenfalls Qualitätskriterien für das Plattform-Feature-Modell. Ein konsistentes und nicht-redundantes Feature-Modell wird erzeugt. Die Konsistenzprüfung von Feature-Modellen ist über ein Satisfiability (SAT)-Solver erreichbar. Findet dieser keine Lösung für den booleschen Ausdruck des Feature-Modells, enthält es Inkonsistenzen. Der Algorithmus aus [vdM07] geht ausschließlich auf die Struktur des Feature-Modells ein und vernachlässigt die dahinter liegende Funktionalität.

9 EXPERIMENTELLE UNTERSUCHUNGEN

Die vorgestellten Konzepte werden im Folgenden an Hand von Beispielen erprobt. Die Vorlagen dafür stammen aus eigenen und fremden Veröffentlichungen, aus offiziellen Spezifikationen und aus der Industrie. Neben der Behandlung von Variabilität zielen die Experimente auch auf die Erprobung des Ansatzes für einzelne Produkte ab. Weiterhin werden die Konzepte zur Überführung von verschiedenartigen Spezifikationen in formale Eigenschaften verdeutlicht.

9.1 VERWENDETE WERKZEUGE

Bei den experimentellen Untersuchungen werden verschiedene Werkzeuge eingesetzt. Diese sind entweder kommerziell verfügbar oder wurden problemspezifisch entwickelt.

9.1.1 *Feature Model Editor*

Der Feature Model Editor ist ein für Eclipse entwickeltes Plugin. Er ermöglicht das graphische Zeichnen und Editieren von Feature-Modellen. Weiterhin kann der boolsche Ausdruck, der alle möglichen Produkte der durch das Feature-Modell beschriebenen Produktlinie repräsentiert, in KNF ausgegeben werden. Diese Ausgabe ist in Textform oder in DIMACS-Form [Rut93] für MiniSAT möglich. Außerdem kann die Anzahl der aus einem Feature-Modell generierbaren Produkte errechnet werden.

9.1.2 *Candogen*

Bei *Candogen* handelt es sich um ein Linux-Kommandozeilenwerkzeug, das den Algorithmus zur Generierung von Cando-Objekten (6.1) implementiert. Es erwartet als Eingabe einen Satz von Eigenschaften und eine Beschreibung der Ein- und Ausgänge des Moduls. Als Ausgabe liefert es eine VHDL-Beschreibung des Moduls. Weiterhin kann die Laufzeit der Generierung ausgegeben werden.

9.1.3 *Properlyze*

Die Vollständigkeitsbewertung (6.2) wird mit dem Werkzeug *Properlyze* durchgeführt. Auch hier bilden ein Eigenschaftssatz und die Modulein- und Ausgänge die Werkzeugeingaben. Ausgegeben werden Widersprüche und Spezifikationslücken sowohl als Eigenschaften als auch als Beschreibungen, die als Signalfolge graphisch dargestellt werden können. Die Laufzeit des Programms kann detailliert ausgegeben werden.

9.1.4 *ModelSim*

ModelSim von *Mentor Graphics* ist ein kommerzielles Simulationswerkzeug für VHDL-Beschreibungen. Es bietet die Möglichkeit, Vollständigkeitsmetriken bei der Simulation zu berücksichtigen. Als Eingabe dienen VHDL- oder Verilog-Beschreibungen. Die Testfälle werden über Stimuli definiert und können entweder manuell eingeben werden oder automatisiert über ein Skript erzeugt bzw. abgearbeitet werden.

9.1.5 *OneSpin 360™ MV*

Die formale Verifikation von Eigenschaften auf erzeugten Cando-Objekten wird mit *OneSpin 360™ MV*, einem Werkzeug der *OneSpin Solutions GmbH*, durchgeführt. Als Eingabe dienen sowohl die VHDL-Beschreibung als auch ein Eigenschaftssatz in ITL. Die im Eigenschaftssatz enthaltenen einzelnen Eigenschaften werden nacheinander formal auf der VHDL-Beschreibung bewiesen. Schlägt der Beweis fehl, so wird ein Gegenbeispiel als Signalfolge ausgegeben.

9.2 AUTOSITZSTEUERUNG

Die vorgestellten Konzepte werden an der Steuerungseinheit eines Autositzes evaluiert.

9.2.1 *Spezifikation*

Die Spezifikation des Autositzes liegt als reiner Text vor und seine Variabilität wird durch das Feature-Modell aus Abb. 36 beschrieben. Aus dem Feature-Modell ergeben sich insgesamt 18 mögliche Produkte. Die Spezifikation des Autositzes hat folgende Eckpunkte:

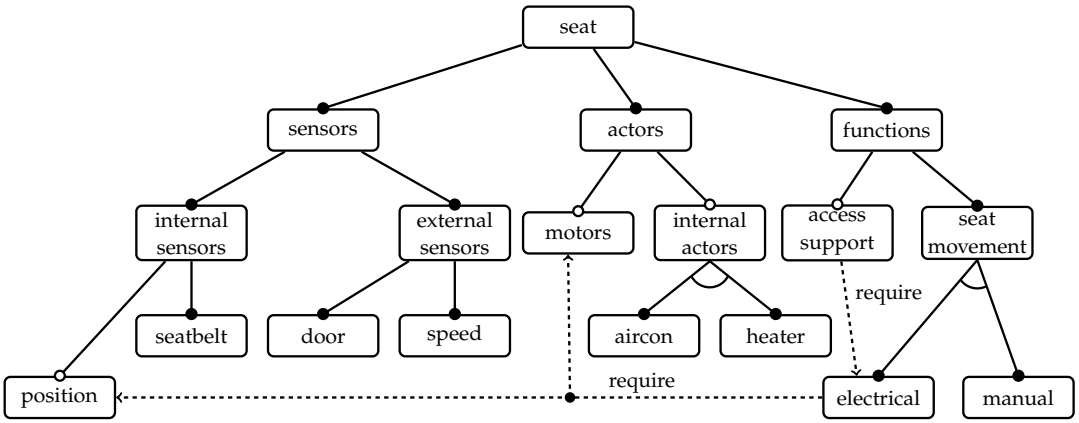


Abbildung 36: Feature-Modell der Autositzfunktionen [SOM10]

1. Der Sitz verfügt über drei Motoren für die Bewegungsrichtung hoch/runter, links/rechts und für die Veränderung des Winkels der Rückenlehne.
2. Jeder Motor wird durch einen Schalter gesteuert, der die Positionen *vor*, *stopp* und *zurück* einnehmen kann.
3. Eine Sitzbewegung ist nur bis zu einer Geschwindigkeit von 15 km/h möglich.
4. Wahlweise können entweder eine Sitzheizung oder eine Klimaanlage im Sitz integriert sein, die über einen Schalter mit den Positionen *aus*, *medium* und *hoch* gesteuert werden.
5. Sitzheizung bzw. Klimaanlage dürfen nur aktiv sein, wenn der Sitzgurt angelegt ist.
6. Da alle Schalter über das selbe Bussystem mit den jeweiligen Aktoren kommunizieren, kann immer nur ein Aktor aktiv sein.
7. Der Sitz kann über eine automatische Einstiegsfunktion verfügen.
8. Wenn die Tür geöffnet wird, dann fährt der Sitz komplett zurück und der Fahrer kann einsteigen.
9. Wenn die Türe schließt, dann fährt der Sitz zurück zur Ausgangsposition.

Diese textuelle Beschreibung muss in formale Eigenschaften übersetzt werden, die auf die einzelnen Features abgebildet werden.

9.2.2 *Extraktion der Eigenschaften*

Der Prozess der Eigenschaftsextraktion im Folgenden anhand der Eigenschaft für die Sitzheizung verdeutlicht.

```
property Sitzheizung is
assume:
  -- true
prove:
  at t+1:
    if PREV(SeatBelt = 1 and f_heater = '1') then
      SHeat_output = PREV(SHeat_request);
    else
      SHeat_output = 0;
    end if;
end property;
```

Die Sitzheizung darf nur aktiv sein, wenn der Sitzgurt angelegt ist, also *SeatBelt=1* gilt. Weiterhin muss das Feature *heater* Teil des Produkts sein, was durch *f_heater='1'* zum Ausdruck kommt. Ist die Sitzheizung installiert und der Sitzgurt angelegt, so wird die Einstellung der Sitzheizung *SHeat_request* dem Ausgang der Sitzheizung *SHeat_output* zugewiesen. Ansonsten ist die Sitzheizung immer aus (*SHeat_output=0*).

Im nächsten Schritt muss das Feature-Modell im Feature-Model-Editor (9.1.1) nachgebildet werden und der boolesche Ausdruck extrahiert werden. Dieser wird dann als separate Eigenschaft, die eine gültige Feature-Kombination *validfm* beschreibt, hinzugefügt. Die Eigenschaft hat folgende Struktur:

```
property Featuremodel is
assume:
  -- true
prove:
  at t+1: if (
    (f_seat = '1') and
    (f_seat = '1' or f_sensors = '0') and
    (f_seat = '1' or f_actors = '0') and
    ...
    (f_electrical = '1' or f_manual = '1' or f_seatmovement = '0') and
    (f_electrical = '0' or f_manual = '0')
    ) then
      validfm = '1';
    else
      validfm = '0';
    end if;
end property;
```

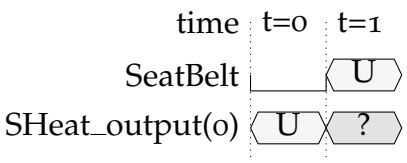
Das Argument der *if*-Anweisung besteht aus dem extrahierten booleschen Ausdruck, der *validfm* entweder wahr oder falsch werden lässt. Es entsteht ein Eigenschaftssatz für die gesamte Produktlinie.

9.2.3 *Vollständigkeitsbewertung*

Für die Prüfung auf Vollständigkeit und Widerspruchsfreiheit wird der Algorithmus aus 6.2 auf den Eigenschaftssatz angewendet. Der Eigenschaftssatz erweist sich nach einer Rechenzeit von drei Minuten als vollständig und widerspruchsfrei. Um Unvollständigkeit zeigen zu können, wird die Eigenschaft für die Sitzheizung wie folgt verändert:

```
property Sitzheizung is
assume:
  -- true
prove:
  at t+1:
    if PREV(SeatBelt = 1 and f_heater = '1') then
      SHeat_output = PREV(SHeat_request);
    end if;
end property;
```

Da es sich bei *SHeat_output* um einen 2-Bit-Wert handelt, ergeben sich vier Lücken. Die Lücken werden als Eigenschaften und als Signalverläufe ausgegeben.

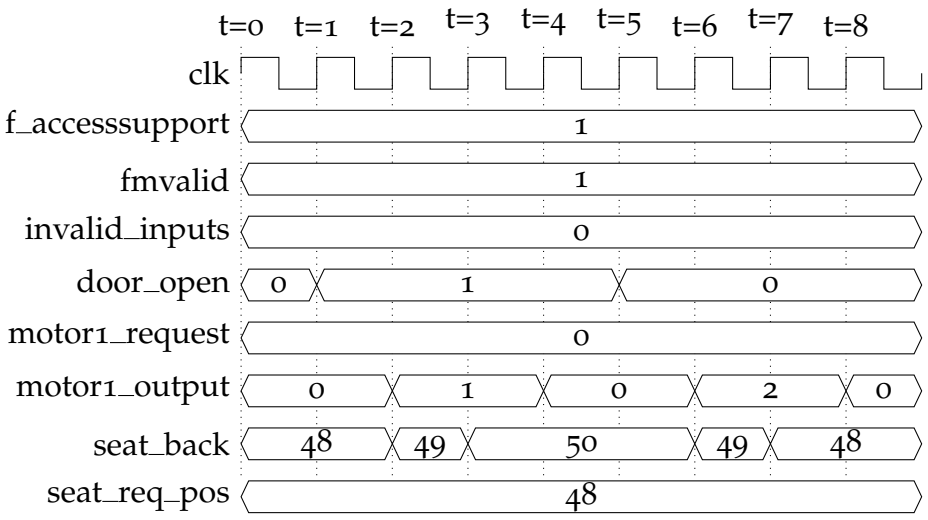


Der oben gezeigt Signalverlauf verdeutlicht die gefundene Lücke für das Bit *SHeat_output(0)*. Sein Wert ist nicht definiert, wenn der Sitzgurt nicht angelegt wird.

Analog zur Analyse auf Vollständigkeit wird der Eigenschaftssatz auf Widerspruchsfreiheit überprüft. Widersprüche werden ebenfalls als Eigenschaften und als Signalverläufe, die den jeweiligen Widerspruch erzeugen, ausgegeben. Ein Widerspruch existiert immer dann, wenn ein Ausgang in den Eigenschaften mehrfach so definiert wird, dass er bei der selben Eingangskombination im selben Zustand des Systems unterschiedliche Werte annehmen muss.

9.2.4 *Simulation des Orakels*

Um ein simulierbares Orakel zu erhalten, werden die Eigenschaften in eine VHDL-Beschreibung transformiert (6.1). Die Generierung des Orakels nimmt eine Rechenzeit von sieben Minuten in Anspruch. Das erzeugte Orakel kann dann simuliert werden (9.1.4). Für die automatische Einstiegshilfe ergibt sich z.B. folgender Signalverlauf.



Da es sich bei VHDL um eine Hardwarebeschreibungssprache handelt, benötigt der Simulator einen Takt *clk* zur Definition der diskreten Zeitpunkte. Wird die Türe geöffnet *door_open=1*, so fährt der Sitz zurück bis *seat_back=50*. Im Anschluss daran wird die Türe wieder geschlossen *door_open=0* und der Sitz fährt zur Ausgangsposition *seat_back=seat_req_pos*. Wichtig ist, dass der Sitz während des Vorgangs nicht durch eine Schaltereingabe verstellt wird *motor1_request=0*. Außerdem muss eine gültige Feature-Kombination ausgewählt sein *fmvalid=1* und die Einstiegshilfe Teil des Produkts *f_accesssupport=1* sein. Weiterhin dürfen keine ungültigen Eingaben während des Vorgangs auftreten *ivalid_inputs=0*.

9.2.5 *FMT unterstützte Testfallgenerierung*

Durch die Verknüpfung des Feature-Modells mit dem Klassifikationsbaum entsteht ein FMT (8.6). Das FMT ermöglicht die strukturierte Generierung Testfällen für Produktlinien. Äquivalenzklassen, die an den Blattknoten des Feature-Modells angehängt werden, repräsentieren die Wertebereiche, in denen sich das Feature bewegen kann. Dadurch können verschiedene Kombinationen von Grenzwerten getestet werden. Weiterhin werden die durchgeführten Tests durch die unter dem FMT liegende Tabelle dokumentiert. Abb. 37 zeigt einen Ausschnitt aus dem FMT des Autositzes. Dem Feature *Speed* werden z.B. die Äquivalenzklassen *[0]*, *[1..15]* und *[16..200]* zugeordnet. Der Autositz soll in diesen Geschwindigkeitsbereichen ein äquivalentes Verhalten zeigen. Bei stehendem Fahrzeug greift die Sitzautomatik und der Sitz kann durch die Schalter bewegt werden. In einem Bereich von 1 km/h und 15 km/h ist nur noch eine Sitzverstellung per Schalter möglich und ab 16 km/h darf der Sitz nicht mehr elektrisch verstellbar sein. Die Äquivalenzklassen ergeben sich also aus der Spezifikation. Testfälle werden in einer Tabelle analog zur Darstellung im Klassifikationsbaum gekennzeichnet. Jede Zeile der Tabelle stellt einen

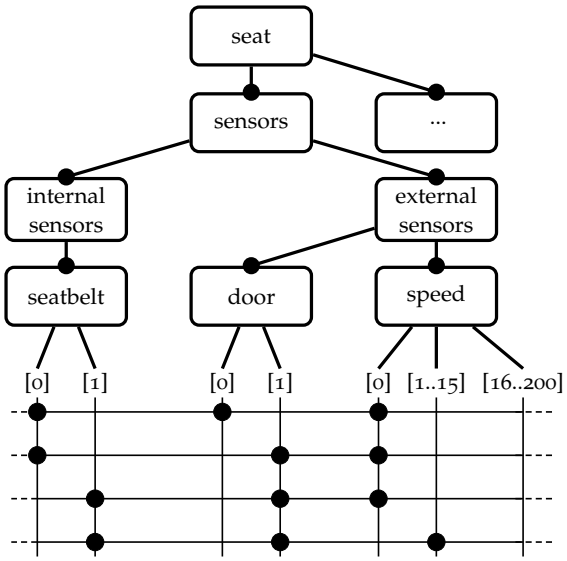


Abbildung 37: Ausschnitt aus dem FMT der Autositzfunktionen

Testfall dar. Die Äquivalenzklasse, in der sich der jeweilige Sensorwert befinden muss, wird durch einen Punkt gekennzeichnet.

Die Repräsentation der Produktlinie als FMT ermöglicht das strukturierte Testen jedes ableitbaren Produkts. Wird ein Feature durch die Produktauswahl entfernt, so werden auch die darunter liegenden Auswahlpunkte entfernt. Es entsteht ein Klassifikationsbaum für das jeweilige Produkt. Das FMT ist also eine Darstellungsform für alle möglichen Klassifikationsbäume der aus der Produktlinie ableitbaren Produkte.

9.2.6 *Beweis von Systemannahmen*

Mit dem Werkzeug OneSpin 360™ MV (9.1.5) können Eigenschaften auf VHDL-Beschreibungen bewiesen werden. Im ersten Schritt wurden die Eigenschaften, aus denen der Code erzeugt wird, wieder auf dem Code bewiesen. So kann sicher gestellt werden, dass der Algorithmus, der zur Erzeugung dient, in diesem Fall fehlerfrei funktioniert und somit die erzeugte VHDL-Beschreibung den Eigenschaften entspricht. Alle Eigenschaften halten auf der VHDL-Beschreibung und der gesamte Beweis nimmt weniger als fünf Sekunden in Anspruch.

Da Widerspruchsfreiheit und Vollständigkeit bereits für den Eigenschaftssatz gezeigt wurde (9.2.3), ist die VHDL-Beschreibung auch vollständig und widerspruchsfrei. Daraus folgt allerdings noch nicht, dass die initialen Anforderungen sinnvoll durch die VHDL-Beschreibung repräsentiert werden. Um das zu zeigen, müssen Sicherheitseigenschaften (3.3.3) über die Anforderungen formuliert und bewiesen werden. Die folgende Eigenschaft beschreibt, dass bei einer Geschwindigkeit von über 15 km/h der Sitz nicht elektrisch bewegt werden darf.

```
property NoMovementSpeed is
assume:
  at t: Speed_Sensor > 15;
prove:
  at t+1:
    motor1_output = 0 and
    motor2_output = 0 and
    motor3_output = 0;
end property;
```

Sie ist nicht Teil des Eigenschaftssatzes, aus dem das Orakel generiert wird, beschreibt aber eine Sicherheitseigenschaft, die durch den ursprünglichen Eigenschaftssatz implizit gegeben sein muss. Die Eigenschaft hält auf dem generierten Orakel. Diese Eigenschaft hält für alle Produkte. Sogar für solche, die eigentlich nicht aus dem Feature-Modell ableitbar wären. Anders sieht es für Sitzheizung und Klimaanlage aus, die sich nach dem Feature-Modell gegenseitig ausschließen.

```
property HeaterOrAircon is
assume:
  at t: validfm = 1;
prove:
  at t+1:
    SAircon_output = 0 or SHeat_output = 0;
end property;
```

Hier muss Teil der Annahme sein, dass das Feature-Modell gültig ist (*validfm=1*). Die Eigenschaft, die anzeigt, dass die automatische Sitzbewegung beim Einsteigen durch ein Drücken des Tasters für die Sitzbewegung unterbrochen wird, ist im Folgenden dargestellt.

```
property AbortAutomatic is
assume:
  at t: Speed_Sensor = 0 and Door_open = '1' and Seat_Back < 50;
  at t: validfm = '1' and f_electrical = '1';
  at t+1: invalid_inputs = '0';
prove:
  at t+1:
    if PREV(motor1_request = 1 and Seat_Back > 0) then
       motor1_output = 1;
    end if;
end property;
```

Hier ist neben der Beschreibung der Situation und der Features, die ausgewählt sein müssen, noch die Annahme, dass *invalid_inputs='0'* gilt, nötig.

Weiterhin wird bewiesen, dass immer nur ein Motor aktiv sein darf und dass der logische Ausdruck, der zur Beschreibung des Feature-Modells dient, die Zusammenhänge im Feature-Modell korrekt widerspiegelt.

Werden die impliziten Systemannahmen zum Eigenschaftssatz hinzugefügt, so sind sie redundant (8.3.1), wenn sie durch den ursprünglichen Eigenschaftssatz erfüllt werden (8.5.2). Diese Vorgehensweise wird mit den auf dem erzeugten Orakel bewiesenen impliziten Systemannahmen ebenfalls erprobt. Tab. 3 zeigt

	PL	AI	VP	AA	OM	HA	SM
Properlyze [s]	6,72	8,49	6,74	8,42	6,71	6,74	6,74
Theoreme [#]	139	147	144	139	141	140	139

Tabelle 3: Beweis impliziter Systemannahmen

die Ergebnisse des Beweises der impliziten Systemannahmen über redundante Eigenschaften. Hierfür muss weder OneSpin 360™ MV verwendet werden, noch ein Orakel generiert werden. Verglichen wird die Laufzeit und die Anzahl der generierten Theoreme für die Eigenschaften der ursprünglichen Produktlinie *PL* und der Produktlinie inklusive aller impliziter Systemannahmen *AI*. Weiterhin werden, um einen Laufzeitvergleich zu erhalten, die Eigenschaften für die Produktlinie nacheinander mit einzelnen impliziten Systemannahmen erweitert (*VP* - nur valide Produkte möglich, *AA* - Einstiegsautomatik kann manuell unterbrochen werden, *OM* - es darf zu jedem Zeitpunkt nur ein Motor laufen, *HA* - entweder Sitzheizung oder Klimaanlage möglich, *SM* - eine Sitzbewegung ist nur bis 15 km/h möglich). Die längste Laufzeit ergibt sich beim maximalen Eigenschaftssatz *AI*. Die Verlängerung der Laufzeit ist vor allem der impliziten Systemannahme *AA* geschuldet. Alle weiteren impliziten Systemannahmen verlängern die Laufzeit nur unerheblich. Für die Verwendung dieser Methodik wird daher empfohlen, zuerst nur den ursprünglichen Eigenschaftssatz *PL* und den Eigenschaftssatz mit allen impliziten Systemannahmen *AI* zu vergleichen. Nur, wenn bei diesem Vergleich Fehler festgestellt werden, müssen die impliziten Systemannahmen einzeln überprüft werden.

Um die Mächtigkeit des Ansatzes zu zeigen, wurde eine Lücke zum Eigenschaftssatz hinzugefügt, indem eine Zeile auskommentiert wird.

```
property Motor2 is
assume:
  -- true
prove:
  at t+1:
     if PREV(motor1_request = 0 and motor3_request = 0
             and Speed_Sensor < 16 and SHeat_request = 0
             and f_electrical = '1') then
       ...
     else
       --motor2_output = 0 and
       Seat_Left = PREV(Seat_Left);
     end if;
end property;
```

Ein Beweis mit OneSpin 360™ MV (9.1.5) ergibt daraufhin, dass zwei implizite Systemannahmen über die Motorbewegung nicht mehr halten. Eine gleichzeitige Bewegung mehrerer Motoren ist nach Abb. 38 (a) möglich. Außerdem kann der Sitz nach Abb. 38 (b) auch bei einer Geschwindigkeit von über 15 km/h bewegt werden. Beide Fehler resultieren aus der Struktur des Orakels, das sich

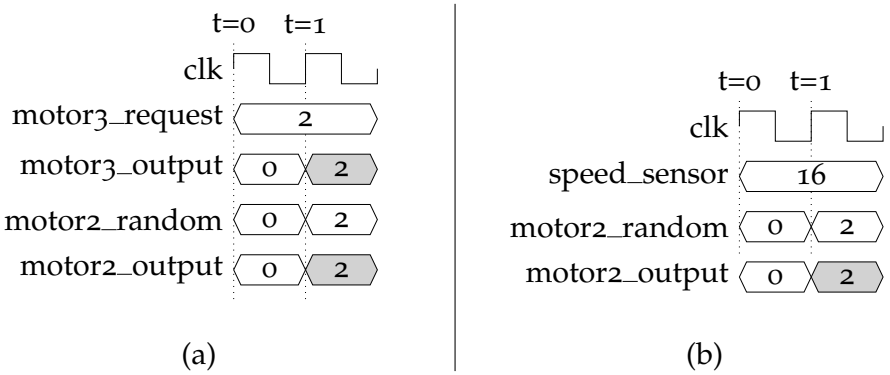


Abbildung 38: Gegenbeispiele in OneSpin 360™ MV

bei Spezifikationslücken zufällig verhalten darf. Zu diesem Zweck wird dem Ausgang *motor2_output* der Zufallswert *motor2_random* zugewiesen.

Der Beweis der impliziten Systemannahmen über die Anzahl der Lücken beim Hinzufügen redundanter Eigenschaften wird ebenfalls durchgeführt. Hier zeigt sich eine Verringerung der Anzahl der Lücken, wenn die Eigenschaften aus den impliziten Systemannahmen über die Motorbewegung aus dem Eigenschaftssatz entfernt werden. Wird z.B. die implizite Systemannahme über die Beschränkung der Sitzbewegung ab einer Geschwindigkeit von über 15 km/h entfernt, so ergeben sich entsprechende Lücken. Eine implizite Systemannahme, die auf einem vollständig determinierten Eigenschaftssatz nicht hält, würde zu Widersprüchen führen.

9.2.7 *Zusammenfassender Vergleich*

Existiert ein Eigenschaftssatz für eine Produktlinie, so kann dieser zur Vollständigkeitsbewertung, zur Generierung von Orakeln und zur Verifikation eingesetzt werden. Die Orakelgenerierung und die Vollständigkeitsbewertung müssen nur einmal vorgenommen werden und beziehen sich dann auf alle möglichen Produkte. Tab. 4 zeigt die statistischen Daten, die während der

	PL	MP	oS	oA	oSoA
Properlyze [min]	2:24	2:21	2:05	0:07	0:07
Candogen [min]	7:00	5:45	4:54	<0:01	<0:01
Theoreme [#]	139	130	127	112	109
normalisierte Eigenschaften [#]	65	64	62	28	24
Zerlegung [ms]	52	44	44	16	16
VHDL-Generierung [ms]	216	200	176	24	16

Tabelle 4: Ergebnisse

Generierung erhoben wurden. Sie wurden für die Produktlinieneigenschaften *PL*, ein maximal ausgestattetes Produkt *MP*, den Autositz ohne Sitzheizung und Klimaanlage *oS*, ohne Einstiegsautomatik *oA* und ohne Sitzheizung, Klimaanlage und Einstiegsautomatik *oSoA* erhoben. Der Eigenschaftssatz für die Produktlinie beinhaltet die meiste Information und benötigt daher auch die größte Rechenzeit. Die Vollständigkeitsbewertung ist für die Produktlinie und das maximal ausgestattete Produkt fast gleich. Die Generierung des Cando-Objekts ist bei dem maximale ausgestatteten Produkt ca. 18% schneller. Schon bei der Generierung von *MP* und *oS* ist der Produktlinienansatz schneller. Der *Break-Even-Point* (Kapitel 5) liegt also bei ca. 2-3 generierten Produkten, was sich mit der Definition aus [PBL05] deckt. Im industriellen Umfeld existieren Produktlinien aus denen Millionen möglicher Produkte abgeleitet werden können.

Die Komplexität liegt bei der Generierung vor allem an der automatischen Einstiegsfunktion. Ein Produkt ohne diese Funktionalität wird deutlich schneller durch Candogen und Properlyze bearbeitet. Die Anzahl der Theoreme sinkt von *PL* zu *oSoA* um 28%, während die Anzahl der normalisierten Eigenschaften um 63% abnimmt.

9.3 HANDYSPIEL

Die Anwendung des Ansatzes auf eine Spezifikation, die in Form von Aktivitätsdiagrammen gegeben ist, wird im Folgenden präsentiert. Als Grundlage dienen Aktivitätsdiagramme, die den Ablauf des Handyspiels *Bomberman* beschreiben [OWES11]. In Abb. 39 ist das Feature-Modell des Handyspiels

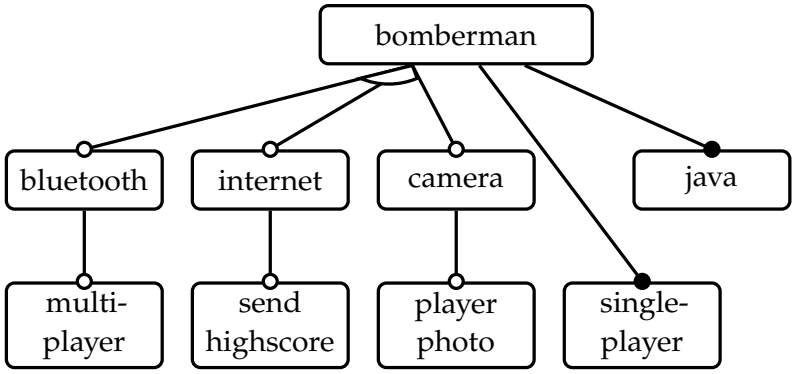


Abbildung 39: Bomberman Feature-Modell [OWES11]

dargestellt. Es besteht aus dem Wurzelknoten *Bomberman* sowie aus den Pflichtknoten *Java* und *Singleplayer*. Weiterhin enthält es eine Odergruppe bestehend aus *Bluetooth*, *Internet* und *Camera*, von der mindestens ein Feature Teil des Produkts sein muss. Abhängig von den gewählten Features aus der Odergruppe bilden *Multiplayer*, *Send Highscore* und *Player Photo* optionale Bestandteile des Produkts. Abb. 40 zeigt drei mögliche Produkte, die aus der Produktlinie

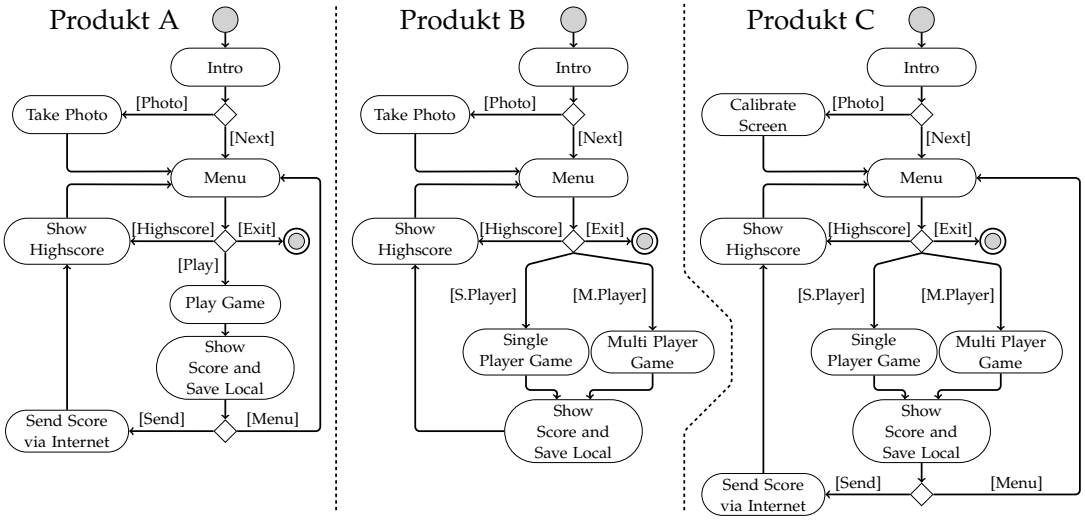


Abbildung 40: Bomberman Aktivitätsdiagramme [OWES11]

generiert werden können. Der Ansatz wird im Folgenden sowohl anhand eines einzelnen Produkts als auch anhand der gesamten Produktlinie erörtert.

9.3.1 *Einzelnes Produkt*

Für die Betrachtung eines einzelnen Produkts wird *Produkt A* gewählt. Bei der korrekten Übersetzung von Aktivitätsdiagrammen in Eigenschaften können keine Spezifikationslücken auftreten. Ein syntaktisch korrekt geschriebenes Aktivitätsdiagramm hat vom Start- bis zum Endknoten immer mindestens eine aktive Aktion. Das Aufschreiben der Eigenschaften zur Definition von Aktivitätsdiagrammen hilft trotzdem bei der Analyse und dem Verständnis des Aktivitätsdiagramms.

9.3.2 *Produktlinie*

Für die Beschreibung der Produktlinie müssen einzelne Teile des Aktivitätsdiagramms mit ihren korrespondierenden Features in Bezug gesetzt werden. Dies wird durch folgende Beispieleigenschaft verdeutlicht:

```
assume:
  at t: reset = '0';
prove:
  at t+1:
    if PREV(activ = a_intro) then
      if (f_playerphoto = '1' and c_intro = TAKEPHOTO) then
        activ = a_takephoto;
      elsif (f_touch = '1' and c_intro = FIRSTGAME) then
        activ = a_touchscreencal;
      else
        activ = a_menu;
      end if;
    end if;
end property;
```

Die Eigenschaft beschreibt den Aktivitätswechsel aus der Aktivität *Intro*. Die Aktivität *Take Photo* kann nur erreicht werden, wenn *Player Photo* Teil des Produkts ist.

Aktivitätsdiagramme im Produktlinienkontext zu verwenden bietet sich also an. Es ist allerdings nötig, eine Funktionalität hinter den jeweiligen Aktivitäten zu hinterlegen. Dadurch wird das Verhaltensmodell der Produktlinie strukturiert und einfacher änderbar.

9.4 A-7E FLUGZEUG

Die Spezifikation des A-7E Flugzeugs [AFB+88] ist in SCR-Notation (4.2.2) verfasst und ist zur öffentlichen Analyse freigegeben. Sie definiert die Reaktion

des Flugzeugs auf Piloteneingaben. Die Steuerung gliedert sich in vier *mode classes* (*alignment*, *navigation*, *navigation update*, *weapon delivery*), die als parallele Zustandsautomaten begriffen werden können.

Um ein System von der Komplexität eines Flugzeugs zu beschreiben, muss ein Großteil der Funktionalität durch einzelne Module gekapselt werden. Ein Modul bietet vordefinierte Funktionalität an und wird über ein generisches Interface mit dem Gesamtsystem verbunden. Die genaue Implementierung innerhalb des Moduls bleibt dem Gesamtsystem verborgen. Weiterhin muss jedes Modul dafür sorgen, dass es sich beim Start des Systems initialisiert, also einen vordefinierten Anfangszustand einnimmt. Die Fehlerbehandlung ist außerdem Teil der Modulfunktionalität. Verletzungen der Interfacespezifikation bzw. interne Fehlerzustände müssen behandelt und dem Gesamtsystem zur Verfügung gestellt werden. Welche Funktionalität auf welche Weise innerhalb von Modulen gekapselt wird, stellt im Normalfall eine Designentscheidung dar. Der Entwickler muss die Wahrscheinlichkeit von Änderungen innerhalb von Funktionsblöcken abschätzen und anhand dessen den Funktionsumfang der Module definieren. In einem System kann Hardware, Software oder Verhalten modularisiert werden. Bei der Hardwaremodularisierung wird die Schnittstelle nicht verändert, während sich die Hardware ändern kann. Im A-7E Flugzeug ist das z.B. bei der *weapon delivery class* der Fall. Je nach Bestückung der Flügelaufhängungen ändert sich die Hardware abhängig vom bestückten /WEAPTYP/, die Schnittstelle und die Instrumente innerhalb des Cockpits bleiben unverändert. In der Software werden meistens Algorithmen modularisiert. So wird einem Sortieralgorithmus immer eine Menge unsortierter Werte übergeben und eine sortierte Menge als Ausgabe erwartet. Der implementierte Algorithmus, der die Sortierung übernimmt, ist allerdings für das Gesamtsystem nicht sichtbar. Systemverhalten wird im Normalfall auch in Software modularisiert. Dabei handelt es sich nicht um Berechnungen, sondern um beobachtbares Verhalten. So kann z.B. der maximale Anstellwinkel eines Flugzeugs in Abhängigkeit der Geschwindigkeit durch ein Softwaremodul berechnet werden. Gibt der Pilot das Signal zur Erhöhung des Anstellwinkels, so kann dieses durch ein Softwaremodul verhindert werden.

9.4.1 *Aufstellen der Eigenschaften*

Im Rahmen dieses Beispiels wird die *navigation update mode class* betrachtet. Die Eigenschaften werden nach den in 7.2 aufgestellten Regeln für die Extraktion von Eigenschaften aus SCR-Anforderungsdokumenten aufgestellt. Verwendet wird ausschließlich die *mode transition table* der *navigation update mode class*.

In der A7E-Spezifikation sind neben den Eingangswerten, die durch Sensoren oder Piloteneingaben auftreten können, weitere Situationen beschrieben, die

Eingaben zusammenfassen. Ein Beispiel hierfür bietet *!station selected!*, das wie folgt definiert ist [AFB+88]:

„/STAnRDY/ = YES for any n ∈ {1 2 3 6 7 8}“

Für /STAnRDY/ gilt:

„An item will be Yes if and only if the pilot has selected the corresponding station using the pushbutton station selectors.“

Jede *station* beschreibt eine Aufhängung unterhalb des Flügels. Die Nummern 4 und 5 befinden sich am Rumpf des Flugzeugs und werden nicht berücksichtigt. Eine *station* kann mit einem /WEAPTYP/ bestückt werden, dessen Eigenschaften in einer Tabelle festgehalten sind. Zu den Eigenschaften gehören der Klartextname, eine Waffenklassifizierung, die Länge des Abwurfimpulses, die minimale Zeit zwischen zwei Abwürfen und die Information, ob an einer Aufhängung mehrere Waffen des selben Typs befestigt sein können. *!station selected!* ist immer dann wahr, wenn der Pilot eine unter dem Flügel befindliche Aufhängung ausgewählt hat und wird durch folgende Eigenschaft repräsentiert:

```
property stationselected is
assume:
  at t: reset = '0';
prove:
  at t+1:
    if PREV(STAnRDY /= b"000000") then
      station_selected = '1';
    else
      station_selected = '0';
    end if;
end property;
```

Durch die Beschreibung von Situationen wird die A7E-Spezifikation kompakter und besser lesbar. Diese Technik wurde daher auch beim Aufstellen der Eigenschaften angewandt.

9.4.2 *Vollständigkeitsbewertung und Orakelgenerierung*

Der Eigenschaftssatz für die *navigation update mode class* umfasst insgesamt sieben Eigenschaften. Nach dem Aufstellen der Eigenschaften erfolgt eine Prüfung auf Vollständigkeit und Widerspruchsfreiheit. Diese endet für die *mode transition table* der *navigation update mode class* nach 0:07 Minuten und zeigt weder Lücken noch Widersprüche in der Spezifikation. Die Generierung des Orakels 2:45 Minuten in Anspruch.

9.5 GEAROMAT

Der Gearomat ist ein Produkt der Gearomat GmbH, dessen Spezifizierung und Entwicklung im Verlauf dieser Arbeit betreut wurde. Neben den bisher gezeigten Beispielen, die Teilaspekte des Ansatzes behandeln, soll der Gearomat dazu dienen, den Ansatz ganzheitlich zu erörtern. Da es sich um ein kommerzielles Produkt handelt, unterliegen Teile der Entwicklung und der Spezifikation der Geheimhaltung. Es wird daher ausschließlich auf die Erfahrungen, die im Zusammenhang mit den in dieser Arbeit präsentierten Konzepten gewonnen wurden, eingegangen.

9.5.1 *Beschreibung des Systems*

Beim Gearomat handelt es sich um einen Automaten, der von einem gewöhnlichen Getränkeautomaten abgeleitet wurde und für Rechenzentren bestimmt ist. Da in Rechenzentren die Wartung meist nachts erfolgt oder sie sich an abgelegenen Orten befinden, müssen auch in diesen Situationen Ersatzteile, wie Kabel, Reinigungstücher oder optische Transceiver, verfügbar sein. Können fehlende Teile nicht besorgt werden, führt das zu hohen Kosten, da die Wartung verschoben werden muss oder im schlimmsten Fall ein Netzwerkausfall droht. Der Gearomat muss folgende Eckdaten erfüllen:

- Sicheres Bezahlsystem
- Dokumentation jedes Verkaufs per Foto oder Video
- Verwaltung des Bestands
- Programmierung von optischen Transceivern
- Touch-Screen Bedienung mit Flash-Oberfläche
- Netzwerkanbindung an einen zentralen Server
- Erweiterbarkeit

Diese Funktionalität wird erreicht, indem ein Standardautomat mit einem eingebettetem System erweitert wird. Weiterhin wird für die Programmierung von optischen Transceivern die flexBox der Firma Flexoptix GmbH verwendet. Die flexBox ist bereits vollständig entwickelt und muss mit dem eingebetteten System, das den Gearomat steuert, verbunden werden. Weiterhin muss der Gearomat mit der Kontrolleinheit des Automaten und mit einem zentralen Server für den Bezahlvorgang kommunizieren.

Abb. 41 verdeutlicht den grundlegenden Aufbau des Systems. Ein möglicher Anwendungsfall ist z.B., dass der Kunde einen optischen Transceiver kaufen möchte und der Kauf fehlerfrei abläuft:

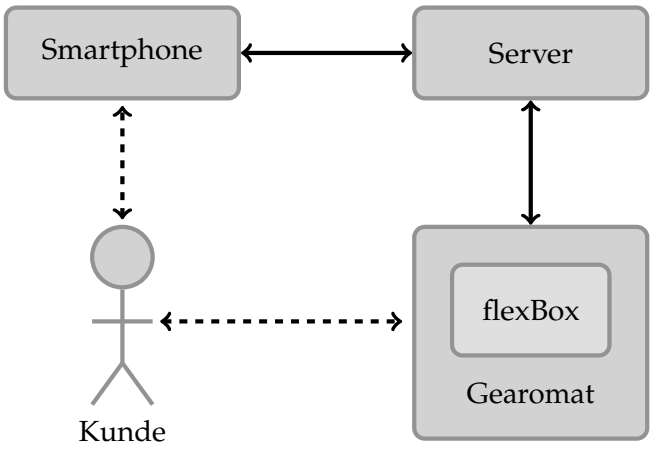


Abbildung 41: Überblick über das Gearomat-System

1. Kunde wählt Produkt auf dem Touch-Screen des Gearomat aus
2. Kunde verbindet sich mit seinem Smartphone unter Eingabe seiner Nutzerdaten mit dem Server
3. Kunde gibt den auf dem Touch-Screen des Gearomaten angezeigten Code auf seinem Smartphone ein, um den Verkauf seinem Nutzerkonto zuzuordnen
4. Kunde sieht den Gesamtpreis auf seinem Smartphone und bestätigt die Bezahlung
5. Server teilt Gearomat die erfolgte Bezahlung mit
6. Gearomat gibt gekauftes Produkt aus
7. Kunde entnimmt optischen Transceiver
8. Gearomat fotografiert Entnahme durch den Kunden
9. Kunde programmiert optischen Transceiver an der flexBox

Es existiert noch eine Vielzahl weiterer Anwendungsfälle. Die Gesamtheit aller Anwendungsfälle dient als Grundlage für die Spezifikation.

9.5.2 *Aufstellen des Feature-Modells*

Der Gearomat ist ein Produkt, das erweiterbar und auf Kundenwünsche anpassbar sein soll. Daher bietet sich das Aufstellen eines Feature-Modells an, um die Möglichkeiten, die der Gearomat bietet, graphisch darzustellen. Abb. 42 zeigt einen Ausschnitt aus dem Feature-Modell des Gearomaten. Das vollständige Feature-Modell enthält noch weitergehende Informationen über den Softwareteil des eingebetteten Systems sowie über die Funktionalität, die in einem optionalen Field Programmable Gate Array (FPGA) beinhaltet sein

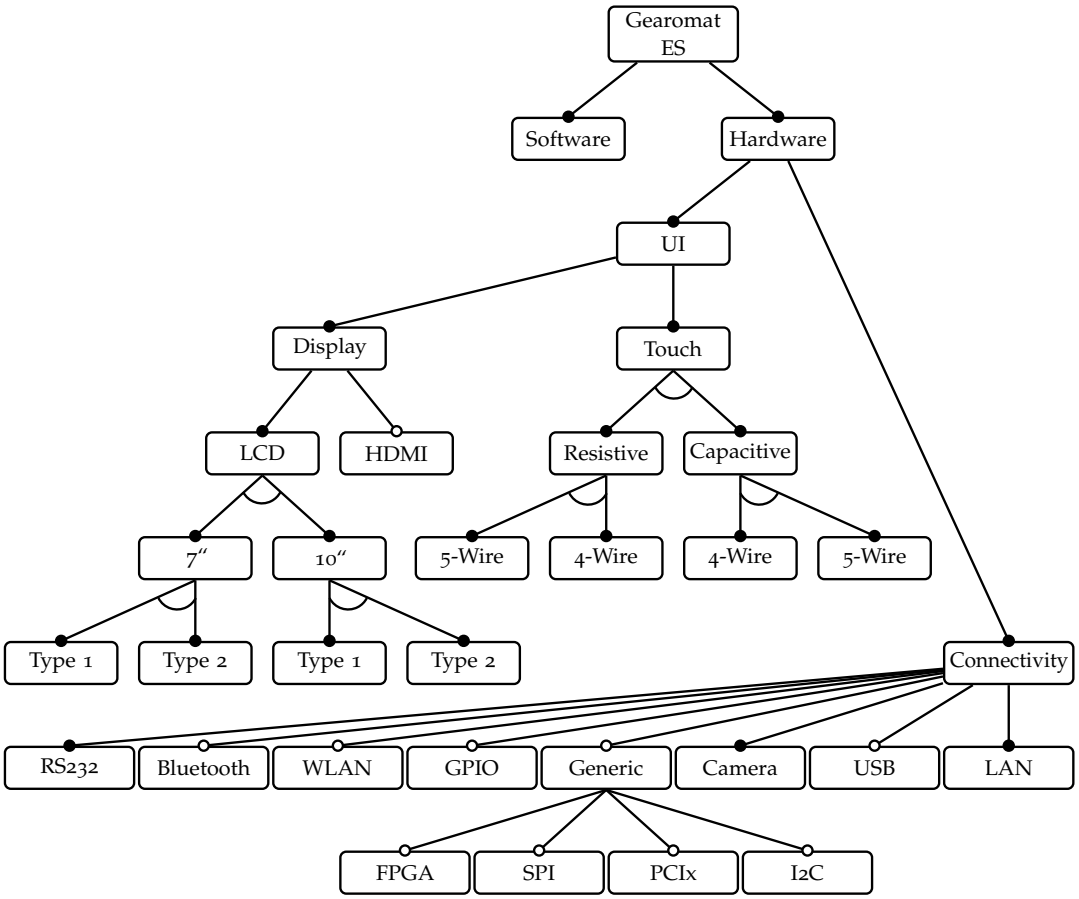


Abbildung 42: Ausschnitt des Gearomat-Feature-Modells

kann. Im Feature-Modell sind die Hardwarekomponenten des eingebetteten Systems dargestellt.

Die Benutzereingaben werden über einen Touchscreen realisiert. Ein Touch-Screen besteht immer aus einem regulären LCD-Display und einer separaten Touch-Oberfläche, die auf den Bildschirm gelegt wird. Das LCD-Display wird direkt mit dem im eingebetteten System enthaltenen Mikroprozessor verbunden. Hier ist zu beachten, dass es unterschiedliche Verbindungsarten gibt, die sich mit Hinblick auf Pegel und Signalleitungen unterscheiden. Je nach Typ der Verbindung muss die Schaltung für die Verbindung angepasst werden. Weiterhin kann bei Touch-Oberflächen zwischen resistiven und kapazitiven Varianten gewählt werden, die wiederum über ein 4-wire oder 5-wire Interface mit dem System verbunden werden.

Die Verbindungsmöglichkeiten des Boards sind sowohl durch den gewählten Mikroprozessor als auch durch die Anforderungen vorgegeben. LAN, RS232 und Kamera müssen immer Teil des Systems sein. Schaltungsbedingt kann LAN nur in Zusammenhang mit USB bestückt werden. Eine ähnliche Abhängigkeit besteht zwischen Bluetooth und WLAN, die sich auf dem selben Modul befinden und daher nur gemeinsam bestückt werden können. Um für

die Zukunft die Erweiterbarkeit des Systems sicherzustellen, müssen noch unterschiedliche Anschlussmöglichkeiten über einen Steckverbinder realisiert werden.

9.5.3 *Nutzen des Feature-Modells*

Das Feature-Modell strukturiert die Menge der möglichen Produkte so, dass sie für alle Projektbeteiligten einfach nachvollziehbar sind. Die Möglichkeiten des fertigen Produkts können einfach verstanden und erörtert werden. Auch kann das Feature-Modell erweitert werden, wenn Features hinzukommen. Die Abschätzung des Aufwands für die Erweiterung der Produktlinie kann von den Entwicklungsverantwortlichen durchgeführt werden. Weiterhin hilft das Feature-Modell während der Entwicklung dabei, den Restaufwand abzuschätzen. Nicht implementierte oder getestete Features können markiert werden. So entsteht ein optischer Eindruck über den Projektstatus.

Für die Produktion ist das Feature-Modell auch von großem Vorteil. Je nach ausgewählten Features, muss die Platine des eingebetteten Systems anders bestückt werden. Die Bestückung wird normalerweise nach einem Bestückungsplan durchgeführt, der aus der Bauteilposition mit Name und der *Bill of Materials*, in der alle Bauteile mit Position genau festgehalten sind, besteht. Je nach ausgewählten Features muss die Bestückung geändert werden. Das Gearomat ES besteht aus über 500 Bauteilen, die entweder bestückt oder nicht bestückt sein können bzw. mit anderen Werten (Widerstand oder Ram-Größe) bestückt werden können. Alleine für den Wechsel des Displays von *Type 1* auf *Type 2* führt zu Veränderungen an 66 Positionen. Durch die Verwendung des Feature-Modells kann dieser Prozess vereinfacht werden, indem jedes Feature auf eine Menge von Bauteilen abgebildet wird. Durch das Zusammenfügen aller Bauteilmengen erhält man die *Bill of Materials* des gesamten Produkts.

Weiterhin kann bei der Software-Entwicklung das Feature-Modell dabei helfen, Abhängigkeiten zwischen einzelnen Modulen zu erkennen und zu dokumentieren. Es kann sogar die Abhängigkeit zwischen einzelnen Softwaremodulen und Hardwarekomponenten dargestellt werden. So muss ein Treiber für das WLAN-Modul nur installiert werden, wenn das WLAN-Modul Teil des Produkts ist. Die Verfügbarkeit von WLAN-Funktionalität in der Software hängt auch von der Wahl des physikalischen WLAN-Moduls ab. Es können automatisiert Konfigurationsdateien und Installationsanleitungen für die Software generiert werden.

9.5.4 *Vollständigkeit der Anforderungen*

Es werden Teile der Anforderungen auf Vollständigkeit geprüft. Dabei handelt es sich vor allem um den schematischen Ablauf des Verkaufsvorgangs inklusive der Behandlung von Fehlern. Da der Gearomat im ersten Schritt nur in einer einzigen Konfiguration ausgeliefert wird, werden, bedingt durch die knappe Entwicklungszeit, nur die Anforderungen für dieses Produkt überprüft.

Für das Gearomat-System werden drei Eigenschaftssätze geschrieben. Einer beschreibt das mögliche Verhalten des Kundens inkl. Smartphone (5 Eigenschaften), einer beschreibt das Verhalten des Ausgabeautomatens (4 Eigenschaften) und einer beschreibt das Verhalten des zentralen Servers (6 Eigenschaften). Diese Aufteilung resultiert in strukturierten und gut lesbaren Eigenschaftssätzen. Die Eigenschaftssätze werden sodann auf Vollständigkeit überprüft und Lücken entfernt.

Im Verlauf des Projekts wird deutlich, dass die Formulierung von Eigenschaftssätzen während dem Verfassen des Anforderungsdokuments von großer Hilfe ist. Der Zwang zum eindeutigen und vollständigen Formulieren der Eigenschaften führt zu einem sehr guten Problemverständnis. Des Weiteren unterstützt die Formulierung der Eigenschaften die Formulierung der Anforderungen. Diese werden mit einer viel größeren Sorgfalt verfasst und zeichnen sich durch explizite Aussagen und Eindeutigkeit aus. Weiterhin werden implizite Annahmen über das Systemverhalten formuliert und in die Spezifikation aufgenommen. Es bietet sich daher auch in zukünftigen Projekten an, das äußere Verhalten des Systems begleitend zum Aufstellen der Anforderungen zu formalisieren. So kann die Konsistenz der Anforderungen dynamisch überprüft werden und der Grad der Vollständigkeit schrittweise auf 100% erhöht werden.

9.5.5 *Orakel*

Das Orakel kann aus den Eigenschaftssätzen generiert werden. Auf seine Ausführung wird allerdings bisher verzichtet. Der Gearomat befindet sich noch in der Entwicklungsphase. Die Verwendung eines Orakels für das äußere Verhalten ist erst für den Freigabetest vorgesehen und hilft zu diesem Zeitpunkt, Testergebnisse vorherzusagen. So kann der Gearomat auch von Mitarbeitern, die nicht der Entwicklungsabteilung zugehörig sind, getestet werden. Die Testergebnisse können auf Grund der Vorhersagen durch das Orakel augenblicklich auf Korrektheit überprüft werden.

10

FAZIT UND AUSBLICK

Produktlinien nehmen einen immer größeren Stellenwert in der Entwicklung ein. Die Kosten für das Erstellen und Warten von Anforderungen für Produktlinien müssen in einem erträglichen Maß gehalten werden. Das Interesse an automatisierten Methoden im Kontext von Produktlinien wächst daher momentan immer stärker im industriellen Umfeld. Formale Methoden gewinnen mit Hinblick auf mögliche Standardisierungen des Entwicklungs- und Testprozesses stetig an Bedeutung. Diese Arbeit präsentiert Konzepte zur Anforderungsanalyse und Orakelgenerierung für Produktlinien eingebetteter Systeme und für einzelne eingebettete Systeme.

Die präsentierten Verfahren vereinigen Techniken aus dem Produktlinienkontext mit solchen, die aus der formalen Verifikation von Hardware bekannt sind. Dadurch werden Anforderungsdokumente unter Beibehaltung der in ihnen enthaltenen Variabilitäts- und Abhängigkeitsinformationen formalisiert. Hierfür muss ein Zusammenhang zwischen den Anforderungen und einem Variabilitätsmodell, wie z.B. dem Feature-Modell, hergestellt werden. Die durch die Anforderungen definierte Funktionalität muss daraufhin mit den enthaltenen Variabilitätsinformationen in eine formale Form gebracht werden.

Liegt ein Anforderungsdokument vollständig oder in Teilen als Eigenschaftssatz vor, so kann dieses mit formalen Methoden bewertet und verarbeitet werden. Die Bewertung des Eigenschaftssatzes bezieht sich im Kontext dieser Arbeit auf die Vollständigkeit, die Widerspruchsfreiheit und die Freiheit von Redundanz. Es können Lücken und Widersprüche in Anforderungsdokumenten von einzelnen Produkten sowie von Produktlinien aufgedeckt werden. Die ansonsten nur manuelle Bewertung der Anforderungen wird unterstützt.

Ein Eigenschaftssatz, der keine unbeabsichtigten Lücken enthält und der frei von Widersprüchen ist, kann in ein Orakel auf Basis eines Cando-Objekts übersetzt werden. Auch hierfür bedienen sich die präsentierten Konzepte eines bekannten Ansatzes aus der formalen Verifikation. Ein so generiertes Orakel kann für zwei Einsatzzwecke dienen. Auf der einen Seite können Testfälle darauf ausgeführt werden und die Ergebnisse mit den Testfällen auf dem realen implementierten System verglichen werden. Auf der anderen Seite ist es möglich, Sicherheitseigenschaften zu formulieren, die für die Spezifikation implizit gelten müssen. Diese können mit Hilfe eines Werkzeuges formal auf dem Orakel bewiesen werden. So kann garantiert werden, dass bestimmte, als unsicher geltende Situationen niemals auftreten können.

Bei der Anwendbarkeit eines Ansatzes, der auf der Übersetzung von Anforderungen in formale Eigenschaften für Produktlinien basiert, ist die Art, in der die Anforderungen vorliegen, entscheidend. Teil dieser Arbeit ist aus diesem Grund auch die Auseinandersetzung mit der Art, in der Anforderungen formuliert sein können. Es werden Anforderungen in Textform, in Tabellenform und als UML-Aktivitätsdiagramme untersucht und Übersetzungsregeln formuliert. So könnte die Extraktion formaler Eigenschaften in Zukunft teilweise automatisiert werden. In welchen Ausmaß sich der Prozess der Extraktion von Eigenschaften aus Anforderungen für Produktlinien automatisieren lässt, muss noch gezeigt werden. Vor allem, wenn die Anforderungen in Textform vorliegen, können Mehrdeutigkeiten und missverständliche Formulierungen die Überführung in formale Eigenschaften erschweren oder unmöglich machen. Klare Vorgaben für die Formulierung der Anforderungen sind nötig bzw. müssen für die Erstellung von Anforderungen für Produktlinien erweitert werden.

Die Verwendung einer formalen Sprache, um die Anforderungen zu definieren, vereinigt mehrere Vorteile in sich. Beim Übersetzen der Anforderungen in die formale Sprache vertieft der Entwickler sein Verständnis der Spezifikation und kann Ungereimtheiten oder nicht sinnvolle Anforderungen identifizieren. Die formale Spezifikation beschreibt das Verhalten des Systems eindeutig und lässt für das zeitliche und logische Verhalten des Systems keine unerwünschten Freiheitsgrade. Die Verwendung einer formalen Sprache bietet sich bereits während dem initialen Aufstellen der Anforderungen an. Daraus resultieren klare, eindeutige und strukturierte Anforderungsdokumente.

Bei der Übersetzung von Anforderungen in eine formale Sprache besteht immer die Herausforderung, die gesamten in den Anforderungen enthaltenen Informationen formal korrekt abzubilden. Die gewählte Formalisierung muss dafür ausdrucksstark genug sein. Das hier hauptsächlich verwendete ITL ist dazu in der Lage, Aussagen über das gültige zeitlich-logische Verhalten von Systemen zu formulieren. Diese müssen während der gesamten Ausführung des Systems gelten. Weiterhin wird gezeigt, dass durch ein Feature-Modell repräsentierte Variabilität in die Eigenschaften mit einfließen kann und so ein Eigenschaftssatz für eine komplette Produktlinie entsteht.

Ein Problem bei der Übersetzung von Anforderungen stellt die Form dar, in der die Anforderungen vorliegen. Diese kann unter Umständen Aussagen enthalten, die nicht mehr in ITL repräsentierbar sind. Des Weiteren können die Anforderungen bereits in einer formalen Form, wie z.B. CSP, vorliegen, die nicht mit ITL vereinbar ist. Eine äquivalenzerhaltende Umformung ist dann nicht mehr möglich. Aus diesem Grund sollte die Anwendung der Vollständigkeits- und Orakelgenerierungsalgorithmen auf weitere formale Sprachen evaluiert werden. Das ist auch nötig, um komplexere Berechnungen formulieren und bearbeiten zu können. Ließen die Algorithmen die Verwen-

dung weiterer formaler Sprachen zu, so würde außerdem der Aufwand, der bei der Übersetzung dieser Sprachen in ITL entsteht, eliminiert.

Die experimentellen Ergebnisse zeigen, dass die beschriebenen Konzepte auf Praxisprobleme anwendbar sind. Größere Anforderungsdokumente müssen allerdings erst analysiert und die jeweiligen Teile identifiziert werden, auf die die Konzepte anwendbar sind. Eigenschaftssätze mit hoher Komplexität stellen ein Problem mit Hinblick auf die Rechenzeit dar, die eine Vollständigkeitsbewertung oder eine Orakelgenerierung unmöglich machen kann. Hier müssen Methoden gefunden werden, die Eigenschaftssätze passend zu vereinfachen. Das kann z.B. durch die Aufteilung des Eigenschaftssatzes in abgeschlossene Module geschehen. Weiterhin muss evaluiert werden, welche Formulierungen bzw. welche Teile von Anforderungen die eigentliche Komplexität mit Hinblick auf die eingesetzten Algorithmen ausmachen.

Von Interesse ist sicher auch die Anwendung des Ansatzes in anderen Domänen. Im Prozessmanagement können Produktlinien helfen, für Entwicklungsprojekte angepasste Prozesse automatisch zu generieren. Das würde dazu führen, dass der Entwicklungsprozess für jedes Projekt genau auf die Anforderungen an Qualität, Geschwindigkeit und Projektgröße zugeschnitten wäre.

Weiterhin sind Produktlinienansätze auf das Management von Produktionsdaten anwendbar. In dieser Domäne ist Vollständigkeit und Wartbarkeit von großem Interesse für die Industrie. Bei der Bestückung von Leiterplatten z.B. existieren häufig unterschiedliche bestückbare Varianten. Die Stücklisten müssen bisher von Hand angelegt und gepflegt werden. Da ein Großteil der Werkzeuge in diesem Bereich durch Skriptsprachen gesteuert werden kann, ist auch hier die Möglichkeit gegeben, Stücklisten und Bestückungspläne automatisiert als einzelne Produkte aus einer Produktlinie abzuleiten. Eine solche Stückliste muss vollständig und widerspruchsfrei sein. Sie muss für jede bestückbare Position eindeutige Vorgaben machen. Für die automatisierte Überprüfung ist der Algorithmus zur Bewertung von Vollständigkeit und Widerspruchsfreiheit ideal geeignet.

Denkbar ist der Einsatz der präsentierten Konzepte auch auf weiteren Domänen, die sich mit dem Test oder der Produktion variantenreicher Systeme auseinandersetzen. Dadurch kann sowohl der Grad der Automatisierbarkeit als auch die Qualität der Anforderungen und damit der entstehenden Produkte erhöht werden.

LITERATURVERZEICHNIS

[AFB+88] Alspaugh, T., S. Faulk, K. Britton, R. Parker, D. Parnas und J. Shore: *Software Requirements for the A-7E Aircraft*, March 1988.

[BH00] Bharadwaj, R. und C. Heitmeyer: *Developing high assurance avionics systems with the SCR requirements method*. 19th Digital Avionics System Conferences, 2000.

[Bor09] Bormann, J.: *Vollständige funktionale Verifikation*. Doktorarbeit, Universität Kaiserslautern, 2009.

[Bos00] Bosch, J.: *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. ACM Press/Addison-Wesley Publishing Co., 2000.

[BY01] Baresi, L. und M. Young: *Test Oracles*. Technical Report, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., 2001.

[CDS06] Cohen, M., M. Dwyer und J. Shi: *Coverage and Adequacy in Software Product Line Testing*. In: *Proceedings of the ISSTA 2006 workshop ROSATEA '06*, Seiten 53–63, New York, NY, USA, 2006. ACM.

[CDS07] Cohen, M., M. Dwyer und J. Shi: *Interaction Testing of Highly-Configurable Systems in the Presence of Constraints*. In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, Seiten 129–139, 2007.

[CE00] Czarnecki, K. und U. Eisenecker: *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., June 2000.

[CHE05] Czarnecki, K., S. Helsen und U. Eisenecker: *Staged Configuration Through Specialization and Multilevel Configuration of Feature Models*. Software Process: Improvement and Practice, Seiten 143–169, 2005.

[CN01] Clements, P. und L. Northrop: *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[CW07] Czarnecki, K. und A. Wasowski: *Feature Diagrams and Logics: There and Back Again*. In: *Proceedings of the 1th International Conference on Software Product Lines*, Seiten 23–34. IEEE Computer Society, 2007.

[Dep06] Department of Defense: *Test Method Standard: Microcircuits (MIL-STD-883G)*, 2006.

[Dep07a] DEPARTMENT OF DEFENSE: *Performance Specification - Integrated Circuits (Microcircuits) Manufacturing (MIL-PRF-38535H)*, 2007.

[Dep07b] DEPARTMENT OF DEFENSE: *Requirements for the Control of Electromagnetic Interference Characteristics of Subsystems and Equipment (MIL-STD-461F)*, 2007.

[Dep08] DEPARTMENT OF DEFENSE: *Test Method Standard - Environmental Engineering Considerations and Laboratory Tests (MIL-STD-810G)*, 2008.

[DIN95] DIN DEUTSCHES INSTITUT FÜR NORMUNG E. V.: *Norm DIN 55350-11: Begriffe zu Qualitätsmanagement und Statistik - Teil 11: Begriffe des Qualitätsmanagements*, 1995.

[DR96] DILLON, L. und Y. RAMAKRISHNA: *Generating Oracles from Your Favorite Temporal Logic Specifications*. In: *In Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Seiten 106–117. ACM Press, 1996.

[Fau01] FAULK, S.: *Product-Line Requirements Specification (PRS): An Approach and Case Study*. In: *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, Seiten 48–55, Washington, DC, USA, 2001. IEEE Computer Society.

[FBWJK92] FAULK, S., J. BRACKETT, P. WARD und JR. J. KIRBY: *The Core Method for Real-Time Requirements*. IEEE Software, 9:22–33, 1992.

[GGWA93] GROCHTMANN, M., K. GRIMM, J. WEGENER und DAIMLER-BENZ AG: *Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor*. In: *Proceedings of EuroSTAR '93*, Seiten 169–176, 1993.

[GR03] GOETZ, R. und C. RUPP: *Psychotherapy for System Requirements*. In: *Proceedings of the 2nd IEEE International Conference on Cognitive Informatics*, ICCI '03, Seiten 75–80, Washington, DC, USA, 2003. IEEE Computer Society.

[Hen80] HENINGER, K.: *Specifying Software Requirements for Complex Systems: New Techniques and Their Application*. IEEE Transactions on Software Engineering, 6:2–13, 1980.

[HJL96] HEITMEYER, C., R. JEFFORDS und B. LABAW: *Automated Consistency Checking of Requirements Specifications*. ACM Transactions on Software Engineering and Methodology, 5:231–261, 1996.

[Hoa04] HOARE, C.: *Communicating Sequential Processes*. Prentice Hall International, 2004.

[HST+08] HEYMANS, P., P. SCHOBBENS, J. TRIGAUX, Y. BONTEMPS, R. MATULEVICIUS und A. CLASSEN: *Evaluating Formal Properties of Feature Diagram Languages*. Software, IET, 2(3):281–302, 2008.

[IEE90] IEEE COMPUTER SOCIETY: *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610)*. The Institute of Electrical and Electronics Engineers, Inc., 1990.

[IEE98a] IEEE COMPUTER SOCIETY: *IEEE Standard for a Software Quality Metrics Methodology (IEEE Std 1061)*. The Institute of Electrical and Electronics Engineers, Inc., 1998.

[IEE98b] IEEE COMPUTER SOCIETY: *Recommended Practice for Software Requirements Specifications (IEEE Std 830)*. The Institute of Electrical and Electronics Engineers, Inc., 1998.

[IEE05] IEEE COMPUTER SOCIETY: *IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850)*. The Institute of Electrical and Electronics Engineers, Inc., 2005.

[IEE07] IEEE COMPUTER SOCIETY: *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800)*. The Institute of Electrical and Electronics Engineers, Inc., 2007.

[IEE08] IEEE COMPUTER SOCIETY: *IEEE Standard Systems and software engineering - Software life cycle processes (IEEE Std 12207)*. The Institute of Electrical and Electronics Engineers, Inc., 2008.

[IEE10] IEEE COMPUTER SOCIETY: *IEEE Standard Classification for Software Anomalies (IEEE Std 1044)*. The Institute of Electrical and Electronics Engineers, Inc., 2010.

[Int11] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *Road vehicles - Functional safety (ISO 26262)*, 2011.

[KAP08] KOOPMAN, P., P. ACHTEN und R. PLASMEIJER: *Testing and Validating the Quality of Specifications*. In: *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, Seiten 41–52, Washington, DC, USA, 2008. IEEE Computer Society.

[KCH+90] KANG, K., S. COHEN, J. HESS, W. NOVAK und A. PETERSON: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technischer Bericht, Carnegie-Mellon University Software Engineering Institute, 1990.

[LGB08] LAGUNA, M. und B. GONZÁLEZ-BAIXAULI: *Product Line Requirements: Multi-Paradigm Variability Models*. In: *Anais do WER '08*, Seiten 211–216, 2008.

[LP08] Lauenroth, K. und K. Pohl: *Dynamic Consistency Checking of Domain Requirements in Product Line Engineering*. In: *RE '08*, Seiten 193–202, Washington, DC, USA, 2008. IEEE Computer Society.

[LSR07] Linden, F. van der, K. Schmid und E. Rommes: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[Mar02] Marciniak, J.: *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[McG01] McGregor, J.: *Testing a Software Product Line*. Technischer Bericht CMU/SEI-2001-TR-022, Carnegie Mellon, Software Engineering Institute, 2001.

[MO10] Markert, F. und S. Oster: *Model-Based Generation of Test Oracles for Embedded Software Product Lines*. In: *Workshop Proceedings of the 14th International Software Product Line Conference*, MAPLE 2010 Workshop Proceedings, Seiten 147–154, 2010.

[MO11] Markert, F. und S. Oster: *A Formal Method to Identify Deficiencies of Functional Requirements for Product Lines of Embedded Systems*. In: *Workshop Proceedings der Software Engineering Konferenz*, GI Workshop. Gesellschaft für Informatik, 2011.

[MT01] Miller, S. und A. Tribble: *Extending the four-variable model to bridge the system-software gap*. In: *20th Digital Avionics Systems Conference*, 2001.

[Obe10] Oberkönig, M.: *Anwendung normalisierter Eigenschaften zur Verbesserung von Qualitätsaussagen in der funktionalen Hardware-Verifikation*. Doktorarbeit, Technische Universität Darmstadt, 2010.

[OMS09] Oster, S., F. Markert und A. Schürr: *Integrated Modeling of Software Product Lines with Feature Models and Classification Trees*. In: *Proceedings of the 13th International Software Product Line Conference (SPLC09)*, MAPLE 2009 Workshop Proceedings, Seiten 75–82. Springer Verlag, 2009.

[OWES11] Oster, S., A. Wübbeke, G. Engels und A. Schürr: *Model-Based Software Product Lines Testing Survey*. In: Zander, J., I. Schieferdecker und P. Mosterman (Herausgeber): *Model-based Testing for Embedded Systems*, Seiten 339–381. CRC Press/Taylor&Francis, 2011.

[PBL05] Pohl, K., G. Böckle und F. van der Linden: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., USA, 2005.

[PM95] PARNAS, D. und J. MADEY: *Functional Documents for Computer Systems*. Sci. Comput. Program., 25:41–61, 1995.

[Pnu77] PNUELI, A.: *The Temporal Logic of Programs*. Symposium on Foundations of Computer Science, Seiten 46–57, 1977.

[Rut93] RUTGERS THE STATE UNIVERSITY OF NEW JERSEY: *Satisfiability Suggested Format*, 1993.

[Sch09] SCHICKEL, M.: *Applications of Property-Based Design in Formal Verification*. Doktorarbeit, Technische Universität Darmstadt, 2009.

[Shi02] SHIMIZU, K.: *Writing, Verifying and Exploiting Formal Verifications of Hardware Designs*. Doktorarbeit, Stanford University, 2002.

[SOM10] SCHÜRR, A., S. OSTER und F. MARKERT: *Model-Driven Software Product Line Testing: An Integrated Approach*. In: *36th International Conference on Current Trends in Theory and Practice of Computer Science*, Lecture Notes in Computer Science (LNCS), Seiten 112–131, 2010.

[Spi89] SPIVEY, J.: *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[TTK04] TEVANLINNA, A., J. TAINA und R. KAUPPINEN: *Product family testing: a survey*. In: *ACM SIGSOFT Software Engineering Notes.*, Seiten 12–17, 2004.

[vdM07] MASSEN, T. VON DER: *Feature-basierte Modellierung und Analyse von Variabilität in Produktlinienanforderungen*. Doktorarbeit, RWTH Aachen, 2007.

[Vig10] VIGENSCHOW, U.: *Testen von Software und Embedded Systems*. dpunkt.verlag GmbH, Deutschland, 2010.

[Wal01] WALLMÜLLER, E.: *Software-Qualitätsmanagement in der Praxis*. Carl Hanser Verlag, 2001.

[ZM06] ZULTNER, R. und G. MAZUR: *The Kano Model: Recent Developments*. In: *The Eighteenth Symposium on Quality Function Development*, Seiten 108–116, 2006.

WISSENSCHAFTLICHER WERDEGANG

PERSÖNLICHE DATEN:

Name, Vorname: Markert, Florian
Geburtsdatum: 26. November 1979
Geburtsort: Frankfurt / Main
Staatsangehörigkeit: deutsch
Familienstand: verheiratet, 2 Kinder

WERDEGANG

seit 05/2011
Projektmanager, Gearomat GmbH, Dietzenbach

seit 01/2008
Wissenschaftlicher Mitarbeiter am Fachgebiet Rechnersysteme, Institut für Datentechnik, Technische Universität Darmstadt

08/2006 — 12/2007
Softwareingenieur für eingebettete Systeme,
Pan Dacom Direkt GmbH, Dreieich-Sprendlingen

STUDIUM

10/1999 — 06/2006
Studium der Elektrotechnik und Informationstechnik, Studienrichtung Datentechnik, Technische Universität Darmstadt

INDUSTRIEPRAKTIKA

05/2004 — 09/2004
Thales Air Traffic Management GmbH,
Kornthal-Münchingen

02/2001 — 04/2001
Incatronic Phönix Messtechnik GmbH, Frankfurt

ZERTIFIZIERUNGEN

ISTQB Certified Tester - Foundation Level

ITIL V3 Foundation Examination

EIGENE VERÖFFENTLICHUNGEN

2011

F. Markert, S. Oster: „A Formal Method to Identify Deficiencies of Functional Requirements for Product Lines of Embedded Systems“, Workshop Proceedings der Software Engineering Konferenz, 2011.

S. Oster, I. Zorcic, F. Markert, M. Lochau: „MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing“, 4th International Workshop on Variability Modelling of Software-Intensive Systems, 2011.

2010

F. Markert, S. Oster: „Model-Based Generation of Test Oracles for Embedded Software Product Lines“, Workshop Proceedings of the 14th International Software Product Line Conference, 2010.

S. Oster, F. Markert, P. Ritter: „Automated Incremental Pairwise Testing of Software Product Lines“, Proceedings of the 14th International Software Product Line Conference, 2010.

A. Schürr, S. Oster, F. Markert: „Model-Driven Software Product Line Testing: An Integrated Approach“, 36th International Conference on Current Trends in Theory and Practice of Computer Science, 2010.

2009

S. Oster, F. Markert, A. Schürr: „Integrated Modeling of Software Product Lines with Feature Models and Classification Trees“, Workshop Proceedings of the 13th International Software Product Line Conference, 2009.

Studien- und Bachelorarbeiten

Till Müller, „Entwurf, Simulation und Synthese des MAC-Layers eines PCI Ethernet Controllers"

Eugen Bayer, „Entwurf, Simulation und Synthese des "Advanced Encryption Standards" (AES) als Verschlüsselungsschicht eines PCI Ethernet Controllers"

David Meister, „Realisierung einer Linux-Umgebung für PowerPCs im FPGA"

Stefan Zügel, „Evaluation verschiedener HDL-Simulatoren mit Hinblick auf Effizienz und Vollständigkeitsmetriken"

Lukas Jung, „Implementierung der Ansteuerung eines Ultraschallsensor-Arrays in einem FPGA und Auswertung unter Linux"

David de la Chevallerie, „Implementierung der Ansteuerung eines GPS-Moduls in einem FPGA und Auswertung unter Linux"

Felix Wienker, „Evaluation und Synthese eines OpenSPARC-Prozessors und Integration auf einem Virtex-5 Entwicklungsboard"

Jens Dorn, „FPGA-Entwicklung eines virtuell analogen, polyphonen Synthesizers"

Björn Richerzhagen, „Implementierung und Synthese eines hardwarebeschleunigten Algorithmus zur Datenkompression"

Diplom- und Masterarbeiten

Alex Schönberger, „Entwicklung und Implementierung eines effizienten Protokolls zur optischen Multi-Gigabit Datenübertragung in einem FPGA"

Atul Athavale, „Automatisierte Generierung von Hard- und Software für Produktlinien eingebetteter Systeme"

Thorsten Kühnel, „Untersuchung von Methoden zur Bestimmung der Qualität von Verifikationsumgebungen in der SoC-Entwicklung"

ERKLÄRUNG

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 26. Januar 2012

Florian Markert